Oracle® Fusion Middleware

Fusion Developer's Guide for Oracle Application Development Framework

11*g* Release 2 (11.1.2.1.0)

E16182-02

September 2011

Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework 11*g* Release 2 (11.1.2.1.0)

E16182-02

Copyright © 2011, Oracle and/or its affiliates. All rights reserved.

Primary Authors: Ralph Gordon (lead), Walter Egan, Peter Jew, Kathryn Munn, Landon Ott, Robin Whitmore

Contributing Author: Odile Sullivan-Tarazi

Contributors: Lynn Munsinger, Steve Muench

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

Preface xlix

Audience xlix
Documentation Accessibility xlix
Related Documents xlix
Conventions l

What's New in This Guide in Release 11.1.2.1.0 li

Part I Getting Started with Fusion Web Applications

1 Introduction to Building Fusion Web Applications with Oracle ADF

1.1 Introduction to Oracle ADF 1-1
1.2 Oracle ADF Architecture 1-2
1.2.1 ADF Business Components 1-3
1.2.2 ADF Model Layer 1-4
1.2.3 ADF Controller 1-5
1.2.4 ADF Faces Rich Client 1-5
1.3 Overview of Building an Application with Oracle ADF 1-6
1.3.1 Creating an Application Workspace 1-7
1.3.2 Modeling with Database Object Definitions 1-10
1.3.3 Creating Use Cases 1-12
1.3.4 Designing Application Control and Navigation Using ADF Task Flows 1-13
1.3.5 Identifying Shared Resources 1-15
1.3.6 Creating a Data Model to Access Data with ADF Business Components 1-15
1.3.6.1 Creating a Layer of Business Domain Objects for Tables 1-16
1.3.6.2 Building the Business Services 1-17
1.3.6.3 Testing and Debugging Business Services with the Oracle ADF Model Tester 1-18
1.3.7 Implementing the User Interface with JSF 1-18
1.3.8 Data Binding with ADF Model Layer 1-20
1.3.9 Validation and Error Handling 1-23
1.3.10 Adding Security 1-24

1.3.11 Testing and Debugging the Web Client Application 1-24
1.3.12 Refactoring Application Artifacts..... 1-25
1.3.13 Deploying a Fusion Web Application 1-25
1.3.14 Integrating a Fusion Web Application 1-25
1.4 Working Productively in Teams..... 1-26
1.4.1 Enforcing Standards..... 1-27
1.4.2 Using a Source Control System 1-28
1.5 Other Resources for Learning Oracle ADF 1-29
1.6 Generation of Complete Web Tier Using Oracle JHeadstart..... 1-29

2 Introduction to the ADF Sample Application

2.1 About the Oracle Fusion Order Demo..... 2-1
2.2 Setting Up the Fusion Order Demo Application..... 2-1
2.2.1 How to Download the Application Resources..... 2-2
2.2.2 How to Install the Fusion Order Demo Schema 2-3
2.2.3 Overview of the Fusion Order Demo Schema..... 2-4
2.2.3.1 Translation Support in the Fusion Order Demo Schema 2-6
2.2.3.2 Lookup Tables in the Fusion Order Demo Schema..... 2-7
2.3 Running the Fusion Order Demo Application StoreFront Module 2-8
2.4 Running the Fusion Order Demo Standalone Applications..... 2-11
2.4.1 How to Run the Standalone Applications..... 2-11
2.4.2 Standalone Applications in the DevGuideExamples Application Workspace 2-13
2.4.3 Standalone Applications in the AdvancedExamples Application Workspace 2-16
2.4.4 Standalone Applications in the AdvancedEntityExamples Application Workspace..... 2-18
2.4.5 Standalone Applications in the AdvancedViewObjectExamples Application Workspace 2-20
2.4.6 Standalone Applications in the AdvancedUIExamples Application Workspaces . 2-22
2.5 Taking a Look at the Fusion Order Demo Application..... 2-23
2.5.1 Anonymous Browsing 2-25
2.5.1.1 Viewing Product Details..... 2-27
2.5.1.2 Browsing the Product Catalog..... 2-29
2.5.1.3 Searching for Products..... 2-30
2.5.2 The Login Process..... 2-33
2.5.3 The Ordering Process..... 2-34
2.5.4 The Customer Registration Process 2-39

Part II Building Your Business Services

3 Getting Started with ADF Business Components

3.1 About ADF Business Components..... 3-1
3.1.1 ADF Business Components Features..... 3-2
3.1.2 ADF Business Components Core Objects 3-3
3.2 Comparison to Familiar 4GL Tools 3-3
3.2.1 Familiar Concepts for Oracle Forms Developers..... 3-3
3.2.1.1 Similarities Between the Application Module and a "Headless" Form Module . 3-4
3.2.1.2 Similarities Between the Entity Object and a Forms Record Manager..... 3-5

3.2.1.3 Similarities Between the View Object and a Data Block ... 3-5
3.2.2 Familiar Concepts for PeopleTools Developers ... 3-5
3.2.2.1 Similarities Between the Application Module and a "Headless" Component ... 3-5
3.2.2.2 Similarities Between the Entity Object and a Record Definition ... 3-6
3.2.2.3 Similarities Between the View Object and a Row Set ... 3-6
3.2.3 Familiar Concepts for Siebel Tools Developers ... 3-6
3.2.3.1 Similarities Between the entity Object and a Table Object ... 3-6
3.2.3.2 Similarities Between the View Object and a Business Component ... 3-7
3.2.3.3 Similarities Between the Application Module and a Business Object ... 3-7
3.2.4 Familiar Functionality for ADO.NET Developers ... 3-7
3.2.4.1 Similarities Between the Application Module and a Data Set ... 3-7
3.2.4.2 Similarities Between the Entity Object and a Data Adapter ... 3-7
3.2.4.3 Similarities Between the View Object and a Data Table ... 3-8
3.3 Overview of Design Time Facilities ... 3-8
3.3.1 Choosing a Connection, SQL Platform, and Data Type Map ... 3-8
3.3.2 Creating New Components Using Wizards ... 3-9
3.3.3 Creating New Components Using the Context Menu ... 3-10
3.3.4 Editing Components Using the Component Overview Editor ... 3-10
3.3.5 Displaying Related Components Using Diagrams ... 3-10
3.3.6 Visualizing, Creating, and Editing Components Using UML Diagrams ... 3-11
3.3.7 Testing Application Modules Using the Oracle ADF Model Tester ... 3-11
3.3.8 Refactoring Components ... 3-12
3.4 Overview of the Oracle ADF Active Data Model ... 3-12
3.4.1 A More Generic Business Service Solution ... 3-12
3.4.2 Typical Scenarios for an Active Data Model ... 3-13
3.4.3 Active Data Model Support for Custom Code ... 3-13
3.5 Overview of the ADF Business Components Implementation ... 3-14
3.5.1 Standard Java and XML ... 3-14
3.5.2 Application Server or Database Independence ... 3-14
3.5.3 Java EE Design Pattern Support ... 3-15
3.5.4 Source Code Organization ... 3-15
3.5.5 Package Naming Conventions ... 3-16
3.5.6 Metadata with Optional Custom Java Code ... 3-17
3.5.6.1 Example of an XML-Only Component ... 3-18
3.5.6.2 Example of a Component with Custom Java Class ... 3-18
3.5.7 Basic Data Types ... 3-19
3.5.8 Generic Versus Strongly-Typed APIs ... 3-21
3.5.9 Custom Interface Support for Client-Accessible Components ... 3-21
3.5.9.1 Framework Client Interfaces for Components ... 3-22
3.5.9.2 Custom Client Interfaces for Components ... 3-22
3.6 Overview of Groovy Scripting Language Support ... 3-22
3.6.1 Referencing Business Components Objects in Groovy Expressions ... 3-25
3.6.2 Referencing Custom Business Components Methods and Attributes in Groovy Expressions 3-26
3.6.2.1 Referencing Members of the Same Business Component ... 3-27
3.6.2.2 Referencing Members of Other Business Components ... 3-28
3.6.3 Manipulating Business Component Attribute Values in Groovy Expressions ... 3-28

4 Creating a Business Domain Layer Using Entity Objects

4.1 About Entity Objects 4-1
4.1.1 Entity Object Use Cases and Examples 4-2
4.1.2 Additional Functionality for Entity Objects 4-2
4.2 Creating Entity Objects and Associations 4-2
4.2.1 How to Create Multiple Entity Objects and Associations from Existing Tables 4-3
4.2.2 How to Create Single Entity Objects Using the Create Entity Wizard 4-5
4.2.3 What Happens When You Create Entity Objects and Associations from Existing Tables 4-5
4.2.3.1 What Happens When Tables Have Foreign Key Relationships 4-6
4.2.3.2 What Happens When a Table Has No Primary Key 4-7
4.2.4 What Happens When You Create an Entity Object for a Synonym or View 4-7
4.2.5 How to Edit an Existing Entity Object or Association 4-7
4.2.6 How to Create Database Tables from Entity Objects 4-8
4.2.7 How to Synchronize an Entity with Changes to Its Database Table 4-8
4.2.7.1 Removing an Attribute Associated with a Dropped Column 4-9
4.2.7.2 Addressing a Data Type Change in the Underlying Table 4-9
4.2.8 How to Store Data Pertaining to a Specific Point in Time 4-10
4.2.9 What Happens When You Create Effective Dated Entity Objects 4-11
4.2.10 What You May Need to Know About Creating Entities from Tables 4-12
4.3 Creating and Configuring Associations 4-12
4.3.1 How to Create an Association 4-12
4.3.2 What Happens When You Create an Association 4-14
4.3.3 How to Change Entity Association Accessor Names 4-14
4.3.4 How to Rename and Move Associations to a Different Package 4-15
4.3.5 What You May Need to Know About Using a Custom View Object in an Association ... 4-16
4.3.6 What You May Need to Know About Composition Associations 4-17
4.4 Creating an Entity Diagram for Your Business Layer 4-17
4.4.1 How to Create an Entity Diagram 4-18
4.4.2 What Happens When You Create an Entity Diagram 4-19
4.4.3 What You May Need to Know About the XML Component Descriptors 4-20
4.4.4 What You May Need to Know About Changing the Names of Components 4-20
4.5 Defining Property Sets 4-20
4.5.1 How to Define a Property Set 4-21
4.5.2 How to Apply a Property Set 4-21
4.6 Defining Attribute Control Hints for Entity Objects 4-22
4.6.1 How to Add Attribute Control Hints 4-22
4.6.2 What Happens When You Add Attribute Control Hints 4-23
4.6.3 How to Define Formatters and Masks 4-23
4.7 Working with Resource Bundles 4-25
4.7.1 How to Set Message Bundle Options 4-26
4.7.2 How to Use Multiple Resource Bundles 4-27
4.7.3 How to Internationalize the Date Format 4-27
4.8 Defining Business Logic Groups 4-28
4.8.1 How to Create a Business Logic Group 4-29
4.8.2 How to Create a Business Logic Unit 4-30

4.8.3 How to Add Logic to a Business Logic Unit ... 4-30
4.8.4 How to Override Attributes in a Business Logic Unit ... 4-31
4.8.5 What Happens When You Create a Business Logic Group ... 4-31
4.8.6 What Happens at Runtime: Invoking a Business Logic Group ... 4-32
4.9 Configuring Runtime Behavior Declaratively ... 4-33
4.9.1 How to Configure Declarative Runtime Behavior ... 4-33
4.9.2 What Happens When You Configure Declarative Runtime Behavior ... 4-34
4.9.3 How to Use Update Batching ... 4-34
4.10 Setting Attribute Properties ... 4-35
4.10.1 How to Set Database and Java Data Types for an Entity Object Attribute ... 4-35
4.10.2 How to Indicate Data Type Length, Precision, and Scale ... 4-37
4.10.3 How to Control the Updatability of an Attribute ... 4-37
4.10.4 How to Make an Attribute Mandatory ... 4-37
4.10.5 How to Define the Primary Key for the Entity ... 4-37
4.10.6 How to Define a Static Default Value ... 4-38
4.10.7 How to Define a Default Value Using a Groovy Expression ... 4-38
4.10.8 What Happens When You Create a Default Value Using a Groovy expression ... 4-39
4.10.9 How to Synchronize with Trigger-Assigned Values ... 4-39
4.10.10 How to Get Trigger-Assigned Primary Key Values from a Database Sequence ... 4-39
4.10.11 How to Protect Against Losing Simultaneously Updated Data ... 4-40
4.10.12 How to Track Created and Modified Dates Using the History Column ... 4-41
4.10.13 How to Configure Composition Behavior ... 4-41
4.10.13.1 Orphan-Row Protection for New Composed Entities ... 4-42
4.10.13.2 Ordering of Changes Saved to the Database ... 4-42
4.10.13.3 Cascade Update of Composed Details from Refresh-On-Insert Primary Keys 4-42
4.10.13.4 Cascade Delete Support ... 4-42
4.10.13.5 Cascade Update of Foreign Key Attributes When Primary Key Changes ... 4-43
4.10.13.6 Locking of Composite Parent Entities ... 4-43
4.10.13.7 Updating of Composing Parent History Attributes ... 4-43
4.10.14 How to Set the Discriminator Attribute for Entity Object Inheritance Hierarchies 4-43
4.10.15 How to Define Alternate Key Values ... 4-44
4.10.16 What Happens When You Define Alternate Key Values ... 4-44
4.10.17 What You May Need to Know About Alternate Key Values ... 4-44
4.11 Adding Transient and Calculated Attributes to an Entity Object ... 4-44
4.11.1 How to Add a Transient Attribute ... 4-45
4.11.2 What Happens When You Add a Transient Attribute ... 4-45
4.11.3 How to Base a Transient Attribute on a Groovy Expression ... 4-46
4.11.4 What Happens When You Base a Transient Attribute on a Groovy Expression ... 4-48
4.11.5 How to Add Java Code in the Entity Class to Perform Calculation ... 4-48
4.12 Creating Business Events ... 4-48
4.12.1 Introducing Event Definitions ... 4-49
4.12.2 Introducing Event Points ... 4-49
4.12.3 What You May Need to Know About Event Points ... 4-50
4.12.4 How to Create a Business Event ... 4-50
4.12.5 What Happens When You Create a Business Event ... 4-51
4.12.6 What You May Need to Know About Payload ... 4-52
4.12.7 How to Publish a Business Event ... 4-52

4.12.8 How to Subscribe to Business Events ... 4-53
4.13 Generating Custom Java Classes for an Entity Object ... 4-54
4.13.1 How to Generate Custom Classes ... 4-54
4.13.2 What Happens When You Generate Custom Classes ... 4-55
4.13.3 What Happens When You Generate Entity Attribute Accessors ... 4-55
4.13.4 How to Navigate to Custom Java Files ... 4-56
4.13.5 What You May Need to Know About Custom Java Classes ... 4-57
4.13.5.1 Framework Base Classes for an Entity Object ... 4-57
4.13.5.2 Safely Adding Code to the Custom Component File ... 4-57
4.13.5.3 Configuring Default Java Generation Preferences ... 4-58
4.13.5.4 Attribute Indexes and InvokeAccessor Generated Code ... 4-58
4.13.6 Programmatic Example for Comparison Using Custom Entity Classes ... 4-59
4.14 Working Programmatically with Entity Objects and Associations ... 4-62
4.14.1 How to Find an Entity Object by Primary Key ... 4-62
4.14.2 How to Access an Associated Entity Using the Accessor Attribute ... 4-63
4.14.3 How to Update or Remove an Existing Entity Row ... 4-65
4.14.4 How to Create a New Entity Row ... 4-66
4.14.5 Assigning the Primary Key Value Using an Oracle Sequence ... 4-67
4.14.6 How to Update a Deleted Flag Instead of Deleting Rows ... 4-68
4.14.6.1 Updating a Deleted Flag When a Row Is Removed ... 4-69
4.14.6.2 Forcing an Update DML Operation Instead of a Delete ... 4-69
4.14.7 How to Control Entity Posting Order to Prevent Constraint Violations ... 4-70
4.14.7.1 Default Post Processing Order ... 4-70
4.14.7.2 Compositions and Default Post Processing Order ... 4-70
4.14.7.3 Overriding postChanges() to Control Post Order ... 4-70
4.14.8 Advanced Entity Association Techniques ... 4-75
4.14.8.1 Modifying Association SQL Clause to Implement Complex Associations ... 4-75
4.14.8.2 Exposing View Link Accessor Attributes at the Entity Level ... 4-75
4.14.8.3 Optimizing Entity Accessor Access by Retaining the Row Set ... 4-76
4.15 Creating Custom, Validated Data Types Using Domains ... 4-77
4.15.1 How to Create a Domain ... 4-77
4.15.2 What Happens When You Create a Domain ... 4-78
4.15.3 What You May Need to Know About Domains ... 4-78
4.15.3.1 Domains as Entity and View Object Attributes ... 4-78
4.15.3.2 DataCreationException in Custom validate() Method ... 4-79
4.15.3.3 String Domains and String Value Aggregation ... 4-79
4.15.3.4 Simple Domains and Built-In Types ... 4-80
4.15.3.5 Simple Domains As Immutable Java Classes ... 4-80
4.15.3.6 Creating Domains for Oracle Object Types When Useful ... 4-80
4.15.3.7 Quickly Navigating to the Domain Class ... 4-82
4.15.3.8 Domains Packaged in the Common JAR ... 4-82
4.15.3.9 Custom Domain Properties and Attributes in Entity and View Objects ... 4-82
4.15.3.10 Inherited Restrictive Properties of Domains in Entity and View Objects ... 4-82
4.16 Creating New History Types ... 4-82
4.16.1 How to Create New History Types ... 4-82
4.16.2 How to Remove a History Type ... 4-84
4.17 Basing an Entity Object on a PL/SQL Package API ... 4-85

4.17.1 How to Create an Entity Object Based on a View ... 4-86
4.17.2 What Happens When You Create an Entity Object Based on a View ... 4-87
4.17.3 How to Centralize Details for PL/SQL-Based Entities into a Base Class ... 4-87
4.17.4 How to Implement the Stored Procedure Calls for DML Operations ... 4-88
4.17.5 How to Add Select and Lock Handling ... 4-89
4.17.5.1 Updating PLSQLEntityImpl Base Class to Handle Lock and Select ... 4-90
4.17.5.2 Implementing Lock and Select for the Product Entity ... 4-91
4.17.5.3 Refreshing the Entity Object After RowInconsistentException ... 4-93
4.18 Basing an Entity Object on a Join View or Remote DBLink ... 4-94
4.18.1 How to Disable the Use of the RETURNING Clause ... 4-94
4.18.2 What Happens at Runtime: Disabling the RETURNING Clause ... 4-94
4.19 Using Inheritance in Your Business Domain Layer ... 4-94
4.19.1 Understanding When Inheritance Can Be Useful ... 4-95
4.19.2 How to Create Entity Objects in an Inheritance Hierarchy ... 4-96
4.19.2.1 Identifying the Discriminator Column and Distinct Values ... 4-97
4.19.2.2 Identifying the Subset of Attributes Relevant to Each Kind of Entity ... 4-97
4.19.2.3 Creating the Base Entity Object in an Inheritance Hierarchy ... 4-98
4.19.2.4 Creating a Subtype Entity Object in an Inheritance Hierarchy ... 4-99
4.19.3 How to Add Methods to Entity Objects in an Inheritance Hierarchy ... 4-100
4.19.3.1 Adding Methods Common to All Entity Objects in the Hierarchy ... 4-100
4.19.3.2 Overriding Common Methods in a Subtype Entity Object ... 4-101
4.19.3.3 Adding Methods Specific to a Subtype Entity Object ... 4-102
4.19.4 What You May Need to Know About Using Inheritance ... 4-103
4.19.4.1 When to Introduce a New Base Entity ... 4-103
4.19.4.2 Subtype Entity Objects and the findByPrimaryKey() Method ... 4-103
4.19.4.3 View Objects with Polymorphic Entity Usages ... 4-103

5 Defining SQL Queries Using View Objects

5.1 About View Objects ... 5-1
5.1.1 View Object Use Cases and Examples ... 5-2
5.1.2 Additional Functionality for View Objects ... 5-4
5.2 Populating View Object Rows from a Single Database Table ... 5-4
5.2.1 How to Create an Entity-Based View Object ... 5-5
5.2.1.1 Creating an Entity-Based View Object from a Single Table ... 5-6
5.2.1.2 Creating a View Object with All the Attributes of an Entity Object ... 5-8
5.2.2 What Happens When You Create an Entity-Based View Object ... 5-9
5.2.3 How to Create an Expert Mode, Read-Only View Object ... 5-10
5.2.4 What Happens When You Create a Read-Only View Object ... 5-13
5.2.5 How to Edit a View Object ... 5-14
5.2.5.1 Overriding the Inherit Properties from Underlying Entity Object Attributes ... 5-14
5.2.5.2 Controlling the Length, Precision, and Scale of View Object Attributes ... 5-15
5.2.5.3 Converting a Read-Only View Object to Allow Attribute Updates ... 5-16
5.2.5.4 Customizing View Object Attribute Display in the Overview Editor ... 5-17
5.2.5.5 Modifying the Order of Attributes in the View Object Source File ... 5-19
5.2.6 How to Show View Objects in a Business Components Diagram ... 5-19
5.3 Populating View Object Rows with Static Data ... 5-20
5.3.1 How to Create Static View Objects with Data You Enter ... 5-20

5.3.2 How to Create Static View Objects with Data You Import ... 5-21
5.3.3 What Happens When You Create a Static List View Object ... 5-23
5.3.4 How to Edit Static List View Objects ... 5-24
5.3.5 What You May Need to Know About Static List View Objects ... 5-24
5.4 Limiting View Object Rows Using Effective Date Ranges ... 5-24
5.4.1 How to Create an Date-Effective View Object ... 5-25
5.4.2 How to Create New View Rows Using Date-Effective View Objects ... 5-26
5.4.3 How to Update Date-Effective View Rows ... 5-26
5.4.4 How to Delete Date-Effective View Rows ... 5-27
5.4.5 What Happens When You Create a Date-Effective View Object ... 5-27
5.4.6 What You May Need to Know About Date-Effective View Objects and View LInks ... 5-28
5.5 Working with Multiple Tables in Join Query Results ... 5-29
5.5.1 How to Create Joins for Entity-Based View Objects ... 5-29
5.5.2 How to Select Additional Attributes from Reference Entity Usages ... 5-33
5.5.3 How to Remove Unnecessary Key Attributes from Reference Entity Usages ... 5-34
5.5.4 How to Hide the Primary Key Attributes from Reference Entity Usages ... 5-34
5.5.5 How to Modify a Default Join Clause to Be an Outer Join When Appropriate ... 5-35
5.5.6 What Happens When You Reference Entities in a View Object ... 5-37
5.5.7 How to Create Joins for Read-Only View Objects ... 5-38
5.5.8 How to Test the Join View ... 5-38
5.5.9 How to Use the Query Builder with Read-Only View Objects ... 5-39
5.5.10 What You May Need to Know About Join View Objects ... 5-40
5.6 Working with Multiple Tables in a Master-Detail Hierarchy ... 5-40
5.6.1 How to Create a Master-Detail Hierarchy for Read-Only View Objects ... 5-40
5.6.2 How to Create a Master-Detail Hierarchy for Entity-Based View Objects ... 5-43
5.6.3 What Happens When You Create Master-Detail Hierarchies Using View Links ... 5-44
5.6.4 How to Enable Active Master-Detail Coordination in the Data Model ... 5-45
5.6.5 How to Test Master-Detail Coordination ... 5-47
5.6.6 How to Access the Detail Collection Using the View Link Accessor ... 5-47
5.6.6.1 Accessing Attributes of Row by Name ... 5-47
5.6.6.2 Programmatically Accessing a Detail Collection Using the View Link Accessor ... 5-48
5.6.7 How to Create a Master-Detail Hierarchy for Entity Objects with Transient-Only Attributes 5-48
5.7 Working with a Single Table in a Recursive Master-Detail Hierarchy ... 5-49
5.7.1 How to Create a Recursive Master-Detail Hierarchy for an Entity-Based View Object ... 5-49
5.7.1.1 Creating an Association-Based, Self-Referential View Link ... 5-50
5.7.1.2 Exposing the View Instance and Filter with a View Criteria ... 5-52
5.7.2 What Happens When You Create a Recursive Master-Detail Hierarchy ... 5-54
5.8 Working with View Objects in Declarative SQL Mode ... 5-55
5.8.1 How to Create SQL-Independent View Objects with Declarative SQL Mode ... 5-56
5.8.1.1 Enabling Declarative SQL Mode for All New View Objects ... 5-57
5.8.1.2 Enabling Declarative SQL Mode for Specific View Objects ... 5-57
5.8.2 How to Filter Declarative SQL-Based View Objects When Table Joins Apply ... 5-59
5.8.3 How to Filter Master-Detail Related View Objects with Declarative SQL Mode ... 5-61

5.8.4 How to Support Programmatic Execution of Declarative SQL Mode View Objects..... 5-61
5.8.4.1 Forcing Attribute Queries for All Declarative SQL Mode View Objects 5-62
5.8.4.2 Forcing Attribute Queries for Specific Declarative SQL Mode View Objects .. 5-62
5.8.5 What Happens When You Create a View Object in Declarative SQL Mode..... 5-63
5.8.6 What Happens at Runtime: Declarative SQL Mode Queries..... 5-64
5.8.7 What You May Need to Know About Overriding Declarative SQL Mode Defaults 5-64
5.8.8 What You May Need to Know About Working Programmatically with Declarative SQL Mode View Objects 5-65
5.9 Working with View Objects in Expert Mode..... 5-66
5.9.1 How to Customize SQL Statements in Expert Mode 5-66
5.9.2 How to Name Attributes in Expert Mode..... 5-66
5.9.3 What Happens When You Enable Expert Mode..... 5-67
5.9.4 What You May Need to Know About Expert Mode 5-67
5.9.4.1 Expert Mode Provides Limited Attribute Mapping Assistance 5-68
5.9.4.2 Expert Mode Drops Custom Edits..... 5-68
5.9.4.3 Expert Mode Ignores Changes to SQL Expressions 5-69
5.9.4.4 Expert Mode Returns Error for SQL Calculations that Change Entity Attributes..... 5-69
5.9.4.5 Expert Mode Retains Formatting of SQL Statement 5-70
5.9.4.6 Expert Mode Wraps Queries as Inline Views..... 5-71
5.9.4.7 Limitation of Inline View Wrapping at Runtime..... 5-71
5.9.4.8 Expert Mode Changes May Affect Dependent Objects 5-72
5.10 Working with Bind Variables..... 5-72
5.10.1 How to Add Bind Variables to a View Object Definition..... 5-73
5.10.2 How to Reference the Current User in a Named Bind Variable Using Groovy..... 5-74
5.10.3 What Happens When You Add Named Bind Variables..... 5-75
5.10.4 How to Test Named Bind Variables 5-75
5.10.5 How to Add a WHERE Clause with Named Bind Variables at Runtime 5-76
5.10.6 How to Set Existing Bind Variable Values at Runtime 5-78
5.10.7 What Happens at Runtime: Dynamic Read-Only View Object WHERE Clause 5-79
5.10.8 What You May Need to Know About Named Bind Variables 5-81
5.10.8.1 An Error Related to Clearing Bind Variables 5-81
5.10.8.2 A Helper Method to Remove Named Bind Variables 5-81
5.10.8.3 Errors Related to Naming Bind Variables..... 5-82
5.10.8.4 Default Value of NULL for Bind Variables..... 5-83
5.11 Working with Named View Criteria..... 5-83
5.11.1 How to Create Named View Criteria Declaratively..... 5-84
5.11.2 What Happens When You Create a Named View Criteria 5-89
5.11.3 What You May Need to Know About Bind Variable Options..... 5-90
5.11.4 What You May Need to Know About Nested Expressions..... 5-91
5.11.5 How to Set User Interface Hints on View Criteria to Support Search Forms..... 5-91
5.11.6 How to Test View Criteria Using the Oracle ADF Model Tester 5-95
5.11.7 How to Create View Criteria Programmatically..... 5-96
5.11.8 What Happens at Runtime: How the View Criteria Is Applied to a View Object .. 5-97
5.11.9 What You May Need to Know About the View Criteria API..... 5-98
5.11.9.1 Referencing Attribute Names in View Criteria..... 5-98

5.11.9.2 Referencing Bind Variables in View Criteria ... 5-98
5.11.9.3 Altering Compound Search Conditions Using Multiple View Criteria ... 5-99
5.11.9.4 Searching for a Row Whose Attribute Value Is NULL Value ... 5-100
5.11.9.5 Searching for Rows Whose Attribute Value Matches a Value in a List ... 5-100
5.11.9.6 Searching Case-Insensitively ... 5-100
5.11.9.7 Clearing View Criteria in Effect ... 5-100
5.11.10 What You May Need to Know About Query-by-Example Criteria ... 5-101
5.12 Working with List of Values (LOV) in View Object Attributes ... 5-101
5.12.1 How to Define a Single LOV-Enabled View Object Attribute ... 5-103
5.12.2 How to Define Cascading Lists for LOV-Enabled View Object Attributes ... 5-104
5.12.2.1 Creating a Data Source View Object to Control the Cascading List ... 5-105
5.12.2.2 Creating a View Accessor to Filter the Cascading List ... 5-106
5.12.3 How to Specify Multiple LOVs for an LOV-Enabled View Object Attribute ... 5-108
5.12.4 How to Define an LOV to Display a Reference Attribute ... 5-110
5.12.5 How to Set User Interface Hints on a View Object LOV-Enabled Attribute ... 5-112
5.12.6 How to Handle Date Conversion for List Type UI Components ... 5-116
5.12.7 How to Automatically Refresh the View Object of the View Accessor ... 5-116
5.12.8 How to Test LOV-Enabled Attributes Using the Oracle ADF Model Tester ... 5-118
5.12.9 What Happens When You Define an LOV for a View Object Attribute ... 5-119
5.12.10 What Happens at Runtime: How an LOV Queries the List Data Source ... 5-120
5.12.11 What You May Need to Know About Lists ... 5-121
5.12.11.1 Inheritance of AttributeDef Properties from Parent View Object Attributes ... 5-121
5.12.11.2 Using Validators to Validate Attribute Values ... 5-122
5.13 Defining UI Hints for View Objects ... 5-122
5.13.1 How to Add Attribute-Specific UI Hints ... 5-122
5.13.2 How to Add View Object UI Hints ... 5-123
5.13.3 How to Access UI Hints Using EL Expressions ... 5-123
5.13.4 What Happens When You Add UI Hints ... 5-124
5.13.5 How to Define UI Category Hints ... 5-125
5.13.6 What Happens When You Assign Attributes to UI Categories ... 5-128
5.13.7 What You May Need to Know About Resource Bundles ... 5-130
5.14 Adding Calculated and Transient Attributes to a View Object ... 5-130
5.14.1 How to Add a SQL-Calculated Attribute ... 5-130
5.14.2 What Happens When You Add a SQL-Calculated Attribute ... 5-131
5.14.3 How to Add a Transient Attribute ... 5-132
5.14.4 How to Add a Validation Rule to a Transient Attribute ... 5-134
5.14.5 What Happens When You Add a Transient Attribute ... 5-135
5.14.6 Adding Java Code in the View Row Class to Perform Calculation ... 5-136
5.14.7 What You May Need to Know About Transient Attributes ... 5-136

6 Testing View Instance Queries

6.1 About View Instance Queries ... 6-1
6.1.1 View Instance Use Cases and Examples ... 6-1
6.1.2 Additional Functionality for View Instances ... 6-1
6.2 Creating an Application Module to Test View Instances ... 6-2
6.2.1 How to Create the Application Module with Individual View Object Instances ... 6-2
6.2.2 How to Create the Application Module with Master-Detail View Object Instances ... 6-3

6.3 Testing View Object Instances Using the Oracle ADF Model Tester 6-5
6.3.1 How to Run the Oracle ADF Model Tester 6-6
6.3.2 How to Test Entity-Based View Objects Interactively 6-8
6.3.3 How to Update the Oracle ADF Model Tester to Display Project Changes 6-10
6.3.4 What Happens When You Use the Oracle ADF Model Tester 6-10
6.3.5 How to Simulate End-User Interaction in the Oracle ADF Model Tester 6-11
6.3.5.1 Testing Master-Detail Coordination 6-13
6.3.5.2 Testing UI Hints 6-13
6.3.5.3 Testing Business Domain Layer Validation 6-13
6.3.5.4 Testing Alternate Language Message Bundles and UI Hints 6-14
6.3.5.5 Testing View Objects That Reference Entity Usages 6-14
6.3.5.6 Testing Row Creation and Default Value Generation 6-14
6.3.5.7 Testing That New Detail Rows Have Correct Foreign Keys 6-15
6.3.6 How to Test Multiuser Scenarios in the Oracle ADF Model Tester 6-15
6.3.7 How to Customize Configuration Options Before Running the Tester 6-15
6.3.8 How to Enable ADF Business Components Debug Diagnostics 6-16
6.3.9 What Happens at Runtime: How View Objects and Entity Objects Cooperate 6-17
6.3.9.1 What Happens at Runtime: After a View Object Executes Its Query 6-17
6.3.9.2 What Happens at Runtime: After a View Row Attribute Is Modified 6-19
6.3.9.3 What Happens at Runtime: After a Foreign Key Attribute is Changed 6-20
6.3.9.4 What Happens at Runtime: After a Transaction is Committed 6-21
6.3.9.5 What Happens at Runtime: After a View Object Requeries Data 6-22
6.3.10 What You May Need to Know About Optimizing View Object Runtime Performance .. 6-23
6.4 Testing View Object Instances Programmatically 6-26
6.4.1 ViewObject Interface Methods for Working with the View Object's Default RowSet 6-26
6.4.1.1 The Role of the Key Object in a View Row or Entity Row 6-27
6.4.1.2 The Role of the Entity Cache in the Transaction 6-28
6.4.2 How to Create a Command-Line Java Test Client 6-29
6.4.2.1 Generating a Test Client with Skeleton Code 6-29
6.4.2.2 Modifying the Skeleton Code to Create the Test Client 6-30
6.4.3 What Happens When You Run a Test Client Program 6-32
6.4.4 What You May Need to Know About Running a Test Client 6-32
6.4.5 How to Count the Number of Rows in a Row Set 6-32
6.4.6 How to Access a Detail Collection Using the View Link Accessor 6-33
6.4.7 How to Iterate Over a Master-Detail-Detail Hierarchy 6-35
6.4.8 How to Find a Row and Update a Foreign Key Value 6-37
6.4.9 How to Create a New Row for a View Object Instance 6-38
6.4.10 How to Retrieve the Row Key Identifying a Row 6-39
6.4.11 How to Authenticate Test Users in the Test Client 6-41

7 Defining Validation and Business Rules Declaratively

7.1 About Declarative Validation 7-1
7.1.1 Declarative Validation Use Cases and Examples 7-2
7.1.2 Additional Functionality for Declarative Validation 7-2
7.2 Understanding the Validation Cycle 7-2

7.2.1 Types of Entity Object Validation Rules ... 7-3
7.2.1.1 Attribute-Level Validation Rules ... 7-3
7.2.1.2 Entity-Level Validation Rules ... 7-3
7.2.2 Understanding Commit Processing and Validation ... 7-3
7.2.3 Understanding the Impact of Composition on Validation Order ... 7-4
7.2.4 Avoiding Infinite Validation Cycles ... 7-4
7.2.5 What Happens When Validations Fail ... 7-4
7.2.6 Understanding Entity Objects Row States ... 7-5
7.2.7 Understanding Bundled Exception Mode ... 7-6
7.3 Adding Validation Rules to Entity Objects and Attributes ... 7-6
7.3.1 How to Add a Validation Rule to an Entity or Attribute ... 7-6
7.3.2 How to View and Edit a Validation Rule on an Entity Object or Attribute ... 7-7
7.3.3 What Happens When You Add a Validation Rule ... 7-7
7.3.4 What You May Need to Know About Entity and Attribute Validation Rules ... 7-8
7.4 Using the Built-in Declarative Validation Rules ... 7-9
7.4.1 How to Ensure That Key Values Are Unique ... 7-9
7.4.2 What Happens When You Use a Unique Key Validator ... 7-10
7.4.3 How to Validate Based on a Comparison ... 7-10
7.4.4 What Happens When You Validate Based on a Comparison ... 7-12
7.4.5 How to Validate Using a List of Values ... 7-13
7.4.6 What Happens When You Validate Using a List of Values ... 7-14
7.4.7 What You May Need to Know About the List Validator ... 7-15
7.4.8 How to Make Sure a Value Falls Within a Certain Range ... 7-15
7.4.9 What Happens When You Use a Range Validator ... 7-16
7.4.10 How to Validate Against a Number of Bytes or Characters ... 7-16
7.4.11 What Happens When You Validate Against a Number of Bytes or Characters ... 7-17
7.4.12 How to Validate Using a Regular Expression ... 7-17
7.4.13 What Happens When You Validate Using a Regular Expression ... 7-19
7.4.14 How to Use the Average, Count, or Sum to Validate a Collection ... 7-19
7.4.15 What Happens When You Use Collection Validation ... 7-20
7.4.16 How to Determine Whether a Key Exists ... 7-20
7.4.17 What Happens When You Use a Key Exists Validator ... 7-22
7.4.18 What You May Need to Know About Declarative Validators and View Accessors ... 7-22
7.5 Using Groovy Expressions For Validation and Business Rules ... 7-23
7.5.1 How to Reference Entity Object Methods in Groovy Validation Expressions ... 7-23
7.5.2 How to Validate Using a True/False Expression ... 7-25
7.5.3 What Happens When You Add a Groovy Expression ... 7-26
7.6 Triggering Validation Execution ... 7-27
7.6.1 How to Specify Which Attributes Fire Validation ... 7-28
7.6.2 What Happens When You Constrain Validation Execution with Triggering Attributes ... 7-29
7.6.3 How to Set Preconditions for Validation ... 7-30
7.6.4 How to Set Transaction-Level Validation ... 7-30
7.6.5 What You May Need to Know About the Order of Validation Execution ... 7-30
7.7 Creating Validation Error Messages ... 7-31
7.7.1 How to Create Validation Error Messages ... 7-31
7.7.2 How to Localize Validation Messages ... 7-31

7.7.3 How to Conditionally Raise Error Messages Using Groovy ... 7-32
7.7.4 How to Embed a Groovy Expression in an Error Message ... 7-32
7.8 Setting the Severity Level for Validation Exceptions ... 7-34
7.9 Bulk Validation in SQL ... 7-34

8 Implementing Validation and Business Rules Programmatically

8.1 About Programmatic Business Rules ... 8-1
8.1.1 Programmatic Business Rules Use Cases and Examples ... 8-2
8.1.2 Additional Functionality for Programmatic Business Rules ... 8-3
8.2 Using Method Validators ... 8-3
8.2.1 How to Create an Attribute-Level Method Validator ... 8-4
8.2.2 What Happens When You Create an Attribute-Level Method Validator ... 8-5
8.2.3 How to Create an Entity-Level Method Validator ... 8-6
8.2.4 What Happens When You Create an Entity-Level Method Validator ... 8-7
8.2.5 What You May Need to Know About Translating Validation Rule Error Messages 8-7
8.3 Assigning Programmatically Derived Attribute Values ... 8-8
8.3.1 How to Provide Default Values for New Rows at Create Time ... 8-8
8.3.1.1 Choosing Between create() and initDefaultExpressionAttributes() Methods ... 8-8
8.3.1.2 Eagerly Defaulting an Attribute Value from a Database Sequence ... 8-8
8.3.2 How to Assign Derived Values Before Saving ... 8-9
8.3.3 How to Assign Derived Values When an Attribute Value Is Set ... 8-10
8.4 Undoing Pending Changes to an Entity Using the Refresh Method ... 8-10
8.4.1 How to Control What Happens to New Rows During a Refresh ... 8-11
8.4.2 How to Cascade Refresh to Composed Children Entity Rows ... 8-11
8.5 Using View Objects for Validation ... 8-11
8.5.1 How to Use View Accessors for Validation Against View Objects ... 8-11
8.5.2 How to Validate Conditions Related to All Entities of a Given Type ... 8-12
8.5.3 What You May Need to Know About Row Set Access with View Accessors ... 8-14
8.6 Accessing Related Entity Rows Using Association Accessors ... 8-14
8.6.1 How to Access Related Entity Rows ... 8-14
8.6.2 How to Access Related Entity Row Sets ... 8-15
8.7 Referencing Information About the Authenticated User ... 8-16
8.8 Accessing Original Attribute Values ... 8-16
8.9 Storing Information About the Current User Session ... 8-16
8.9.1 How to Store Information About the Current User Session ... 8-16
8.9.2 How to Use Groovy to Access Information About the Current User Session ... 8-17
8.10 Accessing the Current Date and Time ... 8-18
8.11 Sending Notifications Upon a Successful Commit ... 8-18
8.12 Conditionally Preventing an Entity Row from Being Removed ... 8-18
8.13 Determining Conditional Updatability for Attributes ... 8-19
8.14 Implementing Custom Validation Rules ... 8-19
8.14.1 How to Create a Custom Validation Rule ... 8-20
8.14.2 Adding a Design Time Bean Customizer for Your Rule ... 8-22
8.14.3 How to Register and Using a Custom Rule in JDeveloper ... 8-22
8.14.3.1 Registering a Custom Validator at the Project Level ... 8-23
8.14.3.2 Registering a Custom Validator at the IDE Level ... 8-23

9 Implementing Business Services with Application Modules

9.1 About Application Modules ... 9-1
9.1.1 Application Module Use Cases and Examples ... 9-2
9.1.2 Additional Functionality for Application Modules ... 9-3
9.2 Creating and Modifying an Application Module ... 9-3
9.2.1 How to Create an Application Module ... 9-4
9.2.2 What Happens When You Create an Application Module ... 9-5
9.2.3 How to Add a View Object to an Application Module ... 9-5
9.2.3.1 Adding a View Object Instance to an Existing Application Module ... 9-5
9.2.3.2 Adding Master-Detail View Object Instances to an Application Module ... 9-6
9.2.3.3 Customizing a View Object Instance that You Add to an Application Module . 9-9
9.2.4 What Happens When You Add a View Object to an Application Module ... 9-11
9.2.5 How to Edit an Existing Application Module ... 9-11
9.2.6 How to Change the Data Control Name Before You Begin Building Pages ... 9-12
9.2.7 What You May Need to Know About Application Module Granularity ... 9-12
9.2.8 What You May Need to Know About View Object Components and View Object Instances 9-13
9.3 Configuring Your Application Module Database Connection ... 9-13
9.3.1 How to Use a JDBC Data Source Connection Type ... 9-14
9.3.2 How to Use a JDBC URL Connection Type ... 9-15
9.3.3 What Happens When You Create an Application Module Database Connection . 9-15
9.3.4 How to Change Your Application Module's Runtime Configuration ... 9-17
9.3.5 How to Change the Database Connection for Your Project ... 9-18
9.4 Defining Nested Application Modules ... 9-18
9.4.1 How to Define a Nested Application Module ... 9-19
9.4.2 What You May Need to Know About Root Application Modules Versus Nested Application Module Usages 9-20
9.5 Creating an Application Module Diagram for Your Business Service ... 9-21
9.5.1 How to Create an Application Module Diagram ... 9-21
9.5.2 What Happens When You Create an Application Module Diagram ... 9-22
9.5.3 How to Use the Diagram to Edit the Application Module ... 9-22
9.5.4 How to Control Diagram Display Options ... 9-22
9.5.5 How to Filtering Method Names Displayed in the Diagram ... 9-23
9.5.6 How to Show Related Objects and Implementation Files in the Diagram ... 9-23
9.5.7 How to Publish the Application Module Diagram ... 9-24
9.5.8 How to Test the Application Module from the Diagram ... 9-24
9.6 Supporting Multipage Units of Work ... 9-24
9.6.1 How to Simulate State Management in the Oracle ADF Model Tester ... 9-25
9.6.2 What Happens at Runtime: How the Application Uses Application Module Pooling and State Management 9-25
9.7 Customizing an Application Module with Service Methods ... 9-27
9.7.1 How to Generate a Custom Class for an Application Module ... 9-27
9.7.2 What Happens When You Generate a Custom Class for an Application Module . 9-28
9.7.3 What You May Need to Know About Default Code Generation ... 9-28
9.7.4 How to Add a Custom Service Method to an Application Module ... 9-29
9.7.5 How to Test the Custom Application Module Using a Static Main Method ... 9-30
9.7.6 What You May Need to Know About Programmatic Row Set Iteration ... 9-32

9.8 Customizing Application Module Message Strings 9-33
9.8.1 How to Add a Resource Bundle to an Application Module 9-33
9.8.2 What Happens When You Add a Resource Bundle to an Application Module 9-35
9.9 Publishing Custom Service Methods to UI Clients 9-35
9.9.1 How to Publish a Custom Method on the Application Module's Client Interface. 9-35
9.9.2 What Happens When You Publish Custom Service Methods 9-36
9.9.3 How to Generate Client Interfaces for View Objects and View Rows 9-37
9.9.4 How to Test Custom Service Methods Using the Oracle ADF Model Tester 9-38
9.9.5 What You May Need to Know About Method Signatures on the Client Interface 9-39
9.9.6 What You May Need to Know About Passing Information from the Data Model 9-40
9.10 Working Programmatically with an Application Module's Client Interface 9-40
9.10.1 How to Work Programmatically with an Application Module's Client Interface .. 9-40
9.10.2 What Happens at Runtime: How the Application Module's Client Interface is Accessed 9-42
9.10.3 How to Access an Application Module Client Interface in a Fusion Web Application 9-43
9.11 Overriding Built-in Framework Methods 9-45
9.11.1 How to Override a Built-in Framework Method 9-45
9.11.2 What Happens When You Override a Built-in Framework Method 9-46
9.11.3 How to Override prepareSession() to Set Up an Application Module for a New User Session 9-47
9.12 Calling a Web Service from an Application Module 9-48
9.12.1 How to Call an External Service Programmatically 9-49
9.12.1.1 Creating a Web Service Proxy Class to Programmatically Access the Service 9-50
9.12.1.2 Calling the Web Service Proxy Template to Invoke the Service 9-50
9.12.1.3 Calling a Web Service Method Using the Proxy Class in an Application Module 9-51
9.12.2 What Happens When You Create the Web Service Proxy 9-51
9.12.3 What Happens at Runtime: When You Call a Web Service Using a Web Service Proxy Class 9-52
9.12.4 What You May Need to Know About Web Service Proxies 9-52
9.12.4.1 Using a Try-Catch Block to Handle Web Service Exceptions 9-52
9.12.4.2 Separating Application Module and Web Services Transactions 9-53
9.12.4.3 Setting Browser Proxy Information 9-53
9.12.4.4 Invoking Application Modules with a Web Service Proxy Class 9-53

10 Sharing Application Module View Instances

10.1 About Shared Application Modules 10-1
10.1.1 Shared Application Module Use Cases and Examples 10-1
10.1.2 Additional Functionality for Shared Application Modules 10-2
10.2 Sharing an Application Module Instance 10-2
10.2.1 How to Create a Shared Application Module Instance 10-3
10.2.2 What Happens When You Define a Shared Application Module 10-4
10.2.3 What You May Need to Know About Design Time Scope of the Shared Application Module 10-6
10.2.4 What You May Need to Know About the Design Time Scope of View Instances of the Shared Application Module 10-6

10.2.5 What You May Need to Know About Managing the Number of Shared Query Collections 10-6
10.2.6 What You May Need to Know About Shared Application Modules and Connection Pooling 10-7
10.3 Defining a Base View Object for Use with Lookup Tables ... 10-7
10.3.1 How to Create a Base View Object Definition for a Lookup Table ... 10-8
10.3.2 What Happens When You Create a Base View Object ... 10-10
10.3.3 How to Define the WHERE Clause of the Lookup View Object Using View Criteria ... 10-12
10.3.4 What Happens When You Create a View Criteria with the Editor ... 10-14
10.3.5 What Happens at Runtime: How a View Instance Accesses Lookup Data ... 10-15
10.4 Accessing View Instances of the Shared Service ... 10-15
10.4.1 How to Create a View Accessor for an Entity Object or View Object ... 10-16
10.4.2 How to Validate Against the Attribute Values Specified by a View Accessor ... 10-18
10.4.3 What Happens When You Define a View Accessor Validator ... 10-19
10.4.4 What You May Need to Know About Dynamic Filtering with View Accessors ... 10-20
10.4.5 How to Create an LOV Based on a Lookup Table ... 10-20
10.4.6 What Happens When You Define an LOV for a View Object Attribute ... 10-22
10.4.7 How to Automatically Refresh the View Object of the View Accessor ... 10-23
10.4.8 What Happens at Runtime: How the Attribute Displays the List of Values ... 10-23
10.4.9 What You May Need to Know About Displaying List of Values From a Lookup Table ... 10-24
10.4.10 What You May Need to Know About Programmatically Invoking Database Change Notifications 10-24
10.4.11 What You May Need to Know About Inheritance of AttributeDef Properties ... 10-25
10.4.12 What You May Need to Know About Using Validators ... 10-25
10.5 Testing View Object Instances in a Shared Application Module ... 10-25
10.5.1 How to Test the Base View Object Using the Oracle ADF Model Tester ... 10-25
10.5.2 How to Test LOV-Enabled Attributes Using the Oracle ADF Model Tester ... 10-27
10.5.3 What Happens When You Use the Oracle ADF Model Tester ... 10-27
10.5.4 What Happens at Runtime: How Another Service Accesses the Shared Application Module Cache 10-27

11 Integrating Service-Enabled Application Modules

11.1 About Service-Enabled Application Modules ... 11-1
11.1.1 Service-Enabled Application Module Use Cases and Examples ... 11-2
11.1.2 Additional Functionality for Service-Enabled Application Modules ... 11-2
11.2 Publishing Service-Enabled Application Modules ... 11-2
11.2.1 How to Enable the Application Module Service Interface ... 11-3
11.2.2 What Happens When You Create an Application Module Service Interface ... 11-8
11.2.2.1 Remote Common Interface ... 11-10
11.2.2.2 Remote Service Schema File ... 11-11
11.2.2.3 Remote Service Definition File ... 11-11
11.2.2.4 Remote Server Class ... 11-12
11.2.2.5 connections.xml ... 11-13
11.2.3 What You May Need to Know About Method Signatures on the ADF Web Service Interface 11-14
11.2.4 How to Service-Enable Individual View Objects ... 11-15

11.2.5 How to Customize the SDO Properties of Service-Enabled View Objects 11-17
11.2.5.1 Excluding Individual SDO Properties in a Generated SDO Component..... 11-17
11.2.5.2 Associating Related SDO Properties Using Complex Data Types..... 11-17
11.2.6 How to Support Nested Processing in Service-Enabled Master-Detail View Objects 11-19
11.2.7 What Happens When You Create SDO Classes..... 11-21
11.2.7.1 Service Data Object Interface 11-21
11.2.7.2 Service Data Object Class 11-21
11.2.7.3 Service Data Object Schema File..... 11-21
11.2.7.4 Service Data Object Result Class and Interface..... 11-22
11.2.8 How to Expose a Declarative Find Operation Filtered By a Required Bind Variable..... 11-23
11.2.9 How to Expose a Custom Find Method Filtered By a Required Bind Variable 11-24
11.2.10 How to Generate Asynchronous ADF Web Service Methods 11-25
11.2.11 What Happens When You Generate Asynchronous ADF Web Service Methods 11-25
11.2.12 What Happens at Runtime: How the Asynchronous Call Is Made 11-27
11.2.13 How to Set Preferences for Generating the ADF Web Service Interface..... 11-28
11.2.14 How to Secure the ADF Web Service for Access By SOAP Clients 11-29
11.2.15 How to Secure the ADF Web Service for Access By RMI Clients 11-32
11.2.15.1 Enabling Authentication for RMI Clients 11-32
11.2.15.2 Enabling Authorization for RMI Clients..... 11-35
11.2.16 How to Grant Test Users Access to the Service..... 11-36
11.2.17 How to Enable Support for Binary Attachments for SOAP Clients..... 11-39
11.2.18 How to Test the Web Service Using Integrated WebLogic Server 11-40
11.2.19 How to Prevent Custom Service Methods from Timing Out..... 11-41
11.2.20 How to Deploy Web Services to Oracle WebLogic Server 11-43
11.3 Accessing Remote Data Over the Service-Enabled Application Module..... 11-45
11.3.1 How to Use Service-Enabled Entity Objects and View Objects..... 11-46
11.3.1.1 Creating Entity Objects Backed by SDO Services..... 11-46
11.3.1.2 Creating View Objects Backed by SDO Services 11-48
11.3.2 What Happens When You Create Service-Backed Business Components 11-48
11.3.3 How to Update the Data Model for Service-Backed Business Components 11-49
11.3.4 How to Configure the Service-Backed Business Components Runtime 11-51
11.3.4.1 Adding the SDO Client Library to the Classpath 11-52
11.3.4.2 Registering the ADF Business Components Service in the Consuming Application's connections.xml for the EJB RMI Protocol 11-53
11.3.4.3 Registering the ADF Business Components Service in the Consuming Application's connections.xml for the SOAP Protocol 11-55
11.3.4.4 Registering the ADF Business Components Service in the Consuming Application's connections.xml for Fabric SDO Binding 11-58
11.3.5 How to Test the Service-Backed Components in the Oracle ADF Model Tester.. 11-59
11.3.6 How to Invoke Operations of the Service-Backed Components in the Consuming Application 11-60
11.3.7 What You May Need to Know About Creating Service Data Objects in the Consuming Application 11-61
11.3.8 What Happens at Runtime: How the Application Accesses the Published Application Module 11-61

11.3.9 What You May Need to Know About Service-Backed Entity Objects and View Objects. 11-61

12 Extending Business Components Functionality

12.1 About Extending Business Components Functionality 12-1
12.1.1 Extending Business Components Use Cases and Examples 12-2
12.1.2 Additional Functionality for Extending Business Components 12-2
12.2 Globally Extending ADF Business Components Functionality 12-2
12.2.1 How To Create a Framework Extension Class 12-2
12.2.2 What Happens When You Create a Framework Extension Class 12-4
12.2.3 How to Base an ADF Component on a Framework Extension Class 12-4
12.2.4 How to Define Framework Extension Classes for All New Components 12-6
12.2.5 How to Define Framework Extension Classes for All New Projects 12-7
12.2.6 What Happens When You Base a Component on a Framework Extension Class .. 12-7
12.2.6.1 XML-Only Components 12-7
12.2.6.2 Components with Custom Java Classes 12-8
12.2.7 What You May Need to Know About Updating the Extends Clause in Custom Component Java Files 12-9
12.3 Creating a Layer of Framework Extensions 12-9
12.3.1 How to Create Your Layer of Framework Extension Layer Classes 12-10
12.3.2 How to Package Your Framework Extension Layer in a JAR File 12-11
12.3.3 How to Create a Library Definition for Your Framework Extension JAR File 12-11
12.4 Customizing Framework Behavior with Extension Classes 12-12
12.4.1 How to Access Runtime Metadata For View Objects and Entity Objects 12-12
12.4.2 How to Implement Generic Functionality Using Runtime Metadata 12-13
12.4.3 How to Implement Generic Functionality Driven by Custom Properties 12-14
12.4.4 What You May Need to Know About the Kinds of Attributes 12-15
12.4.5 What You May Need to Know About Custom Properties 12-15
12.5 Creating Generic Extension Interfaces 12-15
12.6 Invoking Stored Procedures and Functions 12-18
12.6.1 How to Invoke Stored Procedures with No Arguments 12-18
12.6.2 How to Invoke Stored Procedure with Only IN Arguments 12-18
12.6.3 How to Invoke Stored Function with Only IN Arguments 12-19
12.6.4 How to Call Other Types of Stored Procedures 12-21
12.7 Accessing the Current Database Transaction 12-22
12.8 Customizing Business Components Error Messages 12-23
12.8.1 How to Customize Base ADF Business Components Error Messages 12-24
12.8.2 What Happens When You Customize Base ADF Business Components Error Messages 12-26
12.8.3 How to Display Customize Error Messages as Nested Exceptions 12-26
12.8.4 How to Customize Error Messages for Database Constraint Violations 12-28
12.8.5 How to Implement a Custom Constraint Error Handling Routine 12-28
12.8.5.1 Creating a Custom Database Transaction Framework Extension Class 12-28
12.8.5.2 Configuring an Application Module to Use a Custom Database Transaction Class . 12-29
12.9 Creating Extended Components Using Inheritance 12-30
12.9.1 How To Create a Component That Extends Another 12-31
12.9.2 How To Extend a Component After Creation 12-31

12.9.3 What Happens When You Create a Component That Extends Another 12-31
12.9.3.1 Understanding an Extended Component's XML Descriptor 12-31
12.9.3.2 Understanding Java Code Generation for an Extended Component 12-32
12.9.4 What You May Need to Know 12-32
12.9.4.1 You Can Use Parent Classes and Interfaces to Work with Extended Components ... 12-32
12.9.4.2 Class Extends is Disabled for Extended Components 12-34
12.9.4.3 Interesting Aspects You Can Extend for Key Component Types 12-35
12.9.4.4 Extended Components Have Attribute Indices Relative to Parent 12-35
12.10 Substituting Extended Components in a Delivered Application 12-35
12.10.1 How To Substitute an Extended Component 12-36
12.10.2 What Happens When You Substitute 12-37
12.10.3 How to Enable the Substituted Components in the Base Application 12-38

Part III Using the ADF Model Layer

13 Using ADF Model in a Fusion Web Application

13.1 About ADF Data Binding 13-1
13.2 Additional Functionality 13-3
13.3 Exposing Application Modules with Oracle ADF Data Controls 13-3
13.3.1 How an Application Module Data Control Appears in the Data Controls Panel... 13-4
13.3.1.1 How the Data Model and Service Methods Appear in the Data Controls Panel..... 13-6
13.3.1.2 How Transaction Control Operations Appear in the Data Controls Panel 13-6
13.3.1.3 How View Objects Appear in the Data Controls Panel 13-7
13.3.1.4 How Nested Application Modules Appear in the Data Controls Panel 13-9
13.3.2 How to Open the Data Controls Panel 13-10
13.3.3 How to Refresh the Data Controls Panel 13-10
13.3.4 Packaging a Data Control for Use in Another Project 13-11
13.4 Using the Data Controls Panel 13-11
13.4.1 How to Use the Data Controls Panel 13-14
13.4.2 What Happens When You Use the Data Controls Panel 13-16
13.4.3 What Happens at Runtime: How the Binding Context Works 13-17
13.5 Working with the DataBindings.cpx File 13-18
13.5.1 How JDeveloper Creates a DataBindings.cpx File 13-18
13.5.2 What Happens When JDeveloper Creates a DataBindings.cpx File 13-18
13.6 Configuring the ADF Binding Filter 13-20
13.6.1 How JDeveloper Configures the ADF Binding Filter 13-21
13.6.2 What Happens When JDeveloper Configures an ADF Binding Filter 13-21
13.6.3 What Happens at Runtime: How the ADF Binding Filter Works 13-21
13.7 Working with Page Definition Files 13-22
13.7.1 How JDeveloper Creates a Page Definition File 13-22
13.7.2 What Happens When JDeveloper Creates a Page Definition File 13-23
13.7.2.1 Bindings Binding Objects Defined in the Page Definition File 13-26
13.7.2.2 Executable Binding Objects Defined in the Page Definition File 13-28
13.8 Creating ADF Data Binding EL Expressions 13-31
13.8.1 How to Create an ADF Data Binding EL Expression 13-31

13.8.1.1 Opening the Expression Builder from the Property Inspector 13-32
13.8.1.2 Using the Expression Builder 13-32
13.8.2 What You May Need to Know About ADF Binding Properties 13-33
13.9 Using Simple UI First Development 13-34
13.9.1 How to Apply ADF Model Data Binding to Existing UI Components 13-35
13.9.2 What Happens When You Apply ADF Model Data Binding to UI Components 13-36

14 Exposing Web Services Using the ADF Model Layer

14.1 About Web Services in Fusion Web Applications 14-1
14.1.1 Web Services Use Cases and Examples 14-1
14.1.2 Additional Functionality for Web Services in Fusion Applications 14-2
14.2 Creating Web Service Data Controls 14-2
14.2.1 How to Create a Web Service Data Control 14-2
14.2.2 How to Adjust the Endpoint for a Web Service Data Control 14-3
14.2.3 How to Refresh a Web Service Data Control 14-3
14.2.4 What You May Need to Know About Web Service Data Controls 14-4
14.3 Creating a New Web Service Connection 14-6
14.3.1 How to Create a New Web Service Connection 14-6
14.4 Securing Web Service Data Controls 14-7
14.4.1 WS-Security Specification 14-8
14.4.2 Using Key Stores 14-8
14.4.3 How to Define Web Service Data Control Security 14-8

15 Exposing URL Services Using the ADF Model Layer

15.1 About Using ADF Model with URL Services 15-1
15.1.1 URL Services Use Cases and Examples 15-1
15.1.2 Additional Functionality for URL Services 15-2
15.2 Exposing URL Services with ADF Data Controls 15-2
15.2.1 How to Create a URL Connection 15-2
15.2.2 How to Create a URL Service Data Control 15-3
15.2.3 What You May Need to Know About URL Service Data Controls 15-4
15.3 Using URL Service Data Controls 15-4

16 Using Validation in the ADF Model Layer

16.1 About ADF Model Layer Validation 16-1
16.1.1 ADF Model Layer Validation Use Cases and Examples 16-1
16.1.2 Additional Functionality for ADF Model Layer Validation 16-2
16.2 Defining Validation Rules in the ADF Model Layer 16-2
16.2.1 How to Add Validation 16-2
16.2.2 What Happens at Runtime: Model Validation Rules 16-3
16.3 Customizing Error Handling 16-3
16.3.1 How to Customize the Detail Portion of a Message 16-5
16.3.2 How to Write an Error Handler to Deal with Multiple Threads 16-6

17 Designing a Page Using Placeholder Data Controls

17.1 About Placeholder Data Controls 17-1

17.1.1 Placeholder Data Controls Use Cases and Examples ... 17-2
17.1.2 Additional Functionality for Placeholder Data Controls ... 17-2
17.2 Creating Placeholder Data Controls ... 17-3
17.2.1 How to Create a Placeholder Data Control ... 17-3
17.2.2 What Happens When You Create a Placeholder Data Control ... 17-4
17.3 Creating Placeholder Data Types ... 17-5
17.3.1 How to Create a Placeholder Data Type ... 17-6
17.3.2 What Happens When You Create a Placeholder Data Type ... 17-9
17.3.3 How to Configure a Placeholder Data Type Attribute to Be an LOV ... 17-10
17.3.3.1 Configuring an Attribute to Be a Fixed LOV ... 17-10
17.3.3.2 Configuring an Attribute to Be a Dynamic LOV ... 17-11
17.3.4 How to Create Master-Detail Data Types ... 17-13
17.3.5 What Happens When You Create a Master-Detail Data Type ... 17-15
17.3.6 How to Add Sample Data ... 17-15
17.3.6.1 Adding Sample Data Manually ... 17-15
17.3.6.2 Importing Sample Data ... 17-16
17.3.7 What Happens When You Add Sample Data ... 17-18
17.4 Using Placeholder Data Controls ... 17-18
17.4.1 Limitations of Placeholder Data Controls ... 17-18
17.4.2 Creating Layout ... 17-19
17.4.3 Creating a Search Form ... 17-19
17.4.4 Binding Components ... 17-19
17.4.5 Rebinding Components ... 17-19
17.4.6 Packaging Placeholder Data Controls to ADF Library JARs ... 17-19

Part IV Creating ADF Task Flows

18 Getting Started with ADF Task Flows

18.1 About ADF Task Flows ... 18-1
18.1.1 About Unbounded Task Flows ... 18-3
18.1.2 About Bounded Task Flows ... 18-4
18.1.3 About Control Flows ... 18-7
18.1.4 ADF Task Flow Use Cases and Examples ... 18-8
18.1.5 Additional Functionality for ADF Task Flows ... 18-9
18.2 Creating a Task Flow ... 18-10
18.2.1 How to Create a Task Flow ... 18-10
18.2.2 What Happens When You Create a Task Flow ... 18-13
18.2.3 What You May Need to Know About the Default Activity in a Bounded Task Flow ... 18-14
18.2.4 What You May Need to Know About Memory Scope for Task Flows ... 18-15
18.2.5 What Happens at Runtime: Using Task Flows ... 18-17
18.3 Adding Activities to a Task Flow ... 18-17
18.3.1 How to Add an Activity to a Task Flow ... 18-18
18.3.2 What Happens When You Add an Activity to a Task Flow ... 18-19
18.4 Adding Control Flow Rules to Task Flows ... 18-19
18.4.1 How to Add a Control Flow Rule to a Task Flow ... 18-20
18.4.2 How to Add a Wildcard Control Flow Rule ... 18-22

18.4.3 What Happens When You Create a Control Flow Rule ... 18-22
18.4.4 What Happens at Runtime: Evaluating Control Flow Rules ... 18-24
18.5 Testing Task Flows ... 18-24
18.5.1 How to Run a Bounded Task Flow That Contains Pages ... 18-25
18.5.2 How to Run a Bounded Task Flow That Uses Page Fragments ... 18-26
18.5.3 How to Run a Bounded Task Flow That Has Parameters ... 18-26
18.5.4 How to Run a JSF Page When Testing a Task Flow ... 18-27
18.5.5 How to Run the Unbounded Task Flow ... 18-27
18.5.6 How to Set a Run Configuration for a Project ... 18-28
18.6 Refactoring to Create New Task Flows and Task Flow Templates ... 18-29
18.6.1 How to Create a Bounded Task Flow from Selected Activities ... 18-29
18.6.2 How to Create a Task Flow from JSF Pages ... 18-30
18.6.3 How to Convert Bounded Task Flows ... 18-31
18.7 What You Should Know About Task Flow Constraints ... 18-32

19 Working with Task Flow Activities

19.1 About Task Flow Activities ... 19-1
19.1.1 Task Flow Activities Use Cases and Examples ... 19-3
19.1.2 Additional Functionality for Task Flow Activities ... 19-4
19.2 Using View Activities ... 19-4
19.2.1 Passing Control Between View Activities ... 19-6
19.2.1.1 How to Pass Control Between View Activities ... 19-6
19.2.1.2 What Happens When You Pass Control Between View Activities ... 19-7
19.2.2 Bookmarking View Activities ... 19-8
19.2.2.1 How to Create a Bookmarkable View Activity ... 19-9
19.2.2.2 What Happens When You Designate a View as Bookmarkable ... 19-10
19.2.3 Specifying HTTP Redirect for a View Activity ... 19-10
19.2.3.1 How to Specify HTTP Redirect for a View Activity ... 19-10
19.2.3.2 What Happens When You Specify HTTP Redirect for a View Activity ... 19-11
19.3 Using URL View Activities ... 19-11
19.3.1 How to Add a URL View Activity to a Task Flow ... 19-12
19.3.2 What You May Need to Know About URL View Activities ... 19-13
19.4 Using Router Activities ... 19-13
19.4.1 How to Configure Control Flow Using a Router Activity ... 19-15
19.4.2 What Happens When You Configure Control Flow Using a Router Activity ... 19-16
19.5 Using Method Call Activities ... 19-16
19.5.1 How to Add a Method Call Activity ... 19-18
19.5.2 How to Specify Method Parameters and Return Values ... 19-20
19.5.3 What Happens When You Add a Method Call Activity ... 19-21
19.6 Using Task Flow Call Activities ... 19-22
19.6.1 How to Call a Bounded Task Flow Using a Task Flow Call Activity ... 19-22
19.6.2 What Happens When You Call a Bounded Task Flow Using a Task Flow Call Activity ... 19-24
19.6.3 How to Specify Input Parameters on a Task Flow Call Activity ... 19-24
19.6.4 How to Call a Bounded Task Flow Using a URL ... 19-25
19.6.5 What Happens When You Configure a Bounded Task Flow to be Invoked by a URL ... 19-27

19.6.6 What You May Need to Know About Calling a Bounded Task Flow Using a URL 19-27
19.6.7 How to Specify Before and After Listeners..... 19-29
19.6.8 What Happens When You Add a Task Flow Call Activity 19-30
19.6.9 What Happens at Runtime When a Task Flow Call Activity Invokes a Task Flow 19-31
19.7 Using Task Flow Return Activities 19-31
19.8 Using Save Point Restore Activities 19-34
19.9 Using Parent Action Activities..... 19-34
19.10 Using Task Flow Activities with Page Definition Files 19-34
19.10.1 How to Associate a Page Definition File with a Task Flow Activity 19-35
19.10.2 What Happens When You Associate a Page Definition File with a Task Flow Activity .. 19-36

20 Using Parameters in Task Flows

20.1 About Using Parameters in Task Flows 20-1
20.1.1 Task Flow Parameters Use Cases and Examples 20-1
20.1.2 Additional Functionality for Task Flows Using Parameters..... 20-2
20.2 Passing Parameters to a View Activity..... 20-2
20.2.1 How to Pass Parameters to a View Activity 20-2
20.2.2 What Happens When You Pass Parameters to a View Activity..... 20-3
20.2.3 What You May Need to Know About Specifying Parameter Values 20-4
20.3 Passing Parameters to a Bounded Task Flow 20-4
20.3.1 How to Pass an Input Parameter to a Bounded Task Flow..... 20-5
20.3.2 What Happens When You Pass an Input Parameter to a Bounded Task Flow..... 20-7
20.4 Sharing Data Controls Between Task Flows 20-8
20.4.1 How to Share a Data Control Between Task Flows..... 20-9
20.4.2 What Happens When You Share a Data Control Between Task Flows..... 20-9
20.4.3 What You May Need to Know About Managing Transactions..... 20-10
20.5 Specifying a Return Value from a Bounded Task Flow 20-10
20.5.1 How to Specify a Return Value from a Bounded Task Flow 20-11
20.5.2 What Happens When You Specify a Return Value from a Bounded Task Flow .. 20-12

21 Using Task Flows as Regions

21.1 About Using Task Flows in ADF Regions..... 21-1
21.1.1 About Page Fragments and ADF Regions 21-2
21.1.2 About View Ports and ADF Regions 21-3
21.1.3 Task Flows and ADF Region Use Cases and Examples..... 21-4
21.1.4 Additional Functionality for Task Flows that Render in ADF Regions 21-5
21.2 Creating an ADF Region..... 21-5
21.2.1 How to Create an ADF Region 21-6
21.2.2 What Happens When You Create an ADF Region 21-7
21.3 Specifying Parameters for an ADF Region..... 21-8
21.3.1 How to Specify Parameters for an ADF Region..... 21-8
21.3.2 What Happens When You Specify Parameters for an ADF Region..... 21-9
21.4 Specifying Parameters for ADF Regions Using Parameter Maps..... 21-10

21.4.1 How to Create a Parameter Map to Specify Input Parameters for an ADF Region 21-10
21.4.2 What Happens When You Create a Parameter Map to Specify Input Parameters..... 21-11
21.5 Refreshing an ADF Region 21-12
21.5.1 How to Configure the Refresh of an ADF Region 21-13
21.5.2 What You May Need to Know About Refreshing an ADF Region 21-14
21.6 Configuring Activation of an ADF Region 21-15
21.6.1 How to Configure Activation of an ADF Region..... 21-16
21.6.2 What Happens When You Configure Activation of an ADF Region 21-17
21.7 Navigating Outside an ADF Region's Task Flow 21-18
21.7.1 How to Trigger Navigation Outside of an ADF Region's Task Flow 21-18
21.7.2 What Happens When You Configure Navigation Outside a Task Flow 21-19
21.7.3 What You May Need to Know About How a Page Determines the Capabilities of an ADF Region 21-20
21.8 Creating ADF Dynamic Regions 21-20
21.8.1 How to Create an ADF Dynamic Region 21-21
21.8.2 What Happens When You Create an ADF Dynamic Region..... 21-23
21.9 Adding Additional Task Flows to an ADF Dynamic Region..... 21-24
21.9.1 How to Create an ADF Dynamic Region Link 21-25
21.9.2 What Happens When You Create an ADF Dynamic Region 21-26
21.10 Configuring a Page To Render an Unknown Number of Regions..... 21-26
21.10.1 How to Configure a Page to Render an Unknown Number of Regions 21-27
21.10.2 What Happens When You Configure a Page to Render an Unknown Number of Regions 21-28
21.10.3 What You May Need to Know About Configuring a Page to Render an Unknown Number of Regions 21-29

22 Creating Complex Task Flows

22.1 About Creating Complex Task Flows..... 22-1
22.1.1 Complex Task Flows Use Cases and Examples 22-2
22.1.2 Additional Functionality for Complex Task Flows 22-3
22.2 Managing Transactions in Task Flows 22-4
22.2.1 How to Enable Transactions in a Bounded Task Flow 22-5
22.2.2 What Happens When You Specify Transaction Options 22-6
22.2.3 What You May Need to Know About Data Control Scoping with Task Flows..... 22-7
22.3 Reentering Bounded Task Flows..... 22-8
22.3.1 How to Set Reentry Behavior 22-8
22.3.2 How to Set Outcome-Dependent Options 22-9
22.3.3 What You Should Know About Managed Bean Values Upon Task Flow Reentry 22-10
22.4 Handling Exceptions in Task Flows..... 22-10
22.4.1 How to Designate an Activity as an Exception Handler 22-11
22.4.2 What Happens When You Designate an Activity as an Exception Handler..... 22-12
22.4.3 How to Designate Custom Code as an Exception Handler..... 22-12
22.4.4 What Happens When You Designate Custom Code as an Exception Handler 22-13
22.4.5 What You May Need to Know About Handling Exceptions During Transactions 22-14

22.4.6 What You May Need to Know About Handling Validation Errors 22-14
22.5 Configuring Your Application to Use Save Points 22-14
22.5.1 How to Configure Your Fusion Web Application to Use Save Points 22-15
22.5.2 What Happens When You Configure a Fusion Web Application to Use Save Points 22-15
22.5.3 What You May Need to Know About the Database Table for Save Points 22-16
22.6 Using Save Points in Task Flows 22-16
22.6.1 How to Add a Save Point to a Task Flow 22-18
22.6.2 What Happens When You Add Save Points to a Task Flow 22-19
22.6.3 How to Restore a Save Point 22-19
22.6.4 What Happens When You Restore a Save Point 22-19
22.6.5 How to Use the Save Point Restore Finalizer 22-20
22.6.6 What Happens When a Task Flow Invokes a Save Point Restore Finalizer 22-20
22.6.7 How to Enable Implicit Save Points 22-20
22.6.8 What You May Need to Know About Enabling Implicit Save Points 22-21
22.6.9 What You May Need to Know About the Time-to-Live Period for a Save Point 22-22
22.7 Using Train Components in Bounded Task Flows 22-23
22.7.1 Creating a Task Flow as a Train 22-24
22.7.1.1 How to Create a Train in a Bounded Task Flow 22-25
22.7.1.2 What Happens When You Create a Task Flow as a Train 22-26
22.7.2 Invoking a Child Bounded Task Flow from a Train Stop 22-26
22.7.2.1 How to Invoke a Child Bounded Task Flow From a Train Stop 22-27
22.7.3 Grouping Task Flow Activities to Execute Between Train Stops 22-27
22.7.4 Disabling the Sequential Behavior of Train Stops in a Train 22-28
22.7.4.1 How to Disable the Sequential Behavior of a Train 22-29
22.7.4.2 What Happens When You Disable the Sequential Behavior a Train Stop 22-29
22.7.5 Changing the Label of a Train Stop 22-30
22.7.5.1 How to Change the Label of a Train Stop 22-30
22.7.5.2 What Happens When You Change the Label of a Train Stop 22-30
22.7.6 Configuring a Train to Skip a Train Stop 22-31
22.7.6.1 How to Configure a Train to Skip a Train Stop 22-31
22.7.6.2 What Happens When You Configure a Train to Skip a Train Stop 22-31
22.8 Creating Task Flow Templates 22-32
22.8.1 How to Create a Task Flow Template 22-34
22.8.2 What Happens When You Create a Task Flow Template 22-34
22.8.3 What You May Need to Know About Task Flow Templates 22-35
22.9 Creating a Page Hierarchy Using Task Flows 22-35
22.9.1 How to Create a Page Hierarchy 22-36
22.9.1.1 How to Create an XMLMenuModel Metadata File 22-38
22.9.1.2 How to Create a Submenu with a Hierarchy of Group and Child Nodes 22-39
22.9.1.3 How to Attach a Menu Hierarchy to Another Menu Hierarchy 22-40
22.9.2 What Happens When You Create a Page Hierarchy 22-41

23 Using Dialogs in Your Application

23.1 About Using Dialogs in Your Application 23-1
23.1.1 Using Dialogs in Your Application Use Cases and Examples 23-1
23.1.2 Additional Functionality for Using Dialogs in Your Application 23-1

23.2 Running a Bounded Task Flow in a Modal Dialog 23-2
23.2.1 How to Run a Bounded Task Flow in a Modal Dialog 23-3
23.2.2 How to Return a Value From a Modal Dialog 23-4
23.2.3 How to Refresh a Page After a Modal Dialog Returns 23-5
23.2.4 What You May Need to Know About Dialogs in an Application that Uses Task Flows . 23-6
23.3 Using the ADF Faces Dialog Framework 23-7
23.3.1 How to Define a JSF Navigation Rule for Opening a Dialog 23-10
23.3.2 How to Create the JSF Page That Opens a Dialog 23-11
23.3.3 How to Create the Dialog Page and Return a Dialog Value 23-12
23.3.4 What Happens at Runtime: Raising the Return Event from the Dialog 23-14
23.3.5 How to Pass a Value into a Dialog 23-14
23.3.6 What Happens at Runtime: Handling the LaunchEvent 23-16
23.3.7 How to Handle the Return Value 23-16
23.3.8 What Happens at Runtime: Handling the ReturnEvent on the Launching Component .. 23-17

Part V Creating a Databound Web User Interface

24 Getting Started with Your Web Interface

24.1 About Developing a Web Application with ADF Faces 24-1
24.1.1 Page Template and Managed Beans Use Cases and Examples 24-2
24.1.2 Additional Functionality for Page Templates and Managed Beans 24-2
24.2 Using Page Templates 24-2
24.2.1 How to Use ADF Data Binding in ADF Page Templates 24-4
24.2.2 What Happens When You Use ADF Model Layer Bindings on a Page Template . 24-6
24.2.3 How to Add a Databound Page Template to a Page Dynamically 24-7
24.2.4 What Happens at Runtime: How Pages Use Templates 24-8
24.3 Creating a Web Page 24-9
24.4 Using a Managed Bean in a Fusion Web Application 24-9
24.4.1 How to Use a Managed Bean to Store Information 24-12
24.4.2 What Happens When You Create a Managed Bean 24-14
24.4.3 How to Set Managed Bean Memory Scopes in a Server-Cluster Environment 24-15

25 Understanding the Fusion Page Lifecycle

25.1 About the Fusion Page Lifecycle 25-1
25.2 About the JSF and ADF Page Lifecycles 25-3
25.2.1 What You May Need to Know About Partial Page Rendering and Iterator Bindings 25-7
25.2.2 What You May Need to Know About Using the Refresh Property Correctly 25-10
25.2.3 What You May Need to Know About Task Flows and the Lifecycle 25-12
25.3 About Object Scope Lifecycles 25-13
25.3.1 What You May Need to Know About Object Scopes and Task Flows 25-15
25.4 Customizing the ADF Page Lifecycle 25-16
25.4.1 How to Create a Custom Phase Listener 25-16
25.4.2 How to Register a Listener Globally 25-17
25.4.3 What You May Need to Know About Listener Order 25-18

25.4.4 How to Register a Lifecycle Listener for a Single Page 25-18
25.4.5 What You May Need to Know About Extending RegionController for Page Fragments 25-19

26 Creating a Basic Databound Page

26.1 About Creating a Basic Databound Page 26-1
26.1.1 ADF Databound Forms Use Cases and Examples 26-1
26.1.2 Additional Functionality for Databound Forms 26-3
26.2 Using Attributes to Create Text Fields 26-4
26.2.1 How to Create a Text Field 26-4
26.2.2 What Happens When You Create a Text Field 26-5
26.2.2.1 Creating and Using Iterator Bindings 26-5
26.2.2.2 Creating and Using Value Bindings 26-6
26.2.2.3 Using EL Expressions to Bind UI Components 26-7
26.3 Creating a Basic Form 26-8
26.3.1 How to Create a Form 26-8
26.3.2 What Happens When You Create a Form 26-10
26.4 Incorporating Range Navigation into Forms 26-11
26.4.1 How to Insert Navigation Controls into a Form 26-11
26.4.2 What Happens When You Create Command Buttons 26-12
26.4.2.1 Action Bindings for Built-in Navigation Operations 26-12
26.4.2.2 Iterator RangeSize Attribute 26-13
26.4.2.3 EL Expressions Used to Bind to Navigation Operations 26-14
26.4.3 What Happens at Runtime: How Action Events and Action Listeners Work 26-15
26.4.4 What You May Need to Know About the Browser Back Button and Navigating Through Records 26-16
26.5 Creating a Form to Edit an Existing Record 26-16
26.5.1 How to Create Edit Forms 26-16
26.5.2 What Happens When You Use Built-in Operations to Change Data 26-18
26.6 Creating an Input Form 26-19
26.6.1 How to Create an Input Form Using a Task Flow 26-20
26.6.2 What Happens When You Create an Input Form Using a Task Flow 26-21
26.6.3 What Happens at Runtime: CreateInsert Action from the Method Activity 26-22
26.6.4 What You May Need to Know About Displaying Sequence Numbers 26-23
26.7 Using a Dynamic Form to Determine Data to Display at Runtime 26-23
26.7.1 How to Use Dynamic Forms 26-24
26.7.2 What Happens When You Use Dynamic Components 26-25
26.7.3 What Happens at Runtime: How Attribute Values Are Dynamically Determined 26-26
26.8 Modifying the UI Components and Bindings on a Form 26-26
26.8.1 How to Modify the UI Components and Bindings 26-26
26.8.2 What Happens When You Modify Attributes and Bindings 26-27

27 Creating ADF Databound Tables

27.1 About Creating ADF Databound Tables 27-1
27.1.1 ADF Databound Tables Use Cases and Examples 27-1
27.1.2 Additional Functionality for Databound Tables 27-2

27.2 Creating a Basic Table ... 27-3
27.2.1 How to Create a Basic Table ... 27-3
27.2.2 What Happens When You Create a Table ... 27-6
27.2.2.1 Iterator and Value Bindings for Tables ... 27-6
27.2.2.2 Code on the JSF Page for an ADF Faces Table ... 27-7
27.2.3 What You May Need to Know About Setting the Current Row in a Table ... 27-10
27.3 Creating an Editable Table ... 27-11
27.3.1 How to Create an Editable Table ... 27-12
27.3.2 What Happens When You Create an Editable Table ... 27-14
27.4 Creating an Input Table ... 27-15
27.4.1 How to Create an Input Table ... 27-15
27.4.2 What Happens When You Create an Input Table ... 27-16
27.4.3 What Happens at Runtime: How CreateInsert and Partial Page Refresh Work ... 27-17
27.4.4 What You May Need to Know About Creating a Row and Sorting Columns ... 27-17
27.4.5 What You May Need to Know About Create and CreateInsert ... 27-18
27.5 Modifying the Attributes Displayed in the Table ... 27-18
27.5.1 How to Modify the Displayed Attributes ... 27-18
27.5.2 How to Change the Binding for a Table ... 27-19
27.5.3 What Happens When You Modify Bindings or Displayed Attributes ... 27-19

28 Command Components to Invoke Functionality in the View Layer

28.1 About Command Components ... 28-1
28.1.1 Command Component Use Cases and Examples ... 28-1
28.1.2 Additional Functionality for Command Components ... 28-2
28.2 Creating Command Components to Execute Methods ... 28-2
28.2.1 How to Create a Command Component Bound to a Custom Method ... 28-3
28.2.2 What Happens When You Create Command Components Using a Method ... 28-3
28.2.2.1 Defining Method Action Binding ... 28-4
28.2.2.2 Using Parameters in a Method ... 28-4
28.2.2.3 Adding ADF Faces Component Code to JSF Page ... 28-4
28.2.2.4 Using EL Expressions to Bind to Methods ... 28-4
28.2.2.5 Using the Return Value from a Method Call ... 28-5
28.2.3 What Happens at Runtime: Command Button Method Bindings ... 28-6
28.3 Setting Parameter Values Using a Command Component ... 28-6
28.3.1 How to Set Parameters Using setPropertyListener Within a Command Component ... 28-6
28.3.2 What Happens When You Set Parameters ... 28-7
28.3.3 What Happens at Runtime: setPropertyListener for a Command Component ... 28-7
28.4 Overriding Declarative Methods ... 28-8
28.4.1 How to Override a Declarative Method ... 28-8
28.4.2 What Happens When You Override a Declarative Method ... 28-11

29 Displaying Master-Detail Data

29.1 About Displaying Master-Detail Data ... 29-1
29.1.1 Master-Detail Tables, Forms, and Trees Use Cases and Examples ... 29-2
29.1.2 Additional Functionality for Master-Detail Tables, Forms, and Trees ... 29-5
29.2 Common Functionality for Master-Detail Tables, Forms, and Trees ... 29-5

29.2.1 Identifying Master-Detail Objects on the Data Controls Panel ... 29-6
29.3 Using Tables and Forms to Display Master-Detail Objects ... 29-7
29.3.1 How to Display Master-Detail Objects in Tables and Forms ... 29-8
29.3.2 What Happens When You Create Master-Detail Tables and Forms ... 29-9
29.3.2.1 Code Generated in the JSF Page ... 29-9
29.3.2.2 Binding Objects Defined in the Page Definition File ... 29-10
29.3.3 What Happens at Runtime: ADF Iterator for Master-Detail Tables and Forms ... 29-11
29.3.4 What You May Need to Know About Displaying Master-Detail Widgets on Separate Pages 29-12
29.4 Using Trees to Display Master-Detail Objects ... 29-12
29.4.1 How to Display Master-Detail Objects in Trees ... 29-13
29.4.2 What Happens When You Create an ADF Databound Tree ... 29-16
29.4.2.1 Code Generated in the JSF Page ... 29-16
29.4.2.2 Binding Objects Defined in the Page Definition File ... 29-17
29.4.3 What Happens at Runtime: Displaying an ADF Databound Tree ... 29-18
29.5 Using Tree Tables to Display Master-Detail Objects ... 29-18
29.5.1 How to Display Master-Detail Objects in Tree Tables ... 29-19
29.5.2 What Happens When You Create a Databound Tree Table ... 29-19
29.5.2.1 Code Generated in the JSF Page ... 29-19
29.5.2.2 Binding Objects Defined in the Page Definition File ... 29-20
29.5.3 What Happens at Runtime: Events ... 29-20
29.5.4 How to Use the TargetIterator Property ... 29-21
29.6 Using Selection Events with Trees and Tables ... 29-22
29.6.1 How to Use Selection Events with Trees and Tables ... 29-22
29.6.2 What Happens at Runtime: RowKeySet Objects and SelectionEvent Events ... 29-24

30 Creating Databound Selection Lists and Shuttles

30.1 About Selection Lists and Shuttles ... 30-1
30.1.1 Selection Lists and Shuttles Use Cases and Examples ... 30-2
30.1.2 Additional Functionality for Selection Lists and Shuttles ... 30-2
30.2 Creating List of Values (LOV) Components ... 30-3
30.2.1 How to Create an LOV ... 30-8
30.2.2 What Happens When You Create an LOV ... 30-9
30.2.3 What You May Need to Know About List Validators and LOV ... 30-10
30.3 Creating a Selection List ... 30-10
30.3.1 How to Create a Model-Driven List ... 30-12
30.3.2 How to Create a Selection List Containing Fixed Values ... 30-13
30.3.3 How to Create a Selection List Containing Dynamically Generated Values ... 30-14
30.3.4 What Happens When You Create a Model-Driven Selection List ... 30-15
30.3.5 What Happens When You Create a Fixed Selection List ... 30-16
30.3.6 What You May Need to Know About Values in a Selection List ... 30-17
30.3.7 What Happens When You Create a Dynamic Selection List ... 30-18
30.4 Creating a List with Navigation List Binding ... 30-19
30.5 Creating a Databound Shuttle ... 30-19

31 Creating ADF Databound Search Forms

31.1 About Creating Search Forms 31-1
31.1.1 Implicit and Named View Criteria 31-2
31.1.2 List of Values (LOV) Input Fields 31-2
31.1.3 Search Form Use Cases and Examples 31-3
31.1.4 Additional Functionality for Search Forms 31-4
31.2 Creating Query Search Forms 31-4
31.2.1 Named Bind Variables in Query Search Forms 31-11
31.2.2 How to Create a Query Search Form with a Results Table or Tree Table 31-12
31.2.3 How to Create a Query Search Form and Add a Results Component Later 31-13
31.2.4 How to Persist Saved Searches into MDS 31-14
31.2.5 How to Set Default Search Binding Behavior 31-14
31.2.6 What You May Need to Know About Dependent Criterion 31-16
31.2.7 What Happens When You Create a Query Form 31-17
31.2.8 What Happens at Runtime: Search Forms 31-18
31.3 Setting Up Search Form Properties 31-18
31.3.1 How to Set Search Form Properties on the View Criteria 31-19
31.3.2 How to Set Search Form Properties on the Query Component 31-20
31.3.3 How to Create Custom Operators or Remove Standard Operators 31-21
31.4 Creating Quick Query Search Forms 31-23
31.4.1 How to Create a Quick Query Search Form with a Results Table or Tree Table .. 31-24
31.4.2 How to Create a Quick Query Search Form and Add a Results Component Later 31-25
31.4.3 How to Set the Quick Query Layout Format 31-26
31.4.4 What Happens When You Create a Quick Query Search Form 31-26
31.4.5 What Happens at Runtime: Quick Query 31-26
31.5 Creating Standalone Filtered Search Tables from Named View Criteria 31-27
31.5.1 How to Create Filtered Table and Query-by-Example Searches 31-28

32 Using More Complex Databound ADF Faces Components

32.1 About More Complex Databound ADF Faces Components 32-1
32.1.1 Complex Databound ADF Faces Components Use Cases and Examples 32-1
32.1.2 Additional Functionality of Complex Databound ADF Faces Components 32-2
32.2 Using the ADF Faces Calendar Component 32-2
32.2.1 How to Use the ADF Faces Calendar 32-4
32.2.2 What Happens When You Create a Calendar 32-6
32.2.3 What Happens at Runtime: How the Calendar Binding Works 32-8
32.3 Using the ADF Faces Carousel Component 32-8
32.3.1 How to Create a Databound Carousel Component 32-9
32.3.2 What Happens When You Create a Carousel 32-12

33 Creating Databound ADF Data Visualization Components

33.1 About ADF Data Visualization Components 33-1
33.1.1 End User and Presentation Features 33-3
33.1.2 Data Visualization Components Use Cases and Examples 33-3
33.1.3 Additional Functionality for Data Visualization Components 33-4

33.2 Creating Databound Graphs 33-4
33.2.1 How to Create a Graph 33-7
33.2.2 What Happens When You Use the Data Controls Panel to Create a Graph..... 33-10
33.2.3 What You May Need to Know About Using a Graph's Row Selection Listener for Master-Detail Processing 33-11
33.2.4 What You May Need to Know About Using Name-Value Pairs 33-12
33.2.5 How to Create a Graph Using Name-Value Pairs 33-13
33.2.6 How to Create a Databound Spark Chart 33-15
33.3 Creating Databound Gauges 33-17
33.3.1 How to Create a Databound Dial Gauge 33-19
33.3.2 What Happens When You Create a Dial Gauge from a Data Control..... 33-21
33.3.3 How to Create a Databound Status Meter Gauge Set 33-22
33.3.4 What Happens When You Create a Status Meter Gauge from a Data Control..... 33-25
33.4 Creating Databound Pivot Tables 33-26
33.4.1 How to Create a Pivot Table 33-27
33.4.2 What Happens When You Use the Data Controls Panel to Create a Pivot Table. 33-38
33.4.2.1 Bindings for Pivot Tables 33-39
33.4.2.2 Code on the JSF Page for a Pivot Table and Pivot Filter Bar 33-40
33.4.3 What You May Need to Know About Aggregating Attributes in the Pivot Table 33-41
33.4.3.1 Default Aggregation of Duplicate Data Rows..... 33-41
33.4.3.2 Custom Aggregation of Duplicate Rows 33-41
33.4.4 What You May Need to Know About Specifying an Initial Sort for a Pivot Table 33-42
33.5 Creating Databound Geographic Maps..... 33-43
33.5.1 How to Create a Geographic Map with a Point Theme 33-44
33.5.2 How to Create Point Style Items for a Point Theme 33-47
33.5.3 What Happens When You Create a Geographic Map with a Point Theme..... 33-49
33.5.3.1 Binding XML for a Point Theme 33-49
33.5.3.2 XML Code on the JSF Page for a Geographic Map and Point Theme 33-49
33.5.4 What You May Need to Know About Adding Custom Point Style Items to a Map Point Theme 33-50
33.5.5 How to Add a Databound Color Theme to a Geographic Map..... 33-51
33.5.6 What Happens When You Add a Color Theme to a Geographic Map 33-53
33.5.6.1 Binding XML for a Color Theme..... 33-53
33.5.6.2 XML Code on the JSF Page for a Color Theme 33-54
33.5.7 What You May Need to Know About Customizing Colors in a Map Color Theme 33-54
33.5.8 How to Add a Databound Pie Graph Theme to a Geographic Map..... 33-54
33.5.9 What Happens When You Add a Pie Graph Theme to a Geographic Map 33-56
33.5.9.1 Binding XML for a Pie Graph Theme..... 33-56
33.5.9.2 Code on the JSF Page for a Pie Graph Theme 33-57
33.6 Creating Databound Thematic Maps..... 33-57
33.6.1 How to Create a Thematic Map Using ADF Data Controls 33-57
33.6.2 What Happens When You Use Data Controls to Create a Thematic Map..... 33-64
33.6.3 What You May Need to Know About Base Map Location Ids 33-65
33.6.4 How to Add Data Layers to Thematic Maps..... 33-66
33.6.5 How to Configure Drilling in Thematic Maps 33-71
33.6.6 What You May Need to Know About Configuring Master-Detail Relationships 33-73

33.6.7 Styling Areas, Markers, and Images to Display Data ... 33-74
33.6.7.1 How to Style Areas to Display Data ... 33-74
33.6.7.2 How to Style Markers to Display Data ... 33-77
33.6.7.3 What You May Need to Know About Styling Markers ... 33-84
33.6.7.4 What You May Need to Know About Default Style Values for Attribute Groups ... 33-84
33.6.7.5 How to Style Images to Display Data ... 33-85
33.6.7.6 What You May Need to Know About SVG Files ... 33-87
33.6.8 Creating Databound Legends ... 33-87
33.6.9 How to Define a Custom Map Layer ... 33-90
33.7 Creating Databound Gantt Charts ... 33-94
33.7.1 How to Create a Databound Project Gantt Chart ... 33-95
33.7.2 What Happens When You Create a Project Gantt Chart from a Data Control ... 33-98
33.7.3 What You May Need to Know About Summary Tasks in a Project Gantt Chart 33-100
33.7.4 What You May Need to Know About Percent Complete in a Project Gantt Chart ... 33-100
33.7.5 What You May Need to Know About Variance in a Project Gantt Chart ... 33-101
33.7.6 How to Create a Databound Resource Utilization Gantt Chart ... 33-101
33.7.7 What Happens When You Create a Resource Utilization Gantt Chart ... 33-103
33.7.8 How to Create a Databound Scheduling Gantt Chart ... 33-105
33.7.9 What Happens When You Create a Scheduling Gantt Chart ... 33-108
33.8 Creating Databound Hierarchy Viewers ... 33-109
33.8.1 How to Create a Hierarchy Viewer Using ADF Data Controls ... 33-110
33.8.2 What Happens When You Create a Databound Hierarchy Viewer ... 33-114
33.8.3 How to Create a Databound Search in a Hierarchy Viewer ... 33-118

34 Using Contextual Events

34.1 About Creating Contextual Events ... 34-1
34.1.1 Contextual Events Use Cases and Examples ... 34-4
34.1.2 Additional Functionality for Contextual Events ... 34-5
34.2 Creating Contextual Events Declaratively ... 34-5
34.2.1 How to Publish Contextual Events ... 34-5
34.2.2 How to Subscribe to and Consume Contextual Events ... 34-7
34.2.3 What Happens When You Create Contextual Events ... 34-9
34.2.4 How to Control Contextual Events Dispatch ... 34-10
34.2.5 What Happens at Runtime: Contextual Events ... 34-11
34.3 Creating Contextual Events Manually ... 34-11
34.3.1 How to Create Contextual Events Manually ... 34-11
34.4 Creating Contextual Events Using Managed Beans ... 34-13
34.5 Creating Contextual Events Using JavaScript ... 34-14
34.6 Creating the Event Map Manually ... 34-14
34.6.1 How to Create the Event Map Manually ... 34-15
34.7 Registering a Custom Dispatcher ... 34-15
34.7.1 How to Register a Custom Dispatcher ... 34-16

Part VI Completing Your Application

35 Enabling ADF Security in a Fusion Web Application

35.1 About ADF Security 35-1
35.1.1 Integration of ADF Security and Java Security 35-3
35.1.2 ADF Security Use Cases and Examples 35-4
35.1.3 Additional Functionality for ADF Security 35-5
35.2 ADF Security Process Overview 35-6
35.3 Enabling ADF Security 35-9
35.3.1 How to Enable ADF Security 35-9
35.3.2 What Happens When You Enable ADF Security 35-13
35.3.3 What Happens When You Generate a Default Form-Based Login Page 35-17
35.3.4 What You May Need to Know About the Configure ADF Security Wizard 35-17
35.3.5 What You May Need to Know About ADF Authentication 35-17
35.3.6 What You May Need to Know About the Built-In test-all Role 35-17
35.3.7 What You May Need to Know About the valid-users Role 35-18
35.4 Creating Application Roles 35-18
35.4.1 How to Create Application Roles 35-19
35.4.2 What Happens When You Create Application Roles 35-19
35.4.3 What You May Need to Know About Enterprise Roles and Application Roles 35-20
35.5 Defining ADF Security Policies 35-21
35.5.1 How to Make an ADF Resource Public 35-21
35.5.2 What Happens When You Make an ADF Resource Public 35-24
35.5.3 What Happens at Runtime: How the Built-in Roles Are Used 35-24
35.5.4 How to Define Policies for ADF Bounded Task Flows 35-25
35.5.5 How to Define Policies for Web Pages That Reference a Page Definition 35-28
35.5.6 What Happens When You Define the Security Policy 35-31
35.5.7 What Happens at Runtime: How ADF Security Policies Are Enforced 35-32
35.5.8 What You May Need to Know About Defining Policies for Pages with No ADF Bindings 35-33
35.5.9 How to Use Regular Expressions to Define Policies on Groups of Resources 35-34
35.5.10 How to Define Policies for Data 35-35
35.5.10.1 Defining Permission Maps on ADF Entity Objects 35-36
35.5.10.2 Granting Permissions on ADF Entity Objects 35-37
35.5.11 How to Aggregate Resource Grants as Entitlement Grants 35-39
35.5.12 What Happens After You Create an Entitlement Grant 35-41
35.6 Creating Test Users 35-43
35.6.1 How to Create Test Users in JDeveloper 35-43
35.6.2 What Happens When You Create Test Users 35-45
35.6.3 How to Associate Test Users with Application Roles 35-46
35.6.4 What Happens When You Configure Application Roles 35-47
35.7 Creating a Login Page 35-48
35.7.1 How to Create a Login Link Component and Add it to a Public Web Page 35-48
35.7.2 How to Create a Login Page 35-50
35.7.2.1 Creating Login Code for the Backing Bean 35-51
35.7.2.2 Creating an ADF Faces-Based Login Page 35-55
35.7.2.3 Configuring the web.xml File for an ADF Faces-Based Login Page 35-58
35.7.2.4 Ensuring That the Login Page Is Public 35-59
35.7.2.5 Ensuring That the Login Page's Resources Are Accessible 35-59

35.7.3 How to Create a Public Welcome Page 35-60
35.7.3.1 Ensuring That the Welcome Page Is Public 35-60
35.7.3.2 Adding Login and Logout Links 35-60
35.7.3.3 Hiding Links to Secured Pages 35-61
35.7.4 How to Redirect a User After Authentication 35-61
35.7.5 What You May Need to Know About ADF Servlet Logout and Browser Caching 35-63
35.8 Testing Security in JDeveloper 35-63
35.8.1 How to Configure, Deploy, and Run a Secure Application in JDeveloper 35-63
35.8.2 What Happens When You Configure Security Deployment Options 35-64
35.8.3 How to Use the Built-In test-all Application Role 35-66
35.8.4 What Happens at Runtime: How ADF Security Handles Authentication 35-66
35.8.5 What Happens at Runtime: How ADF Security Handles Authorization 35-69
35.9 Preparing the Secure Application for Deployment 35-70
35.9.1 How to Remove the test-all Role from the Application Policy Store 35-71
35.9.2 How to Remove Test Users from the Application Identity Store 35-72
35.9.3 How to Secure Resource Files Using a URL Constraint 35-73
35.10 Disabling ADF Security 35-74
35.10.1 How to Disable ADF Security 35-74
35.10.2 What Happens When You Disable ADF Security 35-75
35.11 Advanced Topics and Best Practices 35-75
35.11.1 Using Expression Language (EL) with ADF Security 35-75
35.11.1.1 How to Evaluate Policies Using EL 35-75
35.11.1.2 What Happens When You Use the Expression Builder Dialog 35-79
35.11.1.3 What You May Need to Know About Delayed Evaluation of EL 35-80
35.11.2 How to Evaluate Policies Using Custom JAAS Permissions and EL 35-80
35.11.2.1 Creating the Custom JAAS Permission Class 35-81
35.11.2.2 Creating the ADF Security Policy Using a Custom Permission 35-82
35.11.2.3 Associating the Rendering of a UI Component with a Custom Permission 35-83
35.11.3 Getting Information from the ADF Security Context 35-85
35.11.3.1 How to Determine Whether Security Is Enabled 35-85
35.11.3.2 How to Determine Whether the User Is Authenticated 35-85
35.11.3.3 How to Determine the Current User Name, Enterprise Name, or Enterprise ID 35-86
35.11.3.4 How to Determine Membership of a Java EE Security Role 35-87
35.11.3.5 How to Determine Permission Using Java 35-87
35.11.4 Best Practices for Working with ADF Security 35-88

36 Testing and Debugging ADF Components

36.1 About ADF Debugging 36-1
36.2 Correcting Simple Oracle ADF Compilation Errors 36-2
36.3 Correcting Simple Oracle ADF Runtime Errors 36-4
36.4 Reloading Oracle ADF Metadata in Integrated WebLogic Server 36-6
36.5 Validating ADF Controller Metadata 36-6
36.6 Using the ADF Logger 36-7
36.6.1 How to Set ADF Logging Levels 36-7
36.6.2 How to Turn On Diagnostic Logging 36-10

36.6.3 How to Create an Oracle ADF Debugging Configuration 36-10
36.6.4 How to Use the Log Analyzer to View Log Messages 36-11
36.6.4.1 Viewing Diagnostic Messages in the Log Analyzer 36-11
36.6.4.2 Using the Log Analyzer to Analyze the ADF Request 36-13
36.6.4.3 Sorting Diagnostic Messages By ADF Events 36-17
36.6.5 What You May Need to Know About the Logging.xml File 36-21
36.6.6 What You May Need to Know About ADF Logging and Oracle WebLogic Server 36-22
36.7 Using the Oracle ADF Model Tester for Testing and Debugging 36-22
36.7.1 How to Run in Debug Mode and Test with the Oracle ADF Model Tester 36-22
36.7.2 How to Run the Oracle ADF Model Tester and Test with a Specific Configuration 36-23
36.7.3 What Happens When You Run the Oracle ADF Model Tester in Debug Mode 36-23
36.7.4 How to Verify Runtime Artifacts in the Oracle ADF Model Tester 36-24
36.7.5 How to Refresh the Oracle ADF Model Tester with Application Changes 36-24
36.8 Using the ADF Declarative Debugger 36-24
36.8.1 Using ADF Source Code with the Debugger 36-26
36.8.2 How to Set Up the ADF Source User Library 36-27
36.8.3 How to Add the ADF Source Library to a Project 36-27
36.8.4 How to Use the EL Expression Evaluator 36-28
36.8.5 How to View and Export Stack Trace Information 36-29
36.9 Setting ADF Declarative Breakpoints 36-29
36.9.1 How to Set and Use Task Flow Activity Breakpoints 36-38
36.9.2 How to Set and Use Page Definition Executable Breakpoints 36-39
36.9.3 How to Set and Use Page Definition Action Binding Breakpoints 36-41
36.9.4 How to Set and Use Page Definition Value Binding Breakpoints 36-42
36.9.5 How to Set and Use Page Definition Contextual Event Breakpoints 36-43
36.9.6 How to Set and Use ADF Lifecycle Phase Breakpoints 36-44
36.9.7 How to Use the ADF Structure Window 36-47
36.9.8 How to Use the ADF Data Window 36-50
36.9.9 What Happens When You Set an ADF Declarative Breakpoint 36-60
36.10 Setting Java Code Breakpoints 36-61
36.10.1 How to Set Java Breakpoints on Classes and Methods 36-61
36.10.2 How to Optimize Use of the Source Editor 36-62
36.10.3 How to Set Breakpoints and Debug Using ADF Source Code 36-63
36.10.4 How to Use Debug Libraries for Symbolic Debugging 36-63
36.10.5 How to Use Different Kinds of Java Code Breakpoints 36-65
36.10.6 How to Edit Breakpoints for Improved Control 36-66
36.10.7 How to Filter Your View of Class Members 36-67
36.10.8 How to Use Common Oracle ADF Breakpoints 36-67
36.11 Regression Testing with JUnit 36-68
36.11.1 How to Obtain the JUnit Extension 36-69
36.11.2 How to Create a JUnit Test Case 36-70
36.11.3 How to Create a JUnit Test Fixture 36-72
36.11.4 How to Create a JUnit Test Suite 36-72
36.11.5 How to Create a Business Components Test Suite 36-73
36.11.6 How to a Create Business Components Test Fixture 36-75

36.11.7 How to Run a JUnit Test Suite as Part of an Ant Build Script 36-76

37 Refactoring a Fusion Web Application

37.1 About Refactoring a Fusion Web Application 37-1
37.1.1 Refactoring Use Cases and Examples 37-1
37.2 Renaming Files 37-2
37.3 Moving JSF Pages 37-2
37.4 Refactoring pagedef.xml Bindings Objects 37-2
37.5 Refactoring ADF Business Components 37-3
37.6 Refactoring ADF Business Component Object Attributes 37-4
37.7 Refactoring Named Elements 37-5
37.8 Refactoring ADF Task Flows 37-6
37.9 Refactoring the DataBindings.cpx File 37-6
37.10 Refactoring Limitations 37-7
37.11 Moving the ADF Business Components Project Configuration File (.jpx) 37-8

38 Reusing Application Components

38.1 About Reusable Components 38-1
38.1.1 Creating Reusable Components 38-3
38.1.1.1 Naming Conventions 38-3
38.1.1.2 The Naming Process for the ADF Library JAR Deployment Profile 38-5
38.1.1.3 Keeping the Relevant Project 38-6
38.1.1.4 Selecting the Relevant Feature 38-6
38.1.1.5 Selecting Paths and Folders 38-6
38.1.1.6 Including Connections Within Reusable Components 38-6
38.1.2 Reusable ADF Components Use Cases and Examples 38-7
38.1.3 Additional Functionality for Reusable ADF Components 38-8
38.2 Common Functionality of Reusable ADF Components 38-8
38.2.1 Using Extension Libraries 38-8
38.2.2 Using the Resource Palette 38-12
38.3 Packaging a Reusable ADF Component into an ADF Library 38-13
38.3.1 How to Package a Component into an ADF Library JAR 38-14
38.3.2 What Happens When You Package a Project to an ADF Library JAR 38-19
38.3.2.1 Application Modules 38-19
38.3.2.2 Data Controls 38-20
38.3.2.3 Task Flows 38-20
38.3.2.4 Page Templates 38-20
38.3.2.5 Declarative Components 38-20
38.3.3 How to Place and Access JDeveloper JAR Files 38-20
38.4 Adding ADF Library Components into Projects 38-21
38.4.1 How to Add an ADF Library JAR into a Project using the Resource Palette 38-22
38.4.2 How to Add an ADF Library JAR into a Project Manually 38-24
38.4.3 What Happens When You Add an ADF Library JAR to a Project 38-25
38.4.4 What You May Need to Know About Using ADF Library Components 38-27
38.4.4.1 Using Data Controls 38-27
38.4.4.2 Using Application Modules 38-27
38.4.4.3 Using Business Components 38-27

38.4.4.4 Using Task Flows ... 38-28
38.4.4.5 Using Page Templates ... 38-29
38.4.4.6 Using Declarative Components ... 38-30
38.4.5 What You May Need to Know About Differentiating ADF Library Components ... 38-30
38.4.6 What Happens at Runtime: Adding ADF Libraries ... 38-30
38.5 Removing an ADF Library JAR from a Project ... 38-31
38.5.1 How to Remove an ADF Library JAR from a Project Using the Resource Palette 38-31
38.5.2 How to Remove an ADF Library JAR from a Project Manually ... 38-32

39 Customizing Applications with MDS

39.1 About Customization and MDS ... 39-1
39.1.1 Customization and Layers: Use Cases and Examples ... 39-2
39.1.2 Static and Dynamic Customization Content ... 39-3
39.1.3 Additional Functionality for Customization ... 39-3
39.2 Developing a Customizable Application ... 39-3
39.2.1 How to Create Customization Classes ... 39-4
39.2.1.1 Customization Classes ... 39-4
39.2.1.2 Implementing the getValue() Method in Your Customization Class ... 39-6
39.2.1.3 Creating a Customization Class ... 39-8
39.2.2 What You May Need to Know About Customization Classes ... 39-9
39.2.3 How to Consume Customization Classes ... 39-10
39.2.3.1 Making Customization Classes Available to JDeveloper at Design Time ... 39-10
39.2.3.2 Making Customization Classes Available to the Application at Run Time ... 39-11
39.2.4 How to Enable Seeded Customizations for View Projects ... 39-11
39.2.5 How to Enable Seeded Customizations in Existing Pages ... 39-12
39.2.6 How to Enable Customizations in Resource Bundles ... 39-13
39.2.7 How to Configure the adf-config.xml file ... 39-14
39.2.8 What Happens When You Create a Customizable Application ... 39-15
39.2.9 What You May Need to Know About Customizable Objects and Applications ... 39-15
39.3 Customizing an Application ... 39-16
39.3.1 Introducing the Customization Developer Role ... 39-16
39.3.2 How to Switch to the Customization Developer Role in JDeveloper ... 39-17
39.3.3 Introducing the Tip Layer ... 39-17
39.3.4 How to Configure Customization Layers ... 39-18
39.3.4.1 Configuring Layer Values Globally ... 39-19
39.3.4.2 Configuring Workspace-Level Layer Values from the Studio Developer Role ... 39-19
39.3.4.3 Configuring Workspace-Level Layer Values from the Customization Developer Role 39-20
39.3.5 How to Customize Metadata in JDeveloper ... 39-21
39.3.6 What Happens When You Customize an Application ... 39-23
39.3.7 How to Customize ADF Library Artifacts in JDeveloper ... 39-24
39.3.7.1 Specifying a Location for ADF Library Customizations ... 39-25
39.3.8 How to View ADF Library Runtime Customizations from Exported JARs ... 39-26
39.3.9 What Happens When You Customize ADF Library Artifacts ... 39-26
39.3.10 How to Package and Deploy Customized Applications ... 39-28

39.3.10.1 Implicitly Creating a MAR Profile 39-28
39.3.10.2 Explicitly Creating a MAR Profile 39-28
39.3.11 What Happens at Runtime in a Customized Application 39-29
39.3.12 What You May Need to Know About Customized Applications 39-29
39.3.12.1 Customization and Integrated Source Control 39-29
39.3.12.2 Editing Resource Bundles in Customized Applications 39-30
39.4 Extended Metadata Properties 39-31
39.4.1 How to Edit Extended Metadata Properties 39-31
39.4.2 How to Enable Customization for Design Time at Runtime 39-32
39.4.2.1 Editing Customization Properties in the Property Inspector 39-33
39.4.2.2 Using a Standalone Annotations File to Specify Type-Level Customization Properties 39-33
39.5 Enabling Runtime Modification of Customization Configuration 39-35

40 Allowing User Customizations at Runtime

40.1 About User Customizations 40-1
40.1.1 Runtime User Customization Use Cases and Examples 40-7
40.1.2 Additional Functionality for Runtime User Customization 40-8
40.2 Enabling Runtime User Customizations for a Fusion Web Application 40-8
40.2.1 How to Enable User Customizations 40-8
40.2.2 What Happens When You Enable User Customizations 40-9
40.3 Configuring User Customizations 40-10
40.3.1 How to Configure Change Persistence 40-11
40.3.2 What Happens When You Configure Change Persistence 40-12
40.4 Controlling User Customizations in Individual JSF Pages 40-13
40.4.1 How to Control User Customizations on a JSF Page 40-13
40.4.2 What Happens at Runtime 40-14
40.4.3 What You May Need to Know About Using Change Persistence on Templates and Regions 40-15
40.5 Implementing Custom User Customizations 40-16
40.5.1 Change Persistence Framework API 40-16
40.5.2 How to Create Code for Custom User Customizations 40-18
40.6 Creating Implicit Change Persistence in Custom Components 40-21
40.6.1 How to Set Implicit Change Persistence For Attribute Values that Use Events ... 40-21
40.6.2 How to Set Implicit Change Persistence For Other Attribute Values 40-22

41 Deploying Fusion Web Applications

41.1 About Deploying Fusion Web Applications 41-1
41.1.1 Developing Applications with Integrated WebLogic Server 41-3
41.1.2 Developing Applications to Deploy to Standalone Application Server 41-3
41.2 Running an ADF Application in Integrated WebLogic Server 41-5
41.2.1 How to Run an Application in Integrated WebLogic Server 41-6
41.2.2 How to Run an Application with Metadata in Integrated WebLogic Server 41-6
41.3 Preparing the Application 41-8
41.3.1 How to Create a Connection to the Target Application Server 41-9
41.3.2 How to Create Deployment Profiles 41-12
41.3.2.1 Creating a WAR Deployment Profile 41-12

41.3.2.2 Creating a MAR Deployment Profile ... 41-13
41.3.2.3 Creating an Application-Level EAR Deployment Profile ... 41-16
41.3.2.4 Delivering Customization Classes as a Shared Library ... 41-17
41.3.2.5 Viewing and Changing Deployment Profile Properties ... 41-18
41.3.2.6 Adding Customization Classes into a JAR ... 41-19
41.3.3 How to Create and Edit Deployment Descriptors ... 41-20
41.3.3.1 Creating Deployment Descriptors ... 41-21
41.3.3.2 Viewing or Modifying Deployment Descriptor Properties ... 41-22
41.3.3.3 Configuring the application.xml File for Application Server Compatibility ... 41-22
41.3.3.4 Configuring the web.xml File for Application Server Compatibility ... 41-23
41.3.3.5 Enabling the Application for Real User Experience Insight ... 41-23
41.3.4 How to Deploy Applications with ADF Security Enabled ... 41-24
41.3.4.1 Applications That Will Run Using Oracle Single Sign-On (SSO) ... 41-24
41.3.4.2 Configuring Security for Weblogic Server ... 41-25
41.3.4.3 Configuring Security for Websphere Application Server ... 41-27
41.3.5 How to Replicate Memory Scopes in a Clustered Environment ... 41-28
41.3.6 How to Enable the Application for ADF MBeans ... 41-28
41.3.7 What You May Need to Know About JDBC Data Source for Oracle WebLogic Server ... 41-29
41.4 Deploying the Application ... 41-30
41.4.1 How to Deploy to the Application Server from JDeveloper ... 41-33
41.4.2 How to Create an EAR File for Deployment ... 41-35
41.4.3 How to Deploy New Customizations Applied to ADF Library ... 41-36
41.4.3.1 Exporting Customization to a Deployed Application ... 41-36
41.4.3.2 Deploying Customizations to a JAR ... 41-37
41.4.4 What You May Need to Know About ADF Libraries ... 41-38
41.4.5 What You May Need to Know About EAR Files and Packaging ... 41-38
41.4.6 How to Deploy the Application Using Scripts and Ant ... 41-38
41.4.7 What You May Need to Know About JDeveloper Runtime Libraries ... 41-38
41.5 Postdeployment Configuration ... 41-38
41.5.1 How to Migrate an Application ... 41-39
41.5.2 How to Configure the Application Using ADF MBeans ... 41-39
41.6 Testing the Application and Verifying Deployment ... 41-39

Part VII Advanced Topics

42 Advanced View Object Techniques

42.1 Advanced View Object Concepts and Features ... 42-1
42.1.1 Limiting the View Object Max Fetch Size to Fetch the First n Rows ... 42-1
42.1.2 Maintaining New Row Consistency in View Objects Based on the Same Entity ... 42-2
42.1.2.1 What Happens at Runtime When View Link Consistency is Enabled ... 42-3
42.1.2.2 How to Change the Default View Link Consistency Setting ... 42-3
42.1.2.3 How to Use a RowMatch to Qualify Which New, Unposted Rows Get Added to a Row Set 42-4
42.1.2.4 What You May Need to Know About the Dynamic WHERE Clause and View Link Consistency 42-5
42.1.3 Understanding View Link Accessors Versus Data Model View Link Instances ... 42-5

42.1.3.1 Enabling a Dynamic Detail Row Set with Active Master-Detail Coordination 42-5
42.1.3.2 Accessing a Stable Detail Row Set Using View Link Accessor Attributes 42-6
42.1.3.3 Accessor Attributes Create Distinct Row Sets Based on an Internal View Object 42-6
42.1.4 Presenting and Scrolling Data a Page at a Time Using the Range 42-7
42.1.5 Efficiently Scrolling Through Large Result Sets Using Range Paging 42-8
42.1.5.1 Understanding How to Oracle Supports "TOP-N" Queries 42-9
42.1.5.2 How to Enable Range Paging for a View Object 42-10
42.1.5.3 What Happens When You Enable Range Paging 42-11
42.1.5.4 What Happens When View Rows are Cached When Using Range Paging 42-12
42.1.5.5 How to Scroll to a Given Page Number Using Range Paging 42-12
42.1.5.6 Estimating the Number of Pages in the Row Set Using Range Paging 42-12
42.1.5.7 Understanding the Tradeoffs of Using a Range Paging Mode 42-12
42.1.6 Setting Up a Data Model with Multiple Masters 42-13
42.1.7 Understanding When You Can Use Partial Keys with findByKey() 42-14
42.1.8 Handling View Object Queries with Primary Keys Defined by Transient Attributes 42-15
42.1.9 Creating Dynamic Attributes to Store UI State 42-16
42.1.10 Working with Multiple Row Sets and Row Set Iterators 42-16
42.1.11 Optimizing View Link Accessor Access By Retaining the Row Set 42-16
42.2 Tuning Your View Objects for Best Performance 42-18
42.2.1 Use Bind Variables for Parameterized Queries 42-18
42.2.1.1 Use Bind Variables to Avoid Re-parsing of Queries 42-18
42.2.1.2 Use Bind Variables to Prevent SQL-Injection Attacks 42-18
42.2.2 Consider Using Entity-Based View Objects for Read-Only Data 42-19
42.2.3 Use SQL Tracing to Identify Ill-Performing Queries 42-21
42.2.4 Consider the Appropriate Tuning Settings for Every View Object 42-22
42.2.4.1 Set the Database Retrieval Options Appropriately 42-22
42.2.4.2 Consider Whether Fetching One Row at a Time is Appropriate 42-23
42.2.4.3 Specify a Query Optimizer Hint if Necessary 42-23
42.2.5 Using Care When Creating View Objects at Runtime 42-24
42.2.6 Use Forward Only Mode to Avoid Caching View Rows 42-24
42.3 Generating Custom Java Classes for a View Object 42-24
42.3.1 How To Generate Custom Classes 42-25
42.3.1.1 Generating Bind Variable Accessors 42-25
42.3.1.2 Generating View Row Attribute Accessors 42-26
42.3.1.3 Exposing View Row Accessors to Clients 42-27
42.3.1.4 Configuring Default Java Generation Preferences 42-28
42.3.2 What Happens When You Generate Custom Classes 42-28
42.3.2.1 Seeing and Navigating to Custom Java Files 42-28
42.3.3 What You May Need to Know About Custom Classes 42-29
42.3.3.1 About the Framework Base Classes for a View Object 42-29
42.3.3.2 You Can Safely Add Code to the Custom Component File 42-29
42.3.3.3 Attribute Indexes and InvokeAccessor Generated Code 42-29
42.4 Working Programmatically with Multiple Named View Criteria 42-31
42.4.1 Applying One or More Named View Criteria 42-31
42.4.2 Removing All Applied Named View Criteria 42-32
42.4.3 Using the Named Criteria at Runtime 42-33

42.5 Performing In-Memory Sorting and Filtering of Row Sets ... 42-34
42.5.1 Understanding the View Object's SQL Mode ... 42-34
42.5.2 Sorting View Object Rows In Memory ... 42-35
42.5.2.1 Combining setSortBy and setQueryMode for In-Memory Sorting ... 42-35
42.5.2.2 Extensibility Points for In-Memory Sorting ... 42-37
42.5.3 Performing In-Memory Filtering with View Criteria ... 42-37
42.5.4 Performing In-Memory Filtering with RowMatch ... 42-40
42.5.4.1 Applying a RowMatch to a View Object ... 42-40
42.5.4.2 Using RowMatch to Test an Individual Row ... 42-42
42.5.4.3 How a RowMatch Affects Rows Fetched from the Database ... 42-42
42.6 Using View Objects to Work with Multiple Row Types ... 42-42
42.6.1 Working with Polymorphic Entity Usages ... 42-42
42.6.2 How to Create a View Object with a Polymorphic Entity Usage ... 42-43
42.6.3 What Happens When You Create a View Object with a Polymorphic Entity Usage ... 42-46
42.6.4 What You May Need to Know About Entity Usages ... 42-47
42.6.4.1 Your Query Must Limit Rows to Expected Entity Subtypes ... 42-47
42.6.4.2 Exposing Selected Entity Methods in View Rows Using Delegation ... 42-47
42.6.4.3 Creating New Rows With the Desired Entity Subtype ... 42-48
42.6.5 Working with Polymorphic View Rows ... 42-49
42.6.6 How to Create a View Object with Polymorphic View Rows ... 42-50
42.6.7 What You May Need to Know About Polymorphic View Rows ... 42-51
42.6.7.1 Selecting Subtype-Specific Attributes in Extended View Objects ... 42-51
42.6.7.2 Delegating to Subtype-Specific Methods After Overriding the Entity Usage ... 42-52
42.6.7.3 Working with Different View Row Interface Types in Client Code ... 42-52
42.6.7.4 View Row Polymorphism and Polymorphic Entity Usage are Orthogonal ... 42-53
42.7 Reading and Writing XML ... 42-54
42.7.1 How to Produce XML for Queried Data ... 42-54
42.7.2 What Happens When You Produce XML ... 42-55
42.7.3 What You May Need to Know About Reading and Writing XML ... 42-57
42.7.3.1 Controlling XML Element Names ... 42-57
42.7.3.2 Controlling Element Suppression for Null-Valued Attributes ... 42-58
42.7.3.3 Printing or Searching the Generated XML Using XPath ... 42-58
42.7.3.4 Using the Attribute Map For Fine Control Over Generated XML ... 42-58
42.7.3.5 Use the Attribute Map Approach with Bi-Directional View Links ... 42-60
42.7.3.6 Transforming Generated XML Using an XSLT Stylesheet ... 42-60
42.7.3.7 Generating XML for a Single Row ... 42-62
42.7.4 How to Consume XML Documents to Apply Changes ... 42-62
42.7.5 What Happens When You Consume XML Documents ... 42-62
42.7.5.1 How ViewObject.readXML() Processes an XML Document ... 42-62
42.7.5.2 Using readXML() to Processes XML for a Single Row ... 42-62
42.8 Using Programmatic View Objects for Alternative Data Sources ... 42-66
42.8.1 How to Create a Read-Only Programmatic View Object ... 42-66
42.8.2 How to Create an Entity-Based Programmatic View Object ... 42-67
42.8.3 Key Framework Methods to Override for Programmatic View Objects ... 42-67
42.8.4 How to Create a View Object on a REF CURSOR ... 42-68
42.8.4.1 The Overridden create() Method ... 42-69

42.8.4.2 The Overridden executeQueryForCollection() Method 42-69
42.8.4.3 The Overridden createRowFromResultSet() Method 42-70
42.8.4.4 The Overridden hasNextForCollectionMethod() 42-70
42.8.4.5 The Overridden releaseUserDataForCollection() Method 42-71
42.8.4.6 The Overridden getQueryHitCount() Method 42-71
42.9 Creating a View Object with Multiple Updatable Entities 42-72
42.9.1 How to Programmatically Create New Rows With Multiple Updatable Entity Usages .. 42-73
42.9.2 What Happens at Runtime: View Row Creation 42-75
42.10 Programmatically Creating View Definitions and View Objects 42-76
42.11 Declaratively Preventing Insert, Update, and Delete 42-78

43 Application State Management

43.1 Understanding Why State Management is Necessary 43-1
43.1.1 Examples of Multi-Step Tasks 43-1
43.1.2 Stateless HTTP Protocol Complicates Stateful Applications 43-2
43.1.3 How Cookies Are Used to Track a User Session 43-2
43.1.4 Performance and Reliability Impact of Using HttpSession 43-3
43.2 Introduction to Fusion Web Application State Management 43-5
43.2.1 Basic Architecture of the Save for Later Facility 43-5
43.2.2 Basic Architecture of the Application Module State Management Facility 43-5
43.2.2.1 Understanding When Passivation and Activation Occurs 43-6
43.2.2.2 How Passivation Changes When Optional Failover Mode is Enabled 43-8
43.2.2.3 About State Management Release Levels 43-9
43.2.2.4 State Management and Subclassed Entity Objects 43-11
43.3 Using Save For Later 43-11
43.4 Setting the Application Module Release Level at Runtime 43-12
43.4.1 How to Set Unmanaged Level 43-12
43.4.2 How to Set Reserved Level 43-12
43.4.3 How to Set Managed Level 43-12
43.4.4 How to Set Release Level in a JSF Backing Bean 43-12
43.4.5 How to Set Release Level in an ADF PagePhaseListener 43-13
43.4.6 How to Set Release Level in an ADF PageController 43-13
43.4.7 How to Set Release Level in a Custom ADF PageLifecycle 43-14
43.5 What Model State Is Saved and When It Is Cleaned Up 43-15
43.5.1 State Information Saved During Passivation 43-15
43.5.2 Where the Model State Is Saved 43-16
43.5.2.1 How Database-Backed Passivation Works 43-16
43.5.2.2 Controlling the Schema Where the State Management Table Resides 43-16
43.5.2.3 Configuring the Type of Passivation Store 43-17
43.5.3 Cleaning Up the Model State 43-17
43.5.3.1 Previous Snapshot Removed When Next One Taken 43-17
43.5.3.2 Passivation Snapshot Removed on Unmanaged Release 43-18
43.5.3.3 Passivation Snapshot Retained in Failover Mode 43-18
43.5.4 Cleaning Up Temporary Storage Tables 43-18
43.6 Timing Out the HttpSession 43-19
43.6.1 How to Configure the Implicit Timeout Due to User Inactivity 43-19

43.6.2 How to Code an Explicit HttpSession Timeout 43-19
43.7 Managing Custom User-Specific Information 43-20
43.7.1 How to Passivate Custom User-Specific Information 43-20
43.7.1.1 What Happens When You Passivate Custom Information 43-22
43.8 Managing the State of View Objects 43-22
43.8.1 How to Manage the State of View Objects 43-22
43.8.2 What You May Need to Know About Passivating View Objects 43-23
43.8.3 How to Manage the State of Transient View Objects and Attributes 43-24
43.8.4 What You May Need to Know About Passivating Transient View Objects 43-25
43.8.5 How to Use Transient View Objects to Store Session-level Global Variables 43-25
43.9 Using State Management for Middle-Tier Savepoints 43-26
43.9.1 How to Use State Management for Savepoints 43-26
43.10 Testing to Ensure Your Application Module is Activation-Safe 43-27
43.10.1 Understanding the jbo.ampool.doampooling Configuration Parameter 43-27
43.10.2 Disabling Application Module Pooling to Test Activation 43-27
43.11 Keeping Pending Changes in the Middle Tier 43-28
43.11.1 How to Confirm That Applications Use Optimistic Locking 43-29
43.11.2 How to Avoid Clashes Using the postChanges() Method 43-29
43.11.3 How to Use the Reserved Level For Pending Database States 43-29

44 Tuning Application Module Pools and Connection Pools

44.1 About Application Module Pooling 44-1
44.1.1 Types of Pools Created When Running the Fusion Web Application 44-2
44.1.1.1 Application Module Pools 44-2
44.1.1.2 Database Connection Pools 44-2
44.1.2 Understanding Application Module and Connection Pools 44-3
44.1.2.1 Single Oracle Application Server Instance, Single Oracle WebLogic Server Instance, Single JVM 44-3
44.1.2.2 Multiple Oracle Application Server Instances, Single Oracle WebLogic Server Instance, Multiple JVMs 44-3
44.2 Setting Pool Configuration Parameters 44-4
44.2.1 How to Set Configuration Properties Declaratively 44-5
44.2.2 What Happens When You Set Configuration Properties Declaratively 44-5
44.2.3 How to Set Configuration Properties as System Parameters 44-6
44.2.4 How to Programmatically Set Configuration Properties 44-7
44.2.5 What You May Need to Know About Configuration Property Scopes 44-8
44.2.6 What You May Need to Know About How Database and Application Module Pools Cooperate 44-9
44.2.7 What You May Need to Know About Application Module Pool Parameters 44-11
44.2.7.1 Pool Behavior Parameters 44-11
44.2.7.2 Pool Sizing Parameters 44-14
44.2.7.3 Pool Cleanup Parameters 44-15
44.2.8 What You May Need to Know About Data Source Configuration 44-17
44.2.9 What You May Need to Know About Database Connection Pool Parameters 44-18
44.3 Initializing Database State and Pooling Considerations 44-20
44.3.1 How to Set Database State Per User 44-20

44.3.2 What You May Need to Know About Database User State and jbo.doconnectionpooling = true 44-21

45 Using the Active Data Service

45.1 About the Active Data Service ... 45-1
45.1.1 Active Data Service Use Cases and Examples ... 45-1
45.1.2 Limitations of the Active Data Service Framework ... 45-2
45.1.3 Active Data Service Framework ... 45-2
45.1.4 Data Transport Modes ... 45-4
45.2 Configuring the Active Data Service ... 45-5
45.2.1 How to Configure the Active Data Service ... 45-6
45.2.2 What You May Need to Know About Transport Modes ... 45-7
45.3 Configuring Components to Use the Active Data Service ... 45-9
45.3.1 How to Configure Components to Use the Active Data Service Without the Active Data Proxy 45-10
45.3.2 How to Configure Components to Use the Active Data Service with the Active Data Proxy 45-10
45.3.3 What You May Need to Know About Displaying Active Data in ADF Trees ... 45-11
45.3.4 What Happens at Runtime: How Components Render When Bound to Active Data ... 45-11
45.3.5 What You May Need to Know About ADS and Google Chrome ... 45-11
45.4 Using the Active Data Proxy ... 45-12
45.4.1 What You May Need to Know About Read Consistency ... 45-17
45.5 Using the Active Data with a Scalar Model ... 45-17

Part VIII Appendices

A Oracle ADF XML Files

A.1 Introduction to the ADF Metadata Files ... A-1
A.2 ADF File Overview Diagram ... A-2
A.2.1 Oracle ADF Data Control Files ... A-2
A.2.2 Oracle ADF Data Binding Files ... A-3
A.2.3 Web Configuration Files ... A-3
A.3 ADF File Syntax Diagram ... A-4
A.4 adfm.xml ... A-5
A.5 *modelProjectName*.jpx ... A-6
A.6 bc4j.xcfg ... A-8
A.7 DataBindings.cpx ... A-9
A.7.1 DataBindings.cpx Syntax ... A-9
A.7.2 DataBindings.cpx Sample ... A-11
A.8 *pageName*PageDef.xml ... A-12
A.8.1 PageDef.xml Syntax ... A-12
A.9 adfc-config.xml ... A-23
A.10 task-flow-definition.xml ... A-25
A.11 adf-config.xml ... A-28
A.12 adf-settings.xml ... A-31
A.13 web.xml ... A-32

A.14 logging.xml A-34

B Oracle ADF Binding Properties

C ADF Security Permission Grants

D Most Commonly Used ADF Business Components Methods

D.1 Methods for Creating Your Own Layer of Framework Base Classes D-1
D.2 Methods Used in the Client Tier D-2
D.2.1 ApplicationModule Interface D-2
D.2.2 Transaction Interface D-3
D.2.3 ViewObject Interface D-4
D.2.4 RowSet Interface D-6
D.2.5 RowSetIterator Interface D-7
D.2.6 Row Interface D-8
D.2.7 StructureDef Interface D-9
D.2.8 AttributeDef Interface D-9
D.2.9 AttributeHints Interface D-10
D.3 Methods Used in the Business Service Tier D-11
D.3.1 Controlling Custom Java Files for Your Components D-11
D.3.2 ApplicationModuleImpl Class D-12
D.3.2.1 Methods You Typically Call on ApplicationModuleImpl D-12
D.3.2.2 Methods You Typically Write in Your Custom ApplicationModuleImpl Subclass ... D-12
D.3.2.3 Methods You Typically Override in Your Custom ApplicationModuleImpl Subclass D-13
D.3.3 DBTransactionImpl2 Class D-14
D.3.3.1 Methods You Typically Call on DBTransaction D-14
D.3.3.2 Methods You Typically Override in Your Custom DBTransactionImpl2 Subclass ... D-15
D.3.4 EntityImpl Class D-16
D.3.4.1 Methods You Typically Call on EntityImpl D-16
D.3.4.2 Methods You Typically Write in Your Custom EntityImpl Subclass D-17
D.3.4.3 Methods You Typically Override in Your Custom EntityImpl Subclass D-17
D.3.5 EntityDefImpl Class D-18
D.3.5.1 Methods You Typically Call on EntityDefImpl D-19
D.3.5.2 Methods You Typically Write in Your Custom EntityDefImpl Class D-19
D.3.5.3 Methods You Typically Override in Your Custom EntityDefImpl D-20
D.3.6 ViewObjectImpl Class D-20
D.3.6.1 Methods You Typically Call on ViewObjectImpl D-20
D.3.6.2 Methods You Typically Write in Your Custom ViewObjectImpl Subclass D-22
D.3.6.3 Methods You Typically Override in Your Custom ViewObjectImpl Subclass D-22
D.3.7 ViewRowImpl Class D-23
D.3.7.1 Methods You Typically Call on ViewRowImpl D-23
D.3.7.2 Methods You Typically Write in Your Custom ViewRowImpl Class D-24
D.3.7.3 Methods You Typically Override in Your Custom ViewRowImpl Subclass ... D-24

E ADF Business Components Java EE Design Pattern Catalog

F ADF Equivalents of Common Oracle Forms Triggers

F.1 Validation and Defaulting (Business Logic) F-1
F.2 Query Processing F-2
F.3 Database Connection F-3
F.4 Transaction "Post" Processing (Record Cache) F-3
F.5 Error Handling F-4

G Performing Common Oracle Forms Tasks in Oracle ADF

G.1 Performing Tasks Related to Data G-1
G.1.1 How to Retrieve Lookup Display Values for Foreign Keys G-1
G.1.2 How to Get the Sysdate from the Database G-2
G.1.3 How to Implement an Isolation Mode That Is Not Read Consistent G-2
G.1.4 How to Implement Calculated Fields G-2
G.1.5 How to Implement Mirrored Items G-3
G.1.6 How to Use Database Columns of Type CLOB or BLOB G-3
G.2 Performing Tasks Related to the User Interface G-3
G.2.1 How to Lay Out a Page G-3
G.2.2 How to Stack Canvases G-4
G.2.3 How to Implement a Master-Detail Screen G-4
G.2.4 How to Implement an Enter Query Screen G-4
G.2.5 How to Implement an Updatable Multi-Record Table G-4
G.2.6 How to Create a Popup List of Values G-4
G.2.7 How to Implement a Dropdown List as a List of Values G-5
G.2.8 How to Implement a Dropdown List with Values from Another Table G-5
G.2.9 How to Implement Immediate Locking G-5
G.2.10 How to Throw an Error When a Record Is Locked G-6

H Data Controls in Fusion Web Applications

H.1 Introduction to Data Controls H-1
H.2 Data Control Feature Implementation Comparison H-2
H.3 Data Control Objects H-3

Glossary

Preface

Welcome to the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*.

Audience

This document is intended for enterprise developers who need to create and deploy database-centric Java EE applications with a service-oriented architecture using the Oracle Application Development Framework (Oracle ADF). This guide explains how to build Fusion web applications using ADF Business Components, ADF Controller, ADF Faces, and JavaServer Faces.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

For more information, see the following documents:

Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework

Oracle Fusion Middleware User Guide for Oracle JDeveloper

Oracle Fusion Middleware Skin Editor User's Guide for Oracle Application Development Framework

Oracle Fusion Middleware Java EE Developer's Guide for Oracle Application Development Framework

Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework

Oracle Fusion Middleware Desktop Integration Developer's Guide for Oracle Application Development Framework

Oracle Fusion Middleware Mobile Browser Developer's Guide for Oracle Application Development Framework

Oracle Fusion Middleware Performance and Tuning Guide

Oracle Fusion Middleware High Availability Guide

Oracle Fusion Middleware Installation Guide for Oracle JDeveloper

Oracle Fusion Middleware Installation Guide for Oracle Application Development Framework Skin Editor

Oracle JDeveloper Online Help

Oracle JDeveloper Release Notes, included with your JDeveloper installation, and on Oracle Technology Network

Oracle Fusion Middleware Java API Reference for Oracle ADF Model

Oracle Fusion Middleware Java API Reference for Oracle ADF Controller

Oracle Fusion Middleware Java API Reference for Oracle ADF Lifecycle

Oracle Fusion Middleware Java API Reference for Oracle ADF Faces

Oracle Fusion Middleware Javascript API Reference for Oracle ADF Faces

Oracle Fusion Middleware Java API Reference for Oracle ADF Data Visualization Components

Oracle Fusion Middleware Java API Reference for Oracle ADF Share

Oracle Fusion Middleware Java API Reference for Oracle Business Component Browser

Oracle Fusion Middleware Generic Domains Java API Reference for Oracle ADF Business Components

Oracle Fusion Middleware interMedia Domains Java API Reference for Oracle ADF Business Components

Oracle Fusion Middleware Java API Reference for Oracle Metadata Service (MDS)

Oracle Fusion Middleware Tag Reference for Oracle ADF Faces

Oracle Fusion Middleware Tag Reference for Oracle ADF Faces Skin Selectors

Oracle Fusion Middleware Data Visualization Tools Tag Reference for Oracle ADF Faces

Oracle Fusion Middleware Data Visualization Tools Tag Reference for Oracle ADF Skin Selectors

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements (for example, menus and menu items, buttons, tabs, dialog controls), including options that you select.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates language and syntax elements, directory and file names, URLs, text that appears on the screen, or text that you enter.

What's New in This Guide in Release 11.1.2.1.0

For Release 11.1.2.1.0, this guide has been updated in several ways. The following table lists the sections that have been added or changed.

For changes made to Oracle JDeveloper and Oracle Application Development Framework (Oracle ADF) for this release, see the What's New page on the Oracle Technology Network at `http://www.oracle.com/technetwork/developer-tools/jdev/documentation/index.html`.

Sections	Changes Made
Chapter 5 Defining SQL Queries Using View Objects	
Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object"	Revised section to describe a best practice for specifying a key attribute when creating read-only view objects.
Chapter 10 Sharing Application Module View Instances	
Section 10.3.1, "How to Create a Base View Object Definition for a Lookup Table"	Revised section to include best practice of selecting a key attribute for read-only view objects that you create for lookup tables.
Chapter 12 Extending Business Component Functionality	
Section 12.8.3, "How to Display Customize Error Messages as Nested Exceptions"	Created new section to describe how to override `skipException()` method in the custom error handler.
Chapter 21 Using Task Flows as Regions	
Section 21.5.1, "How to Configure the Refresh of an ADF Region"	Revised section to clarify that a parent component (for example, a `popup` or `panelTabbed` component) with a `childCreation` attribute effects the refresh behavior of an ADF region.
Chapter 22 Creating Complex Task Flows	

Sections	Changes Made
Section 22.5.3, "What You May Need to Know About the Database Table for Save Points"	Added section to describe the `adfc_cleanup_save_point_table.sql` and `adfc_create_save_point_table.sql` SQL scripts that can be used to manage save points in the `ORADFCSAVPT` database table.
Chapter 39 Customizing Applications with MDS	
Section 39.3.12.2, "Editing Resource Bundles in Customized Applications"	Revised section to describe the need for tagging the `bundleId` element in `adf-config.xml` with `override="true"` to make a resource bundle overrideable.
Chapter 41 Deploying Fusion Web Applications	
Section 41.3.7, "What You May Need to Know About JDBC Data Source for Oracle WebLogic Server"	Revised section to describe limitation about running an Oracle ADF application with a data source created for an XA database.
Chapter 42 Advanced View Object Techniques	
Section 42.6.2, "How to Create a View Object with a Polymorphic Entity Usage"	Revised section to update the procedure for creating a view object based on a polymorphic entity usage.
Section 42.10, "Programmatically Creating View Definitions and View Objects"	Added section to describe the use of the `oracle.jbo.server.ViewDefImpl` API.
Chapter 44 Tuning Application Module Pools and Connection Pools	
Section 44.2.7, "What You May Need to Know About Application Module Pool Parameters"	Revised section to add best practice when enabling failover passivation that involves configuring a pool parameter for Oracle WebLogic Server.

Part I

Getting Started with Fusion Web Applications

Part I contains the following chapters:

- Chapter 1, "Introduction to Building Fusion Web Applications with Oracle ADF"
- Chapter 2, "Introduction to the ADF Sample Application"

1

Introduction to Building Fusion Web Applications with Oracle ADF

This chapter describes the architecture and key functionality of the Oracle Application Development Framework (Oracle ADF) when used to build a Fusion web application that uses ADF Business Components, ADF Model, ADF Controller, and ADF Faces rich client, along with high-level development practices.

This chapter includes the following sections:

- Section 1.1, "Introduction to Oracle ADF"
- Section 1.2, "Oracle ADF Architecture"
- Section 1.3, "Overview of Building an Application with Oracle ADF"
- Section 1.4, "Working Productively in Teams"
- Section 1.5, "Other Resources for Learning Oracle ADF"
- Section 1.6, "Generation of Complete Web Tier Using Oracle JHeadstart"

1.1 Introduction to Oracle ADF

The Oracle Application Development Framework (Oracle ADF) is an end-to-end application framework that builds on Java Platform, Enterprise Edition (Java EE) standards and open-source technologies. You can use Oracle ADF to implement enterprise solutions that search, display, create, modify, and validate data using web, wireless, desktop, or web services interfaces. Because of its declarative nature, Oracle ADF simplifies and accelerates development by allowing users to focus on the logic of application creation rather than coding details. Used in tandem, Oracle JDeveloper and Oracle ADF give you an environment that covers the full development lifecycle from design to deployment, with drag and drop data binding, visual UI design, and team development features built in.

As a companion to this guide, you can download and view the Fusion Order demo application, which helps to illustrate the concepts and procedures in this guide (and other Oracle Fusion Middleware developer guides). The StoreFront module of this application is built using the Fusion web application technology stack, which includes ADF Business Components, ADF Model, ADF Controller, and JavaServer Faces pages with ADF Faces rich client components. Screenshots and code samples from this module are used throughout this guide to provide you with real-world examples of using the Oracle ADF technologies in an application that uses the Fusion web technology stack. For more information about downloading and using the StoreFront module of the Fusion Order Demo application, see Chapter 2, "Introduction to the ADF Sample Application."

1.2 Oracle ADF Architecture

In line with community best practices, applications you build using the Fusion web technology stack achieve a clean separation of business logic, page navigation, and user interface by adhering to a model-view-controller architecture. As shown in Figure 1–1, in an MVC architecture:

- The model layer represents the data values related to the current page
- The view layer contains the UI pages used to view or modify that data
- The controller layer processes user input and determines page navigation
- The business service layer handles data access and encapsulates business logic

Figure 1–1 MVC Architecture Cleanly Separates UI, Business Logic and Page Navigation

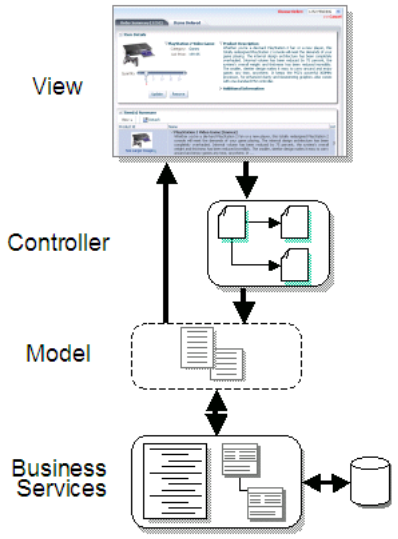
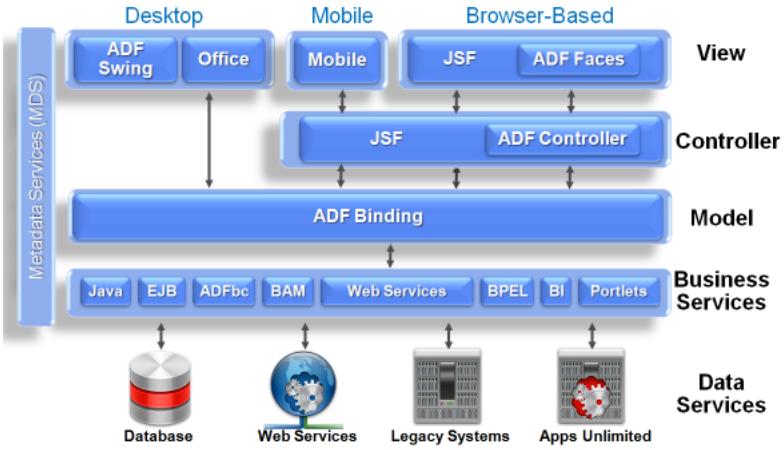


Figure 1–2 illustrates where each ADF module fits in the Fusion web application architecture. The core module in the framework is ADF Model, a data binding facility. The ADF Model layer enables a unified approach to bind any user interface to any business service, without the need to write code. The other modules that make up a Fusion web application technology stack are:

- ADF Business Components, which simplifies building business services.
- ADF Faces rich client, which offers a rich library of ajax-enabled UI components for web applications built with JavaServer Faces (JSF).
- ADF Controller, which integrates JSF with ADF Model. ADF Controller extends the standard JSF controller by providing additional functionality, such as reusable task flows that pass control not only between JSF pages, but also between other activities, for instance method calls or other task flows.

Note: In addition to ADF Faces, Oracle ADF also supports using the Swing, JSP, and standard JSF view technologies. For more information about these technologies, see the *Oracle Fusion Middleware User Guide for Oracle JDeveloper*. Oracle ADF also provides support for using Microsoft Excel as a view layer for your application. For more information, see the *Oracle Fusion Middleware Desktop Integration Developer's Guide for Oracle Application Development Framework*.

Figure 1–2 Simple Oracle ADF Architecture



1.2.1 ADF Business Components

When building service-oriented Java EE applications, you implement your core business logic as one or more business services. These backend services provide clients with a way to query, insert, update, and delete business data as required while enforcing appropriate business rules. ADF Business Components are prebuilt application objects that accelerate the job of delivering and maintaining high-performance, richly functional, database-centric services. They provide you with a ready-to-use implementation of Java EE design patterns and best practices.

As illustrated in Figure 1–3, Oracle ADF provides the following key components to simplify building database-centric business services:

- Entity object

  An entity object represents a row in a database table and simplifies modifying its data by handling all data manipulation language (DML) operations for you. It can encapsulate business logic to ensure that your business rules are consistently enforced. You associate an entity object with others to reflect relationships in the underlying database schema to create a layer of business domain objects to reuse in multiple applications.

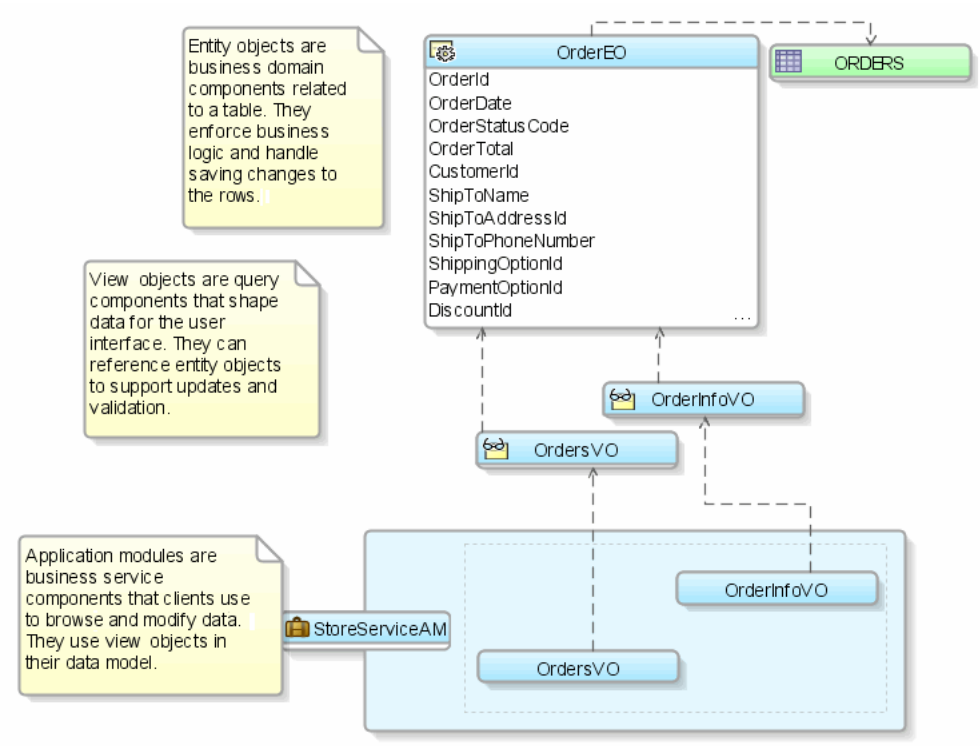
- View object

  A view object represents a SQL query and simplifies working with its results. You use the SQL language to join, filter, sort, and aggregate data into the shape required by the end-user task being represented in the user interface. This includes the ability to link a view object with other view objects to create master-detail hierarchies of any complexity. When end users modify data in the user interface, your view objects collaborate with entity objects to consistently validate and save the changes.

- Application module

  An application module is the transactional component that UI clients use to work with application data. It defines an updateable data model along with top-level procedures and functions (called service methods) related to a logical unit of work related to an end-user task.

Figure 1–3 ADF Business Components Simplify Data Access and Validation



Tip: If you have previously worked with Oracle Forms, note that this combined functionality is the same set of data-centric features provided by the form, data blocks, record manager, and form-level procedures or functions. The key difference in Oracle ADF is that the user interface is cleanly separated from data access and validation functionality. For more information, see Appendix G, "Performing Common Oracle Forms Tasks in Oracle ADF."

1.2.2 ADF Model Layer

The ADF model abstracts the business service implementation, providing a single programming interface for different types of services. Data controls provide this interface by using standard metadata interfaces to describe the service's operations and data collections, including information about the properties, methods, and types involved. In JDeveloper, the functionality and attributes exposed by a business service are indicated by icons in the Data Controls panel. You can drag and drop onto a page to create UI components. JDeveloper automatically creates the bindings from the page and the UI components to the services. At runtime, the ADF Model layer reads the information describing your data controls and data bindings from appropriate XML files and implements the two-way connection between your user interface and your business service.

Oracle ADF provides out-of-the-box data control implementations for the most common business service technologies. Using JDeveloper and Oracle ADF together provides you with a drag and drop data binding experience as you build your user interfaces. Along with support for ADF application modules, ADF Model also provides support for the following service technologies:

- Enterprise JavaBeans (EJB) session beans and JPA entities
- JavaBeans
- Web services
- XML
- CSV files

1.2.3 ADF Controller

In the controller layer, where handling page flow of your web applications is a key concern, ADF Controller provides an enhanced navigation and state management model on top of JSF. JDeveloper allows you to declaratively create task flows where you can pass application control between different types of activities, such as pages, methods on managed beans, case statements, or calls to other task flows.

These task flows can be reused, and can also be nested, both within themselves and within pages. Task flows nested in pages become regions that contain their own set of navigable pages, allowing users to view a number of different pages and functionality without leaving the main page.

1.2.4 ADF Faces Rich Client

ADF Faces rich client (ADF Faces for short), is a set of standard JSF components that include built-in AJAX functionality. AJAX is a combination of asynchronous JavaScript, dynamic HTML (DHTML), XML, and `XmlHttpRequest` communication channels. This combination allows requests to be made to the server without fully rerendering the page. While AJAX allows rich client-like applications to use standard internet technologies, JSF provides server-side control, which reduces the dependency on an abundance of JavaScript often found in typical AJAX applications. To achieve these front-end capabilities, ADF Faces components use a rendering kit that handles displaying the component and also provides the JavaScript objects needed for the rich functionality. This built-in support enables you to build rich applications without needing to code JavaScript, or needing extensive knowledge of the individual technologies on the front or back end.

ADF Faces provides over 150 rich components, including hierarchical data tables, tree menus, in-page dialogs, accordions, dividers, and sortable tables. ADF Faces also provides ADF Data Visualization components, which are Flash- and PNG-enabled components capable of rendering dynamic charts, graphs, gauges, and other graphics that can provide a real-time view of underlying data. Each component also supports customization and skinning, along with internationalization and accessibility.

ADF Faces can also be used in an application that uses the Facelets library. Facelets is a JSF-centric XML view definition technology that provides an alternative to using the JSP engine. For more information about ADF Faces, including the architecture and detailed information about each of the components, see the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Along with ADF Faces, Oracle ADF also supports the following view technologies:

- Apache MyFaces Trinidad: This is the open source code donation from Oracle to the Apache Software Foundation. ADF Faces components are based on these Trinidad components.
- Java Swing and ADF Swing: ADF Swing is the development environment for building Java Swing applications that use the ADF Model layer.
- ADF Mobile: This is a framework that extends Oracle ADF to support mobile users, allowing the development of mobile browser-based or on-device applications that run on mobile devices.
- Microsoft Excel: You can create spreadsheets that are bound to data using the same binding principals as do other view technologies. For more information, see the *Oracle Fusion Middleware Desktop Integration Developer's Guide for Oracle Application Development Framework*.

1.3 Overview of Building an Application with Oracle ADF

Oracle ADF emphasizes the use of the declarative programming paradigm throughout the development process to allow users to focus on the logic of application creation without having to get into implementation details. Using JDeveloper with Oracle ADF, you benefit from a high-productivity environment that automatically manages your application's declarative metadata for data access, validation, page control and navigation, user interface design, and data binding.

At a high level, the development process for a Fusion web application usually involves the following:

- Creating an application workspace: Using a wizard, JDeveloper automatically adds the libraries and configuration needed for the technologies you select, and structures your application into projects with packages and directories.
- Modeling the database objects: You can create an offline replica of any database, and use JDeveloper editors and diagrammers to edit definitions and update schemas.
- Creating use cases: Using the UML modeler, you can create use cases for your application.
- Designing application control and navigation: You use diagrammers to visually determine the flow of application control and navigation. JDeveloper creates the underlying XML for you.
- Identifying shared resources: You use a resource library that allows you to view and use imported libraries by simply dragging and dropping them into your application.
- Creating business components to access data: From your database tables, you create entity objects using wizards or dialogs. From those entity objects, you create the view objects used by the pages in your application. You can implement validation rules and other types of business logic using editors.
- Implementing the user interface with JSF: The Data Controls panel in JDeveloper contains a representation of the view objects in your application. Creating a user interface is as simple as dragging an object onto a page and selecting the UI component you want to display the underlying data. For UI components that are not databound, you use the Component Palette to drag and drop components. JDeveloper creates all the page code for you.

- Binding UI components to data using the ADF Model layer: When you drag an object from the Data Controls panel, JDeveloper automatically creates the bindings between the page and the data model.
- Incorporating validation and error handling: Once your application is created, you use editors to add additional validation and to define error handling.
- Securing the application: You use editors to create roles and populate these with test users. You then use a flat file editor to define security policies for these roles and assign them to specific resources in your application.
- Testing and debugging: JDeveloper includes an integrated application server that allows you to fully test your application without needing to package it up and deploy it. JDeveloper also includes the ADF Declarative Debugger, a tool that allows you to set breakpoints and examine the data.
- Deploying the application: You use wizards and editors to create and edit deployment descriptors, JAR files, and application server connections.

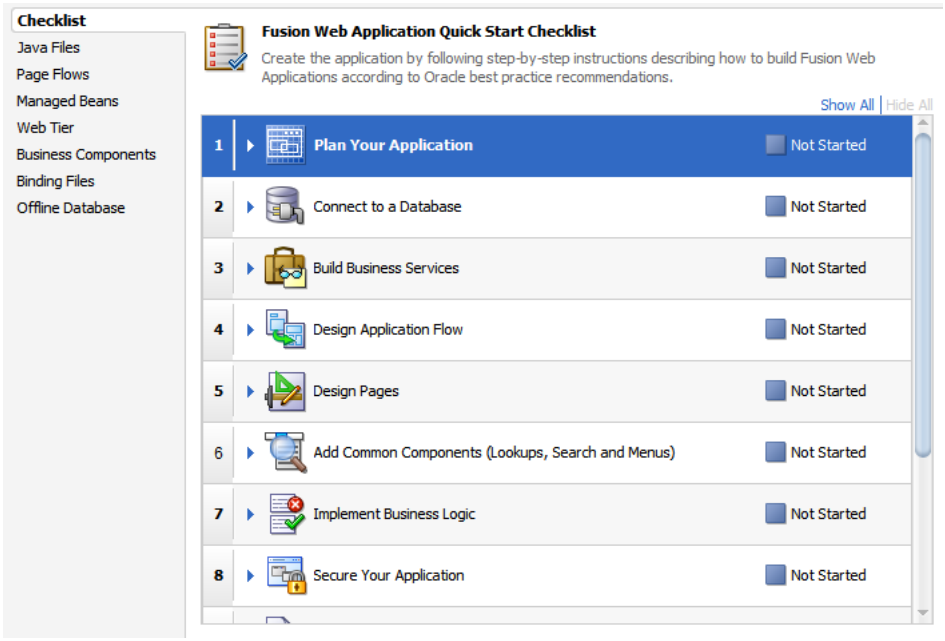
1.3.1 Creating an Application Workspace

The first step in building a new application is to assign it a name and to specify the directory where its source files will be saved. When you create an application using the application templates provided by JDeveloper, it organizes your workspace into projects and creates and organizes many of the configuration files required by the type of application you are creating.

One of these templates is the Fusion Web Application (ADF) template, which provides the correctly configured set of projects you need to create a web application that uses ADF Faces for the view, ADF Controller for the page flow, and ADF Business Components for business services. When you create an application workspace using this template, JDeveloper automatically creates the JSF and ADF configuration files needed for the application.

One part of the application overview is the Fusion Web Application Quick Start Checklist. This checklist provides you with the basic steps for creating a Fusion web application. Included are links to pertinent documentation, prerequisites, and the ability to keep track of status for each step in the checklist, as shown in Figure 1–4.

Figure 1–4 Fusion Web Application Quick Start Checklist



JDeveloper also creates a project named `Model` that will contain all the source files related to the business services in your application, and a project named `ViewController` that will contain all the source files for your ADF Faces view layer, including files for the controller.

JDeveloper adds the following libraries to the data model project:

- ADF Model Runtime
- BC4J Oracle Domains
- BC4J Runtime
- BC4J Security
- MDS Runtime
- MDS Runtime Dependencies
- Oracle JDBC

JDeveloper also adds the following libraries to the view project:

- JSP Runtime
- JSF 2.0
- JSTL 1.2
- ADF Page Flow Runtime
- ADF Controller Runtime
- ADF Controller Schema
- ADF Faces Runtime 11
- ADF Common Runtime
- ADF Web Runtime
- MDS Runtime

- MDS Runtime Dependencies
- Commons Beautils 1.6
- Commons Logging 1.0.4
- Commons Collections 3.1
- ADF DVT Faces Runtime
- ADF DVT Faces Databinding Runtime
- ADF DVT Faces Databinding MDS Runtime

Once you add a JSF page, JDeveloper adds the Oracle JEWT library.

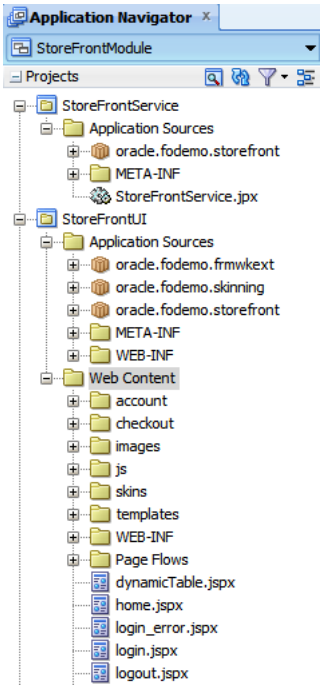
Once the projects are created for you, you can rename them as you need. You can then use JDeveloper to create additional projects, and add the packages and files needed for your application.

Note: If you plan to reuse artifacts in your application (for example, task flows), then you should follow the naming guidelines presented in Chapter 38, "Reusing Application Components" in order to prevent naming conflicts.

Tip: You can edit the default values used in application templates, as well as create your own templates. To do so, choose **Application > Manage Templates**.

Figure 1–5 shows the different projects, packages, directories, and files for the `StoreFrontModule` application, as displayed in the Application Navigator.

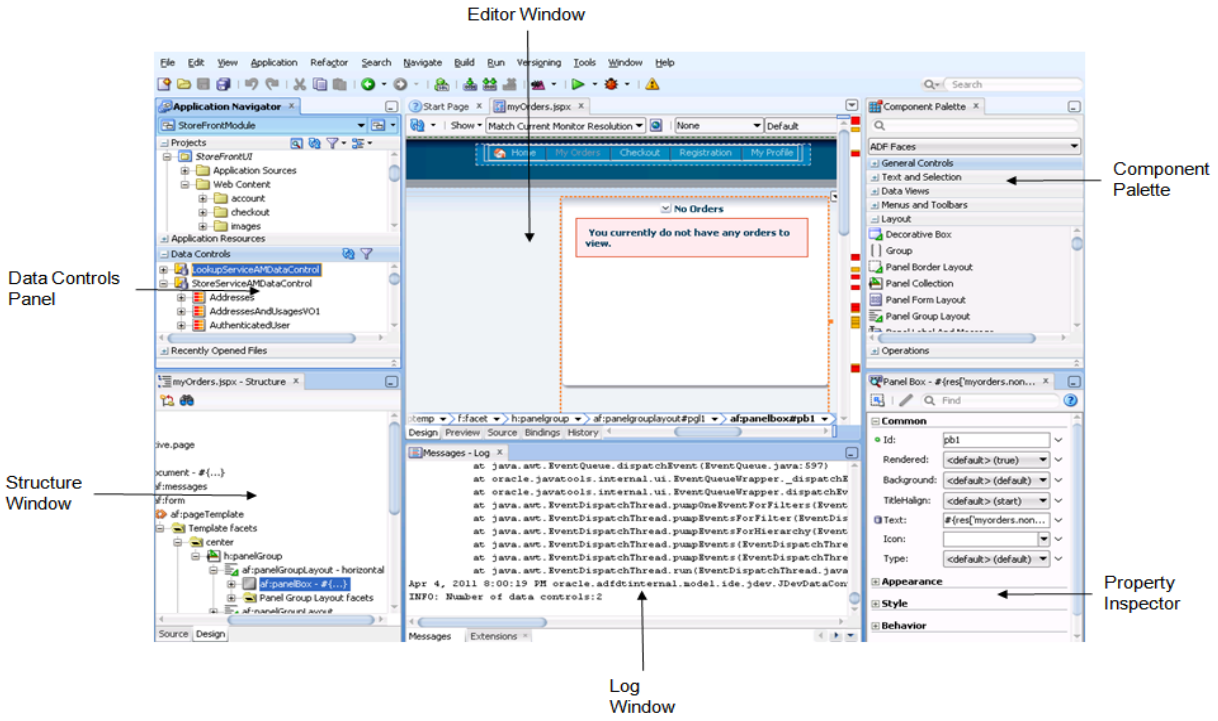
Figure 1–5 StoreFrontModule Application Projects, Packages, and Directories



For more information, see "Managing Applications and Projects" section of the *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.

When you work with your files, you use mostly the editor window, the Structure window, and the Property Inspector, as shown in Figure 1–6. The editor window allows you to view many of your files in a WYSIWYG environment, or you can view a file in an overview editor where you can declaratively make changes, or you can view the source code for the file. The Structure window shows the structure of the currently selected file. You can select objects in this window and then edit the properties for the selection in the Property Inspector.

Figure 1–6 The JDeveloper Workspace



1.3.2 Modeling with Database Object Definitions

In JDeveloper, after you create your application workspace, you can copy database objects from a database schema to an offline database or project where they become available as offline database objects, saved as `.xml` files. You can then create and edit database object definitions within a project using the same editors that you use to create and edit database objects on live database connections. You can also compare your offline database objects with other offline or live database schemas and generate SQL statements (including `CREATE`, `REPLACE`, and `ALTER`).

For example, you can drag a table from a database connection that your application defines onto a database diagram and JDeveloper will give you the choice to model the database object live or offline (to create the `.xml` file representation of the object). Modeling database definitions, such as tables and foreign keys, visually captures the essential information about a schema. You can use the diagram to drag and drop columns and keys to duplicate, move, and create foreign key relationships. Working in offline mode, whenever you model a node on a diagram, JDeveloper creates the underlying offline object and lists it in the Application Navigator. Working with a live schema, JDeveloper updates the live database object as you amend the diagram. You

can create multiple diagrams from the same offline database objects and spread your offline database across multiple projects.

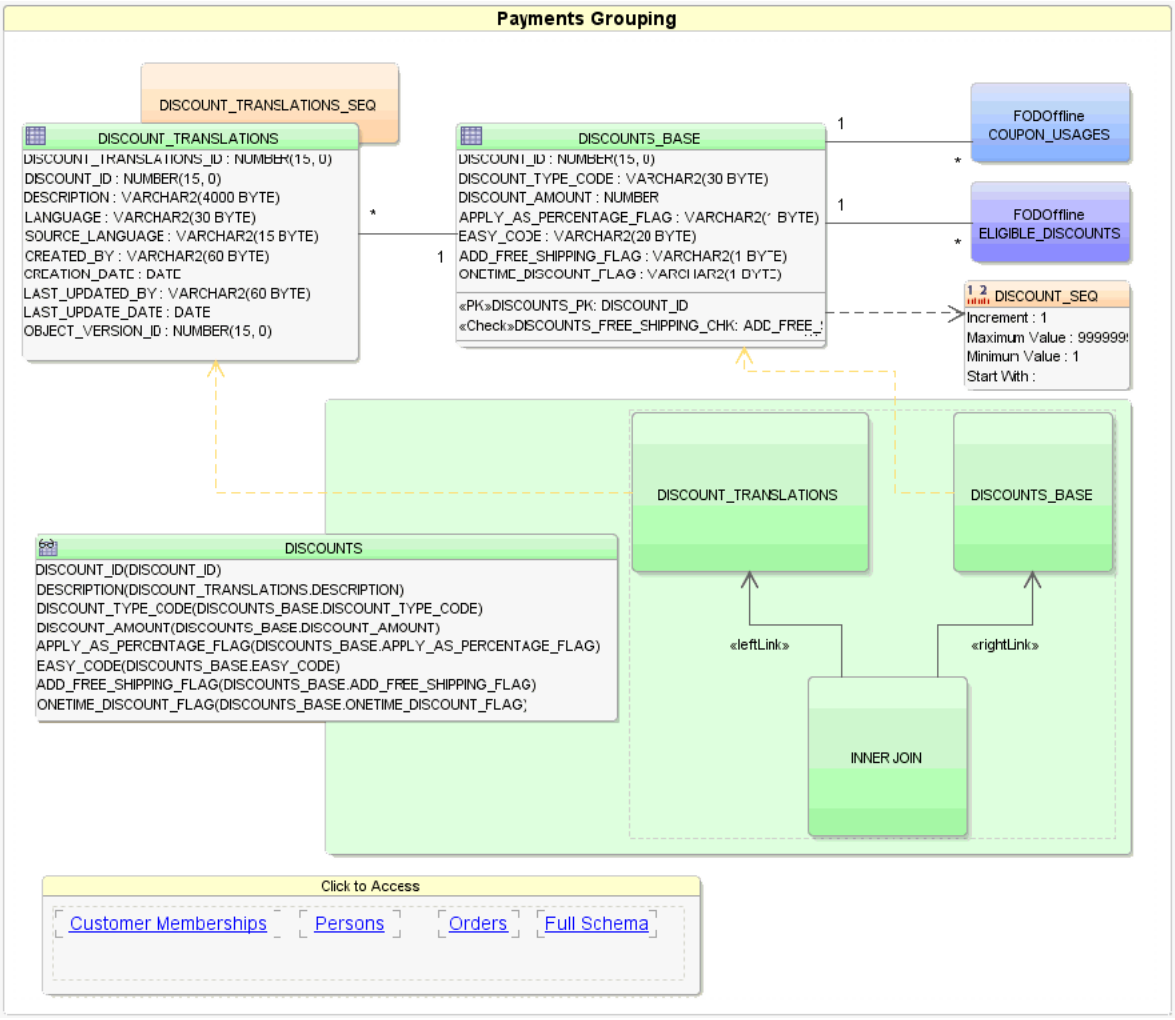
Using a database diagram like the one shown in Figure 1–7 you can visualize the following:

- Tables and their columns
- Foreign key relationships between tables
- Views
- Offline sequences and synonyms

In addition to using the diagram directly to create and edit database objects, you can work with specific database object editors. After you have finished working with the offline database objects, you can generate new and updated database definitions to online database schemas.

When you work with the database diagram you can customize the diagram to change the layout, change how objects are represented and grouped, add diagram annotations to specify dependencies or links (such as URLs), and change visual properties, such as color and font of diagram elements.

Figure 1–7 Database Diagram for Payments Grouping



Specifically, the following customizations were made to the database diagram shown in Figure 1–7:

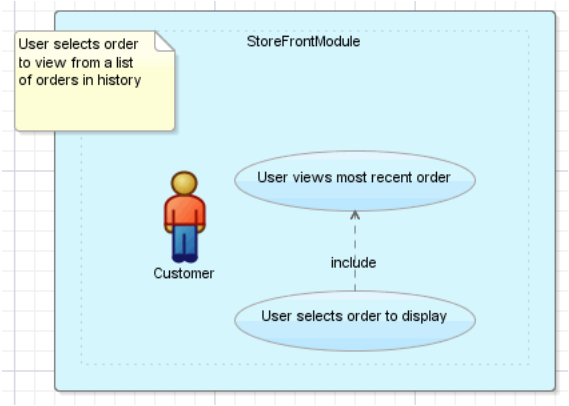
- DISCOUNT_TRANSLATIONS element show the table with constraints not displayed.
- DISCOUNTS_BASE element shows the table with some column definitions hidden (such as CREATED_BY, CREATION_DATE, and LASTUPDATED_BY) plus the diagram element has been sized to fit within the overall diagram (thus truncating some of the detail).
- DISCOUNT_TRANSLATIONS_SEQ element shows a sequence displayed in compact view in contrast to DISCOUNTS_SEQ which shows the sequence properties.
- COUPON_USAGES and ELIGIBLE_DISCOUNTS elements use different colors both in compact view and each identifies their database schema (FODOffline).
- DISCOUNTS element is a view displayed. The element identifies the tables that comprise the view in compact mode and the JOIN type (INNER JOIN). It also identifies the usage relationships on tables with a dotted line.
- DISCOUNTS_BASE and DISCOUNT_TRANSLATIONS elements show a foreign key relationship.
- DISCOUNT_SEQ element uses an annotation (dashed arrow) to represent the dependency with DISCOUNTS_BASE table.
- Payments Grouping element uses HTML link annotations (for example, Customer Memberships) to display other diagrams from the project.
- Payments Grouping element nests elements with group shapes (Click to Access).

For more information about modeling database definitions with database diagrams, see "Creating, Editing, and Dropping Database Objects" in the *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.

1.3.3 Creating Use Cases

After creating an application workspace, you may decide to begin the development process by doing use case modeling to capture and communicate end-user requirements for the application to be built. Figure 1–8 shows a simple diagram created using the UML modeler in JDeveloper. The diagram represents an end user viewing a list of his orders and then drilling down to view the details of an order. Using diagram annotations, you can capture particular requirements about what end users might need to see on the screens that will implement the use case. For example, in this use case, it is noted that the user will select order details for each order listed.

Figure 1–8 Use Case Diagram for Viewing Order History



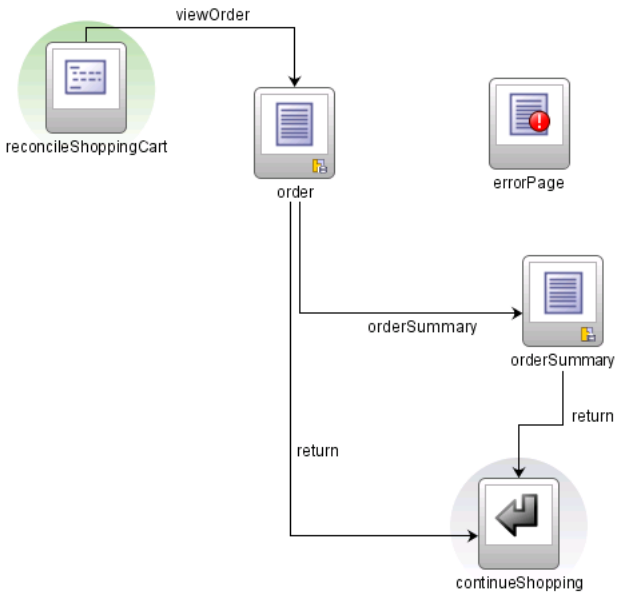
For more information about creating use case diagrams, see the "Getting Started With Application Modeling Using Diagrams" chapter of the *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.

1.3.4 Designing Application Control and Navigation Using ADF Task Flows

By modeling the use cases, you begin to understand the kinds of user interface pages that will be required to implement end-user requirements. At this point, you can begin to design the flow of your application. In a Fusion web application, you use ADF task flows instead of standard JSF navigation flows. Task flows provide a more modular and transaction-aware approach to navigation and application control. Like standard JSF navigation flows, task flows contain mostly viewable pages. However, instead of describing navigation between pages, task flows facilitate transitions between activities. Aside from navigation, task flows can also have nonvisual activities that can be chained together to affect the page flow and application behavior. For example, these nonvisual activities can call methods on managed beans, evaluate an EL expression, or call another task flow. This facilitates reuse, as business logic can be invoked independently of the page being displayed.

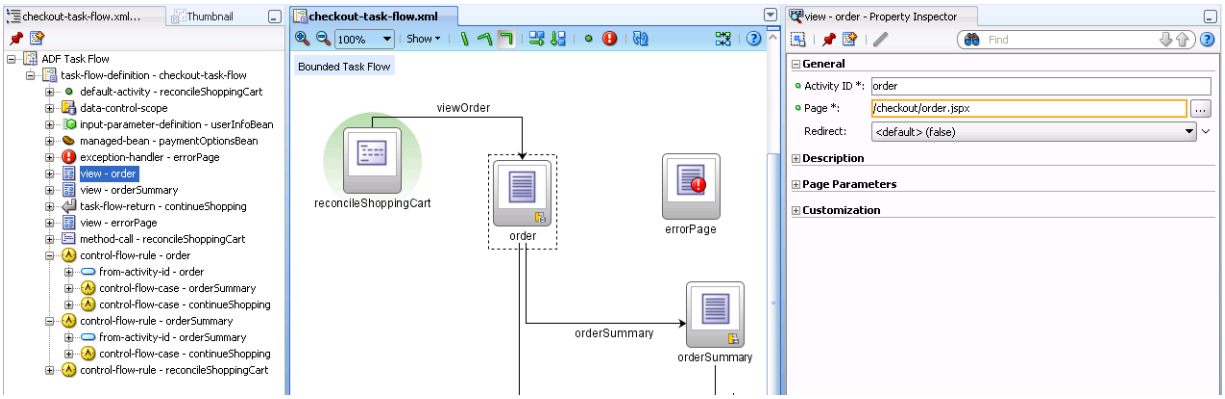
Figure 1–9 shows the `checkout-task-flow` task flow from the StoreFront module of the Fusion Order Demo application. In this task flow, `order` and `orderSummary` are view activities that represent pages, while `reconcileShoppingCart` is a method call activity. When the user enters this flow, the `reconcileShoppingCart` activity is invoked (because it is the entry point for the flow, as denoted by the green circle) and the corresponding method is called. From there, the flow continues to the `order` page. From the `order` page, control can be passed to the `orderSummary` page, or to the `continueShopping` return activity that is the exit point of the flow and passes control back to the home page.

Figure 1–9 Task Flow in the StoreFrontModule Application



ADF Controller provides a mechanism to define navigation using control flow rules. The control flow rule information, along with other information regarding the flow, is saved in a configuration file. Figure 1–10 shows the Structure window for the `checkout-task-flow` task flow. This window shows each of the items configured in the flow, such as the control flow rules. The Property Inspector (by default, located at the bottom right) allows you to set values for the different elements in the flow.

Figure 1–10 Task Flow Elements in the Structure Window and Property Inspector



Aside from pages, task flows can also coordinate page fragments. Page fragments are JSF JSP documents that are rendered as content in other JSF pages. You can create page fragments and the control between them in a bounded task flow as you would create pages, and then insert the entire task flow into another page as a region. Because it is simply another task flow, the region can independently execute methods, evaluate expressions, and display content, while the remaining content on the containing page remains the same. For example, before registering a new user, the application needs to determine what kind of user needs to be created. All the logic to do this is handled in the `user-registration-task-flow` task flow, which is used as a region in the `registerUser` page.

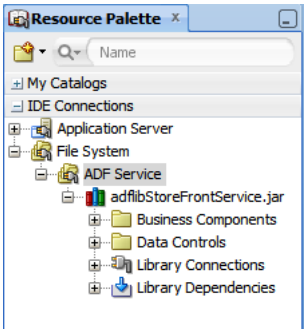
Regions also facilitate reuse. You can create a task flow as a region, determine the pieces of information required by a task and the pieces of information it might return, define those as parameters and return values of the region, then drop the region on any page in an application. Depending on the value of the parameter, a different view can display.

The chapters in Part IV, "Creating ADF Task Flows" contain information about using task flows. For general information about task flows and creating them, see Chapter 18, "Getting Started with ADF Task Flows." For information about task flow activities, see Chapter 19, "Working with Task Flow Activities." If you need to pass parameters into or out of task flows, see Chapter 20, "Using Parameters in Task Flows." For more information about regions, see Chapter 21, "Using Task Flows as Regions." For information about advanced functionality that task flows can provide, such as transactional capabilities and creating mandatory sequences of pages (known as *trains*), see Chapter 22, "Creating Complex Task Flows." For information about using task flows to create dialogs, see Chapter 23, "Using Dialogs in Your Application."

1.3.5 Identifying Shared Resources

You may find that some aspects of your application can be reused throughout the application. For example, you might have one developer that creates the business components, and another that creates the web interface. The business component developer can then save the project and package it as a library. The library can be sent to other developers who can add it to their resource catalog, from which they can drag and drop it onto any page where it's needed. Figure 1–11 shows the Resource Palette in JDeveloper.

Figure 1–11 Resource Palette in JDeveloper



When designing the application, be sure to note all the tasks that can possibly become candidates for reuse. Chapter 38, "Reusing Application Components" provides more information about the ADF artifacts that can be packaged and reused as an ADF library, along with procedures both for creating and using the libraries.

1.3.6 Creating a Data Model to Access Data with ADF Business Components

Typically, when you implement business logic as ADF Business Components, you do the following:

- Create entity objects to represent tables that you expect your application to perform a transaction against. Add validation and business rules as needed.
- Create view objects that work with the entity objects to query and update the database. These view objects will be used to make the data available for display at your view layer. You can also create read-only view objects, which you might use to display static lists.

- Create the application module, which provides the interface to the business services that a consumer of those services (such as the UI layer of your application) will use. This application module contains view object instances in its data model along with any custom methods that users will interact with through the application's web pages.
- If needed, publish your services as web services for remote invocation.

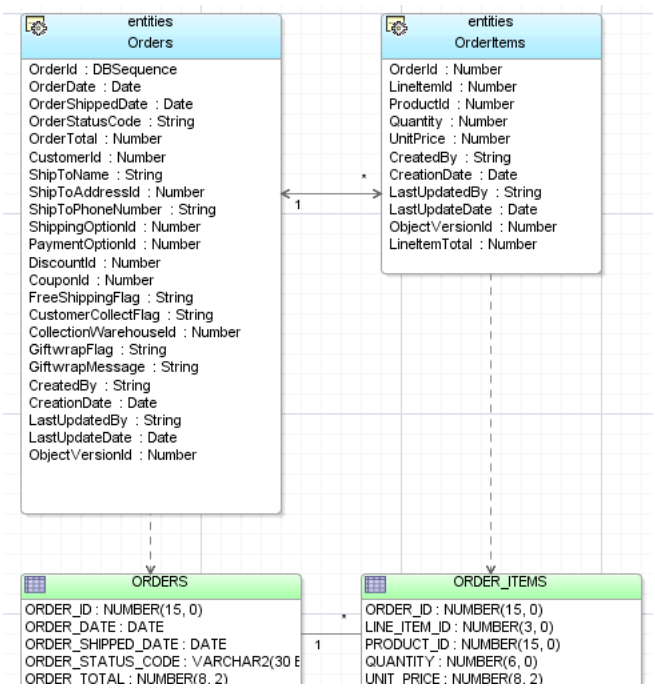
The chapters contained in Part II, "Building Your Business Services" provide information on creating each of these artifacts. The chapters in Part VII, "Advanced Topics" provide additional information, such as extending business objects, tuning, and state management.

1.3.6.1 Creating a Layer of Business Domain Objects for Tables

Once you have an understanding of the data that will be presented and manipulated in your application, if you haven't already done so, you can build your database (for more information, see the "Designing Databases Within Oracle JDeveloper" chapter of the *Oracle Fusion Middleware User Guide for Oracle JDeveloper*). Once the database tables are in place, you can create a set of entity objects that represents them and simplifies modifying the data they contain. When you use entity objects to encapsulate data access and validation related to the tables, any pages you build today or in the future that work with these tables are consistently validated. As you work, JDeveloper automatically configures your project to reference any necessary Oracle ADF runtime libraries your application will need at runtime.

For example, the `StoreFrontService` project of the `StoreFrontModule` application contains the business services needed by the application. Figure 1–12 shows two of the entity objects that represent the database tables in that application.

Figure 1–12 Business Components Diagram Showing Entity Objects and Related Tables



To create the business layer, you first create the entity objects based on your database tables. Any relationships between the tables will be reflected as associations between the corresponding entity objects.

Once the entity objects are created, you can define control and attribute hints that simplify the display of the entities in the UI, and you can also add behaviors to the objects. For more information, see Chapter 4, "Creating a Business Domain Layer Using Entity Objects."

1.3.6.2 Building the Business Services

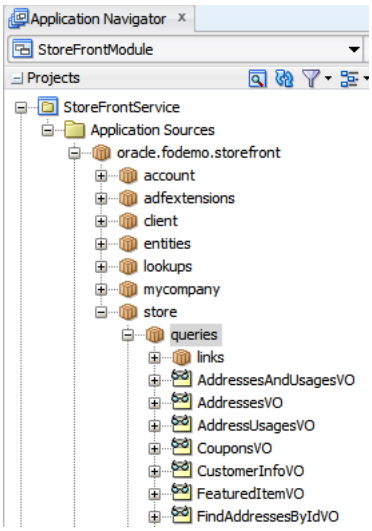
Once the reusable layer of business objects is created, you can implement the application module's data model and service methods with which a UI client can work.

The application module's data model is composed of instances of the view object components you create that encapsulate the necessary queries. View objects can join, project, filter, sort, and aggregate data into the shape required by the end-user task being represented in the user interface. When the end user needs to update the data, your view objects reference entity objects in your reusable business domain layer. View objects are reusable and can be used in multiple application modules.

When you want the data to display in a consistent manner across all view pages that access that data, you can configure metadata on the view object to determine display properties. The metadata allows you to set display properties in one place and then change them as needed, so that you make the change only in one place instead of on all pages that display the data. Conversely, you can also have the query controlled by the data the page requires. All display functionality is handled by the page. For more information, see Chapter 5, "Defining SQL Queries Using View Objects."

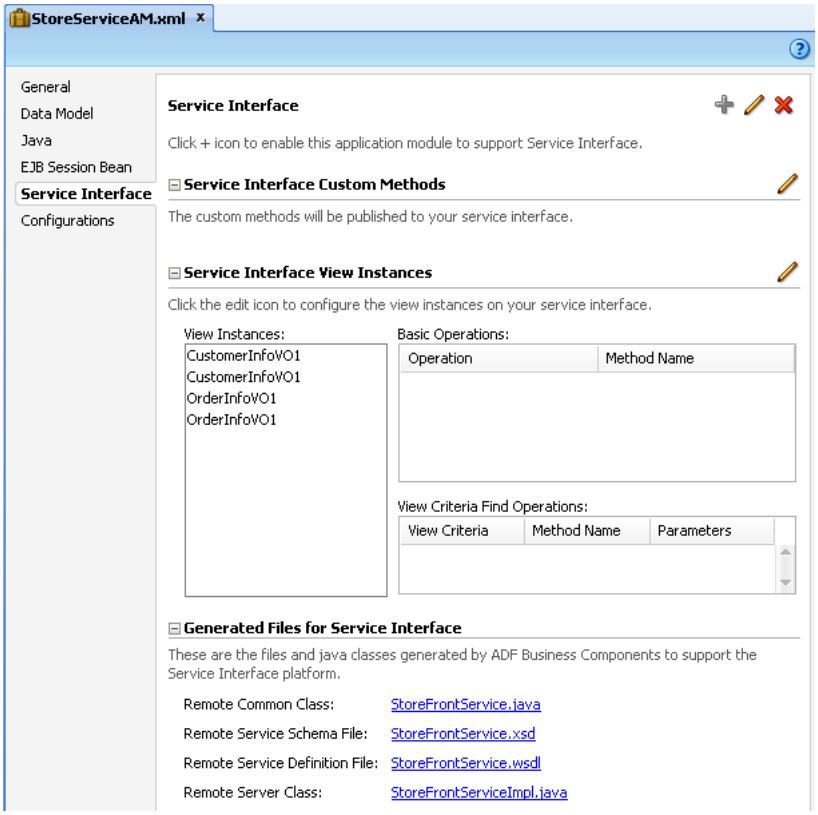
For example, the `StoreFrontService` project contains the `oracle.fodemo.storefront.store.queries` package, which contains many of the queries needed by the `StoreFrontModule` application, as shown in Figure 1–13.

Figure 1–13 View Objects in the StoreFrontModule Application



Additionally, you may find that you need to expose functionality to external applications. You can do this by exposing this functionality through a service interface. For example, the `StoreServiceAM` application module is exposed as a web service. This web service exposes the `CustomerInfo` and `OrderInfo` view instances, as shown in Figure 1–14. For more information, see Chapter 11, "Integrating Service-Enabled Application Modules."

Figure 1–14 StoreFrontModule Application in the Fusion Order Demo Application



1.3.6.3 Testing and Debugging Business Services with the Oracle ADF Model Tester

While you develop your application, you can iteratively test your business services using the Oracle ADF Model Tester. The tester allows you to test the queries, business logic, and validation of your business services without having to use or create a user interface or other client to test your services. Using the tester allows you to test out the latest queries or business rules you've added, and can save you time when you're trying to diagnose problems. For more information about developing and testing application modules, see Chapter 9, "Implementing Business Services with Application Modules."

The tester also interacts with the ADF Declarative Debugger to allow debug your business services. You can set breakpoints on any custom methods you create. For more information, see Section 36.7, "Using the Oracle ADF Model Tester for Testing and Debugging."

1.3.7 Implementing the User Interface with JSF

From the page flows you created during the planning stages, you can double-click the page icons to create the actual JSP files. When you create a JSP for an ADF Faces application, you can choose to create an XML-based JSP document (which uses the extension `*.jspx`) rather than a `*.jsp` file.

Best Practice: Using an XML-based document has the following advantages:

- Simplifies treating your page as a well-formed tree of UI component tags.
- Discourages you from mixing Java code and component tags.
- Allows you to easily parse the page to create documentation or audit reports.

If you want to use Facelets instead of JSP in your application, you can instead create XHTML files. Facelets is a JSF-centric XML view definition technology that provides an alternative to using the JSP engine.

Tip: While Facelet pages can use any well formed XML file, including `.jspx`, when you create a Facelet page in JDeveloper, it is created as an XHTML file.

Best Practice: Use Facelets to take advantage of the following:

- The Facelets layer was created specifically for JSF, which results in reduced overhead and improved performance during tag compilation and execution.
- Facelets is considered the primary view definition technology in JSF 2.0.
- Some future performance enhancements will only be available with Facelets

ADF Faces provides a number of components that you can use to define the overall layout of the page. JDeveloper contains predefined quick start layouts that use these components to provide you with an efficient way to correctly determine the layout of your pages. You can choose from one-, two-, or three-column layouts, and then determine how you want the columns to behave. You can also choose to apply themes to the layouts, which adds color to some of the components for you. For more information see the "Using Quick Layouts" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Oracle ADF also allows you to create and use your own page templates. When creating templates, a developer can determine the layout of the page (either using one of the quick layout templates or creating the layout manually), provide static content that must appear on all pages, and create placeholder attributes that can be replaced with valid values for each page. Each time the template is changed, for example if the layout changes, any page that uses the template will reflect the update.

Most pages in the `StoreFrontModule` application use the `StoreFrontTemplate` template, which provides an area for branding and navigation, a main content area divided into three panes, and a footer area. If the template designer decides to switch the location of the branding and the navigation, all pages that use the template will automatically reflect that change at runtime.

The chapters in Part V, "Creating a Databound Web User Interface" provide information on creating different types of UI functionality, from basic forms to more complex search capabilities.

1.3.8 Data Binding with ADF Model Layer

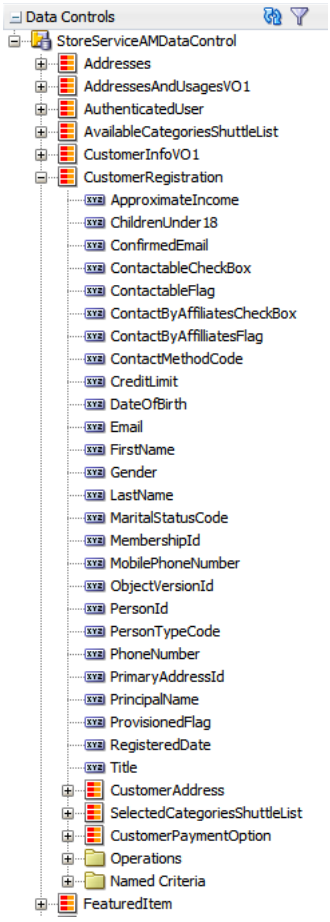
In JSF, you use a simple *expression language* (called EL) to bind to the information you want to present and/or modify (for more information, see http://www.oracle.com/technetwork/java/unifiedel-139263.html). Example expressions look like `#{userInfoBean.principalName}` to reference a particular user's name, or `#{userInfoBean.principalName eq 'SKING'}` to evaluate whether a user's name is SKING or not. At runtime, a generic expression evaluator returns the `String` and `boolean` value of these respective expressions, automating access to the individual objects and their properties without requiring code.

At runtime, the value of certain JSF UI components is determined by the `value` attribute. While a component can have static text as its value, typically the `value` attribute will contain a binding that is an EL expression that the runtime infrastructure evaluates to determine what data to display. For example, an `outputText` component that displays the name of the currently logged-in user might have its `value` attribute set to the expression `#{userInfoBean.principalName}`. Since any attribute of a component can be assigned a value using an EL expression, it's easy to build dynamic, data-driven user interfaces. For example, you could hide a component when a user is not logged in by using a boolean-valued expression like `#{userInfoBean.prinicpalName !=null}` in the UI component's `rendered` attribute. If there is no principal name in the current instantiation of the `userInfoBean`, the `rendered` attribute evaluates to `false` and the component disappears from the page.

In a typical JSF application, you would create objects like the `userInfoBean` object as a managed bean. The JSF runtime manages instantiating these beans on demand when any EL expression references them for the first time. However, in an application that uses the ADF Model layer, instead of binding the UI component attributes to properties or methods on managed beans, JDeveloper automatically binds the UI component attributes to the ADF Model layer, which uses XML configuration files that drive generic data binding features. It implements concepts that enable decoupling the user interface technology from the business service implementation: *data controls* and *declarative bindings*.

Data controls use XML configuration files to describe a service. At design time, visual tools like JDeveloper can leverage that metadata to allow you to declaratively bind UI components to any data control operation or data collection, creating bindings. For example, Figure 1–15 shows the `StoreServiceAMDataControl` data control as it appears in the Data Controls panel of JDeveloper.

Figure 1–15 StoreFrontServiceAMDataControl

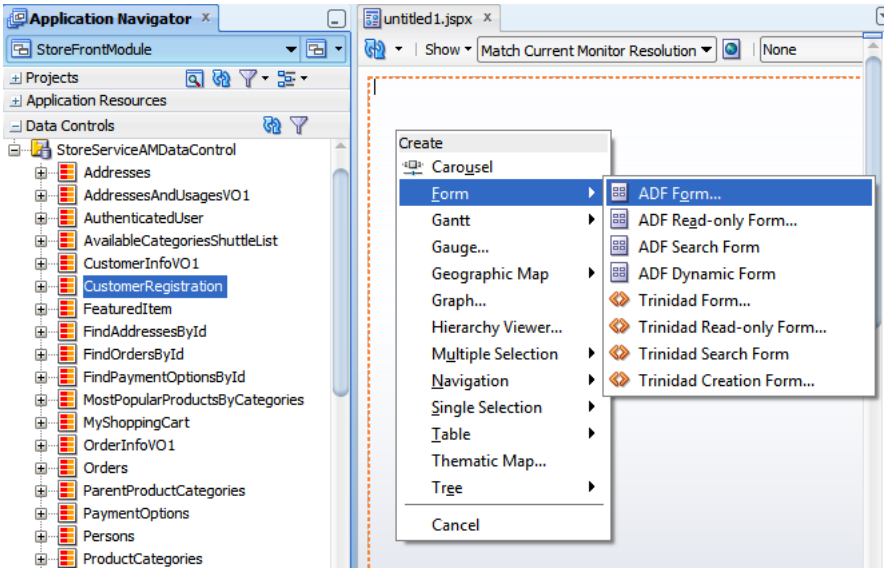


Note that the collections that display in the panel represent the set of rows returned by the query in each view object instance contained in the `StoreServiceAM` application module. For example, the `CustomerRegistration` data collection in the Data Controls panel represents the `CustomerRegistrationVO` view object instance in the `StoreServiceAM's` data model. The `CustomerAddress` data collection appears as a child, reflecting the master-detail relationship set up while building the business service. The attributes available in each row of the respective data collections appear as child nodes. The data collection level Operations node contains the built-in operations that the ADF Model layer supports on data collections, such as `previous`, `next`, `first`, `last`, and so on.

Note: If you create other kinds of data controls for working with web services, XML data retrieved from a URL, JavaBeans, or EJBs, these would also appear in the Data Controls panel with an appropriate display. When you create one of these data controls in a project, JDeveloper creates metadata files that contain configuration information. These additional files do not need to be explicitly created when you are working with Oracle ADF application modules, because application modules are already metadata-driven components, and so contain all the information necessary to be exposed automatically as data controls.

Using the Data Controls panel, you can drag and drop a data collection onto a page in the visual editor, and JDeveloper creates the necessary bindings for you. Figure 1–16 shows the `CustomerRegistration` collection from the `StoreServiceAMDataControl` data control being dragged from the Data Controls panel, and dropped as a form onto a JSF page.

Figure 1–16 Declaratively Creating a Form Using the Data Controls Panel

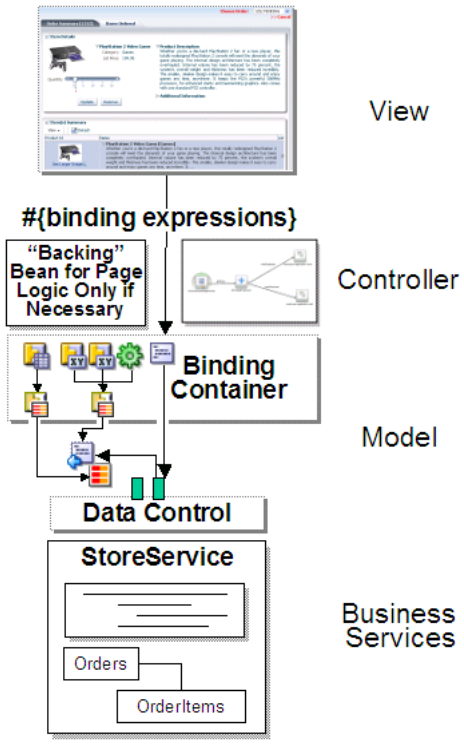


The first time you drop a databound component from the Data Controls panel on a page, JDeveloper creates an associated page definition file. This XML file describes the group of bindings supporting the UI components on a page. The ADF Model uses this file at runtime to instantiate the page's bindings. These bindings are held in a request-scoped map called the *binding container*. Each time you add components to the page using the Data Controls panel, JDeveloper adds appropriate binding entries into this page definition file. Additionally, as you perform drag and drop data binding operations, JDeveloper creates the required tags representing the JSF UI components on the JSF page. For more information about using the Data Controls panel, see Chapter 13, "Using ADF Model in a Fusion Web Application."

Note: You can use dynamic UI components that create the bindings at runtime instead of design time. To use dynamic components, you set control hints on your view objects that determine how the data is to be displayed each time the view object is accessed by a page. This ensures that data is displayed consistently across pages, and also allows you to change in a single location, how the data is displayed instead of having to update each individual page. For more information, see Section 26.7, "Using a Dynamic Form to Determine Data to Display at Runtime."

Figure 1–17 illustrates the architecture of a JSF application when you leverage ADF Model for declarative data binding. By combining ADF Model with JSF, you avoid having to write a lot of the typical managed bean code that would be required for real-world applications.

Figure 1–17 Architecture of a JSF Application Using ADF Model Data Binding

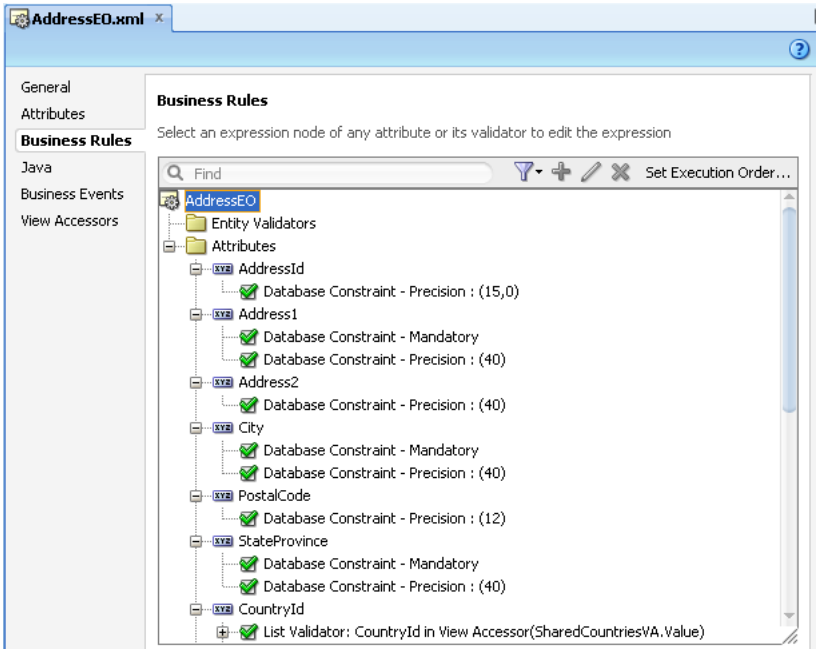


Aside from forms and tables that display or update data, you can also create search forms, and databound charts and graphs. For more information about using data controls to create different types of pages, see the chapters contained in Part V, "Creating a Databound Web User Interface". For more information about the Data Controls panel and how to use it to create any UI data bound component, see Chapter 13, "Using ADF Model in a Fusion Web Application."

1.3.9 Validation and Error Handling

You can add validation to your business objects declaratively using the overview editors for entity and view objects. Figure 1–18 shows the **Business Rules** page of the overview editor for the `AddressEO` entity object.

Figure 1–18 Setting Validation in the Overview Editor



Along with providing the validation rules, you also set the error messages to display when validation fails. To supplement this declarative validation, you can also use Groovy-scripted expressions. For more information about creating validation at the service level, see Chapter 7, "Defining Validation and Business Rules Declaratively."

Additionally, ADF Faces input components have built-in validation capabilities. You set one or more validators on a component either by setting the `required` attribute or by using the prebuilt ADF Faces validators. You can also create your own custom validators to suit your business needs. For more information, see the "Validating and Converting Input" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

You can create a custom error handler to report errors that occur during execution of an ADF application. Once you create the error handler, you only need to register the handler in one of the application's configuration files.

1.3.10 Adding Security

Oracle ADF provides a security implementation that is based on Java Authentication and Authorization Service (JAAS). JAAS is a standard security Application Programming Interface (API) that is added to the Java language through the Java Community Process. It enables applications to authenticate users and enforce authorization. The Oracle ADF implementation of JAAS is permission-based. You define these permissions and then grant them on application roles that you associate with users of the application. For more information about securing your application, see Chapter 35, "Enabling ADF Security in a Fusion Web Application."

1.3.11 Testing and Debugging the Web Client Application

Testing an Oracle ADF web application is similar to testing and debugging any other Java EE application. Most errors result from simple and easy-to-fix problems in the declarative information that the application defines or in the EL expressions that

access the runtime objects of the page's ADF binding container. In many cases, examination of the declarative files and EL expressions resolve most problems.

For errors not caused by the declarative files or EL expressions, JDeveloper includes the ADF Logger, which captures runtime trace messages from the ADF Model layer API. The trace includes runtime messages that may help you to quickly identify the origin of an application error. You can also search the log output for specific errors.

JDeveloper also includes the ADF Declarative Debugger, a tool that allows you to set breakpoints on declarative aspects of Oracle ADF, such as the binding layer, taskflows and more. When a breakpoint is reached, the execution of the application is paused and you can examine the data that the ADF binding container has to work with, and compare it to what you expect the data to be. Chapter 36, "Testing and Debugging ADF Components" contains useful information and tips on how to successfully debug a Fusion web application.

For testing purposes, JDeveloper provides integration with JUnit. You use a wizard to generate regression test cases. For more information, see Section 36.11, "Regression Testing with JUnit."

1.3.12 Refactoring Application Artifacts

Using JDeveloper, you can easily rename or move the different components in your application. For example, you may find that you need to change the name of your view objects after you have already created them. JDeveloper allows you to easily do this and then propagates the change to all affected metadata XML files. For more information, see Chapter 37, "Refactoring a Fusion Web Application."

1.3.13 Deploying a Fusion Web Application

You can deploy a Fusion web application to either the integrated WebLogic server within JDeveloper or to a standalone instance. For more information about deployment, see Chapter 41, "Deploying Fusion Web Applications."

1.3.14 Integrating a Fusion Web Application

You can integrate your Fusion web application with any existing or new applications using service-oriented architecture (SOA) principals provided by Oracle SOA Suite. Oracle SOA Suite includes declarative development tools that allow you to easily integrate multiple applications using services, events, business rules, business process flows, and other SOA technologies.

You can build your Fusion web application so that it can easily integrate with other applications. You can publish your application modules as services.You can also create events that can be used for example, to initiate business processes. For more information, see Chapter 11, "Integrating Service-Enabled Application Modules." Your application modules can also call other web services directly. For more information, see Section 9.12, "Calling a Web Service from an Application Module." You can also integrate your application using task flows. For example, a task flow can be used to initiate a business process flow.

For more information about Oracle SOA Suite, see *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*.

1.4 Working Productively in Teams

Often, applications are built in a team development environment. While a team-based development process follows the development cycle outlined in Section 1.3, "Overview of Building an Application with Oracle ADF,"many times developers are creating the different parts of the application simultaneously. Working productively means team members divide the work, understand how to enforce standards, and manage source files with a source control system, in order to ensure efficient application development.

Before beginning development on any large application, a design phase is typically required to assess use cases, plan task flows and screens, and identify resources that can be shared.

The following list shows how the work for a typical Fusion web application might be broken up once an initial design is in place:

- Infrastructure

  An administrator creates Ant scripts (or other script files) for building and deploying the finished application. SQL scripts are developed to create the database schema used by the application.

- Entity objects

  In a large development environment, a separate development group builds all entity objects for the application. Because the rest of the application depends on these objects, entity objects should be one of the first steps completed in development of the application.

  Once the entity objects are finished, they can be shared with other teams using Oracle ADF libraries (see Section 38.3, "Packaging a Reusable ADF Component into an ADF Library" for more information). The other teams then access the objects by adding to them to a catalog in the Resource Palette. In your own application development process, you may choose not to divide the work this way. In many applications, entity objects and view objects might be developed by the same team (or even one person) and would be stored within one project.

- View objects

  After the entity objects are created and provided either in a library or within the project itself, view objects can be created as needed to display data (in the case of building the UI) or supply service data objects (when data is needed by other applications in a SOA infrastructure).

  When building the Fusion Order Demo application, each developer of a particular page or service was in charge of creating the view objects for that page or service. This was needed because of the tight integration between the view object and its use by a page in the Fusion Order demo; the team who built the UI also built the corresponding view objects.

  During development, you may find that two or more view objects are providing the same functionality. In some cases, these view objects can be easily combined by altering the query in one of the objects so that it meets the needs of each developer's page or service.

  Once the view objects are in place, you can create the application module, data controls, and add any needed custom methods. The process of creating view objects, reviewing for redundancy, and then adding them to the application module can be an iterative one.

- User interface (UI) creation

With a UI design in place, the view objects in place and the data controls created, the UI can be built either by the team that created the view objects (as described in the previous bullet point) or by a separate team. You can also develop using a UI-first strategy, which would allow UI designers to create pages before the data controls are in place. Oracle ADF provides placeholder data controls that UI designers can use early in the development cycle. For more information, see Chapter 17, "Designing a Page Using Placeholder Data Controls."

1.4.1 Enforcing Standards

Because numerous individuals divided into separate teams will be developing the application, you should enforce a number of standards before development begins to ensure that all components of the application will work together efficiently. The following are areas within an application where it is important to have standardization in place when working in a team environment:

- Code layout style

  So that more than one person can work efficiently in the code, it helps to follow specific code styles. JDeveloper allows you to choose how the built-in code editor behaves. While many of the settings affect how the user interacts with the code editor (such as display settings), others affect how the code is formatted. For example, you can select a code style that determines things like the placement of opening brackets and the size of indents. You can also import any existing code styles you may have, or you can create your own and export them for use by the team. For more information, see "How to Set Preferences for the Source Editor" in the *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.

- Package naming conventions

  You should determine not only how packages should be named, but also the granularity of how many and what kinds of objects will go into each package. For example, all managed beans in the StoreFront module are in the `view.managed` package. All beans that contain helper-type methods accessed by other beans are in `util` packages (one for Oracle ADF and one for JSF). All property files are in the `common` package.

- Pages

  You can create templates to be used by all developers working on the UI, as described in Section 1.3.7, "Implementing the User Interface with JSF." This not only ensures that all pages will have the same look and feel, but also allows you to make a change in the template and have the change appear on all pages that use it. For more information, see Section 24.2, "Using Page Templates."

  Aside from using templates, you should also devise a naming standard for pages. For example, you may want to have names reflect where the page is used in the application. To achieve this goal, you can create subdirectories to provide a further layer of organization.

- Connection names: Unlike most JDeveloper and Oracle ADF objects that are created only once per project and by definition have the same name regardless of who sees or uses them, database connection names might be created by individual team members, even though they map to the same connection details. Naming discrepancies may cause unnecessary conflicts. Team members should agree in advance on common, case-sensitive connection names that should be used by every member of the team.

1.4.2 Using a Source Control System

When working in a team environment, you will need to use a source control system. By default, JDeveloper provides support for the Subversion source control system, and others (such as CVS) are available through extensions. You can also create an extension that allows you to work with another system in JDeveloper. For information about using these systems within JDeveloper, see the "Versioning Applications with Source Control" chapter of the *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.

Following are suggestions for using source control with a Fusion web application:

- Checking out files

  Using JDeveloper, you can create a connection to the source control server and use the source control window to check out the source. When you work locally in the files, the pending changes window notifies you of any changed files. You can create a script using Apache Ant (which is integrated into JDeveloper). You can then use the script to build all application workspaces locally. This can ensure that the source files compile before you check the changed files into the source control repository. To find out how to use Apache Ant to create scripts, see the "Building With Apache Ant" section of the *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.

- Automating builds

  Consider running a continuous integration tool. Once files are checked into the source server, the tool can be used to recognize either that files have changed or to check for changed files at determined intervals. At that point, the tool can run an Ant script on the server that copies the full source (note that this should be a copy, and not a checkout), compiles those files, and if the compilation is successful, creates a zip file for consumers (not developers) of the application to use. The script should then clean up the source directory. Running a continuous integration tool will improve confidence in the quality of the code in the repository, encourage developers to update more often, and lead to smaller updates and fewer conflicts. Hudson (`http://hudson-ci.org/`) is one example of a continuous integration tool.

- Updating and committing files

  When working with Subversion, updates and commits should be done at the Working Copy level, not on individual files. If you attempt to commit and update an individual file, there is a chance you will miss a supporting metadata file and thereby corrupt the committed copy of the application.

- Resolving merge conflicts

  When you add or remove business components in a data model project with ADF Business Components, JDeveloper reflects it in the project file (`.jpr`). When you create (or refactor) a component into a new package, JDeveloper reflects that in the project file and in the ADF Business Components project configuration file (`.jpx`). Although the XML format of these project control files has been optimized to reduce occurrences of merge conflicts, merge conflicts may still arise and you will need to resolve them in JDeveloper using the **Resolve Conflicts** option on the context menu of each affected file.

  After resolving merge conflicts in any ADF Business Components XML component descriptor files, the project file (`.jpr`) for a data model project, or the corresponding ADF Business Components project configuration file (`.jpx`), close and reopen the project to ensure that you're working with latest version of the component definitions. To do this, select the project in the Application Navigator,

choose **File** > **Close** from the JDeveloper main menu, and then expand the project again in the Application Navigator.

1.5 Other Resources for Learning Oracle ADF

In addition to this developers guide, Oracle also offers the following resources to help you learn how you can best use Oracle ADF in your applications:

- Cue Cards in JDeveloper: JDeveloper cue cards provide step-by-step support for the application development process using Oracle ADF. They are designed to be used either with the included examples and a sample schema, or with your own data. Cue cards also include topics that provide more detailed background information, viewlets that demonstrate how to complete the steps in the card, and code samples. Cue cards provide a fast, easy way to become familiar with the basic features of Oracle ADF, and to work through a simple end-to-end task.
- Tutorials, technical papers, samples, and more on Oracle Technology Network.
- Oracle Press publishes a number of books on JDeveloper and Oracle ADF.
- Other developer guides for supporting Oracle ADF technology: Oracle provides developer guides for ADF Faces, Oracle ADF Desktop Integration, and Oracle Mobile Browser, to name a few. For more information about related guides, see Related Documents, in this book's Preface.

1.6 Generation of Complete Web Tier Using Oracle JHeadstart

As you'll learn throughout the rest of this guide, JDeveloper and Oracle ADF give you a productive, visual environment for building richly functional, database-centric Java EE applications with a maximally declarative development experience. However, if you are used to working with tools like Oracle Designer that offer *complete* user interface generation based on a higher-level application structure definition, you may be looking for a similar facility for your Java EE development. If so, then the Oracle JHeadstart application generator may be of interest to you. It is an additional extension for JDeveloper that uses Oracle ADF's built-in features to offer complete web-tier generation for your application modules. Starting with the data model you've designed for your ADF business service, you use the integrated editors that JHeadstart adds to the JDeveloper environment to iteratively refine a higher-level application structure definition. These editors controls the functionality and organization of the view objects' information in your generated web user interface. By checking boxes and choosing various options from dropdown lists, you describe a logical hierarchy of pages that can include multiple styles of search regions, list of values (LOVs) with validation, shuttle controls, nested tables, and other features. These declarative choices use terminology familiar to Oracle Forms and Designer users, further simplifying web development. Based on the application structure definition, you generate a complete web application that automatically implements the best practices described in this guide, easily leveraging the most sophisticated features that Oracle ADF and JSF have to offer.

Whenever you run the JHeadstart application generator, rather than generating *code*, it creates (or regenerates) all of the declarative view and controller layer artifacts of your Oracle ADF-based web application. These artifacts use the ADF Model layer and work with your ADF application module as their business service. The generated files are the same kinds you produce when using the JDeveloper built-in visual editors. The key difference is that JHeadstart creates them in bulk, based on a higher-level definition that you can iteratively refine until the generated pages match your end users' requirements as closely as possible. The generated files include:

- JSF Pages with databound ADF Faces UI components
- ADF Model page definition XML files describing each page's data bindings
- JSF navigation rules to handle page flow
- Resource files containing localizable UI strings

Once you've generated a maximal amount of your application's web user interface, you can spend your time using the productive environment of JDeveloper to tailor the results or to concentrate your effort on additional showcase pages that need special attention. Once you've modified a generated page, you can adjust a setting to avoid regenerating that page on subsequent runs of the application generator. Of course, since both the generated pages and your custom designed ones leverage the same ADF Faces UI components, all of your pages automatically inherit a consistent look and feel. For more information on how to get a fully functional trial of JHeadstart for evaluation, including details on pricing, support, and additional services, see the JHeadstart page on the Oracle Technology Network at http://www.oracle.com/technetwork/developer-tools/jheadstart/overview/index.html

2

Introduction to the ADF Sample Application

As a companion to this guide, the StoreFront module of the Fusion Order Demo application was created to demonstrate the use of the Fusion web application technology stack to create transaction-based web applications as required for a web shopping storefront. The demonstration application is used as an example throughout this guide to illustrate points and provide code samples.

Before examining the individual components and their source code in depth, you may find it helpful to install and become familiar with the functionality of the Fusion Order Demo application.

This chapter includes the following sections:

- Section 2.1, "About the Oracle Fusion Order Demo"
- Section 2.2, "Setting Up the Fusion Order Demo Application"
- Section 2.3, "Running the Fusion Order Demo Application StoreFront Module"
- Section 2.4, "Running the Fusion Order Demo Standalone Applications"
- Section 2.5, "Taking a Look at the Fusion Order Demo Application"

2.1 About the Oracle Fusion Order Demo

In this sample application, electronic devices are sold through a storefront-type web application. Customers can visit the web site, register, and place orders for the products. In order to register customers and fulfill orders, currently only a single application is in place. In a future release, several applications, will cooperate. For a detailed description of how the application works at runtime, see Section 2.5, "Taking a Look at the Fusion Order Demo Application."

In order to view and run the demo, you need to install Oracle JDeveloper Studio. You then need to download the application for this demonstration. Instructions to complete these tasks appear in this chapter. For complete details, see Section 2.2, "Setting Up the Fusion Order Demo Application."

Once the application is installed and running, you can view the code using JDeveloper. You can view the application at runtime by logging in as an existing customer and placing an order.

2.2 Setting Up the Fusion Order Demo Application

The Fusion Order Demo application runs using an Oracle database and JDeveloper. The platforms supported are the same as those supported by JDeveloper.

To prepare the environment and run the Fusion Order Demo application, you must:

1. Install Oracle JDeveloper Studio and meet the installation prerequisites. The Fusion Order Demo application requires an existing Oracle database. For details, see Section 2.2.1, "How to Download the Application Resources."
2. Install the Fusion Order Demo application from the Oracle Technology Network. For details, see Section 2.2.2, "How to Install the Fusion Order Demo Schema."
3. Install Mozilla FireFox, version 2.0 or higher, or Internet Explorer, version 7.0 or higher.
4. Run the application on a monitor that supports a screen resolution of 1024 X 768 or higher. For details, see Section 2.3, "Running the Fusion Order Demo Application StoreFront Module."

2.2.1 How to Download the Application Resources

The Fusion Order Demo application requires an existing Oracle database. You run the Fusion Order Demo application using JDeveloper.

Do the following before installing the Fusion Order Demo application:

- Install JDeveloper, as described in the *Oracle Fusion Middleware Installation Guide for Oracle JDeveloper*. You will need to install the Studio configuration of Oracle JDeveloper to view the application's projects and run the application in Integrated WebLogic Server. You can download JDeveloper from:

  `http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html`

  Note: When you download and install JDeveloper, ensure that it is the Studio configuration, not the Java configuration. You can verify these details in JDeveloper from the **Help** > **About** menu option.

- Download the Fusion Order Demo application ZIP file (`FusionOrderDemo.zip`). You can download the ZIP file from:

  `http://www.oracle.com/technetwork/developer-tools/jdev/index-095536.html`

- Install an Oracle database. The Fusion Order Demo application requires a database for its data.

  The SQL scripts were written for an Oracle database, so you will need some version of an Oracle RDBMS, such as 11*g*, or XE. The scripts will not install into Oracle Lite. If you wish to use Oracle Lite or some other database, then you will need to modify the database scripts accordingly. You can download an Oracle database from:

  `http://www.oracle.com/technetwork/index.html`

  Specifically, the small footprint of the Oracle Express Edition (XE) is ideally suited for setting up the database on your local machine. You can download it from:

  `http://www.oracle.com/technetwork/database/express-edition/overview/index.html`

2.2.2 How to Install the Fusion Order Demo Schema

You can download the Fusion Order Demo application from the Oracle Technology Network (OTN) web site.

To download the demo and install the FOD schema to your database:

1. Navigate to `http://www.oracle.com/technetwork/developer-tools/jdev/index-095536.html` and download the ZIP file to a local directory.
2. Start JDeveloper and from the main menu choose **File** > **Open**.
3. In the Open dialog, browse to the location where you extracted the ZIP file to in Step 1 and select **Infrastructure.jws** from the **infrastructure** directory. Click **Open**.
4. In the Application Navigator, expand **MasterBuildScript** and then **Resources**, and double-click **build.properties**.
5. In the editor, modify the properties shown in Table 2–1 for your environment.

Table 2–1 Properties Required to Install the Fusion Order Demo Application

Property	Description
`jdeveloper.home`	The root directory where you have JDeveloper installed. For example: `C:/JDeveloper/11/jdeveloper`
`jdbc.urlBase`	The base JDBC URL for your database in the format `jdbc:oracle:thin:@<yourhostname>`. For example: `jdbc:oracle:thin:@localhost`
`jdbc.port`	The port for your database. For example: `1521`
`jdbc.sid`	The SID of your database. For example: `ORCL` or `XE`
`db.adminUser`	The administrative user for your database. For example: `system`
`db.demoUser.tablespace`	The table space name where FOD users will be installed. For example: `USERS`

6. From the main menu, choose **File > Save All**.
7. In the Application Navigator, under the **Resources** node, right-click **build.xml** and choose **Run Ant Target > buildAll**.
8. In the Enter Property dialog, enter the password for the database system user and click **Continue**.

   Once you enter the password, the Ant build script creates the FOD users and populates the tables in the FOD schema. In the Apache Ant - Log window, you will see a series of SQL scripts and finally:

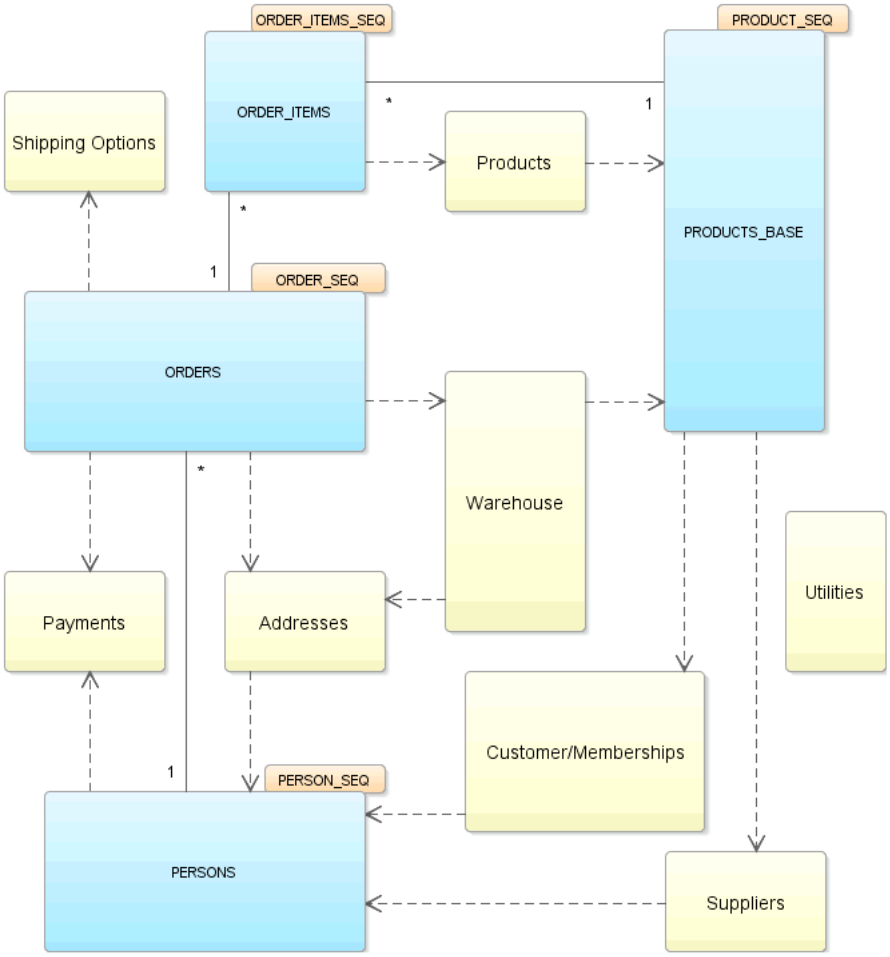
```
buildAll:

BUILD SUCCESSFUL
Total time: nn minutes nn seconds
```

For more information on the demo schema and scripts, see the `README.txt` file in the `MasterBuildScript` project.

2.2.3 Overview of the Fusion Order Demo Schema

Figure 2–1 shows a simplified representation of the schema for the Fusion Order Demo application. The blue shapes in the diagram represent the four core tables. The other tables and views are shown as yellow shapes that sometimes represent several tables to help simplify the diagram. Some of the tables use sequences, but only those used by the core tables are shown.

Figure 2–1 Schema Diagram for the Fusion Order Demo Application



The core tables represented by the blue diagram elements include:

- `PERSONS`: This table stores all the users who interact with the system, including customers, staff, and suppliers. The first and last name, email address, and person type code of each user is stored. A user is uniquely identified by an ID. Other IDs provide foreign keys to tables with address information and, in the case of customer's, membership information.
- `ORDERS`: This table represents activity by specific customers. When an order is created, the date of the order, the total amount of the order, the ID of the customer who created it, and the status of the order are all recorded. After the order is

fulfilled, the order status and order shipped date are updated. All orders are uniquely identified by a sequence-assigned ID.

- `ORDER_ITEMS`: For each order, there may be many order items recorded. The unit price and quantity of each order item are recorded. The order line item and its order ID uniquely identify each order item.
- `PRODUCTS_BASE`: This table stores all of the products available in the store. For each product, the name and cost are recorded. All products are uniquely identified by a sequence-assigned ID. The image of the product and its description are stored in separate tables, which each reference the product ID. The columns `ATTRIBUTE`*x* are reserved for future use with descriptive flexfields (commonly required by Oracle E-Business Suite schema).

The sequences that the core tables use include:

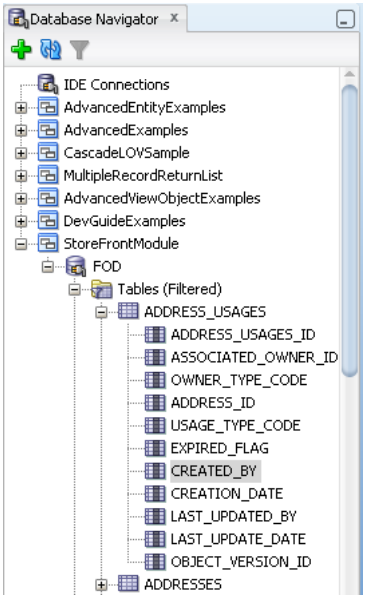
- `PERSON_SEQ`: Populates the ID for for each new person.
- `ORDER_SEQ`: Populates the ID for each new order.
- `ORDERS_ITEMS_SEQ`: Populates the ID for each new order item.
- `PRODUCTS_SEQ`: Populates the ID for each product.

The PL/SQL package `USER_CONTEXT_PKG` contains a procedure `set_app_user_lang()` used to illustrate a simple example of how to set per-user database state from inside an application module.

Note the `SHIPPING_OPTIONS` view is reserved for future use and is not currently used in the Fusion Order Demo application.

To support tracking of change history in the Fusion Order Demo application, every table contains the history column `CREATED_BY`, `CREATION_DATE`, `LAST_UPDATED_BY`, `LAST_UPDATED_DATE`, and `OBJECT_VERSION_ID`, as shown in Figure 2–2.

Figure 2–2 History Columns for Tables in FOD Schema



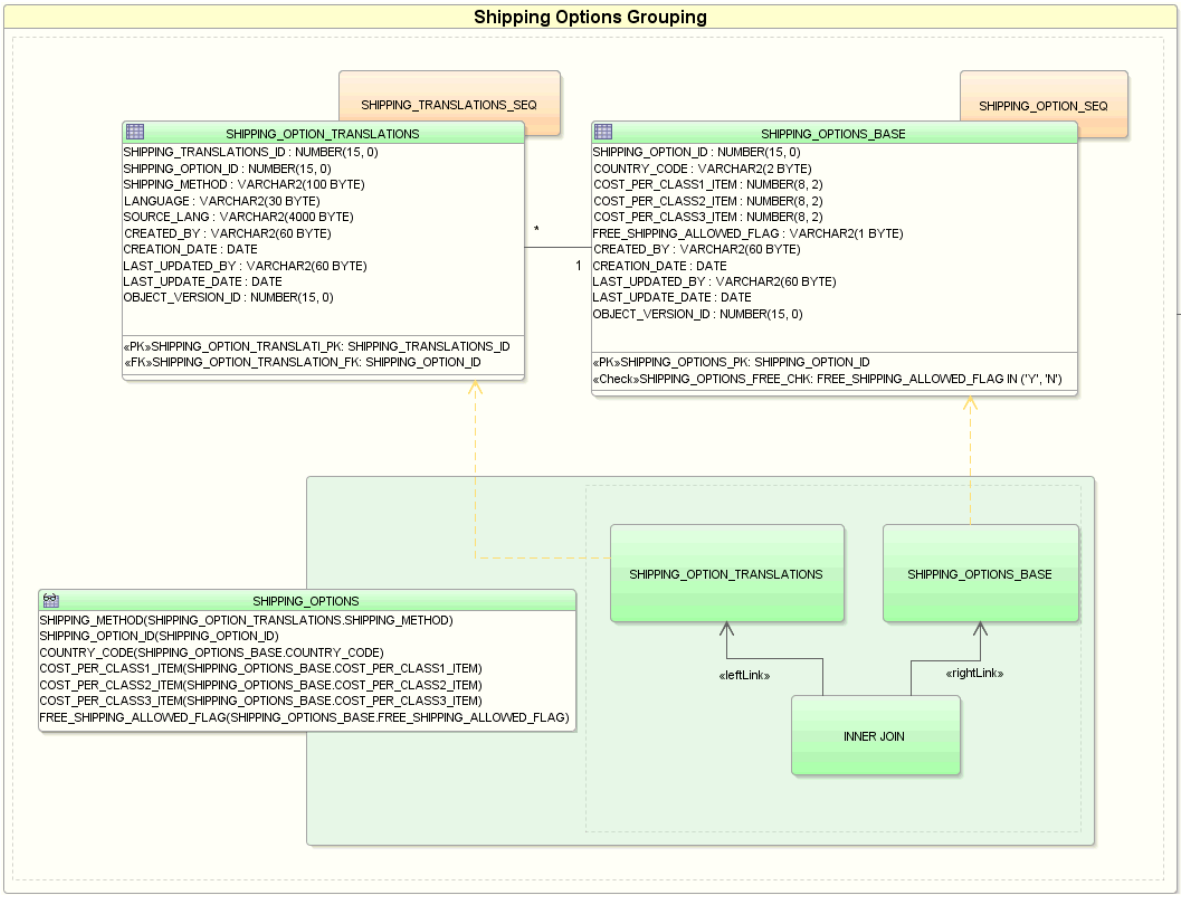
2.2.3.1 Translation Support in the Fusion Order Demo Schema

To support localization of the Fusion Order Demo, the AVAILABLE_LANGUAGES table lists all available languages. In this table, only one row will have the DEFAULT_FLAG set to Y corresponding to the current user's language.

Translations exist for the following base tables: PRODUCTS_BASE (PRODUCT_TRANSLATIONS), PRODUCT_CATEGORIES_BASE (CATEGORY_TRANSLATIONS), SHIPPING_OPTIONS_BASE (SHIPPING_OPTION_TRANSLATIONS), MEMBERSHIPS_BASE (MEMBERSHIP_TRANSLATIONS) and DISCOUNTS_BASE (DISCOUNT_TRANSLATIONS).

Taking the Shipping Options group, as shown in Figure 2–3: SHIPPING_OPTION_TRANSLATIONS is fully populated so that each product has one row for each language. The column LANGUAGE holds the translation language identifier. The entry itself may not yet be translated, in which case the SOURCE_LANGUAGE column holds the language that the entry is currently in. When a value has been translated, SOURCE_LANGUAGE and LANGUAGE will hold the same value. The PL/SQL package USER_CONTEXT_PKG creates the custom USERENV('CLIENT_INFO') variable that specifies the runtime locale used to pull the correct translations from SHIPPING_OPTION_TRANSLATIONS into the SHIPPING_OPTIONS view along with the SHIPPING_OPTIONS_BASE table data. Each order has one set of Shipping Options associated with it.

Figure 2–3 Shipping Options Grouping for the Fusion Order Demo Schema



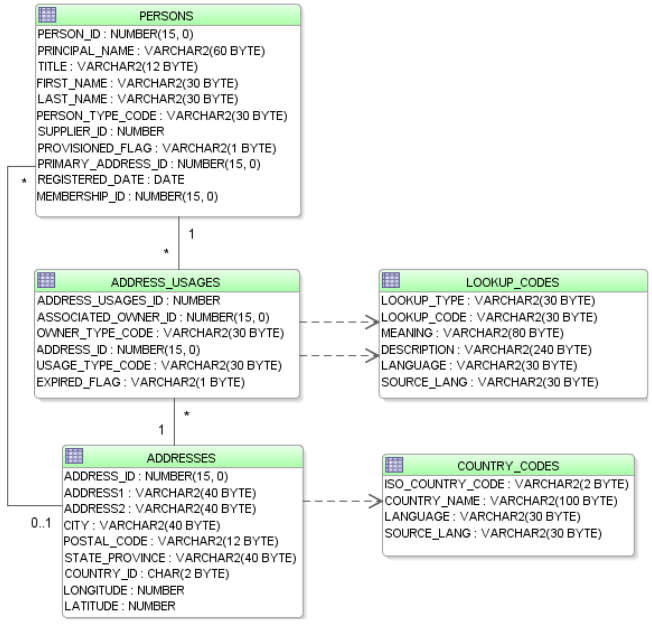
2.2.3.2 Lookup Tables in the Fusion Order Demo Schema

The code lookup table LOOKUP_CODES table contains codes that are used throughout the Fusion Order Demo application. For example, the PERSONS table contains the columns person_type_code, marital_status_code, and gender. These codes have corresponding rows in the LOOKUP_CODES table, discriminating on the lookup_type column. Foreign keys are not defined for these rows, but instead are enforced in the user interface by populating user interface components with LOOKUP_CODES values for a particular lookup type. For example, when creating a new registration (also known as a person) in the user interface, the values that can be used for the person_type_code are populated in a dropdown list from the lookup_code values with lookup_type=person_type_code.

The LOOKUP_CODES table also supports the localization of the user interface. The table uses a combined key of code and language (obtained from runtime locale or preference) to determine the code's meaning. Each code has an entry for each supported language, as described in Section 2.2.3.1, "Translation Support in the Fusion Order Demo Schema."

Using addresses as an example, as shown in Figure 2–4: PERSONS uses an intersection ADDRESS_USAGES to accommodate multiple address information. In addition ADDRESS_USAGES uses LOOKUP_CODES to store both OWNER_TYPE_CODE and USAGE_TYPE_CODE information, returning the MEANING (see table extract in Figure 2–5). ADDRESSES directly accesses COUNTRY_CODES to look up and use the COUNTRY_NAME associated with the COUNTRY_ID stored in ADDRESSES. The PERSONS table also directly stores PRIMARY_ADDRESS_ID by a direct lookup to ADDRESSES.

Figure 2–4 LOOKUP_CODES Usage in the Fusion Order Demo Schema



The correct translation is applied by using the LANGUAGE columns in both LOOKUP_CODES and COUNTRY_CODES with the runtime locale/preference.

Figure 2–5 LOOKUP_CODES Sample Data in the Fusion Order Demo Schema

LOOKUP_TYPE	LOOKUP_CODE	MEANING
OWNER_TYPE_CODE	CUST	Customer
OWNER_TYPE_CODE	SUPP	Supplier
OWNER_TYPE_CODE	STAFF	Staff
USAGE_TYPE_CODE	IN	Invoice Address
USAGE_TYPE_CODE	SH	Shipping Address

The lookup table `DEMO_OPTIONS` defines the various options within the Fusion Order Demo application that are switched on. It also caches general configuration information such as email addresses and phone numbers to use as overrides in this demonstration scenario (for example, where email addresses are fictitious). This table is reserved for future use.

2.3 Running the Fusion Order Demo Application StoreFront Module

The Fusion Order Demo application consists of a web user interface and a business service layer. Specifically, the following projects are part of the Fusion Order Demo application:

- `StoreFrontService`: Provides access to the storefront data and provides transaction support to update data for customer information and orders.
- `StoreFrontUI`: Provides web pages that the customer uses to browse the storefront, place orders, register on the site, view order information, and update the user profile.

You run the StoreFront module of the Fusion Order Demo application in JDeveloper by running the `home.jspx` page in the `StoreFrontUI` project. The `StoreFrontUI` project uses JavaServer Faces (JSF) as the view technology, and relies on the ADF Model layer to interact with ADF Business Components in the `StoreFrontService` project. To learn more about the Fusion Order Demo application and to understand its implementation details, see Section 2.5, "Taking a Look at the Fusion Order Demo Application."

A second module of the Fusion Order Demo application is available to process the orders that you place using the StoreFront module. For example, the WebLogic Fusion Order Demo module uses various internal and external applications, including a customer service application, a credit validation system, and both an internal vendor and external vendor. These composite services are the subject of another developer's guide and are not addressed in the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework*. For details about the WebLogic Fusion Order Demo module used to demonstrate the capabilities of Oracle SOA Suite, see the *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*.

You cannot run the WebLogic Fusion Order Demo module in JDeveloper. When you want to integrate the storefront portion with the composite services portion of the application, you must deploy the Fusion Order Demo application to a SOA-enabled Oracle WebLogic Server. Instructions to deploy any SOA web application to Oracle WebLogic Server, are addressed in the Fusion Order Demo home page on OTN at this link
`http://www.oracle.com/technetwork/developer-tools/jdev/index-095536.html`.

The easiest way to run the Fusion Order Demo application is to open only the StoreFront module in JDeveloper and run the `home.jspx` page in the `StoreFrontUI` project.

To run the StoreFront module of the Fusion Order Demo application:

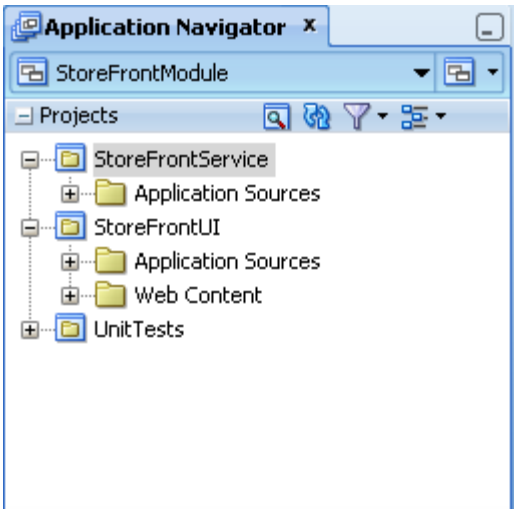
1. Open the application in JDeveloper:

   a. From the main menu, choose **File > Open**.

   b. Navigate to the location where you extracted the demo ZIP file to and select the **StoreFrontModule.jws** application workspace from the **StoreFrontModule** directory. Click **Open**.

   Figure 2–6 shows the Application Navigator after you open the file for the application workspace. For a description of each of the projects in the workspace, see Section 2.5, "Taking a Look at the Fusion Order Demo Application."

Figure 2–6 The Fusion Order Demo Projects in JDeveloper



2. In the Application Navigator, click the **Application Resources** accordion title to expand the panel.
3. In the Application Resources panel, expand the **Connections** and **Database** nodes.
4. Right-click **FOD** connection and choose **Properties**.
5. In the Edit Database Connection dialog, modify the connection information shown in Table 2–2 for your environment.

Table 2–2 Connection Properties Required to Run the Fusion Order Demo Application

Property	Description
Host Name	The host name for your database. For example: `localhost`
JDBC Port	The port for your database. For example: `1521`
SID	The SID of your database. For example: `ORCL` or `XE`

   Do not modify the user name and password **fod/fusion**. These must remain unchanged. Click **OK**.

6. In the Application Navigator, right-click **StoreFrontService** and choose **Rebuild**.

7. In the Application Navigator, right-click **StoreFrontUI** and choose **Run**.

   The Configure Default Domain dialog displays the first time you run the application and start a new domain in Integrated WebLogic Server. Use the dialog to define an administrator password for the new domain. Passwords you enter can be eight characters or more and must have a numeric character.

   The `home.jspx` page within the `StoreFrontUI` project is the default run target. When you run the default target, JDeveloper will launch the browser and display the Fusion Order Demo application home page.

Once the home page appears, you can browse the web site as an anonymous user, or you can choose to log in and place orders (may require registering as a customer first). Because the Fusion Order Demo application implements ADF Security to manage access to Oracle Application Development Framework (Oracle ADF) resources, only the authenticated user will be able to view orders in their cart. Table 2–3 shows the users who are authorized as members of the `fod-users` role to log into the Fusion Order Demo application.

> **Note:** The Fusion Order Demo application ships with predefined user data. The schema for the application defines different types of users including customer, supplier, and staff types. All users are members of the `fod-users` role and are authorized to log in. However, only `ngreenbe` is the user type `CUST` (customer). When you log in as any other user, you will need to register as a customer before you can place an order. These additional users were created to support roles in other modules of the Fusion Order Demo application.

Table 2–3 Supplied Users in the Fusion Order Demo Application

Username	Password	Application Role	Notes
`ngreenbe`	`welcome1`	`fod-users`	Can add items to cart, check out, and view order information. This is the only user who is preregistered as a customer in the StoreFront module of Fusion Order Demo.
`sking`	`welcome1`	`fod-users`, `fod-admin`	Can add items to cart, but must register as a customer to check out and view order information. This user also has administration privileges (`fod-admin`) in the MasterPriceList module of Fusion Order Demo.
`ahunold`	`welcome1`	`fod-users`, `fod-manager`	Can add items to cart, but must register as a customer to check out and view order information. This user also has read-only privileges (`fod-manager`) in the MasterPriceList module of Fusion Order Demo.
`pbrown` / `fdaviet`	`welcome1`	`fod-users`	Can add items to cart, but must register as a customer to check out and view order information. These users may be added to to other roles in a future version of Fusion Order Demo.

2.4 Running the Fusion Order Demo Standalone Applications

The Fusion Order Demo application includes a set of sample applications that allow you to investigate Oracle ADF functionality that does not appear in the StoreFront module. Collectively, these sample applications are referred to as standalone applications. The standalone sample applications appear in five application workspaces, each consisting of several projects, located in the `StandaloneExamples` folder where you extracted the demo ZIP file.

In general, almost all of the standalone applications demonstrate concepts of ADF Business Components and data model projects. References to these standalone applications appear throughout the chapters contained in Part II, "Building Your Business Services" and Part VII, "Advanced Topics" of this developer's guide. As you read sections this guide, you may want to run the corresponding standalone application to investigate the concepts further. For a brief description of each application workspace and links to the documentation, refer to the tables in Section 2.4.2 through Section 2.4.5.

2.4.1 How to Run the Standalone Applications

How you use JDeveloper to run a standalone application depends on the individual application. Some applications are set up to use the interactive testing tool JDeveloper provides for the ADF Business Components data model project (this tool is known as the Oracle ADF Model Tester). Other applications provide Java test clients (with file names like `TestClient`*`Xxx`*`.java`) that use the ADF Business Components API to execute queries and display results. In the case of the Oracle ADF Model Tester, you work entirely in the tool, which essentially provides a convenient user interface for interacting with business components. In the case of the Java clients, the program files output their results and print statements to the JDeveloper Log window.

Familiarize yourself with the following general procedures about how to run the standalone applications. The first procedure describes how to run an application with its provided test client. The second describes how to launch the Oracle ADF Model Tester on the data model project's ADF application module. Then read Section 2.4.2 through Section 2.4.5 for more details about the individual standalone applications.

Before you begin:

- In the Database Navigator, modify the connection information for the FOD database connection so it has the connection information for your environment, as described in Section 2.3, "Running the Fusion Order Demo Application StoreFront Module."
- Some of the standalone applications work with a modified version of the FOD schema. For standalone applications that require schema changes, the application's project will contain a SQL script that you must run within JDeveloper.

  Once you are through investigating a standalone application, you can use the script to back out the schema changes.

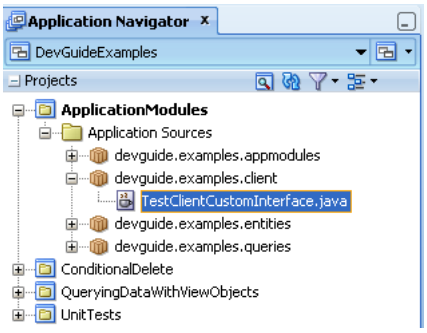
To run a standalone application from its provided test client:

1. Open the application in JDeveloper:
   a. From the main menu, choose **File > Open**.
   b. Navigate to the location where you extracted the demo ZIP file to, open the **StandaloneExamples** directory, then open the desired standalone application directory and then select the application workspace (**.jws**) and click **Open**.

2. In the Application Navigator, expand the project node and locate the test client (`.java`) file node. In some cases, the test client is added to a package located under the **Application Sources** node. In other cases, the **Resources** node contains the test client.

   For example, Figure 2–7 shows the expanded **ApplicationModules** node with the Java file node **TestClientCustomInterface.java** selected.

Figure 2–7 Test Client Selected in Application Navigator



3. Right-click the test client and choose **Run**.

   For the names and location of the test clients provided with the standalone applications, see the tables in Section 2.4.2 through Section 2.4.5.

   The Configure Default Domain dialog displays the first time you run your application and start a new domain in Integrated WebLogic Server. Use the dialog to define an administrator password for the new domain. Passwords you enter can be eight characters or more and must have a numeric character.

4. Examine the JDeveloper Log window for the test client's output.

   Refer to the referenced documentation for details about the expected results.

When the standalone application does not provide a test client to programmatically exercise the ADF Business Components API, you will use the interactive testing tool, known as the Oracle ADF Model Tester.

To run a standalone application in the Oracle ADF Model Tester:

1. Open the application in JDeveloper:

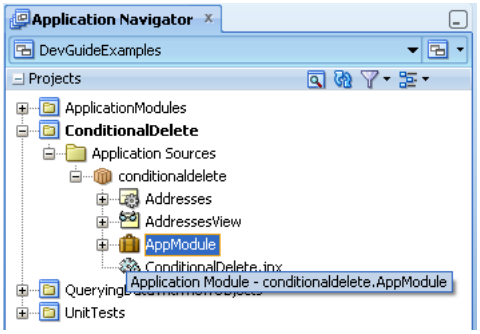
   a. From the main menu, choose **File > Open**.

   b. Navigate to the location where you extracted the demo ZIP file to, open the **StandaloneExamples** directory, then open the desired standalone application directory, and then select the application workspace (**.jws**) and click **Open**.

2. In the Application Navigator, expand the project node and locate the application module in a package under the **Application Sources** node.

   For example, Figure 2–8 shows the expanded **ConditionalDelete** project with the application module **AppModule** selected and a tooltip for the node displayed.

Figure 2–8 Application Module Node Selected in Application Navigator



3. Right-click the application module node and choose **Run**.

   For the names of the runnable application modules, see the tables in Section 2.4.2 through Section 2.4.5.

   The Configure Default Domain dialog displays the first time you run your application and start a new domain in Integrated WebLogic Server. Use the dialog to define an administrator password for the new domain. Passwords you enter can be eight characters or more and must have a numeric character.

4. Use the Oracle ADF Model Tester to interact with the view instances of the standalone application.

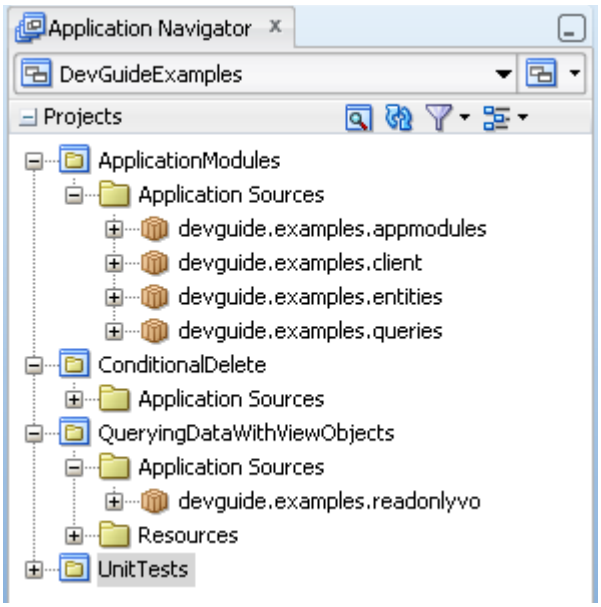
   Refer to the referenced documentation for details about the application. For details about using the tester to interact with the data model, see Section 6.3, "Testing View Object Instances Using the Oracle ADF Model Tester."

2.4.2 Standalone Applications in the DevGuideExamples Application Workspace

Two of the standalone applications in the application workspace `DevGuideExamples` use programmatic test clients to demonstrate concepts related to the ADF Business Components framework. The third application demonstrates framework functionality when you run the application in the Oracle ADF Model Tester.

Figure 2–9 shows the Application Navigator after you open the `DevGuideExamples` application workspace.

Figure 2–9 Runnable Applications in the DevGuideExamples Application Workspace



Note that the test clients for the `DevGuideExamples` standalone applications provide a good starting point for understanding how to exercise methods of the ADF Business Components API. They also make good samples for test clients that you may want to create to test business component queries in a data model project. For background on working with test clients, see Section 6.4, "Testing View Object Instances Programmatically."

Note: The ADF Business Components API is available when you need to generate custom classes to augment the default runtime behavior of the business components. For background about the ADF Business Components framework, see Section 3.5, "Overview of the ADF Business Components Implementation."

Table 2–4 describes the standalone applications in the `DevGuideExamples` application workspace. Examples from these applications appear throughout the chapters contained in Part II, "Building Your Business Services" of this guide.

Table 2–4 Standalone Applications in the DevGuideExamples Application Workspace

Project Name	Runnable Class or Project Target	Documentation
`ApplicationModule`	Run **TestClientCustomInterface.java** in the **devguide.examples.client** package under the **Application Sources** node. Exercises custom methods of **StoreFrontService** application module's client interface and prints to the JDeveloper Log window to indicate the results.	For details about the test client, see Section 9.10.1, "How to Work Programmatically with an Application Module's Client Interface." For details about the methods of the client interface, see the examples in Section 4.14, "Working Programmatically with Entity Objects and Associations."
`ConditionalDelete`	Launch the Oracle ADF Model Tester on **AppModule** under the **Application Sources** node. Overrides a method in the generated entity class that conditionally prevents deletion of entity rows. In the Oracle ADF Model Tester, click **Delete the Current Row** and observe the exception statement. Then, click **Insert a New Row** and delete the new row.	For a description of overriding the `remove()` method that you generate in the entity implementation class file, see Section 8.12, "Conditionally Preventing an Entity Row from Being Removed."

Table 2–4 (Cont.) Standalone Applications in the DevGuideExamples Application Workspace

Project Name	Runnable Class or Project Target	Documentation
QueryingDataWithViewObjects	Run **TestClient.java** under the **Resources** node of the Application Navigator. Programmatically iterates over the PersonList view instance using methods of the Business Components API RowIterator interface and prints to the JDeveloper Log window.	For details about iterating over a collection, see Section 6.4.5, "How to Count the Number of Rows in a Row Set." For details about how to create test clients, see Section 6.4, "Testing View Object Instances Programmatically."
	Run **TestClient2.java** under the **Resources** node of the Application Navigator. Programmatically iterates over the PersonList view instance, accesses the detail collection OrdersToShipToCustomers using a view link accessor attribute, and prints to the JDeveloper Log window.	For details about iterating over a detail collection, see Section 5.6.6, "How to Access the Detail Collection Using the View Link Accessor." For more details about the test client, see Section 6.4.6, "How to Access a Detail Collection Using the View Link Accessor."
	Run **TestClient3.java** under the **Resources** node of the Application Navigator. Programmatically iterates over the PersonList view instance using a strongly-typed PersonsRow interface and prints to the JDeveloper Log window.	For details about iterating over a collection using the view row accessor attribute, see Section 42.3.1.3, "Exposing View Row Accessors to Clients."
	Run **TestClientBindVars.java** under the **Resources** node of the Application Navigator. Programmatically sets the WHERE clause for the PersonList view instance using bind variables to filter the collection and prints to the JDeveloper Log window.	For details about setting bind variables, see Section 5.10.6, "How to Set Existing Bind Variable Values at Runtime." For more details about the test client, see Section 5.10.5, "How to Add a WHERE Clause with Named Bind Variables at Runtime."
	Run **TestClientViewCriteria.java** under the **Resources** node of the Application Navigator. Programmatically sets a view criteria for the PersonList view instance to filter the collection and prints to the JDeveloper Log window.	For details about the ADF Business Component's view criteria API, see Section 5.11.9, "What You May Need to Know About the View Criteria API." For more details about the test client, see Section 5.11.7, "How to Create View Criteria Programmatically."

2.4.3 Standalone Applications in the AdvancedExamples Application Workspace

The standalone applications assembled in the application workspace AdvancedExamples demonstrate advanced concepts that apply to the entire ADF Business Components framework.

Figure 2–10 shows the Application Navigator after you open the AdvancedExamples application workspace.

Figure 2–10 Runnable Applications in the AdvancedExamples Application Workspace

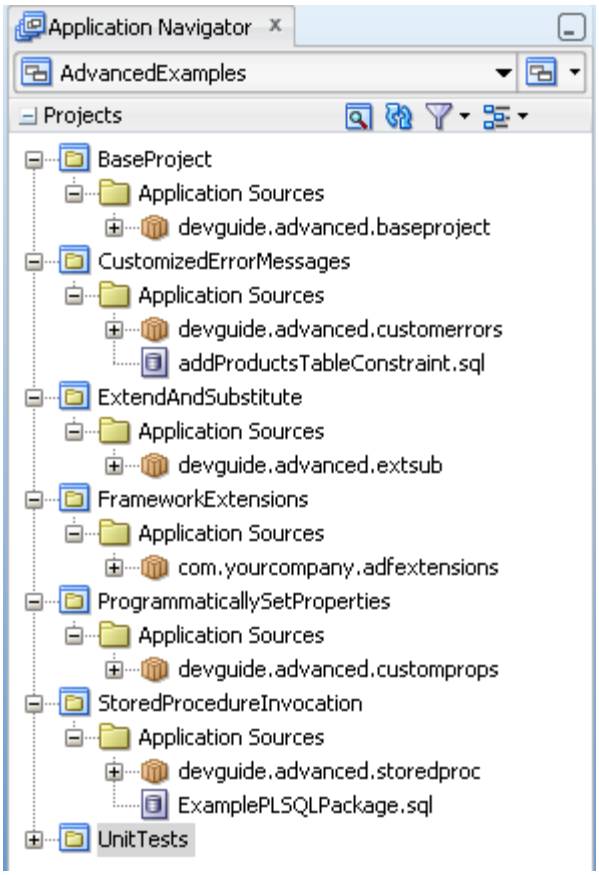


Table 2–5 describes the standalone applications in the `AdvancedExamples` application workspace. Examples from this application workspace are described in Chapter 12, "Extending Business Components Functionality."

Table 2–5 Standalone Applications in the AdvancedExamples Application Workspace

Project Name	Runnable Class or Project Target	Documentation
`BaseProject`	Run **TestClient.java** under the **Application Sources** node of the Application Navigator.	For details about how to extend business components to create a customized versions of the original, see Section 12.9, "Creating Extended Components Using Inheritance."
`CustomizedErrorMessages`	Run the **addProductsTableConstraint.sql** script under the **Application Sources** node of the Application Navigator against the `FOD` connection to set up the additional database objects required for the project. Launch the Oracle ADF Model Tester on **ProductModule** under the **Application Sources** node of the Application Navigator.	For details about how to provide an alternative message string for the builtin error codes in a custom message bundle, see Section 12.8, "Customizing Business Components Error Messages."
`ExtendAndSubstitute`	Not runnable. Programmatically iterates over the `PersonList` view instance using methods of the Business Components API `RowIterator` interface and prints to the JDeveloper Log window.	For details about how to substitute business components, see Section 12.10, "Substituting Extended Components in a Delivered Application."
`FrameworkExtensions`	Not runnable. Provides template class files that you can use to modify your own generated ADF Business Components classes.	For details about framework extensions, see Section 12.3, "Creating a Layer of Framework Extensions."
`ProgrammaticallySetProperties`	Run **TestClient.java** under the **Application Sources** node of the Application Navigator.	For details about how to communicate custom declarative information about business components to the generic code in framework extension classes, see Section 12.4, "Customizing Framework Behavior with Extension Classes."
`StoredProcedureInvocation`	Run the `ExampleSQLPackage.sql` script under the **Application Sources** node of the Application Navigator against the `FOD` connection to set up the additional database objects required for the project. Run **TestClient.java** under the **Application Sources** node of the Application Navigator.	For details about how to code custom Java classes for business components that invoke database stored procedures and functions, see Section 12.6, "Invoking Stored Procedures and Functions."

2.4.4 Standalone Applications in the AdvancedEntityExamples Application Workspace

The standalone applications assembled in the application workspace `AdvancedEntityExamples` demonstrate advanced concepts that apply to ADF Business Components entity objects.

Figure 2–11 shows the Application Navigator after you open the `AdvancedEntityExamples` application workspace.

Figure 2–11 Runnable Applications in the AdvancedEntityExamples Application Workspace

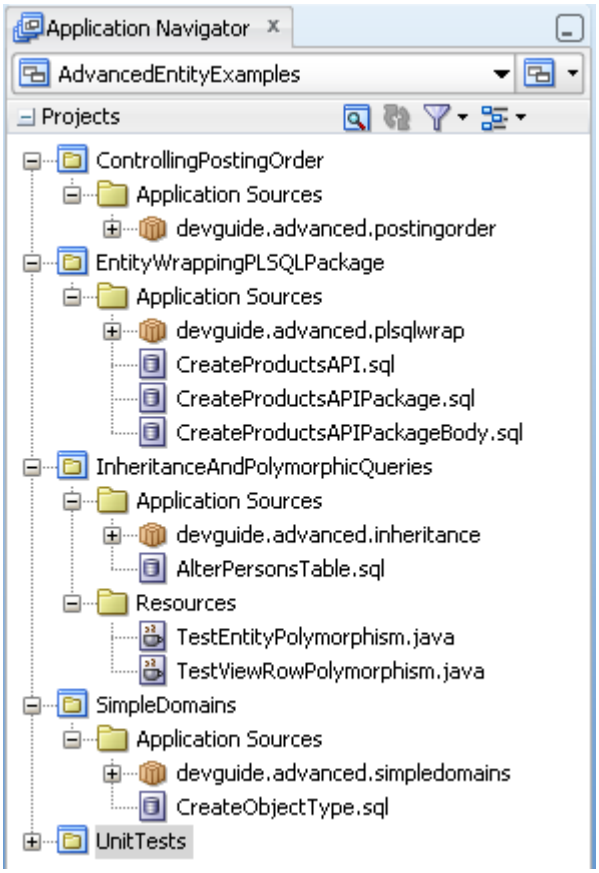


Table 2–6 describes the standalone applications in the `AdvancedEntityExamples` application workspace. Examples from this application workspace are described in Chapter 4, "Creating a Business Domain Layer Using Entity Objects."

Table 2–6 Standalone Applications in the AdvancedEntityExamples Application Workspace

Project Name	Runnable Class or Project Target	Documentation
`ControllingPostingOrder`	Launch the Oracle ADF Model Tester on **ProductsModule** under the **Application Sources** node of the Application Navigator.	For details about controlling the posting order resulting from DML operations to save changes to a number of related entity objects, see Section 4.14.7, "How to Control Entity Posting Order to Prevent Constraint Violations."
`EntityWrappingPLSQLPackage`	Run the **CreateProducts*XXX*.sql** scripts under the **Application Sources** node of the Application Navigator against the `FOD` connection to set up the additional database objects required for the project. Launch the Oracle ADF Model Tester on **ProductsModule** under the **Application Sources** node of the Application Navigator.	For details about overriding the default DML processing event for an entity object to invoke methods in a PL/SQL API PL/SQL package that encapsulates insert, update, and delete access to an underlying table, see Section 4.17, "Basing an Entity Object on a PL/SQL Package API."
`InheritanceAndPolymorphicQueries`	Run the **AlterPersonsTable.sql** script under the **Application Sources** node of the Application Navigator against the `FOD` connection to set up the additional database objects required for the project. Run **TestEntityPolymorphism.java** under the **Resources** node of the Application Navigator. Also, run **TestViewRowPolymorphism.java** under the **Resources** node of the Application Navigator.	For details about creating an entity object inheritance hierarchy, see Section 4.19, "Using Inheritance in Your Business Domain Layer."
`SimpleDomains`	Run the **CreateObjectType.sql** script under the **Application Sources** node of the Application Navigator against the `FOD` connection to set up the additional database objects required for the project. Launch the Oracle ADF Model Tester on **PersonModule** under the **Application Sources** node of the Application Navigator.	For details about creating custom data types, see Section 4.15, "Creating Custom, Validated Data Types Using Domains."

2.4.5 Standalone Applications in the AdvancedViewObjectExamples Application Workspace

The standalone applications assembled in the application workspace `AdvancedViewObjectExamples` demonstrate advanced concepts that apply to ADF Business Components view objects.

Figure 2–12 shows the Application Navigator after you open the `AdvancedViewObjectExamples` application workspace.

Figure 2–12 Runnable Applications in the AdvancedViewObjectExamples Application Workspace

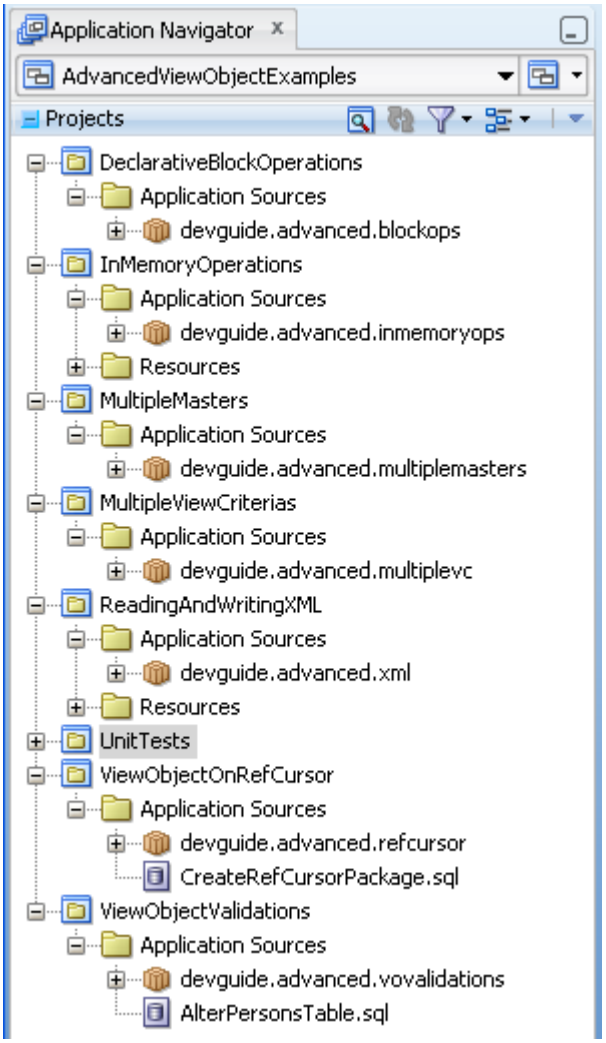


Table 2–4 describes the standalone applications in the `AdvancedViewObjectExamples` application workspace. Examples from this application workspace are described in Chapter 42, "Advanced View Object Techniques."

Table 2–7 Standalone Applications in the AdvancedViewObjectExamples Application Workspace

Project Name	Runnable Class or Project Target	Documentation
`DeclarativeBlockOperations`	Launch the Oracle ADF Model Tester on **AppModule** under the **Application Sources** node of the Application Navigator.	For details about how to use custom metadata properties to control insert, update, or delete on a view object, see Section 42.11, "Declaratively Preventing Insert, Update, and Delete."
`InMemoryOperations`	Launch the Oracle ADF Model Tester on **AppModule** under the **Application Sources** node of the Application Navigator. Illustrates using the in-memory sorting and filtering functionality from the client side using methods on the interfaces in the `oracle.jbo` package.	For details about how to use view objects to perform in-memory searches and sorting to avoid unnecessary trips to the database, see Section 42.5, "Performing In-Memory Sorting and Filtering of Row Sets."
`MultipleMasters`	Launch the Oracle ADF Model Tester on **AppModule** under the **Application Sources** node of the Application Navigator.	For details about creating a view object with multiple updatable entities to support creating new rows, see Section 42.9, "Creating a View Object with Multiple Updatable Entities."
`MultipleViewCriterias`	Run **TestClientMultipleViewCriteria.java** under the **Application Sources** node of the Application Navigator.	For details about how to programmatically filter query results, see Section 42.4, "Working Programmatically with Multiple Named View Criteria."
`ReadingAndWritingXML`	Run **TestClientReadXML.java** under the **Resources** node of the Application Navigator. Then run **TestClientWriteXML.java** under the **Resources** node of the Application Navigator.	For details about how to produce XML from queried data, see Section 42.7, "Reading and Writing XML."
`ViewObjectOnRefCursor`	Run the **CreateRefCursorPackage.sql** script under the **Application Sources** node of the Application Navigator against the `FOD` connection to set up the additional database objects required for the project, Launch the Oracle ADF Model Tester on **OrdersModule** under the **Application Sources** node of the Application Navigator.	For details about how to use PL/SQL to open a cursor to iterate through the results of a query, see Section 42.8.4, "How to Create a View Object on a REF CURSOR."

2.4.6 Standalone Applications in the AdvancedUIExamples Application Workspaces

The standalone applications assembled in the application workspaces in the `AdvancedUIExamples` folder demonstrate advanced concepts that apply to databound ADF Faces components.

Figure 2–13 shows the Application Navigator after you open the application workspaces in the `AdvancedUIExamples` folder.

Figure 2–13 Runnable Applications in the AdvancedUIExamples Application Workspaces

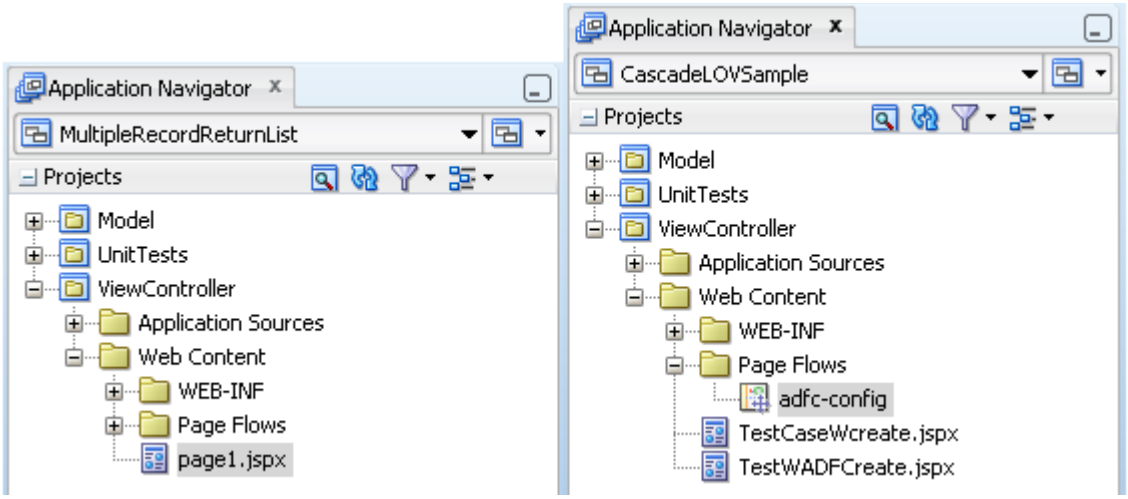


Table 2–4 describes the standalone applications in the `AdvancedUIExamples` folder.

Table 2–8 Standalone Applications in the AdvancedUIExamples Application Workspaces

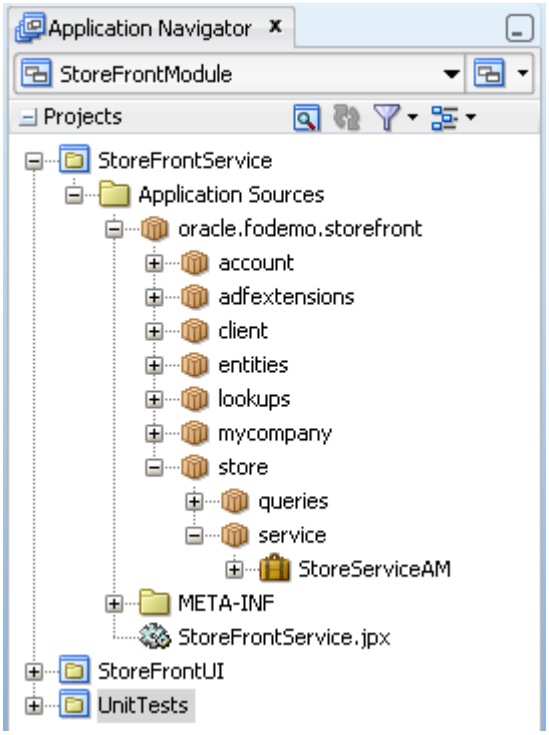
Workspace Name	Runnable Project Target	Documentation
`MultipleRecordReturnList`	Run the **page1.jspx** node under the **Web Content** node in the ViewController project displayed in the Application Navigator.	For details about how to use `selectOneChoice` components to return multiple records, see Section 30.3, "Creating a Selection List."
`CascadeLOVSample`	Run the **adfc-config** node under the **Page Flows** node in the ViewController project displayed in the Application Navigator.	For details about how to use cascading LOVs, see Section 30.2, "Creating List of Values (LOV) Components."

2.5 Taking a Look at the Fusion Order Demo Application

Once you have opened the projects in JDeveloper, you can then begin to review the artifacts within each project. The development environment for the Fusion Order Demo application is divided into two projects: the `StoreFrontService` project and the `StoreFrontUI` project.

The `StoreFrontService` project contains the classes that allow the product data to be displayed in the web application. Figure 2–14 shows the `StoreFrontService` project and its associated directories.

Figure 2–14 The StoreFrontService Project

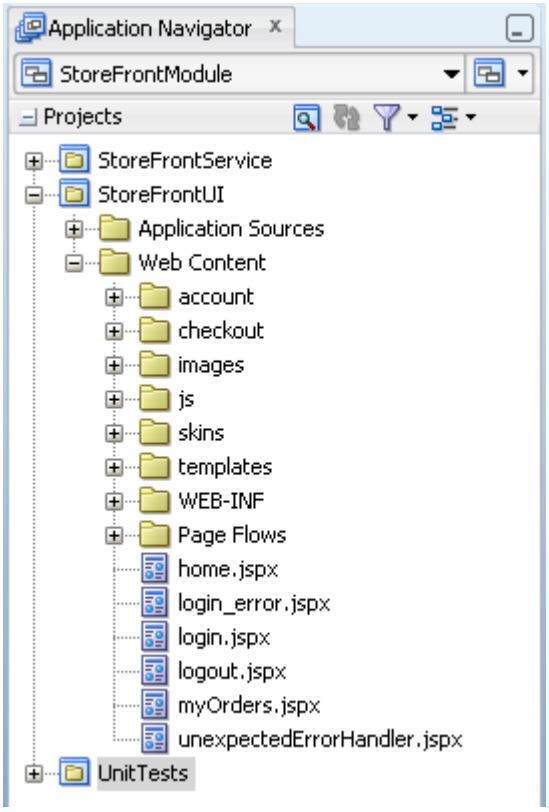


The `StoreFrontService` project contains the following directories:

- `Application Sources`: Contains the files used to access the product data. Included are the metadata files used by Oracle Application Development Framework (Oracle ADF) to bind the data to the view.
- `META-INF`: Contains a file used in deployment.

The `StoreFrontUI` project contains the files for the web interface, including the backing beans, deployment files, and JSPX files. Figure 2–15 shows the `StoreFrontUI` project and its associated directories.

Figure 2–15 The StoreFrontUI Project



The `StoreFrontUI` project contains the following directories:

- `Application Sources`: Contains the code used by the web client, including the managed and backing beans, property files used for internationalization, and the metadata used by Oracle ADF to display bound data.
- `Web Content`: Contains the web files, including the JSP files, images, skin files, deployment descriptors, and libraries.

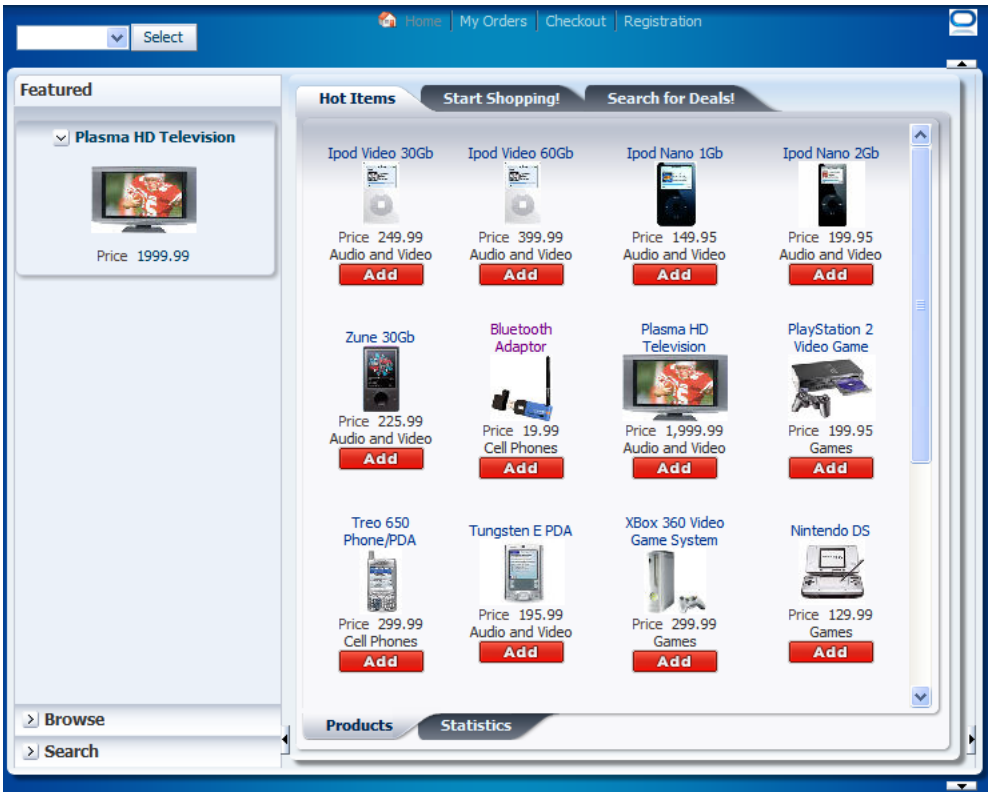
2.5.1 Anonymous Browsing

You start the Fusion Order Demo application by running the `home.jspx` page in the `StoreFrontUI` project. For details about running the application using the default target, `home.jspx` page, see Section 2.3, "Running the Fusion Order Demo Application StoreFront Module."

When you enter the storefront site, the site is available for anonymous browsing. You can use this page to browse the catalog of products without logging into an account. The initial view shows the featured products that the site wishes to promote and gives you access to the full catalog of items. Products are presented as images along with the name of the product. Page regions divide the product catalog area from other features that the site offers.

Figure 2–16 shows the home page.

Figure 2–16 Home Page with Multiple Regions



Where to Find Implementation Details

Following are the sections of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework* that describe how to create a databound web page:

- Providing the structure for the web page

  The home page separates features of the site into regions that are implemented using a combination of ADF Faces templates and JavaServer Faces (JSF) page fragments. ADF Faces templates and the fragments allow you to add ADF databound components. For information about the steps you perform before adding databound user interface components to a web page, see Section 24.1, "About Developing a Web Application with ADF Faces."

- Displaying information on a web page

  To support data binding, the featured items on the tabbed region of the home page use EL (Expression Language) expressions to reference ADF data control usages in the declarative ADF page definition file. The page definition file, which JDeveloper creates for you when you work with the Data Controls panel to drag and drop databound ADF Faces components, is unique to each web page or page fragment. The ADF data control usages enable queries to the database and ultimately work with the JSF runtime to render the databound ADF Faces components, such as the ADF Faces image component used to display images from the `PRODUCT_IMAGES` table. For information about creating a databound web page that references the ADF page definition file, see Section 26.1, "About Creating a Basic Databound Page."

- Managing entry points to the application

The home page is supported by an ADF unbounded task flow. In general, the Fusion web application relies on this ADF Controller feature to define entry points to the application. The unbounded task flow for the entire home page and its page fragments describes view activities for displaying the home page, displaying the orders page, displaying the register user page, and it defines a task flow reference to manage the checkout process. JDeveloper helps you to create the task flow with visual design elements that you drag and drop from the Component Palette. When you create an unbounded task flow, the elements allow you to identify how to pass control from one activity in the application to the next. Because a view activity must be associated with a web page or page fragment, JDeveloper allows you also to create the files for the web page or fragment directly from the task flow diagram. The process of creating a task flow adds declarative definitions to an ADF task flow configuration file. The resulting diagram lets you work with a visual control flow map of the pages and referenced task flows for your application. For more information about specifying the entry points to the application using an ADF unbounded task flows, see Section 18.1, "About ADF Task Flows."

2.5.1.1 Viewing Product Details

To view detailed product information, you can click the product name link for any product in the home page. The product information is laid out with collapsing nodes organized by categories.

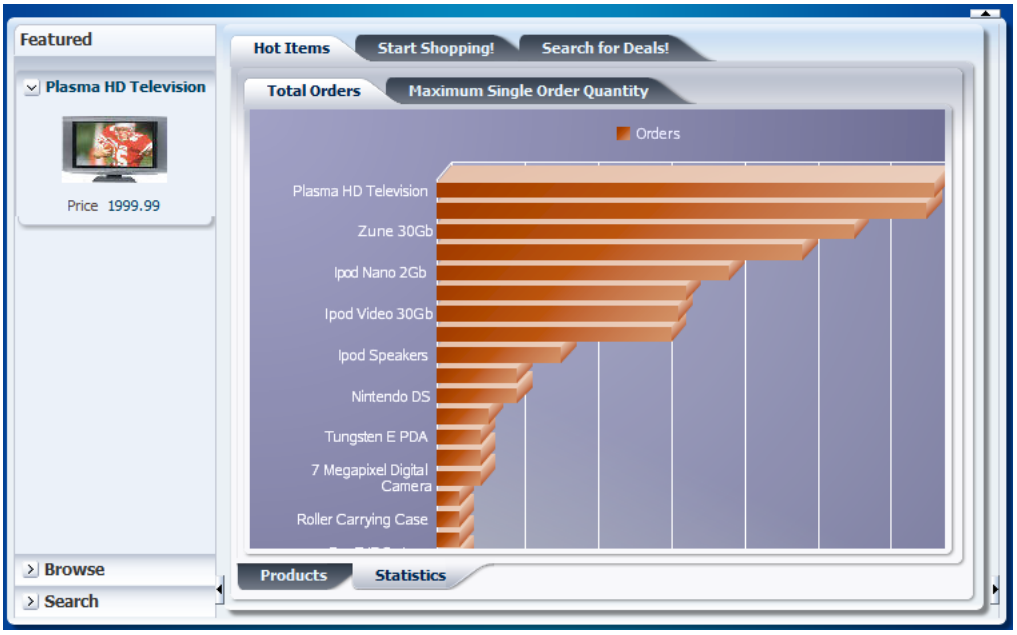
Figure 2–17 shows the detail dialog that you can view for a product.

Figure 2–17 Home Page - Product Details Popup

You can also select the **Statistics** subtab on the home page to view a graphical representation of the number of orders that customers have placed for the featured items. To present the information so that quantities are easily compared, the graph sorts the products by the number of items ordered, in descending order.

Figure 2–18 shows the bar graph used to display the featured products' current order details.

Figure 2–18 Home Page - Statistics for Featured Items



Where to Find Implementation Details

Following are the sections of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework* that describe how to develop the components used to support browsing product details:

- Triggering an action to display data

  To display data from the data model, user interface components in the web page are bound to ADF Model layer binding objects using JSF Expression Language (EL) expressions. For example, when the user clicks on a link to display an informational dialog, the JSF runtime evaluates the EL expression for the dialog's UI component and pulls the value from the ADF Model layer. At design time, when you work with the Data Controls panel to drag an attribute for an item of a data collection into you web page, and then choose an ADF Faces component to display the value, JDeveloper creates all the necessary JSF tag and binding code needed to display and update the associated data. For more information about the Data Controls panel and the declarative binding experience, see Section 13.1, "About ADF Data Binding."

- Displaying data in graphical format

  JDeveloper allows you to create databound components declaratively for your JSF pages, meaning you can design most aspects of your pages without needing to look at the code. By dragging and dropping items from the Data Controls panel, JDeveloper declaratively binds ADF Faces UI components and ADF Data Visualization graph components to attributes on a data control using an ADF

binding. For more information, see Section 33.1, "About ADF Data Visualization Components."

2.5.1.2 Browsing the Product Catalog

To begin browsing, click the **Start Shopping** tab in the home page. This action changes the region of the page used to display details about featured products to a region that displays a product categories tree. You can collapse and expand the branch nodes of the tree to view the various product categories that make up the product catalog. The tree displays the product categories in alphabetical order, by category names. When you want to view all the products in a particular category, click its category node in the tree (for example, click **Electronics**, **Media**, or **Office**). The site refreshes the product information region to display the list of products organized as they appear in the database with an image and an accompanying description.

Figure 2–19 shows the home page with all the products in the **Electronics** category displayed.

Figure 2–19 Home Page - Product Categories View

Where to Find Implementation Details

Following are the sections of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework* that describe how to use tables and forms to display master-detail related objects:

- Dragging and dropping master and detail components

  You can create pages that display master-detail data using the Data Controls panel. The Data Controls panel displays master-detail related objects in a hierarchy that mirrors the one you defined in the data model for the ADF application module, where the detail objects are children of the master objects. All you have to do is drop the collections on the page and choose the type of component you want to use. For example, in the Fusion Order Demo application, the page `home.jspx` displays the master list of product categories in an `af:tree` component and displays the detail list of products in an `af:table` component. For more information about the data model, see Section 3.4, "Overview of the

Oracle ADF Active Data Model." For more information about various types of pages that display master-detail related data, see Section 29.1, "About Displaying Master-Detail Data."

- Sorting data that displays in tables

  When you create an ADF Faces table component you bind the table to the complete collection or to a range of data objects from the collection. The specific components that display the data in the columns are then bound to the attributes of the collection. The iterator binding handles displaying the correct data for each object, while the table component handles displaying each object in a row. You can set the **Sort** property for any column when you want the iterator to perform an order-by query to determine the order. You can also specify an `ORDER BY` clause for the query that the view object in the data model project defines. For more information about binding table components to a collection, see Section 27.1, "About Creating ADF Databound Tables." For more information about creating queries that sort data in the data model, see Section 5.2, "Populating View Object Rows from a Single Database Table."

2.5.1.3 Searching for Products

To search the product catalog, you have several choices. You can begin either by clicking the disclosure icon (a **+** symbol) on the **Search** tab on the panel accordion or by clicking the **Search for Deals** tab in the main region. When you click either of these, the home page displays both regions at once to allow you to enter a search criteria and view the search results. You use the **Search** tab on the accordion panel to perform a simple keyword search against the attributes common to all products, such as product names or product descriptions. When you select the attribute to search on from the dropdown list, the panel renders a search field using an appropriate input component to accept the search criteria. For example, in the case of the default searchable attribute **ProductId**, where a numeric value is expected, the search field uses a spinbox (the ADF Faces component `inputNumberSpinBox`) to return the product ID.

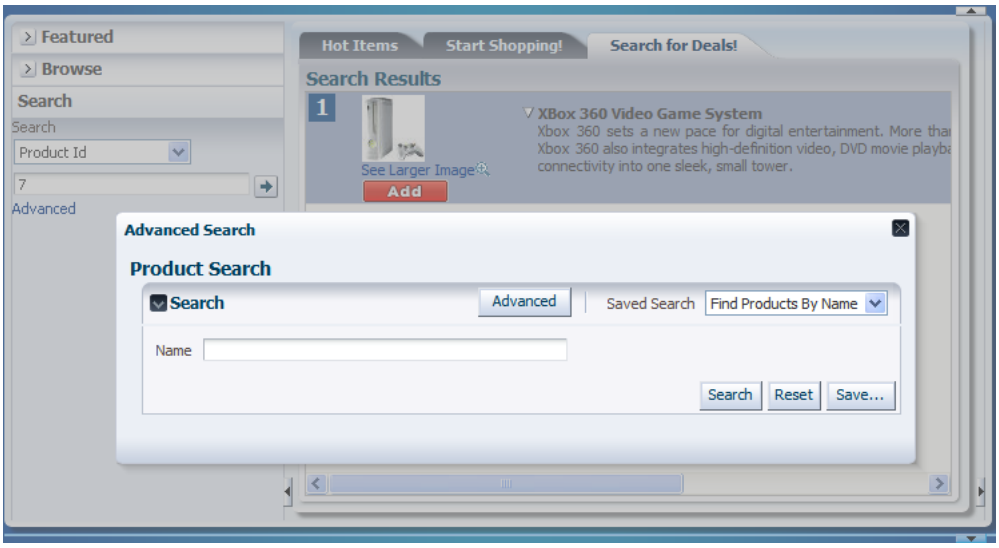
Figure 2–20 shows the home page with the search results returned for the product with an ID equal to 7.

Figure 2–20 Home Page - Search View

As an alternative to entering a simple search, you can use the advanced search feature to define and save search criteria based on any combination of searchable fields that you select for the product. Click the **Advanced** link to open the Advanced Search dialog. Developer-defined saved searches like **Find Products By Name** appear in the **Saved Search** dropdown list.

Figure 2–21 shows the Advanced Search dialog with a single search criteria, **Name**, that the **Find Products By Name** saved search defines.

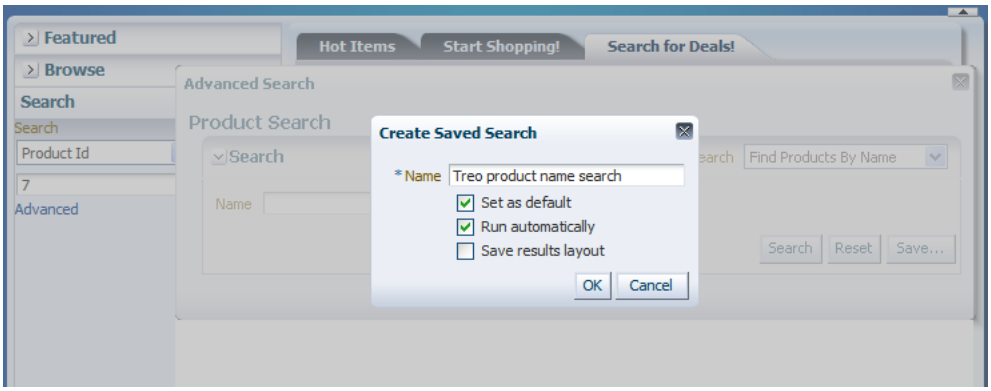
Figure 2–21 Home Page - Advanced Search Dialog



In addition to the developer-defined saved searches available in the Advanced Search dialog, the end user can create saved searches that will persist for the duration of their session. Enter the product search criteria in the Advanced Search dialog, then click the **Save** button to open the Create Saved Search dialog.

Figure 2–22 shows the Create Saved Search dialog that you use to specify how you want to save the search criteria you entered in the Advanced Search dialog. You can name the search, for example, **Treo product name search**, so that it will display in the **Saved Search** dropdown list of the Advanced Search dialog.

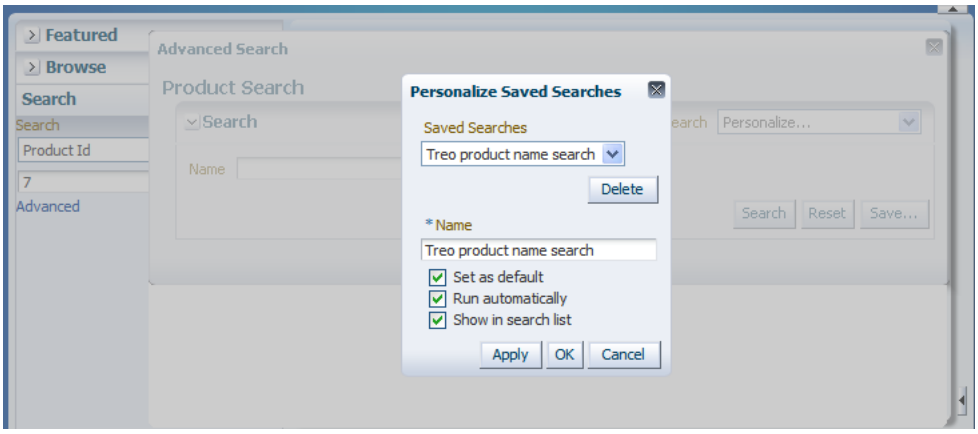
Figure 2–22 Home Page - Advanced Search Dialog - Saved Searches Option



You can also manage your saved searches by selecting the **Personalize** function in the **Saved Search** dropdown list of the Advanced Search dialog.

Figure 2–23 shows the Personalize Saved Search dialog for the **Find Products By Name** search, with **Show in Search List** enabled so that it will appear in the **Saved Search** dropdown list. Note that because this search is not a user-defined saved search, the personalization options appear disabled.

Figure 2–23 Home Page - Advanced Search Dialog - Personalization Option



Where to Find Implementation Details

Following are the sections of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework* that describe how to define queries and create query search forms:

- Defining the query for the search form to display

  A query is associated with an ADF Business Components view object that you create for the data model project to define a particular query against the database. In particular, a query component is the visual representation of the view criteria defined on that view object. If there are multiple view criteria defined, each of the view criteria can be selected from the **Saved Search** dropdown list. These saved searches are created at design time by the developer. For example, in the Fusion Order Demo application, the `ProductsVO` view object defines two view criteria. When the query associated with that view object is run, both view criteria are available for selection. For more information, see Section 31.1, "About Creating Search Forms."

- Creating a quick search form

  A quick query search form has one search criteria field with a dropdown list of the available searchable attributes from the associated data collection. By default, the searchable attributes are all the attributes in the associated view object. You can exclude attributes by setting the attribute's **Display** control hint to **Hide** in the view object. The user can search against the selected attribute or search against all the displayed attributes. The search criteria field type will automatically match the type of its corresponding attribute type. For more information, see Section 31.4, "Creating Quick Query Search Forms."

- Creating a search form

  You create a query search form by dropping a named view criteria item from the Data Controls panel onto a page. You have a choice of dropping only a search panel, dropping a search panel with a results table, or dropping a search panel with a tree table. For more information, see Section 31.2, "Creating Query Search Forms."

- Displaying the results of a query search

  Normally, you would drop a query search panel with the results table or tree table. JDeveloper will automatically wire up the results table or tree table with the query panel. If you drop a query panel by itself and want a separate results component, you can set the query component's `resultComponentId` attribute to

the relative expression of the results component. For example, in the Fusion Order Demo application, the page `home.jspx` displays an `af:table` with the ID `searchT` and the results ID of the advanced search dialog is assigned this ID. For more information, see Section 31.2.3, "How to Create a Query Search Form and Add a Results Component Later."

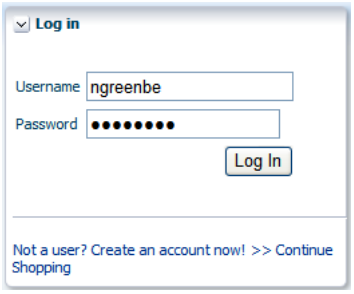
2.5.2 The Login Process

Until you attempt to access secure resources in the storefront site, you are free to browse the product catalog and update the shopping cart as an anonymous user. However, when you click the **My Orders** or **Checkout** links that appear at the top of the home page, you will be challenged by the web container running the site to supply login credentials. The site requires that you enter a valid user name and password before it completes your request to display the linked page.

Note: The Fusion Order Demo application supports the new customer registration process, but that user is not added to the security implementation. Thus, you must use a predefined customer's user name and password to log in, as shown in Table 2–3.

Figure 2–24 shows the login page fragment that displays before you can view order details or purchase items from the store. For demonstration purposes, log in as a customer by entering `ngreenbe` and `welcome1` for the **Username** and **Password**, respectively.

Figure 2–24 Login Region



When you click the **Log In** button, the web container will compare your entries with the credential information stored in its identity store. If the web container is able to authenticate you (because you have entered the user name and password for a registered user), then the web container redirects to the web page specified by your link selection; otherwise, the site prompts you to create an account or to continue browsing as an unauthenticated user.

Where to Find Implementation Details

Following are the sections of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework* that describe how to secure Oracle ADF resources so that users are required to log in to access those resources:

- Enabling fine-grained security to secure Oracle ADF resources

  ADF Security is a framework that provides a security implementation that is based on Java Authentication and Authorization Service (JAAS). The Oracle ADF implementation of JAAS is role-based. You define these roles in JDeveloper and then make permission grants based on these roles to enable fine-grained security

for Oracle ADF resources. JDeveloper supports declaratively defining the Oracle ADF policy store for an ADF bounded task flow or individual web pages associated with their ADF page definition. For information about securing Oracle ADF resources, see Section 35.5, "Defining ADF Security Policies."

- Triggering dynamic user authentication

  When you use ADF Security, authentication is triggered automatically if the user is not yet authenticated and tries to access a page that is not granted to the `anonymous-role` role. After successfully logging in, another check will be done to verify if the authenticated user has view access to the requested page. For more information, see Section 35.3.5, "What You May Need to Know About ADF Authentication."

- Performing permission checking within the web page

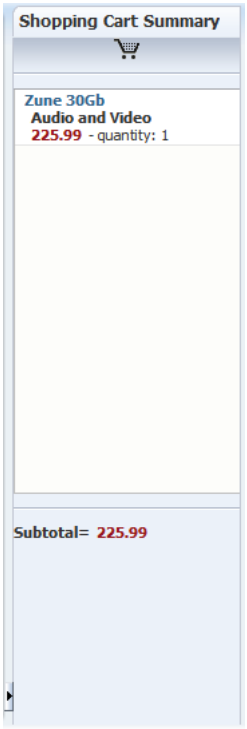
  At runtime, the security policy you define for ADF resources is enforced using standard JAAS permission authorization to determine the user's access rights. If your application requires it, you can use Expression Language (EL) to perform runtime permission checks within the web page to hide components that should not be visible to the user. For example, in the Fusion Order Demo application, the page `myOrders.jpx` uses an expression with the value `userGrantedPermission` to test the user's authorization privileges before displaying their account number. For more information, see Section 35.11.1, "Using Expression Language (EL) with ADF Security."

2.5.3 The Ordering Process

You begin the order process by browsing the product catalog. When you click **Add** next to a product, the site updates the shopping cart region to display the item.

Figure 2–25 shows the cart summary with a single item added. The summary shows a subtotal for the items that appear in the cart.

Figure 2–25 Home Page - Shopping Cart Summary

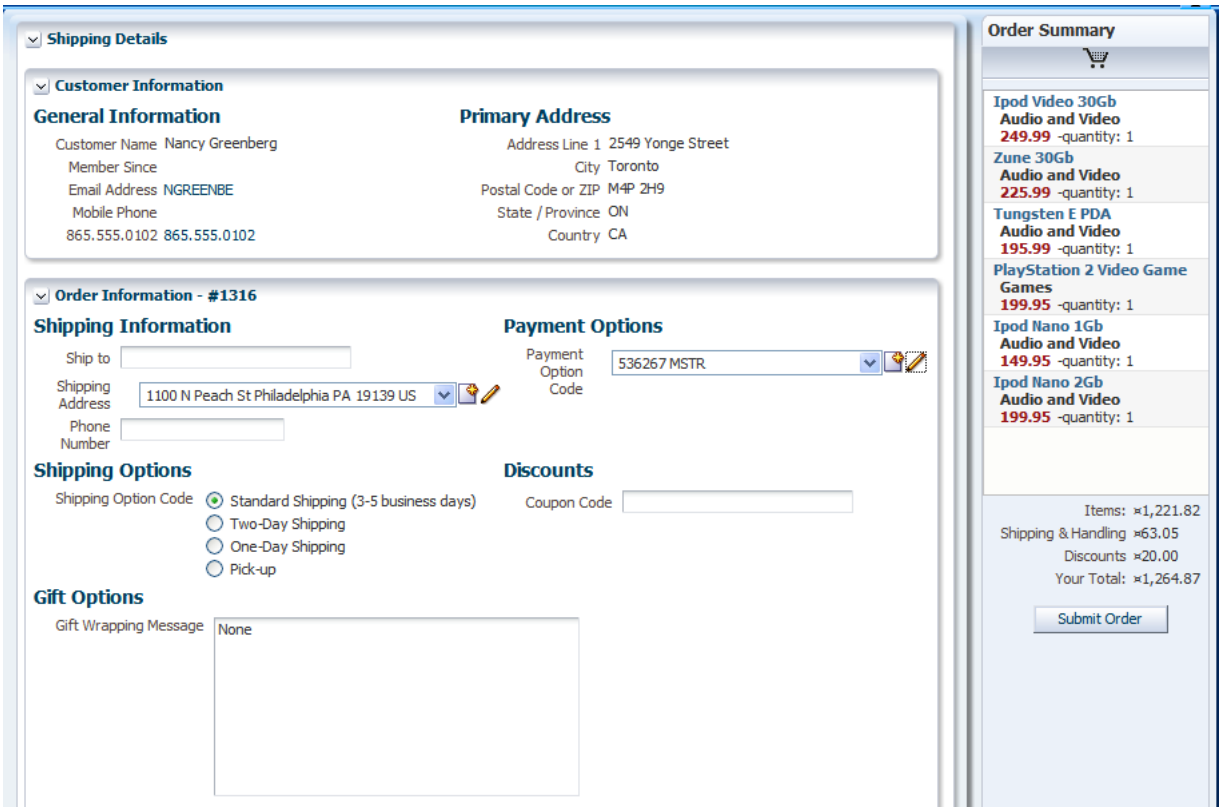


When you are satisfied with the items in the cart, you can complete the order by clicking the **Checkout** link at the top of the home page. To check out and complete the order, you must become an authenticated user, as described in Section 2.5.2, "The Login Process."

After you log in, the site displays the checkout page with order details, such as the name and address of the user you registered as. The order is identified by an **Order Information** number that is generated at runtime and assigned to the order. An Order Summary region displays the order items that comprise the new order. This region is similar to the cart summary on the home page, except that it adds the cost of shipping and deducts any discounts that apply to the order to calculate the total purchase amount.

Figure 2–26 shows the checkout page with an order comprising four order items.

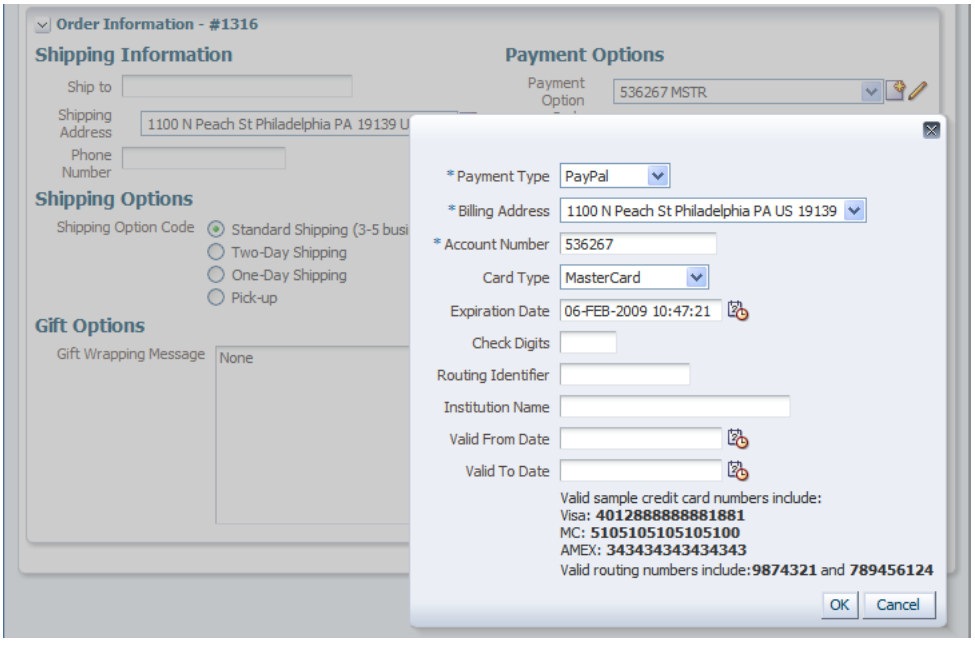
Figure 2–26 Checkout Page - Order Details Form



You can use the checkout page to customize details of the order information. For example, click the **Edit** icon next to the **Payment Option Code** field to display and edit payment funding information for the order.

Figure 2–27 shows the detail dialog for the **Payment Option Code** field.

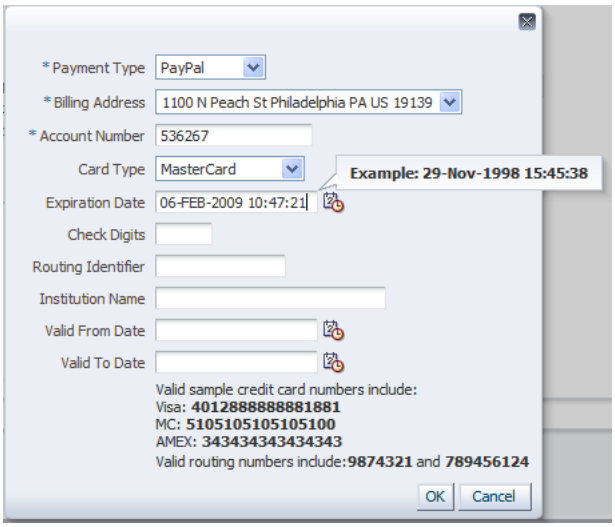
Figure 2–27 Checkout Page - Payment Option Detail Dialog



Many of the fields of the payment options dialog offer user interface hints that guide you to enter specific information.

Figure 2–28 shows an example of a date entry (**06-FEB-2009 10:47:21**) that the format mask (dd-MMM-yyyy hh:mm:ss) defines for the **Expiration Date** field.

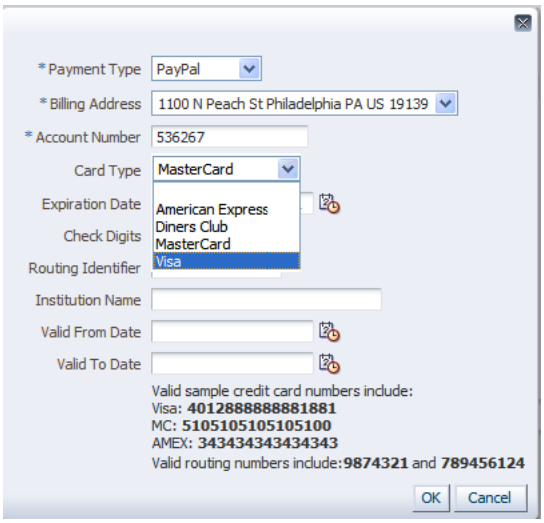
Figure 2–28 Checkout Page - Payment Options Detail Dialog - Date Format Mask



The **Card Type** field displays a dropdown list that allows you to select from a valid list of credit card types.

Figure 2–29 displays the list of values for the **Card Type** field.

Figure 2–29 Checkout Page - Payment Options Detail Dialog - List of Values (LOV) Choice List



If you close the payment options dialog and click the **Submit Order** button in the checkout page, the purchase order is created and sent into a process flow.

After you place an order using the StoreFront module, a second module of the Fusion Order Demo application is available to process the order. For details about the WebLogic Fusion Order Demo module used to demonstrate the capabilities of Oracle

SOA Suite, see *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*. For information about running this portion of the Fusion Order Demo application, see Section 2.3, "Running the Fusion Order Demo Application StoreFront Module."

Where to Find Implementation Details

Following are the sections of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework* that describe how to develop forms like the ones used in the order checkout process:

- Creating a databound edit form

  When you want to create a basic form that collects values from the user, instead of having to drop individual attributes, JDeveloper allows you to drop all attributes for an object at once as an input form. You can create forms that display values, forms that allow users to edit values, and forms that collect values. For example, in the Fusion Order Demo application, the checkout page `orderSummary.jspx` displays one form to display user information and another form to collect shipping information for the user's order. For more information, see Section 26.6, "Creating an Input Form."

- Defining format masks for input forms

  Format masks help ensure the user supplies attribute values in the required format. To facilitate this task, ADF Business Components provides declarative support known as control hints for attributes in the data model project. For example, in the Fusion Order Demo application, the attribute for the `CustomerPaymentOptionVO` view object used to assign the user's credit card expiration date is configured with a format mask hint and enforced in the Payment Options page fragment `paymentOptionsDetails.jsff`. For information on defining format masks for input form components, see Section 5.13, "Defining UI Hints for View Objects."

- Defining a list of values for selection lists

  Input forms displayed in the user interface can utilize databound ADF Faces selection components to display a list of values (LOV) for individual attributes of the data collection. To facilitate this common design task, ADF Business Components provides declarative support to specify the LOV usage for attributes in the data model project. For example, in the Fusion Order Demo application, the three `af:selectOneChoice` components displayed in the Payment Options page fragment `paymentOptionsDetails.jsff` are bound to LOV-enabled attributes configured for the `CustomerPaymentOptionVO` view object. For more information about configuring attributes for LOV usage, see Section 5.12, "Working with List of Values (LOV) in View Object Attributes."

- Keeping track of transient session information

  When you create a data model project that maps attributes to columns in an underlying table, your ADF view objects can include transient attributes that display calculated values (for example, using Java or Groovy expressions) or that are value holders. For example, in the Fusion Order Demo application, the order summary page `orderSummary.jspx` displays the value of the `InvoiceTotal` attribute calculated by the expression defined on the `OrderVO` view object. For more information about defining transient attributes in the data model project, see Section 4.11, "Adding Transient and Calculated Attributes to an Entity Object."

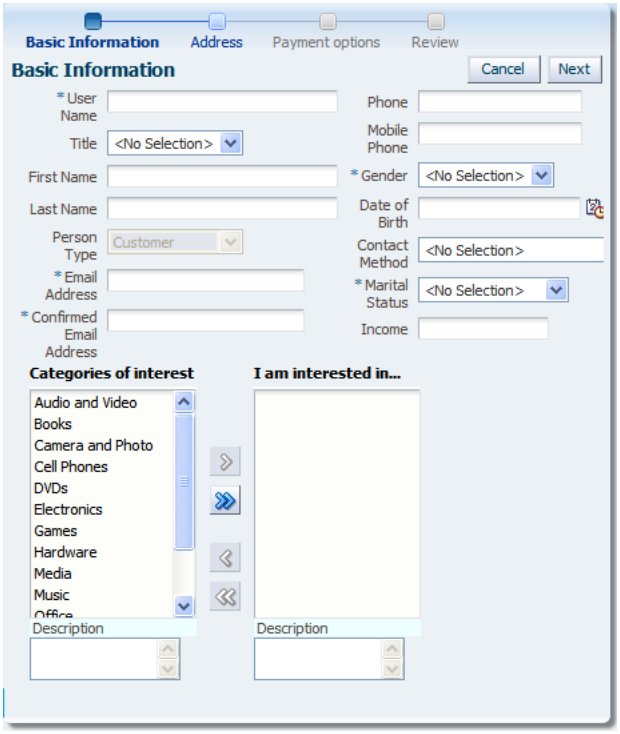
2.5.4 The Customer Registration Process

The site requires that you become an authenticated user before you can display the checkout page. To make it possible for new customers complete the order process, the site needs to provide a way to guide users through customer registration. To begin, click the registration link on the home page and then click **Register as a customer**.

Customer registration progresses in steps, with one screen dedicated to each step. To represent the progression of these steps, the registration page displays a series of train stops labelled **Basic Information**, **Address**, **Payment Options**, and **Review**. To navigate the customer registration process, you can click certain train stops or you can click the **Next** button.

Figure 2–30 shows the first screen in the customer registration process. The **Basic Information** stop of the train is enabled and selected to identify it as the current stop. Notice that the next train stop icon, **Address**, is enabled but not highlighted, while the **Payment options** and **Review** train stop icons appear disabled and grayed out. Together, these train stops signify that you must complete the activity in a sequential flow.

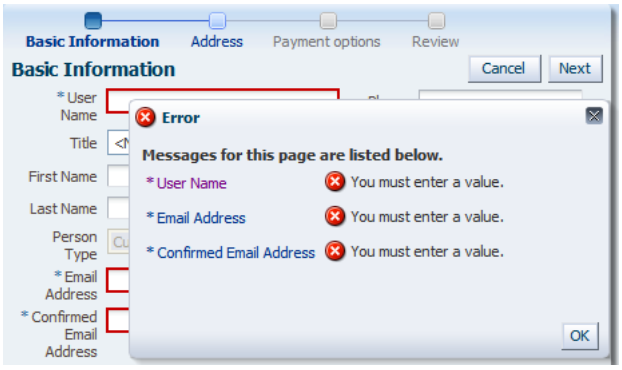
Figure 2–30 Customer Registration Page - Basic Information Form



Before you enter any information into the Basic Information form, click the **Address** train stop. The page displays an error dialog to inform you that specific fields require a value before you can progress to the next step.

Figure 2–31 shows the error dialog with messages stating that the Basic Information form requires a user name and an email address.

Figure 2–31 Customer Registration Page - Basic Information Form with Validation Error Popup

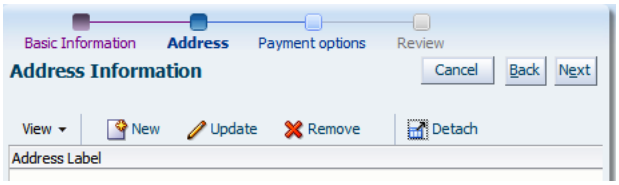


Click **OK** to dismiss the error dialog. Then enter a user name and email address. Be sure to confirm the email address in the form.

Again, click **Next** to progress to the next task. This time, the site should display the Address screen with icon buttons that you can select to create a new address record in the database (or, in the case of an existing customer, to update an existing address record).

Figure 2–32 shows the Address screen with one column for **Address Label** and no row information. Because you are entering information as a new customer, no address record currently exists, so no rows are available to display below these columns.

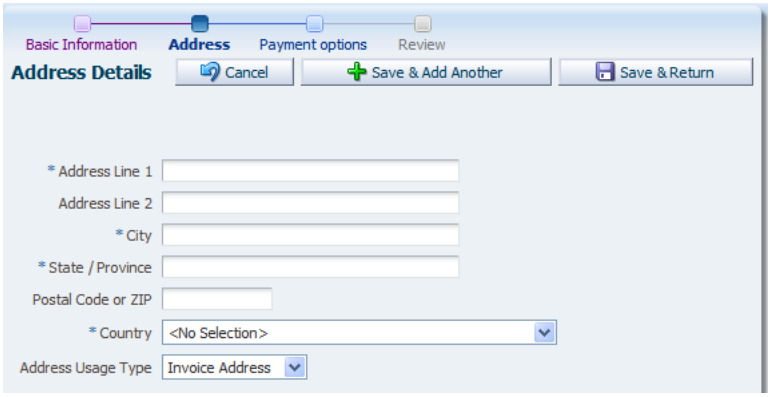
Figure 2–32 Customer Registration Page - Address Input Task



Click **New**. The registration page changes to display an address input form and the current train stop remains on **Address**.

Figure 2–33 shows the empty address input form.

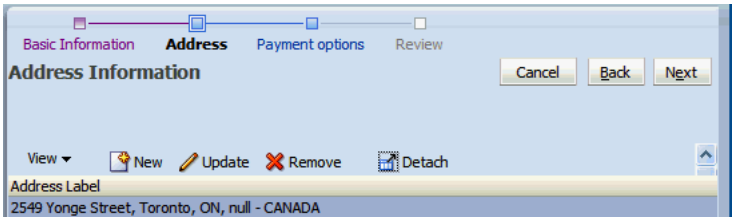
Figure 2–33 Customer Registration Page - Address Input Form



For the fields with an asterisk symbol (*), enter the address information specified. The asterisk symbol indicates that the value is required. Note that you must also select a country from the dropdown list since this information is required by the database. Then click **Save & Return** to create the new address record in the database.

Figure 2–34 shows the Address screen with the row information for the new address record.

Figure 2–34 Customer Registration Page - Address Record Complete



This concludes the tour of the Fusion Order Demo application.

Where to Find Implementation Details

Following are the sections of the *Oracle Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework* that describe how to use complex components like the ADF Faces Train component used in the registration process:

- Grouping activities using a bounded task flow

  Every Fusion web application contains an unbounded task flow, which contains the entry points to the application. The application can then call bounded task flows from activities within the unbounded task flow. For example, in the Fusion Order Demo application, the bounded task flow `checkout-task-flow` controls the flow of the checkout process and calls another bounded task flow `customer-registration-task-flow` to control the flow of the registration process. For information about the bounded task flow, see Section 18.1, "About ADF Task Flows."

- Displaying a progression of related activities using a train component that you associate with a bounded task flow

  You configure train stops based on activities that you select in an ADF bounded task flow and then you add the `af:train` component to your JSF pages. For example, in the Fusion Order Demo application, the bounded task flow `customer-registration-task-flow` defines four train stops for the page fragments `basicInformation.jsff`, `defineAddresses.jsff`, `paymentOptions.jsff`, and `reviewCustomerInfo.jsff`. The page `register.jspx` displays the fragments and each page fragment displays the train component bound to the activities that define the four stops. For information about the bounded task flow and how you can use it to define train stops, see Section 22.7.1, "Creating a Task Flow as a Train."

- Requiring values to complete an input form

  The input form displays attributes of a data collection that you drop from the Data Controls panel. You can set the `required` property of individual components in the form to control whether an attribute value is mandatory. For details about how to customize the `required` property, see Section 26.2, "Using Attributes to Create Text Fields." Alternatively, you can set a display control hint property directly on the attribute where it is defined by an ADF Business Components entity object. The entity object is a data model component that represents a row from a specific

table in the database and that simplifies modifying its associated attributes. For details about using control hints to make an attribute mandatory, see Section 4.10, "Setting Attribute Properties."

Part II

Building Your Business Services

Part II contains the following chapters:

- Chapter 3, "Getting Started with ADF Business Components"
- Chapter 4, "Creating a Business Domain Layer Using Entity Objects"
- Chapter 5, "Defining SQL Queries Using View Objects"
- Chapter 6, "Testing View Instance Queries"
- Chapter 7, "Defining Validation and Business Rules Declaratively"
- Chapter 8, "Implementing Validation and Business Rules Programmatically"
- Chapter 9, "Implementing Business Services with Application Modules"
- Chapter 10, "Sharing Application Module View Instances"
- Chapter 11, "Integrating Service-Enabled Application Modules"
- Chapter 12, "Extending Business Components Functionality"

3

Getting Started with ADF Business Components

This chapter provides an overview of ADF Business Components, including a description of the key features they provide for building your business services.

This chapter includes the following sections:

- Section 3.1, "About ADF Business Components"
- Section 3.2, "Comparison to Familiar 4GL Tools"
- Section 3.3, "Overview of Design Time Facilities"
- Section 3.4, "Overview of the Oracle ADF Active Data Model"
- Section 3.5, "Overview of the ADF Business Components Implementation"
- Section 3.6, "Overview of Groovy Scripting Language Support"

3.1 About ADF Business Components

ADF Business Components and JDeveloper simplify the development, delivery, and customization of business applications for the Java EE platform. With ADF Business Components, developers aren't required to write the application infrastructure code required by the typical Java EE application to:

- Connect to the database
- Retrieve data
- Lock database records
- Manage transactions

ADF Business Components addresses these tasks through its library of reusable software components and through the supporting design time facilities in JDeveloper. Most importantly, developers save time using ADF Business Components since the JDeveloper design time makes typical development tasks entirely declarative. In particular, JDeveloper supports declarative development with ADF Business Components to:

- Author and test business logic in components which automatically integrate with databases
- Reuse business logic through multiple SQL-based views of data, supporting different application tasks
- Access and update the views from browser, desktop, mobile, and web service clients

- Customize application functionality in layers without requiring modification of the delivered application

The goal of ADF Business Components is to make the business services developer more productive.

3.1.1 ADF Business Components Features

ADF Business Components provides a foundation of Java classes that allow your business-tier application components to leverage the functionality provided in the following areas:

Simplifying Data Access

- Design a data model for client displays, including only necessary data
- Include master-detail hierarchies of any complexity as part of the data model
- Implement end-user Query-by-Example data filtering without code
- Automatically coordinate data model changes with business services layer
- Automatically validate and save any changes to the database

Enforcing Business Domain Validation and Business Logic

- Declaratively enforce required fields, primary key uniqueness, data precision-scale, and foreign key references
- Easily capture and enforce both simple and complex business rules, programmatically or declaratively, with multilevel validation support
- Navigate relationships between business domain objects and enforce constraints related to compound components

Supporting Sophisticated UIs with Multipage Units of Work

- Automatically reflect changes made by business service application logic in the user interface
- Retrieve reference information from related tables, and automatically maintain the information when the user changes foreign-key values
- Simplify multistep web-based business transactions with automatic web-tier state management
- Handle images, video, sound, and documents without having to use code
- Synchronize pending data changes across multiple views of data
- Consistently apply prompts, tooltips, format masks, and error messages in any application
- Define custom metadata for any business components to support metadata-driven user interface or application functionality
- Add dynamic attributes at runtime to simplify per-row state management

Implementing High-Performance Service-Oriented Architecture

- Support highly functional web service interfaces for business integration without writing code
- Enforce best-practice interface-based programming style
- Simplify application security with automatic JAAS integration and audit maintenance

- "Write once, run anywhere": use the same business service as plain Java class, EJB session bean, or web service

Streamlining Application Customization

- Extend component functionality after delivery without modifying source code
- Globally substitute delivered components with extended ones without modifying the application
- Deliver application upgrades without losing or having to reapply downstream customizations manually

3.1.2 ADF Business Components Core Objects

ADF Business Components implements the business service through the following set of cooperating components:

- Entity object

  An *entity object* represents a row in a database table and simplifies modifying its data by handling all data manipulation language (DML) operations for you. It can encapsulate business logic for the row to ensure that your business rules are consistently enforced. You associate an entity object with others to reflect relationships in the underlying database schema to create a layer of business domain objects to reuse in multiple applications.

- View object

  A *view object* represents a SQL query. You use the full power of the familiar SQL language to join, filter, sort, and aggregate data into exactly the shape required by the end-user task. This includes the ability to link a view object with others to create master-detail hierarchies of any complexity. When end users modify data in the user interface, your view objects collaborate with entity objects to consistently validate and save the changes.

- Application module

  An *application module* is the transactional component that UI clients use to work with application data. It defines an updatable data model and top-level procedures and functions (called *service methods*) related to a logical unit of work related to an end-user task.

While the base components handle all the common cases through built-in behavior, customization is always possible and the default behavior provided by the base components can be easily overridden or augmented.

3.2 Comparison to Familiar 4GL Tools

ADF Business Components provides components that implement functionality similar to that offered by enterprise 4GL tools. Several key components in ADF Business Components have concepts you may be familiar with in other 4GL tools.

3.2.1 Familiar Concepts for Oracle Forms Developers

ADF Business Components implements all of the data-centric aspects of the familiar Oracle Forms runtime functionality, but in a way that is independent of the user interface. In Oracle Forms, each *form* contains both visual objects (like canvases, windows, alerts, and LOVs), as well as nonvisual objects (like data blocks, relations, and record groups). Individual data block *items* have both visual properties like `Foreground Color` and `Bevel`, as well as nonvisual properties like `Data Type` and

`Maximum Length`. Even the different event-handling triggers that Forms defines fall into visual and nonvisual categories. For example, it's clear that triggers like `WHEN-BUTTON-PRESSED` and `WHEN-MOUSE-CLICKED` are visual in nature, relating to the front-end UI, while triggers like `WHEN-VALIDATE-ITEM` and `ON-INSERT` are more related to the backend data processing. While merging visual and nonvisual aspects definitely simplifies the learning curve, the flip side is that it can complicate reuse. With a cleaner separation of UI-related and data-related elements, it would be easier to redesign the user interface without disturbing backend business logic and easier to repurpose back-end business logic in multiple different forms.

In order to imagine this separation of UI and data, consider reducing a form as you know it to only its nonvisual, data-related aspects. This reduces the form to a container of data blocks, relations, and record groups. This container would continue to provide a database connection for the data blocks to share and would be responsible for coordinating transaction commits or rollbacks. Of course, you could still use the *nonvisual* validation and transactional triggers to augment or change the default data-processing behavior as well. This nonvisual object you are considering is a kind of a "smart data model" or a generic application module, with data and business logic, but no user interface elements. The goal of separating this application module from anything visual is to allow any kind of user interface you need in the future to use it as a data service.

Focus a moment on the role the data blocks would play in this application module. They would query rows of data from the database using SQL, coordinate master/detail relationships with other data blocks, validate user data entry with `WHEN-VALIDATE-RECORD` and `WHEN-VALIDATE-ITEM` triggers, and communicate valid user changes back to the database with `INSERT`, `UPDATE`, and `DELETE` statements when you commit the data service's transaction.

Experience tells you that you need to filter, join, order, and group data for your end-users in a variety of ways to suit the many different tasks. On the other hand, the validation rules that you apply to your business domain data remain basically the same over time. Given these observations, it would be genuinely useful to write business entity validation exactly once, and leverage it consistently anywhere that data is manipulated by users in your applications.

Enabling this flexibility requires further "factoring" of your data block functionality. You need one kind of "SQL query" object to represent each of the many different views of data your application requires, and you need another kind of "business entity" object to enforce business rules and communicate changes to your base table in a consistent way. By splitting things like this, you can have multiple "view objects" with specific SQL queries that present the same business data yet each working with the same underlying "entity object."

Oracle Application Development Framework (Oracle ADF) addresses the UI/data split by providing ready-to-use Java components that implement typical Forms functionality. Responsibilities between the querying and entity-related functions are cleanly separated, resulting in better reuse.

3.2.1.1 Similarities Between the Application Module and a "Headless" Form Module

The application module component is the "data portion" of the form. The application module is a smart data service containing a data model of master-detail-related queries that your client interface needs to work with. It also provides a transaction and database connection used by the components it contains. It can contain form-level procedures and functions, referred to as service methods, that are encapsulated within the service implementation. You can decide which of these procedures and functions should be private and which ones should be public.

3.2.1.2 Similarities Between the Entity Object and a Forms Record Manager

The entity object component implements the "validation and database changes" portion of the data block functionality. In the Forms runtime, this duty is performed by the record manager. The record manager is responsible for keeping track of which of the rows in the data block have changed, for firing the block-level and item-level validation triggers when appropriate, and for coordinating the saving of changes to the database. This is exactly what an entity object does for you. The entity object is a component that represents your business domain entity through an underlying database table. The entity object gives you a single place to encapsulate business logic related to validation, defaulting, and database modification behavior for that business object.

3.2.1.3 Similarities Between the View Object and a Data Block

The `ViewObject` component performs the "data retrieval" portion of the data block functionality. Each view object encapsulates a SQL query, and at runtime each one manages its own query result set. If you connect two or more view objects in master-detail relationships, that coordination is handled automatically. While defining a view object, you can link any of its query columns to underlying entity objects. By capturing this information, the view object and entity object can cooperate automatically for you at runtime to enforce your domain business logic, regardless of the "shape" of the business data required by the user's task.

3.2.2 Familiar Concepts for PeopleTools Developers

If you have developed solutions in the past with PeopleTools, you are familiar with the PeopleTools component structure. ADF Business Components implement the data access functionality you are familiar with from PeopleTools.

3.2.2.1 Similarities Between the Application Module and a "Headless" Component

Oracle ADF adheres to an MVC pattern and separates the model from the view. Pages, which you are familiar with in the PeopleTools Component, are defined in the view layer, using standard technologies like JSF and ADF Faces components for web-based applications or Swing for desktop-fidelity client displays.

The ADF application module defines the data structure, just like the PeopleTools Component Buffer does. By defining master-detail relationships between ADF query components that produce row sets of data, you ensure that any application module that works with the data can reuse the natural hierarchy as required, similar to the scroll levels in the Component Buffer.

Similar to the Component Interface you are familiar with, the application module is a service object that provides access to standard methods, as well as additional developer-defined business logic. In order to present a "headless" data service for a particular user interface, the Component Interface restricts a number of PeopleTools functions that are related to UI interaction. The application module is similar to the Component Interface in that it provides a "headless" data service, but in contrast it does not do this by wrapping a restricted view of an existing user interface. Instead, the application module is designed to deal exclusively with business logic and data access. Rather than building a Component Interface on top of the component, with ADF Business Components you first build the application module service that is independent of user interface, and then build one or more pages on top of this service to accomplish some end-user task in your application.

The application module is associated with a transaction object in the same way that the PeopleTools Component Buffer is. The application module also provides a database

connection for the components it contains. Any logic you associate today with the transaction as Component PeopleCode, in ADF Business Components you would define as logic on the application module.

Logic associated with records in the transaction, that today you write as Component Record PeopleCode or Component Record Field PeopleCode, should probably *not* be defined on the application module. ADF Business Components has view objects that allow for better re-use when the same record appears in different components.

In summary, PeopleTools uses the component for the container concept, whereas ADF Business Components uses the application module. That is where the similarity ends. Do not assume that all of your component code will migrate to an application module. First, understand the concept of the view object, which is the layer between the entity object and the application module. Then, decide which of your component code is suitable for an application module and which is suitable for view objects.

3.2.2.2 Similarities Between the Entity Object and a Record Definition

The entity object is the mapping to the underlying data structure, just like the PeopleTools Record Definition maps to the underlying table or view. You'll often create one entity object for each of the tables that you need to manipulate your application.

Similar to how you declare a set of valid values for fields like "Customer Status" using PeopleTools' translate values, in ADF Business Components you can add declarative validations to the individual attributes of an entity object. Any logic you associate with the record that applies throughout your applications, which today you write as Record PeopleCode or Record Field PeopleCode, can be defined in ADF Business Components on the entity object.

3.2.2.3 Similarities Between the View Object and a Row Set

Just like a PeopleTools row set, a view object can be populated by a SQL query. Unlike a row set, a view object definition can contain business logic.

Any logic which you would find in Component Record PeopleCode is a likely candidate to define on the view object. Component Record PeopleCode is directly tied to the component, but a view object can be associated with different application modules. Whereas you can use the same record definition in many PeopleTools components, Oracle ADF allows you to reuse the business logic across multiple applications.

The view object queries data in exactly the "shape" that is useful for the current application. Many view objects can be built on top of the same entity object.

You can define relationships between view objects to create master-detail structures, just as you find them in the scroll levels in the PeopleTools component.

3.2.3 Familiar Concepts for Siebel Tools Developers

If you have developed solutions in the past with Siebel Tools version 7.0 or earlier, you will find that ADF Business Components implements all of the familiar data access functionality you are familiar with, with numerous enhancements.

3.2.3.1 Similarities Between the entity Object and a Table Object

Like the Siebel Table object, the ADF entity object describes the physical characteristics of a single table, including column names and physical data types. Both objects contain sufficient information to generate the DDL (data definition language) statements to create the physical tables in the database. In ADF Business Components you define associations between entity objects to reflect the foreign keys present in the underlying

tables. These associations allow view object queries used by user interface pages to automatically join business information. ADF Business Components handles list of values (LOV) objects that you reference from data columns through a combination of declarative entity-level validation rules and view object attribute-level LOV definitions. You can also encapsulate other declarative or programmatic business logic with these entity object "table" handlers that is automatically reused in any view of the data you create.

3.2.3.2 Similarities Between the View Object and a Business Component

Like the Siebel Business Component, the ADF view object describes a logical mapping on top of the underlying physical table representation. Both the Siebel Business Component and the ADF view object allow you to provide logical field names, data, and calculated fields that match the needs of the user interface. As with the Siebel Business Component, with the ADF view object you can define view objects that join information from various underlying tables. The related ADF view link is similar to the Siebel Link object and allows you to define master-detail relationships. In ADF Business Components, your view object definitions can exploit the full power of the SQL language to shape the data as required by the user interface.

3.2.3.3 Similarities Between the Application Module and a Business Object

The Siebel Business Object lets you define a collection of business components. The ADF application module performs a similar task, allowing you to create a collection of master-detail view objects that act as a "data model" for a set of related user interface pages. In addition, the application module provides a transaction and database connection context for this group of data views. You can make multiple requests to objects obtained from the application module and these participate in the same transaction.

3.2.4 Familiar Functionality for ADO.NET Developers

If you have developed solutions in the past with Visual Studio 2003 or 2005, you are familiar with using the ADO.NET framework for data access. ADF Business Components implements all of the data access functionality you are familiar with from ADO.NET, with numerous enhancements.

3.2.4.1 Similarities Between the Application Module and a Data Set

The application module component plays the same role as the ADO.NET data set. It is a strongly typed service component that represents a collection of row sets called view object instances, which are similar to ADO.NET data tables. An application module exposes a service interface that surfaces the rows of data in a developer-configurable set of its view instances as an SDO-compatible service (accessible as a web service, or as an SCA composite). The application module works with a related transaction object to provide the context for the SQL queries that the view objects execute. The application module also provides the context for modifications saved to the database by the entity objects, which play the role of the ADO.NET data adapter.

3.2.4.2 Similarities Between the Entity Object and a Data Adapter

The entity object component is like a strongly-typed ADO.NET data adapter. It represents the rows in a particular table and handles the find-by-primary-key, insert, update, delete, and lock operations for those rows. In ADF Business Components, you don't have to specify these statements yourself, but you can override them if you need to. The entity object encapsulates validation or other business logic related to attributes or entire rows in the underlying table. This validation is enforced when data

is modified and saved by the end user using any view object query that references the underlying entity object. One difference in ADF Business Components is that the arbitrary, flexible querying is performed by SQL statements at the view object instance level, but the view objects and entity objects coordinate automatically at runtime.

3.2.4.3 Similarities Between the View Object and a Data Table

The view object component encapsulates a SQL query and manages the set of resulting rows. It can be related to an underlying entity object to automatically coordinate validation and saving of modifications made by the user to those rows. This cooperation between a view object's queried data and an entity object's encapsulated business logic offers all of the benefits of the data table with the clean encapsulation of business logic into a layer of business domain objects. Like ADO.NET data tables, you can easily work with a view object's data as XML or have a view object read XML data to automatically insert, update, or delete rows based on the information it contains.

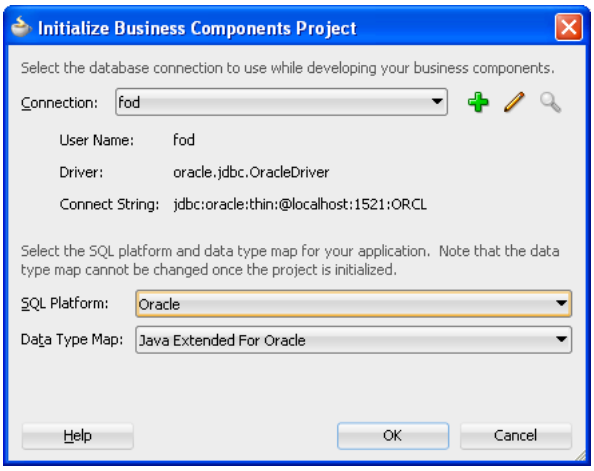
3.3 Overview of Design Time Facilities

JDeveloper includes comprehensive design time support for ADF Business Components. Collectively, these facilities let you create, edit, diagram, test, and refactor the business components.

3.3.1 Choosing a Connection, SQL Platform, and Data Type Map

The first time you create a component, you'll see the Initialize Business Components Project dialog shown in Figure 3–1. You use this dialog to select a design time application resource connection to use while working on your business components in this data model project (the term used in this guide for a project created for ADF Business Components) or to create a new application resource connection by copying an existing IDE-level connection.

Figure 3–1 Initialize Business Components Project Dialog



Since this dialog appears before you create your first business component, you also use it to globally control the SQL platform that the view objects will use to formulate SQL statements. Although the default for an Oracle database connection is always the Oracle SQL platform, other SQL platforms you can choose include OLite (for the Oracle Lite database), SQLServer for a Microsoft SQLServer database, DB2 for an IBM DB2 database, and SQL92 for any other supported SQL92- compliant database.

Note: If you plan to have your application run against both Oracle and non-Oracle databases, you should select the **SQL92** SQL platform when you begin building your application, not later. While this sacrifices some of the Oracle-specific optimizations that are inherent in using the Oracle SQL platform, it makes the application portable to both Oracle and non-Oracle databases.

Additionally, the dialog lets you determine which set of data types that you want the data model project to use. If JDeveloper detects you are using an Oracle database driver, it defaults the **Data Type Map** setting to the **Java Extended for Oracle** type map which uses standard Java types and the optimized types in the `oracle.jbo.domain` package for common data types. When you create a SQL92-compliant application that will run on a non-Oracle database, you can change the data type map to the **Java** setting to globally use only the basic Java data types.

The **Java Extended for Oracle** type map and the **Oracle Domains** type map handle numeric data differently. When you create a new application the default type map **Java Extended for Oracle** maps numeric data to the `java.math.BigDecimal` class, which inherits from `java.math.Number`. The `java.math.BigDecimal` default matches the way the Fusion web application view layer, consisting of ADF Faces components, preserves alignment of numeric data (such as numeric values displayed by ADF Faces input fields in a web page). Whereas the **Oracle Domains** type map, which maps numeric data to the `oracle.jbo.domain.Number` class, may not display the data with the alignment expected by certain ADF Faces components. Aside from this alignment issue, the **Oracle Domains** type map remains a valid choice and applications without ADF Faces components will function without issue.

New Fusion web applications should use the default **Java Extended for Oracle** type. The **Oracle Domains** type map is provided for backward compatibility and for applications that do not use ADF Faces as the view layer technology. Please note that when you migrate an application developed with JDeveloper version 11.1.1.4.0 or earlier, your application will continue to use the **Oracle Domains** type map and will not change to the current default type map **Java Extended for Oracle**.

Note that you cannot change the data type map after you've initialized a data model project. After you have initialized a data model project, you can you can override the SQL platform in the overview editor for the `adf-config.xml` file. Specifying the database type in the `adf-config.xml` file supports generating SQL statements during runtime that can require the actual database type. You can locate the file in the Application Resources panel by expanding the **Descriptors > ADF META-INF** node.

3.3.2 Creating New Components Using Wizards

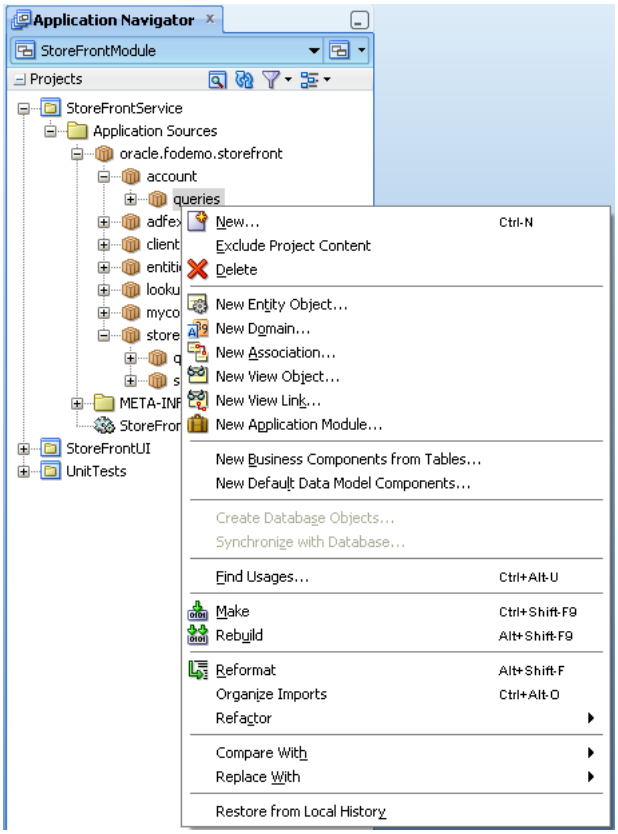
In the New Gallery in the **ADF Business Components** category, JDeveloper offers a wizard to create each kind of business component. Each wizard allows you to specify the component name for the new component and to select the package into which you'd like to organize the component. If the package does not yet exist, the new component becomes the first component in that new package.

The wizard presents a series of panels that capture the necessary information to create the component type. When you click **Finish**, JDeveloper creates the new component by saving its XML component definition file. If you have set your Java generation options to generate classes by default, JDeveloper also creates the initial custom Java class files.

3.3.3 Creating New Components Using the Context Menu

Once a package exists in the Application Navigator, you can quickly create additional business components of any type in the package by selecting it in the Application Navigator and using one of the options on the context menu shown in Figure 3–2.

Figure 3–2 Context Menu Options on a Package to Create Any Kind of Business Component



3.3.4 Editing Components Using the Component Overview Editor

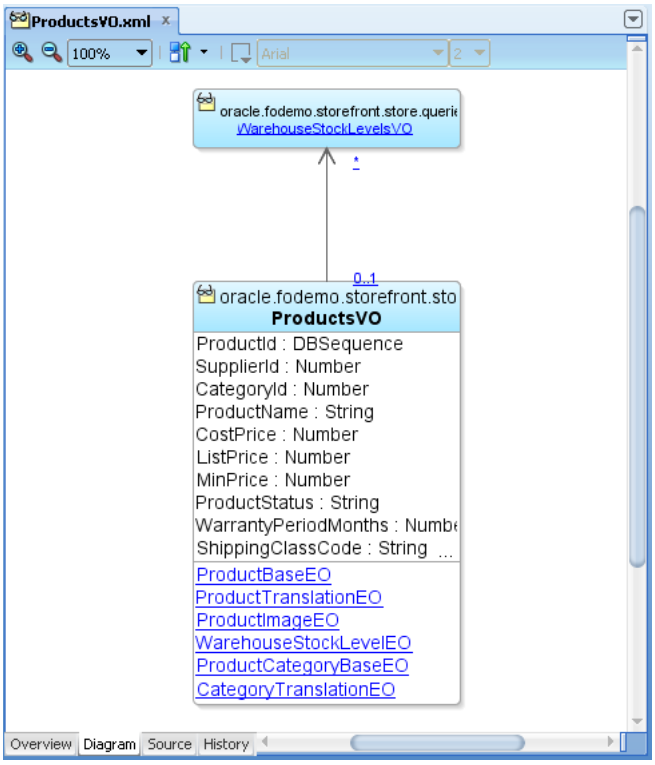
Once a component exists, you can edit it using the respective overview editor that you access either by double-clicking the component in the Application Navigator or by selecting it and choosing the **Edit** option from the context menu. The overview editor presents the same editing options that you see in the wizard but it may arrange them differently. The overview editor allows you to change any aspect of the component. When you click **OK**, JDeveloper updates the components XML component definition file and, if necessary, any of its related custom Java files. Because the overview editor is a JDeveloper editor window, rather than a modal dialog, you can open and view the overview editor for as many components as you require.

3.3.5 Displaying Related Components Using Diagrams

As the number of business components that your project defines increases, you may decide to refactor components to change the relationships that you originally created. To help you understand the relationship between components in the data model project, open any component in the editor window and click the **Diagram** tab. The relationship diagram in the editor identifies the component you are editing in bold text. Related components appear as link text that you can click to display the

relationship diagram for the component identified by the link. For example, Figure 3–3 displays the **Diagram** tab in the editor for the view object `ProductsVO`. The diagram identifies the list of entity objects that `ProductsVO` can access (for example, `ProductBaseEO` and `ProductTranslationEO` and so on), the view link (`ProductsToWarehouseStockLevels`) that defines the view object's relationship to a related view object, and the related view object (`WarehouseStockLevelsVO`) named by the view link. Each of these related components displays as a link that you can click to open the component in the **Diagram** tab for its editor. By clicking on related component links, you can use the diagrams to navigate the component relationships that your project defines.

Figure 3–3 Relationship Diagram Displays Main Object and All Related Components in Component Editor's Diagram Tab



3.3.6 Visualizing, Creating, and Editing Components Using UML Diagrams

JDeveloper offers extensive UML diagramming support for ADF Business Components. You can drop components that you've already created onto a business components diagram to visualize them. You can also use the diagram to create and modify components. The diagrams are kept in sync with changes you make in the editors.

To create a new business components diagram, use the **Business Components Diagram** item in the **ADF Business Components** category of the JDeveloper New Gallery. This category is part of the **Business Tier** choices.

3.3.7 Testing Application Modules Using the Oracle ADF Model Tester

Once you have created an application module component, you can test it interactively using the built-in Oracle ADF Model Tester. To launch the Oracle ADF Model Tester,

select the application module in the Application Navigator or in the business components diagram and choose either **Run** or **Debug** from the context menu.

The Oracle ADF Model Tester presents the view object instances in the application module's data model and allows you to interact with them using a dynamically generated user interface. The tool also provides a list of the application module's client interface methods that you can test interactively by double-clicking the application module node. This tool is invaluable for testing or debugging your business service both before and after you create the web page view layer.

3.3.8 Refactoring Components

At any time, you can select a component in the Application Navigator and choose **Refactor > Rename** from the context menu to rename the component. The Structure window also provides a **Rename** context menu option for details of components, such as view object attributes or view instances of the application module data model, that do not display in the Application Navigator. You can also select one or more components in the navigator by using Ctrl + click and then choosing **Refactor > Move** from the context menu to move the selected components to a new package. References to the old component names or packages in the current data model project are adjusted automatically.

3.4 Overview of the Oracle ADF Active Data Model

One of the key simplifying benefits of using ADF Business Components for your business service implementation is the application module's support for an "active data model" of row sets. The data model defines the business objects specific to your application, while the row sets of each business object contain the data. In the UI portion of the application, the UI components interact with these business objects to perform retrieve, create, edit, and delete operations. When you use ADF Business Components in combination with the ADF Model layer and ADF Faces UI components, the data model is "active" because your UI components will automatically update to reflect any changes to the row sets of these business objects.

Thus, the active data model represents a solution that works across application technology layers to ensure that the UI and data model remain synchronized.

3.4.1 A More Generic Business Service Solution

Using a typical Java EE business service implementation makes the client developer responsible for:

- Invoking service methods to return data to present
- Tracking what data the client has created, deleted, or modified
- Passing the changes back to one or more different service methods to validate and save them

Retrieving, creating, editing, deleting, and saving is a typical sequence of tasks performed during application development. As a result, the ADF application module represents a smarter, more generic solution. Using the application module for your business service, you simply bind client UI components like fields, tables, and trees to the active view object instances in the application module's data model. Your UI components in JSP or JSF pages for the web or mobile devices (as well as desktop-fidelity UIs comprising windows and panels that use Swing) automatically update to reflect any changes to the rows in the view object row sets of the data model. Additionally, if you define custom business service methods for the application

module that happen to produce changes to the data model view instances, those changes will also be automatically reflected in the UI components.

Under the covers, the application module component implements a set of *generic* service methods that allow users to leverage its active data model in a service-oriented architecture (SOA). Both web service and UI clients can easily access an application module's data model using simple APIs. These APIs enable you to search for and modify any information that the application module makes available.

When you build UIs that take advantage of the ADF Model layer for declarative data binding, you generally won't need to write client-side code. The active data model supports declaratively binding UI components in your web pages to view objects in the data model and to custom business service methods. Additionally, when you create web services in a SOA environment they can be bound declaratively to the data model through the data model's web service interface.

3.4.2 Typical Scenarios for an Active Data Model

Without an active data model, you would need to write more code in the client or web service to handle the straightforward, everyday CRUD-style operations. In addition, to keep pages up to date, you would need to manage "refresh flags" that clue the controller layer in to requesting a "repull" of data from the business service to reflect data that might have been modified. When using an ADF application module to implement your business service, you can focus on the business logic at hand, instead of the plumbing to make your business work as your end users expect.

Consider the following three simple, concrete examples of the active data model:

- New data appears in relevant displays without requerying

  A customer logs into the Fusion Order Demo application and displays a list of items in their shopping cart. Then if the customer visits some product pages and creates a new order item, when they return back to display their shopping cart, the new item appears in their list without requiring the application to requery the database.

- Changes caused by business domain logic automatically reflected

  A back office application causes an update to the order status. Business logic encapsulated in the `Orders` entity object in the business domain layer contains a simple rule that updates the last update date whenever the order status attribute is changed. The user interface updates to automatically reflect the last update date that was changed by the logic in the business domain layer.

- Invocation of a business service method by the ADF Model layer binding requeries data and sets current rows

  In a tree display, the user clicks on a specific node in a tree. This action declaratively invokes a business service method by the ADF tree binding on your application module that requeries master-detail information and sets the current rows to an appropriate row in the row set. The display updates to reflect the new master-detail data and current row displayed.

3.4.3 Active Data Model Support for Custom Code

Because the application module supports the active data model, your client user interface will remain up to date. This means you will not need to write code in the client that is related to setting up or manipulating the data model.

Another typical type of client-side code you no longer have to write using ADF Business Components is code that coordinates detail data collections when a row in the master changes. By linking the view objects, you can have the coordination performed automatically for you.

However, when you do need to write custom code, encapsulate that code inside custom methods of your application module component. For example, whenever the programmatic code that manipulates view objects is a logical aspect of implementing your complete business service functionality, you should encapsulate the details by writing a custom method in your application module's Java class. This includes, but is not limited to, code that:

- Configures view object properties to query the correct data to display
- Iterates over view object rows to return an aggregate calculation
- Performs any kind of multistep procedural logic with one or more view objects

By centralizing these implementation details in your application module, you gain the following benefits:

- You make the intent of your code more clear to clients.
- You allow multiple client pages to easily call the same code if needed.
- You simplify regression-testing of your complete business service functionality.
- You keep the option open to improve your implementation without affecting clients.
- You enable *declarative* invocation of logical business functionality in your pages.

3.5 Overview of the ADF Business Components Implementation

Before you begin implementing specific ADF business components, it is a good idea to have some familiarity with ADF Business Components design and implementation.

3.5.1 Standard Java and XML

As is the case with all Oracle ADF technologies, ADF Business Components is implemented in Java. The working, tested components in the framework provide generic, metadata-driven functionality from a rich layer of robust code. ADF Business Components follows the Java EE community best practice of using cleanly separated XML files to store metadata that you define to configure each component's runtime behavior.

Since ADF Business Components is often used for business critical applications, it's important to understand that the full source for Oracle ADF, including ADF Business Components, is available to supported customers through Oracle Worldwide Support. The full source code for Oracle ADF can be an important tool to assist you in diagnosing problems, as described in Section 36.8, "Using the ADF Declarative Debugger." Working with the full source code for Oracle ADF also helps you understand how to correctly extend the base framework functionality to suit your needs, as described in Section 12.4, "Customizing Framework Behavior with Extension Classes."

3.5.2 Application Server or Database Independence

Applications built using ADF Business Components can run on any Java-capable application server, including any Java EE-compliant application server. Because

business components are implemented using plain Java classes and XML files, you can use them in any runtime environment where a Java Virtual Machine is present. This means that services built using ADF Business Components are easy to use both inside a Java EE server — known as the "container" of your application at runtime — and outside.

Customers routinely use application modules in such diverse configurations as command-line batch programs, web services, custom servlets, JSP pages, and desktop-fidelity clients built using Swing.

You can also build applications that work with non-Oracle databases, as described in Section 3.3.1, "Choosing a Connection, SQL Platform, and Data Type Map." However, applications that target Oracle databases will find numerous optimizations built into ADF Business Components.

3.5.3 Java EE Design Pattern Support

ADF Business Components implements all of the popular Java EE design patterns that you would normally need to understand, implement, and debug yourself to create a real-world enterprise Java EE application. If it is important to you to cross-reference the names of these design patterns from the Java EE specifications with their ADF Business Components counterparts, you can refer to Appendix E, "ADF Business Components Java EE Design Pattern Catalog."

3.5.4 Source Code Organization

Since ADF Business Components is implemented in Java, its classes and interfaces are organized into packages. Java packages are identified by dot-separated names that developers use to arrange code into a hierarchical naming structure.

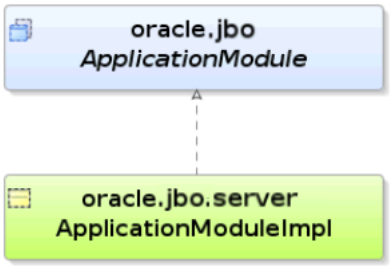
The classes and interfaces that comprise the source code provided by ADF Business Components reside in the `oracle.jbo` package and numerous subpackages. However, in day to day work with ADF Business Components, you'll work typically with classes and interfaces in these two key packages:

- The `oracle.jbo` package, which contains all of the interfaces that are designed for the business service client to work with
- The `oracle.jbo.server` package, which contains the classes that implement these interfaces

> **Note:** The term *client* here refers to any code in the model, view, or controller layers that accesses the application module component as a business service.

Figure 3–4 shows a concrete example of the application module component. The client interface for the application module is the `ApplicationModule` interface in the `oracle.jbo` package. This interface defines the names and signatures of methods that clients can use while working with the application module, but it does not include any specifics about the implementation of that functionality. The class that implements the base functionality of the application module component resides in the `oracle.jbo.server` package and is named `ApplicationModuleImpl`.

Figure 3–4 ADF Business Components Separate Interface and Implementation



3.5.5 Package Naming Conventions

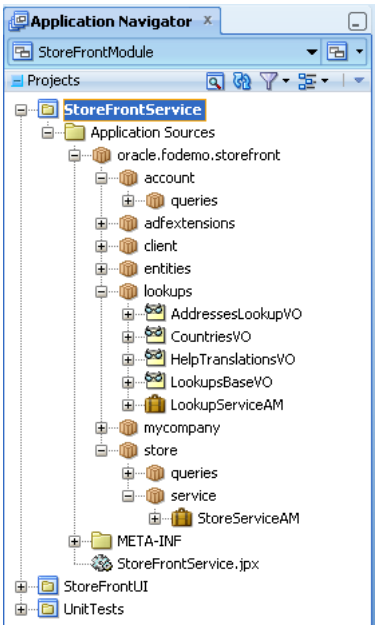
Since ADF Business Components is implemented in Java, the components of your application (including their classes, interfaces, and metadata files) will also be organized into packages.

To ensure that your components won't clash with reusable components from other organizations, choose package names that begin with your organization's name or web domain name. So, for example, the Apache organization chose `org.apache.tomcat` for a package name related to its Tomcat web server, while Oracle picked `oracle.xml.parser` as a package name for its XML parser. Components you create for your own applications might reside in packages with names like `com.`*`yourcompany`*`.`*`yourapp`* and subpackages of these.

As a specific example, the ADF Business Components that make up the main business service for the Fusion Order Demo application are organized into the `oracle.fodemo.storefront` package and its subpackages. As shown in Figure 3–5, these components reside in the `StoreFrontService` project in the `StoreFrontModule` application, and are organized broadly as follows:

- `oracle.fodemo.storefront.account.queries` contains the view objects used in the customer registration process
- `oracle.fodemo.storefront.client` contains test client `.java` files
- `oracle.fodemo.storefront.entities` contains the entity objects
- `oracle.fodemo.storefront.lookups` contains static data view objects and the `LookupServiceAM` shared application module
- `oracle.fodemo.storefront.store.queries` contains the view objects used to manage the storefront
- `oracle.fodemo.storefront.store.service` contains the `StoreServiceAM` application module

Figure 3–5 Organization of ADF Business Components in the Fusion Order Demo Application



In your own applications, you can choose any package organization that you believe best. In particular, keep in mind that you are not constrained to organize components of the same type into a single package.

Because JDeveloper supports component refactoring, you can easily rename components or move them to a different package at any time. This flexibility allows you to easily incorporate inevitable changes into the application as your application evolves.

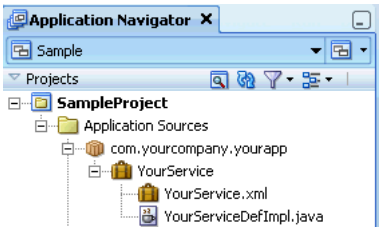
There is no optimal number of components in a package. However, with experience, you'll realize that the best structure for your team falls somewhere between the two extremes of placing all components in a single package and placing each component in its own, separate package.

One thing to consider is that the project is the unit of granularity that JDeveloper supports for reuse in other data model projects. So, you might factor this consideration into how you choose to organize components. For more information, see Section 38.3, "Packaging a Reusable ADF Component into an ADF Library."

3.5.6 Metadata with Optional Custom Java Code

Each kind of component in ADF Business Components comes with built-in runtime functionality that you control through declarative settings. These settings are stored in an XML component definition file with the same name as the component that it represents. When you need to write custom code for a component, for example to augment the component's behavior, you can enable an optional custom Java class for the component in question. Figure 3–6 shows how the Application Navigator displays the XML component definition and optional custom Java class for an application module.

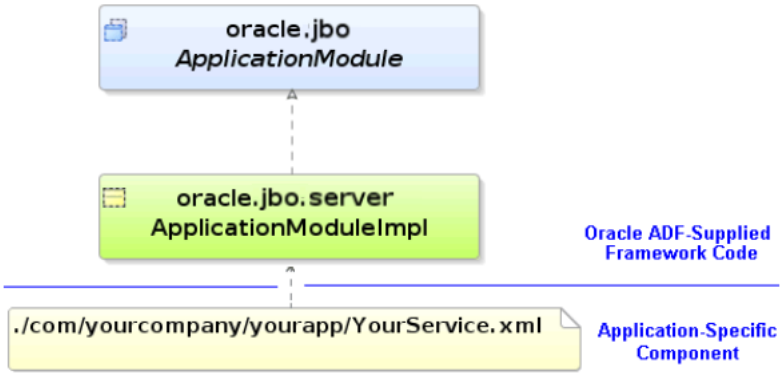
Figure 3–6 Application Navigator Displays Component XML File and Optional Class Files



3.5.6.1 Example of an XML-Only Component

Figure 3–7 illustrates the XML component definition file for an application-specific component like an application module named `YourService` that you create in a package named `com.yourcompany.yourapp`. The corresponding XML component definition resides in a `./com/yourcompany/yourapp` subdirectory of the data model project's source path root directory. That XML file records the name of the Java class it should use at runtime to provide the application module implementation. In this case, the XML records the name of the base `oracle.jbo.server.ApplicationModuleImpl` class provided by Oracle ADF.

Figure 3–7 XML Component Definition File for an Application Module



When used without customization, your component is completely defined by its XML component definition and it will be fully functional without custom Java code or even a Java class file for the component. If you have no need to extend the built-in functionality of a component in ADF Business Components, and no need to write any custom code to handle its built-in events, you can use the component in this XML-only fashion.

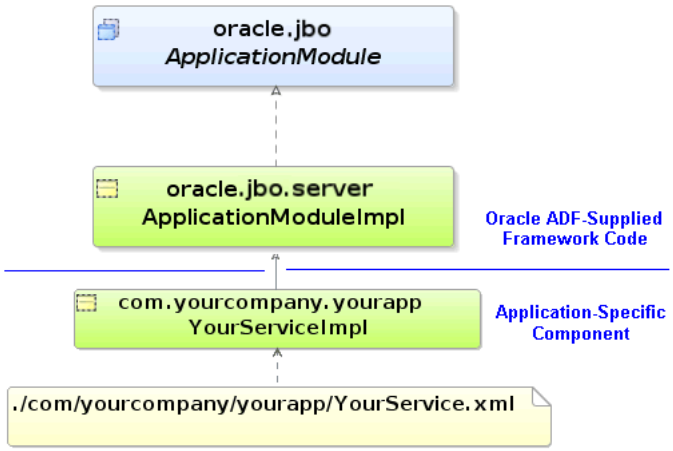
3.5.6.2 Example of a Component with Custom Java Class

When you need to add custom code to extend the base functionality of a component or to handle events, you can enable a custom Java class for any of the key types of ADF Business Components you create. You enable the generation of custom classes for a component on the Java page of its respective overview editor in JDeveloper. When you enable this option, JDeveloper creates a Java source file for a custom class related to the component whose name follows a configurable naming standard. This class, whose name is recorded in the component's XML component definition, provides a place where you can write the custom Java code required by that component. Once you've enabled a custom Java class for a component, you can navigate to it using a

corresponding **Go To *componentName* Class** option in the component's Application Navigator context menu.

Figure 3–8 illustrates what occurs when you enable a custom Java class for the *YourService* application module. A *YourServiceImpl*.java source code file is created in the same source path directory as your component's XML component definition file. The *YourServiceImpl*.xml file is updated to reflect the fact that at runtime the component should use the com.*yourcompany*.*yourapp*.*YourServiceImpl* class instead of the base ApplicationModuleImpl class.

Figure 3–8 Component with Custom Java Class



Note: The examples in this guide use default settings for generated names of custom component classes and interfaces. If you want to change these defaults for your own applications, use the ADF Business Components: Class Naming page of the JDeveloper Preferences dialog. Changes you make only affect newly created components.

3.5.7 Basic Data Types

The Java language provides a number of built-in data types for working with strings, dates, numbers, and other data. When working with ADF Business Components, you can use these types, but by default you'll use an optimized set of types in the oracle.jbo.domain and oracle.ord.im packages. These types, shown in Table 3–1, allow data accessed from the Oracle database to remain in its native, internal format. You will achieve better performance using the optimized data types provided by ADF Business Components by avoiding costly type conversions when they are not necessary.

There are two cases where an optimized data type is not used by default and a Java built-in data type is used instead. To work with string-based data, by default ADF Business Components uses the regular java.lang.String type. Additionally, to work with numeric data, by default ADF Business Components uses the java.math.BigDecimal type, which formats numerals in a way that is consistent with the alignment expected by ADF Faces components. For backward compatibility, and for applications that do not use ADF Faces components, the optimized data type oracle.jbo.domain.Number provided in previous releases remains an alternative data type for numeric data.

Table 3–1 Basic Data Types in the oracle.jbo.domain and oracle.ord.im Packages

Data Type	Package	Represents
`Number` (not used by default)	`oracle.jbo.domain`	Any numerical data. By default, ADF Business Components uses the `java.math.BigDecimal` type to support formatting of numeric values with the alignment expected by ADF Faces components. The `java.math.BigDecimal` type should be used whenever ADF Faces is the chosen view layer technology.
`Date`	`oracle.jbo.domain`	Date with optional time
`DBSequence`	`oracle.jbo.domain`	Sequential integer assigned by a database trigger
`RowID`	`oracle.jbo.domain`	Oracle database ROWID
`Timestamp`	`oracle.jbo.domain`	Timestamp value
`TimestampTZ`	`oracle.jbo.domain`	Timestamp value with time zone information

`TimestampLTZ`	`oracle.jbo.domain`	Timestamp value with local time zone information retrieved from JavaVM or from the ADF Context when configured in the application's `adf-config.xml` with an EL expression:
		`<user-time-zone-config xmlns=`
		`  "http://xmlns.oracle.com/adf/usertimezone/config">`
		`  <user-timezone expression= "EL exp" />`
		`</user-time-zone-config>`
		The EL expression will be evaluated to determine the time zone of the current user; otherwise, the value defaults to the time zone of the JavaVM.

`BFileDomain`	`oracle.jbo.domain`	Binary File (BFILE) object
`BlobDomain`	`oracle.jbo.domain`	Binary Large Object (BLOB)
`ClobDomain`	`oracle.jbo.domain`	Character Large Object (CLOB)
`OrdImageDomain`	`oracle.ord.im`	Oracle Intermedia Image (ORDIMAGE)
`OrdAudioDomain`	`oracle.ord.im`	Oracle Intermedia Audio (ORDAUDIO)
`OrdVideoDomain`	`oracle.ord.im`	Oracle Intermedia Video (ORDVIDEO)
`OrdDocDomain`	`oracle.ord.im`	Oracle Intermedia Document (ORDDOC)
`Struct`	`oracle.jbo.domain`	User-defined object type
`Array`	`oracle.jbo.domain`	User-defined collection type (e.g. `VARRAY`)

Note: If you are not using ADF Faces as your view layer technology, you may use either the `java.math.BigDecimal` type or the `oracle.jbo.domain.Number` type. Be aware, however, that the `oracle.jbo.domain.Number` class has the same class name as the built-in `java.lang.Number` type. Since the Java compiler *implicitly* imports `java.lang.*` into every class, you will need to explicitly import the `oracle.jbo.domain.Number` class into any class that references it. Typically, JDeveloper will follow this practice for you, but if you receive compiler or runtime errors related to "Number is an abstract class" it means you are using `java.lang.Number` instead of `oracle.jbo.domain.Number`. Adding the:

```
import oracle.jbo.domain.Number;
```

line at the top of your class, after the `package` line, prevents these kinds of errors.

3.5.8 Generic Versus Strongly-Typed APIs

When working with application modules, view objects, and entity objects, you can choose to use a set of generic APIs or you can have JDeveloper generate code into a custom Java class to enable a strongly-typed API for that component. For example, when working with an view object, if you wanted to access the value of an attribute in any row of its result, the generic API would look like this:

```
Row row = ordersVO.getCurrentRow();
Date shippedDate = (Date)row.getAttribute("OrderShippedDate");
```

Notice that using the generic APIs, you pass string names for parameters to the accessor, and you have to cast the return type to the expected type, as with `Date` shown in the example.

Alternatively, when you enable the strongly typed style of working you can write code like this:

```
OrdersRow row = (OrdersRow)ordersVO.getCurrentRow();
Date shippedDate = row.getOrderShippedDate();
```

In this case, you work with generated method names whose return type is known at compile time, instead of passing string names and having to cast the results. Typically, it is necessary to use strongly typed accessors when you need to invoke the methods from the business logic code without sacrificing compile-time safety. This can also be useful when you are writing custom validation logic in setter methods, although in this case, you may want to consider using Groovy expressions instead of generating entity and view row implementation classes for Business Components. Subsequent chapters explain how to enable this strongly typed style of working by generating Java classes for business logic that you choose to implement using Java.

3.5.9 Custom Interface Support for Client-Accessible Components

Only these components of the business service as visible to the client:

- Application module, representing the service itself
- View objects, representing the query components
- View rows, representing each row in a given query component's results

The entity objects in the business service implementation is intentionally not designed to be referenced directly by clients. Instead, clients work with the data queried by view objects as part of an application module's data model. Behind the scenes, the view object cooperates automatically with entity objects in the business services layer to coordinate validating and saving data that the user changes. For more information about this runtime interaction, see Section 6.3.9, "What Happens at Runtime: How View Objects and Entity Objects Cooperate."

3.5.9.1 Framework Client Interfaces for Components

The Java interfaces of the `oracle.jbo` package provide a client-accessible API for your business service. This package intentionally does not contain an `Entity` interface, or any methods that would allow clients to directly work with entity objects. Instead, client code works with interfaces like:

- `ApplicationModule`, to work with the application module
- `ViewObject`, to work with the view objects
- `Row`, to work with the view rows

3.5.9.2 Custom Client Interfaces for Components

When you begin adding custom code to your ADF business components that you want clients to be able to call, you can "publish" that functionality to clients for any client-visible component. For each of your components that publishes at least one custom method to clients on its client interface, JDeveloper automatically maintains the related Java interface file. So, assuming you were working with an application module like `StoreServiceAM` in the Fusion Order Demo application, you could have custom interfaces like:

- Custom application module interface

  ```
  StoreServiceAM extends ApplicationModule
  ```

- Custom view object interface

  ```
  OrderItemsInfo extends ViewObject
  ```

- Custom view row interface

  ```
  OrderItemsInfoRowClient extends Row
  ```

Client code can then cast one of the generic client interfaces to the more specific one that includes the selected set of client-accessible methods you've selected for your particular component.

3.6 Overview of Groovy Scripting Language Support

Groovy is a scripting language with Java-like syntax for the Java platform. The Groovy scripting language simplifies the authoring of code by employing dot-separated notation, yet still supporting syntax to manipulate collections, Strings, and JavaBeans. Groovy language expressions in ADF Business Components differs from the Java code that you might use in a Business Components custom Java class because Groovy expressions are executed at runtime, while the strongly typed language of Java is executed at compile-time. Additionally, because Groovy expressions are dynamically compiled, they are stored in the XML definition files of the business components where you use it.

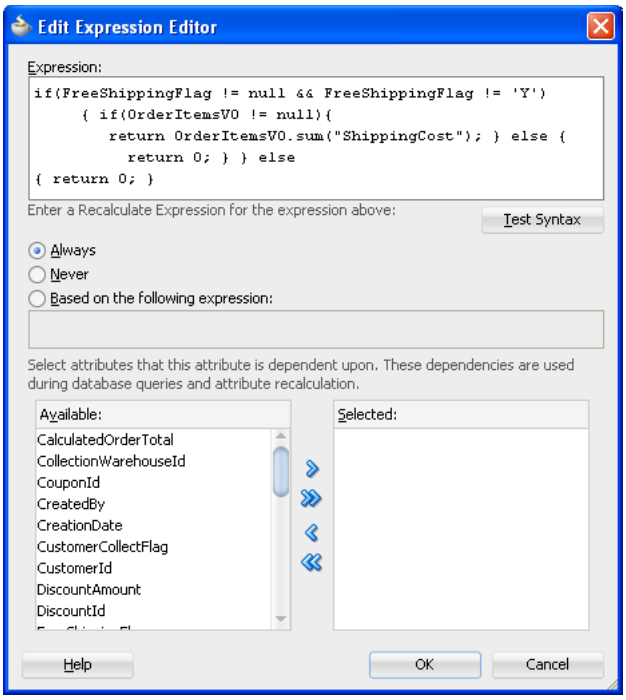
ADF Business Components supports the use of the Groovy scripting language in places where access to entity object and view object attributes is useful, including attribute validators (for entity objects), attribute default values (for either entity objects or view objects), transient attribute value calculations (for either entity objects or view objects), bind variable default values (in view object query statements and view criteria filters), and placeholders for error messages (in entity object validation rules). Additionally, ADF Business Components provides a limited set of built-in keywords that can be used in Groovy expressions.

Specifically, the ADF Business Components framework provides support for the use of Groovy language expressions to perform the following tasks:

- Define a Script Expression validator or Compare validator (see Section 7.5, "Using Groovy Expressions For Validation and Business Rules")
- Define error message tokens for handling validation failure (see Section 7.7.4, "How to Embed a Groovy Expression in an Error Message")
- Handle conditional execution of validators (see Section 7.7.3, "How to Conditionally Raise Error Messages Using Groovy")
- Set the default value of a bind variable in the view object query statement (see Section 5.10, "Working with Bind Variables")
- Set the default value of a bind variable that specifies a criteria item in the view criteria statement (see Section 5.11, "Working with Named View Criteria").
- Define the default value and optional recalculate condition for an entity object attribute (see Section 4.10.6, "How to Define a Static Default Value")
- Determine the value of a transient attribute of an entity object or view object (see Section 4.11, "Adding Transient and Calculated Attributes to an Entity Object" and Section 5.14, "Adding Calculated and Transient Attributes to a View Object")

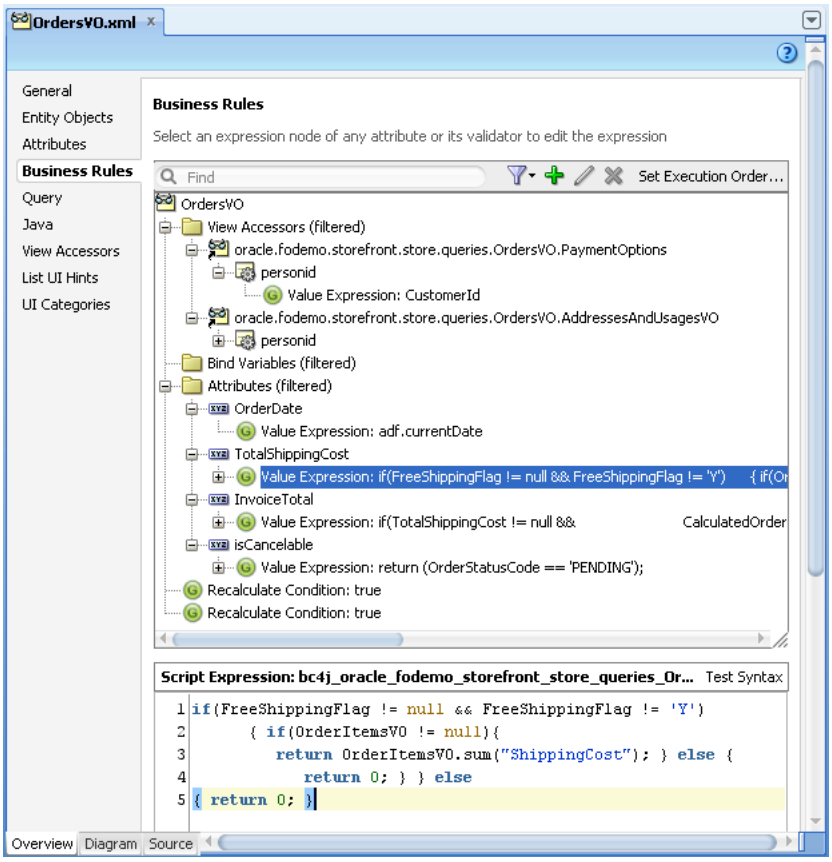
To perform these tasks in JDeveloper, you use expression editor dialogs that are specific to the task. For example, when you want to create a default value for a transient view object attribute, you use the attribute's Edit Expression Editor dialog to enter an expression that determines a runtime value for the attribute. The same dialog also lets you specify when the value should be calculated (known as a *recalculate condition*), as shown in Figure 3–9.

Figure 3–9 Dialog to Edit Expression for Default Attribute Values



Additionally, the overview editor that you use to edit entity objects and view objects displays the Business Rules page, where you can view and edit all expressions used by a single business component. For example, the Business Rules page that you display for a view object lets you view all expressions that the view object uses on its view accessors, bind variables, and attributes. You can filter the display to show only those items with Groovy expressions defined, as shown in Figure 3–10. Although expressions cannot be verified at design time, all expression editors let you test the syntax of the expression before you save it.

Figure 3–10 Overview Editor's Business Rules Page Shows All Expressions Used by a Business Component



For more information about the Groovy language, refer to the following web site:

- http://groovy.codehaus.org/

3.6.1 Referencing Business Components Objects in Groovy Expressions

There is one top-level object named `adf` that allows you access to objects that the framework makes available to the Groovy script. When you reference an Oracle ADF object in a Groovy expression, the Oracle ADF runtime returns wrapper objects that do not correspond to the actual concrete type of the classes. These wrapper objects support all of the method and field types of the wrapped object. Your expressions can use wrapped objects as if they were the actual object. Note, however, any attempt to cast wrappered objects to its concrete type will fail with a `ClassCastException`. In general, when working with the Groovy language it is not necessary to use explicit casting, and in the case of these wrapped ADF Business Components objects, doing so will cause an exception.

The accessible Oracle ADF objects consist of the following:

- `adf.context` - to reference the `ADFContext` object
- `adf.object` - to reference the object on which the expression is being applied (which can also be referenced using the keyword `object`, without the `adf` prefix). Other accessible member names come from the context in which the Groovy script is applied.

- Entity object attributes: The context is an instance of the entity implementation class. Through this object you can reference custom methods of the custom entity implementation class, any methods defined by the base implementation class as specified by the JavaDoc for `EntityImpl`, and you can reference the attributes of the entity instance.
- Entity object script validation rules: The context is the validator object (`JboValidatorContext`) merged with the entity on which the validator is applied. For details about keywords that you can use in this context, see Section 3.6.2.1, "Referencing Members of the Same Business Component."
- View object attributes: The context is an instance of the view row implementation class. Through this object, you can reference custom methods of the custom view row implementation class, any methods defined by the base implementation class as specified by the JavaDoc for `ViewRowImpl`, and you can reference the attributes of the view row instance as defined by the query row set.
- Bind variable in view objects: The context is the variable object itself not the view row. You can reference the `structureDef` property to access other information as well as the `viewObject` property to access the view object in which the bind variable participates. However, access to view object attributes is not supported.
- Bind variable in view accessors: The context is the current view row. The view accessor with bind variable is used to create a cascading List of Value (LOV). The view accessor can derive Groovy-driven values from the current view row in the view accessor view object used to formulate the list of valid choices.
- Transient attributes: The context is the current entity or view row. You can reference attributes by name in the entity or view row in which the attribute appears, as well as public methods on that entity or view row. To access methods on the current object, you must use the `object` keyword to reference the current object (for example, `object.methodName( )`). The `object` keyword is equivalent to the `this` keyword in Java. Without it, in transient expressions, the method will be assumed to exist on the dynamically compiled Groovy script object itself.

- `adf.error` - in validation rules, to access the error handler that allows the validation expression to generate exceptions or warnings
- `adf.userSession` - returns a reference to the ADF Business Components user session (which you can use to reference values in the `userData` hashmap that is part of the session)

You can also reference the current date (time truncated) or current date and time using the following expressions:

- `adf.currentDate`
- `adf.currentDateTime`

3.6.2 Referencing Custom Business Components Methods and Attributes in Groovy Expressions

Groovy script language simplifies the authoring of code that you might write to access methods and attributes of your entity object and view objects.

3.6.2.1 Referencing Members of the Same Business Component

The simplest example of referencing business component members, including methods and attributes that the entity object and view object define, is to reference attributes that exist in the same entity object or view object as the attribute that you apply the expression.

For example, you could define a Groovy expression to calculate the value of a transient attribute `AnnualSalary` on an entity object with an attribute `Sal` that specifies the employee's monthly salary:

```
Sal * 12
```

Or, with Groovy you can write a simple validation rule to compare the attributes of a single view object using syntax like:

```
PromotionDate > HireDate
```

Using Java, this same comparison would look like:

```
((Date)getAttribute("PromotionDate")).compareTo((Date)getAttribute("HireDate")) > 0
```

Note that the current object is passed in to the script as the `this` object, so you can reference an attribute in the current object by simply using the attribute name. For example, in an attribute-level or entity-level Script Expression validator, to refer to an attribute named "HireDate", the script can simply reference `HireDate`.

Similar to referencing attributes, when you define custom methods in an entity implementation class, you can invoke those methods as part of your expression. For example, to define an attribute default value:

```
adf.object.getDefaultSalaryForGrade()
```

A method reference requires the prefix `adf.object` which allows you to reference the same entity that defines the attribute on which the expression is applied. This same prefix also allows you to reference the methods of the base class of the entity implementation class (`EntityImpl.java`) that your custom implementation class extends.

Note that when you want to reference the method of an entity implementation class in a validation rule, you use the `source` prefix:

```
source.getDefaultSalaryForGrade()
```

Use of the `source` prefix is necessary in validators because the `object` keyword implies the validation rule object instead of the entity object (where the method is defined).

To allow you to reference members of the validator object (`JboValidatorContext`), you can use these keywords in your validation rule expression:

- `newValue`: in an attribute-level validator, to access the attribute value being set
- `oldValue`: in an attribute-level validator, to access the current value of the attribute being set

For example, you might use the following expression to specify a dynamic validation rule check of the salary for a salesman.

```
if (Job == "SALESMAN")
{
  return newValue < source.getMaxSalaryForGrade(Job)
}
else
```

```
return true
```

3.6.2.2 Referencing Members of Other Business Components

You can also reference the methods and attributes that entity objects and view objects defines in the expressions you apply to a different entity object attribute or validation rule. This is accomplished by referencing the accessor in the entity association.

For example, if you define an entity with a master-detail association for `Dept` and `Emp`, by default the accessor for the entity association will be named `Dept` and `Emp`, to identity the source and destination data source. Using that accessor in a Groovy expression to set the default value for a new employee's salary based on the location of their department:

```
adf.object.getDefaultSalaryForGrade(Dept.Loc)
```

This expression does not reference the entity even though it has the same name (`Dept`) as the accessor for the association. Instead, assuming a master-detail relationship between departments and employees, referencing the accessor allows the Groovy expression for the employee entity object to walk back to the master department entity and pass in the value of `Loc` from that master.

3.6.3 Manipulating Business Component Attribute Values in Groovy Expressions

You can use the following built-in aggregate functions on Oracle Business Components `RowSet` objects:

- *`rowSetAttr`*`.sum(`*`GroovyExpr`*`)`
- *`rowSetAttr`*`.count(`*`GroovyExpr`*`)`
- *`rowSetAttr`*`.avg(`*`GroovyExpr`*`)`
- *`rowSetAttr`*`.min(`*`GroovyExpr`*`)`
- *`rowSetAttr`*`.max(`*`GroovyExpr`*`)`

These aggregate functions accept a string-value argument that is interpreted as a Groovy expression that is evaluated in the context of each row in the row set as the aggregate is being computed. The Groovy expression must return a numeric value (or number domain).

For example, in a `Dept` entity object you could add a transient attribute that displays the sum of all employee salaries that is calculated by this expression:

```
EmployeesInDept.sum("Sal")
```

To reference the employees of a specific department, the expression supplies the name of the master-detail association's accessor for the destination `Emp` entity. In this case, the accessor is `EmployeesInDept` and salary is interpreted for each record of the `Emp` entity object.

Or, assume that you want the calculation of the salary total for specific departments to include each employee's benefits package, which varies with job role:

```
EmployeesInDept.sum("Sal + adf.object.getBenefitsValue(Job)")
```

4

Creating a Business Domain Layer Using Entity Objects

This chapter describes how to use entity objects to create a reusable layer of business domain objects for use in your Java EE applications.

This chapter includes the following sections:

- Section 4.1, "About Entity Objects"
- Section 4.2, "Creating Entity Objects and Associations"
- Section 4.3, "Creating and Configuring Associations"
- Section 4.4, "Creating an Entity Diagram for Your Business Layer"
- Section 4.5, "Defining Property Sets"
- Section 4.6, "Defining Attribute Control Hints for Entity Objects"
- Section 4.7, "Working with Resource Bundles"
- Section 4.8, "Defining Business Logic Groups"
- Section 4.9, "Configuring Runtime Behavior Declaratively"
- Section 4.10, "Setting Attribute Properties"
- Section 4.11, "Adding Transient and Calculated Attributes to an Entity Object"
- Section 4.12, "Creating Business Events"
- Section 4.13, "Generating Custom Java Classes for an Entity Object"
- Section 4.14, "Working Programmatically with Entity Objects and Associations"
- Section 4.15, "Creating Custom, Validated Data Types Using Domains"
- Section 4.16, "Creating New History Types"
- Section 4.17, "Basing an Entity Object on a PL/SQL Package API"
- Section 4.18, "Basing an Entity Object on a Join View or Remote DBLink"
- Section 4.19, "Using Inheritance in Your Business Domain Layer"

4.1 About Entity Objects

An entity object is the ADF Business Components component that represents a row in the specified data source and simplifies modifying its associated attributes. Importantly, it allows you to encapsulate domain business logic to ensure that your business policies and rules are consistently validated.

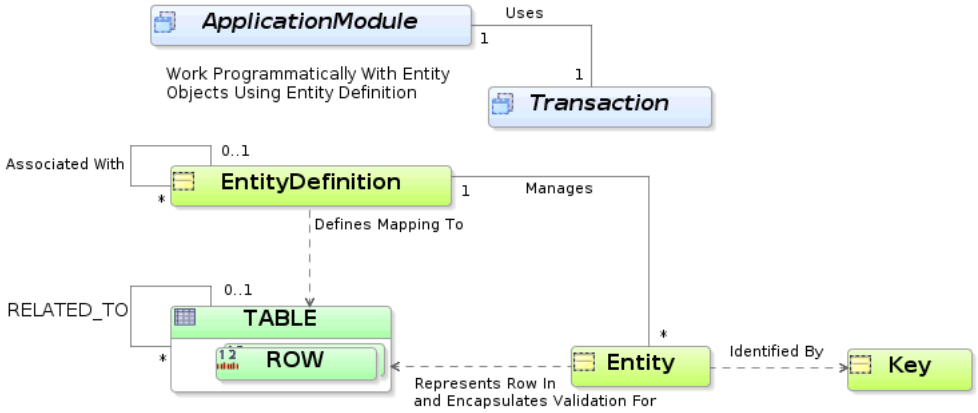
4.1.1 Entity Object Use Cases and Examples

Entity objects support numerous declarative business logic features to enforce the validity of your data. You will typically complement declarative validation with additional custom application logic and business rules to cleanly encapsulate a maximum amount of domain business logic into each entity object. Your associated set of entity objects forms a reusable business domain layer that you can exploit in multiple applications.

The key concepts of entity objects (as illustrated in Figure 4–1) are the following:

- You define an entity object by specifying the database table whose rows it will represent.
- You can create associations to reflect relationships between entity objects.
- At runtime, entity rows are managed by a related entity definition object.
- Each entity row is identified by a related row key.
- You retrieve and modify entity rows in the context of an application module that provides the database transaction.

Figure 4–1 Entity Object Encapsulates Business Logic for a Table



4.1.2 Additional Functionality for Entity Objects

You may find it helpful to understand other ADF features before you start working with entity objects. Following are links to other functionality that may be of interest.

- For information about using declarative validation in entity objects, see Chapter 7, "Defining Validation and Business Rules Declaratively."
- For API documentation related to the `oracle.jbo` package, see the following Javadoc reference document:
  - *Oracle Fusion Middleware Java API Reference for Oracle ADF Model*

4.2 Creating Entity Objects and Associations

If you already have a database schema to work from, the simplest way to create entity objects and associations is to reverse-engineer them from existing tables. When needed, you can also create an entity object from scratch, and then generate a table for it later.

4.2.1 How to Create Multiple Entity Objects and Associations from Existing Tables

To create one or more entity objects, use the Business Components from Tables wizard, which is available from the New Gallery.

Before you begin:

It may be helpful to have an understanding of the options you have for creating entity objects. For more information, see Section 4.2, "Creating Entity Objects and Associations."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To create one or more entity objects and associations from existing tables:

1. In the Application Navigator, right-click the project in which you want to create the entity objects and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Business Components from Tables**, and click **OK**.

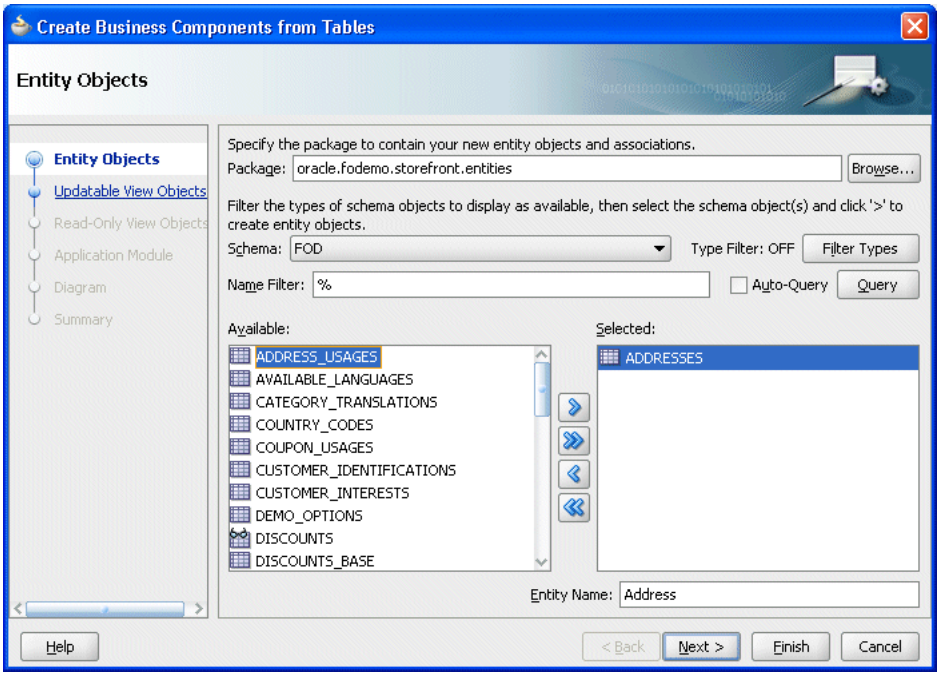
   If this is the first component you're creating in the project, the Initialize Business Components Project dialog appears to allow you to select a database connection.
3. In the Initialize Business Components Project dialog, select the database connection or choose **New** to create a connection. Click **OK**.
4. On the Entity Objects page, do the following to create the entity objects:
   - Enter the package name in which all of the entity objects will be created.
   - Select the tables from the **Available** list for which you want to create entity objects.

     If the **Auto-Query** checkbox is selected, then the list of available tables appears immediately. In the **Name Filter** field, you can optionally enter a full or partial table name to filter the available tables list in real time. As an alternative to the auto-query feature, click the **Query** button to retrieve the list based on an optional table name filter. When no name filter is entered, JDeveloper retrieves all table objects for the chosen schema.
   - Click **Filter Types** if you want to see only a subset of the database objects available. You can filter out tables, views, or synonyms.

   Once you have selected a table from the **Available** list, the proposed entity object name for that table appears in the **Selected** list with the related table name in parenthesis.
   - Select an entity object name in the **Selected** list and use the **Entity Name** field to change the default entity object name.

     Best Practice: Because each entity object instance represents a *single* row in a particular table, name the entity objects with a singular noun (like Address, Order, and Person), instead of their plural counterparts. Figure 4–2 shows what the wizard page looks like after selecting the `ADDRESSES` table in the `FOD` schema, setting a package name of `oracle.fodemo.storefront.entities`, and renaming the entity object in the singular.

Figure 4–2 Create Business Components from Tables Wizard, Entity Objects Page

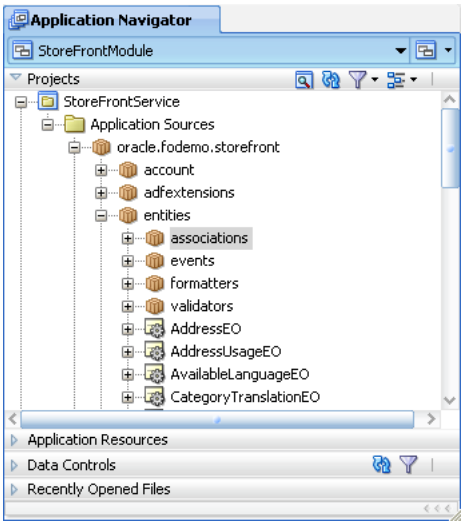


5. When you are satisfied with the selected table objects and their corresponding entity object names, click **Finish**.

The Application Navigator displays the entity objects in the package you specified.

> **Best Practice:** After you create associations, move all of your associations to a separate package so that you can view and manage them separately from the entity objects. In Figure 4–3, the associations have been moved to a subpackage (`associations`) and do not appear in the `entities` package in the Application Navigator. For more information, see Section 4.3.4, "How to Rename and Move Associations to a Different Package."

Figure 4–3 New Entity Objects in Application Navigator



4.2.2 How to Create Single Entity Objects Using the Create Entity Wizard

To create a single entity object, you can use the Create Entity Object wizard, which is available in the New Gallery.

> **Note:** Associations are not generated when you use the Create Entity Object wizard. However, the Business Components from Tables wizard does generate associations. If you use the Create Entity Object wizard to create entity objects, you will need to create the corresponding associations manually.

Before you begin:

It may be helpful to have an understanding of the options you have for creating entity objects. For more information, see Section 4.2, "Creating Entity Objects and Associations."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To create a single entity object:

1. In the Application Navigator, right-click the project in which you want to create the entity object and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Entity Object**, and click **OK**.

   If this is the first component you're creating in the project, the Initialize Business Components Project dialog appears to allow you to select a database connection.
3. In the Initialize Business Components Project dialog, select the database connection or choose **New** to create a connection. Click **OK**.
4. On the Name page, do the following to create the entity object:
   - Enter a name for the entity object.
   - Enter the package name in which the entity object will be created.
   - Click **Browse** (next to the **Schema Object** field) to select the table for which you want to create the entity object.

     Or, if you plan to create the table later, you can enter a name of a table that does not exist.
5. If you manually entered a table name in the **Schema Object** field, you will need to define each attribute on the Attributes page of the wizard. Click **Next**.

   You can create the table manually or generate it, as described in Section 4.2.6, "How to Create Database Tables from Entity Objects."
6. When you are satisfied with the table object and its corresponding entity object name, click **Finish**.

4.2.3 What Happens When You Create Entity Objects and Associations from Existing Tables

When you create an entity object from an existing table, first JDeveloper interrogates the data dictionary to infer the following information:

- The Java-friendly entity attribute names from the names of the table's columns (for example, `USER_ID` -> `UserId`)
- The SQL and Java data types of each attribute based on those of the underlying column
- The length and precision of each attribute
- The primary and unique key attributes
- The mandatory flag on attributes, based on `NOT NULL` constraints
- The relationships between the new entity object and other entities based on foreign key constraints

> **Note:** Since an entity object represents a database row, it seems natural to call it an *entity row*. Alternatively, since at runtime the entity row is an instance of a Java object that encapsulates business logic for that database row, the more object-oriented term *entity instance* is also appropriate. Therefore, these two terms are interchangeable.

JDeveloper then creates the XML component definition file that represents its declarative settings and saves it in the directory that corresponds to the name of its package. For example, when an entity named `Order` appears in the `genericbcmodel.entities` package, JDeveloper will create the XML file `genericbcmodel/entities/Order.xml` under the project's source path. This XML file contains the name of the table, the names and data types of each entity attribute, and the column name for each attribute.

You can inspect the XML description for the entity object by opening the object in the overview editor and clicking the **Source** tab.

> **Note:** If your IDE-level Business Components Java generation preferences so indicate, the wizard may also create an optional custom entity object class (for example, `OrderImpl.java`).

4.2.3.1 What Happens When Tables Have Foreign Key Relationships

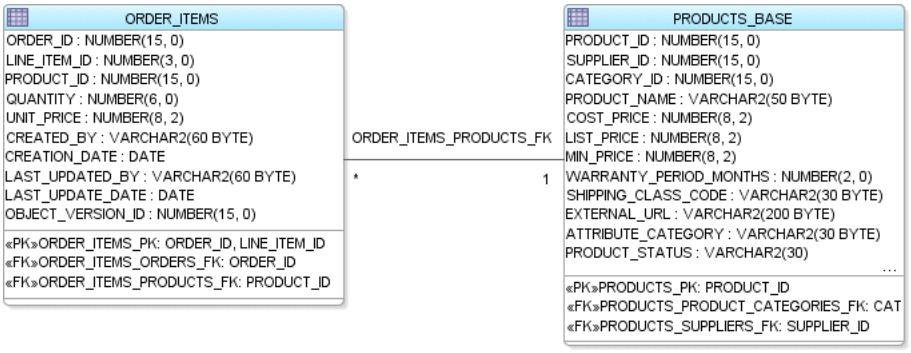
In addition to the entity objects, the Business Components from Tables wizard also generates named association components that capture information about the relationships between entity objects. For example, the database diagram in Figure 4–4 shows that JDeveloper derives default association names like `OrderItemsProductsFkAssoc` by converting the foreign key constraint names to a Java-friendly name and adding the `Assoc` suffix. For each association created, JDeveloper creates an appropriate XML component definition file and saves it in the directory that corresponds to the name of its package.

> **Note:** Associations are generated when you use the Business Components from Tables wizard. However, the Create Entity Object wizard does not generate associations. If you use the Create Entity Object wizard to create entity objects, you will need to create the corresponding associations manually.

By default the associations reverse-engineered from foreign keys are created in the same package as the entities. For example, for the association

`OrderItemsProductsFkAssoc` with entities in the `fodemo.storefront.entities` package, JDeveloper creates the association XML file named `./fodemo/storefront/entities/OrderItemsProductsFkAssoc.xml`.

Figure 4–4 ORDER_ITEMS and PRODUCTS_BASE Tables Related by Foreign Key



4.2.3.2 What Happens When a Table Has No Primary Key

If a table has no primary key constraint, then JDeveloper cannot infer the primary key for the entity object. Since every entity object must have at least one attribute marked as a primary key, the wizard will create an attribute named `RowID` and use the database `ROWID` value as the primary key for the entity. If appropriate, you can edit the entity object later to mark a different attribute as a primary key and remove the `RowID` attribute. When you use the Create Entity Object wizard and you have not set any other attribute as primary key, you will be prompted to use `RowID` as the primary key.

4.2.4 What Happens When You Create an Entity Object for a Synonym or View

When you create an entity object using the Business Components from Tables wizard or the Create Entity Object wizard, the object can represent an underlying table, synonym, or view. The framework can infer the primary key and related associations for a table or synonym by inspecting database primary and foreign key constraints in the data dictionary.

However, when your selected schema object is a database view, then neither the primary key nor associations can be inferred since database views do not have database constraints. In this case, if you use the Business Components from Tables wizard, the primary key defaults to `RowID`. If you use the Create Entity Object wizard, you'll need to specify the primary key manually by marking at least one of its attributes as a primary key. For more information, see Section 4.2.3.2, "What Happens When a Table Has No Primary Key."

When your selected schema object is a synonym, there are two possible outcomes. If the synonym is a synonym for a table, then the wizard and editor behave as if you had specified a table. If instead the synonym refers to a database view, then they behave as if you had specified a view.

4.2.5 How to Edit an Existing Entity Object or Association

After you've created a new entity object or association, you can edit any of its settings in the overview editor. To launch the editor, choose **Open** from the context menu for the entity object or association in the Application Navigator or double-click the object.

By clicking the different tabs of the editor, you can adjust the settings that define the object and govern its runtime behavior.

4.2.6 How to Create Database Tables from Entity Objects

To create database tables based on entity objects, right-click the package in the Application Navigator that contains the entity objects and choose **Create Database Objects** from the context menu. A dialog appears to let you select the entities whose tables you'd like to create. This tool can be used to generate a table for an entity object you created from scratch, or to drop and re-create an existing table.

> **Caution:** This feature does *not* generate a DDL script to run later. It performs its operations directly against the database and will drop existing tables. A dialog appears to confirm that you want to do this before proceeding. For entities based on existing tables, use with caution.

In the overview editor for an association, the **Use Database Key Constraints** checkbox on the Association Properties page controls whether the related foreign key constraint will be generated when creating the tables for entity objects. Selecting this option does not have any runtime implications.

4.2.7 How to Synchronize an Entity with Changes to Its Database Table

Inevitably you (or your DBA) might alter a table for which you've already created an entity object. Your existing entity will not be disturbed by the presence of additional attributes in its underlying table; however, if you want to access the new column in the table in your Java EE application, you'll need to synchronize the entity object with the database table.

For example, suppose you had done the following at the SQL*Plus command prompt to add a new `SECURITY_QUESTION` column to the `PERSONS` table:

```
ALTER TABLE PERSONS ADD (security_question VARCHAR2(60));
```

Then you can use the synchronization feature to add the new column as an attribute on the entity object.

Before you begin:

It may be helpful to have an understanding of the options you have for creating entity objects. For more information, see Section 4.2, "Creating Entity Objects and Associations."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To synchronize an entity with changes to its database table:

1. In the Application Navigator, right-click the desired entity object and choose **Synchronize with Database**.

   The Synchronize with Database dialog shows the list of the actions that can be taken to synchronize the business logic tier with the database.

2. Select the action you want to take:

- Select one or more actions from the list, and click **Synchronize** to synchronize the selected items.
- Click **Synchronize All** to perform all actions in the list.
- Click **Write to File** to save the action list to a text file. This feature helps you keep track of the changes you make.

3. When finished, click **OK** to close the dialog.

4.2.7.1 Removing an Attribute Associated with a Dropped Column

The synchronize feature does not handle dropped columns. When a column is dropped from the underlying database after an entity object has been created, you can delete the corresponding attribute from the entity object. If the attribute is used in other parts of your application, you must remove those usages as well.

Before you begin:

It may be helpful to have an understanding of the options you have for creating entity objects. For more information, see Section 4.2, "Creating Entity Objects and Associations."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To remove an entity attribute:

1. In the Application Navigator, double-click the entity.
2. In the overview editor, click the **Attributes** navigation tab.
3. On the Attributes page, right-click the attribute, and choose **Delete Safely**.

   If there are other usages, the Delete Attributes dialog displays the message "Usages were found."
4. If usages were found, click **View Usages**.

   The Log window shows all usages of the attribute.
5. Work through the list in the Log window to delete all usages of the entity attribute.

4.2.7.2 Addressing a Data Type Change in the Underlying Table

The synchronize feature does not handle changed data types. For a data type change in the underlying table (for example, precision increased), you must locate all usages of the attribute and manually make changes, as necessary.

Before you begin:

It may be helpful to have an understanding of the options you have for creating entity objects. For more information, see Section 4.2, "Creating Entity Objects and Associations."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To locate all usages of an entity attribute:

1. In the Application Navigator, double-click the entity.
2. In the overview editor, click the **Attributes** navigation tab.

3. On the Attributes page, right-click the attribute and choose **Find Usages**.

   If there are other usages, they are displayed in the Log window.

4.2.8 How to Store Data Pertaining to a Specific Point in Time

Effective dated tables are used to provide a view into the data set pertaining to a specific point in time. Effective dated tables are widely used in applications like HRMS and Payroll to answer queries like:

- What was the tax rate for an employee on August 31st, 2005?
- What are the employee's benefits as of October 2004?

In either case, the employee's data may have changed to a different value since then.

The primary difference between the effective dated entity type and the dated entity type is that the dated entity does not cause row splits during update and delete.

When you create an effective dated entity object, you identify the entity as effective dated and specify the attributes of the entity that represent the start and end dates. The start date and end date attributes must be of the Date type.

Additionally, you can specify an attribute that represents the sequence for the effective dated entity and an attribute that represents a flag for the sequence. These attributes allow for tracking of multiple changes in a single day.

Before you begin:

It may be helpful to have an understanding of the options you have for creating entity objects. For more information, see Section 4.2, "Creating Entity Objects and Associations."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To create an effective dated entity object

1. In the Application Navigator, double-click the entity on which you want enable effective dating.
2. In the Property Inspector, expand the **Type** category.

   If necessary, choose **Property Inspector** from the **View** menu to display the Property Inspector.

   If the **Type** category is not displayed in the Property Inspector, click the **General** tab in the overview editor to set the proper focus.
3. From the context menu for the **Effective Date Type** property, choose **Edit**.

   To display the context menu, click the down arrow next to the property field.
4. In the Edit Property dialog, specify the following settings:
   - For **Effective Date Type**, select **EffectiveDated**.
   - For **Start Date Attribute**, select the attribute that corresponds to the start date.
   - For **End Date Attribute**, select the attribute that corresponds to the end date.
5. You can optionally specify attributes that allow for tracking of multiple changes in a single day.

- For **Effective Date Sequence**, select the attribute that stores the sequence of changes.
- For **Effective Date Sequence Flag**, select the attribute that stores a flag indicating the most recent change in the sequence.

Without specifying the **Effective Date Sequence** and **Effective Date Sequence Flag** attributes, the default granularity of effective dating is one day. For this reason, multiple changes in a single day are not allowed. An attempt to update the entity a second time in a single day will result in an exception being thrown. After these two attributes are specified, the framework inserts and updates their values as necessary to track multiple changes in a single day.

6. Click **OK**.

Note: You can also identify the start and end date attributes using the Property Inspector for the appropriate attributes. To do so, select the appropriate attribute in the overview editor and set the **IsEffectiveStartDate** or **IsEffectiveEndDate** property to **true** in the Property Inspector.

4.2.9 What Happens When You Create Effective Dated Entity Objects

When you create an effective dated entity object, JDeveloper creates a transient attribute called `SysEffectiveDate` to store the effective date for the row. Typically the Insert, Update, and Delete operations modify the transient attribute while the ADF Business Components framework decides the appropriate values for the effective start date and the effective end date.

Example 4–1 show some sample XML entries that are generated when you create an effective dated entity. For more information about working with effective dated objects, see Section 5.4, "Limiting View Object Rows Using Effective Date Ranges."

Example 4–1 XML Entries for Effective Dated Entities

```
// In the effective dated entity
<Entity
  ...
  EffectiveDateType="EffectiveDated">

// In the attribute identified as the start date
  <Attribute
    ...
    IsEffectiveStartDate="true">

// In the attribute identified as the end date
  <Attribute
    ...
    IsEffectiveEndDate="true">

// The SysEffectiveDate transient attribute
  <Attribute
    Name="SysEffectiveDate"
    IsQueriable="false"
    IsPersistent="false"
    ColumnName="$none$"
    Type="oracle.jbo.domain.Date"
    ColumnType="$none$"
    SQLType="DATE"/>
```

4.2.10 What You May Need to Know About Creating Entities from Tables

The Business Components from Tables wizard makes it easy to quickly generate many business components at the same time. In practice, this does not mean that you should use it to immediately create entity objects for every table in your database schema just because it is possible to do so. If your application requires all of the tables, then that strategy might be appropriate. But because you can use the wizard whenever needed, you should create the entity objects for the tables that you know will be involved in the application.

Section 9.4, "Defining Nested Application Modules," describes a use case-driven design approach for your business services that can assist you in understanding which entity objects are required to support your application's business logic needs. You can always add more entity objects later as necessary.

4.3 Creating and Configuring Associations

If your database tables have no foreign key constraints defined, JDeveloper won't be able to infer the associations between the entity objects that you create. Since several ADF Business Components runtime features depend on the presence of entity associations, create them manually if the foreign key constraints don’t exist.

4.3.1 How to Create an Association

To create an association, use the Create New Association wizard, which is available in the New Gallery.

Before you begin:

It may be helpful to have an understanding of why you create associations. For more information, see Section 4.3, "Creating and Configuring Associations."

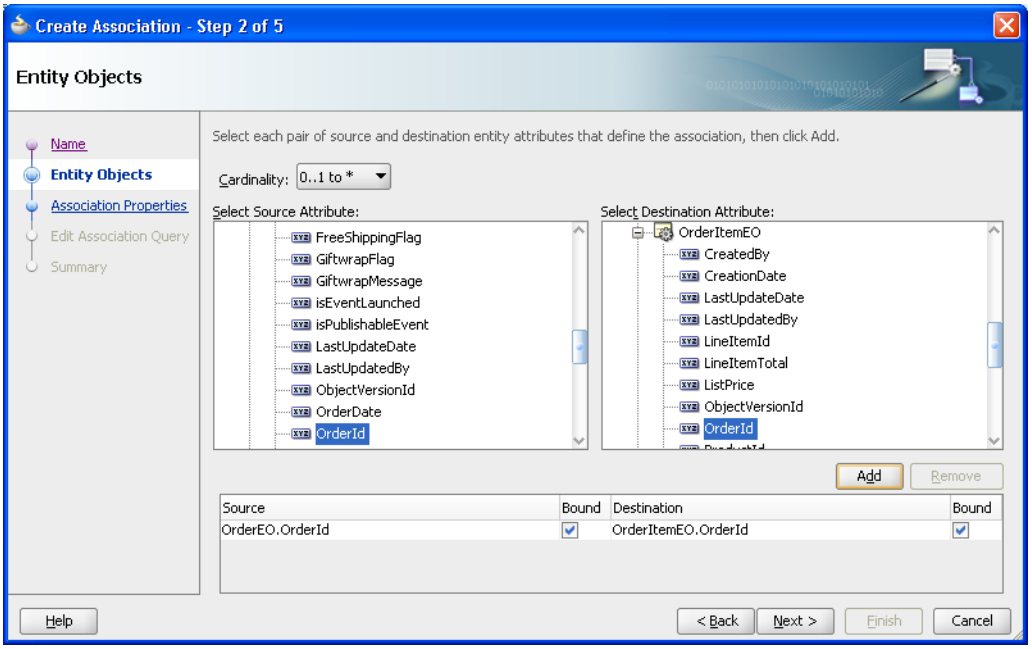
You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To create an association:

1. In the Application Navigator, right-click the project in which you want to create the association and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Association**, and click **OK**.
3. On the Name page, do the following to create the association:
   - Enter the package name in which the association will be created.
   - Enter the name of the association component.
   - Click **Next**.
4. On the Entity Objects page, select the source and destination entity attributes:
   - Select a source attribute from one of the entity objects that is involved in the association to act as the master.
   - Select a corresponding destination attribute from the other entity object involved in the association.

For example, Figure 4–5 shows the selected `OrderId` attribute from the `OrderEO` entity object as the source entity attribute. Because the `OrderItemEO` rows contain an order ID that relates them to a specific `OrderEO` row, you would select this `OrderId` foreign key attribute in the `OrderItemEO` entity object as the destination attribute.

Figure 4–5 Create Association Wizard, Attribute Pairs That Relate Two Entity Objects Defined



5. Click **Add** to add the matching attribute pair to the table of source and destination attribute pairs below.

   By default, the **Bound** checkbox is selected for both the source and destination attribute. This checkbox allows you to specify whether or not the value will be bound into the association SQL statement that is created internally when navigating from source entity to target entity or from target entity to source entity (depending on which side you select).

   Typically, you would deselect the checkbox for an attribute in the relationship that is a transient entity attribute whose value is a constant and therefore should not participate in the association SQL statement to retrieve the entity.

6. If the association requires multiple attribute pairs to define it, you can repeat the preceding steps to add additional source/target attribute pairs.

7. Finally, ensure that the **Cardinality** dropdown correctly reflects the cardinality of the association. The default is a one-to-many relationship. Click **Next**.

   For example, since the relationship between a `OrderEO` row and its related `OrderItemEO` rows is one-to-many, you can leave the default setting.

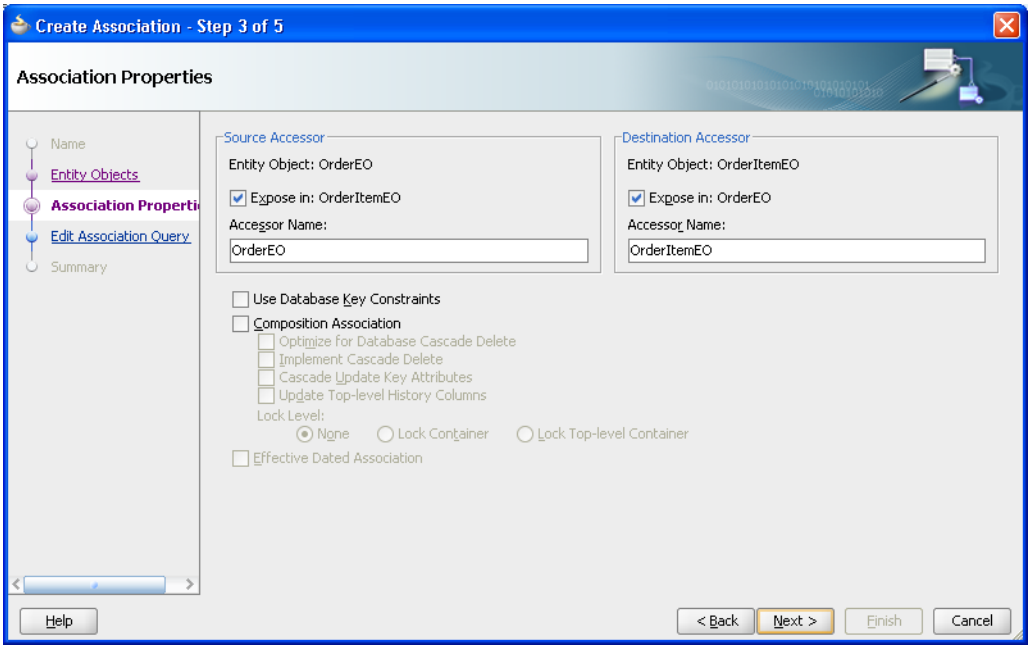
8. On the Association SQL page, you can preview the association SQL predicate that will be used at runtime to access the related destination entity objects for a given instance of the source entity object.

9. On the Association Properties page, disable the **Expose Accessor** checkbox on either the **Source** or the **Destination** entity object when you want to create an association that represents a one-way relationship. The default, bidirectional

navigation is more convenient for writing business validation logic, so in practice, you typically leave these default checkbox settings.

For example, Figure 4–6 shows an association that represents a bidirectional relationship, permitting either entity object to access the related entity row(s) on the other side when needed. In this example, this means that if you are working with an instance of an `OrderEO` entity object, you can easily access the collection of its related `OrderItemEO` rows. With any instance of a `OrderItemEO` entity object, you can also easily access the `Order` to which it belongs.

Figure 4–6 Association Properties Control Runtime Behavior



10. When you are satisfied with the association definition, click **Finish**.

4.3.2 What Happens When You Create an Association

When you create an association, JDeveloper creates an appropriate XML component definition file and saves it in the directory that corresponds to the name of its package. For example, if you created an association named `OrderItemsOrdersFkAssoc` in the `oracle.fodemo.storefront.entities.associations` subpackage, then the association XML file would be created in the `./oracle/fodemo/storefront/entities/associations` directory with the name `OrderItemsOrdersFkAssoc.xml`. At runtime, the entity object uses the association information to automate working with related sets of entities.

4.3.3 How to Change Entity Association Accessor Names

You should consider the default settings for the accessor names on the Association Properties page and decide whether changing the names to something more intuitive is appropriate. The default settings define the names of the accessor attributes you will use at runtime to programmatically access the entities on the other side of the relationship. By default, the accessor names will be the names of the entity object on the other side. Since the accessor names on an entity must be unique among entity object attributes and other accessors, if one entity is related to another entity in *multiple*

ways, then the default accessor names are modified with a numeric suffix to make the name unique.

In an existing association, you can rename the accessor using the Association Properties dialog.

Before you begin:

It may be helpful to have an understanding of why you create associations. For more information, see Section 4.3, "Creating and Configuring Associations."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To rename the entity accessor in an association:

1. In the Application Navigator, double-click the association.
2. In the overview editor, click the **Relationships** navigation tab.
3. On the Relationships page, expand the **Accessors** category and click the **Edit** icon.

   The Association Properties dialog displays the current settings for the association's accessors.
4. Modify the name as necessary, and click **OK** to apply your changes and close the dialog.

4.3.4 How to Rename and Move Associations to a Different Package

Since associations are a component that you typically configure at the outset of your project and don't change frequently thereafter, you might want to move the associations to a different package so that your entity objects are easier to see. Both renaming components and moving them to a different package is straightforward using JDeveloper's refactoring functionality.

Before you begin:

It may be helpful to have an understanding of why you create associations. For more information, see Section 4.3, "Creating and Configuring Associations."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To move a set of business components to a different package:

1. In the Application Navigator, select the components you want to move.
2. Right-click one of the selected components, and choose **Refactor > Move**.
3. In the Move Business Components dialog, enter the name of the package to move the component(s) to, or click **Browse** to navigate to and select the package.
4. Click **OK** to apply your changes and close the dialog.

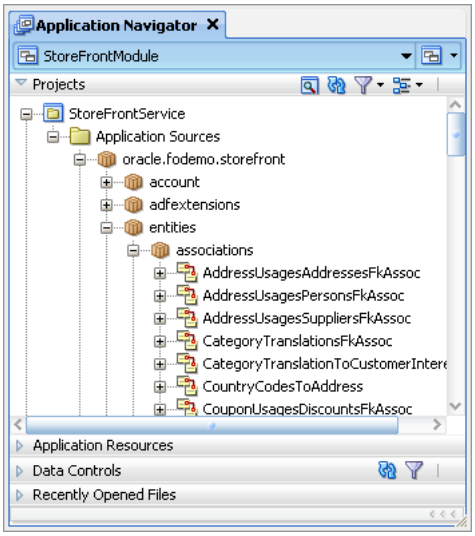
To rename a component:

1. In the Application Navigator, right-click the component you want to rename, and choose **Refactor > Rename**.
2. In the Rename dialog, enter the new name for the component and click **OK**.

When you refactor ADF Business Components, JDeveloper moves the XML and Java files related to the components, and updates any other components that might reference them.

Figure 4–7 shows what the Application Navigator would look like after renaming all of the associations and moving them to the `oracle.fodemo.storefront.associations` subpackage. While you can refactor the associations into any package name you choose, picking a subpackage keeps them logically related to the entities, and allows you to collapse the package of associations to better manage which files display in the Application Navigator.

Figure 4–7 Application Navigator After Association Refactoring



4.3.5 What You May Need to Know About Using a Custom View Object in an Association

You can associate a custom view object with the source end or destination end (or both) of an entity association.

When you traverse entity associations in your code, if the entities are not already in the cache, then the ADF Business Components framework performs a query to bring the entity (or entities) into the cache. By default, the query performed to bring an entity into the cache is the find-by-primary-key query that selects values for all persistent entity attributes from the underlying table. If the application performs a lot of programmatic entity association traversal, you could find that retrieving all of the attributes might be heavy-handed for your use cases.

Entity associations support the ability to associate a custom, entity-based view object with the source entity or destination entity in the association, or both. The primary entity usage of the entity-based view object you supply must match the entity type of the association end for which you use it.

Using a custom view object can be useful because the custom view object's query can include fewer columns and it can include an `ORDER BY` clause. This allows you to control how much data is retrieved when an entity is brought into the cache due to association traversal, as well as the order in which any collections of related entities will appear.

For more information about creating a custom view object, see Section 42.8.2, "How to Create an Entity-Based Programmatic View Object."

4.3.6 What You May Need to Know About Composition Associations

A association represents a relationship between entities, such as `Person` referenced by an `Order` or an `OrderItem` contained in an `Order`. When you create associations, it is useful to know about the kinds of relationships you can represent, and the various options.

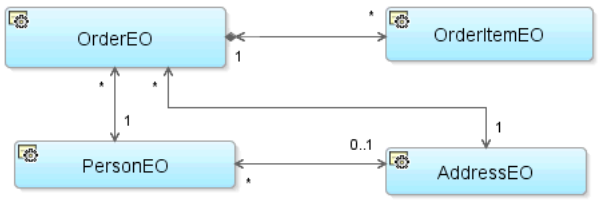
Associations between entity objects can represent two styles of relationships depending on whether the source entity:

- *References* the destination entity
- *Contains* the destination entity as a logical, nested part

Figure 4–8 depicts an application business layer that represents both styles of relationships. For example, an `OrderEO` entry references a `PersonEO`. This relationship represents the first kind of association, reflecting that a `PersonEO` or an `OrderEO` entity object can exist independent from each other. In addition, the removal of an `Order` does not imply the cascade removal of the `Person` to which it was referring.

In contrast, the relationship between `OrderEO` and its collection of related `OrderItemEO` details is stronger than a simple reference. The `OrderItemEO` entries comprise a logical part of the overall `OrderEO`. In other words, an `OrderEO` is composed of `OrderItemEO` entries. It does not make sense for an `OrderItemEO` entity row to exist independently from an `OrderEO`, and when an `OrderEO` is removed — assuming the removal is allowed — all of its composed parts should be removed as well. This kind of logical containership represents the second kind of association, called a *composition*. The UML diagram in Figure 4–8 illustrates the stronger composition relationship using the solid diamond shape on the side of the association which composes the other side of the association.

Figure 4–8 OrderEO Composed of OrderItemEO Entries and References Both PersonEO and AddressEO



The Business Components from Tables Wizard creates composition associations by default for any foreign keys that have the `ON DELETE CASCADE` option. You can use the Create Association wizard or the overview editor for the association to indicate that an association is a composition association. Select the **Composition Association** checkbox on either the Association Properties page of the Create Association wizard or the Relationships page of the overview editor. An entity object offers additional runtime behavior in the presence of a composition. For the settings that control the behavior, see Section 4.10.13, "How to Configure Composition Behavior."

4.4 Creating an Entity Diagram for Your Business Layer

Since your layer of business domain objects represents a key reusable asset for your team, it is often convenient to visualize the business domain layer using a UML model. JDeveloper supports easily creating a diagram for your business domain layer that you and your colleagues can use for reference.

The UML diagram of business components is not just a static picture that reflects the point in time when you dropped the entity objects onto the diagram. Rather, it is a UML-based rendering of the current component definitions, that will always reflect the current state of affairs. What's more, the UML diagram is both a visualization aid and a visual navigation and editing tool. To open the overview editor for any entity object in a diagram, right-click the desired object and choose **Properties** from the context menu or double-click the desired object. You can also perform some entity object editing tasks directly on the diagram, like renaming entities and entity attributes, and adding or removing attributes.

4.4.1 How to Create an Entity Diagram

To create a diagram of your entity objects, you can use the Create Business Components Diagram dialog, which is available in the New Gallery.

Before you begin:

It may be helpful to have an understanding of how entity diagrams are used in the application. For more information, see Section 4.4, "Creating an Entity Diagram for Your Business Layer."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To create an entity diagram that models existing entity objects:

1. In the Application Navigator, right-click the project in which you want to create the entity diagram and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Business Components Diagram**, and click **OK**.
3. In the dialog, do the following to create the diagram:
   - Enter a name for the diagram, for example `Business Domain Objects`.
   - Enter the package name in which the diagram will be created. For example, you might create it in a subpackage like `myproject.model.design`.
4. Click **OK**.
5. To add existing entity objects to the diagram, select them in the Application Navigator and drop them onto the diagram surface.

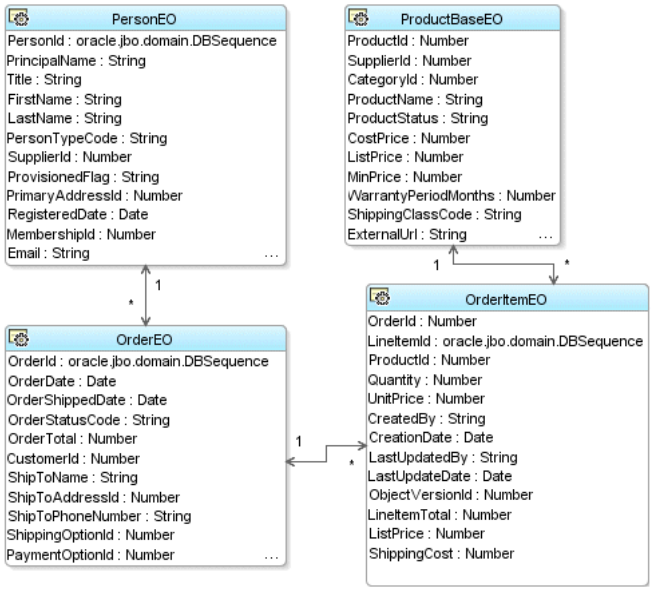
After you have created the diagram you can use the Property Inspector to adjust visual properties of the diagram. For example you can:

- Hide or show the package name
- Change the font
- Toggle the grid and page breaks on or off
- Display association names that may otherwise be ambiguous

You can also create an image of the diagram in `PNG`, `JPG`, `SVG`, or compressed `SVG` format, by choosing **Publish Diagram** from the context menu on the diagram surface.

Figure 4–9 shows a sample diagram that models various entity objects from the business domain layer.

Figure 4–9 UML Diagram of Business Domain Layer

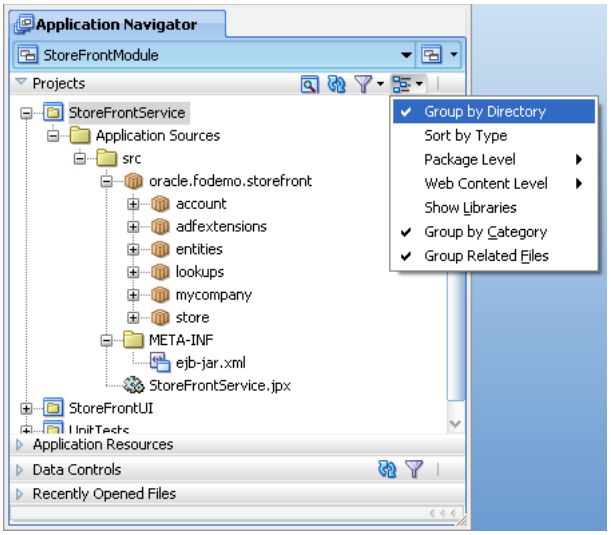


4.4.2 What Happens When You Create an Entity Diagram

When you create a business components diagram, JDeveloper creates an XML file `*.oxd_bc4j` representing the diagram in a subdirectory of the project's model path that matches the package name in which the diagram resides.

By default, the Application Navigator unifies the display of the project contents paths so that ADF components and Java files in the source path appear in the same package tree as the UML model artifacts in the project model path. However, as shown in Figure 4–10, using the **Navigator Display Options** toolbar button on the Application Navigator, you can see the distinct project content path root directories when you prefer.

Figure 4–10 Toggling the Display of Separate Content Path Directories



4.4.3 What You May Need to Know About the XML Component Descriptors

When you include a business component like an entity object to a UML diagram, JDeveloper adds extra metadata to a `<Data>` section of the component's XML component descriptor as shown in Example 4–2. This additional information is used at design time only.

Example 4–2 Additional UML Metadata Added to an Entity Object XML Descriptor

```
<Entity Name="OrderEO" ... >
   <Data>
      <Property Name ="COMPLETE_LIBRARY" Value ="FALSE" />
      <Property Name ="ID"
                Value ="ff16fca0-0109-1000-80f2-8d9081ce706f::::EntityObject" />
      <Property Name ="IS_ABSTRACT" Value ="FALSE" />
      <Property Name ="IS_ACTIVE" Value ="FALSE" />
      <Property Name ="IS_LEAF" Value ="FALSE" />
      <Property Name ="IS_ROOT" Value ="FALSE" />
      <Property Name ="VISIBILITY" Value ="PUBLIC" />
   </Data>
   :
</Entity>
```

4.4.4 What You May Need to Know About Changing the Names of Components

On an entity diagram, the names of entity objects, attributes, and associations can be changed for clarity. Changing names on a diagram does not affect the underlying data names. The name change persists for the diagram only. The new name may contain spaces and mixed case for readability. To change the actual entity object names, attribute names, or association names, open the entity object or association in the overview editor.

4.5 Defining Property Sets

A property set is a named collection of properties, where a property is defined as a name/value pair. Property sets are a convenience mechanism to group properties and then reference them from other ADF Business Components objects. Properties defined in a property set can be configured to be translatable, in which case the translations are stored in a message bundle file owned by the property set.

Property sets can be used for a variety of functions, such as control hints and error messages. A property set may contain control hints and other custom properties, and you can associate them with multiple attributes of different objects.

Note: Take care when defining property sets that contain translatable content. Be sure not to "overload" common terms in different contexts. For example, the term "Name" might be applied to both an object and a person in one language, but then translated into two different terms in a target language. Even though a term in several contexts might be the same in the source language, a separate distinguishable term should be used for each context.

Property sets can be used with entity objects and their attributes, view objects and their attributes, and application modules.

4.5.1 How to Define a Property Set

To define a property set, you create a new property set using a dialog and then specify properties using the Property Inspector.

Before you begin:

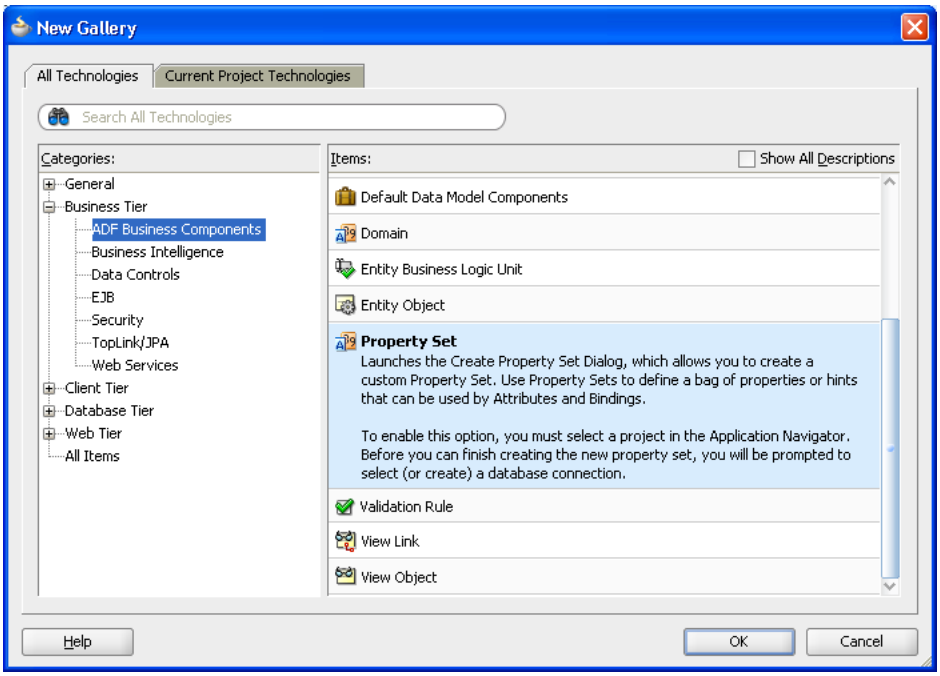
It may be helpful to have an understanding of how property sets can be used. For more information, see Section 4.5, "Defining Property Sets."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To define a property set:

1. In the Application Navigator, right-click the project where you want to create the property set, and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Property Set**, and click **OK**.

Figure 4–11 Property Set in New Gallery



3. In the Create Property Set dialog, enter the name and location of the property set and click **OK**.
4. From the **View** menu, choose **Property Inspector**.
5. In the Property Inspector, define the properties for the property set.

4.5.2 How to Apply a Property Set

After you have created the property set, you can apply the property set to an entity object or attribute, and use the defined properties (or override them, if necessary).

Before you begin:

It may be helpful to have an understanding of how property sets can be used. For more information, see Section 4.5, "Defining Property Sets."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To apply a property set to an entity object or view object:

1. In the Application Navigator, double-click the desired object (entity object or view object).
2. In the overview editor, click the **General** navigation tab and then click the **Edit** icon next to the **Property Set** line.
3. Select the appropriate property set, and click **OK**.

To apply a property set to an attribute:

1. In the Application Navigator, double-click the object (entity object or view object) that contains the attribute.
2. In the overview editor, click the **Attributes** navigation tab, select the attribute you want to edit, and then click the **Details** tab.
3. In the **Property Set** dropdown list, select the appropriate property set.

4.6 Defining Attribute Control Hints for Entity Objects

If you are familiar with previous versions of ADF Business Components, you may have used control hints. Control hints allow you to define label text, tooltip, and format mask hints for entity object attributes. The UI hints you define on your business domain layer are inherited by any entity-based view objects as well. You can also set additional control hints on view objects and application modules in a similar manner.

4.6.1 How to Add Attribute Control Hints

To add attribute control hints to an entity object, use the overview editor.

Before you begin:

It may be helpful to have an understanding of how control hints are used in an entity object. For more information, see Section 4.6, "Defining Attribute Control Hints for Entity Objects."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

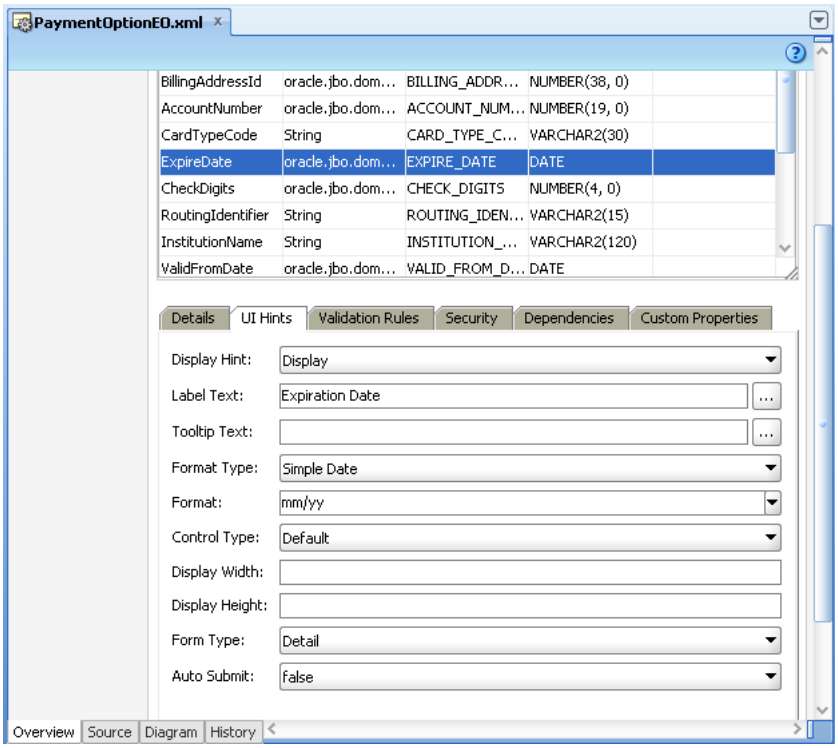
To add attribute control hints to an entity object:

1. In the Application Navigator, double-click the desired entity object.
2. In the overview editor, click the **Attributes** navigation tab, select the attribute you want to edit, and then click the **UI Hints** tab.
3. Specify control hints as necessary.

For example, Figure 4–12 shows control hints defined for the attribute `ExpireDate` of the `PaymentOptionEO` entity object. The defined hints include the following:

- **Format Type** of `Simple Date`
- **Format** mask of `mm/yy`

Figure 4–12 Overview Editor, Attributes Page, UI Hints Tab



Note: Java defines a standard set of format masks for numbers and dates that are different from those used by the Oracle database's SQL and PL/SQL languages. For reference, see the Javadoc for the `java.text.DecimalFormat` and `java.text.SimpleDateFormat` classes.

4.6.2 What Happens When You Add Attribute Control Hints

When you define attribute control hints for an entity object, JDeveloper creates a resource bundle file in which to store them. The hints that you define can be used by generated forms and tables in associated view clients. The type of file and its granularity are determined by Resource Bundle options in the Project Properties dialog. For more information, see Section 4.7, "Working with Resource Bundles."

4.6.3 How to Define Formatters and Masks

When you set the **Format Type** control hint (on the **UI Hints** tab) for an attribute (for example, to **Simple Date**), you can also specify a format mask for the attribute to customize how the UI displays the value. If the mask you want to use is not listed in the **Format** dropdown list, you can simply type it into the field.

Not all formatters require format masks. Specifying a format mask is only needed if that formatter type requires it. For example, the date formatter requires a format mask, but the currency formatter does not. In fact the currency formatter does not support format mask at all.

The mask elements that you can use are defined by the associated Java format class. For information about the mask elements for the Simple Date format type, see the Javadoc for `java.text.SimpleDateFormat`. For information about the mask elements for the Number format type, see the Javadoc for `java.text.DecimalFormat`.

If you have a format mask that you will continue to use on multiple occasions, you can add it to the `formatinfo.xml` file, so that it is available from the **Format** dropdown list on the **UI Hints** tab. The entries in this file define the format masks and formatter classes for a domain class. Example 4–3 shows the format definitions for the `java.util.Date` domain.

> **Note:** You can find the `formatinfo.xml`file in the BC4J subdirectory of the JDeveloper system directory (for example, `C:\Documents and Settings\`*`username`*`\Application Data\JDeveloper\`*`system`*`##\o.BC4J\formatinfo.xml`).

Example 4–3 Format Definitions for java.util.Date in formatinfo.xml

```
<?xml version="1.0"?>
<FORMATTERS>
. . .
  <DOMAIN CLASS="java.util.Date">
     <FORMATTER name="Simple Date" class="oracle.jbo.format.DefaultDateFormatter">
        <FORMAT text="yyyy-MM-dd" />
        <FORMAT text="EEE, MMM d, ''yy"  />
        <FORMAT text="dd-MM-yy" />
        <FORMAT text="dd-MMM-yyyy" />
        <FORMAT text="dd/MMM/yyyy" />
     </FORMATTER>
  </DOMAIN>
. . .
</FORMATTERS>
```

The definition of the format mask belongs to a formatter and a domain class, and includes the text specification of the mask as it appears on the **UI Hints** tab. When you specify the **Format Type** (`FORMATTER name`) for an attribute of a given type (`DOMAIN CLASS`), the masks (`FORMAT text`) appear in the **Format** dropdown list.

To map a formatter to a domain for use with control hints, you can either amend one of the default formatters provided in the `oracle.jbo.format` package, or create a new formatter class by extending the `oracle.jbo.format.Formatter` class. The default formatters provided with JDeveloper aggregate the formatters provided in the `java.text` package.

It is not necessary to create new domain to map a formatter. You can use an existing domain when the business components project contains a domain of the same data type as the formatter.

Before you begin:

It may be helpful to have an understanding of how control hints are used in an entity object. For more information, see Section 4.6, "Defining Attribute Control Hints for Entity Objects."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To define a new format mask:

1. Open the `formatinfo.xml` file in a text editor.
2. Find the domain class and formatter name for which you want to add a format mask.
3. Insert a new `FORMAT` entry within the `FORMATTER` element.

After defining a format mask, you can select the new format mask from the **Format** dropdown list on the **UI Hints** tab.

> **Note:** If you create a new domain for the format mask, the XML definition of the formatter must include a `DOMAIN CLASS` (which can be a new or existing one), the `FORMATTER` (which includes the name and class), and the list of `FORMAT` definitions the formatter class specifies.

4.7 Working with Resource Bundles

When you define translatable strings (such as validator error messages, or attribute control hints for an entity object or view object), by default JDeveloper creates a project-level resource bundle file in which to store them. For example, when you define control hints for an entity object in the `StoreFront` project, JDeveloper creates the message bundle file named `StoreFrontBundle.`*`xxx`* for the package. The hints that you define can be used by generated forms and tables in associated view clients.

The resource bundle option that JDeveloper uses is determined by an option on the Resource Bundle page of the Project Properties dialog. By default JDeveloper sets the option to **Properties Bundle**, which produces a `.properties` file. For more information on this and other resource bundle options, see Section 4.7.1, "How to Set Message Bundle Options."

You can inspect the message bundle file for the entity object by selecting the object in the Application Navigator and looking in the corresponding **Sources** node in the Structure window. The Structure window shows the implementation files for the component you select in the Application Navigator.

Example 4–4 shows a sample message bundle file where the control hint information appears. The first entry in each `String` array is a message key; the second entry is the locale-specific `String` value corresponding to that key.

Example 4–4 Project Message Bundle Stores Locale-Sensitive Control Hints

```
AddressUsageEO_OwnerTypeCode_Error_0=Invalid OwnerTypeCode.
AddressUsageEO_UsageTypeCode_Error_0=Invalid UsageTypeCode.
OwnerTypeCode_CONTROLTYPE=105
PaymentOptionEO_RoutingIdentifier_Error_0=Please enter a valid routing identifier.
PaymentOptionsEO_PaymentTypeCode_Error_0=Invalid PaymentTypeCode.
PaymentTypeCode_CONTROLTYPE=105
PaymentOption_AccountNumber=Please enter a valid Account Number
MinPrice_FMT_FORMATTER=oracle.jbo.format.DefaultCurrencyFormatter
CostPrice_FMT_FORMATTER=oracle.jbo.format.DefaultCurrencyFormatter
UnitPrice_FMT_FORMATTER=oracle.jbo.format.DefaultCurrencyFormatter
OrderEO_GiftMessage=Please supply a message shorter than 200 characters
```

```
OrderEO=Please supply a gift message
DiscountBaseEO_DiscountAmount=Discount must be between 0 and 40%

oracle.fodemo.storefront.entities.PaymentOptionEO.ExpireDate_FMT_FORMAT=mm/yy
#Date range validation for ValidFrom and ValidTo dates
PaymentOptionEO_invalidDateRange_Error_0=Date range is invalid. {0} must be
greater than {1}.
PaymentOptionEO_DateRange_Error_0=Invalid date range.{0} should be greater than
{1}.

oracle.fodemo.storefront.entities.PaymentOptionEO.ValidFromDate_LABEL=Valid From
Date
oracle.fodemo.storefront.entities.PaymentOptionEO.ValidToDate_LABEL=Valid To Date
OrderItemsVO_ImageId_Rule_0=ImageId not found
oracle.fodemo.storefront.store.queries.AddressesVO.Address1_LABEL=Address
oracle.fodemo.storefront.store.queries.AddressesVO.PostalCode_LABEL=Post Code or
ZIP
. . .
```

4.7.1 How to Set Message Bundle Options

The resource bundle option JDeveloper uses to save control hints and other translatable strings is determined by an option on the Resource Bundle page of the Project Properties dialog. By default JDeveloper sets the option to **Properties Bundle** which produces a `.properties` file.

Before you begin:

It may be helpful to have an understanding of how resource bundles are used. For more information, see Section 4.7, "Working with Resource Bundles."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To set resource bundle options for your project

1. In the Application Navigator, right-click the project and choose **Project Properties**.
2. Click **Resource Bundle**.
3. Select whether to use project or custom settings.

   If you select **Use Custom Settings**, the settings apply only to your work with the current project. They are preserved between sessions, but are not recorded with the project and cannot be shared with other users. If you select **Use Project Settings**, your choices are recorded with the project and can be shared with others who use the project.
4. Specify your preference with the following options by selecting or deselecting the option:
   - **Automatically Synchronize Bundle**
   - **Warn About Hard-coded Translatable Strings**
   - **Always Prompt for Description**

   For more information on these options, click **Help** to see the online help.
5. Select your choice of resource bundle granularity.
   - **One Bundle Per Project** (default)

- **One Bundle Per File**
- **Multiple Shared Bundles** (not available for ADF Business Components)

6. Select the type of file to use.
   - **List Resource Bundle**

     The `ListResourceBundle` class manages resources in a name/value array. Each ListResourceBundle class is contained within a Java class file. You can store any locale-specific object in a ListResourceBundle class.
   - **Properties Bundle** (default)

     A text file containing translatable text in name/value pairs. Property files (like the one shown in Example 4–4) can contain values only for String objects. If you need to store other types of objects, you must use a ListResourceBundle instead.
   - **Xliff Resource Bundle**

     The XML Localization Interchange File Format (XLIFF) is an XML-based format for exchanging localization data.
7. Click **OK** to apply your settings and close the dialog.

4.7.2 How to Use Multiple Resource Bundles

When you define translatable strings (for example, for attribute control hints), the Select Text Resource dialog allows you to enter a new string or select one that is already defined in the default resource bundle for the object. You can also use a different resource bundle if necessary. This is helpful when you use a common resource bundle that is shared between projects.

Before you begin:

It may be helpful to have an understanding of how resource bundles are used. For more information, see Section 4.7, "Working with Resource Bundles."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To use strings in a nondefault resource bundle:

1. In the Select Text Resource dialog, select the bundle you want to use from the **Resource Bundle** dropdown list.

   If the desired resource bundle is not included in the **Resource Bundle** dropdown list, click the **Browse** icon to locate and select the resource bundle you want to use.

   The dialog displays the strings that are currently defined in the selected resource bundle.
2. Select an existing string and click **Select**, or enter a new string and click **Save and Select**.

   If you entered a new string it is written to the selected resource bundle.

4.7.3 How to Internationalize the Date Format

Internationalizing the model layer of an application built using ADF Business Components entails producing translated versions of each component message bundle file. For example, the Italian version of the `OrdersImplMsgBundle` message bundle

would be a class named `OrdersImplMsgBundle_it` and a more specific *Swiss* Italian version would have the name `OrdersImplMsgBundle_it_ch`. These classes typically extend the base message bundle class, and contain entries for the message keys that need to be localized, together with their localized translation.

Example 4–5 shows the Italian version of an entity object message bundle. Notice that in the Italian translation, the format masks for `RequestDate` and `AssignedDate` have been changed to `dd/MM/yyyy HH:mm`. This ensures that an Italian user will see a date value like May 3rd, 2006, as `03/05/2006 15:55`, instead of `05/03/2006 15:55`, which the format mask in the default message bundle would produce. Notice the overridden `getContents()` method. It returns an array of messages with the more *specific* translated strings merged together with those that are not overridden from the superclass bundle. At runtime, the appropriate message bundles are used automatically, based on the current user's locale settings.

Example 4–5 Localized Entity Object Component Message Bundle for Italian

```
package devguide.model.entities.common;
import oracle.jbo.common.JboResourceBundle;
public class ServiceRequestImplMsgBundle_it
       extends ServiceRequestImplMsgBundle {
  static final Object[][] sMessageStrings = {
    { "AssignedDate_FMT_FORMAT", "dd/MM/yyyy HH:mm" },
    { "AssignedDate_LABEL", "Assegnato il" },
    { "AssignedTo_LABEL", "Assegnato a" },
    { "CreatedBy_LABEL", "Aperto da" },
    { "ProblemDescription_LABEL", "Problema" },
    { "RequestDate_FMT_FORMAT", "dd/MM/yyyy HH:mm" },
    { "RequestDate_LABEL", "Aperto il" },
    { "RequestDate_TOOLTIP", "La data in cui il ticket è stato aperto" },
    { "Status_LABEL", "Stato" },
    { "SvrId_LABEL", "Ticket" }
  };
  public Object[][] getContents() {
    return super.getMergedArray(sMessageStrings, super.getContents());
  }
}
```

4.8 Defining Business Logic Groups

Business logic groups allow you to encapsulate a set of related control hints, default values, and validation logic. A business logic group is maintained separate from the base entity in its own file, and can be enabled dynamically based on context values of the current row.

This is useful, for example, for an HR application that defines many locale-specific validations (like national identifier or tax law checks) that are maintained by a dedicated team for each locale. The business logic group eases maintenance by storing these validations in separate files, and optimizes performance by loading them only when they are needed.

Each business logic group contains a set of business logic units. Each unit identifies the set of business logic that is loaded for the entity, based on the value of the attribute associated with the business logic group.

For example, you can define a business logic group for an `Employee` entity object, specifying the `EmpRegion` attribute as the discriminator. Then define a business logic unit for each region, one that specifies a range validator for the employee's salary. When the application loads a row from the `Employee` entity, the appropriate validator

for the EmpSalary attribute is loaded (based on the value of the EmpRegion attribute).

In another example, from the StoreFront module of the Fusion Order Demo application, the PersonEO entity object has a business logic group called PersonTypeCodeGroup that uses PersonTypeCode as the discriminator attribute. Because this attribute has three valid values (CUST, STAFF, and SUPP), there are three corresponding business logic units.

In this scenario, each business logic unit contains new or modified business logic that pertains only to that person type:

- The CUST business logic unit contains logic that pertains to customers. For example, it contains a validator that checks for a phone number because all customers must have a phone number.
- The STAFF business logic unit contains logic that pertains to staff members. For example, it contains a validator that constrains the credit limit.
- The SUPP business logic unit contains logic that pertains to suppliers. For example, it contains a validator that makes sure the ContactByAffiliatesFlag attribute is set to N, because suppliers cannot be contacted by affiliates.

4.8.1 How to Create a Business Logic Group

You create the business logic group for an entity object from the overview editor.

Before you begin:

It may be helpful to have an understanding of how business logic groups are used. For more information, see Section 4.8, "Defining Business Logic Groups."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To create a business logic group:

1. In the Application Navigator, double-click the entity for which you want to create a business logic group.
2. In the overview editor, click the **General** navigation tab.
3. On the General page, expand the **Business Logic Groups** section, and click the **Add** icon.
4. In the creation dialog, select the appropriate group discriminator attribute and specify a name for the group.

   Tip: To enhance the readability of your code, you can name the group to reflect the discriminator. For example, if the group discriminator attribute is PersonTypeCode, you can name the business logic group PersonTypeCodeGroup.

5. Click **OK**.

The new business logic group is added to the table in the overview editor. After you have created the group, you can add business logic units to it.

4.8.2 How to Create a Business Logic Unit

You can create a business logic unit from the New Gallery, or directly from the context menu of the entity that contains the business logic group.

Before you begin:

It may be helpful to have an understanding of how business logic groups are used. For more information, see Section 4.8, "Defining Business Logic Groups."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To create a business logic unit:

1. In the Application Navigator, right-click the entity that contains the business logic group and choose **New Entity Business Logic Unit** from the context menu.
2. In the Create Business Logic Unit dialog, specify the name of the base entity and select the appropriate business logic group.
3. Enter a name for the business logic unit.

   The name of each business logic unit must reflect a valid value of the group discriminator attribute with which this business logic unit will be associated. For example, if the group discriminator attribute is `PersonTypeCode`, the name of the business logic unit associated with the `PersonTypeCode` value of `STAFF` must be `STAFF`.
4. Specify the package for the business logic unit.

   Note: The package for the business logic unit does not need to be the same as the package for the base entity or the business logic group. This allows you to develop and deliver business logic units separately from the core application.
5. Click **OK**.

JDeveloper creates the business logic unit and opens it in the overview editor. The name displayed for the business logic unit in the Application Navigator contains the name of the entity object and business logic group in the format *`EntityName_BusLogicGroupName_BusLogicUnitName`*. For example, when you create a business logic unit with the name `CUST` in the `PersonTypeCodeGroup` business logic group of the `PersonEO` entity object, the displayed name of the business logic unit is `PersonEO_PersonTypeCodeGroup_CUST`.

After you have created the unit, you can redefine the business logic for it.

4.8.3 How to Add Logic to a Business Logic Unit

After you have created a business logic unit, you can open it in the overview editor and add business logic (such as adding an entity-level validator) just as you would in the base entity.

Before you begin:

It may be helpful to have an understanding of how business logic groups are used. For more information, see Section 4.8, "Defining Business Logic Groups."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To add an entity validator to a business logic unit:

1. In the Application Navigator, double-click the business logic unit.
2. In the overview editor, click the **Business Rules** navigation tab.
3. On the Business Rules page, select the Entity Validators node and click the **Add** icon.
4. Define your validation rule, and click **OK**.

For example, the `PersonEO` entity object in the StoreFront module of the Fusion Order Demo application has a business logic unit called `PersonEO_PersonTypeCodeGroup_CUST`. This business logic unit has an entity validator that checks for the presence of a phone number to ensure that all persons who are customers have a phone number.

4.8.4 How to Override Attributes in a Business Logic Unit

When you view the Attributes page for the business logic unit (in the overview editor), you can see that the **Extends** column in the attributes table shows that the attributes are "extended" in the business logic unit. Extended attributes are editable only in the base entity, not in the business logic unit. To implement changes in the business logic unit rather than the base entity, you must define attributes as overridden in the business logic unit before you edit them.

Before you begin:

It may be helpful to have an understanding of how business logic groups are used. For more information, see Section 4.8, "Defining Business Logic Groups."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To override attributes in a business logic unit:

1. In the Application Navigator, double-click the business logic unit.
2. In the overview editor, click the **Attributes** navigation tab.
3. On the Attributes page, select the desired attribute and click the **Override** button.

After you make an attribute overridden, you can edit the attribute as you normally would in the tabs below the table. You will notice that in an overridden attribute, you are limited to making modifications to only control hints, validators, and default values.

4.8.5 What Happens When You Create a Business Logic Group

When you create a business logic group, JDeveloper adds a reference to the group in the base entity's XML file. Example 4–6 shows the code added to the base entity's XML file for the business logic group.

Example 4–6 XML Code in the Base Entity for a Business Logic Group

```
<BusLogicGroup
    Name="PersonTypeCodeGroup"
```

```
    DiscrAttrName="PersonTypeCode"/>
```

When you create a business logic unit, JDeveloper generates an XML file similar to that of an entity object. Example 4–7 shows XML code for a business logic unit.

> **Note:** The package for the business logic unit does not need to be the same as the package for the base entity or the business logic group. This allows you to develop and deliver business logic units separately from the core application.

Example 4–7 XML Code for a Business Logic Unit

```
<Entity
  xmlns="http://xmlns.oracle.com/bc4j"
  Name="PersonEO_PersonTypeCodeGroup_CUST"
  Version="11.1.1.54.6"
  Extends="oracle.fodemo.storefront.entities.PersonEO"
  DBObjectType="table"
  DBObjectName="PERSONS"
  BindingStyle="OracleName"
  UseGlueCode="false"
  BusLogicGroupName="PersonTypeCodeGroup"
  BusLogicUnitName="CUST"
  xmlns:validation="http://xmlns.oracle.com/adfm/validation">
  <DesignTime>
    <Attr Name="_codeGenFlag2" Value="Access"/>
    <AttrArray Name="_publishEvents"/>
  </DesignTime>
  <validation:ExpressionValidationBean
    Name="PersonEO_PersonTypeCodeGroup_CUST_Rule_0"
    OperandType="EXPR"
    Inverse="false">
    <validation:MsgIds>
      <validation:Item
        Value="CUST_PHONE_REQUIRED"/>
    </validation:MsgIds>
    <validation:TransientExpression>
      <![CDATA[if (PhoneNumber == null && MobilePhoneNumber == null)
        return false;
        else return true;]]>
    </validation:TransientExpression>
  </validation:ExpressionValidationBean>
  <ResourceBundle>
    <PropertiesBundle
      PropertiesFile="oracle.fodemo.storefront.entities.common.PersonEO_
PersonTypeCodeGroup_CUSTMsgBundle"/>
  </ResourceBundle>
</Entity>
```

4.8.6 What Happens at Runtime: Invoking a Business Logic Group

When a row is loaded in the application at runtime, the entity object decides which business logic units to apply to it.

The base entity maintains a list of business logic groups. Each group references the value of an attribute on the entity, and this value determines which business logic unit to load for that group. This evaluation is performed for each row that is loaded.

If the logic for determining which business logic unit to load is more complex than just a simple attribute value, you can create a transient attribute on the entity object, and use a Groovy expression to determine the value of the transient attribute.

4.9 Configuring Runtime Behavior Declaratively

Entity objects offer numerous declarative features to simplify implementing typical enterprise business applications. Depending on the task, sometimes the declarative facilities *alone* may satisfy your needs. The declarative runtime features that describe the basic persistence features of an entity object are covered in this section, while declarative validation and business rules are covered in Chapter 7, "Defining Validation and Business Rules Declaratively."

> **Note:** It is possible to go beyond the declarative behavior to implement more complex business logic or validation rules for your business domain layer when needed. In Chapter 8, "Implementing Validation and Business Rules Programmatically," you'll see some of the most typical ways that you extend entity objects with custom code.

Also, it is important to note as you develop your application that the business logic you implement, either programmatically or declaratively, should not assume that the attributes of an entity object or view row will be set in a particular order. This will cause problems if the end user enters values for the attributes in an order other than the assumed one.

4.9.1 How to Configure Declarative Runtime Behavior

To configure the declarative runtime behavior of an entity object, use the overview editor.

Before you begin:

It may be helpful to have an understanding of declarative configuration of runtime behavior. For more information, see Section 4.9, "Configuring Runtime Behavior Declaratively."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To configure the declarative runtime behavior of an entity object:

1. In the Application Navigator, double-click an entity object.
2. In the overview editor, click the **General** navigation tab to view the name and package of the entity object, and configure aspects of the object at the entity level, such as its associated schema, alternative keys, custom properties, and security.
   - The **Alternate Keys** section allows you to select entity object attributes mapped to the database that can serve as an alternative primary key. For information on alternative keys, see Section 4.10.15, "How to Define Alternate Key Values."
   - The **Tuning** section allows you to set options to make database operations more efficient when you create, modify, or delete multiple entities of the same

type in a single transaction. For more information, see Section 4.9.3, "How to Use Update Batching."

- The **Custom Properties** section allows you to define custom metadata that you can access at runtime on the entity.
- The **Security** section allows you to define role-based updatability permissions for the entity. For more information, see Chapter 35, "Enabling ADF Security in a Fusion Web Application."
- The **Business Logic Groups** section allows you to add and edit business logic groups. For more information, see Section 4.8, "Defining Business Logic Groups."

3. Click the **Attributes** navigation tab to create or delete attributes that represent the data relevant to an entity object, and configure aspects of the attribute, such as validation rules, custom properties, and security.

   Select an attribute and click the **Edit** icon to access the properties of the attribute. For information on how to set these properties, see Section 4.10, "Setting Attribute Properties."

   Tip: If your entity has a long list of attribute names, there's a quick way to find the one you're looking for. In the Structure window with the **Attributes** node expanded, you can begin to type the letters of the attribute name and JDeveloper performs an incremental search to take you to its name in the tree.

4. Click the **Business Rules** navigation tab to define declarative validators for the entity object and its attributes. For more information, see Chapter 7, "Defining Validation and Business Rules Declaratively."
5. Click the **Java** navigation tab to select the classes you generate for custom Java implementation. You can use the Java classes for such things as defining programmatic business rules, as in Chapter 8, "Implementing Validation and Business Rules Programmatically."
6. Click the **Business Events** navigation tab to define events that your entity object can use to notify others of interesting changes in its state, optionally including some or all of the entity object's attributes in the delivered event. For more information about business events, see Section 4.12, "Creating Business Events."
7. Click the **View Accessors** navigation tab to create and manage view accessors. For more information, see Section 10.4.1, "How to Create a View Accessor for an Entity Object or View Object."

4.9.2 What Happens When You Configure Declarative Runtime Behavior

The declarative settings that describe and control an entity object's runtime behavior are stored in its XML component definition file. When you use the overview editor to modify settings of your entity, JDeveloper updates the component's XML definition file and optional custom Java files.

4.9.3 How to Use Update Batching

You can use update batching to reduce the number of DML statements issued with multiple entity modifications.

By default, the ADF Business Components framework performs a single DML statement (`INSERT`, `UPDATE`, `DELETE`) for each modified entity of a given entity definition type. For example, say you have an Employee entity object type for which multiple instances are modified during typical use of the application. If two instances were created, three existing instances modified, and four existing instances deleted, then at transaction commit time the framework issues nine DML statements (2 `INSERT`s, 3 `UPDATE`s, and 4 `DELETE`s) to save these changes.

If you will frequently be updating more than one entity of a given type in a transaction, consider using the update batching feature for that entity definition type. In the example, update batching (with a threshold of 1) causes the framework to issue just three DML statements: one bulk `INSERT` statement processing two inserts, one bulk `UPDATE` statement processing three updates, and one bulk `DELETE` statement processing four deletes.

Note: If the entity object has any attributes that are set to **Refresh After Insert** or **Refresh After Update**, then the batch update feature is disabled.

Before you begin:

It may be helpful to have an understanding of the declarative configuration of runtime behavior. For more information, see Section 4.9, "Configuring Runtime Behavior Declaratively."

You may also find it useful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To enable update batching for an entity

1. In the Application Navigator, double-click the appropriate entity.
2. In the overview editor, click the **General** navigation tab.
3. On the General page of the overview editor, expand the **Tuning** section, select the **Use Update Batching** checkbox, and specify the appropriate threshold.

This establishes a batch processing threshold beyond which Oracle ADF will process the modifications in a bulk DML operation.

4.10 Setting Attribute Properties

The declarative framework helps you set attribute properties easily. In all cases, you set these properties on the Attributes page of the overview editor.

4.10.1 How to Set Database and Java Data Types for an Entity Object Attribute

The `Persistent` property controls whether the attribute value corresponds to a column in the underlying table, or whether it is just a transient value. If the attribute is persistent, the **Database Column** area lets you change the name of the underlying column that corresponds to the attribute and indicate its column type with precision and scale information (e.g. `VARCHAR2(40)` or `NUMBER(4,2)`). Based on this information, at runtime the entity object enforces the maximum length and precision/scale of the attribute value, and throws an exception if a value does not meet the requirements.

Both the Business Components from Tables wizard and the Create Entity Object wizard infer the Java type of each entity object attribute from the SQL type of the database column type of the column to which it is related.

> **Note:** The project's Type Map setting also plays a role in determining the Java data type. You specify the Type Map setting when you initialize your business components project, before any business components are created. For more information, see Section 3.3.1, "Choosing a Connection, SQL Platform, and Data Type Map."

The **Type** field (on the **Details** tab) allows you to change the Java type of the entity attribute to any type you might need. The **Column Type** field reflects the SQL type of the underlying database column to which the attribute is mapped. The value of the **Column Name** field controls the column to which the attribute is mapped.

Your entity object can handle tables with various column types, as listed in Table 4–1. With the exception of the `java.lang.String` class, the default Java attribute types are all in the `oracle.jbo.domain` and `oracle.ord.im` packages and support efficiently working with Oracle database data of the corresponding type. The dropdown list for the **Type** field includes a number of other common Java types that are also supported.

Table 4–1 Default Entity Object Attribute Type Mappings

Oracle Column Type	Entity Column Type	Entity Java Type
NVARCHAR2(n), VARCHAR2(n),NCHAR VARYING(n),VARCHAR(n)	VARCHAR2	java.lang.String
NUMBER	NUMBER	oracle.jbo.domain.Number
DATE	DATE	oracle.jbo.domain.Date
TIMESTAMP(n), TIMESTAMP(n) WITH TIME ZONE,TIMESTAMP(n) WITH LOCAL TIME ZONE	TIMESTAMP	java.sql.Timestamp
LONG	LONG	java.lang.String
RAW(n)	RAW	oracle.jbo.domain.Raw
LONG RAW	LONG RAW	oracle.jbo.domain.Raw
ROWID	ROWID	oracle.jbo.domain.RowID
NCHAR,CHAR	CHAR	oracle.jbo.domain.Char
CLOB	CLOB	oracle.jbo.domain.ClobDomain
NCLOB	NCLOB	oracle.jbo.domain.NClobDomain
BLOB	BLOB	oracle.jbo.domain.BlobDomain
BFILE	BFILE	oracle.jbo.domain.BFileDomain
ORDSYS.ORDIMAGE	ORDSYS.ORDIMAGE	oracle.ord.im.OrdImageDomain
ORDSYS.ORDVIDEO	ORDSYS.ORDVIDEO	oracle.ord.im.OrdVideoDomain
ORDSYS.ORDAUDIO	ORDSYS.ORDAUDIO	oracle.ord.im.OrdAudioDomain
ORDSYS.ORDDOC	ORDSYS.ORDDOC	oracle.ord.im.OrdDocDomain

Note: In addition to the types mentioned here, you can use any Java object type as an entity object attribute's type, provided it implements the `java.io.Serializable` interface.

4.10.2 How to Indicate Data Type Length, Precision, and Scale

When working with types that support defining a maximum length like `VARCHAR2(n)`, the **Column Type** field (on the **Details** tab) includes the maximum attribute length as part of the value. For example, an attribute based on a `VARCHAR2(10)` column in the database will initially reflect the maximum length of 10 characters by showing `VARCHAR2(10)` as the database column type. If for some reason you want to restrict the maximum length of the `String`-valued attribute to fewer characters than the underlying column will allow, just change the maximum length of the **Column Type** value.

For example, if the `EMAIL` column in the `PERSONS` table is `VARCHAR2(50)`, then by default the `Email` attribute in the `Persons` entity object defaults to the same. But if you know that the actual email addresses are always 8 characters or fewer, you can update the database column type for the `Email` attribute to be `VARCHAR2(8)` to enforce a maximum length of 8 characters at the entity object level.

The same holds for attributes related to database column types that support defining a precision and scale like `NUMBER(p[,s])`. For example, to restrict an attribute based on a `NUMBER(7,2)` column in the database to instead have a precision of 5 and a scale of 1, just update the value of the **Column Type** field to be `NUMBER(5,1)`.

4.10.3 How to Control the Updatability of an Attribute

The `Updatable` property controls when the value of a given attribute can be updated. You can select the following values:

- **Always,** the attribute is always updatable
- **Never**, the attribute is read-only
- **While New**, the attribute can be set during the transaction that creates the entity row for the first time, but after being successfully committed to the database the attribute is read-only

Note: In addition to the static declaration of updatability, you can also add custom code in the `isAttributeUpdateable()` method of the entity to determine the updatability of an attribute at runtime.

4.10.4 How to Make an Attribute Mandatory

Select the **Mandatory** checkbox if the field is required. The mandatory property is enforced during entity-level validation at runtime (and not when the attribute validators are run).

4.10.5 How to Define the Primary Key for the Entity

The `Primary Key` property indicates whether the attribute is part of the key that uniquely identifies the entity. Typically, you use a single attribute for the primary key, but multiattribute primary keys are fully supported.

At runtime, when you access the related `Key` object for any entity row using the `getKey()` method, this `Key` object contains the value of the primary key attribute for the entity object. If your entity object has multiple primary key attributes, the `Key` object contains each of their values. It is important to understand that these values appear in the same *relative* sequential order as the corresponding primary key attributes in the entity object definition.

For example, if the `OrderItemEO` entity object has multiple primary key attributes `OrderId` and `LineItemId`. On the Entity Attribute page of the overview editor, `OrderId` is first, and `LineItemId` is second. An array of values encapsulated by the `Key` object for an entity row of type `OrderItemEO` will have these two attribute values in exactly this order.

It is crucial to be aware of the order in which multiple primary key attributes appear on the Entity Attributes page. If you try to use `findByPrimaryKey()` to find an entity with a multiattribute primary key, and the `Key` object you construct has these multiple primary key attributes in the wrong order, the entity row will not be found as expected.

In addition, to populate the primary key in new rows, you might want to use a trigger to assign the value from the database. For more information, see Section 4.10.10, "How to Get Trigger-Assigned Primary Key Values from a Database Sequence"

4.10.6 How to Define a Static Default Value

The value field on the **Details** tab allows you to specify a static default value for the attribute when the value type is set to **Literal**. For example, you can set the default value of the `ServiceRequest` entity object's `Status` attribute to `Open`, or set the default value of the `User` entity object's `UserRole` attribute to `user`.

Note: When more than one attribute is defaulted for an entity object, the attributes are defaulted in the order in which they appear in the entity object's XML file.

4.10.7 How to Define a Default Value Using a Groovy Expression

You can use a Groovy expression to define a default value for an attribute. This approach is useful if you want to be able to change default values at runtime, but if the default value is always the same, the value is easier to see and maintain using a value field with the **Literal** type (on the **Details** tab). For general information about using Groovy, see Section 3.6, "Overview of Groovy Scripting Language Support."

Before you begin:

It may be helpful to have an understanding of how you set attribute properties. For more information, see Section 4.10, "Setting Attribute Properties."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To define a default value using a Groovy expression:

1. In the Application Navigator, double-click the desired entity object.
2. In the overview editor, click the **Attributes** navigation tab.
3. On the Attributes page, select the attribute you want to edit, and then click the **Details** tab.

4. Select **Expression** for the value type, and click the **Edit** icon (next to the value field).
5. Enter a Groovy expression in the field provided, and click **OK**.

4.10.8 What Happens When You Create a Default Value Using a Groovy expression

When you define a default value using a Groovy expression, a `<TransientExpression>` tag is added to the entity object's XML file within the appropriate attribute. Example 4–8 shows sample XML code for a Groovy expression that returns the current date for a default value.

Example 4–8 Default Date Value

```
<TransientExpression>
    <![CDATA[
        adf.currentDate
    ]]>
</TransientExpression>
```

4.10.9 How to Synchronize with Trigger-Assigned Values

If you know that the underlying column value will be updated by a database trigger during insert or update operations, you can enable the respective **Refresh on Insert** or **Refresh on Update** checkboxes on the **Details** tab to ensure that the framework automatically retrieves the modified value and keeps the entity object and database row in sync. The entity object will use the Oracle SQL `RETURNING INTO` feature, while performing the `INSERT` or `UPDATE` to return the modified column back to your application in a single database roundtrip.

Note: If you create an entity object for a synonym that resolves to a remote table over a DBLINK, use of this feature will give an error at runtime like:

```
JBO-26041: Failed to post data to database during "Update"
## Detail 0 ##
ORA-22816: unsupported feature with RETURNING clause
```

Section 4.18, "Basing an Entity Object on a Join View or Remote DBLink" describes a technique to circumvent this database limitation.

4.10.10 How to Get Trigger-Assigned Primary Key Values from a Database Sequence

One common case for refreshing an attribute after insert occurs when a primary key attribute value is assigned by a `BEFORE INSERT FOR EACH ROW` trigger. Often the trigger assigns the primary key from a database sequence using PL/SQL logic. Example 4–9 shows an example of this.

Example 4–9 PL/SQL Code Assigning a Primary Key from a Database Sequence

```
CREATE OR REPLACE TRIGGER ASSIGN_SVR_ID
BEFORE INSERT ON SERVICE_REQUESTS FOR EACH ROW
BEGIN
 IF :NEW.SVR_ID IS NULL OR :NEW.SVR_ID < 0 THEN
   SELECT SERVICE_REQUESTS_SEQ.NEXTVAL
     INTO :NEW.SVR_ID
     FROM DUAL;
```

```
   END IF;
END;
```

On the **Details** tab (on the Attributes page of the overview editor), you can set the value of the **Type** field to the built-in data type named `DBSequence`, and the primary key will be assigned automatically by the database sequence. Setting this data type automatically selects the **Refresh on Insert** checkbox.

> **Note:** The sequence name shown on the **Sequence** tab is used only at design time when you use the **Create Database Tables** feature described in Section 4.2.6, "How to Create Database Tables from Entity Objects." The sequence indicated here will be created along with the table on which the entity object is based.

When you create a new entity row whose primary key is a `DBSequence`, a unique negative number is assigned as its temporary value. This value acts as the primary key for the duration of the transaction in which it is created. If you are creating a set of interrelated entities in the same transaction, you can assign this temporary value as a foreign key value on other new, related entity rows. At transaction commit time, the entity object issues its `INSERT` operation using the `RETURNING INTO` clause to retrieve the actual database trigger-assigned primary key value. In a composition relationship, any related new entities that previously used the temporary negative value as a foreign key will get that value updated to reflect the actual new primary key of the master.

You will typically also set the **Updatable** property of a DBSequence-valued primary key to **Never**. The entity object assigns the temporary ID, and then refreshes it with the actual ID value after the `INSERT` operation. The end user never needs to update this value.

For information on how to implement this functionality for an association that is not a composition, see Section 4.14.7.3.3, "Associations Based on DBSequence-Valued Primary Keys."

> **Note:** For a metadata-driven alternative to the DBSequence approach, see Section 4.14.5, "Assigning the Primary Key Value Using an Oracle Sequence."

4.10.11 How to Protect Against Losing Simultaneously Updated Data

At runtime, the framework provides automatic "lost update" detection for entity objects to ensure that a user cannot unknowingly modify data that another user has updated and committed in the meantime. Typically, this check is performed by comparing the original values of each persistent entity attribute against the corresponding current column values in the database at the time the underlying row is locked. Before updating a row, the entity object verifies that the row to be updated is still consistent with the current state of the database. If the row and database state are inconsistent, then the entity object raises the `RowInconsistentException`.

You can make the lost update detection more efficient by identifying any attributes of your entity whose values you know will be updated whenever the entity is modified. Typical candidates include a version number column or an updated date column in the row. The change-indicator attribute's value might be assigned by a database trigger you've written and refreshed in the entity object, because you selected the **Refresh on Insert** or **Refresh on Update** option (on the **Details** tab). Alternatively, you can

indicate that the entity object should manage updating the change-indicator attribute's value using the history attribute feature described in Section 4.10.12, "How to Track Created and Modified Dates Using the History Column." To detect whether the row has been modified since the user queried it in the most efficient way, select the **Change Indicator** option to compare only the change-indicator attribute values.

4.10.12 How to Track Created and Modified Dates Using the History Column

If you need to keep track of historical information in your entity object, such as when an entity was created or modified and by whom, or the number of times the entity has been modified, you specify an attribute with the **Track Change History** option selected (on the **Details** tab).

If an attribute's data type is `Number`, `String`, or `Date`, and if it is not part of the primary key, then you can enable this property to have your entity automatically maintain the attribute's value for historical auditing. How the framework handles the attribute depends which type of history attribute you indicate:

- **Created On**: This attribute is populated with the time stamp of when the row was created. The time stamp is obtained from the database.
- **Created By**: The attribute is populated with the name of the user who created the row. The user name is obtained using the `getUserPrincipalName()` method on the `Session` object.
- **Modified On**: This attribute is populated with the time stamp whenever the row is updated/created.
- **Modified By**: This attribute is populated with the name of the user who creates or updates the row.
- **Version Number**: This attribute is populated with a long value that is incremented whenever a row is created or updated.

4.10.13 How to Configure Composition Behavior

An entity object exhibits composition behavior when it creates (or composes) other entities, such as an `OrderEO` entity creating a `OrderItemEO` entity. This additional runtime behavior determines its role as a logical container of other nested entity object parts.

> **Note:** Composition also affects the order in which entities are validated. For more information, see Section 7.2.3, "Understanding the Impact of Composition on Validation Order."

The features that are always enabled for composing entity objects are described in the following sections:

- Section 4.10.13.1, "Orphan-Row Protection for New Composed Entities"
- Section 4.10.13.2, "Ordering of Changes Saved to the Database"
- Section 4.10.13.3, "Cascade Update of Composed Details from Refresh-On-Insert Primary Keys"

The additional features, and the properties that affect their behavior, are described in the following sections:

- Section 4.10.13.4, "Cascade Delete Support"

- Section 4.10.13.5, "Cascade Update of Foreign Key Attributes When Primary Key Changes"
- Section 4.10.13.6, "Locking of Composite Parent Entities"
- Section 4.10.13.7, "Updating of Composing Parent History Attributes"

4.10.13.1 Orphan-Row Protection for New Composed Entities

When a composed entity object is created, it performs an existence check on the value of its foreign key attribute to ensure that it identifies an existing entity as its owning parent entity. At create time, if no foreign key is found or else a value that does not identify an existing entity object is found, the entity object throws an `InvalidOwnerException` instead of allowing an orphaned child row to be created without a well-identified parent entity.

> **Note:** The existence check finds new pending entities in the current transaction, as well as existing ones in the database if necessary.

4.10.13.2 Ordering of Changes Saved to the Database

Composition behavior ensures that the sequence of data manipulation language (DML) operations performed in a transaction involving both composing and composed entity objects is performed in the correct order. For example, an `INSERT` statement for a new composing parent entity object will be performed before the DML operations related to any composed children.

4.10.13.3 Cascade Update of Composed Details from Refresh-On-Insert Primary Keys

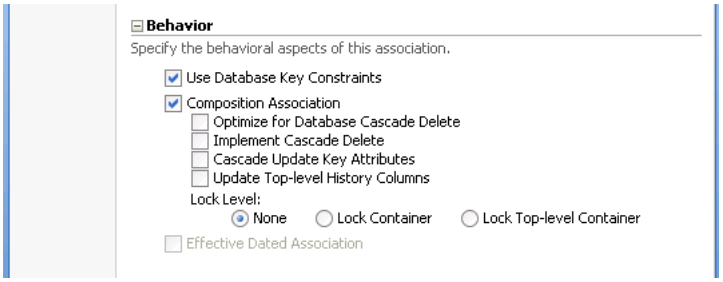
When a new entity row having a primary key configured to refresh on insert is saved, then after its trigger-assigned primary value is retrieved, any composed entities will have their foreign key attribute values updated to reflect the new primary key value.

There are a number of additional composition related features that you can control through settings on the Association Properties page of the Create Association wizard or the overview editor. Figure 4–13 shows the Relationships page for the `OrderItemsOrdersFkAssoc` association between two entity objects: `OrderItemEO` and `OrderEO`.

4.10.13.4 Cascade Delete Support

You can either enable or prevent the deletion of a composing parent while composed children entities exist. When the **Implement Cascade Delete** option (see Figure 4–13) is deselected, the removal of the composing entity object is prevented if it contains any composed children.

Figure 4–13 Composition Settings on Relationship Page of Overview Editor for Associations



When selected, this option allows the composing entity object to be removed unconditionally together with any composed children entities. If the related **Optimize for Database Cascade Delete** option is deselected, then the composed entity objects perform their normal `DELETE` statement at transaction commit time to make the changes permanent. If the option is selected, then the composed entities do not perform the `DELETE` statement on the assumption that the database `ON DELETE CASCADE` constraint will handle the deletion of the corresponding rows.

4.10.13.5 Cascade Update of Foreign Key Attributes When Primary Key Changes

Select the **Cascade Update Key Attributes** option (see Figure 4–13) to enable the automatic update of the foreign key attribute values in composed entities when the primary key value of the composing entity is changed.

4.10.13.6 Locking of Composite Parent Entities

Select the **Lock Top-Level Container** option (see Figure 4–13) to control whether adding, removing, or modifying a composed detail entity row should attempt to lock the composing entity before allowing the changes to be saved.

4.10.13.7 Updating of Composing Parent History Attributes

Select the **Update Top-Level History Columns** option (see Figure 4–13) to control whether adding, removing, or modifying a composed detail entity object should update the **Modified By** and **Modified On** history attributes of the composing parent entity.

4.10.14 How to Set the Discriminator Attribute for Entity Object Inheritance Hierarchies

Sometimes a single database table stores information about several different kinds of logically related objects. For example, a payroll application might work with hourly, salaried, and contract employees all stored in a single `EMPLOYEES` table with an `EMPLOYEE_TYPE` column. In this case, the value of the `EMPLOYEE_TYPE` column contains values like `H`, `S`, or `C` to indicate respectively whether a given row represents an hourly, salaried, or contract employee. And while it is possible that many attributes and behavior are the same for all employees, certain properties and business logic may also depend on the type of employee.

In situations where common information exists across related objects, it may be convenient to represent these different types of entity objects using an inheritance hierarchy. For example, attributes and methods common to all employees can be part of a base `Employee` entity object, while subtype entity objects like `HourlyEmployee`, `SalariedEmployee`, and `ContractEmployee` extend the base `Employee` object and add additional properties and behavior. The **Discriminator** attribute setting is used to indicate which attribute's value distinguishes the type of row. Section 4.19,

"Using Inheritance in Your Business Domain Layer," explains how to set up and use inheritance.

4.10.15 How to Define Alternate Key Values

Database primary keys are often generated from a sequence and may not be data you want to expose to the user for a variety of reasons. For this reason, it's often helpful to have alternate key values that are unique. For example, you might want to enforce that every customer have a unique email address. Because a customer may change their email address, you won't want to use that value as a primary key, but you still want the user to have a unique field they can use for login or other purposes.

Alternate keys are useful for direct row lookups via the `findByKey` class of methods. Alternate keys are frequently used for efficient uniqueness checks in the middle tier. For information on how to find out if a value is unique, see Section 7.4.1, "How to Ensure That Key Values Are Unique."

To define an alternate key, you use the Create Entity Constraint wizard.

Before you begin:

It may be helpful to have an understanding of how you set attribute properties. For more information, see Section 4.10, "Setting Attribute Properties."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To define alternate key values:

1. In the Application Navigator, right-click an entity object and choose **New Entity Constraint**.
2. Follow the steps in the Create Entity Constraint wizard to name your constraint and select the attribute or attributes that participate in the key.
3. On the Properties page, select **Alternate Key** and choose the appropriate **Key Properties** options.

   For more information about the **Key Properties** options, press the F1 key or click **Help**.

4.10.16 What Happens When You Define Alternate Key Values

When you define alternate key values, a hash map is created for fast access to entities that are already in memory.

4.10.17 What You May Need to Know About Alternate Key Values

The Unique key constraint is used only for forward generation of `UNIQUE` constraints in the database, not for alternate key values.

4.11 Adding Transient and Calculated Attributes to an Entity Object

In addition to having attributes that map to columns in an underlying table, your entity objects can include transient attributes that display values calculated (for example, using Java or Groovy) or that are value holders. For example, a transient attribute you create, such as `FullName`, could be calculated based on the concatenated values of `FirstName` and `LastName` attributes.

Once you create the transient attribute, you can perform a calculation in the entity object Java class, or use a Groovy expression in the attribute definition to specify a default value.

If you want to be able to change the value at runtime, you can use a Groovy expression. If the calculated value is not likely to change (for example, if it's a sum of the line items), you can perform the calculation directly in the entity object Java class.

4.11.1 How to Add a Transient Attribute

Use the Attributes page of the overview editor to create a transient attribute.

Before you begin:

It may be helpful to have an understanding of the use of transient and calculated attributes. For more information, see Section 4.11, "Adding Transient and Calculated Attributes to an Entity Object."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To add a transient attribute to an entity object:

1. In the Application Navigator, double-click the entity object.
2. In the overview editor, click the **Attributes** navigation tab, and then click the **New** icon.
3. In the New Entity Attribute dialog, enter a name for the attribute and click **OK**.
4. On the Details tab (in the overview editor), set the Java attribute type and select the **Transient** option.
5. If the value will be calculated, set **Updatable** to **Never**.

4.11.2 What Happens When You Add a Transient Attribute

When you add a transient attribute, JDeveloper updates the XML component definition for the entity object to reflect the new attribute.

The `<Attribute>` tag of a transient attribute has no `TableName` and a `ColumnName` of `$none$`, as shown in Example 4–10.

Example 4–10 XML Code for a Transient Attribute

```
<Attribute
   Name="FullName"
   IsUpdateable="false"
   IsQueriable="false"
   IsPersistent="false"
   ColumnName="$none$"
   Type="java.lang.String"
   ColumnType="$none$"
   SQLType="VARCHAR" >
</Attribute>
```

In contrast, a persistent entity attribute has both a `TableName` and a `ColumnName`, as shown in Example 4–11.

Example 4–11 XML Code for a Persistent Attribute

```
<Attribute
   Name="FirstName"
   IsNotNull="true"
   Precision="30"
   ColumnName="FIRST_NAME"
   Type="java.lang.String"
   ColumnType="VARCHAR2"
   SQLType="VARCHAR"
   TableName="USERS" >
</Attribute>
```

4.11.3 How to Base a Transient Attribute on a Groovy Expression

When creating a transient attribute, you can use a Groovy expression to provide the default value.

Before you begin:

It may be helpful to have an understanding of transient and calculated attributes. For more information, see Section 4.11, "Adding Transient and Calculated Attributes to an Entity Object."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

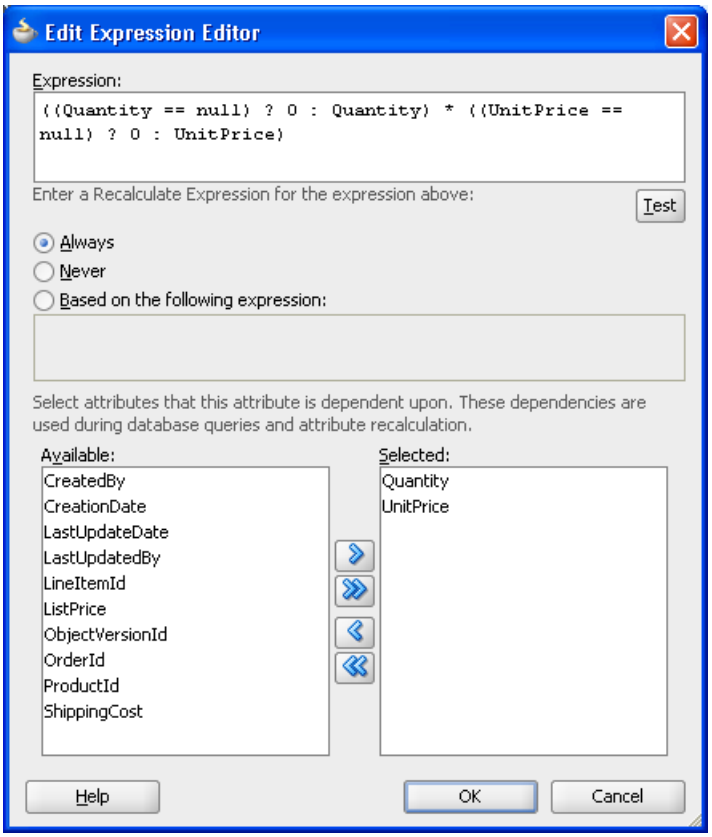
To create a transient attribute based on a Groovy expression:

1. In the Application Navigator, double-click the entity object.
2. In the overview editor, click the **Attributes** navigation tab, and then click the **New** icon.
3. In the New Entity Attribute dialog, enter a name for the attribute and click **OK**.
4. On the Details tab (in the overview editor), set the Java attribute type and select the **Transient** option.
5. If the value will be calculated, set **Updatable** to **Never**.
6. Select **Expression** for the default value type, and click the **Edit** button next to the value field.

   Expressions that you define are evaluated using the Groovy scripting language, as described in Section 3.6, "Overview of Groovy Scripting Language Support." Groovy lets you insert expressions and variables into strings. The expression is saved as part of the entity object definition.
7. In the Edit Expression dialog, enter an expression in the field provided, as shown in Figure 4–14.

   Attributes that you reference can include any attribute that the entity object defines. Do not reference attributes in the expression that are not defined by the entity object.

Figure 4–14 Edit Expression Dialog



8. Select the appropriate recalculate setting.

   If you select **Always** (default), the expression is evaluated each time any attribute in the row changes. If you select **Never**, the expression is evaluated only when the row is created.

9. You can optionally provide a condition for when to recalculate the expression.

   For example, the following expression in the **Based on the following expression** field causes the attribute to be recalculated when either the `Quantity` attribute or the `UnitPrice` attribute are changed:

   ```
   return (adf.object.isAttributeChanged("Quantity") ||
   adf.object.isAttributeChanged("UnitPrice"));
   ```

10. You can also list attributes on which this attribute is dependent.

    In Figure 4–14, the `Quantity` and `UnitPrice` attributes are selected, which causes the attribute to be recalculated when either attribute is changed.

11. Click **OK** to save the expression.

12. Then click **OK** to create the attribute.

> **Note:** If either the value expression or the optional recalculate expression that you define references an attribute from the base entity object, you must define this as a dependency on the **Dependencies** tab (on the Attributes page). On the **Dependencies** tab, locate the attributes in the **Available** list and shuttle each to the **Selected** list.

4.11.4 What Happens When You Base a Transient Attribute on a Groovy Expression

When you base a transient attribute on a Groovy expression, a `<TransientExpression>` tag is added to the entity object's XML file within the appropriate attribute, as shown in Example 4–12.

Example 4–12 Calculating a Transient Attribute Using a Groovy Expression

```
<TransientExpression>
   <![CDATA[
      ((Quantity == null) ? 0 : Quantity) * ((UnitPrice == null) ? 0 : UnitPrice)
   ]]>
</TransientExpression>
```

4.11.5 How to Add Java Code in the Entity Class to Perform Calculation

A transient attribute is a placeholder for a data value. If you change the **Updatable** property of the transient attribute to **While New** or **Always**, then the end user can enter a value for the attribute. If you want the transient attribute to display a calculated value, then you'll typically leave the **Updatable** property set to **Never** and write custom Java code that calculates the value.

After adding a transient attribute to the entity object, to make it a *calculated* attribute you need to:

- Enable a custom entity object class on the Java page of the overview editor, choosing to generate accessor methods
- Write Java code inside the accessor method for the transient attribute to return the calculated value
- Specify each dependent attribute for the transient attribute on the **Dependencies** tab of the Attributes page

For example, after generating the view row class, the Java code to return the transient attribute's calculated value would reside in the getter method for the attribute (such as `FullName`), as shown in Example 4–13.

Example 4–13 Getter Method for a Transient Attribute

```
// Getter method for FullName calculated attribute in UserImpl.java
public String getFullName() {
  // Commented out original line since we'll always calculate the value
  // return (String)getAttributeInternal(FULLNAME);
      return getFirstName()+" "+getLastName();
}
```

To ensure that the transient attribute is reevaluated whenever the attributes to be concatenated (such as `LastName` and `FirstName`) might be changed by the end user, specify the dependent attributes for the transient attribute. On the Dependencies tab of the Attributes page, locate the attributes in the **Available** list and shuttle each to the **Selected** list.

4.12 Creating Business Events

Business events raised from the model layer are useful for launching business processes and triggering external systems synchronization by way of the Oracle Mediator.

Oracle Mediator supports declarative subscriptions which map business events to actions. In other words, you can define and publish a business event (such as a new customer being created) in one component, and then subscribe to that event in another component so that a business process is notified when it occurs. You can then, in the subscribing component, proceed with an action you assign to that event (such as sending a welcome new customer email).

You declaratively define business events at the entity level. You may also specify conditions under which those events should be raised. Business events that meet the specified criteria are raised upon successful commit of the changed data. A business event is raised to the Mediator on a successful create, update, or delete of an entity object.

To implement a business event, you perform the following tasks:

1. Create an event definition, as described in Section 4.12.4, "How to Create a Business Event."
2. Map the event definition to an event point and publish the event definition, as described in Section 4.12.7, "How to Publish a Business Event."

After the business event is published, you can subscribe to the event from another component, as described in Section 4.12.8, "How to Subscribe to Business Events."

4.12.1 Introducing Event Definitions

An event definition describes an event that will be published and raised with an event system Mediator. An event definition is stored in an entity object's XML file with the elements shown in Table 4–2.

Table 4–2 Event Definition Elements for Entity Objects

Element	Description
Event Name	Name of the event, for example, OrderUpdated
Payload	A list of attributes sent to the subscriber. Attributes marked as optional appear on payload only if changed.

4.12.2 Introducing Event Points

An event point is a place from which an event can be raised. On a successful commit, one of the event points shown in Table 4–3 can be raised to the Mediator for each entity in a transaction.

Table 4–3 Example Event Points Raised to the Mediator

DML Type	Event Name	Event Description
CREATE	*Entity*Created	A new *Entity* has been created.
UPDATE	*Entity*Updated	An existing *Entity* has been updated.
DELETE	*Entity*Deleted	An existing *Entity* has been deleted.

Note that no events are raised by default; all events are custom. When you create the event, you can specify the name and DML operation appropriately.

For each event point, you must specify which event definitions should be raised on a particular event point. In other words, you must declaratively map each event definition to an event point.

4.12.3 What You May Need to Know About Event Points

Transactional event delivery, where event delivery is part of the transaction, is not supported by the framework.

Synchronous events, where the publisher waits for further processing until the subscriber has confirmed event reception, is not supported by the framework.

4.12.4 How to Create a Business Event

To create a business event, use the Business Events page of the overview editor.

Before you begin:

It may be helpful to have an understanding of how business events work. For more information, see Section 4.12, "Creating Business Events."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To create a business event:

1. In the Application Navigator, double-click an entity object.
2. In the overview editor, click the **Business Events** navigation tab.
3. On the Business Events page, expand the **Event Definitions** section and click the **New** icon.
4. In the Create Business Event Definition dialog, provide a name that describes this event, such as `EmployeeContactInfoChanged`.
5. In the payload table, click **New** and **Delete** to select the appropriate attributes for this event.

   Alternatively, you can double-click the cell and pick the attributes you want.

   Note: Only attributes of supported types are displayed in the Entity Attribute column. While `ClobDomain` attributes are supported, very large clob data can impact performance.

6. In the **Value Sent** field, choose whether the value should **Always** be sent, or **Only if changed**.

   The **Only if changed** option provides the best performance because the attribute will be considered optional for the payload. If you leave the default **Always**, the payload will require the attribute whether or not the value has changed. For more details about payload efficiency, see Section 4.12.6, "What You May Need to Know About Payload."
7. Use the arrow buttons to rearrange the order of attributes.

   The order that the attributes appear in defines their order in the generated XSD. Since you'll be using the XSD to build your Fabric mediator and BPEL process, you might want the most frequently accessed attributes at the top.
8. Click **OK**.

Repeat the procedure for each business event that you want to define. To publish an event, see Section 4.12.7, "How to Publish a Business Event."

4.12.5 What Happens When You Create a Business Event

When you create a business event, the entity object's XML file is updated with the event definition. Example 4–14 shows an example of the XML code for a business event. JDeveloper also generates an associated XSD file for the event schema that allows specification of required attributes and optional attributes. Required attributes correspond to **Value Sent - Always** in the Create Business Event Definition dialog, whereas optional attributes are those for which you changed **Value Sent** to **Only if changed**.

Example 4–14 XML Code for a Business Event

```
<EventDef
  Name="CustBusEvent1">
  <Payload>
    <PayloadItem
      AttrName="Order.OrderId"/>
    <PayloadItem
      AttrName="LineItemId"/>
    <PayloadItem
      AttrName="ProductBase.ProductId"
      SendOnlyIfChanged="true"/>
  </Payload>
</EventDef>
```

Example 4–15 shows an example of the XSD event schema for a business event.

Example 4–15 XSD Event Schema for a Business Event

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<xs:schema
targetNamespace="/oracle/fodemo/storefront/entities/events/schema/OrderItemEO"
 xmlns="/oracle/fodemo/storefront/entities/events/schema/OrderItemEO"
 elementFormDefault="qualified" attributeFormDefault="unqualified"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="CustBusEvent1Info">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="Order.OrderId" type="DecimalValuePair" minOccurs="1"/>
          <xs:element name="LineItemId" type="DecimalValuePair" minOccurs="1"/>
           <xs:element name="ProductBase.ProductId" type="DecimalValuePair" minOccurs="0"/>
        </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:complexType name="ValuePair" abstract="true"/>
   <xs:complexType name="DecimalValuePair">
      <xs:complexContent>
         <xs:extension base="ValuePair">
            <xs:sequence>
               <xs:element name="newValue" minOccurs="0">
                  <xs:complexType>
                     <xs:complexContent>
                        <xs:extension base="xs:anyType">
                           <xs:attribute name="value" type="xs:decimal"/>
                        </xs:extension>
                     </xs:complexContent>
                  </xs:complexType>
               </xs:element>
               <xs:element name="oldValue" minOccurs="0">
                  <xs:complexType>
```

```
                    <xs:complexContent>
                       <xs:extension base="xs:anyType">
                          <xs:attribute name="value" type="xs:decimal"/>
                       </xs:extension>
                    </xs:complexContent>
                 </xs:complexType>
              </xs:element>
           </xs:sequence>
        </xs:extension>
     </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

Example 4–16 shows an example of the EDL event definition for the entity object.

Example 4–16 EDL Event Definition for the Entity Object

```
<definitions
      targetNamespace="/oracle/fodemo/storefront/entities/events/edl/OrderItemEO"
      xmlns:ns0="/oracle/fodemo/storefront/entities/events/schema/OrderItemEO"
      xmlns="http://schemas.oracle.com/events/edl">
   <schema-import
        namespace="/oracle/fodemo/storefront/entities/events/schema/OrderItemEO"
        location="OrderItemEO.xsd"/>
   <event-definition name="CustBusEvent1">
      <content element="ns0:CustBusEvent1Info"/>
   </event-definition>
</definitions>
```

4.12.6 What You May Need to Know About Payload

The attributes of the associated entity object constitute the payload of a business event. The payload attributes for a business event are defined by the creator of the event. It isn't automatically optimized. When the event is defined, an attribute can be marked as sent **Always** or **Only if changed**. For events fired during creation, only new values are sent. For events fired during an update or delete, the new and old values are sent and only the attributes that should be based on the **Value Sent** setting. For best performance, you should include only the primary key attribute for delete events.

To support composition scenarios (such as a purchase order with line items), a child entity can raise events defined on the parent entity, and events defined on the child entity can include attributes from the parent entity. When a child entity raises an event on a parent entity, only a single event is raised for a particular top-level entity per transaction, regardless of how many times the child entity raises it.

In the case of entity subtypes (for example, a `Staff` entity object is a subtype of the `Persons` entity), ADF Business Components does not support the overriding of business events. Because the subscriber to a business event listens to the event using the event name, overriding of events could cause the event subscriber to receive payload data unintended for that subscriber. Therefore, this capability is not supported.

When defining business events, remember that while `ClobDomain` attributes are supported, very large clob data can have performance implications.

4.12.7 How to Publish a Business Event

To publish a business event, use the Business Events page of the entity objects overview editor.

Before you begin:

It may be helpful to have an understanding of how business events are used in the application. For more information, see Section 4.12, "Creating Business Events."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

You need to have already created the event definition, as described in Section 4.12.4, "How to Create a Business Event," before you can publish it.

To publish a business event:

1. In the Application Navigator, double-click an entity object.
2. In the overview editor, click the **Business Events** navigation tab.
3. On the Business Events page, expand the **Event Publication** section and click the **Edit event publications** icon.
4. In the Edit Event Publications dialog, click **New** to create a new event.
5. Double-click the new cell in **Event** column, and select the appropriate event.
6. Double-click the corresponding cell in **Event Point** column, and select the appropriate event point action.
7. You can optionally define conditions for raising the event using the **Raise Conditions** table.
8. Click **OK**.

4.12.8 How to Subscribe to Business Events

After you have created a business event, you can subscribe and respond to the event.

Before you begin:

It may be helpful to have an understanding of business events. For more information, see Section 4.12, "Creating Business Events."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

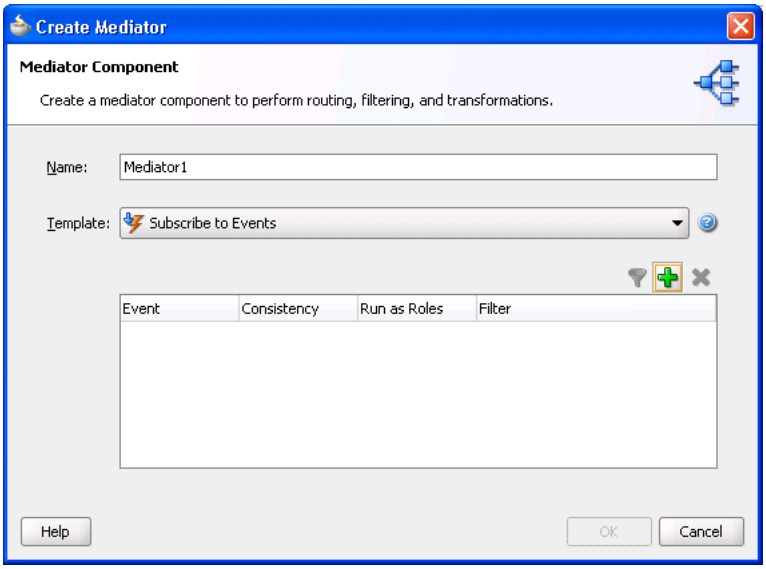
You will also need to complete the following tasks:

- Publish the business event, as described in Section 4.12.7, "How to Publish a Business Event."
- Open the SCA project that will subscribe to the business event.

To subscribe to a business event:

1. Using the file system, copy the XSD and event definition files for the business event into your SCA project's source path.
2. In the Application Navigator, right-click the project, and choose **New**.
3. In the New Gallery, expand **SOA Tier**, select **Service Components** and then **Mediator**, and click **OK**.
4. In the Create Mediator dialog, select the **Subscribe to Events** template, as shown in Figure 4–15.

Figure 4–15 Create Mediator Dialog, Subscribe to Events



5. Click the **Add** icon to add an event.
6. In the Event Chooser dialog, click the **Browse** icon to navigate to and select the event's definition file, and then click **OK**.
7. In the Create Mediator dialog, you can optionally change the **Consistency** option and specify a **Filter** for the event.
8. Click **OK** to generate the mediator.

   The resulting mediator (`.mplan` file) is displayed in the overview editor.
9. You can now click the **Add** icon in the **Routing Rules** section to add a rule for how to respond to the event.

4.13 Generating Custom Java Classes for an Entity Object

As described in this chapter, all of the database interaction and a large amount of declarative runtime functionality of an entity object can be achieved without using custom Java code. When you need to go beyond the declarative features to implement custom business logic for your entities, you'll need to enable custom Java generation for the entities that require custom code. Appendix D, "Most Commonly Used ADF Business Components Methods," provides a quick reference to the most common code that you will typically write, use, and override in your custom entity object and entity definition classes.

4.13.1 How to Generate Custom Classes

To enable the generation of custom Java classes for an entity object, use the Java page of the overview editor.

Before you begin:

It may be helpful to have an understanding of custom Java classes. For more information, see Section 4.13, "Generating Custom Java Classes for an Entity Object."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To generate a custom Java class for an entity object:

1. In the Application Navigator, double-click the entity.
2. In the overview editor, click the **Java** navigation tab, and then click the **Edit Java options** icon.
3. In the Select Java Options dialog, select the types of Java classes you want to generate.
   - Entity Object Class — the most frequently customized, it represents each row in the underlying database table.
   - Entity Collection Class — rarely customized.
   - Entity Definition Class — less frequently customized, it represents the related class that manages entity rows and defines their structure.
4. Click **OK**.

4.13.2 What Happens When You Generate Custom Classes

When you select one or more custom Java classes to generate, JDeveloper creates the Java file(s) you've indicated. For example, assuming an entity object named `fodemo.storefront.entities.OrderEO`, the default names for its custom Java files will be `OrderEOImpl.java` for the entity object class and `OrderEODefImpl.java` for the entity definition class. Both files are created in the same `./fodemo/storefront/entities` directory as the component's XML component definition file.

The Java generation options for the entity object continue to be reflected on subsequent visits to the Java page of the overview editor. Just as with the XML definition file, JDeveloper keeps the generated code in your custom Java classes up to date with any changes you make in the editor. If later you decide you didn't require a custom Java file for any reason, disabling the relevant options on the Java page causes the custom Java files to be removed.

4.13.3 What Happens When You Generate Entity Attribute Accessors

When you enable the generation of a custom entity object class, if you also enable the **Accessors** option, then JDeveloper generates getter and setter methods for each attribute in the entity object. For example, an `OrderEO` entity object that has the corresponding custom `OrderEOImpl.java` class might have methods (like those shown in Example 4–17) generated in it.

Example 4–17 Getter and Setter Methods from OrderEOImpl.java

```
public DBSequence getOrderId() { ... }
public void setOrderId(DBSequence value) { ... }

public Date getOrderDate() { ... }
public void setOrderDate(Date value) { ... }

public String getOrderStatusCode() { ... }
public void setOrderStatusCode(String value) { ... }

public Number getCustomerId() { ... }
```

```
public void setCustomerId(Number value) { ... }

public String getShipToName() { ... }
public void setShipToName(String value) { ... }
```

These methods allow you to work with the row data with compile-time checking of the correct data type usage. That is, instead of writing a line like this to get the value of the `CustomerId` attribute:

```
Number customerId = (Number)order.getAttribute("CustomerId");
```

you can write the code like:

```
Number customerId = order.getCustomerId();
```

You can see that with the latter approach, the Java compiler would catch a typographical error had you accidentally typed `CustomerCode` instead of `CustomerId`:

```
// spelling name wrong gives compile error
Number customerId = order.getCustomerCode();
```

Without the generated entity object accessor methods, an incorrect line of code like the following cannot be caught by the compiler:

```
// Both attribute name and type cast are wrong, but compiler cannot catch it
String customerId = (String)order.getAttribute("CustomerCode");
```

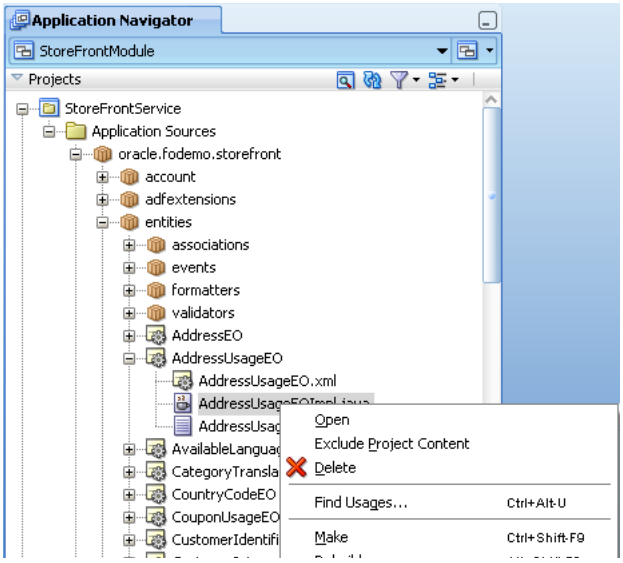
It contains both an incorrectly spelled attribute name, as well as an incorrectly typed cast of the `getAttribute()` return value. When you use the generic APIs on the `Row` interface, which the base `EntityImpl` class implements, errors of this kind raise exceptions at runtime instead of being caught at compile time.

4.13.4 How to Navigate to Custom Java Files

As shown in Figure 4–16, when you've enabled generation of custom Java classes, they also appear as child nodes under the **Application Sources** node for the entity object. As with all ADF components, when you select an entity object in the Application Navigator, the Structure window provides a structural view of the entity. When you need to see or work with the source code for a custom Java file, there are two ways to open the file in the source editor:

- You can right-click the Java file, and choose **Open**, as shown in Figure 4–16.
- You can right-click an item in a node in the Structure window, and choose **Go To Source**.

Figure 4–16 Seeing and Navigating to Custom Java Classes for an Entity Object



4.13.5 What You May Need to Know About Custom Java Classes

The custom Java classes generated by JDeveloper extend the base classes for your entity object, and allow you the flexibility to implement custom code while maintaining the integrity of the generated code. The following sections provide additional information about custom Java classes.

4.13.5.1 Framework Base Classes for an Entity Object

When you use an XML-only entity object, at runtime its functionality is provided by the default ADF Business Components implementation classes. Each custom Java class that is generated extends the appropriate ADF Business Components base class so that your code inherits the default behavior and you can easily add to or customize it. An entity object class will extend `EntityImpl`, while the entity definition class will extend `EntityDefImpl` (both in the `oracle.jbo.server` package).

4.13.5.2 Safely Adding Code to the Custom Component File

Some developers are hesitant to add their own code to generated Java source files. Each custom Java source code file that JDeveloper creates and maintains for you includes the following comment at the top of the file to clarify that it is safe for you to add your own custom code to this file.

```
// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---    Custom code may be added to this class.
// ---    Warning: Do not modify method signatures of generated methods.
// ---------------------------------------------------------------------
```

JDeveloper does not blindly regenerate the file when you click **OK** or **Apply** in an edit dialog. Instead, it performs a smart update to the methods that it needs to maintain, leaving your own custom code intact.

4.13.5.3 Configuring Default Java Generation Preferences

You can generate custom Java classes for your view objects when you need to customize their runtime behavior or when you simply prefer to have strongly typed access to bind variables or view row attributes.

To configure the default settings for ADF Business Components custom Java generation, you can choose **Preferences** from the **Tools** menu and open the Business Components page to set your preferences to be used for business components created in the future. Developers getting started with ADF Business Components should set their preference to generate no custom Java classes by default. As you run into a specific need for custom Java code, you can enable just the bit of custom Java you need for that one component. Over time, you'll discover which set of defaults works best for you.

4.13.5.4 Attribute Indexes and InvokeAccessor Generated Code

The entity object is designed to function based on XML only or as an XML component definition combined with a custom Java class. To support this design choice, attribute values are not stored in private member fields of an entity's class (a file that is not present in the XML-only situation). Instead, in addition to a name, attributes are also assigned a numerical index in the entity's XML component definition based on the zero-based, sequential order of the `<Attribute>` and association-related `<AccessorAttribute>` tags in that file. At runtime, attribute values in an entity row are stored in a sparse array structure managed by the base `EntityImpl` class, indexed by the attribute's numerical position in the entity's attribute list.

For the most part, this private implementation detail is unimportant, since as a developer using entity objects, you are shielded from having to understand this. However, when you enable a custom Java class for your entity object, this implementation detail relates to some of the generated code that JDeveloper maintains in your entity object class. It is sensible to understand what that code is used for. For example, in the custom Java class for a `OrderEO` entity object, each attribute or accessor attribute has a corresponding generated integer enum. JDeveloper ensures that the values of these enums correctly reflect the ordering of the attributes in the XML component definition.

You'll also notice that the automatically maintained, strongly typed getter and setter methods in the entity object class use these attribute enums, as shown in Example 4–18.

Example 4–18 Getter and Setter Methods Using Attribute Constants in the Custom Entity Java Class

```
// In oracle.fodemo.storefront.entities.OrderEOImpl class
public Date getOrderDate() {
    return (Date)getAttributeInternal(ORDERDATE); // <-- Attribute enum
}
public void setOrderDate(Date value) {
    setAttributeInternal(ORDERDATE, value); // <-- Attribute enum
}
```

Another aspect of the maintained code related to entity attribute enums is the `getAttrInvokeAccessor()` and `setAttrInvokeAccessor()` methods. These methods optimize the performance of attribute access by numerical index, which is how generic code in the `EntityImpl` base class typically accesses attribute values when performing generic processing. An example of the `getAttrInvokeAccessor()` method is shown in Example 4–19. The companion `setAttrInvokeAccessor()` method looks similar.

Example 4–19 getAttrInvokeAccessor() Method in the Custom Entity Java Class

```
// In oracle.fodemo.storefront.entities.OrderEOImpl class
/** getAttrInvokeAccessor: generated method. Do not modify. */
protected Object getAttrInvokeAccessor(int index, AttributeDefImpl attrDef)
        throws Exception {
 if ((index >= AttributesEnum.firstIndex()) && (index < AttributesEnum.count())) {
    return AttributesEnum.staticValues()[index - AttributesEnum.firstIndex()].get(this);
 }
  return super.getAttrInvokeAccessor(index, attrDef);
}
```

The rules of thumb to remember about this generated attribute index-related code are the following.

The Do's

- Add custom code if needed inside the strongly typed attribute getter and setter methods.
- Use the overview editor to change the order or type of entity object attributes.

  JDeveloper changes the Java signature of getter and setter methods, as well as the related XML component definition for you.

The Don'ts

- Don't modify the `getAttrInvokeAccessor()` and `setAttrInvokeAccessor()` methods.
- Don't change the values of the attribute index numbers manually.

> **Note:** If you need to manually edit the generated attribute enums because of source control merge conflicts or other reasons, you must ensure that the zero-based ordering reflects the sequential ordering of the `<Attribute>` and `<AccessorAttribute>` tags in the corresponding entity object XML component definition.

4.13.6 Programmatic Example for Comparison Using Custom Entity Classes

To better evaluate the difference between using custom generated entity classes and working with the generic `EntityImpl` class, Example 4–20 shows a version of methods in a custom entity class (`StoreFrontServiceImpl.java`) from a custom application module class (`StoreFrontService2Impl.java`). Some important differences to notice are:

- Attribute access is performed using strongly typed attribute accessors.
- Association accessor attributes return the strongly typed entity class on the other side of the association.
- Using the `getDefinitionObject()` method in your custom entity class allows you to avoid working with fully qualified entity definition names as strings.
- The `createPrimaryKey()` method in your custom entity class simplifies creating the `Key` object for an entity.

Example 4–20 Programmatic Entity Examples Using Strongly Typed Custom Entity Object Classes

```
package devguide.examples.appmodules;

import oracle.fodemo.storefront.entities.OrderEOImpl;
```

```
import oracle.fodemo.storefront.entities.PersonEOImpl;
import oracle.fodemo.storefront.entities.ProductBaseEOImpl;

import oracle.jbo.ApplicationModule;
import oracle.jbo.JboException;
import oracle.jbo.Key;
import oracle.jbo.client.Configuration;
import oracle.jbo.domain.DBSequence;
import oracle.jbo.domain.Number;
import oracle.jbo.server.ApplicationModuleImpl;
import oracle.jbo.server.EntityDefImpl;

// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---    Custom code may be added to this class.
// ---    Warning: Do not modify method signatures of generated methods.
// ---------------------------------------------------------------------
/**
 * This custom application module class illustrates the same
 * example methods as StoreFrontServiceImpl.java, except that here
 * we're using the strongly typed custom Entity Java classes
 * OrderEOImpl, PersonsEOImpl, and ProductsBaseEOImpl instead of working
 * with all the entity objects using the base EntityImpl class.
 */

public class StoreFrontService2Impl extends ApplicationModuleImpl {
    /**This is the default constructor (do not remove).
     */
     public StoreFrontService2Impl() {
     }
     /*
      * Helper method to return an Order by Id
      */
     private OrderEOImpl retrieveOrderById(long orderId) {
       EntityDefImpl orderDef = OrderEOImpl.getDefinitionObject();
       Key orderKey = OrderEOImpl.createPrimaryKey(new DBSequence(orderId));
       return (OrderEOImpl)orderDef.findByPrimaryKey(getDBTransaction(),orderKey);
     }

     /*
      * Find an Order by Id
      */
     public String findOrderTotal(long orderId) {
       OrderEOImpl order = retrieveOrderById(orderId);
       if (order != null) {
         return order.getOrderTotal().toString();
       }
       return null;
     }

     /*
      * Create a new Product and Return its new id
      */
     public long createProduct(String name, String status, String shipCode) {
       EntityDefImpl productDef = ProductBaseEOImpl.getDefinitionObject();
       ProductBaseEOImpl newProduct =
(ProductBaseEOImpl)productDef.createInstance2(getDBTransaction(),null);
         newProduct.setProductName(name);
         newProduct.setProductStatus(status);
```

```
    newProduct.setShippingClassCode(shipCode);
    newProduct.setSupplierId(new Number(100));
    newProduct.setListPrice(new Number(499));
    newProduct.setMinPrice(new Number(479));
    newProduct.setCreatedBy("Test Client");
    newProduct.setLastUpdatedBy("Test Client");
    newProduct.setCategoryId(new Number(5));
  try {
    getDBTransaction().commit();
  }
  catch (JboException ex) {
    getDBTransaction().rollback();
    throw ex;
  }
  DBSequence newIdAssigned = newProduct.getProductId();
  return newIdAssigned.getSequenceNumber().longValue();
}
/*
 * Update the status of an existing order
 */
public void updateRequestStatus(long orderId, String newStatus) {
  OrderEOImpl order = retrieveOrderById(orderId);
  if (order != null) {
    order.setOrderStatusCode(newStatus);
    try {
      getDBTransaction().commit();
    }
    catch (JboException ex) {
      getDBTransaction().rollback();
      throw ex;
    }
  }
}

/*
 * Access an associated Customer entity from the Order entity
 */
public String findOrderCustomer(long orderId) {
  OrderEOImpl svcReq = retrieveOrderById(orderId);
  if (svcReq != null) {
    PersonEOImpl cust  = (PersonEOImpl)svcReq.getPerson();
    if (cust != null) {
      return cust.getFirstName() + " " + cust.getLastName();
    }
    else {
      return "Unassigned";
    }
  }
  else {
    return null;
  }
}

 /*
  * Testing method
  */
  public static void main(String[] args) {
    String        amDef = "devguide.model.StoreFrontService";
    String        config = "StoreFrontServiceLocal";
    ApplicationModule am =
```

```
Configuration.createRootApplicationModule(amDef,config);
         /*
          * NOTE: This cast to use the StoreFrontServiceImpl class is OK since
          * this code is inside a business tier *Impl.java file and not in a
          * client class that is accessing the business tier from "outside".
          */
         StoreFrontServiceImpl service = (StoreFrontServiceImpl)am;
         String total = service.findOrderTotal(1011);
         System.out.println("Status of Order # 1011 = " + total);
         String customerName = service.findOrderCustomer(1011);
         System.out.println("Customer for Order # 1011 = " + customerName);
         try {
             service.updateOrderStatus(1011,"CANCEL");
         }
         catch (JboException ex) {
           System.out.println("ERROR: "+ex.getMessage());
         }
         long id = 0;
         try {
             id = service.createProduct(null, "NEW", "CLASS1");
         }
         catch (JboException ex) {
           System.out.println("ERROR: "+ex.getMessage());
         }
         id = service.createProduct("Canon PowerShot G9", "NEW", "CLASS1");
         System.out.println("New product created successfully with id = "+id);
         Configuration.releaseRootApplicationModule(am,true);
       }
     }
```

4.14 Working Programmatically with Entity Objects and Associations

You may not always need or want UI-based or programmatic clients to work directly with entity objects. Sometimes, you may just want to use an external client program to access an application module and work directly with the view objects in its data model. Chapter 5, "Defining SQL Queries Using View Objects" describes how to easily combine the flexible SQL-querying of view objects with the business logic enforcement and automatic database interaction of entity objects to build powerful applications. The combination enables a fully updatable application module data model, designed to meet the needs of the current end-user tasks at hand, that shares the centralized business logic in your reusable domain business object layer.

However, it is important first to understand how view objects and entity objects can be used on their own before learning to harness their combined power. By learning about these objects in greater detail, you will have a better understanding of when you should use them alone and when to combine them in your own applications.

Since clients don't work *directly* with entity objects, any code you write that works programmatically with entity objects will typically be custom code in a custom application module class or in the custom class of another entity object.

4.14.1 How to Find an Entity Object by Primary Key

To access an entity row, you use a related object called the *entity definition*. At runtime, each entity object has a corresponding entity definition object that describes the structure of the entity and manages the instances of the entity object it describes. After creating an application module and enabling a custom Java class for it, imagine you

wanted to write a method to return a specific order. It might look like the `retrieveOrderById()` method shown in Example 4–21.

Before you begin:

It may be helpful to have an understanding of when to use a programmatic approach for working with entity objects and associations. For more information, see Section 4.14, "Working Programmatically with Entity Objects and Associations."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To find an entity object by primary key:

1. Find the entity definition.

   You obtain the entity definition for an entity object (for example, `OrderEO`) by passing its fully qualified name to the static `getDefinitionObject()` method imported from the `EntityDefImpl` class. The `EntityDefImpl` class in the `oracle.jbo.server` package implements the entity definition for each entity object.

2. Construct a key.

   You build a `Key` object containing the primary key attribute that you want to look up. For example, for the `OrderEO` entity object you create a key containing the single `orderId` value passed into the method as an argument.

3. Find the entity object using the key.

   You use the entity definition's `findByPrimaryKey()` method to find the entity object by key, passing in the current transaction object, which you can obtain from the application module using its `getDBTransaction()` method. The concrete class that represents an entity object row is the `oracle.jbo.server.EntityImpl` class.

4. Return the object or some of its data to the caller.

Example 4–21 show example code for a `retrieveOrderById()` method developed using this basic procedure.

Example 4–21 Retrieving an OrderEO Entity Object by Key

```
/* Helper method to return an Order by Id  */
private OrderEOImpl retrieveOrderById(long orderId) {
  EntityDefImpl orderDef = OrderEOImpl.getDefinitionObject();
  Key orderKey = OrderEOImpl.createPrimaryKey(new DBSequence(orderId));
  return (OrderEOImpl)orderDef.findByPrimaryKey(getDBTransaction(),orderKey);
}
```

> **Note:** The `oracle.jbo.Key` object constructor can also take an Object array to support creating multiattribute keys, in addition to the more typical single-attribute value keys.

4.14.2 How to Access an Associated Entity Using the Accessor Attribute

You can create a method to access an associated entity based on an accessor attribute that requires no SQL code. For example, the method `findOrderCustomer()` might find an order, then access the associated `PersonEO` entity object representing the customer assigned to the order. For an explanation of how associations enable easy

access from one entity object to another, see Section 4.3, "Creating and Configuring Associations."

To prevent a conflict with an existing method in the application module that finds the same associated entity using the same accessor attribute, you can refactor this functionality into a helper method that you can then reuse anywhere in the application module it is required. For example, the `retrieveOrderById()` method (shown in Example 4–21) refactors the functionality that finds an order.

Before you begin:

It may be helpful to have an understanding of when to use a programmatic approach for working with entity objects and associations. For more information, see Section 4.14, "Working Programmatically with Entity Objects and Associations."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To access an associated entity object using the accessor attribute:

1. Find the associated entity by the accessor attribute.

   For example, the `findOrderCustomer()` method uses the `retrieveOrderById()` helper method to retrieve the `OrderEO` entity object by ID.

2. Access the associated entity using the accessor attribute.

   Using the attribute getter method, you can pass in the name of an association accessor and get back the entity object on the other side of the relationship. (Note that Section 4.3.3, "How to Change Entity Association Accessor Names," explains that renaming the association accessor allows it to have a more intuitive name.)

3. Return some of its data to the caller.

   For example, the `findOrderCustomer()` method uses the getter methods on the returned `PersonEO` entity to return the assigned customer's name by concatenating their first and last names.

Notice that you did not need to write any SQL to access the related `PersonEO` entity. The relationship information captured in the ADF association between the `OrderEO` and `PersonEO` entity objects is enough to allow the common task of data navigation to be automated.

Example 4–22 shows the code for `findOrderCustomer()` that uses the helper method.

Example 4–22 Accessing an Associated Entity Using the Accessor Attribute

```
/*  Access an associated Customer entity from the Order entity   */
public String findOrderCustomer(long orderId) {
      //1. Find the OrderEO object
  OrderEOImpl order = retrieveOrderById(orderId);
  if (order != null) {
      //2. Access the PersonEO object using the association accessor attribute
    PersonEOImpl cust  = (PersonEOImpl)order.getPerson();
    if (cust != null) {
      //3. Return attribute values from the associated entity object
      return cust.getFirstName() + " " + cust.getLastName();
    }
    else {
      return "Unassigned";
```

```
    }
  }
  else {
    return null;
  }
}
```

4.14.3 How to Update or Remove an Existing Entity Row

Once you've got an entity row in hand, it's simple to update it or remove it. You could add a method like the `updateOrderStatus()` shown in Example 4–23 to handle the job.

Before you begin:

It may be helpful to have an understanding of when to use a programmatic approach for working with entity objects and associations. For more information, see Section 4.14, "Working Programmatically with Entity Objects and Associations."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To update an entity row:

1. Find the Order by ID.

   Using the `retrieveOrderById()` helper method, the `updateOrderStatus()` method retrieves the `OrderEO` entity object by Id.

2. Set one or more attributes to new values.

   Using the `EntityImpl` class' `setAttribute()` method, the `updateOrderStatus()` method updates the value of the `Status` attribute to the new value passed in.

3. Commit the transaction.

   Using the application module's `getDBTransaction()` method, the `updateOrderStatus()` method accesses the current transaction object and calls its `commit()` method to commit the transaction.

Example 4–23 Updating an Existing Entity Row

```
/* Update the status of an existing order */
public void updateOrderStatus(long orderId, String newStatus) {
      //1. Find the order
  OrderEOImpl order = retrieveOrderById(orderId);
  if (order != null) {
      //2. Set its Status attribute to a new value
    order.setOrderStatusCode(newStatus);
      //3. Commit the transaction
    try {
      getDBTransaction().commit();
    }
    catch (JboException ex) {
      getDBTransaction().rollback();
      throw ex;
    }
  }
}
```

The example for *removing* an entity row would be the same, except that after finding the existing entity, you would use the following line instead to remove the entity before committing the transaction:

```
// Remove the entity instead!
order.remove();
```

4.14.4 How to Create a New Entity Row

In addition to using the entity definition to find existing entity rows, you can also use it to create new ones. In the case of product entities, you could write a `createProduct()` method like the one shown in Example 4–24 to accept the name and description of a new product, and return the new product ID assigned to it. This example assumes that the `ProductId` attribute of the `ProductBaseEO` entity object has been updated to have the `DBSequence` type (see Section 4.10.10, "How to Get Trigger-Assigned Primary Key Values from a Database Sequence"). This setting ensures that the attribute value is refreshed to reflect the value of the trigger from the corresponding database table, assigned to it from the table's sequence in the application schema.

Before you begin:

It may be helpful to have an understanding of when to use a programmatic approach for working with entity objects and associations. For more information, see Section 4.14, "Working Programmatically with Entity Objects and Associations."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To create an entity row:

1. Find the entity definition.

   Using the `getDefinitionObject()` method, the `createProduct()` method finds the entity definition for the `Product` entity.

2. Create a new instance.

   Using the `createInstance2()` method on the entity definition, the `createProduct()` method creates a new instance of the entity object.

   > **Note:** The method name has a `2` at the end. The regular `createInstance()` method has `protected` access and is designed to be customized as described Section D.3.4, "EntityImpl Class" of Appendix D, "Most Commonly Used ADF Business Components Methods." The second argument of type `AttributeList` is used to supply attribute values that *must* be supplied at create time; it is *not* used to initialize the values of all attributes found in the list. For example, when creating a new instance of a composed child entity row using this API, you must supply the value of a composing parent entity's foreign key attribute in the `AttributeList` object passed as the second argument. Failure to do so results in an `InvalidOwnerException`.

3. Set attribute values.

Using the attribute setter methods on the entity object, the `createProduct()` method assigns values for the `Name`, `Status`, and other attributes in the new entity row.

4. Commit the transaction.

   Calling `commit()` on the current transaction object, the `createProduct()` method commits the transaction.

5. Return the trigger-assigned product ID to the caller.

   Using the attribute getter method to retrieve the value of the `ProductId` attribute as a `DBSequence`, and then calling `getSequenceNumber().longValue()`, the `createProduct()` method returns the sequence number as a `long` value to the caller.

Example 4–24 Creating a New Entity Row

```
/* Create a new Product and Return its new id */
public long createProduct(String name, String status, String shipCode) {
      //1. Find the entity definition for the Product entity
  EntityDefImpl productDef = ProductBaseEOImpl.getDefinitionObject();
      //2. Create a new instance of a Product entity
  ProductBaseEOImpl newProduct =
(ProductBaseEOImpl)productDef.createInstance2(getDBTransaction(),null);
      //3. Set attribute values
  newProduct.setProductName(name);
  newProduct.setProductStatus(status);
  newProduct.setShippingClassCode(shipCode);
  newProduct.setSupplierId(new Number(100));
  newProduct.setListPrice(new Number(499));
  newProduct.setMinPrice(new Number(479));
  newProduct.setCreatedBy("Test Client");
  newProduct.setLastUpdatedBy("Test Client");
  newProduct.setCategoryId(new Number(5));
      //4. Commit the transaction
  try {
    getDBTransaction().commit();
  }
  catch (JboException ex) {
    getDBTransaction().rollback();
    throw ex;
  }
      //5. Access the database-trigger-assigned ProductId value and return it
  DBSequence newIdAssigned = newProduct.getProductId();
  return newIdAssigned.getSequenceNumber().longValue();
}
```

4.14.5 Assigning the Primary Key Value Using an Oracle Sequence

As an alternative to using a trigger-assigned value (as described in Section 4.10.10, "How to Get Trigger-Assigned Primary Key Values from a Database Sequence"), you can assign the value to a primary key when creating a new row using an Oracle sequence. This metadata-driven approach allows you to centralize the code to retrieve the primary key into a single Java file that can be reused by multiple entity objects.

Example 4–25 shows a simple `CustomEntityImpl` framework extension class on which the entity objects are based. Its overridden `create()` method tests for the presence of a custom attribute-level metadata property named `SequenceName` and if

detected, populates the attribute's default value from the next number in that sequence.

Example 4–25 CustomEntityImpl Framework Extension Class

```
package sample;

import oracle.jbo.AttributeDef;
import oracle.jbo.AttributeList;
import oracle.jbo.server.EntityImpl;
import oracle.jbo.server.SequenceImpl;

public class CustomEntityImpl extends EntityImpl {
    protected void create(AttributeList attributeList) {
        super.create(attributeList);
        for (AttributeDef def : getEntityDef().getAttributeDefs()) {
            String sequenceName = (String)def.getProperty("SequenceName");
            if (sequenceName != null) {
               SequenceImpl s = new SequenceImpl(sequenceName,getDBTransaction());
               setAttribute(def.getIndex(),s.getSequenceNumber());
            }
        }
    }
}
```

Before you begin:

It may be helpful to have an understanding of when to use a programmatic approach for working with entity objects and associations. For more information, see Section 4.14, "Working Programmatically with Entity Objects and Associations."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To assign the primary key value using an Oracle sequence:

1. Create the `CustomEntityImpl.java` file in your project, and insert the code shown in Example 4–25.
2. In the Application Navigator, double-click the entity you want to edit.
3. In the overview editor, click the **Attributes** navigation tab, and select the attribute you want to edit.
4. Click the **Details** tab, and set the attribute **Type** to **Number**.
5. Click the **Custom Properties** tab, and click the **Add** icon.
6. Create a custom property with `SequenceName` for the name, and the name of the database sequence for the value.

   For example, for a `Dept` entity, you could define the custom property `SequenceName` on its `Deptno` attribute with the value `DEPT_TABLE_SEQ`.

4.14.6 How to Update a Deleted Flag Instead of Deleting Rows

For auditing purposes, once a row is added to a table, sometimes your requirements may demand that rows never be physically deleted from the table. Instead, when the end user deletes the row in the user interface, the value of a `DELETED` column should be updated from "`N`" to "`Y`" to mark it as deleted. You can use two method overrides to alter an entity object's default behavior to achieve this effect.

To accomplish this, you need to perform the following tasks:

- Update a deleted flag when a row is removed, as described in Section 4.14.6.1, "Updating a Deleted Flag When a Row Is Removed."
- Force the entity object to be updated instead of deleted, as described in Section 4.14.6.2, "Forcing an Update DML Operation Instead of a Delete."

4.14.6.1 Updating a Deleted Flag When a Row Is Removed

To update a deleted flag when a row is removed, enable a custom Java class for your entity object and override the `remove()` method to set the deleted flag before calling the `super.remove()` method. Example 4–26 shows what this would look like in the custom Java class of an entity object. It is important to set the attribute *before* calling `super.remove()` since an attempt to set the attribute of a deleted row will encounter the `DeadEntityAccessException`.

This example presumes that you've altered the table to have an additional `DELETED` column, and synchronized the entity object with the database to add the corresponding `Deleted` attribute.

Example 4–26 Updating a Deleted Flag When an Entity Row Is Removed

```
// In your custom Java entity class
public void remove() {
  setDeleted("Y");
  super.remove();
}
```

The row will still be removed from the row set, but it will have the value of its `Deleted` flag modified to "`Y`" in the entity cache. The second part of implementing this behavior involves forcing the entity to perform an `UPDATE` instead of an `DELETE` when it is asked to perform its DML operation. You need to implement both parts for a complete solution.

4.14.6.2 Forcing an Update DML Operation Instead of a Delete

To force an entity object to be updated instead of deleted, override the `doDML()` method and write code that conditionally changes the `operation` flag. When the operation flag equals `DML_DELETE`, your code will change it to `DML_UPDATE` instead. Example 4–27 shows what this would look like in the custom Java class of an entity object.

This example presumes that you've altered the table to have an additional `DELETED` column, and synchronized the entity object with the database to add the corresponding `Deleted` attribute.

Example 4–27 Forcing an Update DML Operation Instead of a Delete

```
// In your custom Java entity class
protected void doDML(int operation, TransactionEvent e) {
   if (operation == DML_DELETE) {
     operation = DML_UPDATE;
   }
   super.doDML(operation, e);
  }
```

With this overridden `doDML()` method in place to complement the overridden `remove()` method described in Section 4.14.6.1, any attempt to remove an entity through any view object with a corresponding entity usage will update the `DELETED`

column instead of physically deleting the row. Of course, in order to prevent "deleted" products from appearing in your view object query results, you will need to appropriately modify their `WHERE` clauses to include only products `WHERE DELETED = 'N'`.

4.14.7 How to Control Entity Posting Order to Prevent Constraint Violations

Due to database constraints, when you perform DML operations to save changes to a number of related entity objects in the same transaction, the order in which the operations are performed can be significant. If you try to insert a new row containing foreign key references *before* inserting the row being referenced, the database can complain with a constraint violation. You must understand what the default order for the processing of entity objects is during commit time and how to programmatically influence that order when necessary.

Note: The example in this section refers to the `ControllingPostingOrder` project of the `AdvancedEntityExamples` application workspace in the `StandaloneExamples` module of the Fusion Order Demo application.

4.14.7.1 Default Post Processing Order

By default, when you commit the transaction the entity objects in the pending changes list are processed in chronological order, in other words, the order in which the entities were added to the list. This means that, for example, if you create a new `Product` and then a new `Supplier` related to that product, the new `Product` will be inserted first and the new `Supplier` second.

4.14.7.2 Compositions and Default Post Processing Order

When two entity objects are related by a *composition*, the strict chronological ordering is modified automatically to ensure that composed parent and child entity rows are saved in an order that prevents violating any constraints. This means, for example, that a new parent entity row is inserted before any new composed children entity rows.

4.14.7.3 Overriding postChanges() to Control Post Order

If your related entities are associated but *not* composed, then you need to write a bit of code to ensure that the related entities get saved in the appropriate order.

4.14.7.3.1 Observing the Post Ordering Problem First Hand Consider the `newProductForNewSupplier()` custom method from a `PostModule` application module in Example 4–28. It accepts a set of parameters and:

1. Creates a new `Product`.
2. Creates a new `Supplier`.
3. Sets the product ID to which the server request pertains.
4. Commits the transaction.
5. Constructs a `Result` Java bean to hold new product ID and supplier ID.
6. Returns the result.

Note: The code makes the assumption that both `Products.ProductId` and `Suppliers.SupplierId` have been set to have `DBSequence` data type to populate their primary keys based on a sequence.

Example 4–28 Creating a New Product, Then a New Supplier, and Returning the New IDs

```
// In PostModuleImpl.java
public Result newProductForNewSupplier(String supplierName,
                                       String supplierStatus,
                                       String productName,
                                       String productStatus,
                                       Number listPrice,
                                       Number minPrice,
                                       String shipCode) {
    oracle.jbo.domain.Date today = new Date(Date.getCurrentDate());
    Number objectId = new Number(0);
    // 1. Create a new product
    ProductsBaseImpl newProduct = createNewProduct();
    // 2. Create a new supplier
    SuppliersImpl newSupplier = createNewSupplier();
    newSupplier.setSupplierName(supplierName);
    newSupplier.setSupplierStatus(supplierStatus);
    newSupplier.setCreatedBy("PostingModule");
    newSupplier.setCreationDate(today);
    newSupplier.setLastUpdatedBy("PostingModule");
    newSupplier.setLastUpdateDate(today);
    newSupplier.setObjectVersionId(objectId);
    // 3. Set the supplier id to which the product pertains
    newProduct.setSupplierId(newSupplier.getSupplierId().getSequenceNumber());
    newProduct.setProductName(productName);
    newProduct.setProductStatus(productStatus);
    newProduct.setListPrice(listPrice);
    newProduct.setMinPrice(minPrice);
    newProduct.setShippingClassCode(shipCode);
    newProduct.setCreatedBy("PostingModule");
    newProduct.setCreationDate(today);
    newProduct.setLastUpdatedBy("PostingModule");
    newProduct.setLastUpdateDate(today);
    newProduct.setObjectVersionId(objectId);
    // 4. Commit the transaction
    getDBTransaction().commit();
    // 5. Construct a bean to hold new supplier id and product id
    Result result = new Result();
    result.setProductId(newProduct.getProductId().getSequenceNumber());
    result.setSupplierId(newSupplier.getSupplierId().getSequenceNumber());
    // 6. Return the result
    return result;
}
private ProductsBaseImpl createNewProduct(){
   EntityDefImpl productDef = ProductsBaseImpl.getDefinitionObject();
   return (ProductsBaseImpl) productDef.createInstance2(getDBTransaction(), null);
}

private SuppliersImpl createNewSupplier(){
   EntityDefImpl supplierDef = SuppliersImpl.getDefinitionObject();
   return (SuppliersImpl) supplierDef.createInstance2(getDBTransaction(), null);
}
```

If you add this method to the application module's client interface and test it from a test client program, you get an error:

```
oracle.jbo.DMLConstraintException:
JBO-26048: Constraint "PRODUCT_SUPPLIER_FK" violated during post operation:
"Insert" using SQL Statement
"BEGIN
  INSERT INTO PRODUCTS(
    SUPPLIER_NAME,SUPPLIER_STATUS,PRODUCT_NAME,
    PRODUCT_STATUS,LIST_PRICE,MIN_PRICE, SHIPPING_CLASS_CODE)
   VALUES (?,?,?,?,?,?,?)
   RETURNING PRODUCT_ID INTO ?;
END;".
## Detail 0 ##
java.sql.SQLException:
ORA-02291: integrity constraint (FOD.PRODUCT_SUPPILER_FK) violated
           - parent key not found
```

When the `PRODUCTS` row is inserted, the database complains that the value of its `SUPPLIER_ID` foreign key doesn't correspond to any row in the `SUPPLIERS` table. This occurred because:

- The code created the `Product` before the `Supplier`
- `Products` and `Suppliers` entity objects are associated but not composed
- The DML operations to save the new entity rows is done in chronological order, so the new `Product` gets inserted before the new `Supplier`.

4.14.7.3.2 Forcing the Supplier to Post Before the Product To remedy the problem of attempting to add a product with a not-yet-valid supplier ID, you could reorder the lines of code in the example to create the `Supplier` first, then the `Product`. While this would address the immediate problem, it still leaves the chance that another application developer could create things in an incorrect order.

The better solution is to make the entity objects *themselves* handle the posting order so the posting will work correctly regardless of the order of creation. To do this, you need to override the `postChanges()` method in the entity that contains the foreign key attribute referencing the associated entity object and write code as shown in Example 4–29. In this example, since it is the `Product` that contains the foreign key to the `Supplier` entity, you need to update the `Product` to conditionally force a related, new `Supplier` to post before the service request posts itself.

The code tests whether the entity being posted is in the `STATUS_NEW` or `STATUS_MODIFIED` state. If it is, it retrieves the related product using the `getSupplier()` association accessor. If the related `Supplier` also has a post-state of `STATUS_NEW`, then *first* it calls `postChanges()` on the related parent row before calling `super.postChanges()` to perform its own DML.

Example 4–29 Overriding postChanges() in ProductsBaseImpl to Post Supplier First

```
// In ProductsBaseImpl.java
public void postChanges(TransactionEvent e) {
  /* If current entity is new or modified */
  if (getPostState() == STATUS_NEW ||
      getPostState() == STATUS_MODIFIED) {
    /* Get the associated supplier for the product */
    SuppliersImpl supplier = getSupplier();
    /* If there is an associated supplier */
    if (supplier != null) {
      /* And if its post-status is NEW */
```

```
      if (supplier.getPostState() == STATUS_NEW) {
        /*
         * Post the supplier first, before posting this
         * entity by calling super below
         */
        supplier.postChanges(e);
      }
    }
  }
  super.postChanges(e);
}
```

If you were to re-run the example now, you would see that without changing the creation order in the `newProductForNewSupplier()` method's code, entities now post in the correct order — first new `Supplier`, then new `Product`. Yet, there is still a problem. The constraint violation still appears, but now for a different reason.

If the primary key for the `Suppliers` entity object were user-assigned, then the code in Example 4–29 would be all that is required to address the constraint violation by correcting the post ordering.

Note: An alternative to the programmatic technique discussed here, which solves the problem at the Java EE application layer, is the use of deferrable constraints at the database layer. If you have control over your database schema, consider defining (or altering) your foreign key constraints to be `DEFERRABLE INITIALLY DEFERRED`. This database setting causes the database to defer checking the constraint until transaction commit time. When this is done, the application can perform DML operations in any order, provided that by `COMMIT` time all appropriate related rows have been saved and would alleviate the parent/child ordering. However, you would still need to write the code to cascade-update the foreign key values if the parent's primary key is assigned from a sequence, as described in Section 4.14.7.3.3, "Associations Based on DBSequence-Valued Primary Keys," and Section 4.14.7.3.4, "Refreshing References to DBSequence-Assigned Foreign Keys."

In this example, however, the `Suppliers.SupplierId` is assigned from a database sequence, and not user-assigned. So when a new `Suppliers` entity row gets posted, its `SupplierId` attribute is refreshed to reflect the database-assigned sequence value. The foreign key value in the `Products.SupplierId` attribute referencing the new supplier is "orphaned" by this refreshing of the supplier's ID value. When the product's row is saved, its `SUPPLIER_ID` value still doesn't match a row in the `SUPPLIERS` table, and the constraint violation occurs again. The next two sections discuss the solution to address this "orphaning" problem.

4.14.7.3.3 Associations Based on DBSequence-Valued Primary Keys

Recall from Section 4.10.10, "How to Get Trigger-Assigned Primary Key Values from a Database Sequence," that when an entity object's primary key attribute is of `DBSequence` type, during the transaction in which it is created, its numerical value is a unique, temporary negative number. If you create a number of associated entities in the same transaction, the relationships between them are based on this temporary negative key value. When the entity objects with `DBSequence`-value primary keys are posted, their primary key is refreshed to reflect the correct database-assigned sequence number,

leaving the associated entities that are still holding onto the temporary negative foreign key value "orphaned."

For entity objects based on a *composition*, when the parent entity object's `DBSequence`-valued primary key is refreshed, the composed children entity rows automatically have their temporary negative foreign key value updated to reflect the owning parent's refreshed, database-assigned primary key. This means that for composed entities, the "orphaning" problem does not occur.

However, when entity objects are related by an association that is not a composition, you need to write a little code to insure that related entity rows referencing the temporary negative number get updated to have the refreshed, database-assigned primary key value. The next section outlines the code required.

4.14.7.3.4 Refreshing References to DBSequence-Assigned Foreign Keys

When an entity like `Suppliers` in this example has a `DBSequence`-valued primary key, and it is referenced as a foreign key by other entities that are associated with (but not composed by) it, you need to override the `postChanges()` method as shown in Example 4–30 to save a reference to the row set of entity rows that might be referencing this new `Suppliers` row. If the status of the current `Suppliers` row is `New`, then the code assigns the `RowSet`-valued return of the `getProduct()` association accessor to the `newProductsBeforePost` member field before calling `super.postChanges()`.

Example 4–30 Saving Reference to Entity Rows Referencing This New Supplier

```
// In SuppliersImpl.java
RowSet newProductsBeforePost = null;
public void postChanges(TransactionEvent TransactionEvent) {
  /* Only bother to update references if Product is a NEW one */
  if (getPostState() == STATUS_NEW) {
    /*
     * Get a rowset of products related
     * to this new supplier before calling super
     */
    newProductsBeforePost = (RowSet)getProductsBase();
  }
  super.postChanges(TransactionEvent);
}
```

This saved `RowSet` object is then used by the overridden `refreshFKInNewContainees()` method shown in Example 4–31. It gets called to allow a new entity row to cascade-update its refreshed primary key value to any other entity rows that were referencing it before the call to `postChanges()`. It iterates over the `ProductsBaseImpl` rows in the `newProductsBaseBeforePost` row set (if non-null) and sets the new supplier ID value of each one to the new sequence-assigned supplier value of the newly posted `Suppliers` entity.

Example 4–31 Cascade-Updating Entity Rows with New SupplierId Value

```
// In SuppliersImpl.java
protected void refreshFKInNewContainees() {
  if (newProductsBeforePost != null) {
    Number newSupplierId = getSupplierId().getSequenceNumber();
    /*
     * Process the rowset of products that referenced
     * the new supplier prior to posting, and update their
     * SupplierId attribute to reflect the refreshed SupplierId value
     * that was assigned by a database sequence during posting.
```

```
     */
    while (newProductsBeforePost.hasNext()) {
      ProductsBaseImpl svrReq =
        (ProductsBaseImpl)newProductsBeforePost.next();
      product.setSupplierId(newSupplierId);
    }
    closeNewProductRowSet();
  }
}
```

After implementing this change, the code in Example 4–28 runs without encountering any database constraint violations.

4.14.8 Advanced Entity Association Techniques

This section describes several advanced techniques for working with associations between entity objects.

4.14.8.1 Modifying Association SQL Clause to Implement Complex Associations

When you need to represent a more complex relationship between entities than one based only on the equality of matching attributes, you can modify the association's SQL clause to include more complex criteria. For example, sometimes the relationship between two entities depends on effective dates. A `Product` may be related to a `Supplier`, but if the name of the supplier changes over time, each row in the `SUPPLIERS` table might include additional `EFFECTIVE_FROM` and `EFFECTIVE_UNTIL` columns that track the range of dates in which that product row is (or was) in use. The relationship between a `Product` and the `Supplier` with which it is associated might then be described by a combination of the matching `SupplierId` attributes and a condition that the product's `RequestDate` lie between the supplier's `EffectiveFrom` and `EffectiveUntil` dates.

You can set up this more complex relationship in the overview editor for the association. First, add any additional necessary attribute pairs on the Relationship page, which in this example would include one (`EffectiveFrom`, `RequestDate`) pair and one (`EffectiveUntil`, `RequestDate`) pair. Then, on the Query page you can edit the **Where** field to change the WHERE clause to be:

```
(:Bind_SupplierId = Product.SUPPLIER_ID) AND
(Product.REQUEST_DATE BETWEEN :Bind_EffectiveFrom
                              AND :Bind_EffectiveUntil)
```

For more information about creating associations, see Section 4.3, "Creating and Configuring Associations."

4.14.8.2 Exposing View Link Accessor Attributes at the Entity Level

When you create a view link between two entity-based view objects, on the View Link Properties page, you have the option to expose view link accessor attributes both at the view object level as well as at the entity object level. By default, a view link accessor is exposed only at the view object level of the destination view object. By selecting the appropriate **In Entity Object:** ***SourceEntityName*** or **In Entity Object:*****DestinationEntityName*** checkbox, you can opt to have JDeveloper include a view link attribute in either or both of the source or destination entity objects. This can provide a handy way for an entity object to access a set of related view rows, especially when the query to produce the rows depends only on attributes of the current row.

4.14.8.3 Optimizing Entity Accessor Access by Retaining the Row Set

Each time you retrieve an entity association accessor row set, by default the entity object creates a new `RowSet` object to allow you to work with the rows. This does *not* imply re-*executing* the query to produce the results each time, only creating a new instance of a `RowSet` object with its default iterator reset to the "slot" before the first row. To force the row set to refresh its rows from the database, you can call its `executeQuery()` method.

Since there is a small amount of overhead associated with creating the row set, if your code makes numerous calls to the same association accessor attributes, you can consider enabling the association accessor row set retention for the source entity object in the association. You can enable retention of the association accessor row set using the overview editor for the entity object that is the source for the association accessor. Select **Retain Association Accessor Rowset** in the **Tuning** section of the General page of the overview editor for the entity object.

Alternatively, you can enable a custom Java *entity collection* class for your entity object. As with other custom entity Java classes you've seen, you do this on the Select Java Options dialog that you open from the Java page of the overview editor for the entity object. In the dialog, select **Generate Entity Collection Class**. Then, in the *`YourEntityColl`*`Impl` class that JDeveloper creates for you, override the `init()` method, and add a line after `super.init()` that calls the `setAssociationAccessorRetained()` method passing `true` as the parameter. It affects all association accessor attributes for that entity object.

When this feature is enabled for an entity object, since the association accessor row set it not re-created each time, the current row of its default row set iterator is also retained as a side-effect. This means that your code will need to explicitly call the `reset()` method on the row set you retrieve from the association accessor to reset the current row in its default row set iterator back to the "slot" before the first row.

Note, however, that with accessor retention enabled, your failure to call `reset()` each time before you iterate through the rows in the accessor row set can result in a subtle, hard-to-detect error in your application. For example, say you iterate over the rows in an association accessor row set as shown in Example 4–32 to calculate some aggregate total.

Example 4–32 Iterating Over a Row Set Incorrectly

```
// In your custom Java entity class
RowSet rs = (RowSet)getProducts();
while (rs.hasNext()) {
  ProductImpl r = (ProductImpl)rs.next();
  // Do something important with attributes in each row
}
```

The first time you work with the accessor row set, the code will work. However, since the row set (and its default row set iterator) are retained, the second and any subsequent time you access the row set, the current row will already be at the end of the row set and the while loop will be skipped because `rs.hasNext()` will be `false`. Instead, with this feature enabled, write your accessor iteration code as shown in Example 4–33.

Example 4–33 Iterating Over a Row Set and Resetting to the First Row

```
// In your custom Java entity class
RowSet rs = (RowSet)getProducts();
rs.reset(); // Reset default row set iterator to slot before first row!
while (rs.hasNext()) {
```

```
  ProductImpl r = (ProductImpl)rs.next();
  // Do something important with attributes in each row
}
```

4.15 Creating Custom, Validated Data Types Using Domains

When you find yourself repeating the same sanity-checking validations on the values of similar attributes across multiple entity objects, you can save yourself time and effort by creating your own data types that encapsulate this validation. For example, imagine that across your business domain layer there are numerous entity object attributes that store strings that represent email addresses. One technique you could use to ensure that end users always enter a valid email address everywhere one appears in your business domain layer is to:

- Use a basic `String` data type for each of these attributes
- Add an attribute-level method validator with Java code that ensures that the String value has the format of a valid email address for each attribute

However, these approaches can become tedious quickly in a large application. Fortunately, ADF Business Components offers an alternative that allows you to create your own `EmailAddress` data type that represents an email address. After centralizing all of the sanity-checking regarding email address values into this new custom data type, you can use the `EmailAddress` as the type of every attribute in your application that represents an email address. By doing this, you make the intention of the attribute values more clear to other developers and simplify application maintenance by putting the validation in a single place. ADF Business Components calls these developer-created data types *domains*.

Domains are Java classes that extend the basic data types like `String`, `Number`, and `Date` to add constructor-time validation to ensure the candidate value passes relevant sanity checks. They offer you a way to define custom data types with cross-cutting behavior such as basic data type validation, formatting, and custom metadata properties in a way that is inherited by any entity objects or view objects that use the domain as the Java type of any of their attributes.

Note: The example in this section refers to the `SimpleDomains` project in the `AdvancedEntityExamples` application workspace in the `StandaloneExamples` module of the Fusion Order Demo application.

4.15.1 How to Create a Domain

To create a domain, use the Create Domain wizard. This wizard is available from the New Gallery in the **ADF Business Components** category.

Before you begin:

It may be helpful to have an understanding of using domains. For more information, see Section 4.15, "Creating Custom, Validated Data Types Using Domains."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To create a domain:

1. In the Application Navigator, right-click the project for which you want to create a domain and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Domain**, and click **OK**.
3. In the Create Domain wizard, on the Name page, specify a name for the domain and a package in which it will reside. To create a domain based on a simple Java type, leave **Domain for an Oracle Object Type** unselected.
4. Click **Next**.
5. On the Settings page, indicate the base type for the domain and the database column type to which it will map.

   For example, if you were creating a domain called `ShortEmailAddress` to hold eight-character short email addresses, you would set the base type to `String` and the **Database Column Type** to `VARCHAR2(8)`. You can set other common attribute settings on this panel as well.
6. Click **Finish** to create your domain.

4.15.2 What Happens When You Create a Domain

When you create a domain, JDeveloper creates its XML component definition in the subdirectory of your project's source path that corresponds to the package name you chose. For example, if you created the `ShortEmailAddress` domain in the `devguide.advanced.domains` package, JDeveloper would create the `ShortEmailAddress.xml` file in the `./devguide/advanced/domains` subdirectory. A domain always has a corresponding Java class, which JDeveloper creates in the `common` subpackage of the package where the domain resides. This means it would create the `ShortEmailAddress.java` class in the `devguide.advanced.domains.common` package. The domain's Java class is automatically generated with the appropriate code to behave in a way that is identical to one of the built-in data types.

4.15.3 What You May Need to Know About Domains

Domains can be created as a variety of different types, and have different characteristics than standard attributes. The sections that follow describe some of the things you may need to know about when working with domains.

4.15.3.1 Domains as Entity and View Object Attributes

After you've created a domain in a project, it appears among the list of available data types in the **Attribute Type** dropdown list in the entity object and view object wizards and dialogs. To use the domain as the type of a given attribute, just pick it from the list.

Note: The entity-mapped attributes in an entity-based view object inherit their data type from their corresponding underlying entity object attribute. Therefore, if the entity attribute uses a domain type, the matching view object attribute will as well. For transient or SQL-derived view object attributes, you can directly set the type to use a domain since it is not inherited from any underlying entity.

4.15.3.2 DataCreationException in Custom validate() Method

Typically, the only coding task you need to do for a domain is to write custom code inside the generated `validate()` method. Your implementation of the `validate()` method should perform your sanity checks on the candidate value being constructed, and throw a `DataCreationException` in the `oracle.jbo` package if the validation fails.

In order to throw an exception message that is translatable, you can create a message bundle class similar to the one shown in Example 4–34. Create it in the same `common` package as your domain classes themselves. The message bundle returns an array of {*MessageKeyString*,*TranslatableMessageString*} pairs.

Example 4–34 Custom Message Bundle Class for Domain Exception Messages

```
package devguide.advanced.simpledomains.common;

import java.util.ListResourceBundle;

public class ErrorMessages extends ListResourceBundle {
  public static final String INVALID_SHORTEMAIL = "30002";
  public static final String INVALID_EVENNUMBER = "30003";
  private static final Object[][] sMessageStrings = new String[][] {
      { INVALID_SHORTEMAIL,
        "A valid short email address has no @-sign or dot."},
      { INVALID_EVENNUMBER,
        "Number must be even."}
    };

  /**
   * Return String Identifiers and corresponding Messages
   * in a two-dimensional array.
   */
  protected Object[][] getContents() {
    return sMessageStrings;
  }
}
```

4.15.3.3 String Domains and String Value Aggregation

Since `String` is a base JDK type, a domain based on a `String` aggregates a `private mData String` member field to hold the value that the domain represents. Then, the class implements the `DomainInterface` expected by the ADF runtime, as well as the `Serializable` interface, so the domain can be used in method arguments or return types of the custom client interfaces of Oracle ADF components.

Example 4–35 shows the `validate()` method for a simple `ShortEmailAddress` domain class. It tests to make sure that the `mData` value does not contains an at-sign or a dot, and if it does, then the method throws `DataCreationException` referencing an appropriate message bundle and message key for the translatable error message.

Example 4–35 Simple ShortEmailAddress String-Based Domain Type with Custom Validation

```
public class ShortEmailAddress
               extends Object implements DomainInterface, Serializable {
  private String mData;
  // . . .
  /**Implements domain validation logic and throws a JboException on error. */
  protected void validate() {
    int atpos = mData.indexOf('@');
```

```
    int dotpos = mData.lastIndexOf('.');
    if (atpos > -1 || dotpos > -1) {
      throw new DataCreationException(ErrorMessages.class,
      ErrorMessages.INVALID_SHORTEMAIL,null,null);
    }
  }
  // . . .
}
```

4.15.3.4 Simple Domains and Built-In Types

Other simple domains based on a built-in type in the `oracle.jbo.domain` package extend the base type, as shown in Example 4–36. It illustrates the `validate()` method for a simple Number-based domain called `EvenNumber` that represents even numbers.

Example 4–36 Simple EvenNumber Number-Based Domain Type with Custom Validation

```
public class EvenNumber extends Number {
  // . . .
  /**
   * Validates that value is an even number, or else
   * throws a DataCreationException with a custom
   * error message.
   */
  protected void validate() {
    if (getValue() % 2 == 1) {
      throw new DataCreationException(ErrorMessages.class,
      ErrorMessages.INVALID_EVENNUMBER,null,null);
    }
  }
  // . . .
}
```

4.15.3.5 Simple Domains As Immutable Java Classes

When you create a simple domain based on one of the basic data types, it is an *immutable* class. That means that once you've constructed a new instance of it like this:

```
ShortEmailAddress email = new ShortEmailAddress("emailaddress1");
```

You cannot change its value. If you want to reference a different short email address, you just construct another one:

```
ShortEmailAddress email = new ShortEmailAddress("emailaddress2");
```

This is not a new concept because it's the same way that `String`, `Number`, and `Date` classes behave, among others.

4.15.3.6 Creating Domains for Oracle Object Types When Useful

Oracle Database supports the ability to create user-defined types in the database. For example, you could create a type called `POINT_TYPE` using the following DDL statement:

```
create type point_type as object (
   x_coord number,
   y_coord number
);
```

If you use user-defined types like `POINT_TYPE`, you can create domains based on them, or you can reverse-engineer tables containing columns of object type to have JDeveloper create the domain for you. You can use the Create Domain wizard to create Oracle object type domains manually.

To manually create an Oracle object type domain:

1. In the Application Navigator, right-click the project for which you want to create a domain and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Domain**, and click **OK**.
3. In the Create Domain wizard, on the Name page, select the **Domain for an Oracle Object Type** checkbox, then select the object type for which you want to create a domain from the **Available Types** list.
4. Click **Next**.
5. On the Settings page, use the **Attribute** dropdown list to switch between the multiple domain properties to adjust the settings as appropriate.
6. Click **Finish** to create the domain.

To reverse-engineer an Oracle object type domain:

In addition to manually creating object type domains, when you use the Business Components from Tables wizard and select a table containing columns of an Oracle object type, JDeveloper creates domains for those object types as part of the reverse-engineering process. For example, imagine you created a table like this with a column of type `POINT_TYPE`:

```
create table interesting_points(
  id number primary key,
  coordinates point_type,
  description varchar2(20)
);
```

If you create an entity object for the `INTERESTING_POINTS` table in the Business Components from Tables wizard, then you get both an `InterestingPoints` entity object and a `PointType` domain. The latter is generated, based on the `POINT_TYPE` object type, because it was required as the data type of the `Coordinates` attribute of the `InterestingPoints` entity object.

Unlike simple domains, object type domains are mutable. JDeveloper generates getter and setter methods into the domain class for each of the elements in the object type's structure. After changing any domain properties, when you set that domain as the value of a view object or entity object attribute, it is treated as a single unit. Oracle ADF does not track which domain properties have changed, only that a domain-valued attribute value has changed.

> **Note:** Domains based on Oracle object types are useful for working programmatically with data whose underlying type is an oracle object type. They can also simplify passing and receiving structure information to stored procedures. However, support for working with object type domains in the ADF binding layer is complete, so it's not straightforward to use object domain-valued attributes in declaratively databound user interfaces.

4.15.3.7 Quickly Navigating to the Domain Class

After selecting a domain in the Application Navigator, you can quickly navigate to its implementation class using one of the following methods:

- In the Application Navigator, right-click the domain and choose **Go to Domain Class**.
- In the Structure window, double-click the domain class.

4.15.3.8 Domains Packaged in the Common JAR

When you create a business components archive, as described in Section 38.3.1, "How to Package a Component into an ADF Library JAR," the domain classes and message bundle files in the *.common subdirectories of your project's source path get packaged into the `*CSCommon.jar`. They are classes that are common to both the middle-tier application server and to an eventual remote client you might need to support.

4.15.3.9 Custom Domain Properties and Attributes in Entity and View Objects

You can define custom metadata properties on a domain. Any entity object or view object attribute based on that domain inherits those custom properties as if they had been defined on the attribute itself. If the entity object or view object attribute defines the same custom property, its setting takes precedence over the value inherited from the domain.

4.15.3.10 Inherited Restrictive Properties of Domains in Entity and View Objects

JDeveloper enforces the declarative settings you impose at the domain definition level: they cannot be made *less* restrictive for the entity object or view object for an attribute based on the domain type. For example, if you define a domain to have its **Updatable** property set to **While New**, then when you use your domain as the Java type of an entity object attribute, you can set **Updatable** to be **Never** (more restrictive) but you cannot set it to be **Always**. Similarly, if you define a domain to be **Persistent**, you cannot make it transient later. When sensible for your application, set declarative properties for a domain to be as lenient as possible, so you can later make them more restrictive as needed.

4.16 Creating New History Types

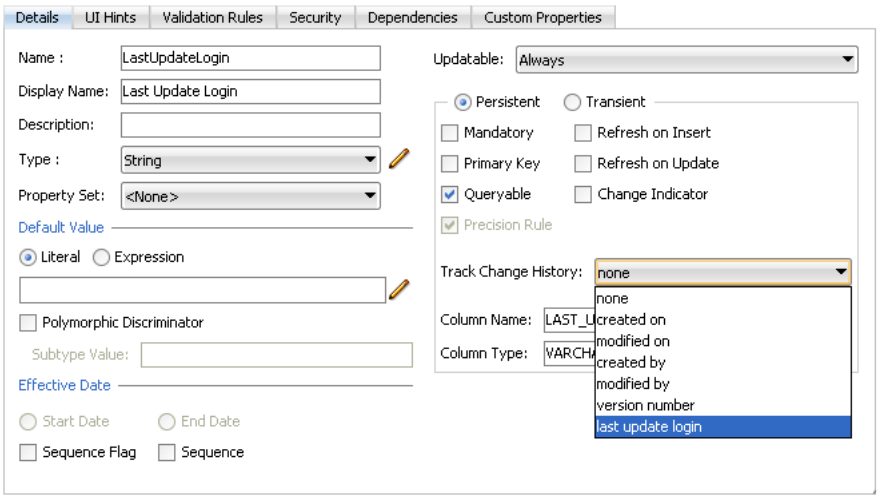
History types are used to track data specific to a point in time. JDeveloper ships with a number of history types, but you can also create your own. For more information on the standard history types and how to use them, see Section 4.10.12, "How to Track Created and Modified Dates Using the History Column."

4.16.1 How to Create New History Types

You are not limited to the history types provided: you can add or remove custom history types using the History Types page in the Preferences dialog, and then write custom Java code to implement the desired behavior. The code to handle custom history types should be written in your application-wide entity base class for reuse.

Figure 4–17 shows a custom type called `last update login` with type ID of `11`. Assume that `last_update_login` is a foreign key in the `FND_LOGINS` table.

Figure 4–17 New History Types in the Overview Editor



Before you begin:

It may be helpful to have an understanding of history types. For more information, see Section 4.16, "Creating New History Types."

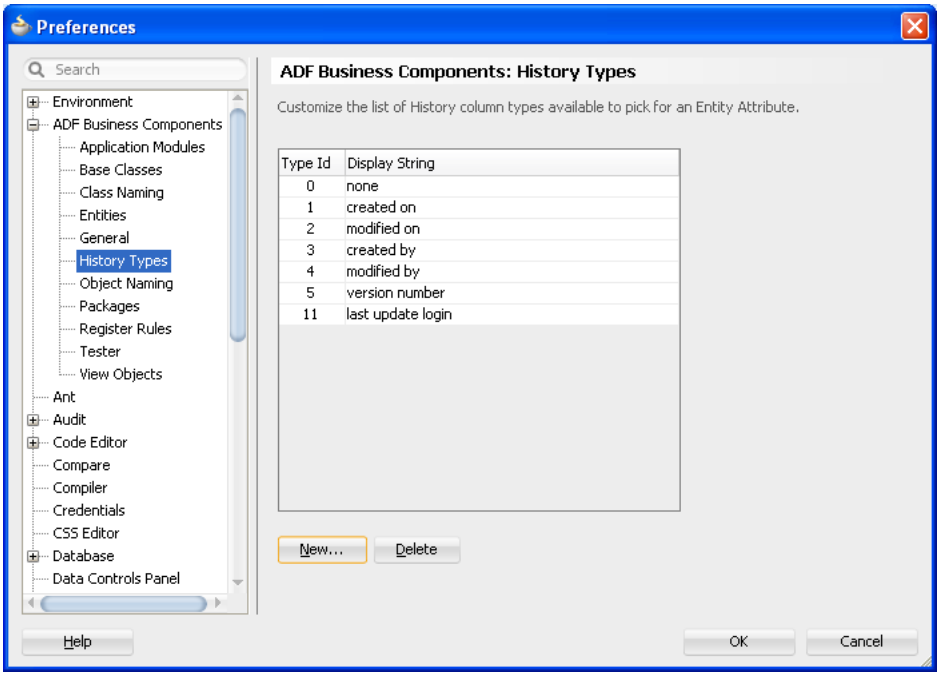
You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To create a custom history type:

1. From the **Tools** menu, choose **Preferences**.
2. In the Preferences dialog, expand **ADF Business Components** and click **History Types**.
3. On the History Types page, click **New**.
4. In the New History Type dialog, enter a string value for the name (spaces are allowed) and a numerical ID.

   The **Type Id** must be an integer between 11 and 126. The numerical values 0-10 are reserved for internal use. The display string is displayed in the **Track Change History** dropdown list the next time you use the overview editor. Figure 4–18 shows the new history type in the Preferences dialog.

Figure 4–18 Custom History Type in Preferences



5. Open the `EntityImpl.java` file and add a definition similar to the one in Example 4–37.

Example 4–37 History Type Definition

```
private static final byte LASTUPDATELOGIN_HISTORY_TYPE = 11;
```

6. Override the `getHistoryContextForAttribute(AttributeDefImpl attr)` method in the `EntityImpl` base class with code similar to Example 4–38.

Example 4–38 Overriding getHistoryContextForAttribute()

```
@Override
protected Object getHistoryContextForAttribute(AttributeDefImpl attr) {
    if (attr.getHistoryKind() == LASTUPDATELOGIN_HISTORY_TYPE) {
        // Custom History type logic goes here
    }
    else {
        return super.getHistoryContextForAttribute(attr);
    }
}
```

4.16.2 How to Remove a History Type

Because they are typically used for auditing values over the life of an application, it is rare that you would want to remove a history type. However, in the event that you need to do so, perform the following tasks:

1. Remove the history type from the JDeveloper history types list in the Preferences dialog.
2. Remove any custom code you implemented to support the history type in the base `EntityImpl.getHistoryContextForAttribute` method.

3. Remove all usages of the history type in the entity attribute metadata. Any attribute that you have defined to use this history type must be edited.

Before you begin:

It may be helpful to have an understanding of history types. For more information, see Section 4.16, "Creating New History Types."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To remove a history type from the JDeveloper history types list:

1. From the **Tools** menu, choose **Preferences**.
2. In the Preferences dialog, expand **ADF Business Components** and click **History Types**.
3. On the History Types page, select the history type that you want to remove and click **Delete**.

4.17 Basing an Entity Object on a PL/SQL Package API

If you have a PL/SQL package that encapsulates insert, update, and delete access to an underlying table, you can override the default DML processing event for the entity object that represents that table to invoke the procedures in your PL/SQL API instead. Often, such PL/SQL packages are used in combination with a companion database view. Client programs read data from the underlying table using the database view, and "write" data back to the table using the procedures in the PL/SQL package.

For example, say you want to create a `Product` entity object based on such a combination of a view and a package.

Given the `PRODUCTS` table in the Fusion Order Demo schema, consider a database view named `PRODUCTS_V`, created using the following DDL statement:

```
create or replace view products_v
as select product_id,name,image,description from products;
```

In addition, consider the simple `PRODUCTS_API` package shown in Example 4–39 that encapsulates insert, update, and delete access to the underlying `PRODUCTS` table.

Example 4–39 Simple PL/SQL Package API for the PRODUCTS Table

```
create or replace package products_api is
  procedure insert_product(p_prod_id number,
                           p_name varchar2,
                           p_supplier_id number,
                           p_list_price number,
                           p_min_price number,
                           p_ship_code varchar2);
  procedure update_product(p_prod_id number,
                           p_name varchar2,
                           p_supplier_id number,
                           p_list_price number,
                           p_min_price number,
                           p_ship_code varchar2);
  procedure delete_product(p_prod_id number);
end products_api;
```

To create an entity object based on this combination of database view and PL/SQL package, you would perform the following tasks:

1. Create a view-based entity object, as described in Section 4.17.1, "How to Create an Entity Object Based on a View."
2. Create a base class for the entity object, as described in Section 4.17.3, "How to Centralize Details for PL/SQL-Based Entities into a Base Class."
3. Implement the appropriate stored procedure calls, as described in Section 4.17.4, "How to Implement the Stored Procedure Calls for DML Operations."
4. Handle selecting and locking functionality, if necessary, as described in Section 4.17.5, "How to Add Select and Lock Handling."

Note: The example in these sections refers to the `EntityWrappingPLSQLPackage` project of the `AdvancedEntityExamples` application workspace in the `StandaloneExamples` module of the Fusion Order Demo application.

4.17.1 How to Create an Entity Object Based on a View

To create an entity object based on a view, you use the Create Entity Object wizard.

Before you begin:

It may be helpful to have an understanding of how entity objects can use the PS/SQL API. For more information, see Section 4.17, "Basing an Entity Object on a PL/SQL Package API."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

You will need to launch the Create Entity Object wizard as described in Section 4.2.2, "How to Create Single Entity Objects Using the Create Entity Wizard," and proceed through the wizard with the exceptions noted in the following procedure.

To create an entity object based on a view:

1. On the Name page, give the entity a name like `Product`.
2. In the Select Schema Object dialog, select the **Views** checkbox in the **Object Type** section.

   This enables the display of the available database views in the current schema in when you click **Query**.
3. Select the desired database view in the **Available Objects** list.
4. On the Attribute Settings page, use the **Select Attribute** dropdown list to choose the attribute that will act as the primary key, then select the **Primary Key** checkbox for that attribute.

Note: When defining the entity based on a view, JDeveloper cannot automatically determine the primary key attribute since database views do not have related constraints in the database data dictionary.

4.17.2 What Happens When You Create an Entity Object Based on a View

By default, an entity object based on a view performs all of the following directly against the underlying database view:

- `SELECT` statement (for `findByPrimaryKey()`)
- `SELECT FOR UPDATE` statement (for `lock()`), and
- `INSERT`, `UPDATE`, `DELETE` statements (for `doDML()`)

To use stored procedure calls, you will need to override the `doDML()` operations (as described in Section 4.17.3, "How to Centralize Details for PL/SQL-Based Entities into a Base Class"), and possibly override the `lock()` and `findByPrimaryKey()` handling (as described in Section 4.17.4, "How to Implement the Stored Procedure Calls for DML Operations").

4.17.3 How to Centralize Details for PL/SQL-Based Entities into a Base Class

If you plan to have more than one entity object based on a PL/SQL API, it's a smart idea to abstract the generic details into a base framework extension class. In doing this, you'll be using several of the concepts described in Chapter 12, "Extending Business Components Functionality." Start by creating a `PLSQLEntityImpl` class that extends the base `EntityImpl` class that each one of your PL/SQL-based entities can use as their base class. As shown in Example 4–40, you'll override the `doDML()` method of the base class to invoke a different helper method based on the operation.

> **Note:** If you are already using an extended entity implementation class for your entity, you can extend it further with the `PLSQLEntityImpl` class. For example, if you have a framework extension class named `zzEntityImpl`, you would create a `PLSQLEntityImpl` class that extends the `zzEntityImpl` class.

Example 4–40 Overriding doDML() to Call Different Procedures Based on the Operation

```
// In PLSQLEntityImpl.java
protected void doDML(int operation, TransactionEvent e) {
  // super.doDML(operation, e);
  if (operation == DML_INSERT)
    callInsertProcedure(e);
  else if (operation == DML_UPDATE)
    callUpdateProcedure(e);
  else if (operation == DML_DELETE)
    callDeleteProcedure(e);
}
```

In the `PLSQLEntityImpl.java` base class, you can write the helper methods so that they perform the default processing like this:

```
// In PLSQLEntityImpl.java
/* Override in a subclass to perform non-default processing */
protected void callInsertProcedure(TransactionEvent e) {
  super.doDML(DML_INSERT, e);
}
/* Override in a subclass to perform non-default processing */
protected void callUpdateProcedure(TransactionEvent e) {
  super.doDML(DML_UPDATE, e);
}
/* Override in a subclass to perform non-default processing */
protected void callDeleteProcedure(TransactionEvent e) {
```

```
  super.doDML(DML_DELETE, e);
}
```

After putting this infrastructure in place, when you base an entity object on the `PLSQLEntityImpl` class, you can use the **Source > Override Methods** menu item to override the `callInsertProcedure()`, `callUpdateProcedure()`, and `callDeleteProcedure()` helper methods and perform the appropriate stored procedure calls for that particular entity.

> **Note:** If you do not override these helper methods in a subclass, they will perform the default processing as defined in the superclass. You only need to override the operations in the `doDML()` method that you want to provide alternative processing for.

To simplify the task of implementing these calls, you could add the `callStoredProcedure()` helper method (described in Section 12.6, "Invoking Stored Procedures and Functions") to the `PLSQLEntityImpl` class as well. This way, any PL/SQL-based entity objects that extend this class can leverage the helper method.

4.17.4 How to Implement the Stored Procedure Calls for DML Operations

To implement the stored procedure calls for DML operations, you will need to create a custom Java class for the entity object and override the operations in it.

Before you begin:

It may be helpful to have an understanding of entity objects that are based on a PL/SQL package API. For more information, see Section 4.17, "Basing an Entity Object on a PL/SQL Package API."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To create the custom Java class with the override methods:

1. In the Application Navigator, double-click the entity object (for example, `Products`).
2. In the overview editor, click the **Java** navigation tab.
3. On the Java page of the overview editor, click the **Edit Java options** icon.
4. In the Select Java Options dialog, click **Classes Extend**.
5. In the Override Base Classes dialog, in the **Row** field, enter the package and class of the `PLSQLEntityImpl` class, or click **Browse** to search and select it.
6. Then select **Generate Entity Object Class**, and click **OK**.
7. In the Application Navigator, double-click the generated entity object class (for example, `ProductsImpl.java`).
8. From the **Source** menu, choose **Override Methods**.
9. In the Override Methods dialog, select the `callInsertProcedure()`, `callUpdateProcedure()`, and `callDeleteProcedure()` methods, and click **OK**.
10. Then enter the necessary code to override these procedures.

Example 4–41 shows some sample code that you would write in these overridden helper methods.

Example 4–41 Leveraging a Helper Method to Invoke Insert, Update, and Delete Procedures

```
// In ProductsImpl.java
protected void callInsertProcedure(TransactionEvent e) {
    callStoredProcedure("products_api.insert_product(?,?,?,?,?,?)",
         new Object[] { getProductId(), getProductName(), getSupplierId(),
               getListPrice(), getMinPrice(), getShippingClassCode() });
}
protected void callUpdateProcedure(TransactionEvent e) {
    callStoredProcedure("products_api.update_product(?,?,?,?,?,?)",
         new Object[] { getProductId(), getProductName(), getSupplierId(),
                        getListPrice(), getMinPrice(), getShippingClassCode() });
}
protected void callDeleteProcedure(TransactionEvent e) {
    callStoredProcedure("products_api.delete_product(?)",
         new Object[] { getProductId() });
}
```

At this point, if you create a default entity-based view object called `Products` for the `Products` entity object and add an instance of it to a `ProductsModule` application module you can quickly test inserting, updating, and deleting rows from the `Products` view object instance in the Oracle ADF Model Tester.

Often, overriding just the insert, update, and delete operations will be enough. The default behavior that performs the `SELECT` statement for `findByPrimaryKey()` and the `SELECT FOR UPDATE` statement for the `lock()` against the database view works for most basic kinds of views.

However, if the view is complex and does not support `SELECT FOR UPDATE` or if you need to perform the `findByPrimaryKey()` and `lock()` functionality using additional stored procedures API's, then you can use the technique described in Section 4.17.5, "How to Add Select and Lock Handling."

4.17.5 How to Add Select and Lock Handling

You can handle the `lock()` and `findByPrimaryKey()` functionality of an entity object by invoking stored procedures if necessary. Imagine that the `PRODUCTS_API` package were updated to contain the two additional procedures shown in Example 4–42. Both the `lock_product` and `select_product` procedures accept a primary key attribute as an `IN` parameter and return values for the remaining attributes using `OUT` parameters.

Example 4–42 Additional Locking and Select Procedures for the PRODUCTS Table

```
/* Added to PRODUCTS_API package */
  procedure lock_product(p_prod_id number,
                         p_name OUT varchar2,
                         p_supplier_id OUT number,
                         p_list_price OUT number,
                         p_min_price OUT number,
                         p_ship_code OUT varchar2);
  procedure select_product(p_prod_id number,
                         p_name OUT varchar2,
                         p_supplier_id OUT number,
                         p_list_price OUT number,
```

```
                                p_min_price OUT number,
                                p_ship_code OUT varchar2);
```

To add select and lock handling, you will need to perform the following tasks:

1. Update the base class to handle lock and select, as described in Section 4.17.5.1, "Updating PLSQLEntityImpl Base Class to Handle Lock and Select."
2. Update the entity object implementation class to implement the lock and select behaviors, as described in Section 4.17.5.2, "Implementing Lock and Select for the Product Entity."
3. Override the `lock()` method in the entity object implementation class to refresh the entity object after a `RowInconsistentException` has occurred, as described in Section 4.17.5.3, "Refreshing the Entity Object After RowInconsistentException."

4.17.5.1 Updating PLSQLEntityImpl Base Class to Handle Lock and Select

You can extend the `PLSQLEntityImpl` base class to handle the `lock()` and `findByPrimaryKey()` overrides using helper methods similar to the ones you added for insert, update, delete. At runtime, both the `lock()` and `findByPrimaryKey()` operations invoke the lower-level entity object method called `doSelect(boolean lock)`. The `lock()` operation calls `doSelect()` with a `true` value for the parameter, while the `findByPrimaryKey()` operation calls it passing `false` instead.

Example 4–43 shows the overridden `doSelect()` method in `PLSQLEntityImpl` to delegate as appropriate to two helper methods that subclasses can override as necessary.

Example 4–43 Overriding doSelect() to Call Different Procedures Based on the Lock Parameter

```
// In PLSQLEntityImpl.java
protected void doSelect(boolean lock) {
  if (lock) {
    callLockProcedureAndCheckForRowInconsistency();
  } else {
    callSelectProcedure();
  }
}
```

The two helper methods are written to just perform the default functionality in the base `PLSQLEntityImpl` class:

```
// In PLSQLEntityImpl.java
/* Override in a subclass to perform non-default processing */
protected void callLockProcedureAndCheckForRowInconsistency() {
  super.doSelect(true);
}
/* Override in a subclass to perform non-default processing */
protected void callSelectProcedure() {
  super.doSelect(false);
}
```

Notice that the helper method that performs locking has the name `callLockProcedureAndCheckForRowInconsistency()`. This reminds you that you need to perform a check to detect at the time of locking the row whether the newly selected row values are the same as the ones the entity object in the entity cache believes are the current database values.

To assist subclasses in performing this old-value versus new-value attribute comparison, you can add one final helper method to the `PLSQLEntityImpl` class like this:

```
// In PLSQLEntityImpl
protected void compareOldAttrTo(int attrIndex, Object newVal) {
  if ((getPostedAttribute(attrIndex) == null && newVal != null) ||
      (getPostedAttribute(attrIndex) != null && newVal == null) ||
      (getPostedAttribute(attrIndex) != null && newVal != null &&
       !getPostedAttribute(attrIndex).equals(newVal))) {
    throw new RowInconsistentException(getKey());
  }
}
```

4.17.5.2 Implementing Lock and Select for the Product Entity

With the additional infrastructure in place in the base `PLSQLEntityImpl` class, you can override the `callSelectProcedure()` and `callLockProcedureAndCheckForRowInconsistency()` helper methods in the entity object implementation class (for example, `ProductsImpl`). Because the `select_product` and `lock_product` procedures have `OUT` arguments, as described in Section 12.6.4, "How to Call Other Types of Stored Procedures," you need to use a JDBC `CallableStatement` object to perform these invocations.

Example 4–44 shows the code you would use to invoke the `select_product` procedure for the `ProductsImpl` entity object implementation class. It's performing the following basic steps:

1. Creating a `CallableStatement` for the PLSQL block to invoke.
2. Registering the `OUT` parameters and types, by one-based bind variable position.
3. Setting the `IN` parameter value.
4. Executing the statement.
5. Retrieving the possibly updated column values.
6. Populating the possibly updated attribute values in the row.
7. Closing the statement.

Example 4–44 Invoking the Stored Procedure to Select a Row by Primary Key

```
// In ProductsImpl.java
protected void callSelectProcedure() {
    String stmt = "begin products_api.select_product(?,?,?,?,?,?);end;";
    // 1. Create a CallableStatement for the PLSQL block to invoke
    CallableStatement st =
        getDBTransaction().createCallableStatement(stmt, 0);
    try {
        // 2. Register the OUT parameters and types
        st.registerOutParameter(2, VARCHAR2);
        st.registerOutParameter(3, NUMBER);
        st.registerOutParameter(4, NUMBER);
        st.registerOutParameter(5, NUMBER);
        st.registerOutParameter(6, VARCHAR2);

        // 3. Set the IN parameter value
        st.setObject(1, getProductId());
        // 4. Execute the statement
        st.executeUpdate();
        // 5. Retrieve the possibly updated column values
```

```
            String possiblyUpdatedName = st.getString(2);
            String possiblyUpdatedSupplierId = st.getString(3);
            String possiblyUpdatedListPrice = st.getString(4);
            String possiblyUpdatedMinPrice = st.getString(5);
            String possiblyUpdatedShipCode = st.getString(6);

            // 6. Populate the possibly updated attribute values in the row
            populateAttribute(PRODUCTNAME, possiblyUpdatedName, true, false,
                              false);
            populateAttribute(SUPPLIERID, possiblyUpdatedSupplierId, true,
                              false, false);
            populateAttribute(LISTPRICE, possiblyUpdatedListPrice, true, false,
                              false);
            populateAttribute(MINPRICE, possiblyUpdatedMinPrice, true, false,
                              false);
            populateAttribute(SHIPPINGCLASSCODE, possiblyUpdatedShipCode, true,
                              false, false);
        } catch (SQLException e) {
            throw new JboException(e);
        } finally {
            if (st != null) {
                try {
                    // 7. Closing the statement
                    st.close();
                } catch (SQLException e) {
                }
            }
        }
    }
```

Example 4–45 shows the code to invoke the `lock_product` procedure. It's doing basically the same steps as those in Example 4–44, with just the following two interesting differences:

- After retrieving the possibly updated column values from the `OUT` parameters, it uses the `compareOldAttrTo()` helper method inherited from the `PLSQLEntityImpl` to detect whether or not a `RowInconsistentException` should be thrown as a result of the row lock attempt.
- In the `catch (SQLException e)` block, it is testing to see whether the database has thrown the error:

  ```
  ORA-00054: resource busy and acquire with NOWAIT specified
  ```

  and if so, it again throws the ADF Business Components `AlreadyLockedException` just as the default entity object implementation of the `lock()` functionality would do in this situation.

Example 4–45 Invoking the Stored Procedure to Lock a Row by Primary Key

```
// In ProductsImpl.java
protected void callLockProcedureAndCheckForRowInconsistency() {
    String stmt = "begin products_api.lock_product(?,?,?,?,?,?);end;";
    CallableStatement st =
             getDBTransaction().createCallableStatement(stmt, 0);
        try {
            st.registerOutParameter(2, VARCHAR2);
            st.registerOutParameter(3, NUMBER);
            st.registerOutParameter(4, NUMBER);
            st.registerOutParameter(5, NUMBER);
            st.registerOutParameter(6, VARCHAR2);
```

```
            st.setObject(1, getProductId());
            st.executeUpdate();
            String possiblyUpdatedName = st.getString(2);
            String possiblyUpdatedSupplierId = st.getString(3);
            String possiblyUpdatedListPrice = st.getString(4);
            String possiblyUpdatedMinPrice = st.getString(5);
            String possiblyUpdatedShipCode = st.getString(6);
            compareOldAttrTo(PRODUCTNAME, possiblyUpdatedName);
            compareOldAttrTo(SUPPLIERID, possiblyUpdatedSupplierId);
            compareOldAttrTo(LISTPRICE, possiblyUpdatedListPrice);
            compareOldAttrTo(MINPRICE, possiblyUpdatedMinPrice);
            compareOldAttrTo(SHIPPINGCLASSCODE, possiblyUpdatedShipCode);
        } catch (SQLException e) {
            if (Math.abs(e.getErrorCode()) == 54) {
            throw new AlreadyLockedException(e);
         } else {
            throw new JboException(e);
         }
    } finally {
        if (st != null) {
            try {
                st.close();
            } catch (SQLException e) {
            }
        }
    }
}
```

With these methods in place, you have and entity object that wraps the PL/SQL package (in this case, a `Products` entity object with the `PRODUCTS_API` package) for all of its database operations. Due to the clean separation of the data querying functionality of view objects and the data validation and saving functionality of entity objects, you can now leverage this entity object in any way you would use a normal entity object. You can build as many different view objects as necessary that use this entity object as their entity usage.

4.17.5.3 Refreshing the Entity Object After RowInconsistentException

You can override the `lock()` method to refresh the entity object after a `RowInconsistentException` has occurred. Example 4–46 shows code that can be added to the entity object implementation class to catch the `RowInconsistentException` and refresh the entity object.

Example 4–46 Overridden lock() Method to Refresh Entity Object on RowInconsistentException

```
// In the entity object implementation class
@Override
public void lock() {
  try {
    super.lock();
  }
  catch (RowInconsistentException ex) {
    this.refresh(REFRESH_UNDO_CHANGES);
    throw ex;
  }
}
```

4.18 Basing an Entity Object on a Join View or Remote DBLink

If you need to create an entity object based on either of the following:

- Synonym that resolves to a remote table over a `DBLINK`
- View with `INSTEAD OF` triggers

Then you will encounter the following error if any of its attributes are marked as **Refresh on Insert** or **Refresh on Update**:

```
JBO-26041: Failed to post data to database during "Update"
## Detail 0 ##
ORA-22816: unsupported feature with RETURNING clause
```

These types of schema objects do not support the `RETURNING` clause, which by default the entity object uses to more efficiently return the refreshed values in the same database roundtrip in which the `INSERT` or `UPDATE` operation was executed.

4.18.1 How to Disable the Use of the RETURNING Clause

Because some types of schema objects do not support the `RETURNING` clause, you might need to disable the `RETURNING` clause in your entity object. The following procedures explains how to do that.

Before you begin:

It may be helpful to have an understanding of the types of schema objects don't support the `RETURNING` clause. For more information, see Section 4.18, "Basing an Entity Object on a Join View or Remote DBLink."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

To disable the use of the RETURNING clause for an entity object of this type:

1. Enable a custom entity definition class for the entity object.
2. In the custom entity definition class, override the `createDef()` method to call `setUseReturningClause(false)`.
3. If the **Refresh on Insert** attribute is the primary key of the entity object, you must specify some other attribute in the entity as an alternate unique key by setting the **Unique Key** property on it.

4.18.2 What Happens at Runtime: Disabling the RETURNING Clause

At runtime, when you have disabled the use of the `RETURNING` clause as described in Section 4.18.1, "How to Disable the Use of the RETURNING Clause,", the entity object implements the **Refresh on Insert** and **Refresh on Update** behavior using a separate `SELECT` statement to retrieve the values to refresh after insert or update as appropriate.

4.19 Using Inheritance in Your Business Domain Layer

Inheritance is a powerful feature of object-oriented development that can simplify development and maintenance when used appropriately. As shown in Section 12.9, "Creating Extended Components Using Inheritance," ADF Business Components supports using inheritance to create new components that extend existing ones in

order to add additional properties or behavior or modify the behavior of the parent component. Inheritance can be useful in modeling the different kinds of entities in your reusable business domain layer.

> **Note:** The example in this section refers to the `InheritanceAndPolymorphicQueries` project of the `AdvancedEntityExamples` application workspace in the `StandaloneExamples` module of the Fusion Order Demo application.

4.19.1 Understanding When Inheritance Can Be Useful

Your application's database schema might contain tables where different logical kinds of business information are stored in rows of the same table. These tables will typically have one column whose value determines the kind of information stored in each row. For example, the Fusion Order Demo application's `PERSONS` table stores information about customers, suppliers, and staff in the same table. It contains a `PERSON_TYPE_CODE` column whose value — `STAFF`, `CUST`, or `SUPP` — determines what kind of `PERSON` the row represents.

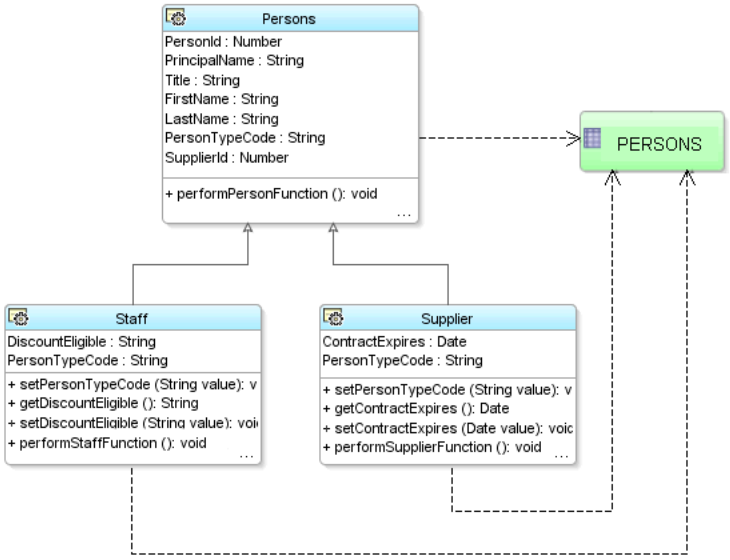
While the Fusion Order Demo implementation doesn't contain this differentiation, it's reasonable to assume that revisions of the application might require:

- Managing additional database-backed attributes that are specific to suppliers or specific to staff
- Implementing common behavior for all users that is different for suppliers or staff
- Implementing new functionality that is specific to only suppliers or only staff

Figure 4–19 shows what the business domain layer would look like if you created distinct `Persons`, `Staff`, and `Supplier` entity objects to allow distinguishing the different kinds of business information in a more formal way inside your application. Since suppliers and staff are special *kinds* of persons, their corresponding entity objects would extend the base `Persons` entity object. This base `Persons` entity object would contain all of the attributes and methods common to all types of users. The `performPersonFunction()` method in the figure represents one of these common methods.

Then, for the `Supplier` and `Staff` entity objects you can add specific additional attributes and methods that are unique to that kind of user. For example, `Supplier` has an additional `ContractExpires` attribute of type `Date` to track when the supplier's current contract expires. There is also a `performSupplierFunction()` method that is specific to suppliers. Similarly, the `Staff` entity object has an additional `DiscountEligible` attribute to track whether the person qualifies for a staff discount. The `performStaffFunction()` is a method that is specific to staff.

Figure 4–19 Distinguishing Persons, Suppliers, and Staff Using Inheritance



By modeling these different kinds of persons as distinct entity objects in an inheritance hierarchy in your domain business layer, you can simplify having them share common data and behavior and implement the aspects of the application that make them distinct.

4.19.2 How to Create Entity Objects in an Inheritance Hierarchy

To create entity objects in an inheritance hierarchy, you use the Create Entity Object wizard to create each entity.

The example described here assumes that you've altered the FOD application's `PERSONS` table by executing the following DDL statement to add two new columns to it:

```
alter table persons add (
  discount_eligible varchar2(1),
  contract_expires date
);
```

To create entity objects in an inheritance hierarchy, you will perform the following tasks:

1. Identify the discriminator column and values, as described in Section 4.19.2.1, "Identifying the Discriminator Column and Distinct Values."
2. Identify the subset of attributes for each entity object, as described in Section 4.19.2.2, "Identifying the Subset of Attributes Relevant to Each Kind of Entity."
3. Create the base entity object, as described in Section 4.19.2.3, "Creating the Base Entity Object in an Inheritance Hierarchy."
4. Create the subtype entity objects, as described in Section 4.19.2.4, "Creating a Subtype Entity Object in an Inheritance Hierarchy."

4.19.2.1 Identifying the Discriminator Column and Distinct Values

Before creating entity objects in an inheritance hierarchy based on a table containing different kinds of information, you should first identify which column in the table is used to distinguish which kind of row it is.

For example, in the Fusion Order Demo application's `PERSONS` table, this is the `PERSON_TYPE_CODE` column. Since it helps partition or "discriminate" the rows in the table into separate groups, this column is known as the *discriminator column*.

Next, determine the valid values that the discriminator column takes on in your table. You might know this, or you could execute a simple SQL statement in the JDeveloper **SQL Worksheet** to determine the answer. To access the worksheet:

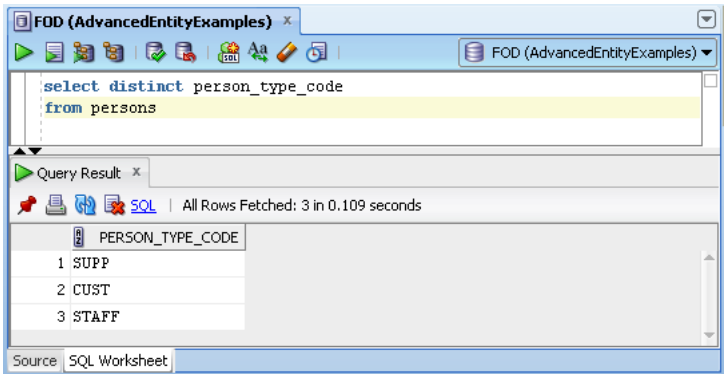
1. With the application open in JDeveloper, choose **Database Navigator** from the **View** menu.
2. Expand the workspace node, and select the connection.

   In this example, expand the **AdvancedEntityExamples** node and select the `FOD` connection.
3. Right-click the database connection, and choose **Open SQL Worksheet**.

   In this example, the database connection is FOD.

Figure 4–20 shows the results of performing a `SELECT DISTINCT` query in the SQL Worksheet on the `PERSON_TYPE_CODE` column in the `PERSONS` table. It confirms that the rows are partitioned into three groups based on the `PERSON_TYPE_CODE` discriminator values: `SUPP`, `STAFF`, and `CUST`.

Figure 4–20 Using the SQL Worksheet to Find Distinct Discriminator Column Values



4.19.2.2 Identifying the Subset of Attributes Relevant to Each Kind of Entity

Once you know how many different kinds of business entities are stored in the table, you will also know how many entity objects to create to model these distinct items. You'll typically create one entity object per kind of item. Then, to help determine which entity should act as the base of the hierarchy, you need to determine which subset of attributes is relevant to each kind of item.

For example, assume you determine that all of the attributes except `ContractExpires` and `DiscountEligible` are relevant to all users, and that:

- `ContractExpires` is specific to suppliers
- `DiscountEligible` is specific to staff.

This information leads you to determine that the `Persons` entity object should be the base of the hierarchy, with the `Supplier` and `Staff` entity objects each extending `Persons` to add their specific attributes.

4.19.2.3 Creating the Base Entity Object in an Inheritance Hierarchy

To create the base entity object in an inheritance hierarchy, you use the Create Entity Object wizard.

Before you begin:

It may be helpful to have an understanding of entity objects in an inheritance hierarchy. For more information, see Section 4.19, "Using Inheritance in Your Business Domain Layer."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

You will also need to determine the discriminator column and values, as described in Section 4.19.2.1, "Identifying the Discriminator Column and Distinct Values," and determine the attributes for each entity object, as described in Section 4.19.2.2, "Identifying the Subset of Attributes Relevant to Each Kind of Entity."

To create the base entity object

1. In the Application Navigator, right-click the project you want to add the entity object to, and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Entity Object**, and click **OK**.
3. In the Create Entity Object wizard, on the Name Page, provide a name and package for the entity, and select the schema object on which the entity will be based.

   In this example, name the entity object Persons and base it on the `PERSONS` table.
4. On the Attributes page, select the attributes in the **Entity Attributes** list that are not relevant to the base entity object (if any) and click **Remove** to remove them.

   In this example, you would remove the `DiscountEligible` and `ContractExpires` attributes from the list.
5. On the Attribute Settings page, use the **Select Attribute** dropdown list to choose the attribute that will act as the discriminator for the family of inherited entity objects and select the **Discriminator** checkbox to identify it as such. Importantly, you must also supply a **Default Value** for this discriminator attribute to identify rows of this base entity type.

   In this example, you would select the `PersonTypeCode` attribute, mark it as a discriminator attribute, and set its **Default Value** to the value "`cust`".

   > **Note:** Leaving the **Default Value** blank for a discriminator attribute is legal. A blank default value means that a row with the discriminator column value `IS NULL` will be treated as this base entity type.

6. Then click **Finish** to create the entity object.

4.19.2.4 Creating a Subtype Entity Object in an Inheritance Hierarchy

To create a subtype entity object in an inheritance hierarchy, you use the Create Entity Object wizard.

Before you begin:

It may be helpful to have an understanding of entity objects in an inheritance hierarchy. For more information, see Section 4.19, "Using Inheritance in Your Business Domain Layer."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

You will also need to perform the following tasks:

1. Determine the discriminator column and values, as described in Section 4.19.2.1, "Identifying the Discriminator Column and Distinct Values."
2. Determine the attributes for each entity object, as described in Section 4.19.2.2, "Identifying the Subset of Attributes Relevant to Each Kind of Entity."
3. Create the parent entity object from which your new entity object will extend, as described in Section 4.19.2.3, "Creating the Base Entity Object in an Inheritance Hierarchy."
4. Make sure that the parent entity has a discriminator attribute already identified.

   If it does not, use the overview editor to set the **Discriminator** property on the appropriate attribute of the parent entity before creating the inherited child.

To create the new subtype entity object in the hierarchy:

1. In the Application Navigator, right-click the project you want to add the entity object to, and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Entity Object**, and click **OK**.
3. In the Create Entity Object wizard, on the Name Page, provide a name and package for the entity, and click the **Browse** button next to the **Extends** field to select the parent entity from which the entity being created will extend.

   In this example, you would name the new entity `Staff` and select the `Persons` entity object in the **Extends** field.
4. On the Attributes page, the **Entity Attributes** list displays the attributes from the underlying table that are not included in the base entity object. Select the attributes you do not want to include in this entity object and click **Remove**.

   In this example, because you are creating the `Staff` entity you would remove the `ContractExpires` attribute and leave the `DiscountEligible` attribute.
5. Click **Override** to select the discriminator attribute so that you can customize the attribute metadata to supply a distinct **Default Value** for the `Staff` subtype.

   In this example, you would override the `PersonTypeCode` attribute.
6. On the Attribute Settings page, use the **Select Attribute** dropdown list to select the discriminator attribute. Change the **Default Value** field to supply a distinct default value for the discriminator attribute that defines the entity subtype being created.

   In this example, you would select the `PersonTypeCode` attribute and change its **Default Value** to the value "`staff`".

7. Click **Finish** to create the subtype entity object.

> **Note:** You can repeat the same steps to define the `Supplier` entity object that extends `Persons` to add the additional `ContractExpires` attribute and overrides the **Default Value** of the `UserRole` discriminator attribute to have the value "`supp`".

4.19.3 How to Add Methods to Entity Objects in an Inheritance Hierarchy

To add methods to entity objects in an inheritance hierarchy, enable the custom Java class for the entity object and use the source editor to add the method. Methods that are common to all entity objects in the hierarchy are added to the base entity, while subtype-specific methods are added to the subtype. You can also override methods from the base entity object in the subtypes as necessary.

4.19.3.1 Adding Methods Common to All Entity Objects in the Hierarchy

To add a method that is common to all entity objects in the hierarchy, you add the method to the implementation class of the base entity object.

Before you begin:

It may be helpful to have an understanding of entity objects in an inheritance hierarchy. For more information, see Section 4.19, "Using Inheritance in Your Business Domain Layer."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

You will also need to create the base entity object and subtypes in the hierarchy, as described in Section 4.19.2, "How to Create Entity Objects in an Inheritance Hierarchy."

Too add a method common to all entity objects in a hierarchy:

1. In the Application Navigator, double-click the base entity object implementation class (for example, `PersonsImpl.java`).

   If the base entity object doesn't have a custom Java implementation class, you'll need to create it.

   a. In the Application Navigator, double-click the entity object (for example, `Persons`).

   b. In the overview editor, click the **Java** navigation tab.

   c. On the Java page of the overview editor, click the **Edit Java options** icon.

   d. Then select **Generate Entity Object Class**, and click **OK**.

2. In the source editor, add the method.

   For example, you could add the following method to the `PersonsImpl` class for the base `Persons` entity object:

```
// In PersonsImpl.java
public void performPersonFunction() {
  System.out.println("## performPersonFunction as Customer");
}
```

Because this is the base entity object class, the methods you implement here are inherited by all subtype entity objects in the hierarchy.

4.19.3.2 Overriding Common Methods in a Subtype Entity Object

To override a method in a subtype entity object that is common to all entity objects in the hierarchy, you modify the common method inherited from the base entity object in the implementation class of the subtype entity object.

Before you begin:

It may be helpful to have an understanding of entity objects in an inheritance hierarchy. For more information, see Section 4.19, "Using Inheritance in Your Business Domain Layer."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

You will also need to perform the following tasks:

1. Create the base entity object and subtypes in the hierarchy, as described in Section 4.19.2, "How to Create Entity Objects in an Inheritance Hierarchy."
2. Create the common method in the base entity object, which your subtype entity object will override, as described in Section 4.19.3.1, "Adding Methods Common to All Entity Objects in the Hierarchy."

Too override a method in a subtype entity object:

1. In the Application Navigator, double-click the subtype entity object implementation class (for example, `StaffImpl.java`).

   If the subtype entity object doesn't have a custom Java implementation class, you'll need to create it.

   a. In the Application Navigator, double-click the entity object (for example, `Staff`).

   b. In the overview editor, click the **Java** navigation tab.

   c. On the Java page of the overview editor, click the **Edit Java options** icon.

   d. Then select **Generate Entity Object Class**, and click **OK**.
2. With the subtype entity object implementation class open in the source editor, choose **Override Methods** from the **Source** menu.
3. In the Override Methods dialog, select the method you want to override (for example, the `performPersonFunction()` method), and click **OK**.
4. In the source editor, customize the overridden method's implementation.

   For example, you could override the `performPersonFunction()` method in the `StaffImpl` class for the `Staff` subtype entity object and change the implementation to look like this:

```
// In StaffImpl.java
public void performPersonFunction() {
  System.out.println("## performPersonFunction as Staff");
}
```

When working with instances of entity objects in a subtype hierarchy, sometimes you will process instances of multiple different subtypes. Because the `Staff` and

`Supplier` entity objects are special kinds of `Persons`, you can write code that works with all of them using the more generic `PersonsImpl` type that they all have in common. When doing this generic kind of processing of classes that might be one of a *family* of subtypes in a hierarchy, Java will always invoke the most specific override of a method available.

This means that invoking the `performPersonFunction()` method on an instance of `PersonsImpl` that happens to really be the more specific `StaffImpl` subtype, will the result in printing out the following:

```
## performPersonFunction as Staff
```

instead of the default result that regular `PersonsImpl` instances would get:

```
## performPersonFunction as Customer
```

4.19.3.3 Adding Methods Specific to a Subtype Entity Object

To add a method that is specific to a subtype entity object in the hierarchy, you simply add the method in the implementation class of the subtype using the source editor.

Before you begin:

It may be helpful to have an understanding of entity objects in an inheritance hierarchy. For more information, see Section 4.19, "Using Inheritance in Your Business Domain Layer."

You may also find it helpful to understand additional functionality that can be added using other entity object features. For more information, see Section 4.1.2, "Additional Functionality for Entity Objects."

You will also need to create the base entity object and subtypes in the hierarchy, as described in Section 4.19.2, "How to Create Entity Objects in an Inheritance Hierarchy."

Too add a method specific to a subtype entity object:

1. In the Application Navigator, double-click the subtype entity object implementation class (for example, `SupplierImpl.java`).

   If the base entity object doesn't have a custom Java implementation class, you'll need to create it.

   a. In the Application Navigator, double-click the entity object (for example, `Supplier`).

   b. In the overview editor, click the **Java** navigation tab.

   c. On the Java page of the overview editor, click the **Edit Java options** icon.

   d. Then select **Generate Entity Object Class**, and click **OK**.

2. In the source editor, add the method.

   For example, you could add a `performSupplierFunction()` method to the `SuppierImpl` class for the base `Supplier` entity object:

   ```
   // In SupplierImpl.java
   public void performSupplierFunction() {
     System.out.println("## performSupplierFunction called");
   }
   ```

4.19.4 What You May Need to Know About Using Inheritance

When using inheritance, you can also introduce a new base entity, find subtype entities using a primary key, and create view objects with polymorphic entity usages.

4.19.4.1 When to Introduce a New Base Entity

In the `InheritanceAndPolymorphicQueries` example project, the `Persons` entity object corresponds to a concrete kind of row in the `PERSONS` table and it also played the role of the base entity in the hierarchy. In other words, all of its attributes were common to all entity objects in the hierarchy.

A situation might arise, however, where the `Persons` entity object required a property that was specific to customers, but not common to staff or suppliers. Typically, for example, customers participate in customer satisfaction surveys, but staff and suppliers do not. The `Persons` entity would require a `LastSurveyDate` attribute to handle this requirement, but it wouldn't make sense to have `Staff` and `Supplier` entity objects inherit it.

In this case, you can introduce a new entity object (for example, `BasePersons`) to act as the base entity in the hierarchy. It would have all of the attributes common to all `Persons`, `Staff`, and `Supplier` entity objects. Then each of the three entities that correspond to concrete rows that appear in the table could have some attributes that are inherited from `BasePersons` and some that are specific to the individual subtype. In the `BasePersons` type, so long as you mark the `PersonTypeCode` attribute as a discriminator attribute, you can just leave the **Default Value** blank (or some other value that does *not* occur in the `PERSON_TYPE_CODE` column in the table). Because you will not use instances of the `BasePersons` entity in the application, it doesn't matter what its discriminator default value is.

4.19.4.2 Subtype Entity Objects and the findByPrimaryKey() Method

When you use the `findByPrimaryKey()` method on an entity definition, it only searches the entity cache for the entity object type on which you call it. In the `InheritanceAndPolymorphicQueries` example project, this means that if you call `PersonsImpl.getDefinitionObject()` to access the entity definition for the `Persons` entity object when you call `findByPrimaryKey()` on it, you will only find entities in the cache that happen to be customers. Sometimes this is exactly the behavior you want.

However, if you want to find an entity object by primary key allowing the possibility that it might be a subtype in an inheritance hierarchy, then you can use the `findByPKExtended()` methodfrom the `EntityDefImpl` class instead.

For example, if you have created subtypes of the `Persons` entity object, this alternative finder method would find an entity object by primary key whether it is a customer, supplier, or staff. You can then use the Java `instanceof` operator to test which type you found, and then cast the `PersonsImpl` object to the more specific entity object type to work with features specific to that subtype.

4.19.4.3 View Objects with Polymorphic Entity Usages

When you create an entity-based view object with an entity usage corresponding to a base entity object in an inheritance hierarchy, you can configure the view object to query rows corresponding to multiple different subtypes in the base entity's subtype hierarchy. Each row in the view object will use the appropriate subtype entity object as the entity row part, based on matching the value of the discriminator attribute. See Section 42.6.2, "How to Create a View Object with a Polymorphic Entity Usage," for specific instructions on setting up and using these view objects.

5

Defining SQL Queries Using View Objects

This chapter describes how to create view objects that join, filter, sort, and aggregate data for use in the application.

This chapter includes the following sections:

- Section 5.1, "About View Objects"
- Section 5.2, "Populating View Object Rows from a Single Database Table"
- Section 5.3, "Populating View Object Rows with Static Data"
- Section 5.4, "Limiting View Object Rows Using Effective Date Ranges"
- Section 5.5, "Working with Multiple Tables in Join Query Results"
- Section 5.6, "Working with Multiple Tables in a Master-Detail Hierarchy"
- Section 5.7, "Working with a Single Table in a Recursive Master-Detail Hierarchy"
- Section 5.8, "Working with View Objects in Declarative SQL Mode"
- Section 5.9, "Working with View Objects in Expert Mode"
- Section 5.10, "Working with Bind Variables"
- Section 5.11, "Working with Named View Criteria"
- Section 5.12, "Working with List of Values (LOV) in View Object Attributes"
- Section 5.13, "Defining UI Hints for View Objects"
- Section 5.14, "Adding Calculated and Transient Attributes to a View Object"

5.1 About View Objects

A *view object* is an Oracle Application Development Framework (Oracle ADF) component that encapsulates a SQL query and simplifies working with its results. There are several types of view objects that you can create in your data model project:

- Read-only view objects when updates to data are not necessary (can also be entity-based)
- Entity-based view objects when data updates will be performed
- Static data view objects for data defined by the view object itself
- Programmatically populated view objects (for more information, see Chapter 42, "Advanced View Object Techniques")

An *entity-based view object* can be configured to support updatable rows when you create view objects that map their attributes to the attributes of one or more existing

entity objects. The mapped entity object is saved as an entity usage in the view object definition. In this way, entity-based view objects cooperate automatically with entity objects to enable a fully updatable data model. The entity-based view object queries just the data needed for the client-facing task and relies on its mapped entity objects to automatically validate and save changes made to its view rows. Like the read-only view object, an entity-based view object encapsulates a SQL query, it can be linked into master-detail hierarchies, and it can be used in the data model of your application modules.

View objects with no entity usage definition are always read-only. They do not pick up entity-derived default values, they do not reflect pending changes, and they do not reflect updated reference information. In contrast to entity-based view objects, *read-only view objects* require you to write the query using the SQL query language. The Create View Object wizard and overview editor for entity-based view objects, on the other hand, simplify this task by helping you to construct the SQL query declaratively. For this reason, it is almost always preferable to create a non-updatable, entity-mapped view object, even when you want to create a view object just to read data. Additionally, as an alternative to creating view objects that specify a SQL statement at design time, you can create entity-mapped view objects that dynamically generate SQL statements at runtime.

There remain a few situations where it is still preferable to create a non-entity-mapped view object to read data, including SQL-based validation, Unions, and Group By queries.

When a view object has one or more underlying entity usages, you can create new rows, and modify or remove queried rows. The entity-based view object coordinates with underlying entity objects to enforce business rules and to permanently save the changes to the database. In addition, entity-based view objects provide these capabilities that do not exist with read-only view objects:

- Changes in cache (updates, inserts, deletes) managed by entities survive the view object's execution boundary.
- Changes made to relevant entity object attributes through other view objects in the same transaction are immediately reflected.
- Attribute values of new rows are initialized to the values from the underlying entity object attributes.
- Changes to foreign key attribute values cause reference information to get updated.
- Validation for row (entity) level is supported.
- Composition feature, including validation, locking, ordered-updates is supported.
- Support for effective dating, change indicator, and business events.

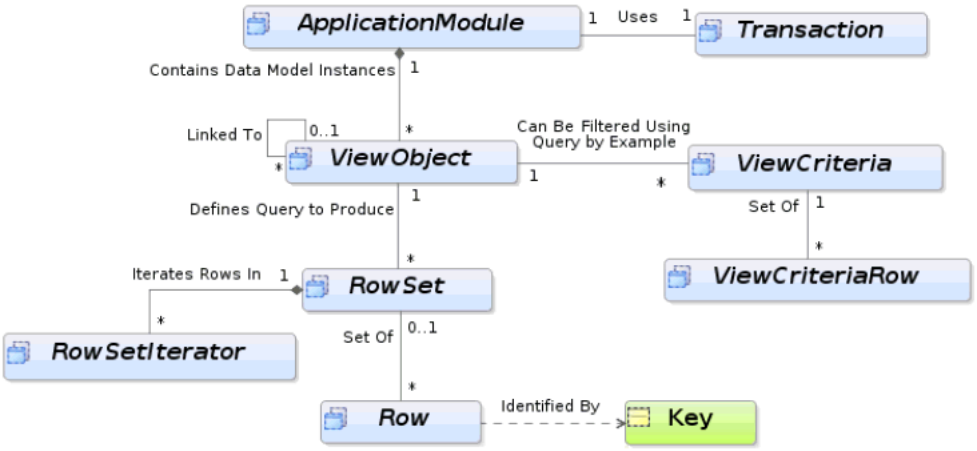
5.1.1 View Object Use Cases and Examples

This chapter helps you understand these view object concepts as illustrated in Figure 5–1:

- You define a view object by providing a SQL query (either defined explicitly or declaratively).
- You use view object instances in the context of an application module that provides the database transaction for their queries.
- You can link a view object to one or more others to create master-detail hierarchies.

- At runtime, the view object executes your query and produces a set of rows (represented by a `RowSet` object).
- Each row is identified by a corresponding row key.
- You iterate through the rows in a row set using a row set iterator.
- You can filter the row set a view object produces by applying a set of Query-by-Example criteria rows.

Figure 5–1 A View Object Defines a Query and Produces a Row Set of Rows

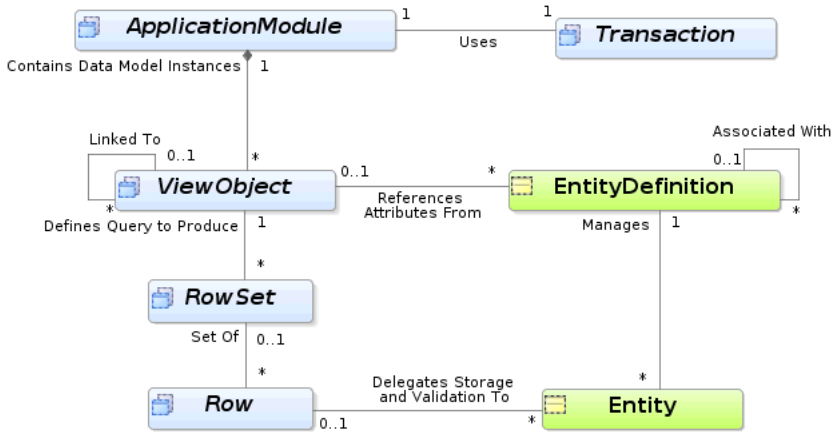


This chapter explains how instances of entity-based view objects contained in the data model of your application module enable clients to search for, update, insert, and delete business services layer information in a way that combines the full data shaping power of SQL with the clean, object-oriented encapsulation of reusable domain business objects. And all without requiring a line of code.

This chapter helps you to understand these entity-based view object concepts as illustrated in Figure 5–2:

- You define an updatable view object by referencing attributes from one or more entity objects.
- You can use multiple, associated entity objects to simplify working with reference information.
- You can define view links based on underlying entity associations.
- You use your entity-based view objects in the context of an application module that provides the transaction.
- At runtime, the view row delegates the storage and validation of its attributes to underlying entity objects.

Figure 5–2 View Objects and Entity Objects Collaborate to Enable an Updatable Data Model



5.1.2 Additional Functionality for View Objects

You may find it helpful to understand other ADF features before you start working with view objects. Following are links to other functionality that may be of interest.

- For additional information about using Groovy script wherever expressions are supported in view object definitions, see Section 3.6, "Overview of Groovy Scripting Language Support."
- For details about using the interactive Oracle ADF Model Tester to validate view object query results, see Chapter 6, "Testing View Instance Queries."
- For details about creating a data model consisting of view object instances, see Chapter 9, "Implementing Business Services with Application Modules."
- For a quick reference to the most common code that you will typically write, use, and override in your custom view object classes, see Appendix D, "Most Commonly Used ADF Business Components Methods."
- For API documentation related to the `oracle.jbo` package, see the following Javadoc reference document:
  - *Oracle Fusion Middleware Java API Reference for Oracle ADF Model*

5.2 Populating View Object Rows from a Single Database Table

View objects provide the means to retrieve data from a data source. In the majority of cases, the data source will be a database and the mechanism to retrieve data is the SQL query. ADF Business Components can work with JDBC to pass this query to the database and retrieve the result.

When view objects use a SQL query, query columns map to view object attributes in the view object. The definition of these attributes, saved in the view object's XML definition file, reflect the properties of these columns, including data types and precision and scale specifications.

Performance Tip: If the query associated with the view object contains values that may change from execution to execution, use bind variables. Using bind variables in the query allows the query to reexecute without needing to reparse the query on the database. You can add bind variables to the view object in the Query page of the overview editor for the view object. For more information, see Section 5.10, "Working with Bind Variables."

Using the same Create View Object wizard, you can create view objects that either map to the attributes of existing entity objects or not. Only entity-based view objects automatically coordinate with mapped entity objects to enforce business rules and to permanently save data model changes. Additionally, you can disable the Updatable feature for entity-based view objects and work entirely declaratively to query read-only data. Alternatively, you can use the wizard or editor's expert mode to work directly with the SQL query language, but the view object you create will not support the transaction features of the entity-based view object.

While there is a small amount of runtime overhead associated with the coordination between view object rows and entity object rows, weigh this against the ability to keep the view object definition entirely declarative and maintain a customizable view object. Queries that cannot be expressed in entity objects, and that therefore require expert-mode query editing, include Unions and Group By queries. Expert mode-based view objects are also useful in SQL-based validation queries used by the view object-based Key Exists validator. Again, it is worth repeating that, by definition, using expert mode to define a SQL query means the view object must be read-only.

For more information about the differences between entity-based view objects and read-only view objects, see Section 5.1, "About View Objects."

5.2.1 How to Create an Entity-Based View Object

Creating an entity-based view object is the simplest way to create a view object. It is even easier than creating an expert-mode, read-only view object, since you don't have to type in the SQL statement yourself. An entity-based view object also offers significantly more runtime functionality than its expert-mode counterpart.

In an entity-based view object, the view object and entity object play cleanly separated roles:

- The view object is the data *source*: it retrieves the data using SQL.
- The entity object is the data *sink*: it handles validating and saving data changes.

Because view objects and entity objects have cleanly separated roles, you can build a hundred different view objects — projecting, filtering, joining, sorting the data in whatever way your user interfaces require, application after application — without any changes to the reusable entity object. In fact, it is possible that the development team responsible for the core business services layer of entity objects might be completely separate from another team responsible for the specific application modules and view objects needed to support the end-user environment. This relationship is enabled by metadata that the entity-based view object encapsulates. The metadata specifies how the `SELECT` list columns are related to the attributes of one or more underlying entity objects.

Your entity-based view object may be based on more than one database table. To use database joins to add multiple tables to the view object, see Section 5.5, "Working with Multiple Tables in Join Query Results."

5.2.1.1 Creating an Entity-Based View Object from a Single Table

To create an entity-based view object, use the Create View Object wizard, which is available from the New Gallery.

Before you begin:

It may be helpful to have an understanding of entity-based view objects. For more information, see Section 5.2.1, "How to Create an Entity-Based View Object."

You will need to complete this task:

Create the desired entity objects as described in Section 4.2.1, "How to Create Multiple Entity Objects and Associations from Existing Tables."

To create an entity-based view object from a single table:

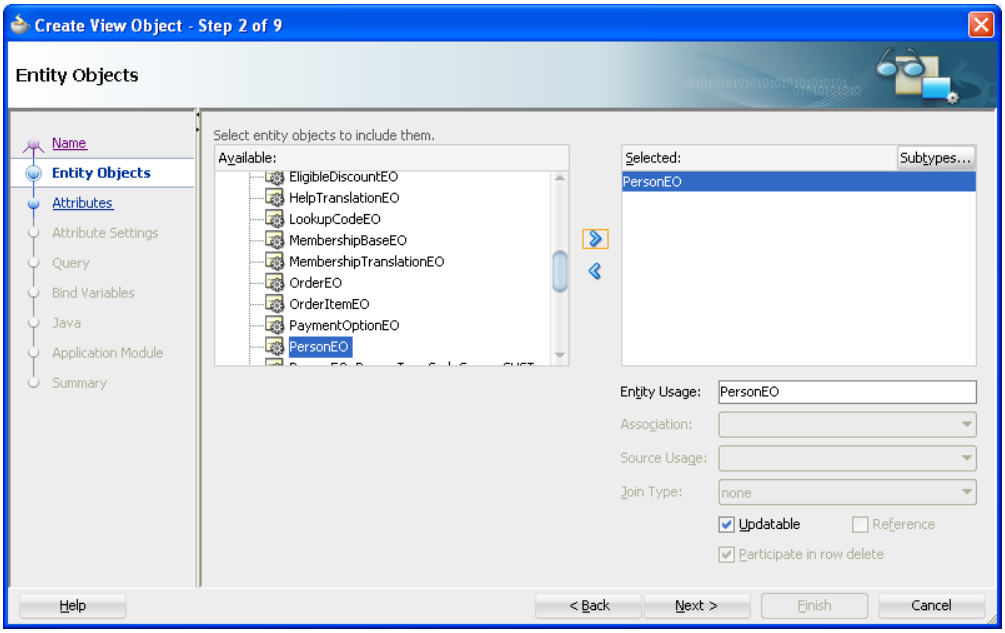
1. In the Application Navigator, right-click the project in which you want to create the view object and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **View Object**, and click **OK**.

   If this is the first component you're creating in the project, the Initialize Business Components Project dialog appears to allow you to select a database connection.
3. In the Initialize Business Components Project dialog, select the database connection or choose **New** to create a connection. Click **OK**.
4. In the Create View Object wizard, on the Name page, enter a package name and a view object name. Keep the default setting **Entity object** selected to indicate that you want this view object to manage data with its base entity object. Click **Next**.
5. On the Entity Objects page, select an entity object whose data you want to use in the view object. Click **Next**.

   An entry in this list is known as an *entity usage*, since it records the entity objects that the view object will be using. Each entry could also be thought of as an *entity reference*, since the view object references attributes from that entity. For information about working table joins to create additional entity usages, see Section 5.5, "Working with Multiple Tables in Join Query Results."

   For example, Figure 5–3 shows the result after shuttling the `PersonEO` entity object into the **Selected** list.

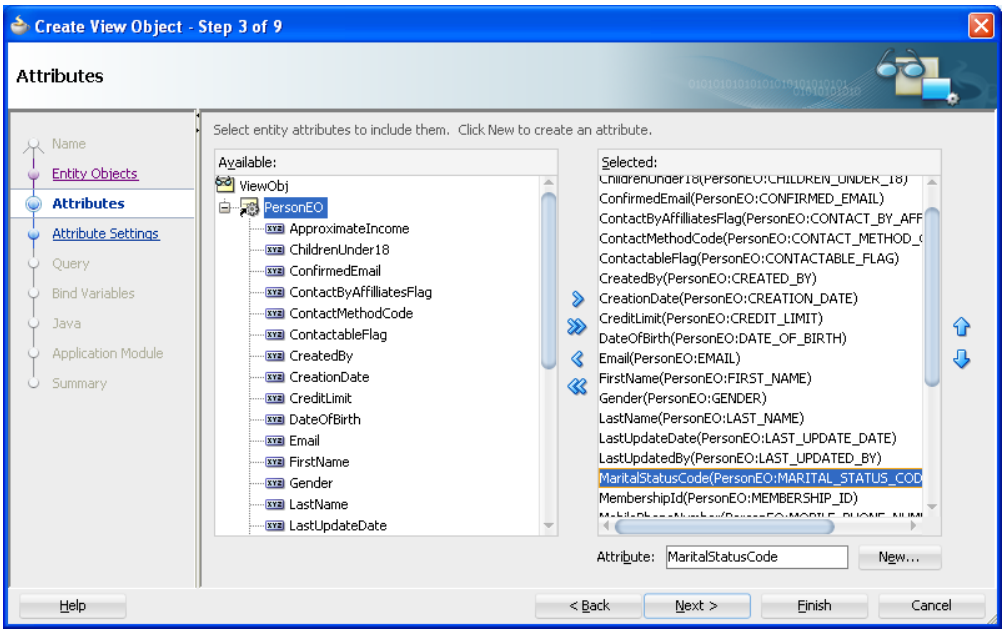
Figure 5–3 Create View Object Wizard, Entity Objects Page



6. On the Attributes page, select the attributes you want to include from each entity usage in the **Available** list and shuttle them to the **Selected** list. Click **Next**.

   For example, Figure 5–4 shows the attributes have been selected from the `PersonEO`.

Figure 5–4 Create View Object Wizard, Attributes Page



7. On the Attribute Settings page, optionally, use the **Select Attribute** dropdown list to switch between the view object attributes in order to change their names or any of their initial settings.

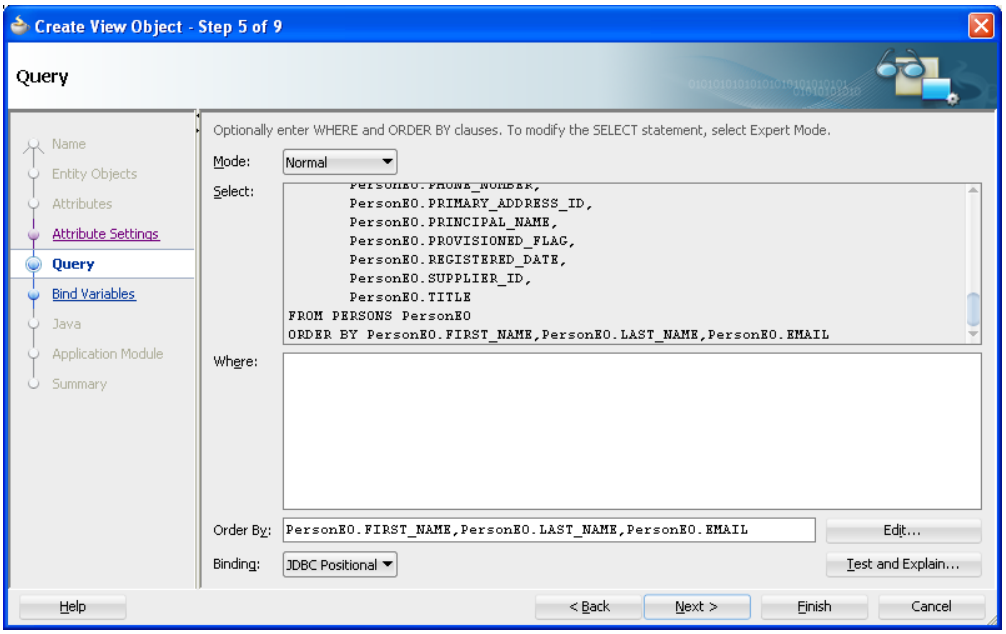
   For more information about any of the attribute settings, press F1 or click **Help**.

8. On the Query page, optionally, add a `WHERE` and `ORDER BY` clause to the query to filter and order the data as required. JDeveloper automatically generates the SELECT statement based on the entity attributes you've selected.

   Do not include the `WHERE` or `ORDER BY` keywords in the **Where** and **Order By** field values. The view object adds those keywords at runtime when it executes the query.

   For example, Figure 5–5 shows the `ORDER BY` clause is specified to order the data by first name, last name, and email.

Figure 5–5 Create View Object Wizard, Query Page



9. When you are satisfied with the view object, click **Finish**.

5.2.1.2 Creating a View Object with All the Attributes of an Entity Object

When you want to allow the client to work with *all* of the attributes of an underlying entity object, you can use the Create View Object wizard as described in Section 5.2.1.1, "Creating an Entity-Based View Object from a Single Table." After selecting the entity object, simply select *all* of its attributes on the Attributes page. However, for this frequent operation, there is an even quicker way to perform the same task in the Application Navigator.

Before you begin:

It may be helpful to have an understanding of entity-based view objects. For more information, see Section 5.2.1, "How to Create an Entity-Based View Object."

You will need to complete this task:

Create the desired entity objects as described in Section 4.2.1, "How to Create Multiple Entity Objects and Associations from Existing Tables."

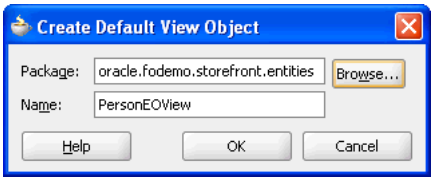
To create a default entity-based view object:

1. In the Application Navigator, right-click the entity object and choose **New Default View Object**.

2. Provide a package and component name for the new view object in the Create Default View Object dialog.

   In the Create Default View Object dialog you can click **Browse** to select the package name from the list of existing packages. For example, in Figure 5–6, clicking Browse locates `oracle.fodemo.storefront.enties` package on the classpath for the `StoreFrontService` project in the `StoreFrontModule` application.

Figure 5–6 Shortcut to Creating a Default View Object for an Entity Object



The new entity-based view object created will be identical to one you could have created with the Create View Object wizard. By default, it will have a single entity usage referencing the entity object you selected in the Application Navigator, and will include all of its attributes. It will initially have neither a `WHERE` nor `ORDER BY` clause, and you may want to use the overview editor for the view object to:

- Remove unneeded attributes
- Refine its selection with a `WHERE` clause
- Order its results with an `ORDER BY` clause
- Customize any of the view object properties

5.2.2 What Happens When You Create an Entity-Based View Object

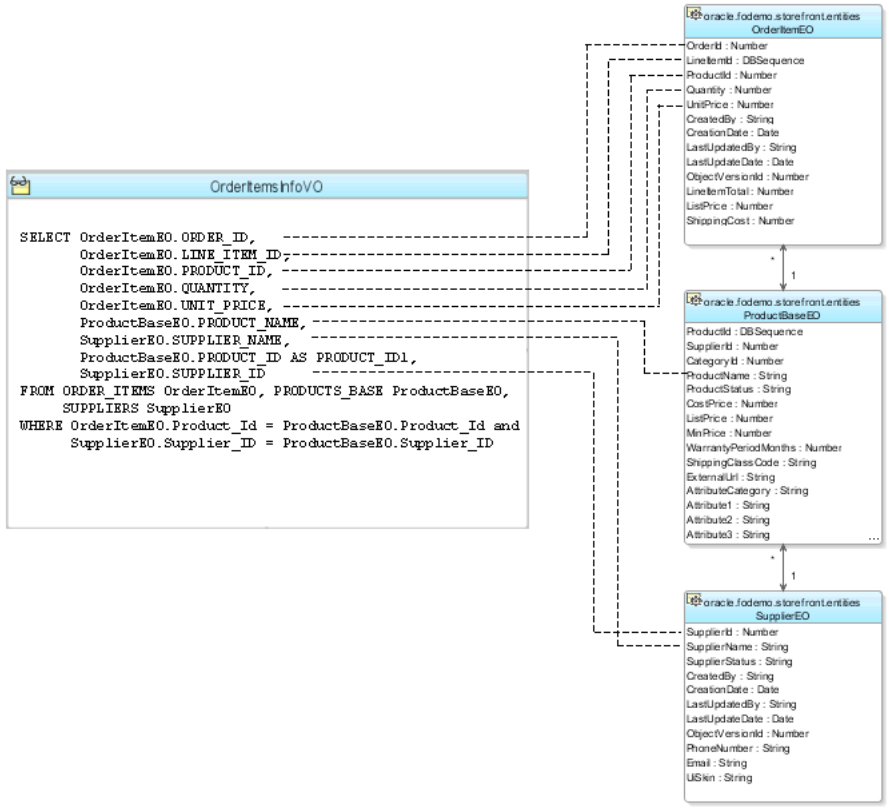
When you create a view object, JDeveloper creates the XML component definition file that represents the view object's declarative settings and saves it in the directory that corresponds to the name of its package. For example, the view object `Orders`, added to the `queries` package, will have the XML file `./queries/Orders.xml` created in the project's source path.

To view the view object settings, expand the desired view object in the Application Navigator, select the XML file under the expanded view object, and open the Structure window. The Structure window displays the list of definitions, including the SQL query, the name of the entity usage, and the properties of each attribute. To open the file in the editor, double-click the corresponding **.xml** node.

> **Note:** If your IDE-level Business Components Java generation preferences so indicate, the wizard may also create an optional custom view object class `OrdersImpl.java` and/or a custom view row class `OrdersRowImpl.java` class.

Figure 5–7 depicts the entity-based view object `OrderItemsInfoVO` and the three entity usages referenced in its query statement. The dotted lines represent the metadata captured in the entity-based view object's XML component definition that map `SELECT` list columns in the query to attributes of the entity objects used in the view object. The query of the entity-based view object joins data from a primary entity usage (`OrderItemEO`) with that from secondary reference entity usages (`ProductBaseEO` and `SupplierEO`).

Figure 5–7 View Object Encapsulates a SQL Query and Entity Attribute Mapping Metadata



5.2.3 How to Create an Expert Mode, Read-Only View Object

When you need full control over the SQL statement, the Create View Object wizard lets you specify that you want a view object to be read-only. In this case, you will not benefit from the declarative capabilities to define a non-updatable entity-based view object. However, there are a few situations where it is desirable to create read-only view objects using expert mode. Primarily, the read-only view object that you create will be useful when you need to write Unions or Group By queries. Additionally, you can use a read-only view object if you need to create SQL-based validation queries used by the view object-based Key Exists validator, provided that you have marked a key attribute.

> **Best Practice:** Unlike entity-based view objects, read-only view objects that you create in expert mode, will not define a key attribute by default. While it is possible to create a read-only view object without defining its key attribute, in expert mode it is a best practice to select the attribute that corresponds to the queried table's primary key and mark it as the key attribute. The presence of a key attribute ensures the correct runtime behavior for row set navigation. For example, the user interface developer may create an LOV component based on the read-only view object collection. Without a key attribute to specify the row key value, the LOV may not behave properly and a runtime error can result.

For more information about the tradeoffs between working with entity-based view objects that you define as non-updatable and strictly read-only view objects, see Section 42.2.2, "Consider Using Entity-Based View Objects for Read-Only Data."

To create a read-only view object, use the Create View Object wizard, which is available from the New Gallery.

Before you begin:

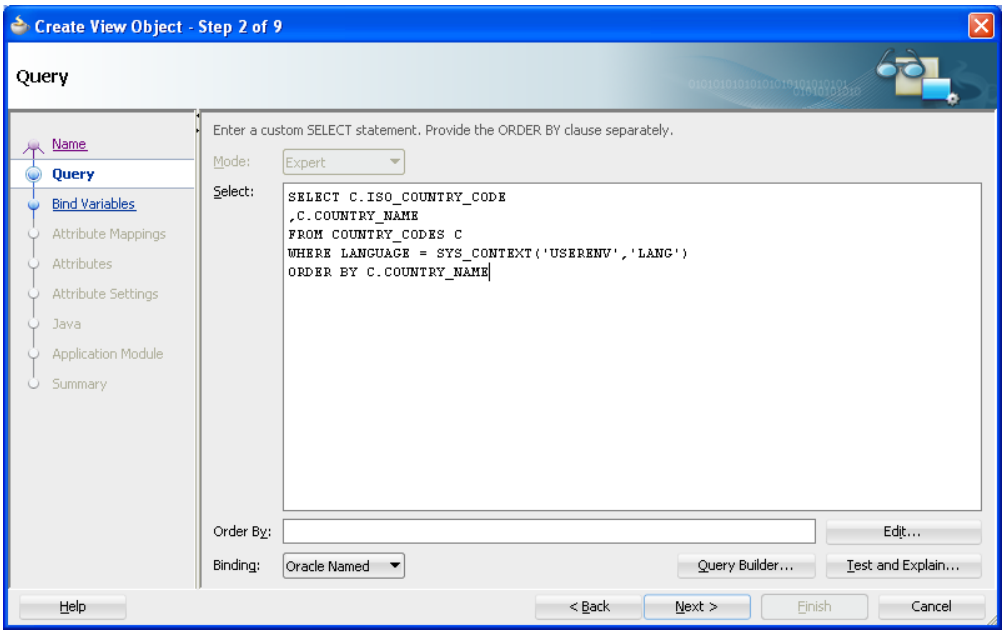
It may be helpful to have an understanding of view objects. For more information, see Section 5.2, "Populating View Object Rows from a Single Database Table."

To create a read-only view object:

1. In the Application Navigator, right-click the project in which you want to create the view object and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **View Object**, and click **OK**.

   If this is the first component you're creating in the project, the Initialize Business Components Project dialog appears to allow you to select a database connection.
3. In the Initialize Business Components Project dialog, select the database connection or choose **New** to create a connection. Click **OK**.
4. In the Create View Object wizard, on the Name page, enter a package name and a view object name. Select **SQL query** to indicate that you want this view object to manage data with read-only access. Click **Next**.
5. On the Query page, use one of the following techniques:
   - Paste any valid SQL statement into the **Query Statement** box. The query statement can use a `WHERE` clause and an `Order By` clause. For example, Figure 5–8 shows a query statement that uses a `WHERE` clause and an `Order By` clause to query a list of country codes in the language used by the application.
   - Click **Query Builder** to open the SQL Statement dialog and use the interactive query builder.

Figure 5–8 Create View Object Wizard, Query Page

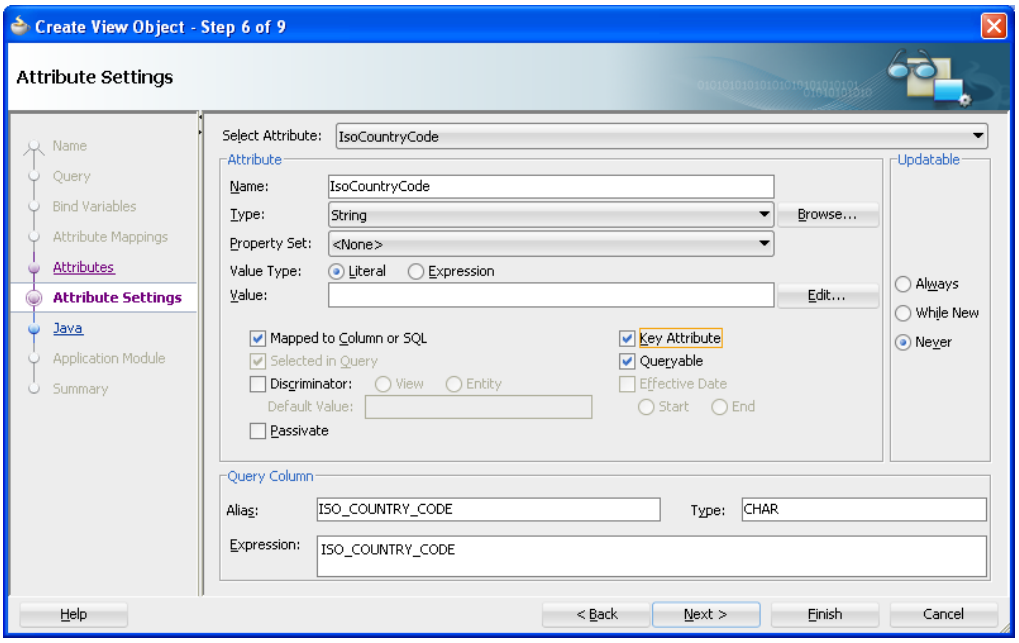


Note: If the Entity Objects page displays instead of the Query page, go back to Step 1 of the wizard and ensure that you've selected **Read-only Access**.

6. After entering or building the query statement, click **Next**.
7. On the Bind Variables page, do one of the following:
   - If the query does not reference any bind variables, click **Next** to skip Step 6.
   - To add a bind variable and work with it in the query, see Section 5.10.1, "How to Add Bind Variables to a View Object Definition."
8. On the Attribute Settings page, from the **Select Attribute** dropdown, select the attribute that corresponds to the primary key of the queried table and then enable the **Key Attribute** checkbox.

   Because the read-only view object is not based on an entity object, the Create View Object wizard does not define a key attribute by default. Failure to define the key attribute can result in unexpected runtime behavior for ADF Faces components with a data control based on the read-only view object collection. In the case of read-only view objects, define the key attribute, as shown in Figure 5–9.

Figure 5–9 Create View Object Wizard, Attribute Settings Page



9. On the Attribute Mappings page, click **Finish**.

Note: In the ADF Business Components wizards and editors, the default convention is to use camel-capped attribute names, beginning with a capital letter and using uppercase letters in the middle of the name to improve readability when the name comprises multiple words.

5.2.4 What Happens When You Create a Read-Only View Object

When you create a view object, JDeveloper first parses the query to infer the following from the columns in the SELECT list:

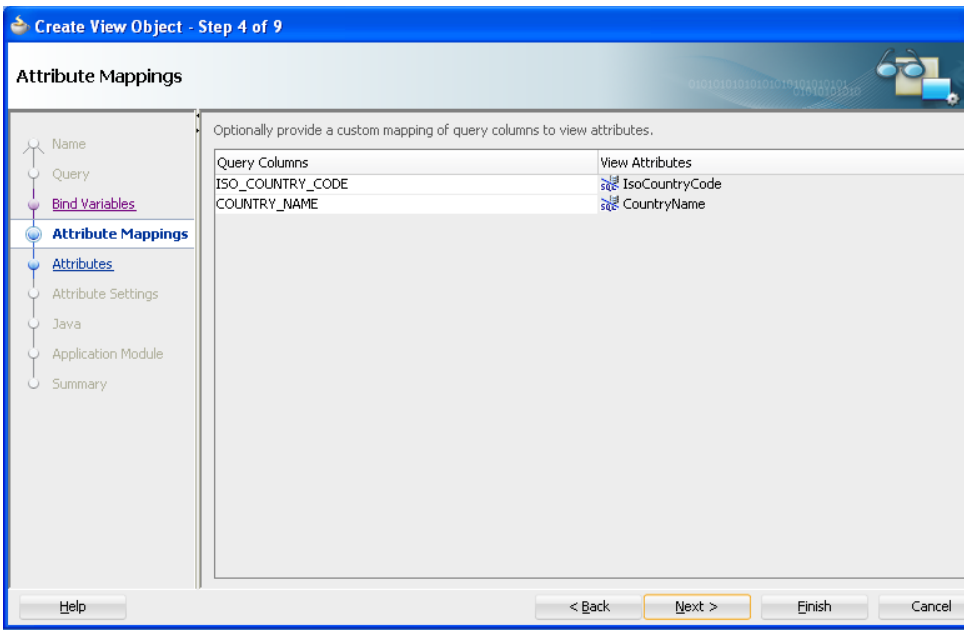
- The Java-friendly view attribute names (for example, `CountryName` instead of `COUNTRY_NAME`)

  By default, the wizard creates Java-friendly view object attribute names that correspond to the `SELECT` list column names, as shown in Figure 5–10.

  For information about using view object attribute names to access the data from any row in the view object's result set by name, see Section 6.4, "Testing View Object Instances Programmatically."

- The SQL and Java data types of each attribute

Figure 5–10 Create View Object Wizard, Attribute Mappings Page



Each part of an underscore-separated column name like *`SOME_COLUMN_NAME`* is turned into a camel-capped word (like *`SomeColumnName`*) in the attribute name. While the view object attribute names correspond to the underlying query columns in the `SELECT` list, the attribute names at the view object level need not match necessarily.

> **Tip:** You can rename the view object attributes to any names that might be more appropriate without changing the underlying query.

JDeveloper then creates the XML component definition file that represents the view object's declarative settings and saves it in the directory that corresponds to the name of its package. For example, the XML file created for a view object named `CountriesVO` in the `lookups` package is `./lookups/CountriesVO.xml` under the project's source path.

To view the view object settings, expand the desired view object in the Application Navigator, select the XML file under the expanded view object, and open the Structure window. The Structure window displays the list of definitions, including the SQL

query, the name of the entity usage, and the properties of each attribute. To open the file in the editor, double-click the corresponding **.xml** node.

> **Note:** If your IDE-level Business Components Java generation preferences so indicate, the wizard may also create an optional custom view object class `CountriesVOImpl.java` and/or a custom view row class `CountriesVORowImpl.java` class.

5.2.5 How to Edit a View Object

After you've created a view object, you can edit any of its settings in the overview editor for the view object.

> **Performance Tip:** How you configure the view object to fetch data plays a large role in the runtime performance of the view object. For information about the tuning parameters that you can edit to optimize performance, see Section 6.3.10, "What You May Need to Know About Optimizing View Object Runtime Performance."

Before you begin:

It may be helpful to have an understanding of view objects. For more information, see Section 5.2, "Populating View Object Rows from a Single Database Table."

To edit a view object definition:

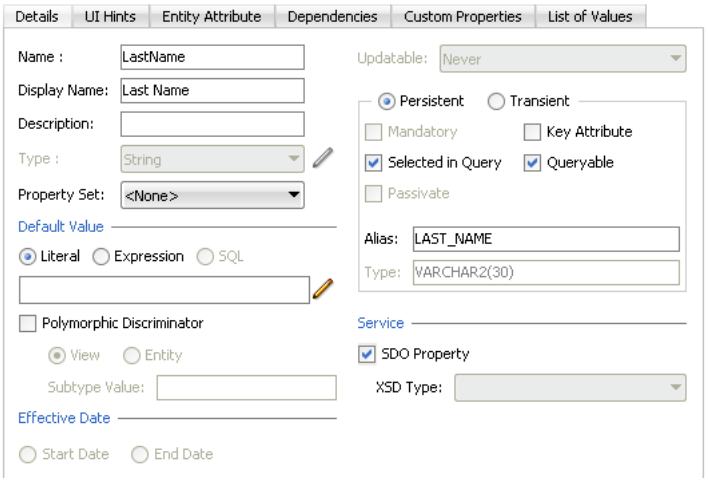
1. In the Application Navigator, double-click the view object to open the overview editor.
2. Click a navigation tab to open an editor page where you can adjust the SQL query, change the attribute names, add named bind variables, add UI controls hints, control Java generation options, and edit other settings.

5.2.5.1 Overriding the Inherit Properties from Underlying Entity Object Attributes

One interesting aspect of entity-based view objects is that each attribute that relates to an underlying entity object attribute inherits that attribute's properties. Figure 5–11 shows the Details section of the view object editor's Attributes page with an inherited attribute selected. You can see that fields like the Java attribute type and the query column type are disabled and their values are inherited from the related attribute of the underlying entity object to which this view object is related. Some properties like the attribute's data type are inherited and cannot be changed at the view object level.

Other properties like `Queryable` and `Updatable` are inherited but can be overridden as long as their overridden settings are more restrictive than the inherited settings. For example, the attribute from underlying entity object might have an **Updatable** setting of **Always**. As shown Figure 5–11, the Details section of the Attributes page of the view object overview editor allows you to set the corresponding view object attribute to a more restrictive setting like **While New** or **Never**. However, if the attribute in the underlying entity object had instead an **Updatable** setting of **Never**, then the editor would not allow the view object's related attribute to have a less restrictive setting like **Always**.

Figure 5–11 View Object Attribute Properties Inherited from Underlying Entity Object



5.2.5.2 Controlling the Length, Precision, and Scale of View Object Attributes

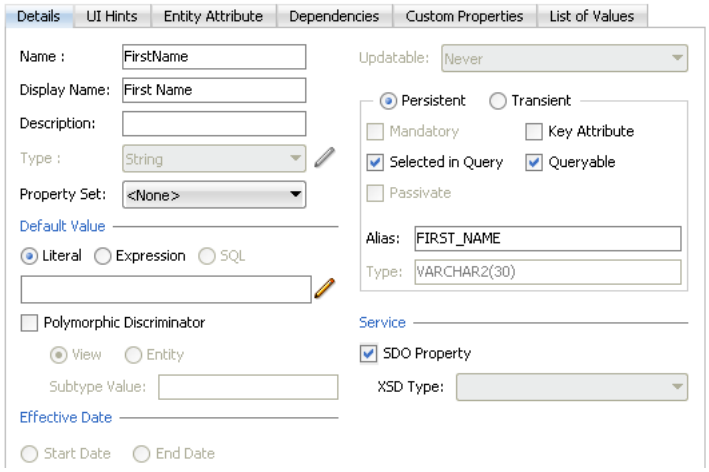
When you display a particular attribute of the view object in the overview editor, you can see and change the values of the declarative settings that control its runtime behavior. One important property is the **Type** in the **Query Column** section, shown in Figure 5–11. This property records the SQL type of the column, including the length information for `VARCHAR2` columns and the precision and scale information for `NUMBER` columns.

JDeveloper tries to infer the type of the column automatically, but for some SQL expressions the inferred value might default to `VARCHAR2(255)`. You can update the **Type** value for this type of attribute to reflect the correct length if you know it. In the case of read-only view objects, this property is editable in the Details section that you display in the Attributes page of the overview editor for the view object. In the case of entity-based view objects, you must edit the **Type** property in the Details section of the Attributes page of the overview editor that you display for the entity object, as described in Section 4.10.2, "How to Indicate Data Type Length, Precision, and Scale."

For example, `VARCHAR2(30)` which shows as the **Type** for the `FirstName` attribute in Figure 5–12 means that it has a maximum length of 30 characters. For a `NUMBER` column, you would indicate a **Type** of `NUMBER(7,2)` for an attribute that you want to have a precision of 7 digits and a scale of 2 digits after the decimal.

> **Performance Tip:** Your SQL expression can control how long the describe from the database says the column is. Use the `SUBSTR()` function around the existing expression. For example, if you specify `SUBSTR(`*`yourexpression`*`, 1, 15)`, then the describe from the database will inform JDeveloper that the column has a maximum length of 15 characters.

Figure 5–12 Custom Attribute Settings in the Attribute Details Section



5.2.5.3 Converting a Read-Only View Object to Allow Attribute Updates

When you use the Create View Object wizard to create a read-only view object, by default the attributes of the view object will not be updateable. Later you may decide to convert the view object to one that permits updates to its SQL-mapped table columns. However, this cannot be accomplished by merely changing the attribute's **Updateable** property. To convert a read-only view object to one that is updateable, you must add an entity usage that maps to the same table as the one used to create the read-only view object. Choosing an entity usage that defines the same table ensures that you can then remap the SQL-derived view attributes to entity usage attributes corresponding to the same table columns.

Before you begin:

It may be helpful to have an understanding of view objects. For more information, see Section 5.2, "Populating View Object Rows from a Single Database Table."

You may also find it helpful to have an understanding of entity-based view objects. For more information, see Section 5.2.1, "How to Create an Entity-Based View Object."

You will need to complete this task:

Create the desired entity objects as described in Section 4.2.1, "How to Create Multiple Entity Objects and Associations from Existing Tables."

To modify a read-only view object to allow updates:

1. In the Application Navigator, double-click the read-only view object.
2. In the overview editor, click the **Entity Objects** navigation tab.
3. In the Entity Objects page, expand the **Available** list and double-click the entity object that describes the attributes of the read-only view object.

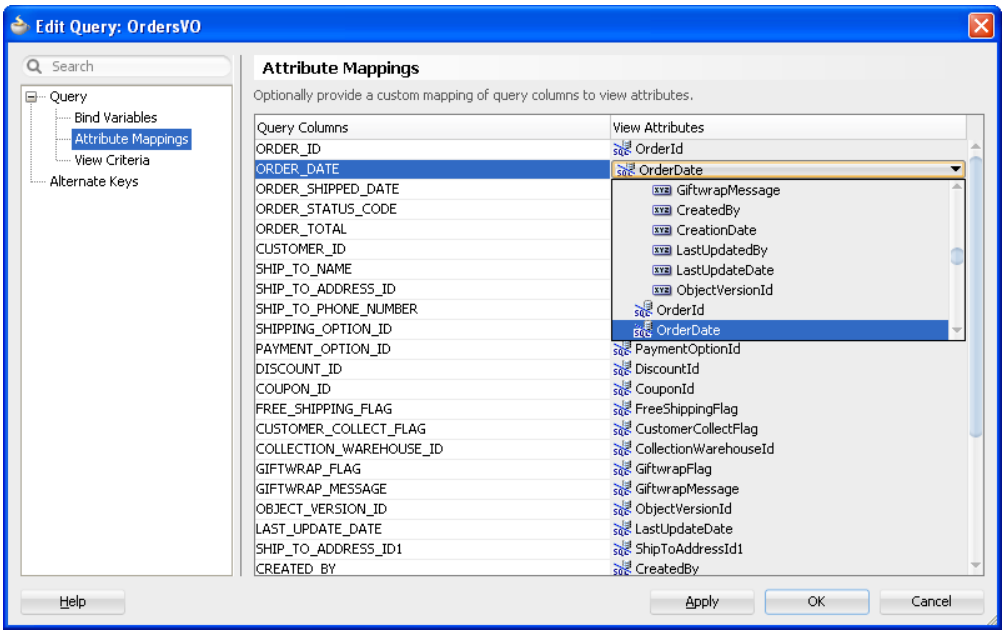
   The entity object that you double-click will appear in the **Selected** list as an entity usage. You will need to remap the SQL-derived attributes to corresponding attributes defined by the entity usage.
4. Click the **Query** navigation tab, and in the Query page, click the **Edit SQL Query** button.
5. In the Edit Query dialog, click **Query** and then click **Attribute Mappings**.

6. In the Attribute Mappings page, perform the following steps to convert all SQL-derived attributes to their corresponding entity usage mapped attribute.

   a. Click an attribute field in the **View Attributes** column and scroll to the top of the dropdown list to locate the entity usage attributes.

   b. In the entity usage attribute list, select the attribute corresponding to the read-only attribute that you want to remap, as shown in Figure 5–11.

Figure 5–13 Specifying an Entity-Derived Attribute in the Edit Query Dialog



7. Click **OK**.

5.2.5.4 Customizing View Object Attribute Display in the Overview Editor

When you edit view objects in the overview editor, you can customize the Attributes page of the overview editor to make better use of the attributes table displayed for the view object.

Customization choices that you make for the attributes table include the list of attribute properties to display as columns in the attributes table, the order that the columns appear (from left to right) in the attributes table, the sorting order of the columns, and the width of the columns. The full list of columns that you can choose to display correspond to the properties that you can edit for the attribute.

For example, you can add the **Updatable** property as a column to display in the attributes table when you want to quickly determine which attributes of your view object are updatable. Or, you can add the attributes' **Label** property as a column and see the same description as the end user. Or, you might want to view the list of attributes based on their entity usages. In this case, you can display the **Entity Usage** column and sort the entire attributes table on this column.

When you have set up the attributes table with the list of columns that you find most useful, you can apply the same set of columns to the attributes table displayed for other view objects by right-clicking the attributes table and choose **Apply to All View Objects**.

Before you begin:

It may be helpful to have an understanding of view objects. For more information, see Section 5.2, "Populating View Object Rows from a Single Database Table."

To customize the attributes table display:

1. In the Application Navigator, double-click the view object.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, click the dropdown menu to the right of the attribute column headers (just below the attributes table's button bar) and choose **Select Columns**.
4. In the Select Columns dialog, perform any of the following actions.
   a. Click the left/right shuttle buttons to change the list of visible columns in the attributes table of the overview editor. The overview editor displays only those columns corresponding to the attribute properties that appear the **Selected** list.
   b. Click one of the **Move Selection** buttons to change the position of the columns in the attributes table of the overview editor. The overview editor displays the attribute properties arranged from left to right starting with the property at the top of the **Selected** list.
5. Click **OK**.
6. On the Attributes page of the overview editor, perform any of the following actions.
   a. Click any column header to sort all columns in the attributes table by the selected column.

      This feature is particularly useful when you want to focus on a particular column. For example, in the case of an entity-based view object, you can click the **Entity Usage** column header to group attributes in the attributes table by their underlying entity objects. To save this setting across all view objects that you display in the overview editor, click the dropdown menu to the right of the column headers and choose **Apply to All View Objects**.
   b. Click any column header border and drag to adjust the width of the attributes table's column.
   c. Click the dropdown list to the right of the column headers and select among the list of displayed columns to change the visibility of a column in the current attributes table display.

      This feature lets you easily hide columns when you want to simplify the attributes table display in the current view object overview editor.
7. To extend the changes in the columns (including column list, column order, column sorting, and column width) to all other view object overview editors, click the dropdown menu to the right of the column headers and choose **Apply to All View Objects**.

   This feature allows you to easily compare the same attributes across view objects. The overview editor will apply the column selections (and order) that you make in the Select Columns dialog and the current attributes table's column sorting and column widths to all view objects that you edit. View objects that are currently displayed in an open overview editor are not updated with these settings; you must close the open view object overview editor and then reopen the view object to see these settings applied.

5.2.5.5 Modifying the Order of Attributes in the View Object Source File

After you create a view object definition, you may decide to change the order of the attributes queried by the view object. This view object editing feature allows you to easily change the order that the attributes will appear in the attributes table displayed on the Attributes page of the view object overview editor. Because this feature acts on specific attributes and alters the XML definition of the current view object, it does not apply to other view objects that you may edit. Alternatively, you can sort the display of attributes on the Attribute page of the view object overview editor without affecting the source file by clicking any column header in the overview editor's attributes table.

Before you begin:

It may be helpful to have an understanding of view objects. For more information, see Section 5.2, "Populating View Object Rows from a Single Database Table."

To modify the order of attributes in the view object source file:

1. In the Application Navigator, double-click the view object.
2. In the overview editor, click the **Attributes** navigation tab and click **Set Source Order**.
3. In the Set Source Order dialog, select the attribute you want to reposition and click one of the **Move Selection** button.
4. Click **OK**.

   This feature has no affect on other view objects that you may edit; it only affects the current view object.

5.2.6 How to Show View Objects in a Business Components Diagram

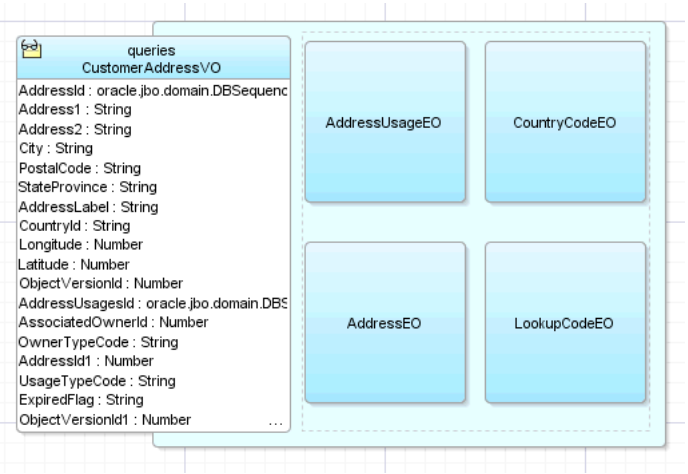
JDeveloper's UML diagramming lets you create a Business Components diagram to visualize your business services layer. In addition to supporting entity objects, JDeveloper's UML diagramming allows you to drop view objects onto diagrams as well to visualize their structure and entity usages. For example, if you create a new Business Components Diagram named `StoreFrontService Data Model` in the `oracle.fodemo.storefront` package, and drag the `CustomerAddressVO` view object from the Application Navigator onto the diagram, its entity usages would display, as shown in Figure 5–14. When viewed as an expanded node, the diagram shows a compartment containing the view objects entity usages.

Before you begin:

It may be helpful to have an understanding of view objects. For more information, see Section 5.2, "Populating View Object Rows from a Single Database Table."

You may also find it helpful to understand how to the diagram, see Section 4.4, "Creating an Entity Diagram for Your Business Layer."

Figure 5–14 View Object and Its Entity Usages in a Business Components Diagram



5.3 Populating View Object Rows with Static Data

ADF Business Components lets you create view objects in your data model project with rows that you populate at design time. Typically, you create view objects with static data when you have a small amount of data to maintain and you do not expect that data to change frequently. The decision whether to use a lookup table from the database or whether to use a static view object based on a list of hardcoded values depends on the size and nature of the data. The static view object is useful when you have no more than 100 entries to list. Any larger number of rows should be read from the database with a conventional table-based view object. The static view object has the advantage of being easily translatable. However, all of the rows of a static view object will be retrieved at once and therefore, using no more than 100 entries yields the best performance.

> **Best Practice:** When you need to create a view object to access a small list of static data, you should use the static view object rather than query the database. The static view object is ideal for lists not exceeding 100 rows of data. Because the Create View Object wizard saves the data in a resource message file, these data are easily translatable.

Static list view objects are useful as an LOV data source when it is not desirable to query the database to supply the list of values. Suppose your order has the following statuses: open, closed, pending. You can create a static view object with these values and define an LOV on the static view object's status attribute. Because the wizard stores the values of the status view object in a translatable resource file, the UI will display the status values using the resource file corresponding to the application's current locale.

5.3.1 How to Create Static View Objects with Data You Enter

You use the Create View Object wizard to create static view objects. The wizard lets you define the desired attributes (columns) and enter as many rows of data as necessary. The wizard displays the static data table as you create it.

Note: Because the data in a static view object does not originate in database tables, the view object will be read-only.

You can also use the Create View Object wizard to create the attributes based on data from a comma-separated value (CSV) file format like a spreadsheet file. The wizard will attempt to create the attributes that you define in the wizard with data from the first row of the flat file.

Before you begin:

It may be helpful to have an understanding of static view objects. For more information, see Section 5.3, "Populating View Object Rows with Static Data."

To manually create attributes for a static view object:

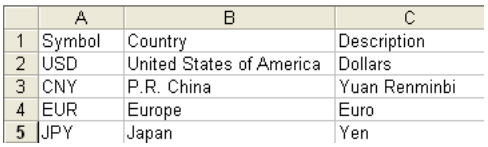
1. In the Application Navigator, right-click the project in which you want to create the static list view object and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **View Object**, and click **OK.**

   If this is the first component you're creating in the project, the Initialize Business Components Project dialog appears to allow you to select a database connection.
3. In the Initialize Business Components Project dialog, select the database connection or choose **New** to create a connection. Click **OK**.
4. In the Create View Object wizard, on the Name page, enter a package name and a view object name. Select **Static list** to indicate that you want to supply static list data for this view object. Click **Next**.
5. On the Attributes page, click **New** to add an attribute that corresponds to the columns in the static data table. In the New View Object Attribute dialog, enter a name and select the attribute type. Click **OK** to return to the wizard, and click **Next**.
6. On the Attribute Settings page, do nothing and click **Next**.
7. On the Static List page, click the **Add** icon to enter the data directly into the wizard page. The attributes you defined will appear as the columns for the static data table.
8. On the Application Module pages, do nothing and click **Next**.
9. On the Summary page, click **Finish**.

5.3.2 How to Create Static View Objects with Data You Import

Using the Import feature of the Create View Object wizard, you can create a static data view object with attributes based on data from a comma-separated value (CSV) file format like a spreadsheet file. The wizard will use the first row of a CSV flat file to identify the attributes and will use the subsequent rows of the CSV file for the data for each attribute. For example, if your application needs to display choices for international currency, you might define the columns Symbol, Country, and Description in the first row and then add rows to define the data for each currency type, as shown in Figure 5–15.

Figure 5–15 Sample Data Ready to Import from CSV Flat File



	A	B	C
1	Symbol	Country	Description
2	USD	United States of America	Dollars
3	CNY	P.R. China	Yuan Renminbi
4	EUR	Europe	Euro
5	JPY	Japan	Yen

Before you begin:

It may be helpful to have an understanding of static view objects. For more information, see Section 5.3, "Populating View Object Rows with Static Data."

To create attributes of a static view object based on a flat file:

1. In the Application Navigator, right-click the project in which you want to create the static list view object and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **View Object**, and click **OK**.

   If this is the first component you're creating in the project, the Initialize Business Components Project dialog appears to allow you to select a database connection.
3. In the Initialize Business Components Project dialog, select the database connection or choose **New** to create a connection. Click **OK**.
4. In the Create View Object wizard, on the Name page, enter a package name and a view object name. Select **Static list** to indicate that you want to supply static list data for this view object. Click **Next**.
5. On the Attributes page, optionally, click **New** to add an attribute that corresponds to the columns in the static data table. In the New View Object Attribute dialog, enter a name and select the attribute type. Click **OK** to return to the wizard, and click **Next**.

   When the static data will be loaded from a CSV flat file, you can optionally skip this step. If you do not create the attributes yourself, the wizard will attempt to use the first row of the CSV file to create the attributes. However, if you create the attributes in the wizard, then the attributes you create must match the order of the columns defined by the flat file. If you have created fewer attributes than columns, the wizard will ignore extra columns during import. Conversely, if you create more attributes than columns, the wizard will define extra attributes with the value `NULL`.
6. On the Attribute Settings page, do nothing and click **Next**.
7. On the Static List page, click **Import** to locate the CSV file and display the data in the wizard. Verify the data and edit the values as needed.

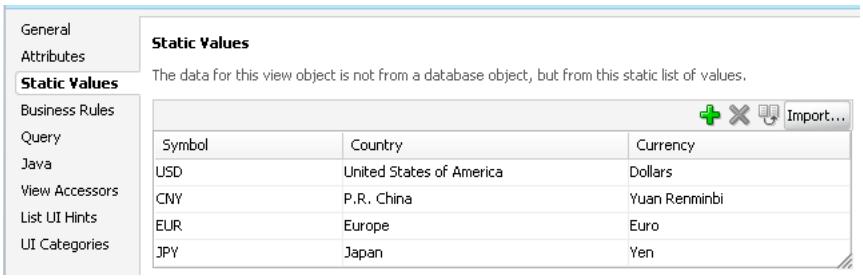
   To edit an attribute value, double-click in the value field.
8. Optionally, click the **Add** icon or **Remove** icon to change the number of rows of data. Click **Next**.

   To enter values for the attributes of a new row, double-click in the value field.
9. On the Application Module page, do nothing and click **Next**.
10. On the Summary page, click **Finish**.

5.3.3 What Happens When You Create a Static List View Object

When you create a static view object, the overview editor for the view object displays the rows of data that you defined in the wizard. You can use the editor to define additional data, as shown in Figure 5–16.

Figure 5–16 Static Values Page Displays Data



The generated XML definition for the static view object contains one transient attribute for each column of data. For example, if you import a CSV file with data that describes international currency, your static view object might contain a transient attribute for `Symbol`, `Country`, and `Description`, as shown in Example 5–1.

Example 5–1 XML Definition for Static View Object

```
<ViewObject
...
// Transient attribute for first column
  <ViewAttribute
    Name="Symbol"
    IsUpdateable="false"
    IsSelected="false"
    IsPersistent="false"
    PrecisionRule="true"
    Precision="255"
    Type="java.lang.String"
    ColumnType="VARCHAR2"
    AliasName="Symbol"
    SQLType="VARCHAR"/>
// Transient attribute for second column
  <ViewAttribute
    Name="Country"
    IsUpdateable="false"
    IsPersistent="false"
    PrecisionRule="true"
    Precision="255"
    Type="java.lang.String"
    ColumnType="VARCHAR"
    AliasName="Country"
    SQLType="VARCHAR"/>
// Transient attribute for third column
  <ViewAttribute
    Name="Description"
    IsUpdateable="false"
    IsPersistent="false"
    PrecisionRule="true"
    Precision="255"
    Type="java.lang.String"
    ColumnType="VARCHAR"
    AliasName="Description"
```

```
    SQLType="VARCHAR"/>
  <StaticList
    Rows="4"
    Columns="3"/>
// Reference to file that contains static data
  <ResourceBundle>
    <PropertiesBundle
      PropertiesFile="model.ModelBundle"/>
  </ResourceBundle>
</ViewObject>
```

Because the data is static, the overview editor displays no Query page and the generated XML definition for the static view object contains no query statement. Instead, the `<ResourceBundle>` element in the XML definition references a resource bundle file. The resource bundle file describes the rows of data and also lets you localize the data. When the default resource bundle type is used, the file *ModelName*`Bundle.properties` appears in the data model project, as shown in Example 5–2.

Example 5–2 Default Resource Bundle File for Static View Object

```
model.ViewObj.SL_0_0=USD
model.ViewObj.SL_0_1=United States of America
model.ViewObj.SL_0_2=Dollars
model.ViewObj.SL_1_0=CNY
model.ViewObj.SL_1_1=P.R. China
model.ViewObj.SL_1_2=Yuan Renminbi
model.ViewObj.SL_2_0=EUR
model.ViewObj.SL_2_1=Europe
model.ViewObj.SL_2_2=Euro
model.ViewObj.SL_3_0=JPY
model.ViewObj.SL_3_1=Japan
model.ViewObj.SL_3_2=Yen
```

5.3.4 How to Edit Static List View Objects

When you need to make changes to the static list table, double-click the view object in the Application Navigator to open the overview editor for the view object. You can add and delete attributes (columns in the static list table), add or delete rows (data in the static list table), sort individual rows, and modify individual attribute values. The editor will update the view object definition file and save the attribute names in the message bundle file.

5.3.5 What You May Need to Know About Static List View Objects

The static list view object has a limited purpose in the application module's data model. Unlike entity-based view objects, static list view objects will not be updatable. You use the static list view object when you want to display read-only data to the end user and you do not want to create a database table for the small amount of data the static list table contains.

5.4 Limiting View Object Rows Using Effective Date Ranges

Applications that need to query data over a specific date range can generate *date-effective row sets*. To define an date-effective view object you must create an entity-based view object that is based on an date-effective entity object. User control over the view object's effective date usage is supported by metadata on the view object

at design time. At runtime, ADF Business Components generates the query filter that will limit the view rows to an effective date.

5.4.1 How to Create an Date-Effective View Object

Whether or not the query filter for an effective date will be generated depends on the value of the **Effective Dated** property displayed in the Property Inspector for the view object (to view the property, click the **General** tab in the overview editor for the view object and expand the **Name** category in the Property Inspector).

> **Note:** Because the date-effective view object must be based on an date-effective entity object, setting a view object's **Effective Dated** property to **True** without an underlying date-effective entity object, will result in a runtime exception.

The overview editor for the view object does not display the date-effective query clause in the `WHERE` clause. You can use the Explain Plan dialog or Test Query dialog to view the clause. A typical query filter for effective dates looks like this:

```
(:Bind_SysEffectiveDate BETWEEN Person.EFFECTIVE_START_DATE AND
Person.EFFECTIVE_END_DATE)
```

At runtime, the bind value for the query is obtained from a property of the root application module. In order to set the effective date for a transaction, use code similar to the following snippet:

```
am.setProperty(ApplicationModule.EFF_DT_PROPERTY_STR, new
Date("2008-10-01));
```

If you do not set `EFF_DT_PROPERTY_STR` on the application module, the current date is used in the query filter, and the view object returns the effective rows filtered by the current date.

The view object has its own transient attribute, `SysEffectiveDate`, that you can use to set the effective date for view rows. Otherwise, the `SysEffectiveDate` attribute value for new rows and defaulted rows is derived from the application module. ADF Business Components propagates the effective date from the view row to the entity object during DML operations only.

Before you begin:

It may be helpful to have an understanding of data-effective row sets. For more information, see Section 5.4, "Limiting View Object Rows Using Effective Date Ranges."

You will need to complete these tasks:

1. Create an effective dated entity object as described in Section 4.2.8, "How to Store Data Pertaining to a Specific Point in Time."
2. Use the Create View Object wizard to create the entity-based view object as described in Section 5.2.1, "How to Create an Entity-Based View Object."

   The view object you create should be based on the effective dated entity object you created. In the Attributes page of the wizard, be sure to add the date-effective attributes that specify the start date and end date on the entity object to the **Selected** list for the view object.

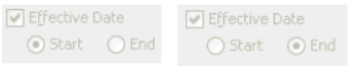
To enable effective dates for a view object using the SysEffectiveDate attribute:

1. In the Application Navigator, double-click the view object you created based on the effective dated entity object.
2. In the overview editor, click the **General** navigation tab.
3. In the Property Inspector, expand the **Name** category.

   If the **Name** category is not displayed in the Property Inspector, click the **General** navigation tab in the overview editor to set the proper focus.
4. In the Name category, verify that the context menu for the **Effective Dated** property displays **True**.
5. In the overview editor, click the **Attributes** navigation tab and select the attribute for the start date, and then click the **Details** tab and verify that **Effective Date** is enabled and that **Start** is selected, as shown in Figure 5–17.

   Verify that the attribute for the end date is also enabled correctly, as shown in the figure. Note that these fields appear grayed out to indicate that they cannot be edited for the view object.

Figure 5–17 View Object Overview Editor Displays Effective Date Settings



No additional steps are required once you have confirmed that the view object has inherited the desired attributes from the date-effective entity object.

5.4.2 How to Create New View Rows Using Date-Effective View Objects

Creating (inserting) date-effective rows is similar to creating or inserting ordinary view rows. The start date and end date can be specified as follows:

- The user specifies the effective date on the application module. The start date is set to the effective date, and the end date is set to end of time.
- The user specifies values for the start date and the end date (advanced).

In either case, during entity validation, the new row is checked to ensure that it does not introduce any gaps or overlaps. During post time, ADF Business Components will acquire a lock on the previous row to ensure that the gap or overlaps are not created upon the row insert.

5.4.3 How to Update Date-Effective View Rows

You can update view rows by using API calls like `Row.setAttribute()`. ADF Business Components supports various modes to initiate the row update.

To set the update mode, invoke the `Row.setEffectiveDateMode(int mode)` method with one of the following mode constants.

- `CORRECTION` (Correction Mode)

  The effective start date and effective end dates remain unchanged. The values of the other attributes may change. This is the standard row update behavior.
- `UPDATE` (Update Mode)

  The effective end date of the row will be set to the effective date. All user modifications to the row values are reverted on this row. A new row with the

modified values is created. The effective start date of the new row is set to the effective date plus one day, and the effective end date is set to end of time. The new row will appear after the transaction is posted to the database.

- `UPDATE_OVERRIDE` (Update Override Mode)

  The effective end date of the modified row will be set to the effective date. The effective start date of the next row is set to effective date plus one day.

- `UPDATE_CHANGE_INSERT` (Change Insert Mode)

  The effective end date of the modified row should be set to the effective date. All user modifications to the row values are reverted on this row. A new row with the modified values will be created. The effective start date of the new row is set to effective date plus one day, and the effective end date is set to effective start date of the next row minus one day. The new row will appear after the transaction is posted to the database.

5.4.4 How to Delete Date-Effective View Rows

ADF Business Components supports various modes to initiate the row deletion. You can mark view rows for deletion by using API calls like `RowSet.removeCurrentRow()` or `Row.remove()`.

To set the deletion mode, invoke the `Row.setEffectiveDateMode(int mode)` method with one of the following mode constants.

- `DELETE` (Delete Mode)

  The effective date of the row is set to the effective date. The operation for this row is changed from delete to update. All rows with the same noneffective date key values and with an effective start date greater than the effective date are deleted.

- `DELETE_NEXT_CHANGE` (Delete Next Change Mode)

  The effective end date of the row is set to the effective end date of the next row with the same noneffective date key values. The operation for this row is changed from delete to update. The next row is deleted.

- `FUTURE_CHANGE` (Delete Future Change Mode)

  The effective end date of the row is set to the end of time. The operation for this row is changed from delete to update. All future rows with the same noneffective date key values are deleted.

- `ZAP` (Zap Mode)

  All rows with the same non-effective date key values are deleted.

The effective date mode constants are defined on the row interface as well.

5.4.5 What Happens When You Create a Date-Effective View Object

When you create an date-effective view object, the view object inherits the transient attribute `SysEffectiveDate` from the entity object to store the effective date for the row. Typically, the insert/update/delete operations modify the transient attribute while Oracle ADF decides the appropriate values for effective start date and effective end date.

The query displayed in the overview editor for the date-effective view object does not display the `WHERE` clause needed to filter the effective date range. To view the full query for the date-effective view object, including the `WHERE` clause, edit the query

and click **Explain Plan** in the Edit Query dialog. The following sample shows a typical query and query filter for effective dates:

```
SELECT OrdersVO.ORDER_ID,
       OrdersVO.CREATION_DATE,
       OrdersVO.LAST_UPDATE_DATE
FROM ORDERS OrdersVO
WHERE (:Bind_SysEffectiveDate BETWEEN OrdersVO.CREATION_DATE AND
                                      OrdersVO.LAST_UPDATE_DATE)
```

Example 5–3 shows sample XML entries that are generated when you create an date-effective view object.

Example 5–3 XML Definition for Date-Effective View Object

```
<ViewObject
...
// Property that enables date-effective view object.
  IsEffectiveDated="true">
  <EntityUsage
    Name="Orders1"
    Entity="model.OrdersDatedEO"
    JoinType="INNER JOIN"/>
// Attribute identified as the start date
  <ViewAttribute
    Name="CreationDate"
    IsNotNull="true"
    PrecisionRule="true"
    IsEffectiveStartDate="true"
    EntityAttrName="CreationDate"
    EntityUsage="Orders1"
    AliasName="CREATION_DATE"/>
// Attribute identified as the end date
  <ViewAttribute
    Name="LastUpdateDate"
    IsNotNull="true"
    PrecisionRule="true"
    IsEffectiveEndDate="true"
    EntityAttrName="LastUpdateDate"
    EntityUsage="Orders1"
    AliasName="LAST_UPDATE_DATE"/>
// The SysEffectiveDate transient attribute
  <ViewAttribute
    Name="SysEffectiveDate"
    IsPersistent="false"
    PrecisionRule="true"
    Type="oracle.jbo.domain.Date"
    ColumnType="VARCHAR2"
    AliasName="SysEffectiveDate"
    Passivate="true"
    SQLType="DATE"/>
</ViewObject>
```

5.4.6 What You May Need to Know About Date-Effective View Objects and View LInks

Effective dated associations and view links allow queries to be generated that take the effective date into account. The effective date of the driving row is passed in as a bind parameter during the query execution.

While it is possible to create a noneffective dated association between two entities when using the Create Association wizard or Create View Link wizard, JDeveloper will by default make the association or link effective dated if one of the ends is effective dated. However, when the association or view link exists between an effective dated and a noneffective dated object, then at runtime ADF Business Components will inspect the effective dated nature of the view object or entity object before generating the query clause and binding the effective date. The effective date is first obtained from the driving row. If it is not available, then it is obtained from the property `EFF_DT_PROPERTY_STR` of the root application module. If you do not set `EFF_DT_PROPERTY_STR` for the application module, the current date is used in the query filter on the driving row and applied to the other side of the association or view link.

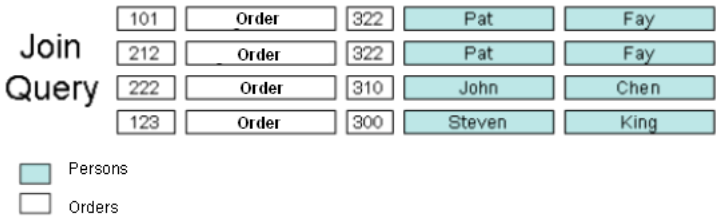
5.5 Working with Multiple Tables in Join Query Results

Many queries you will work with will involve multiple tables that are related by foreign keys. In this scenario, you join the tables in a single view object query to show additional descriptive information in each row of the main query result. You use the Create View Object wizard to define the query using declarative options. Whether your view object is read-only or entity-based determines how you can define the join:

- When you work with entity-based view objects, the Create View Object wizard uses an existing association defined between the entities to automatically build the view object's join `WHERE` clause. You can declaratively specify the type of join you want to result from the entity objects. Inner join (equijoin) and outer joins are both supported.
- When you work with read-only view objects, you will use the SQL Builder dialog to build the view object's join `WHERE` clause. In this case, you must select the columns from the tables that you want to join.

Figure 5–18 illustrates the rows resulting from two tables queried by a view object that defines a join query. The join is a single flattened result.

Figure 5–18 Join Query Result



5.5.1 How to Create Joins for Entity-Based View Objects

It is extremely common in business applications to supplement information from a *primary* business domain object with secondary reference information to help the end user understand what foreign key attributes represent. Take the example of the `OrderItems` entity object. It contains foreign key attribute of type `Number` like:

- `ProductId`, representing the product to which the order item pertains

From experience, you know that showing an end user exclusively these "raw" numerical values won't be very helpful. Ideally, reference information from the view object's related entity objects should be displayed to improve the application's usability. One typical solution involves performing a join query that retrieves the

combination of the primary and reference information. This is equivalent to populating "dummy" fields in each queried row with reference information based on extra queries against the lookup tables.

When the end user can *change* the foreign key values by editing the data, this presents an additional challenge. Luckily, entity-based view objects support easily including reference information that's always up to date. The key requirement to leverage this feature is the presence of associations between the entity object that act as the view object's primary entity usage and the entity objects that contribute reference information.

To include reference entities in a join view object, use the Create View Object wizard. The Create View Object wizard lets you specify the type of join:

- **Inner Join**

  Select when you want the view object to return all rows between two or more entity objects, where each entity defines the same primary key column. The inner join view object will not return rows when a primary key value is missing from the joined entities.

- **Outer Join**

  Select when you want the view object to return all rows that exist in one entity object, even though corresponding rows do not exist in the joined entity object. Both left and right outer join types are supported. The left and right designation refers to the source (left) and destination (right) entity object named in an association. For details about changing the default inner join to an outer join, see Section 5.5.5, "How to Modify a Default Join Clause to Be an Outer Join When Appropriate."

Both inner joins and outer joins are supported with the following options:

- **Reference**

  Select when you want the data from the entity object to be treated as reference information for the view object. Automatic lookup of the data is supported and attribute values will be dynamically fetched from the entity cache when a controlling key attribute changes. For details about how this setting affects runtime behavior, see Section 42.9.2, "What Happens at Runtime: View Row Creation."

- **Updatable**

  Deselect when you want to prevent the view object from modifying any entity attributes in the entity object. By default, the first entity object (primary) in the **Selected** list is updatable and subsequent entity objects (secondary) are not updatable. To understand how to create a join view object with *multiple* updatable entity usages, see Section 42.9, "Creating a View Object with Multiple Updatable Entities."

- **Participate in row delete**

  Select when you have defined the entity as updatable and you want the action of removing rows in the UI to delete the participating reference entity object. This option is disabled for the primary entity. For example, while it may be possible to delete an order item, it should not be possible to delete the order when a remove row is called from the join view object.

Before you begin:

It may be helpful to have an understanding of how the type of view object effects joins. For more information, see Section 5.5, "Working with Multiple Tables in Join Query Results."

You will need to complete this task:

Create the desired entity objects as described in Section 4.2.1, "How to Create Multiple Entity Objects and Associations from Existing Tables."

To create a view object that joins entity objects:

1. In the Application Navigator, right-click the project in which you want to create the view object and choose **New**.

   When you want to modify an existing view object that you created to include reference information from its related entity objects, double-click the view object and open the Entity Objects page in the overview editor for the view object.

2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **View Object**, and click **OK**.

3. In the Create View Object wizard, on the Name page, enter a package name and a view object name. Keep the default setting **Entity object** enabled to indicate that you want this view object to manage data with its base entity object. Click **Next**.

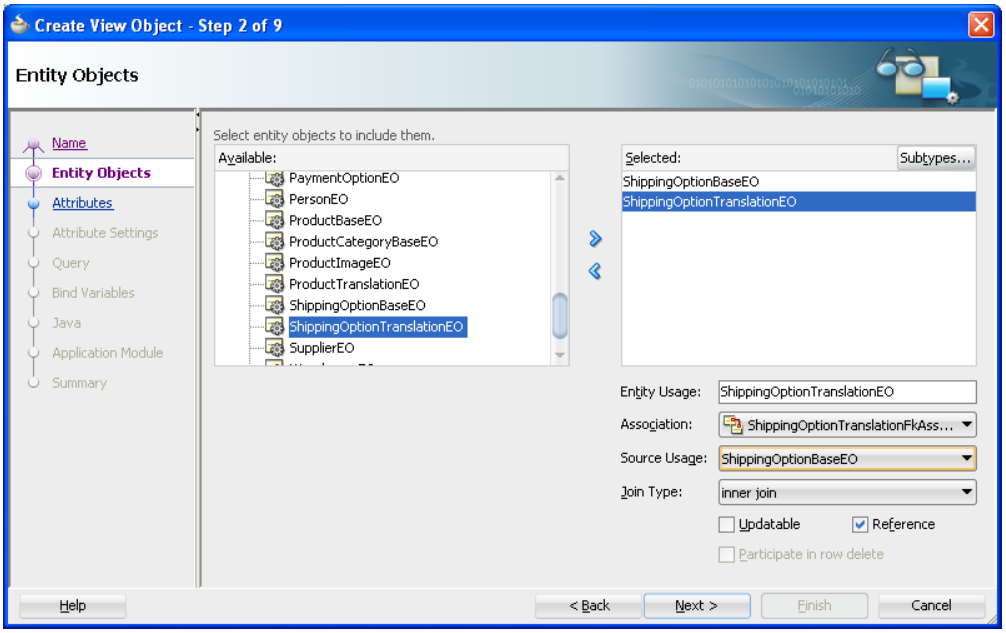
4. In the Entity Objects page, the first entity usage in the **Selected** list is known as the *primary* entity usage for the view object. Select the primary entity object from the **Available** list and shuttle it to the **Selected** list.

   The list is not limited to a single, primary entity usage.

5. To add additional, secondary entity objects to the view object, select them in the **Available** list and shuttle them to the **Selected** list.

   The **Association** dropdown list shows you the name of the association that relates the selected secondary entity usage to the primary one. For example, Figure 5–19 shows the result of adding one secondary reference entity usage, `ShippingOptionTranslationEO`, in addition to the primary `ShippingOptionBaseEO` entity usage. The association that relates to this secondary entity usage is `ShippingOptionTranslationFkAssociation`.

Figure 5–19 Create View Object Wizard, Entity Objects Page



6. Optionally, use the **Alias** field to give a more meaningful name to the entity usage when the default name is not clear.

7. If you add multiple entity usages for the same entity, use the **Association** dropdown list to select which association represents that usage's relationship to the primary entity usage. Click **Next**.

   For each secondary entity usage, the **Reference** option is enabled to indicate that the entity provides reference information and that it is not the primary entity. The **Updatable** option is disabled. This combination represents the typical usage. However, when you want to create a join view object with multiple, updatable entity usages, see Section 42.9, "Creating a View Object with Multiple Updatable Entities."

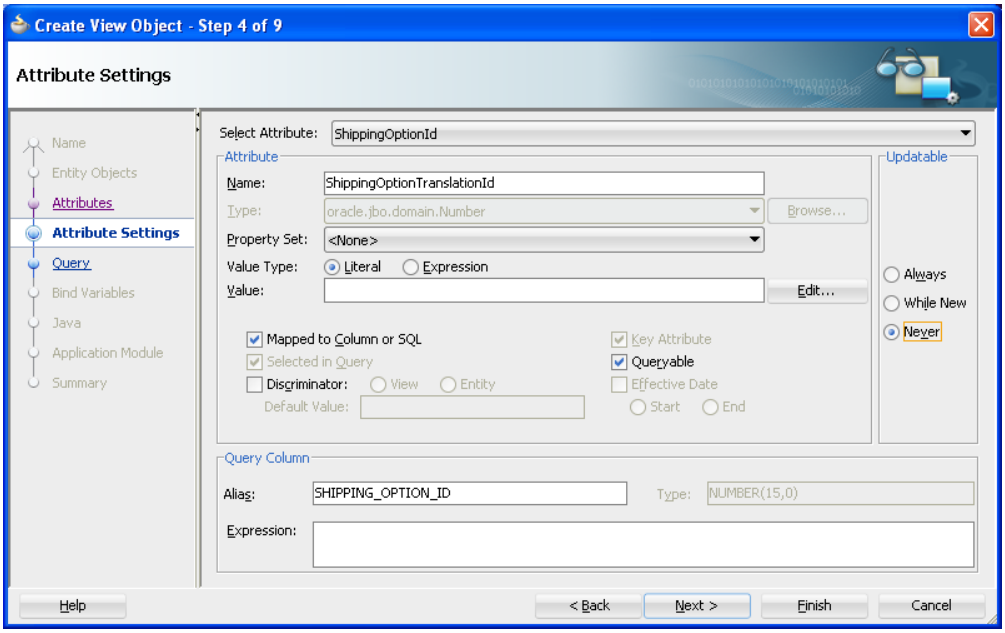
   Secondary entity usages that are updatable can also have the **Participate in row delete** option enabled. This will allow secondary entity attributes to appear `NULL` when the primary entity is displayed.

8. On the Attributes page, select the attributes you want each entity object usage to contribute to the view object. Click **Next**.

9. On the Attribute Settings page, you can rename an attribute when the names are not as clear as they ought to be.

   The same attribute name often results when the reference and secondary entity objects derive from the same table. Figure 5–20 shows the attribute `ShippingOptionId` in the **Select Attribute** dropdown list, which has been renamed to `ShippingOptionTranslationId` in the **Name** field.

Figure 5–20 Create View Object Wizard, Attribute Settings Page



10. Click **Finish**.

5.5.2 How to Select Additional Attributes from Reference Entity Usages

After adding secondary entity usages, you can use the overview editor for the view object to select the specific, additional *attributes* from these new usages that you want to include in the view object.

> **Tip:** The overview editor lets you sort attributes displayed in the Attributes page by their entity usages. By default, the attributes table displays attributes in the order they appear in the underlying entity object. To sort the attributes by entity usage, click the header for the **Entity Usage** column of the attributes table. If the **Entity Usage** column does not appear in the attributes table, click the dropdown menu on the top-right corner of the table (below the button bar) and choose **Select Columns** to add the column to the **Selected** list.

Before you begin:

It may be helpful to have an understanding of how the type of view object effects joins. For more information, see Section 5.5, "Working with Multiple Tables in Join Query Results."

You will need to complete this task:

Create the desired entity objects as described in Section 4.2.1, "How to Create Multiple Entity Objects and Associations from Existing Tables."

To select attributes from a secondary entity usage:

1. In the Application Navigator, double-click the view object.
2. In the overview editor, click the **Attributes** navigation tab and click the dropdown menu on the **Create new attribute** icon and choose **Add Attribute from Entity** to view the list of available entity-derived attributes.
3. In the Attributes dialog, select the desired attribute and add it to the **Selected** list.

Note that even if you didn't intend to include them, JDeveloper automatically verifies that the primary key attribute from each entity usage is part of the **Selected** list. If it's not already present in the list, JDeveloper adds it for you. When you are finished, the overview editor Query page shows that JDeveloper has included the new columns in the `SELECT` statement.

4. Click **OK**.

5.5.3 How to Remove Unnecessary Key Attributes from Reference Entity Usages

The view object attribute corresponding to the primary key attribute of the primary entity usage acts as the primary key for identifying the view row. When you add secondary entity usages, JDeveloper marks the view object attributes corresponding to their primary key attributes as part of the view row key as well. When your view object consists of a single updatable primary entity usage and a number of reference entity usages, the primary key attribute from the primary entity usage is enough to uniquely identify the view row. Further key attributes contributed by secondary entity usages are not necessary and you should disable their **Key Attribute** settings.

For example, based on the view object with primary entity usage `ShippingOptionEO`, you could disable the **Key Attribute** property for the `ShippingOptionTranslationEO` entity usage so that this property is no longer selected for this additional key attribute: `ShippingTranslationsId`.

Before you begin:

It may be helpful to have an understanding of how the type of view object effects joins. For more information, see Section 5.5, "Working with Multiple Tables in Join Query Results."

You will need to complete this task:

Create the desired entity objects as described in Section 4.2.1, "How to Create Multiple Entity Objects and Associations from Existing Tables."

To remove unnecessary key attributes:

1. In the Application Navigator, double-click the view object.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, in the attributes table, select the key attribute (identified by the key icon in the **Name** column), and then click the **Details** tab and deselect the **Key Attribute** property.

5.5.4 How to Hide the Primary Key Attributes from Reference Entity Usages

Since you generally won't want to display the primary key attributes that were automatically added to the view object, you can set the attribute's **Display Hint** property to **Hide**.

Before you begin:

It may be helpful to have an understanding of how the type of view object effects joins. For more information, see Section 5.5, "Working with Multiple Tables in Join Query Results."

You will need to complete this task:

Create the desired entity objects as described in Section 4.2.1, "How to Create Multiple Entity Objects and Associations from Existing Tables."

To hide the primary key attribute:

1. In the Application Navigator, double-click the view object.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the primary key attribute (identified by the key icon in the **Name** column), and then click the **UI Hints** tab and select **Hide** in the **Display Hint** dropdown list.

5.5.5 How to Modify a Default Join Clause to Be an Outer Join When Appropriate

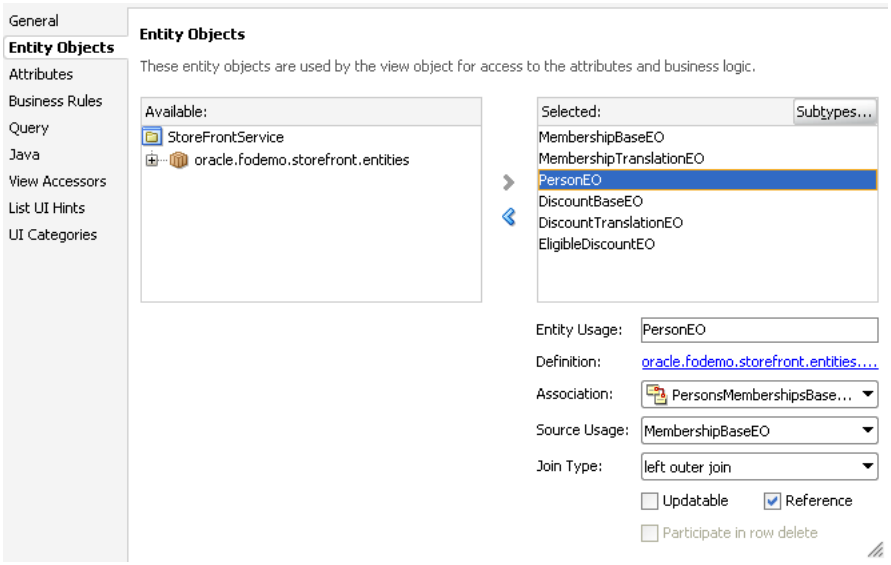
When you add a secondary entity usage to a view object, the entity usage is related to an entity usage that precedes it in the list of selected entities. This relationship is established by an entity association displayed in the **Association** dropdown list in the Entity Objects page of the overview editor for the view object. You use the **Association** dropdown list in the editor to select the entity association that relates the secondary entity usage to the desired preceding entity usage in the **Selected** list. The name of the preceding entity usage is identified in the **Source Usage** dropdown list.

When JDeveloper creates the `WHERE` clause for the join between the table for the primary entity usage and the tables for related secondary entity usages, by default it always creates inner joins. You can modify the default inner join clause to be a left or right outer join when appropriate. The left designation refers to the source entity object named in the selected association. This is the entity identified in the **Source Usage** dropdown list. The right designation refers to the current secondary entity usage that you have selected in the **Selected** list.

In the left outer join, you will include all rows from the left table (related to the entity object named in the **Source Usage** list) in the join, even if there is no matching row from the right table (related to the current secondary entity object selection). The right outer join specifies the reverse scenario: you will include all rows from the right table (related to the entity object named in the **Source Usage** list) in the join, even if there is no matching row from the left table (related to the current secondary entity object selection).

For example, assume that a person is not yet assigned a membership status. In this case, the `MembershipId` attribute will be `NULL`. The default inner join condition will not retrieve these persons from the `MEMBERSHIPS_BASE` table. Assuming that you want persons without membership status to be viewable and updatable through the `MembershipDiscountsVO` view object, you can use the Entity Objects page in the overview editor for the view object to change the query into an left outer join to the `MEMBERSHIPS_BASE` table for the possibly null `MEMBERSHIP_ID` column value. When you add the person entity to the view object, you would select the **left outer join** as the join type. As shown in Figure 5–21, the association `PersonsMembershipsBaseFkAssoc` identifies a source usage `MembershipBaseEO` on the left side of the join and the selected `PersonEO` entity usage on the right side. The view object `MembershipDiscountsVO` joins the rows related to both of these entity objects and defines a left outer join for `PersonEO` to allow the view object to return rows from the table related to `MembershipBaseEO` even if they do not have a match in the table related to `PersonEO`.

Figure 5–21 Setting an Outer Join to Return NULL Rows from Joined Entities



The view object's updated `WHERE` clause includes the addition `(+)` operator on the right side of the equals sign for the related table whose data is allowed to be missing in the left outer join:

```
PersonEO.MEMBERSHIP_ID = MembershipBaseEO.MEMBERSHIP_ID(+)
```

Before you begin:

It may be helpful to have an understanding of how the type of view object effects joins. For more information, see Section 5.5, "Working with Multiple Tables in Join Query Results."

You will need to complete this task:

Create the desired entity objects as described in Section 4.2.1, "How to Create Multiple Entity Objects and Associations from Existing Tables."

To change an inner join type to an outer join:

1. In the Application Navigator, double-click the view object.
2. In the overview editor, click the **Entity Objects** navigation tab.

   The entity object you select represents the table on the right side of the join.
3. In the Entity Objects page, in the **Selected** list, select the entity object that you want to change the join type for.

   The entity object you select represents the table on the right side of the join.
4. In the **Association** dropdown list, if only one association is defined, leave it selected; otherwise, select among the list of entity object associations that relate the secondary entity object to the desired entity object. The entity usage that represents the joined table will be displayed in the **Source Usage** dropdown list.

   The entity object in the **Source Usage** dropdown list that you choose through the association selection represents the table on the left side of the join.
5. In the **Join Type** dropdown list, decide how you want the view object to return rows from the joined entity objects:

- **left outer join** will include rows from the left table in the join, even if there is no matching row from the right table.
- **right outer join** will include rows from the right table in the join, even if there is no matching row from the left table.

The **Source Usage** dropdown list is the left side of the join and the current entity usage in the **Selected** list is the right side.

5.5.6 What Happens When You Reference Entities in a View Object

When you create a join view object to include secondary entity usages by reference, JDeveloper updates the view object's XML component definition to include information about the additional entity usages. For example, the `ShippingOptionsVO.xml` file for the view object includes an additional reference entity usage. You will see this information recorded in the multiple `<EntityUsage>` elements. For example, Example 5–4 shows an entity usage entry that defines the primary entity usage.

Example 5–4 Primary Entity Usage

```
<EntityUsage
   Name="ShippingOptionBaseEO"
   Entity="oracle.fodemo.storefront.entities.ShippingOptionBaseEO"/>
```

The secondary reference entity usages will have a slightly different entry, including information about the association that relates it to the primary entity usage, like the entity usage shown in Example 5–5.

Example 5–5 Secondary Reference Entity Usage

```
<EntityUsage
   Name="ShippingOptionTranslationEO"
   Entity="oracle.fodemo.storefront.entities.ShippingOptionTranslationEO"
   Association="oracle.fodemo.storefront.entities.associations.
                  ShippingOptionTranslationFkAssoc"
   AssociationEnd="oracle.fodemo.storefront.entities.associations.
                  ShippingOptionTranslationFkAssoc.ShippingOptionTranslation"
   SourceUsage="oracle.fodemo.storefront.store.queries.ShippingOptionsVO.
                  ShippingOptionBaseEO"
   ReadOnly="true"
   Reference="true"/>
```

Each attribute entry in the XML file indicates which entity usage it references. For example, the entry for the `ShippingOptionId` attribute in Example 5–6 shows that it's related to the `ShippingOptionBaseEO` entity usage, while the `ShippingMethod` attribute is related to the `ShippingOptionTranslationEO` entity usage.

Example 5–6 Entity Usage Reference of View Object Attribute

```
   <ViewAttribute
      Name="ShippingOptionId"
      IsNotNull="true"
      EntityAttrName="ShippingOptionId"
      EntityUsage="ShippingOptionBaseEO"
      AliasName="SHIPPING_OPTION_ID" >
   </ViewAttribute>
...
   <ViewAttribute
```

```
    Name="ShippingMethod"
    IsUpdatable="true"
    IsNotNull="true"
    EntityAttrName="ShippingMethod"
    EntityUsage="ShippingOptionTranslationEO"
    AliasName="SHIPPING_METHOD" >
</ViewAttribute>
```

The Create View Object wizard uses this association information at design time to automatically build the view object's join `WHERE` clause. It uses the information at runtime to enable keeping the reference information up to date when the end user changes foreign key attribute values.

5.5.7 How to Create Joins for Read-Only View Objects

To create a read-only view object joining two tables, use the Create View Object wizard.

Before you begin:

It may be helpful to have an understanding of how the type of view object effects joins. For more information, see Section 5.5, "Working with Multiple Tables in Join Query Results."

To create a read-only view object joining two tables:

1. In the Application Navigator, right-click the project in which you want to create the view object and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **View Object**, and click **OK**.
3. In the Initialize Business Components Project dialog, select the database connection or choose **New** to create a connection. Click **OK**.
4. In the Create View Object wizard, on the Name page, enter a package name and a view object name. Select **SQL query** to indicate that you want this view object to manage data with read-only access. Click **Next**.
5. On the Query page, use one of the following techniques to create the SQL query statement that joins the desired tables:
   - Paste any valid SQL statement into the **Query Statement** box.
   - Click **Query Builder** to open the SQL Statement dialog and use the interactive query builder, as described in Section 5.5.9, "How to Use the Query Builder with Read-Only View Objects."
6. After entering or building the query statement, click **Next**.
7. On the Bind Variables page, do one of the following:
   - If the query does not reference any bind variables, click **Next** to skip Step 3.
   - To add a bind variable and work with it in the query, see Section 5.10.1, "How to Add Bind Variables to a View Object Definition."
8. On the Attribute Mappings page, click **Finish**.

5.5.8 How to Test the Join View

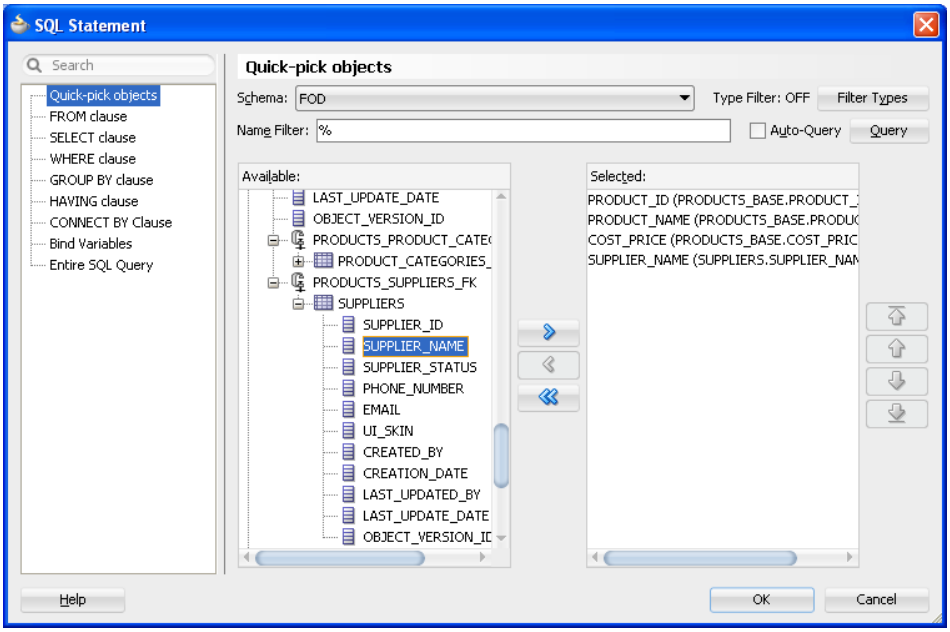
To test the new view object, edit the application module and on the Data Model page add an instance of the new view object to the data model. Then, use the Oracle ADF

Model Tester to verify that the join query is working as expected. For details about editing the data model and running the Oracle ADF Model Tester, see Section 6.3, "Testing View Object Instances Using the Oracle ADF Model Tester."

5.5.9 How to Use the Query Builder with Read-Only View Objects

The Quick-pick objects page of the SQL Statement dialog lets you view the tables in your schema, including the foreign keys that relate them to other tables. To include columns in the select list of the query, shuttle the desired columns from the **Available** list to the **Selected** list. For example, Figure 5–22 shows the result of selecting the `PRODUCT_ID`, `PRODUCT_NAME`, and `COST_PRICE` columns from the `PRODUCTS` table, along with the `SUPPLIER_NAME` column from the `SUPPLIERS` table. The column from the second table appears, beneath the `PRODUCTS_SUPPLIERS_FK` foreign key in the **Available** list. When you select columns from tables joined by a foreign key, the query builder automatically determines the required join clause for you.

Figure 5–22 View Object Query Builder to Define a Join



Optionally, use the `WHERE` clause page of the SQL Statement dialog to define the expression. To finish creating the query, click **OK** in the SQL Statement dialog. The Edit Query dialog will show a query like the one shown in Example 5–7.

Example 5–7 Creating a Query Using SQL Builder

```
SELECT
    PRODUCTS_BASE.PRODUCT_ID PRODUCT_ID,
    PRODUCTS_BASE.PRODUCT_NAME PRODUCT_NAME,
    PRODUCTS_BASE.COST_PRICE COST_PRICE,
    SUPPLIERS.SUPPLIER_NAME SUPPLIER_NAME
FROM
    PRODUCTS_BASE JOIN SUPPLIERS USING (SUPPLIER_ID)
```

You can use the Attributes page of the Create View Object wizard to rename the view object attribute directly as part of the creation process. Renaming the view object here saves you from having to edit the view object again, when you already know the attribute names that you'd like to use. As an alternative, you can also alter the default

Java-friendly name of the view object attributes by assigning a column alias, as described in Section 5.9.2, "How to Name Attributes in Expert Mode."

5.5.10 What You May Need to Know About Join View Objects

If your view objects reference multiple entity objects, they are displayed as separate entity usages on a business components diagram.

5.6 Working with Multiple Tables in a Master-Detail Hierarchy

Many queries you will work with will involve multiple tables that are related by foreign keys. In this scenario, you can create separate view objects that query the related information and then link a "source" view object to one or more "target" view objects to form a master-detail hierarchy.

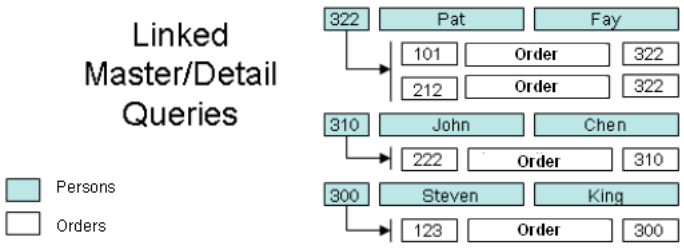
There are two ways you might handle this situation. You can either:

- Create a view link that defines how the source and target view objects relate.
- Create a view link based on an association between entity objects when the source and target view objects are based on the underlying entity objects' association.

In either case, you use the Create View Link wizard to define the relationship.

Figure 5–23 illustrates the multilevel result that master-detail linked queries produce.

Figure 5–23 Linked Master-Detail Queries



5.6.1 How to Create a Master-Detail Hierarchy for Read-Only View Objects

When you want to show the user a set of master rows, and for each master row a set of coordinated detail rows, then you can create view links to define how you want the master and detail view objects to relate. For example, you could link the `Persons` view object to the `Orders` view object to create a master-detail hierarchy of customers and the related set of orders they have placed.

To create the view link, use the Create View Link wizard.

Before you begin:

It may be helpful to have an understanding of the ways to create a master-detail hierarchy. For more information, see Section 5.6, "Working with Multiple Tables in a Master-Detail Hierarchy."

You will need to complete this task:

Create the desired read-only view objects as described in Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object."

To create a view link between read-only view objects:

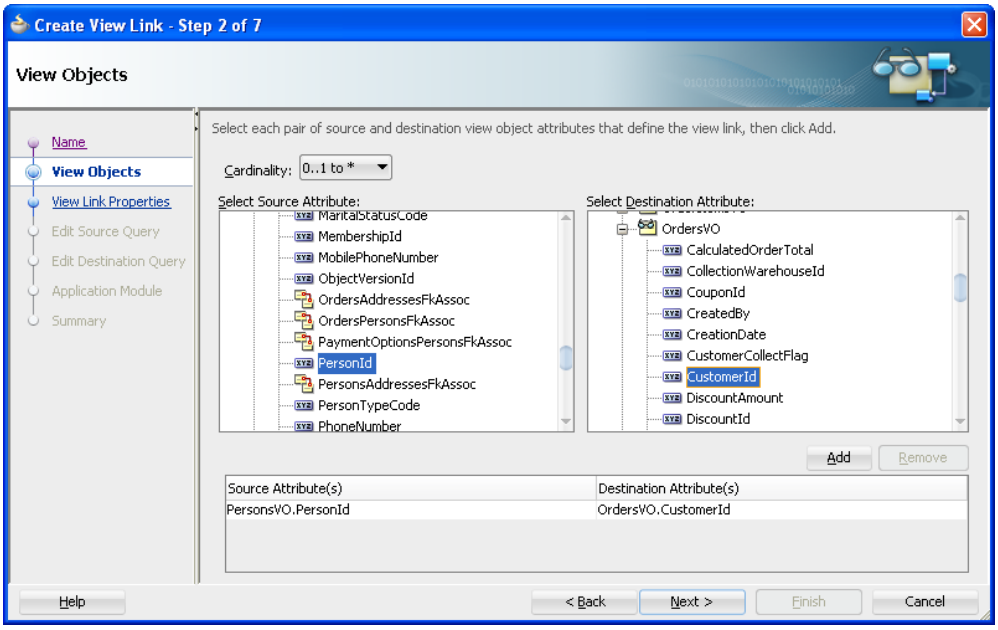
1. In the Application Navigator, right-click the project in which you want to create the view object and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **View Link**, and click **OK**.
3. In the Create View Link wizard, on the Name page, enter a package name and a view link name. For example, given the purpose of the view link, a name like `OrdersPlacedBy` is a valid name. Click **Next**.
4. On the View Objects page, select a "source" attribute from the view object that will act as the master.

   For example, Figure 5–24 shows the `PersonId` attribute selected from the `PersonsVO` view object to perform this role. Click **Next**.
5. On the View Objects page, select a corresponding destination attribute from the view object that will act as the detail.

   For example, if you want the detail query to show orders that were placed by the currently selected customer, select the `CustomerId` attribute in the `OrdersVO` to perform this role.
6. Click **Add** to add the matching attribute pair to the table of source and destination attribute pairs below. When you are finished defining master and detail link, click **Next**.

   Figure 5–24 shows just one (`PersonId`,`CustomerId`) pair. However, if you require multiple attribute pairs to define the link between master and detail, repeat the steps for the View Objects page to add additional source-target attribute pairs.

Figure 5–24 Defining Source/Target Attribute Pairs While Creating a View Link



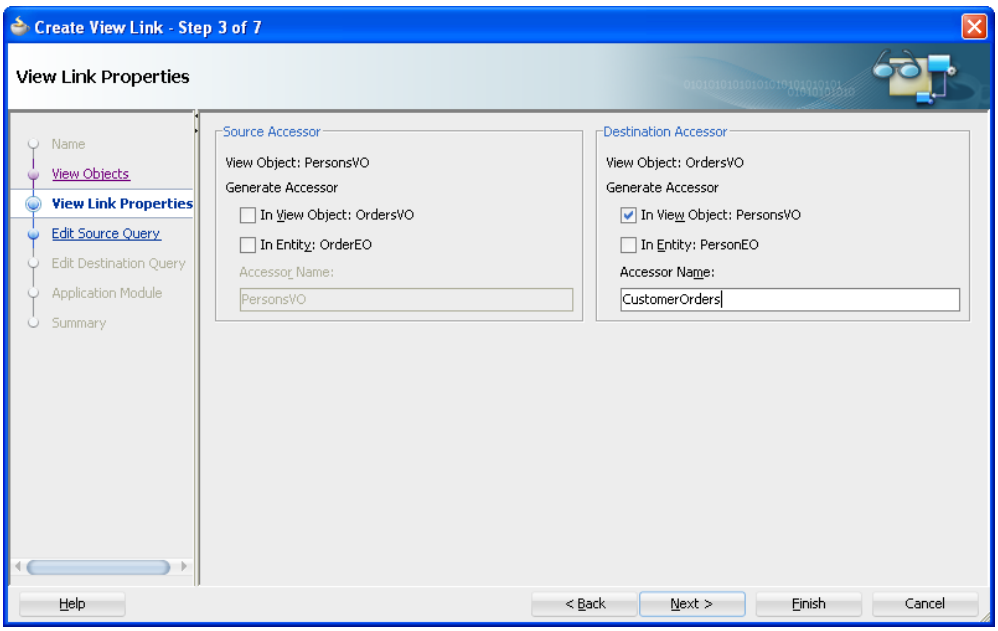
7. On the View Link Properties page, you can use the **Accessor Name** field to change the default name of the accessor that lets you programmatically access the destination view object.

By default, the accessor name will match the name of the destination view object. For example, you might change the default accessor name `OrdersVO` to `CustomerOrders` to better describe the master-detail relationship that the accessor defines.

8. Also on the View Link Properties page, you control whether the view link represents a one-way relationship or a bidirectional one.

   By default, a view link is a one-way relationship that allows the current row of the source (master) to access a set of related rows in the destination (detail) view object. For example, in Figure 5–25, the checkbox settings indicate that you'll be able to access a detail collection of rows from `OrdersVO` for the current row in `PersonsVO`, but not vice versa. In this case, this behavior is specified by the checkbox setting in the **Destination Accessor** group box for `OrdersVO` (the **Generate Accessor In View Object: PersonsVO** box is selected) and checkbox setting in the **Source Accessor** group box for `PersonsVO` (the **Generate Accessor In View Object: OrdersVO** box is not selected).

Figure 5–25 View Link Properties Control Name and Direction of Accessors



9. On the Edit Source Query page, preview the view link SQL predicate that will be used at runtime to access the master row in the source view object and click **Next**.

10. On the Edit Destination Query page, preview the view link SQL predicate that will be used at runtime to access the correlated detail rows from the destination view object for the current row in the source view object and click **Next**.

11. On the Application Module page, add the view link to the data model for the desired application module and click **Finish**.

    By default the view link will not be added to the application module's data model. Later you can add the view link to the data model using the overview editor for the application module.

5.6.2 How to Create a Master-Detail Hierarchy for Entity-Based View Objects

Just as with read-only view objects, you can link entity-based view objects to other view objects to form master-detail hierarchies of any complexity. The only difference in the creation steps involves the case when both the master and detail view objects are entity-based view objects and their respective entity usages are related by an association. In this situation, since the association captures the set of source and destination attribute pairs that relate them, you create the view link just by indicating which association it should be based on.

To create an association-based view link, you use the Create View Link wizard.

Before you begin:

It may be helpful to have an understanding of the ways to create a master-detail hierarchy. For more information, see Section 5.6, "Working with Multiple Tables in a Master-Detail Hierarchy."

You will need to complete this task:

Create the desired entity-based view objects as described in Section 5.2.1, "How to Create an Entity-Based View Object."

To create an association-based view link

1. In the Application Navigator, right-click the project in which you want to create the view object and choose **New**.

   To avoid having to type in the package name in the Create View Link wizard, you can choose **New View Link** on the context menu of the **links** package node in the Application Navigator.

2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **View Link**, and click **OK**.

3. In the Create View Link wizard, on the Name page, supply a package and a component name.

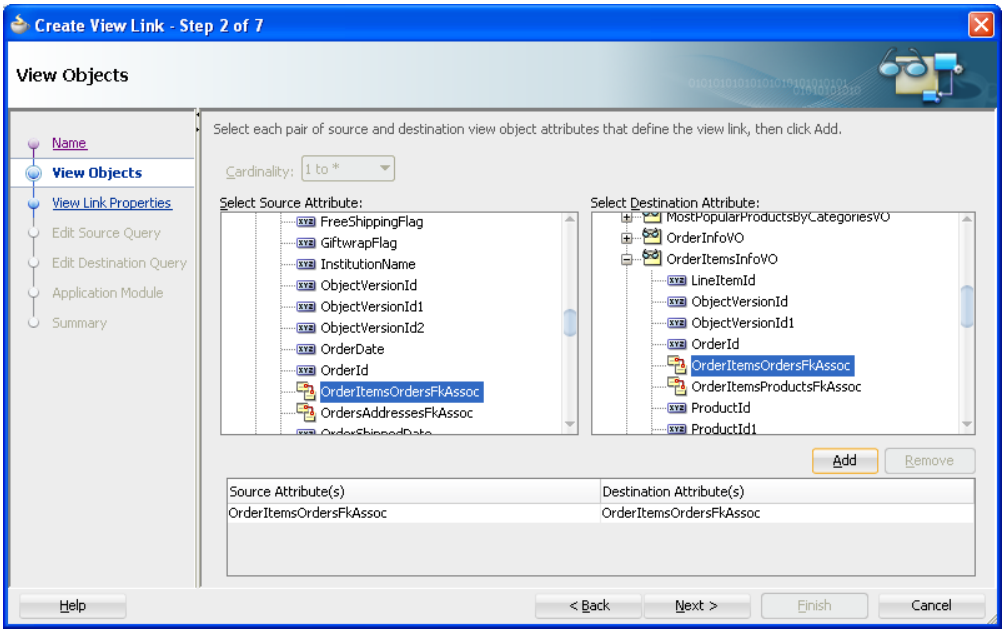
4. On the View Objects page, in the **Select Source Attribute** tree expand the source view object in the desired package. In the **Select Destination Attribute** tree expand the destination view object.

   For entity-based view objects, notice that in addition to the view object attributes, relevant associations also appear in the list.

5. Select the same association in both **Source** and **Destination** trees. Then click **Add** to add the association to the table below.

   For example, Figure 5–26 shows the same `OrderItemsOrdersFkAssoc` association in both **Source** and **Destination** trees selected.

Figure 5–26 Master and Detail Related by an Association Selection



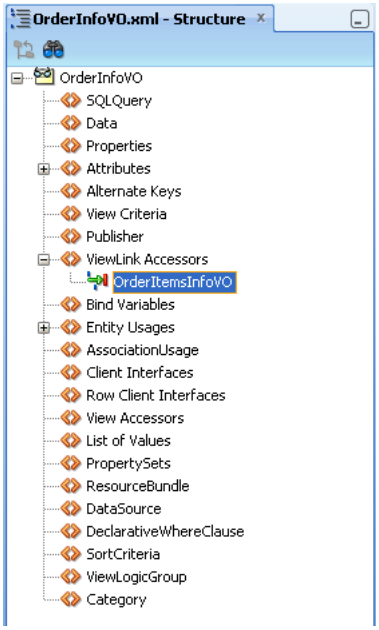
6. Click **Finish**.

5.6.3 What Happens When You Create Master-Detail Hierarchies Using View Links

When you create a view link or an association-based view link, JDeveloper creates the XML component definition file that represents its declarative settings and saves it in the directory that corresponds to the name of its package. For example, if the view link is named `OrderInfoToOrderItemsInfo` and it appears in the `queries.links` package, then the XML file created will be `./queries/link/OrderInfoToOrderItemsInfo.xml` under the project's source path. This XML file contains the declarative information about the source and target attribute pairs you've specified and, in the case of an association-based view link, contains the declarative information about the association that relates the source and target view objects you've specified.

In addition to saving the view link component definition itself, JDeveloper also updates the XML definition of the *source* view object in the view link relationship to add information about the view link accessor you've defined. As a confirmation of this, you can select the source view object in the Application Navigator and inspect its details in the Structure window. As shown in Figure 5–27, you can see the defined accessor under the **ViewLink Accessors** node for the `OrderItemsInfoVO` source view object of the `OrderInfoToOrderItemsInfo` view link.

Figure 5–27 View Object with View Link Accessor in the Structure Window



Note: A view link defines a basic master-detail relationship between two view objects. However, by creating more view links you can achieve master-detail hierarchies of any complexity, including:

- Multilevel master-detail-detail
- Master with multiple (peer) details
- Detail with multiple masters

The steps to define these more complex hierarchies are the same as the ones covered in Section 5.6.2, "How to Create a Master-Detail Hierarchy for Entity-Based View Objects," you just need to create it one view link at time.

5.6.4 How to Enable Active Master-Detail Coordination in the Data Model

When you enable programmatic navigation to a row set of correlated details by defining a view link as described in Section 5.6.2, "How to Create a Master-Detail Hierarchy for Entity-Based View Objects," the view link plays a *passive* role, simply defining the information necessary to retrieve the coordinated detail row set when your code requests it. The view link accessor attribute is present and programmatically accessible in any result rows from any instance of the view link's source view object. In other words, programmatic access does not require modifying the application module's data model.

However, since master-detail user interfaces are such a frequent occurrence in enterprise applications, the view link can be also used in a more *active* fashion so you can avoid needing to coordinate master-detail screen programmatically. You opt to have this active master-detail coordination performed by *explicitly* adding an instance of a view-linked view object to your application module's data model.

To enable active master-detail coordination, open the application module in the overview editor and select the Data Model page.

Before you begin:

It may be helpful to have an understanding of the ways to create a master-detail hierarchy. For more information, see Section 5.6, "Working with Multiple Tables in a Master-Detail Hierarchy."

You will need to complete this task:

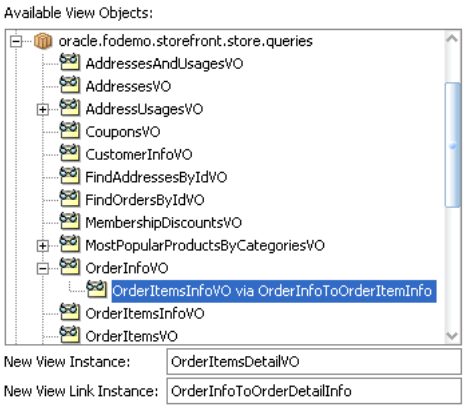
Create the desired view objects as described in Section 5.2.1, "How to Create an Entity-Based View Object" and Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object."

To add a detail instance of a view object:

1. In the Application Navigator, double-click the application module.
2. In the overview editor, click the **Data Model** navigation tab.
3. In the Data Model page, expand the **View Object Instances** section and, in the **Available View Objects** list, select the detail view object node that is indented beneath the master view object.

   Note that the list shows the detail view object *twice*: once on its own, and once as a detail view object via the view link. For example, in Figure 5–28 you would select the detail view object `OrderItemsInfoVO via OrderInfoToOrderItemInfo` instead of the view object labeled as `OrderItemsInfoVO` (which, in this case, appears beneath the highlighted view object).

Figure 5–28 Detail View Object Selection from Available View Objects



4. Enter a name for the detail instance you're about to create in the **Name View Instance** field below the **Available View Objects** list.

   For example, Figure 5–28 shows the name `OrderItemsDetailVO` for the instance of the `OrderItemsInfoVO` view object that is a detail view.

5. In the **Data Model** list, select the instance of the view object that you want to be the actively-coordinating master.
6. Click **Add Instance** to add the detail instance to the currently selected master instance in the data model, with the name you've chosen.

   For example, in Figure 5–29, the **Data Model** list shows a master-detail hierarchy of view object instances with `OrderItemsDetailVO` as the detail view object.

Figure 5–29 Data Model with View Linked View Object

5.6.5 How to Test Master-Detail Coordination

To test active master-detail coordination, launch the Oracle ADF Model Tester on the application module by choosing **Run** from its context menu in the Application Navigator. The Oracle ADF Model Tester data model tree shows the view link instance that is actively coordinating the detail view object instance with the master view object instance. You can double-click the view link instance node in the tree to open a master-detail data view page in the Oracle ADF Model Tester. Then, when you use the toolbar buttons to navigate in the master view object — changing the view object's current row as a result — the coordinated set of details is automatically refreshed and the user interface stays in sync.

If you double-click another view object that is not defined as a master and detail, a second tab will open to show its data; in that case, since it is not actively coordinated by a view link, its query is not constrained by the current row in the master view object.

For information about editing the data model and running the Oracle ADF Model Tester, see Section 6.3, "Testing View Object Instances Using the Oracle ADF Model Tester."

5.6.6 How to Access the Detail Collection Using the View Link Accessor

To work with view links effectively, you should also understand that view link accessor attributes return a `RowSet` object and that you can access a detail collection using the view link accessor programmatically.

5.6.6.1 Accessing Attributes of Row by Name

At runtime, the `getAttribute()` method on a `Row` object allows you to access the value of any attribute of that row in the view object's result set by name. The view link accessor behaves like an additional attribute in the current row of the source view object, so you can use the same `getAttribute()` method to retrieve its value. The only practical difference between a regular view attribute and a view link accessor attribute is its data type. Whereas a regular view attribute typically has a scalar data type with a value like `303` or `ngreenbe`, the value of a view link accessor attribute is a row set of zero or more correlated detail rows. Assuming that `curUser` is a `Row` object from some instance of the `Orders` view object, you can write a line of code to retrieve the detail row set of order items:

```
RowSet items = (RowSet)curUser.getAttribute("OrderItems");
```

Note: If you generate the custom Java class for your view row, the type of the view link accessor will be `RowIterator`. Since at runtime the return value will always be a `RowSet` object, it is safe to cast the view link attribute value to `RowSet`.

5.6.6.2 Programmatically Accessing a Detail Collection Using the View Link Accessor

Once you've retrieved the `RowSet` object of detail rows using a view link accessor, you can loop over the rows it contains just as you would loop over a view object's row set of results, as shown in Example 5–8.

Example 5–8 Programmatically Accessing a Detail Collection

```
while (items.hasNext()) {
  Row curItem = items.next();
  System.out.println("--> (" + curItem.getAttribute("LineItemId") + ") " +
                     curItem.getAttribute("LineItemTotal"));
}
```

For information about creating a test client, see Section 6.4.6, "How to Access a Detail Collection Using the View Link Accessor."

5.6.7 How to Create a Master-Detail Hierarchy for Entity Objects with Transient-Only Attributes

When you link entity-based view objects to form master-detail hierarchies, the view objects and their respective entity usages are typically related by an association. At runtime, the association constructs an internal association view object that enables a query to be executed to enable the master-detail coordination. An exception to this scenario occurs when the entity object that participates in an association consists of all nonpersistent attributes. This may occur when you define an entity object with transient attributes that you wish to populate programmatically at runtime. In this case, you can use the association overview editor that links the nonpersistent entity object to select an association view object to perform the query.

Before you begin:

It may be helpful to have an understanding of associations. For more information, see Section 4.2.3, "What Happens When You Create Entity Objects and Associations from Existing Tables."

You will need to complete these tasks:

- Create the entity-based view object for the nonpersistent entity object, as described in Section 5.2.1, "How to Create an Entity-Based View Object."
- Use the Create Association wizard to create an association between the nonpersistent entity object and a database-derived entity object. For details about creating an association, see Section 4.3, "Creating and Configuring Associations."

To customize an association for nonpersistent entity objects:

1. In the Application Navigator, double-click the association.
2. In the overview editor, click the **Tuning** navigation tab and then click the **Edit accessors** button.

3. In the Custom Views dialog, select **Use Custom View Object** for the non-persistent entity object.
4. In the **Select View Object** list, select the view object that you created based on the non-persistent entity-object.
5. Click **OK** to save the definition.

5.7 Working with a Single Table in a Recursive Master-Detail Hierarchy

A recursive data model is one that utilizes a query that names source and destination attributes in a master-detail relationship based on a single table. In a typical master-detail relationship, the source attribute is supplied by the primary key attribute of the master view object and the destination attribute is supplied by foreign key attribute in the detail view object. For example, a typical master-detail relationship might relate the `DepartmentId` attribute on the `DEPARTMENT` table and the corresponding `DepartmentId` attribute on the `EMPLOYEE` table. However, in a recursive data model, the source attribute `EmployeeId` and the target attribute `ManagerId` both exist in the `EMPLOYEE` table. The query for this relationship therefore involves only a single view object. In this scenario, you create the view object for a single base entity object that specifies both attributes and then you define a self-referential view link to configure this view object as both the "source" and the "target" view object to form a master-detail hierarchy.

After you create the view link, there are two ways you can handle the recursive master-detail hierarchy in the data model project. You can either:

- Create a data model that exposes two instances of the same view object, one playing the role as master and the other playing the role as detail, actively coordinated by a view link instance. This can be useful when you anticipate needing to show a single level of master rows and detail rows at a time in two separate tables.
- Create a data model that exposes only a single instance of the view object, and use the view link accessor attribute in each row to access a row set of details. This is the more typical use case of the two because it allows you to display (or programmatically work with) the recursive master-detail hierarchy to any number of levels that exist in the data. For example, to show the recursive hierarchy in a `tree` or `treeTable` component, you would use this approach, as described in Section 29.4.1, "How to Display Master-Detail Objects in Trees."

5.7.1 How to Create a Recursive Master-Detail Hierarchy for an Entity-Based View Object

In a recursive master-detail hierarchy, the attributes of the view object that you select for the source and destination in the view link will typically be the same pair of attributes that define the self-referential association between the underlying entity object, if this association exists. While this underlying association is not required to create the view link, it does simplify the creation of the view link, so you will first create a foreign key association for the base entity object of the view object.

To create the recursive master-detail hierarchy:

1. Create the foreign key association for the base entity object of the view object.
2. Create the data model to expose the view object with a view criteria that will filter the view instance's results to include only those rows you want to see at the "root" of the hierarchy.

5.7.1.1 Creating an Association-Based, Self-Referential View Link

To create an association, you use the Create Association wizard. Then the association will appear as a selection choice when you use the Create View Link wizard. The view link will be self-referential because the association you select for the source and the destination view object names the *same* entity object, which is derived from a single database table.

Before you begin:

It may be helpful to have an understanding of the recursive data model. For more information, see Section 5.7, "Working with a Single Table in a Recursive Master-Detail Hierarchy."

You will need to complete this task:

- When you create the view link JDeveloper won't be able to infer the association between the source and destination attributes of the entity object. To support the recursive hierarchy, you can use the Create Association wizard to create an association between the source attribute and the destination attribute. On the Entity Objects page, select the same entity object to specify the source and destination attributes and leave all other default selections unchanged in the wizard. For details about creating an association, see Section 4.3, "Creating and Configuring Associations."

  For example, assume the recursive master-detail hierarchy displays a list of employees based on their management hierarchy. In this scenario, you would create the association based on the `Employees` entity object. On the Entity Objects page of the Create Association wizard, you would select `Employees.EmployeeId` as the source attribute and `Employee.ManagerId` as the destination attribute. The entity object `Employees` supplies both attributes to ensure the association is self-referential.

To create an association-based, self-referential view link:

1. In the Application Navigator, right-click the project in which you want to create the view object and choose **New**.

   To avoid having to type in the package name in the Create View Link wizard, you can choose **New View Link** on the context menu of the **links** package node in the Application Navigator.

2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **View Link**, and click **OK**.

3. In the Create View Link wizard, on the Name page, supply a package and a component name.

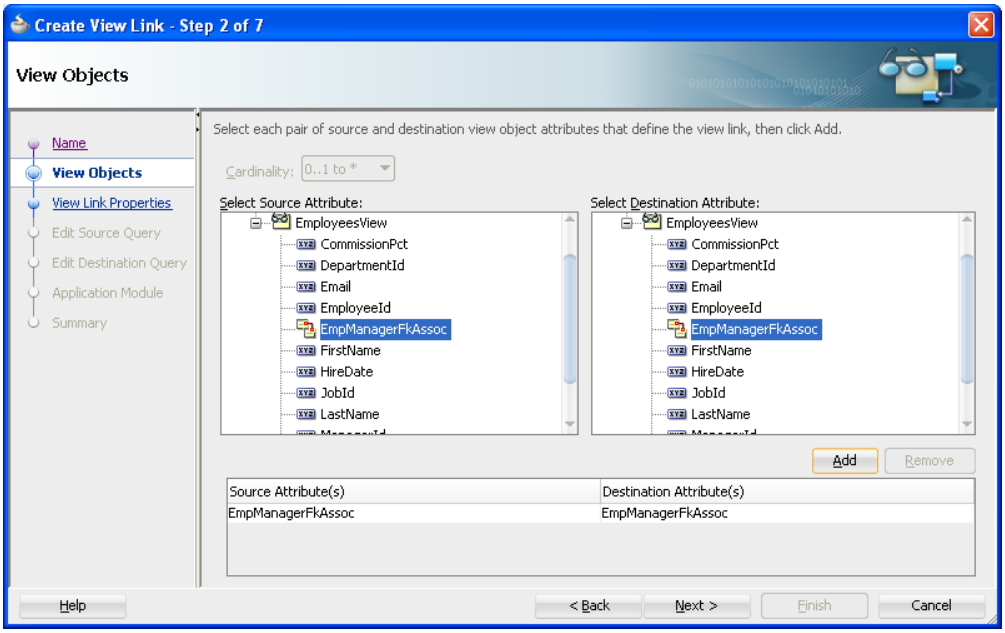
4. On the View Objects page, in the **Select Source Attribute** tree expand the source view object in the desired package. In the **Select Destination Attribute** tree expand the destination view object.

   For entity-based view objects, notice that in addition to the view object attributes, relevant associations also appear in the list.

5. Select the same association in both **Source** and **Destination** trees. Then click **Add** to add the association to the table below.

   For example, Figure 5–30 shows the same `EmpManagersFkAssoc` association in both **Source** and **Destination** trees selected. The view link is self-referential because the definition of the association names the source and destination attribute on the *same* entity object (in this case, `Employees`).

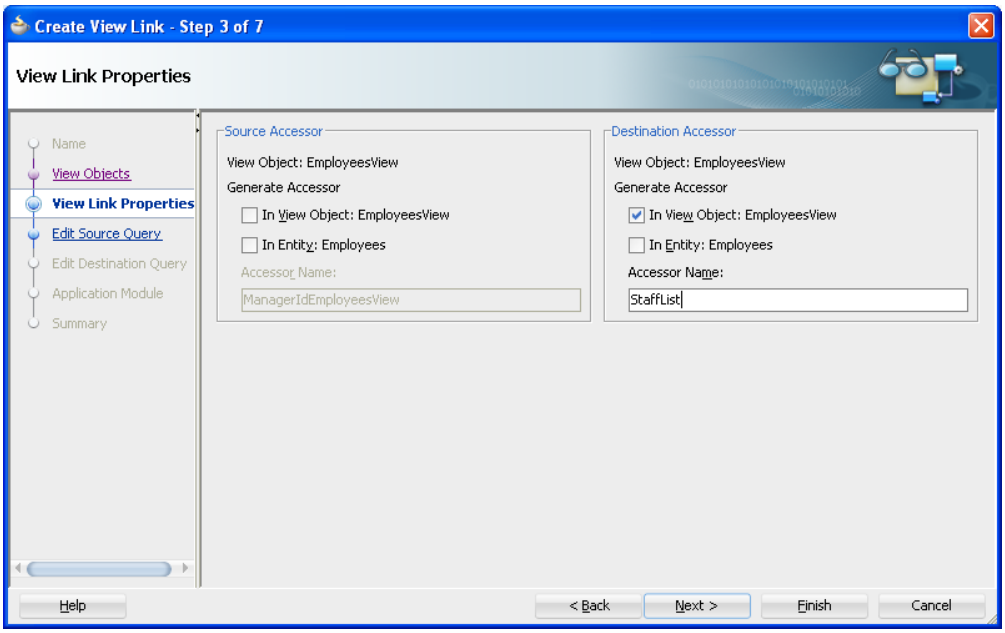
Figure 5–30 Master and Detail Related by a Self-Referential Association Selection



6. On the View Link Properties page, leave the default selections unchanged, but edit the accessor name of the destination accessor to provide a meaningful name.

   For example, Figure 5–31 shows the destination accessor has been renamed from `EmployeesView` to `StaffList`. This name will be exposed in the binding editor when the user interface developer populates the ADF Faces tree component by selecting this accessor. The name you provide will make clear to the UI developer the purpose of the accessor; in this case, to generate a list of employees associated with each manager.

Figure 5–31 Renamed Destination Accessor in View LInk



7. Click **Finish**.

5.7.1.2 Exposing the View Instance and Filter with a View Criteria

When you are ready to expose the view object in your project's data model, you will configure the view instance in the data model to use a view criteria to filter the initial value in the root of the tree. For example, assume the recursive master-detail hierarchy displays a list of employees based on their management hierarchy. In this scenario, you'll configure the view criteria's bind variable to specify the employee ID of the employee that you want to be the root value of the entire hierarchy. In this case, the root value of the recursive hierarchy of managers and employees would be the employee ID of the highest level manager in the organization.

Before you begin:

It may be helpful to have an understanding of the recursive data model. For more information, see Section 5.7, "Working with a Single Table in a Recursive Master-Detail Hierarchy."

You will need to complete this task:

- Create the entity-based view object and create a view criteria that will filter the view instance's results to include only those rows you want to see at the "root" of the hierarchy. To create a view criteria that uses a bind variable to filter the view object, see Section 5.11, "Working with Named View Criteria."

  For example, in a recursive hierarchy of managers and employees, you would create the entity-based view object `EmployeesView`. After you create the view object in the Create View Object wizard, you can use the Query page of the overview editor to create a bind variable and view criteria which allow you to identify the employee or employees that will be seen at the top of the hierarchy. If only a single employee should appear at the root of the hierarchy, then the view criteria in this scenario will filter the employees using a bind variable for the employee ID (corresponding to the source attribute) and the `WHERE` clause shown in the Create View Criteria dialog would look like `( (Employees.EMPLOYEE_ID = :TheEmployeeId ) )`, where `TheEmployeeId` is the bind variable name. For more information on creating a view criteria that uses a bind variable to filter the view object, see Section 5.12.2.1, "Creating a Data Source View Object to Control the Cascading List."

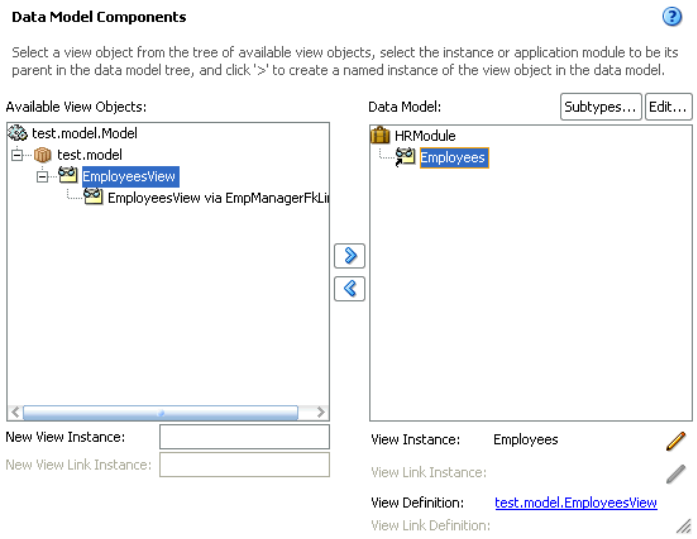
To define the view object instance in an existing application module:

1. In the Application Navigator, double-click the application module.
2. In the overview editor, click the **Data Model** navigation tab.
3. In the Data Model page, expand the **View Object Instances** section and, in the **Available View Objects** list, select the view object definition that you defined the view criteria to filter.

   The **New View Instance** field below the list shows the name that will be used to identify the next instance of that view object that you add to the data model.
4. To change the name before adding it, enter a different name in the **New View Instance** field.
5. With the desired view object selected, shuttle the view object to the **Data Model** list.

   Figure 5–32 shows the view object `EmployeesView` has been renamed to `Employees` before it was shuttled to the **Data Model** list.

Figure 5–32 Data Model Displays Added View Object Instance

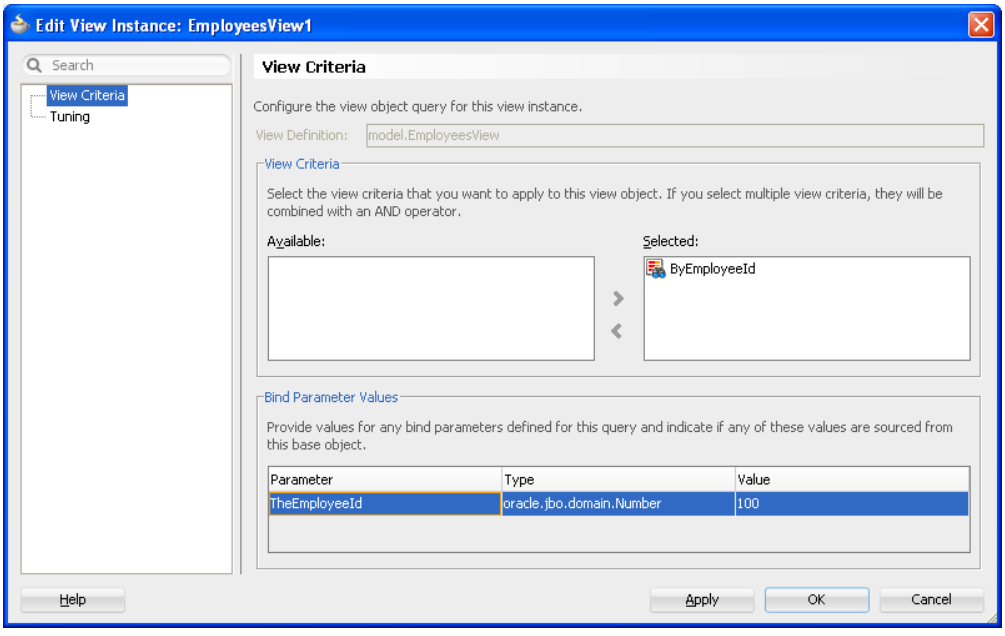


6. To filter the view object instance so that you specify the root value of the hierarchy, select the view object instance you added and click **Edit**.

7. In the Edit View Instance dialog, shuttle the view criteria you created to the **Selected** list and enter the bind parameter value that corresponds to the root of the hierarchy.

   Figure 5–33 shows the view object `ByEmployeeId` view criteria with the bind parameter `TheEmployeeId` set to the value `100` corresponding to the employee that is at the highest level of the hierarchy.

Figure 5–33 View Criteria Filters View Instance



8. Click **OK**.

5.7.2 What Happens When You Create a Recursive Master-Detail Hierarchy

When you create an self-referential view link, JDeveloper creates the XML component definition file that represents its declarative settings and saves it in the directory that corresponds to the name of its package. This XML file contains the declarative information about the source and target attribute pairs that the association you selected specifies and contains the declarative information about the association that relates the source and target view object you selected.

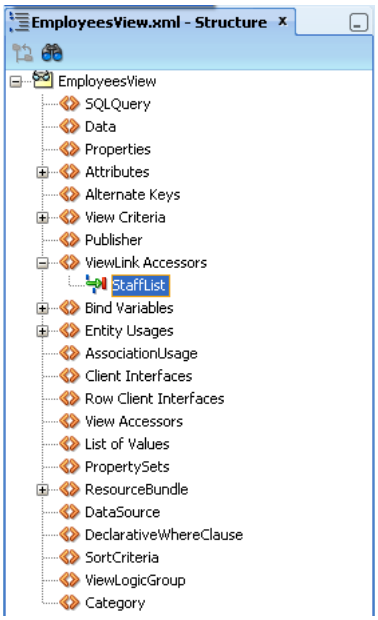
Example 5–9 shows how the `EmpManagerFkLink` defines the same view object `EmployeesView` for the source and destination in its XML component definition file.

Example 5–9 Self-Referential View Link Defined in XML

```
<ViewLink
  xmlns="http://xmlns.oracle.com/bc4j"
  Name="EmpManagerFkLink"
  EntityAssociation="test.model.EmpManagerFkAssoc">
  <ViewLinkDefEnd
    Name="EmployeesView1"
    Cardinality="1"
    Owner="test.model.EmployeesView"
    Source="true">
    <DesignTime>
      <Attr Name="_finderName" Value="ManagerIdEmployeesView"/>
      <Attr Name="_isUpdateable" Value="true"/>
    </DesignTime>
    <AttrArray Name="Attributes">
      <Item Value="test.model.EmployeesView.EmployeeId"/>
    </AttrArray>
  </ViewLinkDefEnd>
  <ViewLinkDefEnd
    Name="EmployeesView2"
    Cardinality="-1"
    Owner="test.model.EmployeesView">
    <DesignTime>
      <Attr Name="_finderName" Value="StaffList"/>
      <Attr Name="_isUpdateable" Value="true"/>
    </DesignTime>
    <AttrArray Name="Attributes">
      <Item Value="test.model.EmployeesView.ManagerId"/>
    </AttrArray>
  </ViewLinkDefEnd>
</ViewLink>
```

In addition to saving the view link component definition itself, JDeveloper also updates the XML definition of the view object to add information about the view link accessor you've defined. As a confirmation of this, you can select the view object in the Application Navigator and inspect its details in the Structure window. As shown in Figure 5–34, you can see the defined accessor under the **ViewLink Accessors** node for the `EmployeesView` view object of the `EmpManagerFkLink` view link.

Figure 5–34 View Object with View Link Accessor in the Structure Window



5.8 Working with View Objects in Declarative SQL Mode

At runtime, when ADF Business Components works with JDBC to pass a query to the database and retrieve the result, the mechanism to retrieve the data is the SQL query. As an alternative to creating view objects that specify a SQL statement at design time, you can create entity-based view objects that contain no SQL statements. This capability of the ADF Business Components design time and runtime is known as *declarative SQL mode*. When the data model developer works with the wizard or editor for a view object in declarative SQL mode, they require no knowledge of SQL. In declarative SQL mode, the view object's metadata causes the ADF Business Components runtime to generate the SQL query statements as follows:

- Optionally, generates `SELECT` and `FROM` lists based on the rendered web page's databound UI components' usage of one or more entity objects' attributes

  Specifying the runtime query statement based solely on databound UI component attribute usage is an optimization that you control at the level of each view object attribute by changing the attribute's `IsSelected` property setting. By default, the property setting is `IsSelected=true` for each attribute that you add to the view object in declarative SQL mode. The default setting specifies the added attribute will be selected in the SQL statement regardless of whether or not the attribute is exposed in the UI by a databound component. For details about changing the property setting to optimize the runtime query statement, see Section 5.8.1, "How to Create SQL-Independent View Objects with Declarative SQL Mode."

- Optionally, generates a `WHERE` clause based on a view criteria that you add to the view object definition

- Optionally, generates an `ORDERBY` clause based on a sort criteria that you add to the view object definition.

- Optionally, augments the `WHERE` clause to support table joins based on named view criteria that you add to the view object definition

- Optionally, augments the `WHERE` clause to support master-detail view filtering based on a view criteria that you add to either the source or destination of a view link definition

Additionally, the SQL statement that a declarative SQL mode view object generates at runtime will be determined by the SQL platform specified in the Business Components page of the Project Properties dialog.

> **Note:** Currently, the supported platforms for runtime SQL generation are SQL92 (ANSI) style and Oracle style. For information about setting the SQL platform for your project, see Section 3.3.1, "Choosing a Connection, SQL Platform, and Data Type Map."

Declarative SQL mode selection is supported in JDeveloper as a setting that you can apply either to the entire data model project or to individual view objects that you create. The ADF Business Components design time also allows you to override the declarative SQL mode project-level setting for any view object you create.

The alternatives to declarative SQL mode are normal mode and expert mode. When you work in either of those modes, the view object definitions you create at design time always contain the entire SQL statement based on the SQL platform required by your application module's defined database connection. Thus the capability of SQL independence does not apply to view objects that you create in normal or expert mode. For information about using the wizard and editor to customize view objects when SQL is desired at design time, see Section 5.2, "Populating View Object Rows from a Single Database Table."

5.8.1 How to Create SQL-Independent View Objects with Declarative SQL Mode

All view objects that you create in JDeveloper rely on the same design time wizard and editor. However, when you enable declarative SQL mode, the wizard and editor change to support customizing the view object definition without requiring you to display or enter any SQL. For example, the Query page of the Create View Object wizard with declarative SQL mode enabled lacks the **Generated SQL** field present in normal mode.

Additionally, in declarative SQL mode, since the wizard and editor do not allow you to enter `WHERE` and `ORDERBY` clauses, you provide equivalent functionality by defining a view criteria and sort criteria respectively. In declarative SQL mode, these criteria appear in the view object metadata definition and will be converted at runtime to their corresponding SQL clause. When the databound UI component has no need to display filtered or sorted data, you may omit the view criteria or sort criteria from the view object definition.

Otherwise, after you enable declarative SQL mode, the basic procedure to create a view object with ensured SQL independence is the same as you would follow to create any entity-based view object. For example, you must still ensure that your view object encapsulates the desired entity object metadata to support the intended runtime query. As with any entity-based view object, the columns of the runtime-generated `FROM` list must relate to the attributes of one or more of the view object's underlying entity objects. In declarative SQL mode, you automatically fulfill this requirement when working with the wizard or editor when you add or remove the attributes of the entity objects on the view object definition.

If you prefer to optimize the declarative SQL query so that the `SELECT` and `FROM` clauses of the SQL query statement are based solely on whether or not the attributes

you add to the view object are rendered at runtime by a databound UI component, then you must disable the **Selected in Query** checkbox (sets `IsSelected=false` for the view object definition) for all added attributes. By default, the `IsSelected` property is `true` for any attribute that you add to the view object in declarative SQL mode. The default setting means the added attribute will be selected in the SQL statement regardless of whether or not the attribute is exposed by a databound UI component. When you create a new view object in declarative SQL mode, you can use the Attribute Settings page of the Create View Object wizard to change the setting for each attribute. If you need to alter this setting after you generate the view object, you can use the Property Inspector to change the **Selected in Query** property setting for one or more attributes that you select in the Attributes page of the view object editor.

> **Performance Tip:** A view object instance configured to generate SQL statements dynamically will requery the database during page navigation if a subset of all attributes with the same list of key entity objects is used in the subsequent page navigation. Thus performance can be improved by activating a superset of all the required attributes to eliminate a subsequent query execution.

5.8.1.1 Enabling Declarative SQL Mode for All New View Objects

You can enable declarative SQL mode as a global preference so that it is the Create View Object wizard's default mode, or you can leave the setting disabled and select the desired mode directly in the wizard. The editor for a view object also lets you select and change the mode for an existing view object definition.

Before you begin:

It may be helpful to have an understanding of declarative SQL mode. For more information, see Section 5.8.1, "How to Create SQL-Independent View Objects with Declarative SQL Mode."

To enable declarative SQL mode for all new view objects:

1. From the main menu, choose **Tools > Preferences**.
2. In the Preferences dialog, expand the **ADF Business Components** node and choose **View Objects**.
3. On the ADF Business Components: View Object page, select **Enable Declarative SQL mode for new objects** and click **OK**.

   To predetermine how the `FROM` list will be generated at runtime you can select **Include all attributes in runtime-generated query**, as described in Section 5.8.4, "How to Support Programmatic Execution of Declarative SQL Mode View Objects."

5.8.1.2 Enabling Declarative SQL Mode for Specific View Objects

There are no unique requirements for creating entity-based view objects in declarative SQL mode. The basic procedure to create a view object with ensured SQL independence is the same as you would follow to create any entity-based view object. To create an entity-based view object in declarative SQL mode, use the Create View Object wizard, which is available from the New Gallery. The editor for a view object lets you select and change the mode for an existing view object definition.

Before you begin:

It may be helpful to have an understanding of declarative SQL mode. For more information, see Section 5.8.1, "How to Create SQL-Independent View Objects with Declarative SQL Mode."

To create declarative SQL-based view objects:

1. In the Application Navigator, right-click the project in which you want to create the view objects and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **View Object**, and click **OK**.
3. On the Name page, enter a package name and a view object name. Keep the default setting **Entity object** selected to indicate that you want this view object to manage data with its base entity object. Click **Next**.

   Any other choice for the data selection will disable declarative SQL mode in the Create View Object wizard.
4. On the Entity Objects page, select the entity object whose data you want to use in the view object. Click **Next**.

   When you want to create a view object that joins entity objects, you can add secondary entity objects to the list. To create more complex entity-based view objects, see Section 5.5.1, "How to Create Joins for Entity-Based View Objects."
5. On the Attributes page, select at least one attribute from the entity usage in the **Available** list and shuttle it to the **Selected** list. Attributes you do not select will not be eligible for use in view criteria and sort criteria. Click **Next**.

   You should select any attribute that you intend to customize (in the Attribute Settings page) or any attribute that you intend to use in a view criteria or sort criteria (in the Query page). Additionally, the tables that appear in the `FROM` list of the runtime-generated query will be limited to the tables corresponding to the attributes of the entity objects you select.
6. On the Attribute Settings page, optionally, use the **Select Attribute** dropdown list to switch between the view object attributes in order to change their names or any of their initial settings.
7. Use the **Select Attribute** dropdown list to switch between the previously selected view object attributes and deselect **Selected in Query** for each attribute that you want to be selected in the SQL statement based solely on whether or not the attribute is rendered by a databound UI component. Click **Next**.

   By default, the **Selected in Query** checkbox is enabled for all view object attributes that you add in declarative SQL mode. This default setting will generate a SQL statement with all added attributes selected. When you deselect the checkbox for an attribute, the `IsSelected` property is set to `false` and whether or not the attribute is selected will be determined at runtime by the databound UI component's usage of the attribute.
8. On the Query page, select **Declarative** in the **Query Mode** dropdown list if it is not already displayed. The wizard changes to declarative SQL mode.

   If you did not select **Enable declarative SQL mode for new objects**, in the Preferences dialog, the wizard displays the default query mode, **Normal**. Changing the mode to **Declarative** in the wizard allows you to override the default mode for this single view object.

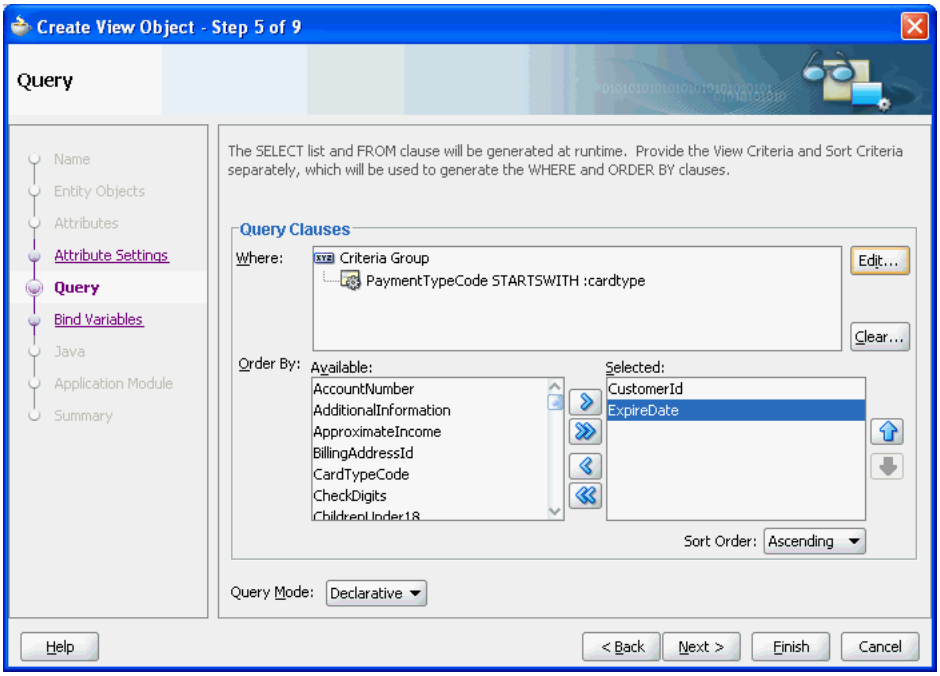
9. Optionally, define **Where** and **Order By** criteria to filter and order the data as required. At runtime, ADF Business Components automatically generates the corresponding SQL statements based on the criteria you create.

   Click **Edit** next to the **Where** field to define the view criteria you will use to filter the data. The view criteria you enter will be converted at runtime to a `WHERE` clause that will be enforced on the query statement. For information about specifying view criteria, see Section 5.11, "Working with Named View Criteria."

   In the **Order By** field select the desired attribute in the **Available** list and shuttle it to the **Selected** list. Attributes you do not select will not appear in the SQL `ORDERBY` clause generated at runtime. Add additional attributes to the **Selected** list when you want the results to be sorted by more than one column. Arrange the selected attributes in the list according to their sort precedence. Then for each sort attribute, assign whether the sort should be performed in ascending or descending order. Assigning the sort order to each attribute ensures that attributes ignored by the UI component still follow the intended sort order.

   For example, as shown in Figure 5–35, to limit the `CustomerCardStatus` view object to display only the rows in the `CUSTOMERS` table for customers with a specific credit card code, the view criteria in the **Where** field limits the `CardTypeCode` attribute to a runtime-determined value. To order the data by customer ID and the customer's card expiration date, the **Order By** field identifies those attributes in the **Selected** list.

Figure 5–35 Creating View Object Wizard, Query Page with Declarative Mode Selected



10. Click **Finish**.

5.8.2 How to Filter Declarative SQL-Based View Objects When Table Joins Apply

When you create an entity-based view object you can reference more than one entity object in the view object definition. In the case of view objects you create in declarative SQL mode, whether the base entity objects are activated from the view object definition will depend on the requirements of the databound UI component at

runtime. If the UI component displays attribute values from multiple entity objects, then the SQL generated at runtime will contain a `JOIN` operation to query the appropriate tables.

Just as with any view object that you create, it is possible to filter the results from table joins by applying named view criteria. In the case of normal mode view objects, all entity objects and their attributes will be referenced by the view object definition and therefore will be automatically included in the view object's SQL statement. However, by delaying the SQL generation until runtime with declarative SQL mode, there is no way to know whether the view criteria should be applied.

Note: In declarative SQL mode, you can define a view criteria to specify the `WHERE` clause (optional) when you create the view object definition. This type of view criteria when it exists will always be applied at runtime. For a description of this usage of the view criteria, see Section 5.8.1, "How to Create SQL-Independent View Objects with Declarative SQL Mode."

Because a SQL `JOIN` may not always result from a view object defined in declarative SQL mode with multiple entity objects, named view criteria that you define to filter query results should be applied conditionally at runtime. In other words, named view criteria that you create for declarative SQL-based view objects need not be applied as required, automatic filters. To support declarative SQL mode, named view criteria that you apply to a view object created in declarative SQL mode can be set to apply only on the condition that the UI component is bound to the attributes referenced by the view criteria. The named view criteria once applied will, however, support the UI component's need to display a filtered result set.

Before you begin:

It may be helpful to have an understanding of SQL independence at runtime. For more information, see Section 5.8, "Working with View Objects in Declarative SQL Mode."

You will need to complete these tasks:

- Create the view object with declarative SQL mode enabled as described in Section 5.8.1, "How to Create SQL-Independent View Objects with Declarative SQL Mode."
- Create a view criteria to filter the results from table joins as described in Section 5.11.1, "How to Create Named View Criteria Declaratively."

  You will use the Edit View Criteria dialog to create the named view criteria. You then enable its conditional usage by setting the **appliedIfJoinSatisfied** property in the Property Inspector.

To define a view criteria to filter only when the join is satisfied:

1. In the Application Navigator, double-click the SQL-independent view object that supports table joins.
2. In the overview editor, click the **Query** navigation tab.
3. In Query page, expand the **View Criteria** section and select the view criteria you created to filter the SQL-independent view object.
4. With the view criteria selected, open the Property Inspector and set the **AppliedIfJoinSatisfied** property to **true**.

The property value **true** means you want the view criteria to be applied only on the condition that the UI component requires the attributes referenced by the view criteria. The default value **false** means that the view criteria will automatically be applied at runtime. In the case of declarative SQL mode-based view objects, the value **true** ensures that the query filter will be appropriate to needs of the view object's databound UI component.

5.8.3 How to Filter Master-Detail Related View Objects with Declarative SQL Mode

Just as with normal mode view objects, you can link view objects that you create in declarative SQL mode to other view objects to form master-detail hierarchies of any complexity. The steps to create the view links are the same as with any other entity-based view object, as described in Section 5.6.2, "How to Create a Master-Detail Hierarchy for Entity-Based View Objects." However, in the case of view objects that you create in declarative SQL mode, you can further refine the view object results in the Source SQL or Destination SQL dialog for the view link by selecting a previously defined view criteria in the Create View Link wizard or the overview editor for the view link.

Before you begin:

It may be helpful to have an understanding of SQL independence at runtime. For more information, see Section 5.8, "Working with View Objects in Declarative SQL Mode."

You will need to complete these tasks:

1. Create the master and detail view objects with declarative SQL mode enabled as described in Section 5.8.1, "How to Create SQL-Independent View Objects with Declarative SQL Mode."
2. Define the desired view criteria for either the source (master) view object or the destination (detail) view object as described in Section 5.8.2, "How to Filter Declarative SQL-Based View Objects When Table Joins Apply."
3. Create the view link as described in Section 5.6.2, "How to Create a Master-Detail Hierarchy for Entity-Based View Objects."

To filter a view link source or view link destination:

1. In the Application Navigator, double-click the view link you created to form the master-detail hierarchy:
2. In the overview editor navigation list, select **Query** and expand the **Source** or **Destination** sections.
3. In the **View Criteria** dropdown list, select the desired view criteria. The dropdown list will be empty if no view criteria exist for the view object.

   If the overview editor does not display a dropdown list for view criteria selection, then the view objects you selected for the view link were not created in declarative SQL mode. For view objects created in normal or expert mode, you must edit the `WHERE` clause to filter the data as required.

5.8.4 How to Support Programmatic Execution of Declarative SQL Mode View Objects

Typically, when you define a declarative SQL mode view object, the attributes that get queried at runtime will be determined by the requirements of the databound UI component as it is rendered in the web page. This is the runtime-generation capability that makes view objects independent of the design time database's SQL platform. However, you may also need to execute the view object programmatically without

exposing it to an ADF data binding in the UI. In this case, you can enable the **Include all attributes in runtime-generated query** option to ensure that a programmatically executed view object has access to all of the entity attributes.

> **Note:** Be careful to limit the use of the **Include all attributes in runtime-generated query** option to programmatically executed view objects. If you expose the view object with this setting enabled to a databound UI component, the runtime query will include all attributes.

The **Include all attributes in runtime-generated query** option can be specified as a global preference setting or as a setting on individual view objects. Both settings may be used in these combinations:

- Enable the global preference so that every view object you create includes all attributes in the runtime query statement.
- Enable the global preference, but disable the setting on view objects that will not be executed programmatically and therefore should not include all attributes in the runtime query statement.
- Disable the global preference (default), but enable the setting on view objects that will be executed programmatically and therefore should include all attributes in the runtime query statement.

5.8.4.1 Forcing Attribute Queries for All Declarative SQL Mode View Objects

You can enable the global preference to force attribute queries for all declarative SQL mode view objects as the default mode, or you can leave the option disabled and enable the option directly in the overview editor for a specific view object.

Before you begin:

It may be helpful to have an understanding of how the **Include all attributes in runtime-generated query** option supports programmatic execution of declarative SQL mode view objects. For more information, see Section 5.8.4, "How to Support Programmatic Execution of Declarative SQL Mode View Objects."

To set the global preference to include all attributes in the query:

1. From the main menu, choose **Tools > Preferences**.
2. In the Preferences dialog, expand **ADF Business Components** and select **View Objects**.
3. On the ADF Business Components: View Object page, select **Enable Declarative SQL mode for new objects**.
4. Select **Include all attributes in runtime-generated query** to force all attributes of the view object's underlying entity objects to participate in the query and click **OK**.

   Enabling this option sets a flag in the view object definition but you will still need to add entity object selections and entity object attribute selections to the view object definition.

5.8.4.2 Forcing Attribute Queries for Specific Declarative SQL Mode View Objects

You can change the view object setting in the **Tuning** section of the overview editor's General page. The overview editor only displays the **Include all attributes in**

runtime-generated query option if you have created the view object in declarative SQL mode.

Before you begin:

It may be helpful to have an understanding of how the **Include all attributes in runtime-generated query** option supports programmatic execution of declarative SQL mode view objects. For more information, see Section 5.8.4, "How to Support Programmatic Execution of Declarative SQL Mode View Objects."

To set the view object-specific preference to include all attributes in the query:

1. When you want to force all attributes for specific view objects, create the view object in the Create View Object wizard and be sure that you have enabled declarative SQL mode.

   You can verify this in the overview editor. In the overview editor, click the **Query** navigation tab and click the **Edit SQL Query** button along the top of the page. In the Edit Query dialog, verify that the **SQL Mode** dropdown list shows the selection **Declarative**.

2. In the overview editor, click the **General** navigation tab.

3. In the General page, expand the **Tuning** section and select **Include all attributes in runtime-generated query**.

   Enabling this option forces all attributes of the view object's underlying entity objects to participate in the query. When enabled, it sets a flag in the view object definition but you will still need to add entity object selections and entity object attribute selections to the view object definition.

5.8.5 What Happens When You Create a View Object in Declarative SQL Mode

When you create the view object in declarative SQL mode, three properties get added to the view object's metadata: `SelectListFlags`, `FromListFlags`, and `WhereFlags`. Properties that are absent in declarative SQL mode are the normal mode view object's `SelectList`, `FromList`, and `Where` properties, which contain the actual SQL statement (or, for expert mode, the `SQLQuery` element). Example 5–10 shows the three view object metadata flags that get enabled in declarative SQL mode to ensure that SQL will be generated at runtime instead of specified as metadata in the view object's definition.

Example 5–10 View Object Metadata with Declarative SQL Mode Enabled

```
<ViewObject
  xmlns="http://xmlns.oracle.com/bc4j"
  Name="CustomerCardStatus"
  SelectListFlags="1"
  FromListFlags="1"
  WhereFlags="1"
   ...
```

Similar to view objects that you create in either normal or expert mode, the view object metadata also includes a `ViewAttribute` element for each attribute that you select in the Attribute page of the Create View Object wizard. However, in declarative SQL mode, when you "select" attributes in the wizard (or add an attribute in the overview editor), you are not creating a `FROM` or `SELECT` list in the design time. The attribute definitions that appear in the view object metadata only determine the list of potential entities and attributes that will appear in the runtime-generated statements. For

information about how ADF Business Components generates these SQL lists, see Section 5.8.6, "What Happens at Runtime: Declarative SQL Mode Queries."

Example 5–11 shows the additional features of declarative SQL mode view objects, including the optional declarative `WHERE` clause (`DeclarativeWhereClause` element) and the optional declarative `ORDERBY` clause (`SortCriteria` element).

Example 5–11 View Object Metadata: Declarative View Criteria and Sort Criteria

```
<DeclarativeWhereClause>
    <ViewCriteria
        Name="CustomerStatusWhereCriteria"
        ViewObjectName="oracle.fodemo.storefront.store.queries.CustomerCardStatus"
        Conjunction="AND"
        Mode="3"
        AppliedIfJoinSatisfied="false">
        <ViewCriteriaRow
            Name="vcrow60">
            <ViewCriteriaItem
                Name="CardTypeCode"
                ViewAttribute="CardTypeCode"
                Operator="STARTSWITH"
                Conjunction="AND"
                Required="Optional">
            <ViewCriteriaItemValue
                Value=":cardtype"
                IsBindVarValue="true"/>
            </ViewCriteriaItem>
        </ViewCriteriaRow>
    </ViewCriteria>
    </DeclarativeWhereClause>
    <SortCriteria>
        <Sort
        Attribute="CustomerId"/>
    <Sort
        Attribute="CardTypeCode"/>
    </SortCriteria>
```

5.8.6 What Happens at Runtime: Declarative SQL Mode Queries

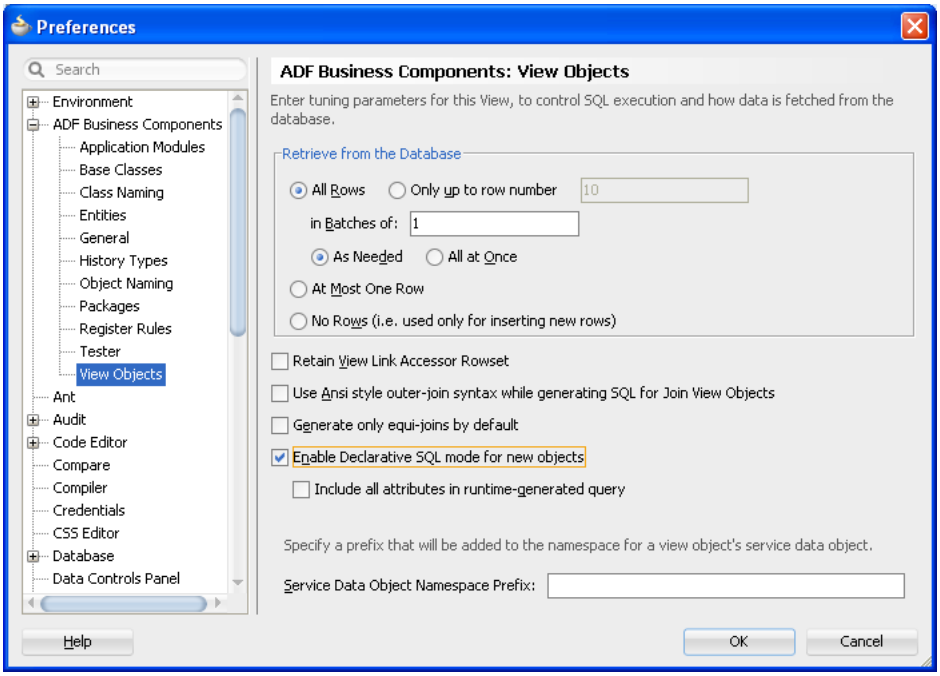
At runtime, when a declarative SQL mode query is generated, ADF Business Components determines which attributes were defined from the metadata `ViewCriteria` element and `SortCriteria` element. It then uses these attributes to generate the `WHERE` and `ORDERBY` clauses. Next, the runtime generates the `FROM` list based on the tables corresponding to the entity usages defined by the metadata `ViewAttribute` elements. Finally, the runtime builds the `SELECT` statement based on the attribute selection choices the end user makes in the UI. As a result, the view object in declarative SQL mode generates all SQL clauses entirely at runtime. The runtime-generated SQL statements will be based on the platform that appears in the project properties setting. Currently, the runtime supports SQL92 (ANSI) style and Oracle style platforms.

5.8.7 What You May Need to Know About Overriding Declarative SQL Mode Defaults

JDeveloper lets you control declarative SQL mode for all new view objects you add to your data model project or for individual view objects you create or edit. These settings may be used in these combinations:

- Enable the global preference in the Preferences dialog (select **Tools > Preferences**). Every view object you create will delay SQL generation until runtime. Figure 5–36 shows the global preference **Enable declarative SQL for new objects** set to enabled.
- Enable the global preference in the Preferences dialog, but change the SQL mode for individual view objects. In this case, unless you change the SQL mode, the view objects you create will delay SQL generation until runtime.
- Disable the global preference (default) in the Preferences dialog, but select declarative SQL mode for individual view objects. In this case, unless you change the SQL mode, view objects you create will contain SQL statements.

Figure 5–36 Preferences Dialog with Declarative SQL Mode Enabled



To edit the SQL mode for a view object you have already created, open the Query page in the Edit Query dialog and select **Declarative** from the **SQL Mode** dropdown list. To display the Edit Query dialog, open the view object in the overview editor, select **Query** from the navigation list and click the **Edit SQL Query** button. The same option appears in the Query page of the Create View Object wizard.

5.8.8 What You May Need to Know About Working Programmatically with Declarative SQL Mode View Objects

As a convenience to developers, the view object implementation API allows individual attributes to be selected and deselected programmatically. This API may be useful in combination with the view objects you create in declarative SQL mode and intend to execute programmatically. Example 5–12 shows how to call `selectAttributeDefs()` on the view object when you want to add a subset of attributes to those already configured with SQL mode enabled.

Example 5–12 ViewObjectImpl API with SQL Mode View Objects

```
ApplicationModule am = Configuration.createRootApplicationModule(amDef, config);
ViewObjectImpl vo = (ViewObjectImpl) am.findViewObject("CustomerVO");
```

```
vo.resetSelectedAttributeDefs(false);
vo.selectAttributeDefs(new String[] {"FirstName, "LastName"});
vo.executeQuery();
```

The call to `selectAttributeDefs()` adds the attributes in the array to a private member variable of `ViewObjectImpl`. A call to `executeQuery()` transfers the attributes in the private member variable to the actual select list. It is important to understand that these `ViewObjectImpl` attribute calls are not applicable to the client layer and are only accessible inside the `Impl` class of the view object on the middle tier.

Additionally, you might call `unselectAttributeDefs()` on the view object when you want to deselect a small subset of attributes after *enabling* the **Include all attributes in runtime-generated query** option. Alternatively, you can call `selectAttributeDefs()` on the view object to select a small subset of attributes after *disabling* the **Include all attributes in runtime-generated query** option.

Caution: Be careful not to expose a declarative SQL mode view object executed with this API to the UI since only the value of the **Include all attributes in runtime-generated query** option will be honored.

5.9 Working with View Objects in Expert Mode

When defining entity-based view objects, you can fully specify the `WHERE` and `ORDER BY` clauses, whereas, by default, the `FROM` clause and `SELECT` list are automatically derived. The names of the tables related to the participating entity usages determine the `FROM` clause, while the `SELECT` list is based on the:

- Underlying column names of participating entity-mapped attributes
- SQL expressions of SQL-calculated attributes

When you require full control over the `SELECT` or `FROM` clause in a query, you can enable expert mode.

Tips: The view object editors and wizard in the JDeveloper provide full support for generating SQL from choices that you make. For example, two such options allow you to declaratively define outer joins and work in declarative SQL mode (where no SQL is generated until runtime).

5.9.1 How to Customize SQL Statements in Expert Mode

To enable expert mode, select **Expert Mode** from the **SQL Mode** dropdown list on the Query panel of the Create View Object wizard. You can also modify the SQL statement of an existing entity-based view object in the view object overview editor. In the overview editor, navigate to the Query page and click the **Edit SQL Query** button. In the Edit Query dialog, select **Expert Mode** from the **SQL Mode** dropdown list.

5.9.2 How to Name Attributes in Expert Mode

If your SQL query includes a calculated expression, use a SQL alias to assist the Create View Object wizard in naming the column with a Java-friendly name. Example 5–13 shows a SQL query that includes a calculated expression.

Example 5–13 SQL Query with Calculated Expression

```
select PERSON_ID, EMAIL,
       SUBSTR(FIRST_NAME,1,1)||'. '||LAST_NAME
from PERSONS
order by EMAIL
```

Example 5–14 uses a SQL alias `USER_SHORT_NAME` to assist the Create View Object wizard in naming the column with a Java-friendly name. The wizard will display **UserShortName** as the name of the attribute derived from this calculated expression.

Example 5–14 SQL Query with SQL Alias

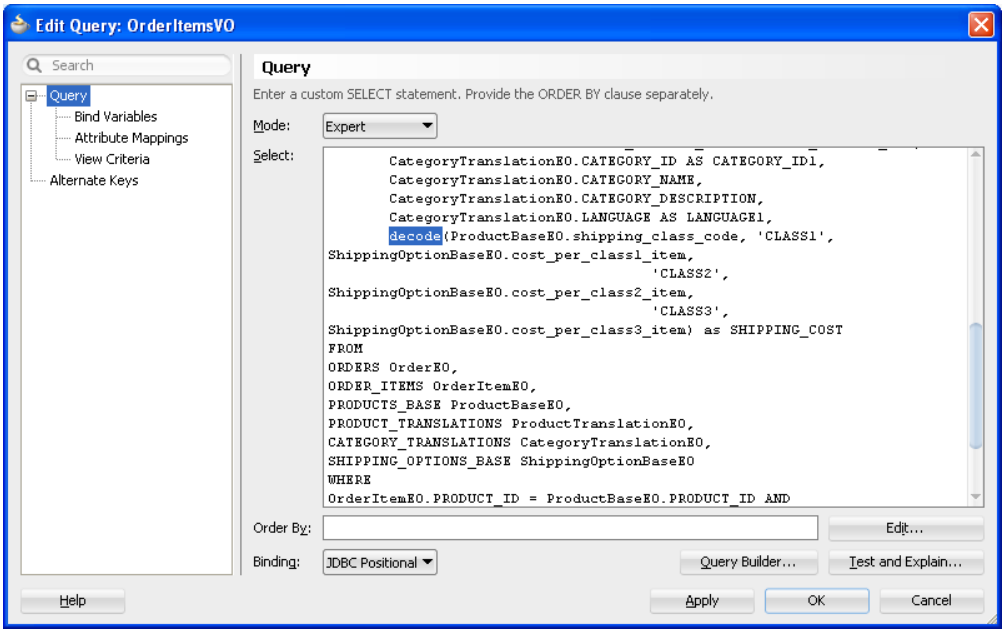
```
select PERSON_ID, EMAIL,
       SUBSTR(FIRST_NAME,1,1)||'. '||LAST_NAME AS USER_SHORT_NAME
from PERSONS
order by EMAIL
```

5.9.3 What Happens When You Enable Expert Mode

When you enable expert mode, the read-only **Generated Statement** section of the Query page becomes a fully editable **Query Statement** text box, displaying the full SQL statement. Using this text box, you can change every aspect of the SQL query.

For example, Figure 5–37 shows the Query page of the Edit Query dialog for the `OrderItems` view object. It's an expert mode, entity-based view object that references a PL/SQL function `decode` that obtains its input values from an expression set on the `ShippingCost` attribute.

Figure 5–37 OrderItems Expert Mode View Object



5.9.4 What You May Need to Know About Expert Mode

When you define a SQL query using expert mode in the Edit Query dialog, you type a SQL language statement directly into the editor. Using this mode places some responsibility on the Business Components developer to understand how the view

object handles the metadata resulting from the query definition. Review the following information to familiarize yourself with the behavior of the Edit Query dialog that you use in expert mode.

5.9.4.1 Expert Mode Provides Limited Attribute Mapping Assistance

The automatic cooperation of a view object with its underlying entity objects depends on correct attribute-mapping metadata saved in the XML component definition. This information relates the view object attributes to corresponding attributes from participating entity usages. JDeveloper maintains this attribute mapping information in a fully automatic way for normal entity-based view objects. However, when you decide to use expert mode with a view object, you need to pay attention to the changes you make to the `SELECT` list. That is the part of the SQL query that directly relates to the attribute mapping. Even in expert mode, JDeveloper continues to offer some assistance in maintaining the attribute mapping metadata when you do the following to the `SELECT` list:

- Reorder an expression without changing its column alias

  JDeveloper reorders the corresponding view object attribute and maintains the attribute mapping.

- Add a new expression
- JDeveloper adds a new SQL-calculated view object attribute with a corresponding camel-capped name based on the column alias of the new expression.
- Remove an expression

  JDeveloper converts the corresponding SQL-calculated or entity-mapped attribute related to that expression to a transient attribute.

However, if you rename a column alias in the `SELECT` list, JDeveloper has no way to detect this, so it is treated as if you removed the old column expression and added a new one of a different name.

After making any changes to the `SELECT` list of the query, visit the Attribute Mappings page to ensure that the attribute-mapping metadata is correct. The table on this page, which is disabled for view objects in normal mode, becomes enabled for expert mode view objects. For each view object attribute, you will see its corresponding SQL column alias in the table. By clicking into a cell in the **View Attributes** column, you can use the dropdown list that appears to select the appropriate entity object attribute to which any entity-mapped view attributes should correspond.

> **Note:** If the view attribute is SQL-calculated or transient, a corresponding attribute with a "SQL" icon appears in the **View Attributes** column to represent it. Since neither of these type of attributes are related to underlying entity objects, there is no entity attribute related information required for them.

5.9.4.2 Expert Mode Drops Custom Edits

When you disable expert mode for a view object, it will return to having its `SELECT` and `FROM` clause be derived again. JDeveloper warns you that doing this might cause your custom edits to the SQL statement to be lost. If this is what you want, after acknowledging the alert, your view object's SQL query reverts back to the default.

5.9.4.3 Expert Mode Ignores Changes to SQL Expressions

Consider a `Products` view object with a SQL-calculated attribute named `Shortens` whose SQL expression you defined as `SUBSTR(NAME,1,10)`. If you switch this view object to expert mode, the **Query Statement** box will show a SQL query similar to the one shown in Example 5–15.

Example 5–15 SQL-Calculated Attribute Expression in Expert Mode

```
SELECT Products.PROD_ID,
       Products.NAME,
       Products.IMAGE,
       Products.DESCRIPTION,
       SUBSTR(NAME,1,10) AS SHORT_NAME
FROM PRODUCTS Products
```

If you go back to the attribute definition for the `Shortens` attribute and change the **SQL Expression** field from `SUBSTR(NAME,1,10)` to `SUBSTR(NAME,1,15)`, then the change will be saved in the view object's XML component definition. Note, however, that the SQL query in the **Query Statement** box will remain as the original expression. This occurs because JDeveloper never tries to *modify* the text of an expert mode query. In expert mode, the developer is in full control. JDeveloper attempts to adjust metadata as a result of some kinds of changes you make yourself to the expert mode SQL statement, but it does not perform the reverse. Therefore, if you change view object metadata, the expert mode SQL statement is not updated to reflect it.

Therefore, you need to update the expression in the expert mode SQL statement itself. To be completely thorough, you should make the change *both* in the attribute metadata *and* in the expert mode SQL statement. This would ensure — if you (or another developer on your team) ever decides to toggle expert mode *off* at a later point in time — that the automatically derived `SELECT` list would contain the correct SQL-derived expression.

> **Note:** If you find you had to make numerous changes to the view object metadata of an expert mode view object, you can avoid having to manually translate any effects to the SQL statement by copying the text of your customized query to a temporary backup file. Then, you can disable expert mode for the view object and acknowledge the warning that you will lose your changes. At this point JDeveloper will rederive the correct generated SQL statement based on all the new metadata changes you've made. Finally, you can enable expert mode once again and reapply your SQL customizations.

5.9.4.4 Expert Mode Returns Error for SQL Calculations that Change Entity Attributes

When changing the `SELECT` list expression that corresponds to *entity-mapped* attributes, don't introduce SQL calculations into SQL statements that change the value of the attribute when retrieving the data. To illustrate the problem that will occur if you do this, consider the query for a simple entity-based view object named `Products` shown in Example 5–16.

Example 5–16 Query Statement Without SQL-Calculated Expression

```
SELECT Products.PROD_ID,
       Products.NAME,
       Products.IMAGE,
```

```
       Products.DESCRIPTION
FROM PRODUCTS Products
```

Imagine that you wanted to limit the name column to display only the first ten characters of the name of a product query. The correct way to do that would be to introduce a new SQL-calculated field, such as `ShortName` with an expression like `SUBSTR(Products.NAME,1,10)`. One way you should *avoid* doing this is to switch the view object to expert mode and change the `SELECT` list expression for the entity-mapped NAME column to the include the SQL-calculate expression, as shown in Example 5–17.

Example 5–17 Query Statement with SQL-Calculated Expression

```
SELECT Products.PROD_ID,
       SUBSTR(Products.NAME,1,10) AS NAME,
       Products.IMAGE,
       Products.DESCRIPTION
FROM PRODUCTS Products
```

This alternative strategy would initially appear to work. At runtime, you see the truncated value of the name as you are expecting. However, if you modify the row, when the underlying entity object attempts to lock the row it does the following:

- Issues a `SELECT FOR UPDATE` statement, retrieving all columns as it tries to lock the row.
- If the entity object successfully locks the row, it compares the original values of all the persistent attributes in the entity cache as they were last retrieved from the database with the values of those attributes just retrieved from the database during the lock operation.
- If any of the values differs, then the following error is thrown:

  ```
  (oracle.jbo.RowInconsistentException)
  JBO-25014: Another user has changed the row with primary key [...]
  ```

If you see an error like this at runtime even though you are the *only* user testing the system, it is most likely due to your inadvertently introducing a SQL function in your expert mode view object that changed the selected value of an entity-mapped attribute. In Example 5–17, the `SUBSTR(Products.NAME,1,10)` function introduced causes the original selected value of the `Name` attribute to be truncated. When the row-lock SQL statement selects the value of the `NAME` column, it will select the entire value. This will cause the comparison shown in Example 5–17 to fail, producing the "phantom" error that another user has changed the row.

The same thing would happen with `NUMBER`-valued or `DATE`-valued attributes if you inadvertently apply SQL functions in expert mode to truncate or alter their retrieved values for entity-mapped attributes.

Therefore, if you need to present altered versions of entity-mapped attribute data, introduce a new SQL-calculated attribute with the appropriate expression to handle the task.

5.9.4.5 Expert Mode Retains Formatting of SQL Statement

When you change a view object to expert mode, its XML component definition changes from storing parts of the query in separate XML attributes, to saving the entire query in a single `<SQLQuery>` element. The query is wrapped in an XML `CDATA` section to preserve the line formatting you may have done to make a complex query be easier to understand.

5.9.4.6 Expert Mode Wraps Queries as Inline Views

If your expert-mode view object:

- Contains a `ORDERBY` clause specified in the **Order By** field of the Query Clauses page at design time, or
- Has a dynamic `WHERE` clause or `ORDERBY` clause applied at runtime using `setWhereClause()` or `setOrderByClause()`

then its query gets nested into an inline view before applying these clauses. For example, suppose your expert mode query was defined like the one shown in Example 5–18.

Example 5–18 Expert Mode Query Specified At Design Time

```
select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from PERSONS
union all
select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from INACTIVE_PERSONS
```

Then, at runtime, when you set an additional `WHERE` clause like `email = :TheUserEmail`, the view object nests its original query into an inline view like the one shown in Example 5–19.

Example 5–19 Runtime-Generated Query with Inline Nested Query

```
SELECT * FROM(
select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from PERSONS
union all
select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from INACTIVE_PERSONS) QRSLT
```

And, the view object adds the dynamic `WHERE` clause predicate at the end, so that the final query the database sees looks like the one shown in Example 5–20.

Example 5–20 Runtime-Generated Query with Dynamic WHERE Clause

```
SELECT * FROM(
select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from PERSONS
union all
select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from INACTIVE_PERSONS) QRSLT
WHERE email = :TheUserEmail
```

This query "wrapping" is necessary in general for expert mode queries, because the original query could be arbitrarily complex, including SQL `UNION`, `INTERSECT`, `MINUS`, or other operators that combine multiple queries into a single result. In those cases, simply "gluing" the additional runtime `WHERE` clause onto the end of the query text could produce unexpected results. For example, the clause might apply only to the *last* of several `UNION`'ed statements. By nesting the original query verbatim into an inline view, the view object guarantees that your additional `WHERE` clause is correctly used to filter the results of the original query, regardless of how complex it is.

5.9.4.7 Limitation of Inline View Wrapping at Runtime

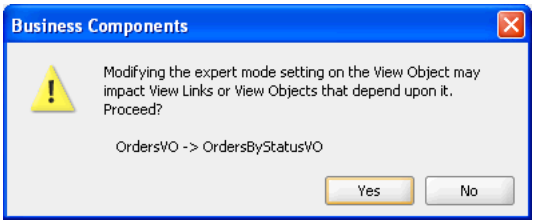
Inline view wrapping of expert mode view objects, limits a dynamically added `WHERE` clause to refer only to columns in the `SELECT` list of the original query. To avoid this

limitation, when necessary you can disable the use of the inline view wrapping by calling `setNestedSelectForFullSql(false)`.

5.9.4.8 Expert Mode Changes May Affect Dependent Objects

When you modify a view object query to be in expert mode after you have already created the view links that involve that view object or after you created other view objects that extend the view object, JDeveloper will warn you with the alert shown in Figure 5–38. The alert reminds you that you should revisit these dependent components to ensure their SQL statements still reflect the correct query.

Figure 5–38 Proactive Reminder to Revisit Dependent Components



For example, if you were to modify the `OrdersVO` view object to use expert mode, because the `OrdersByStatusVO` view object extends it, you need to revisit the extended component to ensure that its query still logically reflects an extension of the modified parent component.

5.10 Working with Bind Variables

Bind variables provide you with the means to supply attribute values at runtime to the view object or view criteria. All bind variables are defined at the level of the view object and used in one of the following ways:

- You can type the bind variable directly into the `WHERE` clause of your view object's query to include values that might change from execution to execution. In this case, bind variables serve as placeholders in the SQL string whose value you can easily change at runtime without altering the text of the SQL string itself. Since the query doesn't change, the database can efficiently reuse the same parsed representation of the query across multiple executions, which leads to higher runtime performance of your application.
- You can select the bind variable from a selection list to define the attribute value for a view criteria in the Edit View Criteria dialog you open on the view object. In this case, the bind variables allow you to change the values for attributes you will use to filter the view object row set. For more information about filtering view object row sets, see Section 5.11, "Working with Named View Criteria."

  If the view criteria is to be used in a seeded search, you have the option of making the bind variable updatable by the end user. With this updatable option, end users will be expected to enter the value in a search form corresponding to the view object query.

Bind variables that you add to a `WHERE` clause require a valid value at runtime, or a runtime exception error will be thrown. In contrast, view criteria execution need not require the bind variable value if the view criteria item for which the bind variable is assigned is not required. To enforce this desired behavior, the Bind Variable dialog lets you can specify whether a bind variable is required or not.

You can define a default value for the bind variable or write scripting expressions for the bind variable that includes dot notation access to attribute property values. Expressions are based on the Groovy scripting language, as described in Section 3.6, "Overview of Groovy Scripting Language Support."

5.10.1 How to Add Bind Variables to a View Object Definition

To add a named bind variable to a view object, use the Query page of the overview editor for the view object. You can define as many bind variables as you need.

Before you begin:

It may be helpful to have an understanding of the support for bind variables at the level of view objects. For more information, see Section 5.10, "Working with Bind Variables."

You will need to complete this task:

Create the desired view objects as described in Section 5.2.1, "How to Create an Entity-Based View Object," and Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object."

To define a named bind variable:

1. In the Application Navigator, double-click the view object.
2. In the overview editor, click the **Query** navigation tab.
3. In the Query page, expand the **Bind Variables** section and click the **Create new bind variable** button.
4. In the Bind Variable dialog, enter the name and data type for the new bind variable.

   Because the bind variables share the same namespace as view object attributes, specify names that don't conflict with existing view object attribute names. As with view objects attributes, by convention bind variable names are created with an initial capital letter, but you can rename it as desired.
5. Optionally, specify a default value for the bind variable:
   - When you want the value to be determined at runtime using an expression, enter a Groovy scripting language expression, select the **Expression** value type and enter the expression in the **Value** field. Optionally, click **Edit** to open the Expression dialog. The Expression dialog gives you a larger text area to write the expression. For example, you might want to define a bind variable to filter view instances based on the current user, as described in Section 5.10.2, "How to Reference the Current User in a Named Bind Variable Using Groovy."
   - When you want to define a default value, select the **Literal** value type and enter the literal value in the **Value** field.
6. Decide on one of the following runtime usages for the bind variable:
   - When you want the value to be supplied to a SQL `WHERE` clause using a bind variable in the clause, select the **Required** checkbox. This ensures that a runtime exception will be thrown if the value is not supplied. For more information, see Section 5.10.8.3, "Errors Related to Naming Bind Variables."
   - When you want the value to be supplied to a view criteria using a bind variable in the view criteria, only select the **Required** checkbox when you need to reference the same bind variable in a SQL `WHERE` clause or when you want to use the bind variable as the assigned value of a view criteria item that

is specifically defined as required by a view criteria that is applied to a view object. When **Required** is unselected this ensures that the value is optional and that no runtime exception will be thrown if the bind variable is not resolved. For example, view criteria with bind variables defined can be used to create Query-by-Example search forms in the user interface. For more information, see Section 5.11, "Working with Named View Criteria."

7. In the Bind Variable dialog, click the **Control Hints** tab and specify hints like **Label Text**, **Format Type**, **Format** mask, and others.

   The view layer will use bind variable control hints when you build user interfaces like search pages that allow the user to enter values for the named bind variables. The **Updatable** checkbox controls whether the end user will be allowed to change the bind variable value through the user interface. If a bind variable is not updatable, then its value can only be changed programmatically by the developer.

8. Click **OK**.

After defining the bind variables, the next step is to reference them in the SQL statement. While SQL syntax allows bind variables to appear both in the `SELECT` list and in the `WHERE` clause, you'll typically use them in the latter context, as part of your `WHERE` clause. For example, Example 5–21 shows the bind variables `LowUserId` and `HighUserId` introduced into a SQL statement created using the Query page in the overview editor for the view object.

Example 5–21 Bind Variables in the WHERE Clause of View Object SQL Statement

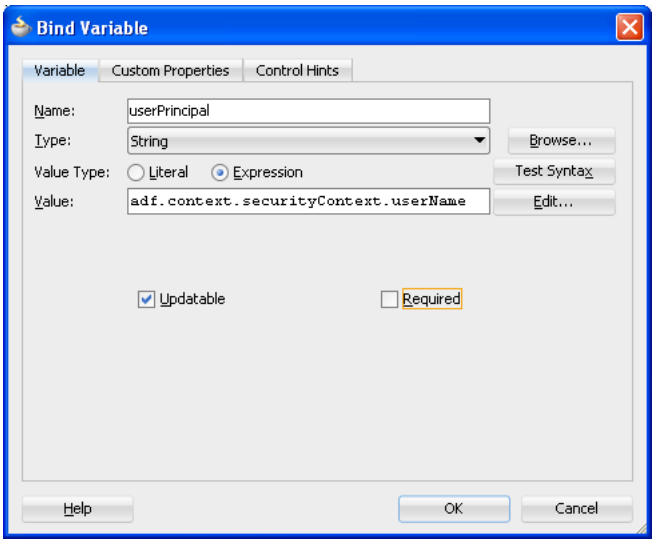
```
select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from PERSONS
where (upper(FIRST_NAME) like upper(:TheName)||'%'
   or  upper(LAST_NAME)  like upper(:TheName)||'%')
  and PERSON_ID between :LowUserId and :HighUserId
order by EMAIL
```

Notice that you reference the bind variables in the SQL statement by prefixing their name with a colon like `:TheName` or `:LowUserId`. You can reference the bind variables in any order and repeat them as many times as needed within the SQL statement.

5.10.2 How to Reference the Current User in a Named Bind Variable Using Groovy

You can use the Groovy expression `adf.context.securityContext.userName` to set the default value for the named bind variable that you use to provide the current user in a view instance filter. Specifically, you can use the bind variable in a named view criteria that you define to filter a view object instance in the data model for the project. For example, in the StoreFront module of the Fusion Order Demo application, the named bind variable `userPrincipal` is defined for the `PersonsVO` view object, as shown in Figure 5–39.

Figure 5–39 Groovy Expression Used to Set userPrincipal Bind Variable



The `PersonsVO` view object also defines the `AuthenticatedUserByPrincipalCriteria` view criteria. This view criteria defines a filter for the `PrincipalName` attribute of the `PersonsVO` with the bind variable `userPrincipal` providing the value. In this example, the bind variable `userPrincipal` is defined with **Updatable** enabled. This ensures that the view criteria is able to set the value obtained at runtime from the ADF security context. Since the bind variable is not used in the SQL `WHERE` clause for the `PersonsVO` view object, the **Required** field is unselected. This ensures that the value is optional and that no runtime exception will be thrown if the bind variable is not resolved.

Then in the data model for the `StoreFrontService` project, where the `PersonsVO` specifies the view definition for the usage `AuthenticatedUser`, the view criteria `AuthenticatedUserByPrincipalCriteria` with the named bind variable is defined as the view usage's runtime filter. For details about creating view instances for your project's data model, see Section 9.2.3.3, "Customizing a View Object Instance that You Add to an Application Module."

5.10.3 What Happens When You Add Named Bind Variables

Once you've added one or more named bind variables to a view object, you gain the ability to easily see and set the values of these variables at runtime. Information about the name, type, and default value of each bind variable is saved in the view object's XML component definition file. If you have defined UI hints for the bind variables, this information is saved in the view object's component message bundle file along with other UI hints for the view object.

5.10.4 How to Test Named Bind Variables

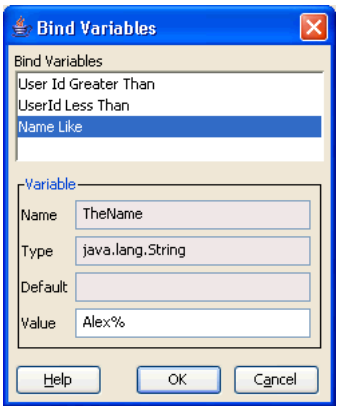
The Oracle ADF Model Tester allows you to interactively inspect and change the values of the named bind variables for any view object, which can really simplify experimenting with your application module's data model when named bind parameters are involved. For more information about editing the data model and running the Oracle ADF Model Tester, see Section 6.3, "Testing View Object Instances Using the Oracle ADF Model Tester."

The first time you execute a view object in the Oracle ADF Model Tester to display the results in the data view page, a Bind Variables dialog will appear, as shown in Figure 5–40.

The Bind Variables dialog lets you:

- View the name, as well as the default and current values, of the particular bind variable you select from the list
- Change the value of any bind variable by updating its corresponding **Value** field before clicking **OK** to set the bind variable values and execute the query
- Inspect and set the bind variables for the view object in the current data view page, using the **Edit Bind Parameters** button in the toolbar — whose icon looks like ":id"
- Verify UI hints are correctly set up by showing the label text hint in the **Bind Variables** list and by formatting the **Value** attribute using the respective format mask

Figure 5–40 Setting Bind Variables in the Oracle ADF Model Tester



If you defined the bind variable in the Bind Variables dialog with the **Reference** checkbox deselected (the default), you will be able to test view criteria and supply the bind variable with values as needed. Otherwise, if you selected the **Reference** checkbox, then you must supply a value for the bind variable in the Oracle ADF Model Tester. The Oracle ADF Model Tester will throw the same exception seen at runtime for any view object whose SQL statement use bind variables that do not resolve with a supplied value.

5.10.5 How to Add a WHERE Clause with Named Bind Variables at Runtime

Using the view object's `setWhereClause()` method, you can add an additional filtering clause at runtime. This runtime-added `WHERE` clause predicate does *not* replace the design-time generated predicate, but rather further narrows the query result by adding to the existing design time `WHERE` clause. Whenever the dynamically added clause refers to a value that might change during the life of the application, you should use a named bind variable instead of concatenating the literal value into the `WHERE` clause predicate.

For example, assume you want to further filter the `PersonList` view object at runtime based on the value of the `PERSON_TYPE_CODE` column in the table. Also assume that you plan to search sometimes for rows where `PERSON_TYPE_CODE = 'CUST'` and other times for rows where `PERSON_TYPE_CODE = 'SUPP'`. While slightly fewer lines of code, Example 5–22 is not desirable because it changes the

WHERE clause twice just to query two different *values* of the same PERSON_TYPE_CODE column.

Example 5–22 Incorrect Use of setWhereClause() Method

```
// Don't use literal strings if you plan to change the value!
vo.setWhereClause("person_type_code = 'CUST'");
// execute the query and process the results, and then later...
vo.setWhereClause("person_type_code = 'person'");
```

Instead, you should add a WHERE clause predicate that references named bind variables that you define at runtime as shown in Example 5–23.

Example 5–23 Correct Use of setWhereClause() Method and Bind Variable

```
vo.setWhereClause("person_type_code = :ThePersonType");
vo.defineNamedWhereClauseParam("ThePersonType", null, null);
vo.setNamedWhereClauseParam("ThePersonType","CUST");
// execute the query and process the results, and then later...
vo.setNamedWhereClauseParam("ThePersonType","person");
```

This allows the text of the SQL statement to stay the same, regardless of the value of PERSON_TYPE_CODE you need to query on. When the query text stays the same across multiple executions, the database will return the results without having to reparse the query.

If you later need to remove the dynamically added WHERE clause and bind variable, you should do so the next time you need them to be different, just before executing the query. This will prevent the type of SQL execution error as described in Section 5.10.8.1, "An Error Related to Clearing Bind Variables." Avoid calling removeNamedWhereClauseParam() in your code immediately after setting the WHERE clause. For a useful helper method to assist with this removal, see Section 5.10.8.2, "A Helper Method to Remove Named Bind Variables."

An updated test client class illustrating these techniques would look like what you see in Example 5–24. In this case, the functionality that loops over the results several times has been refactored into a separate executeAndShowResults() method. The program first adds an additional WHERE clause of person_id = :ThePersonId and then later replaces it with a second clause of person_type_code = :ThePersonType.

Example 5–24 TestClient Program Exercising Named Bind Variable Techniques

```
package devguide.examples.readonlyvo.client;

import oracle.jbo.ApplicationModule;
import oracle.jbo.Row;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;
import oracle.jbo.domain.Number;

public class TestClientBindVars {
  public static void main(String[] args) {
    String        amDef = "devguide.examples.readonlyvo.PersonService";
    String        config = "PersonServiceLocal";
    ApplicationModule am =
     Configuration.createRootApplicationModule(amDef,config);
    ViewObject vo = am.findViewObject("PersonList");
    // Set the two design time named bind variables
    vo.setNamedWhereClauseParam("TheName","shelli%");
```

```
    vo.setNamedWhereClauseParam("HighUserId", new Number(215));
    executeAndShowResults(vo);
    // Add an extra where clause with a new named bind variable
    vo.setWhereClause("person_type_code = :ThePersonId");
    vo.defineNamedWhereClauseParam("ThePersonId", null, null);
    vo.setNamedWhereClauseParam("ThePersonId",new Number(116));
    executeAndShowResults(vo);
    vo.removeNamedWhereClauseParam("ThePersonId");
    // Add an extra where clause with a new named bind variable
    vo.setWhereClause("person_type_code = :ThePersonType");
    vo.defineNamedWhereClauseParam("ThePersonType", null, null);
    vo.setNamedWhereClauseParam("ThePersonType","SUPP");
    // Show results when :ThePersonType = 'SUPP'
    executeAndShowResults(vo);
    vo.setNamedWhereClauseParam("ThePersonType","CUST");
    // Show results when :ThePersonType = 'CUST'
    executeAndShowResults(vo);
    Configuration.releaseRootApplicationModule(am,true);
  }
  private static void executeAndShowResults(ViewObject vo) {
    System.out.println("---");
    vo.executeQuery();
    while (vo.hasNext()) {
      Row curUser = vo.next();
      System.out.println(curUser.getAttribute("PersonId")+" "+
                         curUser.getAttribute("ShortName"));
    }
  }
}
```

However, if you run this test program, you may actually get a runtime error like the one shown in Example 5–25.

Example 5–25 Runtime Error Resulting From a SQL Parsing Error

```
oracle.jbo.SQLStmtException: JBO-27122: SQL error during statement preparation.
Statement:
SELECT * FROM (select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from PERSONS
where (upper(FIRST_NAME) like upper(:TheName)||'%'
   or  upper(LAST_NAME)  like upper(:TheName)||'%')
  and PERSON_ID between :LowUserId and :HighUserId
order by EMAIL) QRSLT  WHERE (person_type_code = :ThePersonType)
## Detail 0 ##
java.sql.SQLException: ORA-00904: "PERSON_TYPE": invalid identifier
```

The root cause, which appears after the `## Detail 0 ##` in the stack trace, is a SQL parsing error from the database reporting that `PERSON_TYPE_CODE` column does not exist even though the `PERSONS` table definitely has a `PERSON_TYPE_CODE` column. The problem occurs due to the mechanism that view objects use by default to apply additional runtime `WHERE` clauses on top of read-only queries. Section 5.10.7, "What Happens at Runtime: Dynamic Read-Only View Object WHERE Clause," explains a resolution for this issue.

5.10.6 How to Set Existing Bind Variable Values at Runtime

To set named bind variables at runtime, use the `setNamedWhereClauseParam()` method on the `ViewObject` interface. In JDeveloper, you can choose **Refactor > Duplicate** from the main menu to create a new `TestClientBindVars` class based on

the existing `TestClient.java` class as shown in Section 6.4.2, "How to Create a Command-Line Java Test Client." In the test client class, you can set the values of the bind variables using a few additional lines of code. For example, the `setNamedWhereClauseParam()` might take as arguments the bind variables `HighUserId` and `TheName` as shown in Example 5–26.

Example 5–26 Setting the Value of Named Bind Variables Programmatically

```
// changed lines in TestClient class
ViewObject vo = am.findViewObject("PersonList");
vo.setNamedWhereClauseParam("TheName","alex%");
vo.setNamedWhereClauseParam("HighUserId", new Number(315));
vo.executeQuery();
// etc.
```

Running the test client class shows that your bind variables are filtering the data. For example, the resulting rows for the `setNamedWhereClauseParam()` method shown in Example 5–26 may show only two matches based on the name `alex` as shown in Example 5–27.

Example 5–27 Result of Bind Variables Filtering the Data in TestClient Class

```
303 ahunold
315 akhoo
```

Whenever a view object's query is executed, you can view the actual bind variable values in the runtime debug diagnostics like the sample shown in Example 5–28.

Example 5–28 Debug Diagnostic Sample with Bind Variable Values

```
[256] Bind params for ViewObject: PersonList
[257] Binding param "LowUserId": 0
[258] Binding param "HighUserId": 315
[259] Binding param "TheName": alex%
```

This information that can be invaluable when debugging your applications. Notice that since the code did not set the value of the `LowUserId` bind variable, it took on the default value of `0` (zero) specified at design time. Also notice that the use of the `UPPER()` function in the `WHERE` clause and around the bind variable ensured that the match using the bind variable value for `TheName` was performed case-insensitively. The sample code set the bind variable value to "`alex%`" with a lowercase "`a`", and the results show that it matched `Alexander`.

5.10.7 What Happens at Runtime: Dynamic Read-Only View Object WHERE Clause

If you dynamically add an additional `WHERE` clause at runtime to a read-only view object, its query gets nested into an inline view before applying the additional `WHERE` clause.

For example, suppose your query was defined as shown in Example 5–29.

Example 5–29 Query Specified At Design Time

```
select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from PERSONS
where (upper(FIRST_NAME) like upper(:TheName)||'%'
   or  upper(LAST_NAME)  like upper(:TheName)||'%')
  and PERSON_ID between :LowUserId and :HighUserId
order by EMAIL
```

At runtime, when you set an additional WHERE clause like person_type_code = :ThePersonType as the test program did in Example 5–24, the framework nests the original query into an inline view like the sample shown in Example 5–30.

Example 5–30 Runtime-Generated Query with Inline Nested Query

```
SELECT * FROM(
select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from PERSONS
where (upper(FIRST_NAME) like upper(:TheName)||'%'
   or  upper(LAST_NAME)  like upper(:TheName)||'%')
  and PERSON_ID between :LowUserId and :HighUserId
order by EMAIL) QRSLT
```

Then the framework adds the dynamic WHERE clause predicate at the end, so that the final query the database sees is like the sample shown in Example 5–31.

Example 5–31 Runtime-Generated Query with Dynamic WHERE Clause

```
SELECT * FROM(
select PERSON_ID, EMAIL, FIRST_NAME, LAST_NAME
from PERSONS
where (upper(FIRST_NAME) like upper(:TheName)||'%'
   or  upper(LAST_NAME)  like upper(:TheName)||'%')
  and PERSON_ID between :LowUserId and :HighUserId
order by EMAIL) QRSLT
WHERE person_type_code = :ThePersonType
```

This query "wrapping" is necessary in the general case since the original query could be arbitrarily complex, including SQL UNION, INTERSECT, MINUS, or other operators that combine multiple queries into a single result. In those cases, simply "gluing" the additional runtime WHERE clause onto the end of the query text could produce unexpected results because, for example, it might apply only to the *last* of several UNION'ed statements. By nesting the original query verbatim into an inline view, the view object guarantees that your additional WHERE clause is correctly used to filter the results of the original query, regardless of how complex it is. The consequence (that results in an ORA-00904 error) is that the dynamically added WHERE clause can refer only to columns that have been selected in the original query.

The simplest solution is to add the dynamic query column names to the end of the query's SELECT list on the Edit Query dialog (click the **Edit SQL Query** button on the Query page of the overview editor for the view object). Just adding the new column name at the end of the existing SELECT list — of course, preceded by a comma — is enough to prevent the ORA-00904 error: JDeveloper will automatically keep your view object's attribute list synchronized with the query statement. Alternatively, Section 5.9.4.7, "Limitation of Inline View Wrapping at Runtime" explains how to disable this query nesting when you don't require it.

The test client program in Example 5–24 now produces the results shown in Example 5–32.

Example 5–32 Named Bind Variables Resulting From Corrected TestClient

```
---
116 S. Baida
---
116 S. Baida
---
```

```
---
116 S. Baida
```

5.10.8 What You May Need to Know About Named Bind Variables

There are several things you may need to know about named bind variables, including the runtime errors that are displayed when bind variables have mismatched names and the default value for bind variables.

5.10.8.1 An Error Related to Clearing Bind Variables

You need to ensure that your application handles changing the value of bind variables properly for use with activation and passivation of the view object instance settings at runtime. For example, before you deploy the application, you will want to stress-test your application in JDeveloper by disabling application module pooling, as described in Section 43.10, "Testing to Ensure Your Application Module is Activation-Safe." Following the instructions in that section effectively simulates the way your application will manage the passivation store when you eventually deploy the application.

When the application reactivates the pending state from the passivation store upon subsequent requests during the same user session, the application will attempt to set the values of any dynamically added named `WHERE` clause bind variables. Changing the values to null before passivation take places will prevent the bind variable values from matching the last time the view object was executed and the following error will occur during activation:

```
(oracle.jbo.SQLStmtException) JBO-27122: SQL error during statement preparation.
(java.sql.SQLException) Attempt to set a parameter name that does not occur in
SQL: 1
```

Do not change the value of the bind variables (or other view object instance settings) just after executing the view object. Rather, if you will not be re-executing the view object again during the same block of code (and therefore during the same HTTP request), you should defer changing the bind variable values for the view object instance until the next time you need them to change, just before executing the query. To accomplish this, use the following pattern:

1. (Request begins and application module is acquired)
2. Call `setWhereClause(null)` to clear `WHERE` clause
3. Call `setWhereClauseParam(null)` to clear the `WHERE` clause bind variables
4. Call `setWhereClause()` that references *n* bind variables
5. Calling `setWhereClauseParam()` to set the *n* values for those *n* bind variables
6. Calling `executeQuery()`
7. (Application module is released)

5.10.8.2 A Helper Method to Remove Named Bind Variables

The helper method `clearWhereState()` that you can add to your `ViewObjectImpl` framework ensures that declaratively defined bind variables are not removed. Example 5–33 shows the use of `clearWhereState()` to safely remove named bind variables that have been added to the view instance at runtime.

Example 5–33 Helper Method to Clear Named Bind Variables Values Programmatically

```
protected void clearWhereState() {
```

```
        ViewDefImpl viewDef = getViewDef();
        Variable[] viewInstanceVars = null;
        VariableManager viewInstanceVarMgr = ensureVariableManager();
        if (viewInstanceVarMgr != null) {
        viewInstanceVars = viewInstanceVarMgr.getVariablesOfKind
                                      (Variable.VAR_KIND_WHERE_CLAUSE_PARAM);
        if (viewInstanceVars != null) {
              for (Variable v: viewInstanceVars) {
                  // only remove the variable if its not on the view def.
                  if (!hasViewDefVariableNamed(v.getName())) {
                    removeNamedWhereClauseParam(v.getName());
                  }
              }
         }
    }
    getDefaultRowSet().setExecuteParameters(null, null, true);
    setWhereClause(null);
    getDefaultRowSet().setWhereClauseParams(null);
}
private boolean hasViewDefVariableNamed(String name) {
    boolean ret = false;
    VariableManager viewDefVarMgr = getViewDef().ensureVariableManager();
    if (viewDefVarMgr != null) {
        try {
            ret = viewDefVarMgr.findVariable(name) != null;
        }
        catch (NoDefException ex) {
               // ignore
        }
    }
    return ret;
}
```

5.10.8.3 Errors Related to Naming Bind Variables

You need to ensure that the list of named bind variables that you reference in your SQL statement matches the list of named bind variables that you've defined in the **Bind Variables** section of the overview editor's Query page for the view object. Failure to have these two agree correctly can result in one of the following two errors at runtime.

If you use a named bind variable in your SQL statement but have not defined it, you'll receive an error like this:

```
(oracle.jbo.SQLStmtException) JBO-27122: SQL error during statement preparation.
## Detail 0 ##
(java.sql.SQLException) Missing IN or OUT parameter at index:: 1
```

On the other hand, if you have defined a named bind variable, but then forgotten to reference it or mistyped its name in the SQL, then you will see an error like this:

```
oracle.jbo.SQLStmtException: JBO-27122: SQL error during statement preparation.
## Detail 0 ##
java.sql.SQLException: Attempt to set a parameter name that does not occur in the
SQL: LowUserId
```

To resolve either of these errors, double-check that the list of named bind variables in the SQL matches the list of named bind variables in the **Bind Variables** section of the overview editor's Query page for the view object. Additionally, open the Bind Variables dialog for the bind variable and verify that the **Reference** checkbox is not

still deselected (the default). To use the bind variable in a SQL statement, you must select the **Reference** checkbox.

5.10.8.4 Default Value of NULL for Bind Variables

If you do not supply a default value for your named bind variable, it defaults to the `NULL` value at runtime. This means that if you have a `WHERE` clause like:

```
PERSON_ID = :ThePersonId
```

and you do not provide a default value for the `ThePersonId` bind variable, it will default to having a `NULL` value and cause the query to return no rows. Where it makes sense for your application, you can leverage SQL functions like `NVL()`, `CASE`, `DECODE()`, or others to handle the situation as you require. For example, the following `WHERE` clause fragment allows the view object query to match any name if the value of `:TheName` is `null`.

```
upper(FIRST_NAME) like upper(:TheName)||'%'
```

5.11 Working with Named View Criteria

A view criteria you define lets you specify filter information for the rows of a view object collection. The view criteria object is a row set of one or more view criteria rows, whose attributes mirror those in the view object. The view criteria definition comprises query conditions that augment the `WHERE` clause of the target view object. Query conditions that you specify apply to the individual attributes of the target view object.

The key difference between a view object row of query results and a view criteria row is that the data type of each attribute in the view criteria row is `String`. This key difference supports Query-by-Example operators and therefore allows the user to enter conditions such as "`OrderId > 304`", for example.

The Edit View Criteria dialog lets you create view criteria and save them as part of the view object's definition, where they appear as named view criteria. You use the Query page of the overview editor to define view criteria for specific view objects. View criteria that you define at design time can participate in these scenarios where filtering results is desired at runtime:

- Supporting Query-by-Example search forms that allow the end user to supply values for attributes of the target view object.

  For example, the end user might input the value of a customer name and the date to filter the results in a web page that displays the rows of the `CustomerOrders` view object. The web page designer will see the named view criteria in the JDeveloper Data Controls panel and, from them, easily create a search form. For more information about the utilizing the named view criteria in the Data Controls panel, see Section 31.2, "Creating Query Search Forms."

- Filtering the list of values (LOV) components that allow the end user may select from one attribute list (displayed in the UI as an LOV component).

  The web page designer will see the attributes of the view object in the JDeveloper Data Controls panel and, from them, easily create LOV controls. For more information about utilizing LOV-enabled attributes in the Data Controls panel, see Section 30.3, "Creating a Selection List."

- Validating attribute values using a view accessor with a view criteria applied to filter the view accessor results.

For more information about create view accessor validators, see Section 10.4.2, "How to Validate Against the Attribute Values Specified by a View Accessor."

- Creating the application module's data model from a single view object definition with a unique view criteria applied for each view instance.

  The single view object query modified by view criteria is useful with look up data that must be shared across the application. In this case, a base view object definition queries the lookup table in the database and the view criteria set the lookup table's `TYPE` column to define application-specific views. To define view instances in the data model using the view criteria you create for a base view object definition, see Section 10.3.3, "How to Define the WHERE Clause of the Lookup View Object Using View Criteria."

Additionally, view criteria have full API support, and it is therefore possible to create and apply view criteria to view objects programmatically.

5.11.1 How to Create Named View Criteria Declaratively

View criteria have a number of uses in addition to applying them to declarative queries at runtime. In all usages, the named view criteria definition consists of a set of attribute requirements that you specify to filter individual view object results. The features of the view criteria definition that you can use will depend on its intended usage.

To define view criteria for the view object you wish to filter, you open the view object in the overview editor and use the **View Criteria** section of the Query page. A dedicated editor (the Create View Criteria dialog) that you open from the **View Criteria** section helps you to build a `WHERE` clause using attribute names instead of the target view object's corresponding SQL column names. You may define multiple named view criteria for each view object.

Before you work with the Create View Criteria dialog to create named view criteria, familiarize yourself with the usages described in Section 5.11, "Working with Named View Criteria." The chapter references provide additional details that will help you to anticipate using the appropriate features of the Create View Criteria dialog. For example, when you create a view criteria to specify the searchable attributes of a search form, the view criteria condition defines a simple list of attributes (a subset of the view object's attributes) to be presented to the user, but then the view criteria definition requires that you specify UI hints (model-level properties) to control the behavior of those attributes in the search form. The Create View Criteria dialog displays all the UI hints in a separate tabbed page that you select for the view criteria you are defining. Whereas, when your view criteria is intended to specify view instances in the data model, you can define arbitrarily complex query filter conditions, but you can ignore the UI hints features displayed by the Create View Criteria dialog.

Each view criteria definition consists of the following elements:

- One or more view criteria rows consisting of an arbitrary number of view criteria groups or an arbitrary number of references to another named view criteria already defined for the current view object.
- Optional view criteria groups consisting of an arbitrary number of view criteria items.
- View criteria items consisting of an attribute name, an attribute-appropriate operator, and an operand. Operands can be a literal value when the filter value is defined or a bind variable that can optionally utilize a scripting expression that includes dot notation access to attribute property values.

Expressions are based on the Groovy scripting language, as described in Section 3.6, "Overview of Groovy Scripting Language Support."

When you define a view criteria, you control the source of the filtered results. You can limit the results of the filtered view object to:

- Just the database table specified by the view object
- Just the in-memory results of the view object query
- Both the database and the in-memory results of the view object query.

Filtering on both database tables and the view object's in-memory results allows you to filter rows that were created in the transaction but not yet committed to the database.

View criteria expressions you construct in the Edit View Criteria dialog use logical conjunctions to specify how to join the selected criteria item or criteria group with the previous item or group in the expression:

- `AND` conjunctions specify that the query results meet both joined conditions. This is the default for each view criteria item you add.
- `OR` conjunctions specify that the query results meet either or both joined conditions. This is the default for view criteria groups.

Additionally, you may create nested view criteria when you want to filter rows in the current view object based on criteria applied to view-linked detail views. A nested view criteria group consists of an arbitrary number of nested view criteria items. You can use nested view criteria when you want to have more controls over the logical conjunctions among the various view criteria items. The nested criteria place restrictions on the rows that satisfy the criteria under the nested criteria's parent view criteria group. For example, you might want to query both a list of employees with (`Salary > 3000`) and belonging to (`DeptNo = 10` or `DeptNo = 20`). You can define a view criteria with the first group with one item for (`Salary > 3000`) and a nested view criteria with the second group with two items `DeptNo = 10` and `DeptNo =20`.

Before you begin:

It may be helpful to have an understanding of the ways you can use view criteria. The usage you intend will affect the best practices for creating named view criteria. For more information about the supported usages, see Section 5.11, "Working with Named View Criteria."

You will need to complete these tasks:

- Create the desired view objects as described in Section 5.2.1, "How to Create an Entity-Based View Object," and Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object."
- If the view criteria will use a bind variable in the operand, create the bind variable as described in Section 5.10.1, "How to Add Bind Variables to a View Object Definition."

To define a named view criteria:

1. In the Application Navigator, double-click the view object for which you want to create the named view criteria.
2. In the overview editor, click the **Query** navigation tab.
3. In the Query page, expand the **View Criteria** section and click the **Create new view criteria** button.

4. In the Create View Criteria dialog, enter the name of the view criteria to identify its usage in your application.
5. In the **Query Execution Mode** dropdown list, decide how you want the view criteria to filter the view object query results.

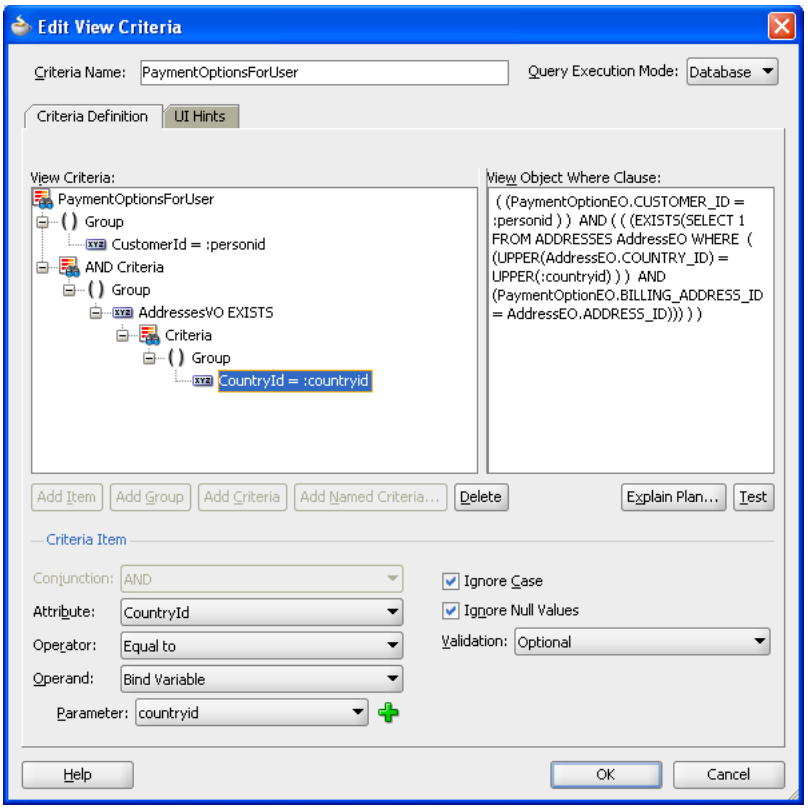
   You can limit the view criteria to filter the database table specified by the view object query, the in memory row set produced by the query, or both the database table and the in-memory results.

   Choosing **Both** may be appropriate for situations where you want to include rows created as a result of enforced association consistency. In this case, in-memory filtering is performed after the initial fetch.
6. Click one of these **Add** buttons to define the view criteria.
   - **Add Item** to add a single criteria item. The editor will add the item to the hierarchy beneath the current group or view criteria selection. By default each time you add an item, the editor will choose the next attribute to define the criteria item. You can change the attribute to any attribute that the view object query defines.
   - **Add Group** to add a new group that will compose criteria items that you intend to add to it. When you add a new group, the editor inserts the `OR` conjunction into the hierarchy. You can change the conjunction as desired.
   - **Add Criteria** to add a view criteria that you intend to define. This selection is an alternative to adding a named criteria that already exists in the view object definition. When you add a new view criteria, the editor inserts the `AND` conjunction into the hierarchy. You can change the conjunction as desired. Each time you add another view criteria, the editor nests the new view criteria beneath the current view criteria selection in the hierarchy. The root node of the hierarchy defines the named view criteria that you are currently editing.

     Search forms that the UI designer will create from view criteria are not able to use directly nested view criteria. For more information about defining nested expressions for use with search forms, see Section 5.11.4, "What You May Need to Know About Nested Expressions."
   - **Add Named Criteria** to add a view criteria that the view object defines. The named criteria must appear in the overview editor for the view object you are defining the view criteria.
7. Select a view criteria item node in the view criteria hierarchy and define the added node in the **Criteria Item** section.
   - Choose the desired **Attribute** for the criteria item. By default the editor adds the first one in the list.

     Optionally, you can add a nested view criteria inline when a view link exists for the current view object you are editing. The destination view object name will appear in the **Attribute** dropdown list. Selecting a view object lets you filter the view criteria based on view criteria items for the nested view criteria based on a view link relationship. For example, `AddressVO` is linked to the `PaymentOptionsVO` and a view criteria definition for `PaymentOptionsVO` will display the destination view object `AddressVO`. You could define the nested view criteria to filter payment options based on the `CountryId` attribute of the current customer, as specified by the `CustomerId` criteria item, as shown in Figure 5–41.

Figure 5–41 Edit View Criteria Dialog with Nested View Criteria Specified



- Choose the desired **Operator**.

  The list displays only the operators that are appropriate for the selected attribute or view object. In the case of a view object selection, the **exists** operator applies to a view criteria that you will define (or reference) as an operand. In the case of String and Date type attributes, the **Between** and **Not between** operators require you to supply two operand values to define the range. In the case of Date type attributes, you can select operators that test for a date or date range (with date values entered in the format YYYY-MM-DD). For example, for December 16th, 2010, enter 2010-12-16.

  JDeveloper does not support the `IN` operator. However, you can create a view criteria with the `IN` operator using the API, as described in Section 5.11.7, "How to Create View Criteria Programmatically."

8. Choose the desired **Operand** for the view criteria item selection.

   - Select **Literal** when you want to supply a value for the attribute or when you want to define a default value for a user-specified search field for a Query-by-Example search form. When the view criteria defines a query search form for the user interface, you may leave the **Value** field empty. In this case, the user will supply the value. You may also provide a value that will act as a search field default value that the user will be able to override. The value you supply in the **Value** field can include wildcard characters `*` or `%`.
   - Select **Bind Variable** when you want the value to be determined at runtime using a bind variable. If the variable was already defined for the view object, select it from the **Parameters** dropdown list. Otherwise, click **New** to display the Bind Variable dialog that lets you create a new bind variable on the view

object. For more information about creating bind variables, see Section 5.10.1, "How to Add Bind Variables to a View Object Definition."

When you define bind variables on the view object for use by the view criteria, you must specify that the variable is not required by the SQL query that the view object defines. To do this, deselect the **Required** checkbox in the Bind Variables dialog, as explained in Section 5.10.1, "How to Add Bind Variables to a View Object Definition."

For further discussion about view criteria use cases for bind variables and literals, see Section 5.11.3, "What You May Need to Know About Bind Variable Options."

9. For each item, group, or nested view criteria that you define, optionally change the default conjunction to specify how the selection should be joined.
   - **AND** conjunction specifies that the query results meet both joined conditions. This is the default for each view criteria item or view nested view criteria that you add.
   - **OR** conjunction specifies that the query results meet either or both joined conditions. This is the default for view criteria groups.
10. Verify that the view criteria definition is valid by doing one of the following:
    - Click **Explain Plan** to visually inspect the view criteria's generated `WHERE` clause.
    - Click **Test** to allow JDeveloper to verify that the `WHERE` clause is valid.
11. To prevent the attribute to be filtered based on the case of the runtime-supplied value, leave **Ignore Case** selected.

    The criteria item can be a literal value that you define or a runtime parameter that the end user supplies. This option is supported for attributes of type String only. The default disables case sensitive searches.
12. In the **Validation** dropdown list, decide whether the view criteria item is a required or an optional part of the attribute value comparison in the generated `WHERE` clause.
    - **Selectively Required** means that the `WHERE` clause will ignore the view criteria item at runtime if no value is supplied and there exists at least one criteria item at the same level that has a criteria value. Otherwise, an exception is thrown.
    - **Optional** means the view criteria is added to the `WHERE` clause only if the value is non-`NULL`. The default **Optional** for each new view criteria item means no exception will be generated for null values.
    - **Required** means that the `WHERE` clause will fail to execute and an exception will be thrown when no value is supplied for the criteria item.
13. If the view criteria uses a bind variable as the operand, decide whether the `IS NULL` condition is the generated in the `WHERE` clause. This field is enabled only if you have selected **Optional** for the validation of the bind variable.
    - Leave **Ignore Null Values** selected (default) when you want to permit the view criteria to return a result even when the bind variable value is not supplied at runtime. For example, suppose you define a view criteria to allow users to display a cascading list of countries and states (or provinces) through a bind variable that takes the `countryID` as the child list's controlling attribute. In this case, the default behavior for the view criteria execution

returns the list of *all* states if the user makes no selection in the parent LOV (an empty `countryId` field). The generated `WHERE` clause would look similar to `(((CountryEO.COUNTRY_ID =:bvCountryId) OR (:bvCountryId IS NULL)))`, where the test for a null value guarantees that the child list displays a result even when the bind variable is not set. When validation is set to **Required** or **Optionally Required**, the view criteria expects to receive a value and thus this option to ignore null values is disabled.

- Deselect **Ignore Null Values** when you expect the view criteria to return a null result when the bind variable value is not supplied at runtime. In the example of the cascading lists, the view criteria execution returns *no* states if the user makes no selection with an empty `countryID` field. In this case, the generated `WHERE` clause would look similar to `((CountryEO.COUNTRY_ID=:bvCountryId))`, where the test for null is not performed, which means the query is expected to function correctly with a null value bind variable.

  Note that the validation settings **Required** or **Optionally Required** also remove the null value condition but support a different use case. They should be used in combination with **Ignore Null Values** feature to achieve the desired runtime behavior. For more details about the interaction of these features, see Section 5.11.3, "What You May Need to Know About Bind Variable Options."

14. Click **OK**.

5.11.2 What Happens When You Create a Named View Criteria

The Create View Criteria dialog in JDeveloper lets you easily create view criteria and save them as named definitions. These named view criteria definitions add metadata to the XML component definition file that represents the target view object's declarative settings. Once defined, named view criteria appear by name in the Query page of the overview editor for the view object.

To view the view criteria, expand the desired view object in the Application Navigator, select the XML file under the expanded view object, open the Structure window, and expand the View Criteria node. Each view criteria definition for a view object contains one or more `<ViewCriteriaRow>` elements corresponding to the number of groups that you define in the Create View Criteria dialog. Example 5–34 shows the `ProductsVO.xml` file with the `<ViewCriteria>` definition `FindByProductNameCriteria` and a single `<ViewCriteriaRow>` that defines a developer-seeded search for products using the bind variable `:bvProductName`. Any UI hints that you selected to customize the behavior of a developer-seeded search will appear in the `<ViewCriteria>` definition as attributes of the `<CustomProperties>` element. For details about specific UI hints for view criteria, see Section 5.11.5, "How to Set User Interface Hints on View Criteria to Support Search Forms."

Example 5–34 FindByProductNameCriteria View Criteria in the ProductsVO View Object Definition

```
<ViewObject
    xmlns="http://xmlns.oracle.com/bc4j"
    Name="ProductsVO"
    ... >
  <SQLQuery>
    ...
  </SQLQuery>
  ...
```

```
  <ViewCriteria
    Name="FindByProductNameCriteria"
    ViewObjectName="oracle.fodemo.storefront.store.queries.ProductsVO"
    Conjunction="AND">
    <Properties>
      <CustomProperties>
        <Property
          Name="mode"
          Value="Basic"/>
        <Property
          Name="autoExecute"
          Value="false"/>
        <Property
          Name="showInList"
          Value="true"/>
        <Property
          Name="displayName"
          Value="Find Products By Name"/>
        <Property
          Name="displayOperators"
          Value="InAdvancedMode"/>
        <Property
          Name="allowConjunctionOverride"
          Value="true"/>
      </CustomProperties>
    </Properties>
    <ViewCriteriaRow
      Name="vcrow87">
      <ViewCriteriaItem
        Name="ProductName"
        ViewAttribute="ProductName"
        Operator="CONTAINS"
        Conjunction="AND"
        Value=":bvProductName"
        UpperColumns="1"
        IsBindVarValue="true"
        Required="Optional"/>
    </ViewCriteriaRow>
  </ViewCriteria>
  ...
</ViewObject>
```

Additionally, when you create view objects and specify them as instances in an application module, JDeveloper automatically creates a data control to encapsulate the collections (view instances) that the application module contains. JDeveloper then populates the Data Controls panel with these collections and any view criteria that you have defined, as shown in Section 13.3.1.3, "How View Objects Appear in the Data Controls Panel."

5.11.3 What You May Need to Know About Bind Variable Options

The view criteria filter that you define using a bind variable expects to obtain its value at runtime. This can be helpful in a variety of user interface scenarios. To support a particular use case, familiarize yourself with the combination of the **Validation** and **Ignore Null Values** settings shown in Table 5–1.

Table 5–1 Use Cases for Bind Variable Options in View Criteria

Validation	Ignore Null Values	Use Cases	Notes
`Optional`	`True` (Default)	Configure cascading List of Values (LOV) where the parent LOV value is optional. Generate an optional search field in a search form.	This combination generates the SQL query `(ProductName = :bind) OR (:bind IS NULL)`. When used for cascading LOVs, no selection in the parent LOV returns all rows in the child LOV. Note that the preferred implementation for an optional search field is a view criteria item with a literal operand type.
`Optional`	`False`	Configure cascading LOVs where the parent LOV value is required.	This combination generates the SQL query `(ProductName = :bind)`. When used for cascading LOVs, no selection in the parent LOV returns no rows in the child LOV. Avoid this combination for search forms, because when the user leaves the search field blank the search will attempt to find rows where this field is explicitly NULL. A better way to achieve this is for the user to explicitly select the "IS NULL" operator in advanced search mode.
`Required`	`False` (default)	Generate a required search field in a search form.	This combination generates the SQL query `ProductName = :bind`. Avoid this setting for cascading LOVs, because no selection in the parent LOV will cause a validation error. Note that the preferred implementation for a required search field is a view criteria item with a literal operand (not a bind variable) type.

5.11.4 What You May Need to Know About Nested Expressions

Search forms that the UI designer will create from view criteria are not able to work with all types of nested expressions. Specifically, search forms do not support expressions with directly nested view criteria. This type of nested expression defines one view criteria as a direct child of another view criteria. Query search forms do support nested expressions where you nest the view criteria as a child of a criteria item which is itself a child of a view criteria. For more information about using view criteria to create search forms, see Section 31.1.1, "Implicit and Named View Criteria."

5.11.5 How to Set User Interface Hints on View Criteria to Support Search Forms

Named view criteria that you create for view object collections can be used by the web page designer to create Query-by-Example search forms. Web page designers select your named view criteria from the JDeveloper Data Controls panel to create search forms for the Fusion web application. In the web page, the search form utilizes an ADF Faces query search component that will be bound initially to the named view criteria selected in the Data Controls panel. At runtime, the end user may select among all other named view criteria that appear in the Data Controls panel. Named view criteria that the end user can select in a search form are known as *developer-seeded searches*. The

query component automatically displays these seeded searches in its **Saved Search** dropdown list. For more information about creating search forms and using the ADF query search component, see Section 31.2, "Creating Query Search Forms."

Note: By default, any named view criteria you create in the Edit View Criteria dialog will appear in the Data Controls panel. As long as the **Show In List** option appears selected in the UI Hints page of the Edit View Criteria dialog, JDeveloper assumes that the named view criteria should be available as a developer-seeded search. When you want to create a named view criteria that you do not want the end user to see in search forms, deselect the **Show In List** option in the dialog. For example, you might create a named view criteria only for an LOV-enabled attribute and so you would need to deselect **Show In List**.

Because developer-seeded searches are created in the data model project, the UI Hints page of the Edit View Criteria dialog lets you specify the default properties for the query component's runtime usage of individual named view criteria. At runtime, the query component's behavior will conform to the selections you make for the following seeded search properties:

Search Region Mode: Select the mode that you want the query component to display the seeded search as. The **Basic** mode has all features of **Advanced** mode, except that it does not allow the end user to dynamically modify the displayed search criteria fields. The default is **Basic** mode for a view criteria you define in the Edit View Criteria dialog.

Query Automatically: Select when you want the query associated with the named view criteria to be executed and the results displayed in the web page. Any developer-seeded search with this option enabled will automatically be executed when the end user selects it from the query component's **Saved Search** list. Deselect when the web page designer prefers not to update the previously displayed results until the end user submits the search criteria values on the form. Additionally, when a search form is invoked from a task flow, the search form will appear empty when this option is deselected and populated when enabled. By default, this option is disabled for a view criteria you define in the Edit View Criteria dialog.

Show Operators: Determine how you want the query component to display the operator selection field for the view criteria items to the end user. For example, select **Always** when you want to allow the end user to customize the operators for criteria items (in either basic or advanced modes) or select **Never** when you want the view criteria to be executed using the operators it defines. Note that the end user cannot change the operator for criteria items that you specify with a bind variable because bind variables may be used in more than one criteria item.

Show Match All and Match Any: Select to allow the query component to display the **Match All** and **Match Any** radio selection buttons to the end user. When these buttons are present, the end user can use them to modify the search to return matches for all view criteria items or any one view criteria item. This is equivalent to enforcing `AND` (match all) or `OR` (match any) conjunctions between view criteria items. Deselect when you want the view criteria to be executed using the view criteria item conjunctions it defines. In this case, the query component will not display the radio selection buttons.

Rendered Mode: Select individual view criteria items from the view criteria tree component and choose whether you want the selected item to appear in the search form when the end user toggles the query component between basic mode and

advanced mode. The default for every view criteria item is **All**. The default mode permits the query component to render an item in either basic or advanced mode. By changing the **Rendered Mode** setting for individual view criteria items, you can customize the search form's appearance at runtime. For example, you may want basic mode to display a simplified search form to the end user, reserving advanced mode for displaying a search form with the full set of view criteria items. In this case, you would select **Advanced** for the view criteria item that you do not want displayed in the query component's basic mode. In contrast, when you want the selected view criteria item to be rendered only in basic mode, select **Basic**. Set any item that you do not want the search form to render in either basic or advanced mode to **Never**.

Note: When your view criteria includes an item that should not be exposed to the user, use the **Rendered Mode** setting **Never** to prevent it from appearing in the search form. For example, a view criteria may be created to search for products in the logged-in customer's cart; however, you may want to prevent the user from changing the customer ID to display another customer's cart contents. In this scenario, the view criteria item corresponding to the customer ID would be set to the current customer ID using a named bind variable. Although the bind variable definition might specify the variable as not required and not updatable, with the UI hint property **Display** set to **Hide**, only the **Rendered Mode** setting determines whether or not the search form displays the value.

Support Multiple Value Selection: Select when you want to allow the end user to make multiple selections for an individual criteria item that the query component displays. This option is only enabled when the view object attribute specified by the view criteria item has a List of Values (LOV) defined. Additionally, multiple selections will only be supported by the query component when the end user selects the operator **equal to** or **not equal to**. For example, if the criteria item names an attribute `CountryId` and this attribute derives its values from a list of country IDs accessed by the attribute's associated LOV, then selecting this option would allow the end user to submit the query with multiple country selections. At runtime, the query component will generate the appropriate query clause based on the end user's operator selection.

Display Width: Enter the character width to be used by the control that displays this criteria item in the query component. The value you enter will override the display width control hint defined for the criteria item's corresponding view object attribute. For example, in an edit form the attribute control hint may allow text of 1024 length, but in the search form you might want to limit the field for the criteria item to 20 character length.

Show In List: Select to ensure that the view criteria is defined as a developer-seeded query. Deselect when the named view criteria you are defining is not to be used by the query search component to display a search form. Your selection determines whether the named view criteria will appear in the query search component's **Saved Search** dropdown list of available seeded searches. By default, this option is enabled for a view criteria you define in the Edit View Criteria dialog.

Display Name: Enter the name of the seeded search that you want to appear in the query component's **Saved Search** dropdown list or click the **...** button (to the right of the edit field) to select a message string from the resource bundle associated with the view object. The display name will be the name by which the end user identifies the seeded search. When you select a message string from the resource bundle, JDeveloper saves the string's corresponding message key in the view object definition file. At

runtime, the UI locates the string to display based on the end user's locale setting and the message key in the localized resource bundle. When you do not specify a display name, the view criteria name displayed in the Edit View Criteria dialog will be used by default.

To create a seeded search for use by the ADF query search component, you select **Show In List** in the UI Hints page of the Edit View Criteria dialog. You deselect **Show In List** when you do not want the end user to see the view criteria in their search form.

Before you begin:

It may be helpful to have an understanding of view criteria. For more information, see Section 5.11, "Working with Named View Criteria."

You will need to complete these tasks:

- Create the desired view objects as described in Section 5.2.1, "How to Create an Entity-Based View Object," and Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object."
- Create the view criteria as described in Section 5.11.1, "How to Create Named View Criteria Declaratively."

To customize a named view criteria for the user interface:

1. In the Application Navigator, double-click the view object that defines the named view criteria you want to use as a seeded search.
2. In the overview editor, click the **Query** navigation tab.
3. In the Query page, expand the **View Criteria** section and double-click the named view criteria that you want to allow in seeded searches.
4. In Edit View Criteria dialog, click the **UI Hints** tab and ensure that **Show In List** is selected.

   This selection determines whether or not the query component will display the seeded search in its **Saved Search** dropdown list.
5. Enter a user-friendly display name for the seeded search to be added to the query component **Saved Search** dropdown list.

   When left empty, the view criteria name displayed in the Edit View Criteria dialog will be used by the query component.
6. Optionally, enable **Query Automatically** when you want the query component to automatically display the search results whenever the end user selects the seeded search from the **Saved Search** dropdown list.

   By default, no search results will be displayed.
7. Optionally, apply **Criteria Item UI Hints** to customize whether the query component renders individual criteria items when the end user toggles the search from between basic and advanced mode.

   By default, all view criteria items defined by the seeded search will be displayed in either mode.

   If a rendered criteria item is of type `Date`, you must also define UI hints for the corresponding view object attribute. Set the view object attribute's Format Type hint to **Simple Date** and set the Format Mask to an appropriate value, as described in Section 5.13.1, "How to Add Attribute-Specific UI Hints." This will allow the search form to accept date values.
8. Click **OK**.

5.11.6 How to Test View Criteria Using the Oracle ADF Model Tester

To test the view criteria you added to a view object, use the Oracle ADF Model Tester, which is accessible from the Application Navigator.

The Oracle ADF Model Tester, for any view object instance that you browse, lets you bring up the View Criteria dialog, as shown in Figure 5–42. The dialog allows you to create a view criteria comprising one or more view criteria rows.

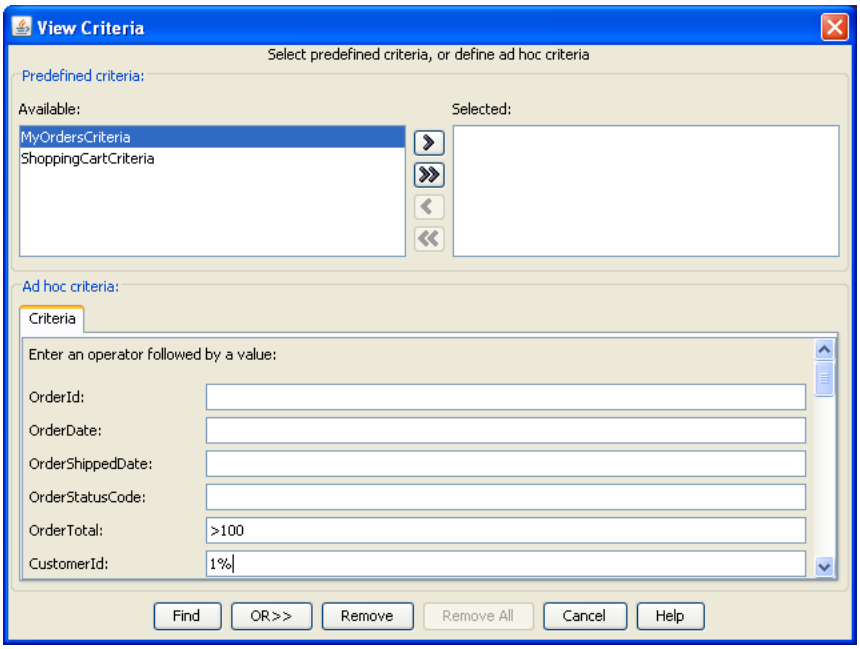
To apply criteria attributes from a single view criteria row, click the **Specify View Criteria** toolbar button in the browser and enter Query-by-Example criteria in the desired fields, then click **Find**.

To test view criteria using the Oracle ADF Model Tester:

1. In the Application Navigator, expand the project containing the desired application module and view objects.
2. Right-click the application module and choose **Run**.
3. In the Oracle ADF Model Tester, double-click the view instance you want to filter to open it in the tester, and then right-click the view instance and choose **Find**.

   Alternatively, after you double-click a view instance, you can click the **Specify View Criteria** toolbar button to test the view criteria.
4. In the View Criteria dialog, perform one of the following tasks:
   - To test a view criteria that you added to the view object in your project, select from the list and click **Find**. Any additional criteria that you enter in the ad hoc Criteria panel will be added to the filter.
   - To test ad hoc criteria attributes from a single view criteria row, enter the desired values for the view criteria and click **Find**. For example, Figure 5–42 shows the filter to return all customers who possess a customer ID that begins with the number "`1`" and placed an order in the amount greater than `100`.
   - To test additional ad hoc view criteria rows, click the **OR** tab and use the additional tabs that appear to switch between pages, each representing a distinct view criteria row. When you click **Find**, the Oracle ADF Model Tester will create and apply the view criteria to filter the result.

Figure 5–42 View Criteria Dialog in Oracle ADF Model Tester



5.11.7 How to Create View Criteria Programmatically

Example 5–35 shows the `main()` method finds the `PersonList` view object instance to be filtered, creates a view criteria for the attributes of this view object, and applies the view criteria to the view object.

To create a view criteria programmatically, follow these basic steps (as illustrated in Example 5–35):

1. Find the view object instance to be filtered.
2. Create a view criteria row set for the view object.
3. Use the view criteria to create one or more empty view criteria rows
4. Set attribute values to filter on the appropriate view criteria rows.

   You can use the single method `setAttribute()` on the view criteria rows to set attribute name, comparison operator, and value to filter on. Alternatively, use `ensureCriteriaItem()`, `setOperator()`, and `setValue()` on the view criteria rows to set attribute name, comparison operator, and value to filter on individually.
5. Add the view criteria rows to the view criteria row set.
6. Apply the view criteria to the view object.
7. Execute the query.

The last step to execute the query is important, since a newly applied view criteria is applied to the view object's SQL query only at its next execution.

Example 5–35 Creating and Applying a View Criteria

```
package devguide.examples.readonlyvo.client;

import oracle.jbo.ApplicationModule;
import oracle.jbo.Row;
```

```
import oracle.jbo.ViewCriteria;
import oracle.jbo.ViewCriteriaRow;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;

public class TestClientViewCriteria {
    public static void main(String[] args) {
        String amDef = "devguide.examples.readonlyvo.PersonService";
        String config = "PersonServiceLocal";
        ApplicationModule am =
            Configuration.createRootApplicationModule(amDef, config);
        // 1. Find the view object to filter
        ViewObject vo = am.findViewObject("PersonList");
        // Work with your appmodule and view object here
        Configuration.releaseRootApplicationModule(am, true);
        // 2. Create a view criteria row set for this view object
        ViewCriteria vc = vo.createViewCriteria();
        // 3. Use the view criteria to create one or more view criteria rows
        ViewCriteriaRow vcr1 = vc.createViewCriteriaRow();
        ViewCriteriaRow vcr2 = vc.createViewCriteriaRow();
        // 4. Set attribute values to filter on in appropriate view criteria rows
        vcr1.setAttribute("PersonId","> 200");
        vcr1.setAttribute("Email","d%");
        vcr1.setAttribute("PersonTypeCode","STAFF");
        vcr2.setAttribute("PersonId","IN (204,206)");
        vcr2.setAttribute("LastName","Hemant");
        // 5. Add the view criteria rows to the view critera row set
        vc.add(vcr1);
        vc.add(vcr2);
        // 6. Apply the view criteria to the view object
        vo.applyViewCriteria(vc);
        // 7. Execute the query
        vo.executeQuery();
        while (vo.hasNext()) {
        Row curPerson = vo.next();
        System.out.println(curPerson.getAttribute("PersonId") + " " +
        curPerson.getAttribute("Email"));
        }
    }
}
```

Running the `TestClientViewCriteria` example produces the results shown in Example 5–35:

```
206 SHEMANT
```

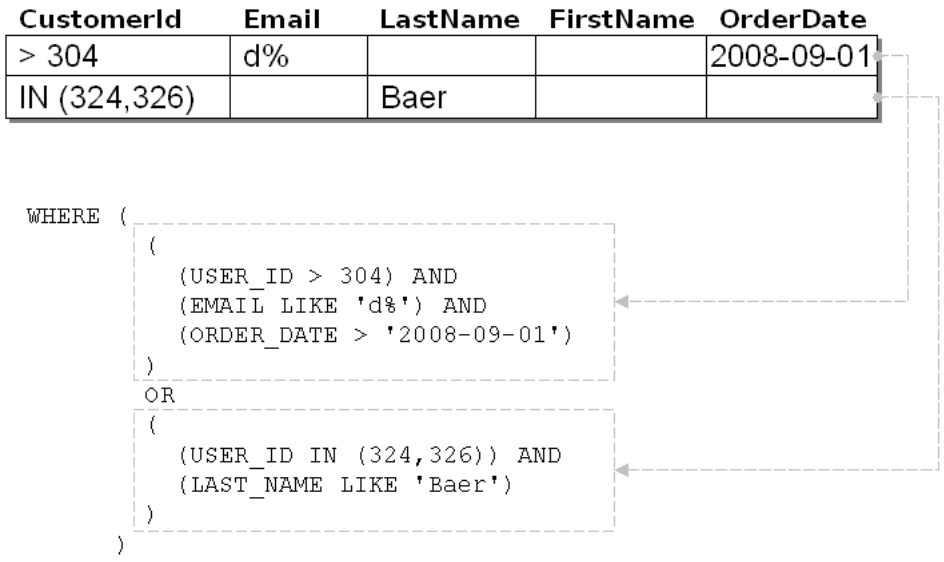
5.11.8 What Happens at Runtime: How the View Criteria Is Applied to a View Object

When you apply a view criteria containing one or more view criteria rows to a view object, the next time it is executed it augments its SQL query with an additional `WHERE` clause predicate corresponding to the Query-by-Example criteria that you've populated in the view criteria rows. As shown in Figure 5–43, when you apply a view criteria containing multiple view criteria rows, the view object augments its design time `WHERE` clause by adding an additional runtime `WHERE` clause based on the non-`null` example criteria attributes in each view criteria row.

A corollary of the view criteria feature is that each time you apply a new view criteria (or remove an existing one), the text of the view object's SQL query is effectively changed. Changing the SQL query causes the database to reparse the statement the next time it is executed. You can eliminate the reparsing and improve the performance

of a view criteria as described in Section 5.11.10, "What You May Need to Know About Query-by-Example Criteria."

Figure 5–43 View Object Automatically Translates View Criteria Rows into Additional Runtime WHERE Filter



5.11.9 What You May Need to Know About the View Criteria API

When you need to perform tasks that the Edit View Criteria dialog does not support, review the View Criteria API. For example, programmatically, you can alter compound search conditions using multiple view criteria rows, search for a row whose attribute value is `NULL`, search case insensitively, and clear view criteria in effect.

5.11.9.1 Referencing Attribute Names in View Criteria

The `setWhereClause()` method allows you to add a dynamic `WHERE` clause to a view object, as described in Section 6.4.1, "ViewObject Interface Methods for Working with the View Object's Default RowSet." You can also use `setWhereClause()` to pass a string that contains literal database column names like this:

```
vo.setWhereClause("LAST_NAME LIKE UPPER(:NameToFind)");
```

In contrast, when you use the view criteria mechanism, shown in Example 5–35, you must reference the view object attribute name instead, like this:

```
ViewCriteriaItem vc_item1 = vc_row1.ensureCriteriaItem("UserId");
vc_item1.setOperator(">");
vc_item1.setValue("304");
```

The view criteria rows are then translated by the view object into corresponding `WHERE` clause predicates that reference the corresponding column names. The programmatically set `WHERE` clause is `AND`-ed with the `WHERE` clauses of any view criteria that have been defined for the view object at design time.

5.11.9.2 Referencing Bind Variables in View Criteria

When you want to set the value of a view criteria item to a bind variable, use `setIsBindVarValue(true)`, like this:

```
ViewCriteriaItem vc_item1 = vc_row1.ensureCriteriaItem("UserId");
vc_item1.setIsBindVarValue(true);
vc_item1.setValue(":VariableName");
```

5.11.9.3 Altering Compound Search Conditions Using Multiple View Criteria

When you add multiple view criteria, you can call the `setConjunction()` method on a view criteria to alter the conjunction used between the predicate corresponding to that view criteria and the one for the previous view criteria. The legal constants to pass as an argument are:

- `ViewCriteriaComponent.VC_CONJ_AND`
- `ViewCriteriaComponent.VC_CONJ_NOT`
- `ViewCriteriaComponent.VC_CONJ_UNION`
- `ViewCriteriaComponent.VC_CONJ_OR` (*default)*

The `NOT` value can be combined with `AND` or `OR` to create filter criteria like:

```
( PredicateForViewCriteria1 ) AND (NOT ( PredicateForViewCriteria2 ) )
```

or

```
( PredicateForViewCriteria1 ) OR (NOT ( PredicateForViewCriteria2 ) )
```

The syntax to achieve compound search conditions requires using Java's bitwise `OR` operator like this:

```
vc2.setConjunction(ViewCriteriaComponent.VC_CONJ_AND | ViewCriteriaComponent.VC_
CONJ_NOT);
```

> **Performance Tip:** Use the `UNION` value instead of an `OR` clause when the `UNION` query can make use of indices. For example, if the view criteria searches for `sal > 2000 or job = 'CLERK'` this query may turn into a full table scan. Whereas if you specify the query as the union of two inner view criteria, and the database table has an index on `sal` and an index on `job`, then the query can take advantage of these indices and the query performance will be significantly better for a large data set.

The limitation for the `UNION` clause is that it must be defined over one view object. This means that the `SELECT` and the `FROM` list will be the same for inner queries of the `UNION` clause. To specify a `UNION` query, call `setConjunction()` on the outer view criteria like this:

```
vc.setConjunction(ViewCriteriaComponent.VC_CONJ_UNION);
```

The outer view criteria should contain inner queries whose results will be the union. For example, suppose you want to specify the union of these two view criteria:

- A view criteria named `MyEmpJob`, which searches for `Job = 'SALESMAN'`.
- A view criteria named `MyEmpSalary`, which searches for `Sal = 1500`.

To create the `UNION` query for these two view criteria, you would make the calls shown in Example 5–36.

Example 5–36 Applying the Union of Two View Criteria

```
vcu = voEmp.createViewCriteria();
vcm = voEmp.getViewCriteriaManager();

vcu.setConjunction(ViewCriteria.VC_CONJ_UNION);
vcu.add(vcm.getViewCriteria("MyEmpJob"));
vcu.add(vcm.getViewCriteria("MyEmpSal"));

voEmp.applyViewCriteria(vcu);
```

When this view criteria is applied, it will return rows where `Job` is `SALESMAN` or `Sal` is greater than `1500`.

When you use a `UNION` view criteria, be sure that only one of the applied view criteria has the `UNION` conjunction. Other view criteria that you apply will be applied to each inner query of the `UNION` query.

5.11.9.4 Searching for a Row Whose Attribute Value Is NULL Value

To search for a row containing a `NULL` value in a column, populate a corresponding view criteria row attribute with the value "`IS NULL`" or use `ViewCriteriaItem.setOperator("ISBLANK")`.

5.11.9.5 Searching for Rows Whose Attribute Value Matches a Value in a List

To search for all rows with a value in a column that matches any value in a list of values that you specify, populate a corresponding view criteria row attribute with the comma-separated list of values and use the `IN` operator. For example, to filter the list of persons by IDs that match 204 and 206, set:

```
vcr.setAttribute("PersonId","IN (204,206)");
```

Note that there must be a space between the `IN` operator and the brace:

- `IN (204,206)` is correct.
- `IN(204,206)` throws a `SQLSyntaxErrorException` error.

5.11.9.6 Searching Case-Insensitively

To search case-insensitively, call `setUpperColumns(true)` on the view criteria row to which you want the case-insensitivity to apply. This affects the `WHERE` clause predicate generated for `String`-valued attributes in the view object to use `UPPER(COLUMN_NAME)` instead of `COLUMN_NAME` in the predicate. Note that the value of the supplied view criteria row attributes for these `String`-valued attributes must be uppercase or the predicate won't match. In addition to the predicate, it also possible to use `UPPER()` on the value. For example, you can set `UPPER(ename) = UPPER("scott")`.

5.11.9.7 Clearing View Criteria in Effect

To clear any view criteria in effect, you can call `getViewCriteria()` on a view object and then delete all the view criteria rows from it using the `remove()` method, passing the zero-based index of the criteria row you want to remove. If you don't plan to add back other view criteria rows, you can also clear all the view criteria in effect by simply calling `applyViewCriteria(null)` on the view object.

5.11.10 What You May Need to Know About Query-by-Example Criteria

For performance reasons, you want to avoid setting a bind variable as the value of a view criteria item in these two cases:

- In the specialized case where the value of a view criteria item is defined as selectively required and the value changes from non-`NULL` to `NULL`.

  In this case, the SQL statement for the view criteria will be regenerated each time the value changes from non-`NULL` to `NULL`.

- In the case where the value of the view criteria item is optional and that item references an attribute for an indexed column.

  In the case of optional view criteria items, an additional SQL clause `OR (:Variable IS NULL)` is generated, and the clause does not support using column indices.

In either of the following cases, you will get better performance by using a view object whose `WHERE` clause contains the named bind variables, as described in Section 5.10.1, "How to Add Bind Variables to a View Object Definition." In contrast to the view criteria filtering feature, when you use named bind variables, you can change the *values* of the search criteria without changing the *text* of the view object's SQL statement each time those values change.

5.12 Working with List of Values (LOV) in View Object Attributes

Edit forms displayed in the user interface portion of your application can utilize LOV-enabled attributes that you define in the data model project to predetermine a list of values for individual input fields. When the user submits the form with their selected values, ADF data bindings in the ADF Model layer update the value on the view object attributes corresponding to the databound fields. To facilitate this common design task, ADF Business Components provides declarative support to specify the LOV usage in the user interface.

Defining an LOV for attributes of a view object in the data model project greatly simplifies the task of working with list controls in the user interface. Because you define the LOV on the individual attributes of the view object, you can customize the LOV usage for an attribute once and expect to see the list component in the form wherever the attribute appears.

> **Note:** In order for the LOV to appear in the UI, the LOV usage must exist before the user interface designer creates the databound form. Defining an LOV usage for an attribute referenced by an existing form will not change the component that the form displays to an LOV.

You can define an LOV for any view object attribute that you anticipate the user interface will display as a selection list. The characteristics of the attribute's LOV definition depend on the requirements of the user interface. The information you gather from the user interface designer will determine the best solution. For example, you might define LOV attributes in the following cases:

- When you need to display attribute values resulting from a view object query against a business domain object.

  For example, define LOV attributes to display the list of suppliers in a purchase order form.

- When you want to display attribute values resulting from a view object query that you wish to filter using a parameter value from any attribute of the LOV attribute's current row.

  For example, define LOV attributes to display the list of supplier addresses in a purchase order form but limit the addresses list based on the current supplier.

  If you wish, you can enable a second LOV to drive the value of the parameter based on a user selection. For example, you can let the user select the current supplier to drive the supplier addresses list. In this case, the two LOVs are known as a *cascading list*.

Before you can define the LOV attribute, you must create a data source view object in your data model project that queries the eligible rows for the attribute value you want the LOV to display. After this, you work entirely on the base view object to define the LOV. The base view object is the one that contains the primary data for display in the user interface. The LOV usage will define the following additional view object metadata:

- A view accessor to access the data source for the LOV attribute. The view accessor is the ADF Business Components mechanism that lets you obtain the full list of possible values from the row set of the data source view object.
- Optionally, supplemental values that the data source may return to attributes of the base view object other than the data source attribute for which the list is defined.
- User interface hints, including the type of list component to display, attributes to display from the current row when multiple display attributes are desirable, and a few options specific to the choice list component.

The general process for defining the LOV-enabled attribute relies on the Create List of Values dialog that you display for the base view object attribute.

To define the LOV-enabled attribute, follow this general process:

1. Open the Create List of Values dialog for the base attribute.
2. Create a new view accessor definition to point to the data source view object or select an existing view accessor that the base view object already defines.

   Optionally, you can filter the view accessor by creating a view criteria using a bind variable that obtains its value from any attribute of base view object's current row.
3. Select the list attribute from the view accessor's data source view object.

   This maps the attribute you select to the current attribute of the base view object.
4. Optionally, select list return values to map any supplemental values that your list returns to the base view object.
5. Select user interface hints to specify the list's display features.
6. Save the attribute changes.

Note: If you create a view criteria to filter the data source view object, you may also set an LOV on the attribute of the base view object that you use to supply the value for the view criteria bind variable. You set cascading LOV lists when you want the user's selection of one attribute to drive the options displayed in a second attribute's list.

Once you create the LOV-enabled attribute, the user interface designer can create the list component in the web page by dragging the LOV-enabled attribute's collection from the Data Controls panel. For further information about creating a web page that display the list, see Chapter 30, "Creating Databound Selection Lists and Shuttles." Specifically, for more information about working with LOV-enabled attributes in the web page, see Section 30.3.1, "How to Create a Model-Driven List."

5.12.1 How to Define a Single LOV-Enabled View Object Attribute

When an edit form needs to display a list values that is not dependent on another selection in the edit form, you can define a view accessor to point to the list data source. For example, assume that a purchase order form contains a field that requires the user to select the order item's supplier. In this example, you would first create a view accessor that points to the data source view object (`SuppliersView`). You would then set the LOV on the `SupplierDesc` attribute of the base view object (`PurchaseOrdersView`). Finally, you would reference that view accessor from the LOV-enabled attribute (`SupplierDesc`) of the base view object and select the data source attribute (`SupplierDesc`).

You will use the Create List of Values dialog to define an LOV-enabled attribute for the base view object. The dialog lets you select an existing view accessor or create a new one to save with the LOV-attribute definition.

Before you begin:

It may be helpful to have an understanding of LOV-enabled attributes. For more information, see Section 5.12, "Working with List of Values (LOV) in View Object Attributes."

You will need to complete this task:

Create the desired view objects as described in Section 5.2.1, "How to Create an Entity-Based View Object," and Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object."

To define an LOV that displays values from a view object attribute:

1. In the Application Navigator, double-click the view object that contains the attribute you wish to enable as an LOV.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the attribute that is to display the LOV, and then click the **List of Values** tab and click the **Add list of values** button.

   Use the Create List of Values dialog to create the LOV on the attribute you have currently selected in the attribute list of the overview editor. JDeveloper assigns a unique name to identify the LOV usage. For example, the metadata for the attribute `SupplierDesc` will specify the name `SupplierDescLOV` to indicate that the attribute is LOV enabled.
4. In the Create List of Values dialog, click the **Create new view accessor** button to add a view accessor to the view object you are currently editing.

   Alternatively, you can expand **List Data Source** and select among the existing view accessors. The dropdown list displays all view accessors that you have added to the view object you are editing.
5. In the View Accessors dialog, select the view object definition or shared view instance that defines the data source for the attribute and shuttle it to the view accessors list.

By default, the view accessor you create will display the same name as the view object. You can edit the accessor name to supply a unique name. For example, assign the name `SuppliersViewAccessor` for the `SuppliersView` view object.

The view instance is a view object usage that you have defined in the data model of a shared application module. For more information about using shared view instances in an LOV, see Section 10.4.5, "How to Create an LOV Based on a Lookup Table."

6. Click **OK** to save the view accessor definition for the view object.
7. In the Create List of Values dialog, expand **List Data Source** and select the view accessor you created for the base view object to use as the data source. Then select the same attribute from this view accessor that will provide the list data for the LOV-enabled attribute.

   The editor creates a default mapping between the list data source attribute and the LOV-enabled attribute. For example, the attribute `SuppliersDesc` from the `PurchaseOrdersView` view object would map to the attribute `SuppliersDesc` from the `SuppliersViewAccessor` view accessor.

   The editor does not allow you to remove the default attribute mapping for the attribute for which the list is defined.
8. Optionally, when you want to specify supplemental values that your list returns to the base view object, click the **Create return attribute map** button in the **List Return Values** section and map the desired base view object attributes with attributes accessed by the view accessor.

   Supplemental attribute return values are useful when you do not require the user to make a list selection for the attributes, yet you want those values, as determined by the current row, to participate in the update. For example, to map the attribute `SupplierAddress` from the `PurchaseOrdersView` view object, you would choose the attribute `SupplierAddress` from the `SuppliersViewAccessor` view accessor.
9. Click **OK**.

5.12.2 How to Define Cascading Lists for LOV-Enabled View Object Attributes

When the application user interface requires a list of values in one input field to be dependent on the user's entry in another field, you can create attributes that will display as cascading lists in the user interface. In this case, the list of possible values for the LOV-enabled attributes might be different for each row. As the user changes the current row, the LOV values vary based on the value of one or more controlling attribute values in the LOV-enabled attribute's view row. To apply the controlling attribute to the LOV-enabled attribute, you will create a view accessor to access the data source view object with the additional requirement that the accessor filters the list of possible values based on the current value of the controlling attribute. To filter the LOV-enabled attribute, you can edit the view accessor to add a named view criteria with a bind variable to obtain the user's selection.

For example, assume that a purchase order form contains a field that requires the user to select the supplier's specific site and that the available sites will depend on the order's already specified supplier. To implement this requirement, you would first create a view accessor that points to the data source view object. The data source view object will be specific to the LOV usage, because it must perform a query that filters the available supplier sites based on the user's supplier selection. You might name this data source view object definition `SupplierIdsForCurrentSupplierSite` to help

distinguish it from the SupplierSitesView view object that the data model already contains. The data source view object will use a named view criteria (SupplierCriteria) with a single view criteria item set by a bind variable (TheSupplierId) to obtain the user's selection for the controlling attribute (SupplierId).

You would then set the LOV on the SupplierSiteId attribute of the base view object (PurchaseOrdersView). You can then reference the view accessor that points to the data source view object from the LOV-enabled attribute (PurchaseOrdersView.SupplierSiteId) of the base view object. Finally, you must edit the LOV-enabled attribute's view accessor definition to specify the corresponding attribute (SupplierIdsForCurrentSupplierSite.SupplierSiteId) from the view object as the data source and, importantly, source the value of the bind variable from the view row's result using the attribute SupplierId.

To define cascading lists for LOV-enabled view object attributes:

1. Create a data source view object to control the cascading list.
2. Create a view accessor to filter the cascading list.

5.12.2.1 Creating a Data Source View Object to Control the Cascading List

The data source view object defines the controlling attribute for the LOV-enabled attribute. To make the controlling attribute accessible to the LOV-enabled attribute of the base view object, you must define a named view criteria to filter the data source attribute based on the value of another attribute. Because the value of the controlling attribute is expected to change at runtime, the view criteria uses a bind variable to set the controlling attribute.

Before you begin:

It may be helpful to have an understanding of cascading LOV-enabled attributes. For more information, see Section 5.12.2, "How to Define Cascading Lists for LOV-Enabled View Object Attributes."

To define the view criteria for the data source to be referenced by the LOV-enabled attribute:

1. In the Application Navigator, double-click the view object that you created to query the list of all possible values for the controlling attribute.

   For example, if the LOV-enabled attribute SupplierSiteId depends on the controlling attribute SupplierId value, you might have created the data source view object SupplierIdsForCurrentSupplierSite to query the list of all supplier sites.

2. In the overview editor, click the **Query** navigation tab.
3. In the Query page, expand the **Bind Variables** section and click the **Create new bind variable** button to add a bind variable to the data source view object.

   For example, for a data source view object SupplierIdsForCurrentSupplierSite used to query the list of all supplier sites, you would create the bind variable TheSupplierId, since it will be the controlling attribute for the LOV-enabled attribute.

4. In the Bind Variable dialog, enter the name and type of the bind variable. Leave all other options unchanged and click **OK**.

By default, the view accessor you create will display the same name as the view object instance. You can edit the accessor name to supply a unique name. For example, assign the name `CurrencyLookupViewAccessor` for the `CurrencyLookupView` view object instance.

5. In Query page of the overview editor, expand the **View Criteria** section and click the **Create new view criteria** button to add the view criteria to the data source view object you are currently editing.
6. In the Create View Criteria dialog, click **Add Group** and define a single **Criteria Item** for the group as follows:
   - Enter a **Criteria Name** to identify the view criteria. For example, you might enter the name `SupplierCriteria` for the `SupplierIdsForCurrentSupplierSite`.
   - Select the controlling attribute from the **Attributes** list. For example, you would select the `SupplierSiteId` attribute from the `SupplierIdsForCurrentSupplierSite`.
   - Select **equal to** from the view criteria **Operator** list.
   - Select **Bind Variable** from the view criteria **Operand** list.
   - Select the name of the previously defined bind variable from the **Parameter** list.
   - Select among the following bind variable configuration options to determine whether or not the value is required by the parent LOV:

     Optional from the **Validation** menu and deselect **Ignore Null Values** when you want to configure cascading LOVs where the parent LOV value is required. This combination supports the cascading LOV use case where no selection in the parent LOV returns *no* rows in the child LOV. The `WHERE` clause shown in the Edit View Criteria dialog should look similar to `((SupplierIdsForCurrentSupplierSite.SUPPLIER_ID = :TheSupplierId)).`

     Optional from the **Validation** menu and leave **Ignore Null Values** selected (default) when you want to configure cascading LOVs where the parent LOV value is optional. This combination supports the cascading LOV use case where no selection in the parent LOV returns *all* rows in the child LOV. The `WHERE` clause shown in the Edit View Criteria dialog should look similar to `(((SupplierIdsForCurrentSupplierSite.SUPPLIER_ID = :TheSupplierId) OR ( :TheSupplierId IS NULL ))).`

     For more details about these settings, see Section 5.11.3, "What You May Need to Know About Bind Variable Options." Do not select **Required** for the Validation option for cascading LOVs, because no selection in the parent LOV will cause a validation error.
7. Click **OK**.

5.12.2.2 Creating a View Accessor to Filter the Cascading List

To populate the cascading LOV-enabled attribute, you must first set up a named view criteria on a data source view object. To make the LOV-enabled attribute of the base view object dependent on the controlling attribute of the data source view object, you then add a view accessor to the LOV-enabled attribute of the base view object and reference the previously defined data source view object's named view criteria.

Before you begin:

It may be helpful to have an understanding of cascading LOV-enabled attributes. For more information, see Section 5.12.2, "How to Define Cascading Lists for LOV-Enabled View Object Attributes."

You will need to complete this task:

Create the data source view object and named view criteria as described in Section 5.12.2.1, "Creating a Data Source View Object to Control the Cascading List."

To create a view accessor that filters display values for an LOV-enabled attribute based on the value of another attribute in the same view row:

1. In the Application Navigator, double-click the base view object that contains the attribute you want to use the filtered view accessor as the list data source.

   For example, the base view object `PurchaseOrdersView` might contain the attribute `SupplierSiteId` that will depend on the value of the controlling attribute `SupplierId`.

2. In the overview editor, click the **Attributes** navigation tab.

3. In the Attributes page, select the attribute that is to filter the cascading LOV, and then click the **List of Values** tab and click the **Add list of values** button.

4. In the Create List of Values dialog, click the **Create new view accessor** button to add a view accessor to the view object you are currently editing.

   Alternatively, you can expand **List Data Source** and select among the existing view accessors. The dropdown list displays all view accessors that you have added to the view object you are editing.

5. In the View Accessors dialog, select the view object instance name you created for data source view object and shuttle it to the view accessors list.

6. With the new view accessor selected in the dialog, click **Edit**.

7. In the Edit View Accessor dialog, apply the previously defined view criteria to the view accessor and provide a value for the bind variable as follows:

   - Click the data source view object's view criteria in the **Available** list and add it to the Selected list. For example, you would select `SupplierCriteria` from the `SupplierIdsForCurrentSupplierSite` view object definition.
   - Set the value for the bind variable to the name of the controlling attribute. The attribute name must be identical to the base view object's controlling attribute. For example, if the base view object `PurchaseOrdersView` contains the LOV-enabled attribute `SupplierSiteId` that depends on the value of the controlling attribute `SupplierId`, you would enter `SupplierId` for the bind variable value.
   - Select the name of the previously defined bind variable from the **Parameter** list.
   - Select **Required** from the **Usage** dropdown list.

8. Click **OK** to save the view accessor definition for the base view object.

9. In the Attributes page of the overview editor, select the attribute that is to display the LOV, and then click the **List of Values** tab and click the **Add list of values** button.

10. In the Create List of Values dialog, expand **List Data Source** and select the view accessor you created for the data source view object instance to use as the data source. Then select the controlling attribute from this view accessor that will serve to filter the attribute you are currently editing.

    The editor creates a default mapping between the view object attribute and the LOV-enabled attribute. You use separate attributes in order to allow the bind variable (set by the user's controlling attribute selection) to filter the LOV-enabled attribute. For example, the LOV-enabled attribute `SupplierId` from the `PurchaseOrdersView` view object would map to the controlling attribute `SupplierSiteId` for the `SupplierIdsForCurrentSupplierSiteViewAccessor`. The runtime automatically supports these two cascading LOVs where the row set and the base row attribute differ.

11. Click **OK**.

5.12.3 How to Specify Multiple LOVs for an LOV-Enabled View Object Attribute

Another way to vary the list of values that your application user interface can display is to define multiple list of values for a single LOV-enabled view object attribute. In contrast to a cascading list, which varies the list contents based on a dependent LOV list selection, an LOV-enabled switcher attribute with multiple LOVs lets you vary the entire LOV itself. The LOV choice to display is controlled at runtime by the value of an attribute that you have defined specifically to resolve to the name of the LOV to apply.

For example, you might want to define one LOV to apply in a create or edit form and another LOV to apply for a search component. In the first case, the LOV-enabled attribute that the form can use is likely to be an entity-based view accessor that is shared across all the view objects that reference the entity. The entity-based view accessor is useful for user interface forms because a single accessor definition can apply to each instance of the LOV in the forms. However, in the case of the search component, LOV definitions based on view accessors derived from an underlying entity will not work. The LOV definitions for search components must be based on view accessors defined in the view object. Note that when the user initiates a search, the values in the criteria row will be converted into `WHERE` clause parameters. Unlike a regular view row displayed in create or edit type forms, the criteria row is not backed by an entity. In this scenario, one LOV uses the entity-based accessor as a data source and a second LOV uses the view object-based accessor as a data source.

To address this requirement to define multiple LOV lists that access the same attribute, you add a switcher attribute to the base view object. For example, you might add a `ShipperLOVSwitcher` attribute for the `Orders` view object that resolves through an expression to the name of the LOV to display. Such an expression can specify two LOVs that may apply to the `ShipperID` attribute:

```
(adf.isCriteriaRow) ? "LOV_ShipperID_ForSearch" : "LOV_
ShipperID"
```

This expression would appear in the **Value** field of the switcher attribute. At runtime, in the case of the search component, the expression resolves to the value that identifies the view object-based accessor LOV. In the case of the create or edit form, the expression resolves to the value that identifies the entity-based accessor LOV.

You will use the Create List of Values dialog to add multiple LOV lists to an attribute of the base view object. You will also use the List of Values section in the Attributes page of the overview editor for the base view object to define the default LOV to display and the switcher attribute to apply.

Before you begin:

It may be helpful to have an understanding of LOV-enabled attributes. For more information, see Section 5.12, "Working with List of Values (LOV) in View Object Attributes."

You will need to complete this task:

Create the first LOV list for the attribute as described in Section 5.12.1, "How to Define a Single LOV-Enabled View Object Attribute."

To specify additional LOV lists for a view object attribute with an existing LOV:

1. In the Application Navigator, double-click the view object that contains the attribute for which you want to specify multiple LOV lists.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the desired attribute, and then click the **List of Values** tab and click the **Add list of values** button.
4. In the Create List of Values dialog, define the first LOV as described in Section 5.12.1, "How to Define a Single LOV-Enabled View Object Attribute."

   When you define the LOV, change the name of the LOV to match the value returned by the attribute that you will use to determine which LOV your application applies to the LOV-enabled attribute.
5. After you define the first LOV, return to the **List of Values** section of the Attributes page of the overview editor and, with the original attribute selected, click the **Add List of Values** button.

   If you have selected the correct attribute from the Attributes page of the overview editor, the **List of Values** section should display your previously defined LOV.
6. In the Create List of Values dialog, repeat the procedure described in Section 5.12.1, "How to Define a Single LOV-Enabled View Object Attribute" to define each subsequent LOV.

   The name of each LOV must correspond to a unique value returned by the attribute that determines which LOV to apply to the LOV-enabled attribute.

   You can define the LOV using any accessor and any attribute. There are no restrictions on the type of LOV lists that you can add to an attribute with multiple LOV lists specified.

   After you finish defining the second LOV, the **List of Values** section in the view object overview editor changes to display additional features that you will use to control the selection of the LOV.
7. In the Attributes page of the overview editor, click the **List of Values** tab and use the **List of Values Switcher** dropdown list to select the attribute that will return the name of the List of Value to use.

   The dropdown list displays the attributes of the base view object. If you want your application to dynamically apply the LOV from the LOVs you have defined, your view object must define an attribute whose values resolve to the names of the LOVs you defined. If you have not added this attribute to the view object, be sure that the dropdown list displays **<None Specified>**. In this case, at runtime your application will display the LOV-enabled attribute with the default LOV and it will not be possible to apply a different LOV.
8. To change the default LOV to apply at runtime, choose the **Default** radio button corresponding to the desired LOV definition.

The default LOV selection determines which list of values your application will display when the **List of Values Switcher** dropdown list displays **<None Specified>**. Initially, the first LOV in the overview editor **List of Values** section is the default.

9. To change the component that your application will use to display the various LOV lists, select from desired component from the **List Type UI Hint** dropdown list.

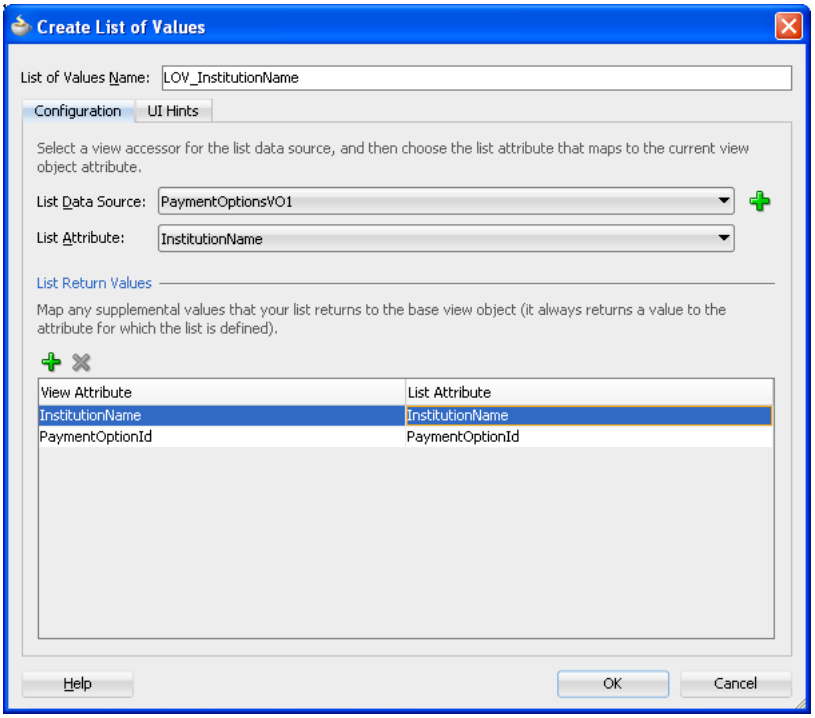
   The component you select will apply to all LOV lists. For a description of the available components, see Table 5–2.

5.12.4 How to Define an LOV to Display a Reference Attribute

Reference attributes that your view objects define are often desirable attributes to use as the source for LOV lists. Reference attributes belong to secondary entity usages that you have added to the view object to provide meaningful information beyond the entity usage's primary key attribute. For example, when you create an `OrderInfo` view object, the view object may define a secondary entity usage for `PaymentOptionsEO` to include billing information for the order, including the list of credit cards the end user has added to their account.

An LOV that you define for the secondary entity object needs to be able to update the primary attribute value, but to be meaningful to the end user, you typically hide the primary attribute in the list and display one or more reference attributes instead. In this case, your LOV might display the list of credit cards by institution name, but hide the payment option ID value that gets updated by the end user's selection. Figure 5–44 shows the LOV defined on the reference attribute `InstitutionName`. The **List Return Values** section of the Create List of Values dialog lists the reference attribute and then lists the primary key attribute `PaymentOptionId` to provide supplemental values.

Figure 5–44 Create View Criteria Dialog with Reference Attribute Specified



Before you begin:

It may be helpful to have an understanding of reference attributes in secondary entity usages. For more information, see Section 5.12, "Working with List of Values (LOV) in View Object Attributes."

You will need to complete this task:

Create the entity-based view object and add a secondary entity usage that defines the desired LOV attributes, as described in Section 5.12.1, "How to Define a Single LOV-Enabled View Object Attribute." By default, the secondary entity usage will include the primary key attribute that you want the LOV to update.

To define reference attributes for an LOV:

1. In the Application Navigator, double-click the view object that contains the secondary entity usage attribute you wish to enable as an LOV.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the desired reference attribute that is to display the LOV, click the **List of Values** tab, and then click the **Add list of values** button.

   Use the Create List of Values dialog to create the LOV on the attribute you have currently selected in the attribute list of the overview editor. JDeveloper assigns a unique name to identify the LOV usage. For example, the metadata for the attribute `SupplierDesc` will specify the name `SupplierDescLOV` to indicate that the attribute is LOV-enabled.
4. In the Create List of Values dialog, click the **Create new view accessor** button to add a view accessor to the view object you are currently editing.

   Alternatively, you can expand **List Data Source** and select among the existing view accessors. The dropdown list displays all the view accessors that you have added to the view object you are editing.
5. In the View Accessors dialog, select the view object definition or shared view instance that defines the data source for the attribute and shuttle it to the view accessors list.

   By default, the view accessor you create will display the same name as the view object. You can edit the accessor name to supply a unique name. For example, assign the name `SuppliersViewAccessor` for the `SuppliersView` view object.
6. Click **OK** to save the view accessor definition for the view object.
7. In the Create List of Values dialog, expand **List Data Source** and select the view accessor you created for the base view object to use as the data source. Then select the same attribute from this view accessor that will provide the list data for the LOV-enabled attribute.

   The editor creates a default mapping between the list data source attribute and the LOV-enabled attribute. For example, the attribute `SuppliersDesc` from the `PurchaseOrdersView` view object would map to the attribute `SuppliersDesc` from the `SuppliersViewAccessor` view accessor.

   The editor does not allow you to remove the default attribute mapping for the attribute for which the list is defined.
8. Optionally, when you want to specify supplemental values that your list returns to the base view object, click the **Create return attribute map** button in the **List Return Values** section and map the desired base view object attributes with attributes accessed by the view accessor.

Supplemental attribute return values are useful when you do not require the user to make a list selection for the attributes, yet you want those values, as determined by the current row, to participate in the update. For example, to map the attribute `SupplierAddress` from the `PurchaseOrdersView` view object, you would choose the attribute `SupplierAddress` from the `SuppliersViewAccessor` view accessor.

9. Click **OK**.

5.12.5 How to Set User Interface Hints on a View Object LOV-Enabled Attribute

When you know how the view object attribute that you define as an LOV should appear in the user interface, you can specify additional properties of the LOV to determine its display characteristics. These properties, or *UI hints*, augment the attribute hint properties that ADF Business Components lets you set on any view object attribute. Among the LOV UI hints for the LOV-enabled attribute is the type of component the user interface will use to display the list. For a description of the available components, see Table 5–2. (Not all ADF Faces components support the default list types, as noted in the Table 5–2.)

Table 5–2 List Component Types for List Type UI Hint

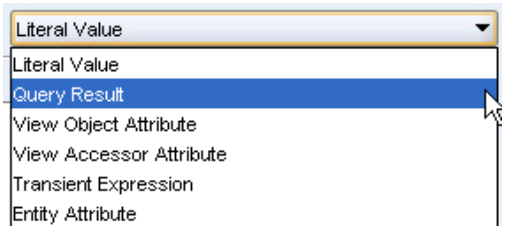
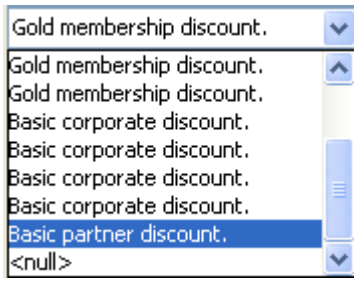
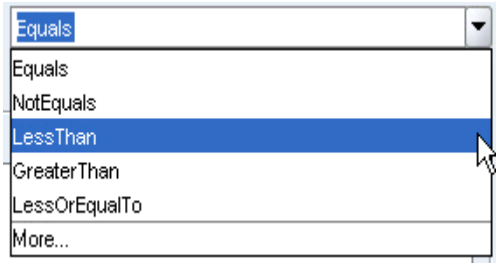
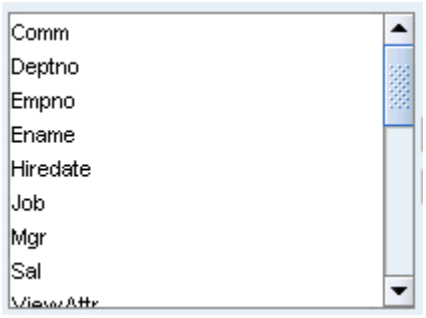
LOV List Component Type	Usage
Choice List 	This component does not allow the user to type in text, only select from the dropdown list.
Combo Box 	This component allows the user to type text or select from the dropdown list. This component sometimes supports auto-complete as the user types. This component is *not* supported for ADF Faces.
Combo Box with List of Values 	This component is the same the as the combo box, except that the last entry (More...) opens a List of Values lookup dialog that supports query with filtering when enabled for the LOV attribute in its UI hints. The default UI hint enables queries on all attributes. Note that when the LOV attribute appears in a table component, the list type changes to an **Input Text with List of Values** component.

Table 5–2 (Cont.) List Component Types for List Type UI Hint

LOV List Component Type	Usage
Input Text with List of Values 	This component displays an input text field with an LOV button next to it. The List of Values lookup dialog opens when the user clicks the button or enters an invalid value into the text field. The List of Values lookup dialog for this component supports query with filtering when enabled in the UI hints for the LOV attribute. The default UI hint enables queries on all attributes. This component may also support auto-complete when a unique match exists.
List Box 	This component takes up a fixed amount of real estate on the screen and is scrollable (as opposed to the choice list, which takes up a single line until the user clicks on it).
Radio Group 	This component displays a radio button group with the selection choices determined by the LOV attribute values. This component is most useful for very short, fixed lists.

Before you begin:

It may be helpful to have an understanding of LOV-enabled attributes. For more information, see Section 5.12, "Working with List of Values (LOV) in View Object Attributes."

You will need to complete this task:

Create the LOV list for the attribute as described in Section 5.12.1, "How to Define a Single LOV-Enabled View Object Attribute."

To set view object attribute UI hints for an LOV-enabled attribute:

1. In the Application Navigator, double-click the view object that contains the attribute that you want to customize.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the desired attribute and then click the **List of Values** tab.
4. In the List of Values section, select the LOV list that you want to customize and click the **Edit list of values** button.
5. In the Edit List of Values dialog, click the **UI Hints** tab.

6. In the UI Hints page, select a default list type as the type of component to display the list.

   For a description of the available components, see Table 5–2.

   The list component displayed by the web page and the view object's default list type must match at runtime or a method-not-found runtime exception results. To avoid this error, confirm the desired list component with the user interface designer. You can also edit the default list type to match, so that, should the user interface designer subsequently change the component used in the web page, the two stay in sync.

7. Optionally, select additional display attributes to add values to the display.

   The list of additional attributes is derived from the LOV-enabled attribute's view row. The additional attribute values can help the end user select an item from the list.

8. If you selected the **Combo Box with List of Values** type component, by default, the dropdown list for the component will display the first 10 records from the data source. This limit also serves to keep the view object fetch size small. To change the number of records the dropdown list of a Combo Box with List of Values component can display, enter the number of records for **Query Limit**.

   Because **Query Limit** also controls the number of rows the view object will fetch (its sets the view object definition `ListRangeSize` property), specifying a large value for **Query Limit** is not recommended. The end user can open the component's LOV lookup dialog to access the full set of records (by clicking the component's lookup icon). **Query Limit** is disabled for all other component types and those components place no restriction on the number of rows that the LOV will access.

   For details about the `ListRangeSize` property, see Section 5.12.10, "What Happens at Runtime: How an LOV Queries the List Data Source."

9. If you selected a component type that allows the user to open a List of Values lookup dialog to select a list value (this includes either the **Combo Box with List of Values** type component or **Input Text with List of Values** type component), by default, the lookup dialog will display a search form that will allow the user to search on all queryable attributes of the data source view object (the one defined by the LOV-enabled attribute's view accessor). Decide how you want to customize these components.

   a. When you select the **Combo Box with List of Values** type component and you have added a large number of attributes to the **Selected** list, use **Show in Combo Box** to improve the readability of the dropdown list portion of the component. To limit the attribute columns to display in the dropdown list that the Combo Box with List of Values component displays, choose **First** from **Show in Combo Box** and enter a number corresponding to the number of attributes from the top of the **Selected** list that you want the dropdown list to display (this combination means you are specifying the "first" *x* number of attributes to display from the Create List of Values dialog's **Selected** list). Limiting the number of attribute columns to display in the dropdown list ensures that the user does not have to horizontally scroll to view the full list, but it does not limit the number of attribute columns to display in the List of Values lookup dialog. This option is disabled for all list component types except Combo Box with List of Values.

   b. You can limit the attributes to display in the List of Values lookup dialog by selecting a view criteria from the **Include Search Region** dropdown list. To

appear in the dropdown list, the view criteria must already have been defined on the data source view object (the one that the LOV-enabled attribute's view accessor defines). Click the **Edit View Criteria** button to set search form properties for the selected view criteria. For more information about customizing view criteria for search forms, see Section 5.11.5, "How to Set User Interface Hints on View Criteria to Support Search Forms."

c. You can prepopulate the results table of the List of Values lookup dialog by selecting **Query List Automatically**. The List of Values lookup dialog will display the results of the query when the user opens the dialog. If you leave this option deselected, no results will be displayed until the user submits the search form.

10. Alternatively, if you prefer not to display a search region in the List of Values lookup dialog, select **<No Search>** from the **Include Search Region** dropdown list. In this case, the List of Values lookup dialog will display only attributes you add to the **Display Attributes** list.

11. If you selected a choice type component to display the list, you can specify a **Most Recently Used Count** as an alternative to displaying all possible values.

    For example, your form might display a choice list of `SupplierId` values to drive a purchase order form. In this case, you can allow the user to select from a list of their most recently viewed suppliers, where the number of supplier choices is determined by the count you enter. The default count `0` (zero) for the choice list displays all values for the attribute.

12. If you selected a **Combo Box with List of Values** type component to display the list, you can select a view criteria from the **Filter Combo Box Using** dropdown list to limit the list of valid values the LOV will display.

    When you enable **Filter Combo Box Using**, the dropdown list displays the existing view criteria from the view object definition related to the LOV's view accessor. If the dropdown list displays no view criteria, then the data source view object defines no view criteria. When you do not enable this feature, the Combo Box with List of Values component derives its values from the full row set returned by the view accessor. The filtered Combo Box with List of Values is a useful feature when you want to support the use of an LOV with popup search dialog or LOV with a dropdown list that has a limited set of valid choices. For details about using the Combo Box with List of Values component in user interfaces, see Section 31.1.2, "List of Values (LOV) Input Fields."

13. Decide how you want the list component to handle a `NULL` value choice to display in the list component. This option is not enabled for every list component type that you can select.

    If you enable **Include "No Selection" Item**, you can also determine how the `NULL` value selection should appear in the list by making a selection from the dropdown list. For example, when you select **Labeled Item**, you can enter the desired label in the edit field to the right of the dropdown list or you can click the **...** button (to the right of the edit field) to select a message string from the resource bundle associated with the view object. When you select a message string from the resource bundle, JDeveloper saves the string's corresponding message key in the view object definition file. At runtime, the UI locates the string to display based on the current user's locale setting and the message key in the localized resource bundle.

14. Click **OK**.

5.12.6 How to Handle Date Conversion for List Type UI Components

When the LOV-enabled attribute of the view object is bound to date information (such as the attribute `OrderShippedDate`), by default Oracle ADF assumes a format for the field like `yyyy-MM-dd hh:mm:ss`, which combines date and time. This combined date-time format is specified by the ADF Business Components Date domain class (`jbo.domain.Date`) and creates a conversion issue for the ADF Faces component when the user selects a date supplied by the LOV-enable attribute. When the ADF Faces component is unable to convert the domain type to the Date type, the user interface invalidates the input field and displays the message `Error: The date is not in the correct format`.

To avoid this potential conversion error, configure a UI hint setting for the date value attribute of the view object that you want to enable for an LOV. The UI hint you specify will define a date-only mask, such as `yyyy-MM-dd`. Subsequently, any ADF Faces component that references the attribute will perform the conversion based on a pattern specified by its EL value-binding expression (such as `#{bindings.Hiredate.format}`) and will reference the UI hint format instead of the ADF Business Components domain date-time. The conversion error results when the EL expression evaluates to null because no format mask has been specified.

Before you begin:

It may be helpful to have an understanding of LOV-enabled attributes. For more information, see Section 5.12, "Working with List of Values (LOV) in View Object Attributes."

You may also find it helpful to understand support for UI hints at the level of view objects. For more information about UI hints, see Section 5.13, "Defining UI Hints for View Objects."

You will need to complete this task:

Create the LOV list for the attribute as described in Section 5.12.1, "How to Define a Single LOV-Enabled View Object Attribute."

To set a UI hint to match the date format for the LOV-enable attribute:

1. In the Application Navigator, double-click the view object.
2. In the overview editor, click the **Attributes** navigation tab.

   Alternatively, display the Property Inspector for the selected attribute and click the **UI Hints** navigation tab. The Property Inspector provides a way to customize the attribute's UI hints without using the overview editor.
3. In the Attributes page, select the date-value attribute that you want to customize with UI hints, and then click the **UI Hints** tab.
4. In the UI Hints section, select **Simple Date** for the **Format Type** and choose the format with the date only mask.

   Mapping of the ADF Business Components domain type to its available formatters is provided in the `formatinfo.xml` file in the BC4J subdirectory of the JDeveloper system directory (for example, `C:\Documents and Settings\<username>\Application Data\JDeveloper\system<version#>\o.BC4J.\formatinfo.xml`).

5.12.7 How to Automatically Refresh the View Object of the View Accessor

If you need to ensure that your view accessor always queries the latest data from the database table, you may be able to set the **Auto Refresh** property on the data source

view object. This property allows the view object instance to refresh itself after a change in the database. You can enable this feature for any view instance that your application modules define. Once you enable this property on a view object, it ensures that changes a user commits to a database table will become available to any other user working with the same database table. A typical use case is to enable auto refresh for the data source view object when you define a view accessor for a LOV-enabled view object attribute.

Because the auto-refresh feature relies on the database change notification feature, observe these restrictions when enabling auto-refresh for your view object:

- The view objects should query as few read-only tables as possible. This will ensure the best performance and prevent the database invalidation queue from becoming too large.
- The application module that contains updateable, auto-refresh view instances must be configured to lock rows during updates.
- The database user must have database notification privileges. For example, to accomplish this with a SQL*Plus command use `grant change notification to <user name>`.

When you enable auto refresh for the view object, at runtime, prior to executing the view object query, the framework will use the JDBC API to register the view object query to receive Oracle database change notifications for underlying data changes. When the view object receives a notification (because its underlying data has changed), the row sets of the view object are marked as dirty and the framework will refresh the row set on the next server trip from the client to the middle tier. At that point, the dirty collections will be discarded and the request for the updated data will trigger a new query execution by the view object. For example, assume that a user can create or edit a calendar entry but cannot edit calendar entries added by other users. When the user creates a new entry, then in the same server trip the calendar entries that other users modified or entered will be updated. But when another user creates a calendar entry, the view object receives a notification and waits for the next server trip before it refreshes itself; the delay to perform the update prevents contention among various users to read the same data.

> **Best Practice:** Use optimistic row locking for web applications. Optimistic locking, the default configuration setting, assumes that multiple transactions can complete without affecting each other. Optimistic locking therefore allows auto-refresh to proceed without locking the rows being refreshed. Pessimistic row locking, prevents the row set refresh and causes the framework to throw an exception anytime the row set has a transaction pending (for example, a user may be in the process of adding a new row). To ensure that the application module configuration uses optimistic row locking, open the **Properties** tab of the Business Components Configuration dialog and confirm the property `jbo.locking.mode` is set to `optimistic`.

Before you begin:

It may be helpful to have an understanding of LOV-enabled attributes. For more information, see Section 5.12, "Working with List of Values (LOV) in View Object Attributes."

To register a view object to receive data change notifications:

1. In the Application Navigator, double-click the view object that you want to receive database change notifications.
2. In the Property Inspector expand the **Tuning Database Retrieve** section, and select **True** for the **Auto Refresh** property.

5.12.8 How to Test LOV-Enabled Attributes Using the Oracle ADF Model Tester

To test the LOV you created for a view object attribute, use the Oracle ADF Model Tester, which is accessible from the Application Navigator.

The Oracle ADF Model Tester, for any view object instance that you browse, will display any LOV-enabled attributes using one of two component types you can select in the UI Hints page of the List of Values dialog. Currently, only a Choice List component type and Input Text with List of Values component type are supported. Otherwise, the Oracle ADF Model Tester uses the default choice list type to display the LOV-enabled attribute.

Before you begin:

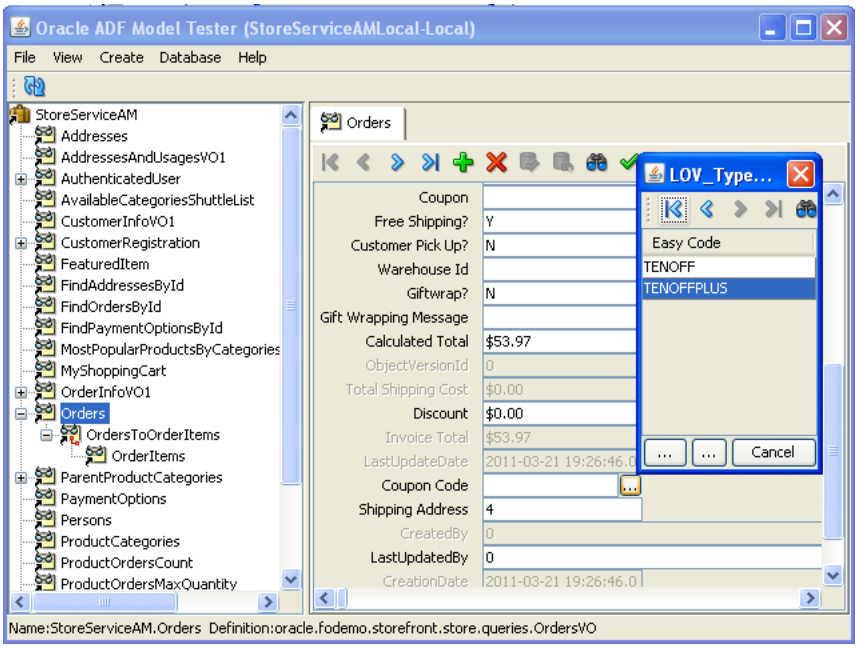
It may be helpful to have an understanding of LOV-enabled attributes. For more information, see Section 5.12, "Working with List of Values (LOV) in View Object Attributes."

To test an LOV using the Oracle ADF Model Tester:

1. In the Application Navigator, expand the project containing the desired application module and view objects.
2. Right-click the application module and choose **Run**.
3. In the Select Business Components Configuration dialog, select the desired application module configuration from the **Configuration Name** list to run the Oracle ADF Model Tester.
4. Click **Connect** to start the application module using the selected configuration.
5. In the Oracle ADF Model Tester, select the desired view object from the section on the left. The Oracle ADF Model Tester displays the LOV-enabled attribute values in a dropdown list unless you specified the component type as an Input Text with List of Value, UI hint.

   Figure 5–45 shows an LOV-enabled attribute, `TypeCouponCode` for the `OrdersVO`, that specifies an input text field and List of Values dialog as the UI hint list type. The Input Text with List of Values component is useful when you want to display the choices in a separate LOV dialog. Other list types are not supported by the Oracle ADF Model Tester.

Figure 5–45 Displaying LOV-Enabled Attributes in the Oracle ADF Model Tester



5.12.9 What Happens When You Define an LOV for a View Object Attribute

When you define an LOV for a view object attribute, the view object metadata defines the following additional information, as shown in Example 5–37 for the `OrdersVO.TypedCouponCode` attribute in the Fusion Order Demo application.

- The `<ViewAttribute>` element names the attribute, points to the list binding element that defines the LOV behavior, and specifies the component type to display in the web page. For example, the LOV-enabled attribute `TypedCouponCode` points to the list binding named `LOV_TypedCouponCode` and defines the `CONTROLTYPE` input text field with list (`input_text_lov`) to display the LOV data.

  When the user interface designer creates the web page using the Data Controls panel, the `<CONTROLTYPE Value="namedType"/>` definition determines the component that JDeveloper will add to the web page. When the component type definition in the data model project does not match the component type displayed in the web page, a runtime exception will result. For more information, see Section 5.12.10, "What Happens at Runtime: How an LOV Queries the List Data Source."

- The `<ListBinding>` element defines the behavior of the LOV. It also identifies a view accessor to access the data source for the LOV-enabled attribute. The view accessor is the ADF Business Components mechanism that lets you obtain the full list of possible values from the row set of the data source view object. For example, `ListVOName="Coupon"` points to the `Coupons` view accessor, which accesses the view object `CouponsVO`.

- The `<ListBinding>` element maps the list data source attribute to the LOV-enabled attribute. For example, the `ListAttrNames` item `EasyCode` is mapped to the LOV-enabled attribute `TypedCouponCode`.

- Optionally, the `<ListBinding>` element defines supplemental values that the data source may return to attributes of the base view object other than the data source attribute for which the list is defined. For example, `DerivedAttrNames`

item `CouponId` is a supplemental value set by the `ListAttrNames` item `DiscountId`.

- The `<ListBinding>` element also identifies one or more attributes to display from the current row and provides a few options that are specific to the choice list type component. For example, the `ListDisplayAttrNames` item `EasyCode` is the only attribute displayed by the LOV-enabled attribute `TypedCouponCode`. In this example, the value `none` for `NullValueFlag` means the user cannot select a blank item from the list.

Example 5–37 View Object MetaData For LOV-Attribute Usage

```
<ViewAttribute
    Name="TypedCouponCode"
    LOVName="LOV_TypedCouponCode"
. . .
    <Properties>
        <SchemaBasedProperties>
            <CONTROLTYPE Value="input_text_lov"/>
        </SchemaBasedProperties>
    </Properties>
</ViewAttribute>
. . .
<ListBinding
    Name="LOV_TypedCouponCode"
    ListVOName="Coupons"
    ListRangeSize="-1"
    NullValueFlag="none"
    NullValueId="LOV_TypedCouponCode_NullValueId"
    MRUCount="0">
    <AttrArray Name="AttrNames">
        <Item Value="TypedCouponCode"/>
    </AttrArray>
    <AttrArray Name="DerivedAttrNames">
        <Item Value="CouponId"/>
    </AttrArray>
    <AttrArray Name="ListAttrNames">
        <Item Value="EasyCode"/>
        <Item Value="DiscountId"/>
    </AttrArray>
    <AttrArray Name="ListDisplayAttrNames">
        <Item Value="EasyCode"/>
    </AttrArray>
</ListBinding>

. . .
<ViewAccessor
    Name="Coupons"
    ViewObjectName="oracle.fodemo.storefront.store.queries.CouponsVO"/>
```

5.12.10 What Happens at Runtime: How an LOV Queries the List Data Source

The ADF Business Components runtime adds view accessors in the attribute setters of the view row and entity object to facilitate the LOV-enabled attribute behavior. In order to display the LOV-enabled attribute values in the user interface, the LOV facility fetches the data source, and finds the relevant row attributes and mapped target attributes.

The number of data objects that the LOV facility fetches is determined in part by the `ListRangeSize` setting in the LOV-enabled attribute's list binding definition which

is specified in the Edit List of Values dialog that you display on the attribute from the view object overview editor. If the number of records fetched is very large, the default value for `ListRangeSize` may truncate the values available to the dropdown list component used to display the records. The default number of fetched records for LOV queries depends on the type of list component used to display the records. In the case of the Combo Box with List of Values component and the Input Text with List of Values component, the default value for `ListRangeSize` is `10`. In the case of all other types of list components that you can select (including choice list, combo box, list box, and radio button group), the default value for `ListRangeSize` is set to `-1`. The value `-1` means that the user will be able to view all the data objects from the data source. The `ListRangeSize` value has no effect on the records that the end user can search on in the lookup dialog displayed for the two List of Values type components. For more information about how each list component displays values, see Section 5.12.5, "How to Set User Interface Hints on a View Object LOV-Enabled Attribute."

Note that although you can alter the `ListRangeSize` value in the metadata definition for the `<ListBinding>` element, setting the value to a discrete number of records (for example, `ListRangeSize="5"`) most likely will not provide the user with the desired selection choices. Instead, if the value is `-1` (default for simple list components without a LOV dialog), then no restrictions are made to the number of records the list component will display, and the user will have access to the full set of values.

> **Performance Tip:** To limit the set of values a LOV displays, use a view accessor to filter the LOV binding, as described in Section 5.12.1, "How to Define a Single LOV-Enabled View Object Attribute." Additionally, in the case of component types that display a choice list, you can change the **Most Recently Used Count** setting to limit the list to display the user's previous selections, as described in Section 5.12.5, "How to Set User Interface Hints on a View Object LOV-Enabled Attribute."

Note, a runtime exception will occur when a web page displays a UI component for an LOV-enabled attribute that does not match the view object's `CONTROLTYPE` definition. When the user interface designer creates the page in JDeveloper using the Data Controls panel, JDeveloper automatically inserts the list component identified by the **Default List Type** selection you made for the view object's LOV-enabled attribute in the List UI Hint dialog. However, if the user interface designer changes the list type subsequent to creating the web page, you will need to edit the selection in the List UI Hint dialog to match.

5.12.11 What You May Need to Know About Lists

There are several things you may need to know about LOVs that you define for attributes of view objects, including how to propagate LOV-enabled attributes from parent view objects to child view objects (by extending an existing view object) and when to use validators instead of an LOV to manage a list of values.

5.12.11.1 Inheritance of AttributeDef Properties from Parent View Object Attributes

When a view object extends another view object, you can create the LOV-enabled attribute on the base object. Then when you define the child view object in the overview editor, the LOV definition will be visible on the corresponding view object attribute. This inheritance mechanism allows you to define an LOV-enabled attribute once and later apply it across multiple view objects instances for the same attribute.

You can also use the overview editor to extend the inherited LOV definition. For example, you may add extra attributes already defined by the base view object's query to display in selection list. Alternatively, you can define a view object that uses a custom `WHERE` clause to query the supplemental attributes not already queried by the based view object. For information about customizing entity-based view objects, see Section 5.10, "Working with Bind Variables."

5.12.11.2 Using Validators to Validate Attribute Values

If you have created an LOV-enabled attribute for a view object, there is no need to validate the attribute using a List Validator. You only use an attribute validator when you do not want the list to display in the user interface, but still need to restrict the list of valid values. List validation may be a simple static list or it may be a list of possible values obtained through a view accessor you define. Alternatively, you might prefer to use Key Exists validation when the attribute displayed in the UI is one that references a key value (such as a primary, foreign, or alternate key). For information about declarative validation in ADF Business Components, see Chapter 7, "Defining Validation and Business Rules Declaratively."

5.13 Defining UI Hints for View Objects

One of the built-in features of ADF Business Components is the ability to define UI hints on view objects and attributes of view objects. UI hints are settings that the view layer can use to automatically display the queried information to the user in a consistent, locale-sensitive way. For example, in web pages, a UI developer may access UI hint values by entering EL expressions utility methods defined on the `bindings` name space and specified for ADF binding instance names.

JDeveloper stores the hints in resource bundle files that you can easily localize for multilingual applications.

5.13.1 How to Add Attribute-Specific UI Hints

To create UI hints for attributes of a view object, use the overview editor for the view object, which is accessible from the Application Navigator. You can also display and edit UI hints using the Property Inspector that you display for an attribute.

Before you begin:

It may be helpful to have an understanding of attribute UI hints. For more information, see Section 5.13, "Defining UI Hints for View Objects."

You will need to complete this task:

Create the desired view objects as described in Section 5.2.1, "How to Create an Entity-Based View Object," and Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object."

To customize view object attribute with UI hints:

1. In the Application Navigator, double-click the view object.
2. In the overview editor, click the **Attributes** navigation tab.

   Alternatively, display the Property Inspector for the selected attribute and click the **UI Hints** navigation tab. The Property Inspector provides a way to customize the attribute's control hints without using the overview editor.
3. In the Attributes page, select the attribute that you want to customize with UI hints, and then click the **UI Hints** tab and define the desired hints.

For example, for an attribute `UserId`, you might enter a value for its **Label Text** hint like "`Id`" or set the **Format Type** to Number, and enter a **Format** mask of `00000`.

Note: Java defines a standard set of format masks for numbers and dates that are different from those used by the Oracle database's SQL and PL/SQL languages. For reference, see the Javadoc for the `java.text.DecimalFormat` and `java.text.SimpleDateFormat` classes.

5.13.2 How to Add View Object UI Hints

To create UI hints for attributes of a view object, use the overview editor for the view object, which is accessible from the Application Navigator. You can also display and edit several additional UI hints using the Property Inspector that you display for the view object.

Before you begin:

It may be helpful to have an understanding of UI hints. For more information, see Section 5.13, "Defining UI Hints for View Objects."

You will need to complete this task:

Create the desired view objects as described in Section 5.2.1, "How to Create an Entity-Based View Object," and Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object."

To customize view objects with UI hints:

1. In the Application Navigator, double-click the view object.
2. In the overview editor, click the **General** navigation tab.
3. In the General page, enter a **Display Name** to define an EL accessible hint for the view object name.

   For example, for a view object `OrdersVO`, you might enter a value for its **Display Name** hint like "`Order`".
4. With the General page displayed in the overview editor, open the Property Inspector for the view object and expand the **UI Hints** section, and then enter additional hints as needed.

   For example, for a view object `OrdersVO`, you might enter a value for the **Display Name (Plural)** hint like "`Orders`" and, for the **Description** hint, you might enter a value like "`customer orders`".

5.13.3 How to Access UI Hints Using EL Expressions

A UI developer can access UI hints using EL expressions and display the hint values as data in a web page. The UI developer may access UI hints through the ADF binding instances that they create after dropping databound components into their web pages.

In the case of the view object hints, the UI developer accesses the view object hints through the iterator binding defined for the view object. For example, assume that you have configured the view object UI hints as follows.

- `OrdersVO` view object **Display Name** hint = `Order`
- `OrdersVO` view object **Display Name (Plural)** hint = `Orders`

- `OrdersVO` view object **Description** hint = `customer orders`

The UI developer might display a header that makes use of these hints like this:

```
Showing customer orders number 10 of 51 Orders.
```

Example 5–38 shows that the EL expression that produces the above text. In this EL expression the iterator binding `OrdersVO1Iterator` provides access to the view object hints. The names of the EL expression utility methods match the property names defined in the view object XML definition file for the UI hints. For example, the view object property name `labelPlural`, which defines the **Display Name (Plural)** hint, corresponds to the utility method name used in the expression `bindings.OrdersVO1Iterator.hints.labelPlural`.

Example 5–38 EL to Access View Object UI HInts

```
<af:panelHeader id="ph1"
           text="Showing #{bindings.OrdersVO1Iterator.hints.description} number
                         #{bindings.Orderno.inputValue} of
                         #{bindings.OrdersVO1.estimatedRowCount}
                         #{bindings.OrdersVO1Iterator.hints.labelPlural}.">
```

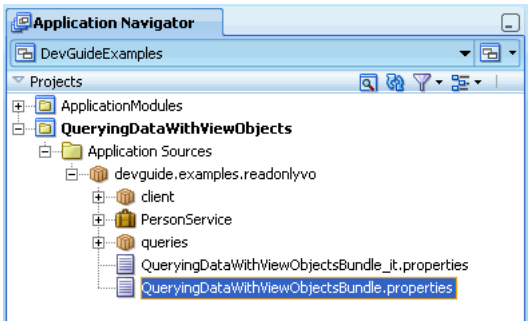
5.13.4 What Happens When You Add UI Hints

When you define attribute UI hints for a view object or view object attributes, by default JDeveloper creates a project-level resource bundle file in which to store them. For example, when you define UI hints for a view object in the `StoreFront` project, JDeveloper creates the message bundle file named `StoreFrontBundle.xxx` for the package. The hints that you define can be used by generated forms and tables in associated view clients.

The type of resource bundle file that JDeveloper uses and the granularity of the file are determined by settings on the Resource Bundle page of the Project Properties dialog. By default, JDeveloper sets the option to **Properties Bundle** and generates one `.properties` file for the entire data model project.

Alternatively, if you select the option in the Project Properties dialog to generate one resource bundle per file, you can inspect the message bundle file for any view object by selecting the object in the Application Navigator and looking under the corresponding **Sources** node in the Structure window. The Structure window shows the implementation files for the component you select in the Application Navigator. You can inspect the resource bundle file for the view object by expanding the parent package of the view object in the Application Navigator, as shown in Figure 5–46.

Figure 5–46 Resource Bundle File in Application Navigator



For more information on the resource bundle options you can select, see Section 4.7.1, "How to Set Message Bundle Options."

Example 5–39 shows a sample message bundle file where the UI hint information appears. The first entry in each `String` array is a message key; the second entry is the locale-specific `String` value corresponding to that key.

Example 5–39 Resource File with Locale-Sensitive UI Hints

```
devguide.examples.readonlyvo.queries.Persons.PersonId_FMT_FORMATTER=
                                    oracle.jbo.format.DefaultNumberFormatter
devguide.examples.readonlyvo.queries.Persons.PersonId_FMT_FORMAT=00000
devguide.examples.readonlyvo.queries.Persons.PersonId_LABEL=Id
devguide.examples.readonlyvo.queries.Persons.Email_LABEL=Email Address
devguide.examples.readonlyvo.queries.Persons.LastName_LABEL=Surname
devguide.examples.readonlyvo.queries.Persons.FirstName_LABEL=Given Name
```

5.13.5 How to Define UI Category Hints

UI categories provide the means to group attributes that a view object defines. The category names that you create are identifiers to be used by the dynamic rendering user interface to group attributes for display. The user interface will render the attribute with other attributes of the same category. You can use the category hint to aid the user interface to separate a large list of view object attributes into smaller groups related by categories.

Additionally, you can specify the field order hint to reorder how the user interface will render the attribute values within its category. For example, if a view object defines four attributes `attributeA`, `attributeB`, `attributeC`, and `attributeD` and you specify the field order 4, 3, 2, and 1 respectively for each attribute, then wherever the user interface renders the category, the attributes of that category will appear in the order attributeD, attributeC, attributeB, and attributeA.

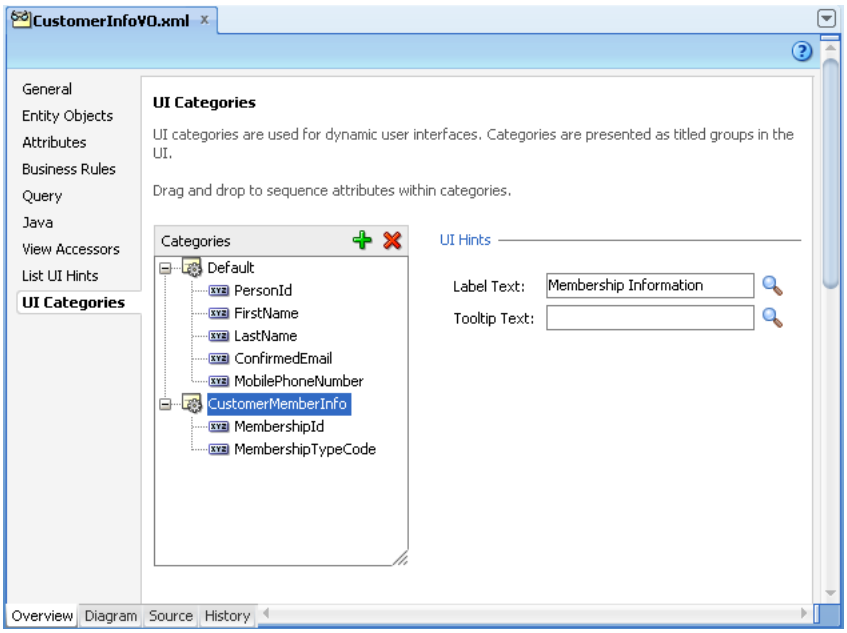
> **Note:** Use the UI Categories page in the overview editor for the view object to change the order of the attributes listed within a category you've created. JDeveloper automatically assigns and maintains the field order values of the attributes based on their order in the list, and you do not need to edit numeric values to define the field order hint.

The category and field order hints will be utilized by any dynamic rendering user interface that displays the attribute, including dynamic forms and search forms:

- In the case of dynamic forms, the attributes from each category will appear in a separate tab.
- In the case of search forms, the order of the form's individual view criteria is determined by the field order assigned to the attribute upon which the view criteria items are based.

To create UI categories for attributes of a view object, use the overview editor for the view object, which is accessible from the Application Navigator. You can create and edit categories for the entire view object using the UI Categories page, as shown in Figure 5–47.

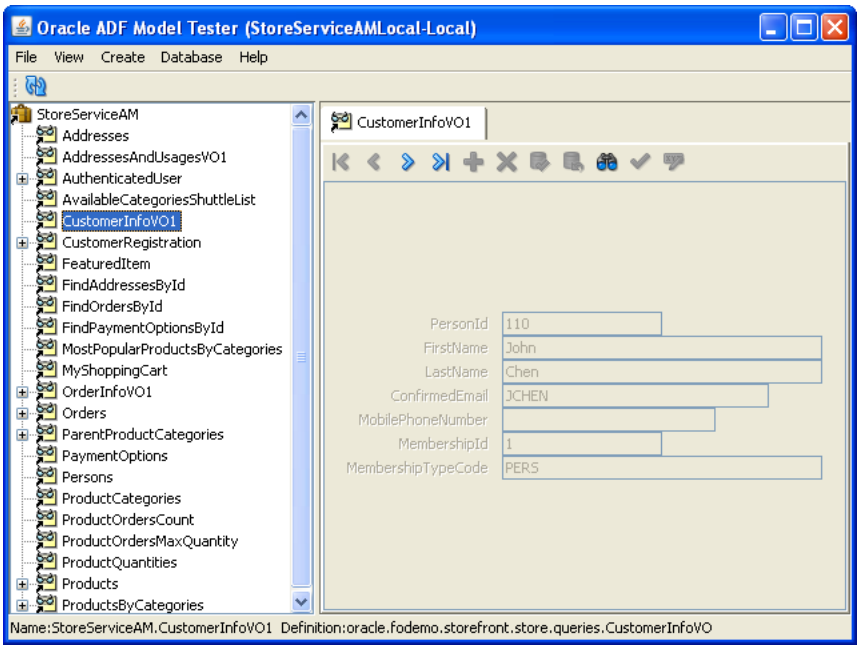
Figure 5–47 Attribute UI Categories in View Object Overview Editor



When you assign a view object attribute to a category that you create in the UI Categories page, the order of the attributes displayed in the category list determines its numeric field order. The UI Categories page lets you change the field order of the attributes you've assigned to a category by dragging and dropping attributes within the list. When you drag and drop the attributes into the list, JDeveloper automatically maintains the correct sequence of the field order hints within the category. In the editor for the view object, you can view the assigned category name and field order value of individual attributes in the **UI Hints** tab of the Attributes page. JDeveloper synchronizes the **UI Hints** tab definitions with the changes you make in the UI Categories page.

Each category can have a label and tooltip text string resource to be utilized by the user interface when the category is rendered. You can localize these resource strings in the resource bundle file that you select to store the entries. For example, Figure 5–48 shows the attributes `MembershipId` and `MembershipTypeCode` with labels **Membership ID** and **Membership Type Code** in a category with the label **Membership Information**.

Figure 5–48 UI Categories Displayed in Oracle ADF Model Tester



Before you begin:

It may be helpful to have an understanding of attribute UI hints. For more information, see Section 5.13, "Defining UI Hints for View Objects."

You will need to complete this task:

Create the desired view objects, as described in Section 5.2.1, "How to Create an Entity-Based View Object," and Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object."

To create user interface categories and reorder the attributes for display:

1. In the Application Navigator, double-click the view object.
2. In the overview editor, click the **UI Categories** navigation tab.
3. In the UI Categories page, click the **Create New Category** button to create the category and then right-click the new category and choose **Rename**.
4. In the Rename dialog, enter the name of the category.
5. In the overview editor, in the **UI Hints** section, enter the user interface **Label Text** and **Tooltip Text** for the category.

   The text that you enter will be added to the default resource bundle file for the project, as described in Section 5.13.4, "What Happens When You Add UI Hints." To select a different resource bundle file to store the label and tooltip strings, click the browse (. . .) button beside the text field.
6. In the **Categories** list, expand the **Default** category and scroll the list of attributes to locate the attribute you want to add to a category.

   The **Default** category displays all the attributes that the view object you are editing defines. This category is displayed only by the overview editor as a selection list for the creation of UI Category hints and is not used as a category to render attributes in the user interface.

7. Select the desired attribute and drag it into the new category you created. If the category contains more than one attribute, the position of the attribute that you drop into the list will determine its field order value.

   JDeveloper automatically assigns a numeric value to the field order hint based on the sequence of the attributes that appear within a UI Category list. The attribute you place at the top of a category list will be rendered by the user interface first, the attribute second in the list will be rendered second, and so on.

8. To view the field order hint for the attribute, in the overview editor, click the **Attributes** navigation tab, select the attribute in the page, and then click the **UI Hints** tab.

   Tip: Do not edit the field order hint in the **UI Hints** tab; instead, use the UI Categories page of the view object editor to reorder the attributes in the list by dragging and dropping them. You can also right-click an attribute within a category list and choose **Move Up**, **Move Down**, or **Move to Category** to change the field order within the categories.

   The field order hint does not appear in the overview editor UI Categories page. The hint value is visible in the source for the view object definition or in the **UI Hints** tab that you display in the view object editor for a specific attribute.

5.13.6 What Happens When You Assign Attributes to UI Categories

When you define attribute UI categories for a view object, JDeveloper updates the view object's XML component definition file. JDeveloper adds the `CATEGORY` and the `FIELDORDER` UI hints in the `<SchemaBasedProperties>` element of the `<ViewAttribute>` element. The definition of the categories appears in a new `<Category>` element.

The metadata in Example 5–40 shows that the `PersonId` attribute's `CATEGORY` hint refers to the `AccountInformation` category and the `FirstName` attribute's `CATEGORY` hint refers to the `UserInformation` category. The definition for both categories appears in `<Category>` elements. The `FIELDORDER` hint for each attribute specifies a numeric value, which JDeveloper assigns and maintains based on the order of the attributes in the UI categories lists you create in the overview editor. As shown in Example 5–40, the `FIELDORDER` hint is a decimal value. A decimal value is used by JDeveloper to allow you to insert new attributes into a category without requiring JDeveloper to change all the existing attribute values and still be able to maintain the correct order.

Note that the default field order value for the first attribute in a category is assigned by JDeveloper as `0.0`. The field order value can be changed to any number to sort the category list, and it is not an index. The field order numeric values do not need to be contiguous.

Example 5–40 View Object MetaData for Attribute UI Category Hints

```
<Category
    Name="AccountInformation">
    <Properties>
      <SchemaBasedProperties>
        <LABEL
          ResId="CUSTOMER_DETAILS"/>
        <TOOLTIP
          ResId="AccountInformation_TOOLTIP"/>
```

```
      </SchemaBasedProperties>
    </Properties>
</Category>
<Category
    Name="UserInformation">
    <Properties>
      <SchemaBasedProperties>
        <TOOLTIP
          ResId="UserInformation_TOOLTIP"/>
        <LABEL
          ResId="CUSTOMER_DETAILS"/>
      </SchemaBasedProperties>
    </Properties>
</Category>
...
<ViewAttribute
    Name="PersonId"
    IsNotNull="true"
    PrecisionRule="true"
    EntityAttrName="PersonId"
    EntityUsage="Person"
    AliasName="PERSON_ID">
    <Data>
      <Property
        Name="OWNER_SCOPE"
        Value="INSTANCE"/>
      ...
    </Data>
    <Properties>
      <SchemaBasedProperties>
        <CATEGORY
          Value="AccountInformation"/>
        <FIELDORDER
          Value="0.0"/>
      </SchemaBasedProperties>
    </Properties>
  </ViewAttribute>
...
<ViewAttribute
    Name="FirstName"
    PrecisionRule="true"
    EntityAttrName="FirstName"
    EntityUsage="Person"
    AliasName="FIRST_NAME">
    <Data>
      <Property
        Name="OWNER_SCOPE"
        Value="INSTANCE"/>
      ...
    <Properties>
      <SchemaBasedProperties>
        <CATEGORY
          Value="UserInformation"/>
        <FIELDORDER
          Value="0.0"/>
      </SchemaBasedProperties>
    </Properties>
</ViewAttribute>
```

5.13.7 What You May Need to Know About Resource Bundles

Internationalizing the model layer of an application built using ADF Business Components entails producing translated versions of each component's resource bundle file. For example, the Italian version of the `QueryDataWithViewObjectsBundle.properties` file would be a file named `QueryDataWithViewObjectsBundle_it.properties`, and a more specific *Swiss* Italian version would have the name `QueryDataWithViewObjectsBundle_it_ch.properties`.

Resource bundle files contain entries for the message keys that need to be localized, together with their localized translation. For example, assuming you didn't want to translate the number format mask for the Italian locale, the Italian version of the `QueryDataWithViewoObjects` view object message keys would look like what you see in Example 5–41. At runtime, the resource bundles are used automatically, based on the current user's locale settings.

Example 5–41 Localized View Object Component Resource Bundle for Italian

```
devguide.examples.readonlyvo.queries.Persons.PersonId_FMT_FORMATTER=
                                        oracle.jbo.format.DefaultNumberFormatter
devguide.examples.readonlyvo.queries.Persons.PersonId_FMT_FORMAT=00000
devguide.examples.readonlyvo.queries.Persons.PersonId_LABEL=Codice Utente
devguide.examples.readonlyvo.queries.Persons.Email_LABEL=Indirizzo Email
devguide.examples.readonlyvo.queries.Persons.LastName_LABEL=Cognome
devguide.examples.readonlyvo.queries.Persons.FirstName_LABEL=Nome
```

5.14 Adding Calculated and Transient Attributes to a View Object

In addition to having attributes that map to underlying entity objects, your view objects can include calculated attributes that don't map to any entity object attribute value. The two kinds of calculated attributes are known as:

- *SQL-calculated attributes*, when their value is retrieved as an expression in the SQL query's SELECT list
- *Transient attributes*, when their value is not retrieved as part of the query

A view object can include an entity-mapped attribute which itself is a transient attribute at the entity object level.

5.14.1 How to Add a SQL-Calculated Attribute

You use the overview editor for the view object to add a SQL-calculated attribute.

Before you begin:

It may be helpful to have an understanding of calculated attributes. For more information, see Section 5.14, "Adding Calculated and Transient Attributes to a View Object."

You will need to complete this task:

Create the desired view objects as described in Section 5.2.1, "How to Create an Entity-Based View Object," and Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object."

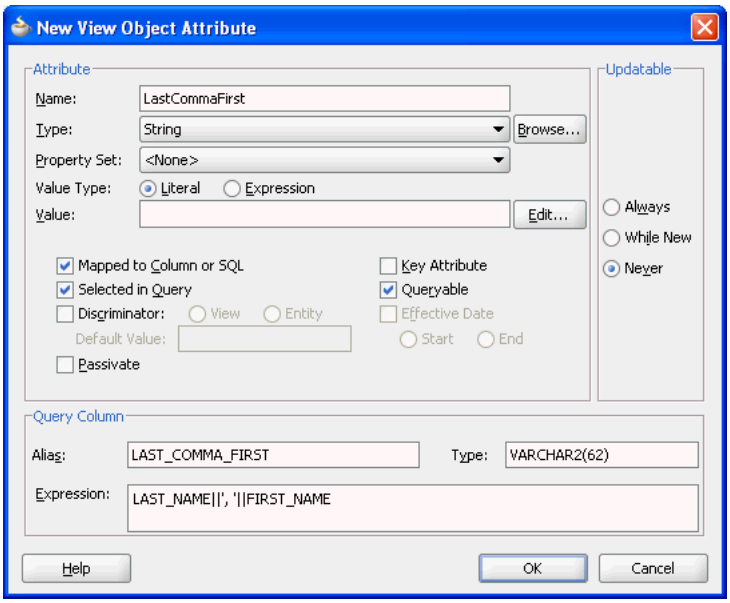
To add a SQL-calculated attribute to a view object:

1. In the Application Navigator, double-click the view object for which you want to define a SQL-calculated attribute.

2. In the overview editor, click the **Attributes** navigation tab and click the **Create new attribute** button.
3. In the New View Object Attribute dialog, enter a name for the attribute.
4. Set the Java attribute type to an appropriate value.
5. Select the **Mapped to Column or SQL** checkbox.
6. Provide a SQL expression in the **Expression** field.

   For example, to change the order of first name and last name, you could write the expression `LAST_NAME||', '||FIRST_NAME`, as shown in Figure 5–49.

Figure 5–49 New SQL-Calculated Attribute



7. Consider changing the SQL column alias to match the name of the attribute.
8. Verify the database query column type and adjust the length (or precision/scale) as appropriate.
9. Click **OK**.

5.14.2 What Happens When You Add a SQL-Calculated Attribute

When you add a SQL-calculated attribute in the overview editor for the view object, JDeveloper updates the XML component definition for the view object to reflect the new attribute. The entity-mapped attribute's `<ViewAttribute>` tag looks like the sample shown in Example 5–42. The entity-mapped attribute inherits most of it properties from the underlying entity attribute to which it is mapped.

Example 5–42 Metadata For Entity-Mapped Attribute

```
<ViewAttribute
   Name="LastName"
   IsNotNull="true"
   EntityAttrName="LastName"
   EntityUsage="User1"
   AliasName="LAST_NAME" >
</ViewAttribute>
```

Whereas, in contrast, a SQL-calculated attribute's `<ViewAttribute>` tag looks like sample shown in Example 5–43. As expected, the tag has no `EntityUsage` or `EntityAttrName` property, and includes datatype information along with the SQL expression.

Example 5–43 Metadata For SQL-Calculated Attribute

```
<ViewAttribute
   Name="LastCommaFirst"
   IsUpdatable="false"
   IsPersistent="false"
   Precision="62"
   Type="java.lang.String"
   ColumnType="VARCHAR2"
   AliasName="FULL_NAME"
   Expression="LAST_NAME||&#39;, &#39;||FIRST_NAME"
   SQLType="VARCHAR" >
</ViewAttribute>
```

Note: The `'` is the XML character reference for the apostrophe. You reference it by its numerical ASCII code of 39 (decimal). Other characters in literal text that require similar construction in XML are the less-than, greater-than, and ampersand characters.

5.14.3 How to Add a Transient Attribute

Transient attributes are often used to provide subtotals or other calculated expressions that are not stored in the database.

Before you begin:

It may be helpful to have an understanding of transient attributes. For more information, see Section 5.14, "Adding Calculated and Transient Attributes to a View Object."

You will need to complete this task:

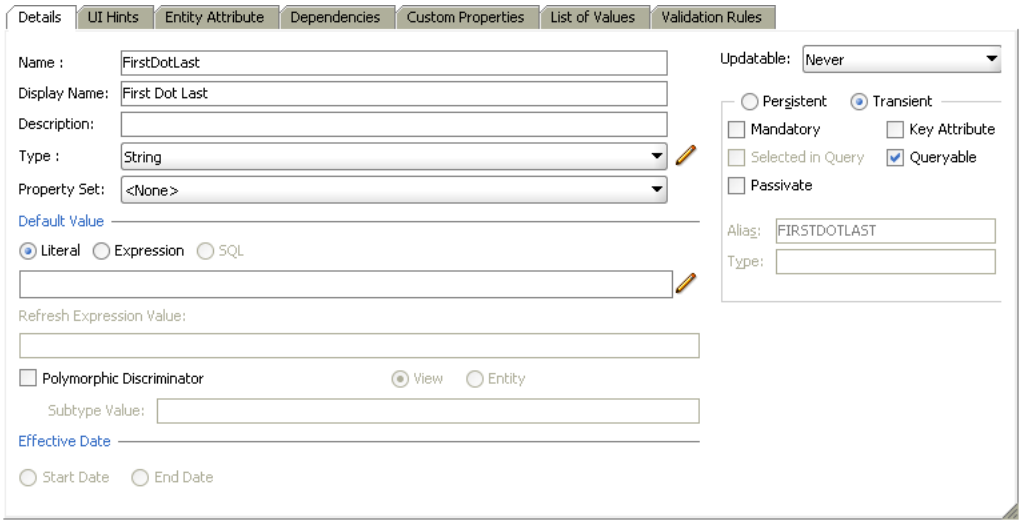
Create the desired view objects as described in Section 5.2.1, "How to Create an Entity-Based View Object," and Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object."

To add a transient attribute to a view object:

1. In the Application Navigator, double-click the view object for which you want to define a transient attribute.
2. In the overview editor, click the **Attributes** navigation tab and click the **Create new attribute** button.
3. In the New View Object Attribute dialog, enter a name for the attribute and click **OK**.
4. In the Attributes page of the overview editor, select the new attribute, and then click the **Details** tab and verify that **Transient** appears selected.
5. In the Details section, change **Type** to an appropriate Java type.

   For example, a calculated attribute that concatenates a first name and a last name would have the type `String`, as shown in Figure 5–50.

Figure 5–50 New Transient Attribute



To create a transient attribute based on an expression:

1. In the Application Navigator, double-click the view object for which you want to define a transient attribute.
2. In the overview editor, click the **Attributes** navigation tab and click the **Create new attribute** icon.
3. In the New View Object Attribute dialog, enter a name for the attribute and click **OK**.
4. In the Attributes page of the overview editor, select the new attribute, and then click the **Details** tab and verify that **Transient** appears selected.
5. In the Details section, change **Type** to an appropriate Java type.
6. For the **Default Value**, select **Expression** and click **Edit value** to define an expression that calculates the value of the attribute.

   Expressions you define will be evaluated using the Groovy Expression Language. Groovy lets you insert expressions and variables into strings. The expression will be saved as part of the view object definition. For more information about Groovy, see Section 3.6, "Overview of Groovy Scripting Language Support."
7. In the Edit Expression dialog, enter an expression in the field provided.

   Attributes that you reference can include any attribute that the base entity objects define. Do not reference attributes in the expression that are not defined by the view object's underlying entity objects.
8. Select the appropriate recalculate setting.

   If you select **Always** (default), the expression is evaluated each time any attribute in the row changes. If you select **Never**, the expression is evaluated only when the row is created.
9. You can optionally provide a condition for when to recalculate the expression.

   For example, the following expression in the **Based on the following expression** field causes the attribute to be recalculated when either the Quantity attribute or the `UnitPrice` attribute are changed:

```
return (adf.object.isAttributeChanged("Quantity") ||
adf.object.isAttributeChanged("UnitPrice"));
```

10. When either the value expression or the optional recalculate expression that you define references an attribute from the base entity object, you must define this as a dependency in the Edit Expression dialog. Locate each attribute in the **Available** list and shuttle it to the **Selected** list.
11. Click **OK**.

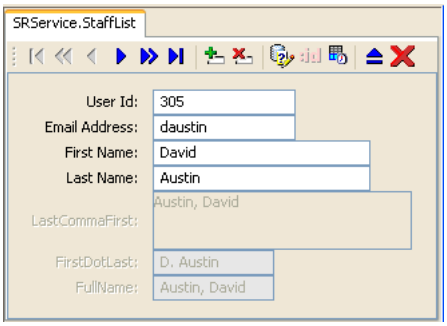
A view object can include an entity-mapped attribute which itself is a transient attribute at the entity object level.

To add a transient attribute from an entity object to an entity-based view object:

1. In the Application Navigator, double-click the view object for which you want to add a transient attribute based on an entity usage.
2. In the overview editor, click the **Attributes** navigation tab and click the dropdown menu on the **Create new attribute** icon and choose **Add Attribute from Entity** to view the list of available entity-derived attributes.
3. In the Attributes dialog, move the desired transient attribute from the **Available** list into the **Selected** list.
4. Click **OK**.

If you use the Oracle ADF Model Tester to test the data model, you can see the usage of your transient attributes. Figure 5–51 shows three attributes that were created using a SQL-calculated attribute (`LastCommaFirst`), a transient attribute (`FirstDotLast`) and an entity-derived transient attribute (`FullName`).

Figure 5–51 View Object with Three Kinds of Calculated Attributes



5.14.4 How to Add a Validation Rule to a Transient Attribute

Attribute-level validation rules are triggered for a particular view object transient attribute when either the end user or the program code attempts to modify the attribute's value. Since you cannot determine the order in which attributes will be set, attribute-level validation rules should be used only when the success or failure of the rule depends exclusively on the candidate value of that single attribute.

The process for adding a validation rule to an view object transient attribute is similar to the way you create any declarative validation rule, and is done using the Add Validation Rule dialog. You can open this dialog from the overview editor for the view object by clicking the **Add Validation Rule** icon in the Validation Rules section of the Attributes page. You must first select the transient attribute from the attributes list and the transient attribute must be defined as updatable. Validation rules cannot be defined for read-only transient attributes.

Before you begin:

It may be helpful to have an understanding of attribute UI hints. For more information, see Section 5.13, "Defining UI Hints for View Objects."

You may also find it helpful to understand the different types of validation rules you can define. For more information, see Section 7.4, "Using the Built-in Declarative Validation Rules."

You will need to complete this task:

Create the desired view objects as described in Section 5.2.1, "How to Create an Entity-Based View Object," and Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object."

To add a validation rule for a transient attribute:

1. In the Application Navigator, double-click the desired view object.
2. Click the **Attributes** navigation tab on the overview editor.
3. In the Attributes page of the overview editor, select the transient attribute, and then click the **Details** tab and verify that **Updatable** displays either **While New** or **Always**.

   Validation rules for transient attributes must be updatable. The value **Never** specifies a read-only attribute.
4. In the Attributes page, click the **Validation Rules** tab and then click the **Add Validation Rule** icon.
5. In the Add Validation Rule dialog, select the type of validation rule desired from the **Rule Type** dropdown list.
6. Use the dialog settings to configure the new rule.

   The controls will change depending on the kind of validation rule you select. For more information about the different validation rules, see Section 7.4, "Using the Built-in Declarative Validation Rules."
7. Optionally, click the **Validation Execution** tab and enter criteria for the execution of the rule, such as dependent attributes and a precondition expression. For more information, see Section 7.6, "Triggering Validation Execution."
8. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see Section 7.7, "Creating Validation Error Messages."
9. Click **OK**.

5.14.5 What Happens When You Add a Transient Attribute

When you add a transient attribute in the overview editor for a view object, JDeveloper updates the XML component definition for the view object to reflect the new attribute. A transient attribute's `<ViewAttribute>` tag in the XML is similar to the SQL-calculated one, but it lacks an `Expression` property.

When you base a transient attribute on a Groovy expression, a `<TransientExpression>` tag is created within the appropriate attribute, as shown in Example 5–44.

Example 5–44 Calculating a Transient Attribute Using a Groovy Expression

```
<TransientExpression>
```

```
   <![CDATA[
      ((Quantity == null) ? 0 : Quantity) * ((UnitPrice == null) ? 0 : UnitPrice)
   ]]>
</TransientExpression>
```

5.14.6 Adding Java Code in the View Row Class to Perform Calculation

A transient attribute is a placeholder for a data value. If you change the **Updatable** property of the transient attribute to **While New** or **Always**, the end user can enter a value for the attribute. If you want the transient attribute to display a calculated value, then you'll typically leave the **Updatable** property set to **Never** and write custom Java code that calculates the value.

After adding a transient attribute to the view object, to make it a *calculated* transient attribute you need to enable a custom view row class and choose to generate accessor methods, in the Java dialog that you open clicking the **Edit** icon on the Java page of the overview editor for the view object. Then you would write Java code inside the accessor method for the transient attribute to return the calculated value. Example 5–44 shows the `StaffListRowImpl.java` view row class contains the Java code to return a calculated value in the `getLastCommaFirst()` method.

```
// In StaffListRowImpl.java
public String getFirstDotLast() {
  // Commented out this original line since we're not storing the value
  // return (String) getAttributeInternal(FIRSTDOTLAST);
  return getFirstName().substring(0,1)+". "+getLastName();
}
```

5.14.7 What You May Need to Know About Transient Attributes

The view object includes the SQL expression for your SQL-calculated attribute in the `SELECT` list of its query at runtime. The database is the one that evaluates the expression, and it returns the result as the value of that column in the query. The value is reevaluated each time you execute the query.

6 Testing View Instance Queries

This chapter describes how to interactively test view instance query results using the Oracle ADF Model Tester provided in JDeveloper. This chapter also explains how to use the Business Components API to access view object instances in a test client outside of JDeveloper.

This chapter includes the following sections:

- Section 6.1, "About View Instance Queries"
- Section 6.2, "Creating an Application Module to Test View Instances"
- Section 6.3, "Testing View Object Instances Using the Oracle ADF Model Tester"
- Section 6.4, "Testing View Object Instances Programmatically"

6.1 About View Instance Queries

JDeveloper includes an interactive application module testing tool that you can use to test all aspects of its data model without having to use your application user interface or write a test client program. Running the Oracle ADF Model Tester can often be the quickest way of exercising the data functionality of your business service during development.

> **Note:** When you want to test an application module programmatically, you can write a test client. For more information, see Section 6.4.2, "How to Create a Command-Line Java Test Client."

6.1.1 View Instance Use Cases and Examples

Using the Oracle ADF Model Tester, you can simulate an end user interacting with your application module data model before you have started to build any custom user interface of your own. Even *after* you have your UI pages constructed, you will come to appreciate using the Oracle ADF Model Tester to assist in diagnosing problems when they arise. You can reproduce the issues in the Oracle ADF Model Tester to discover if the issue lies in the view or controller layers of the application, or is instead a problem in the business service layer application module itself.

6.1.2 Additional Functionality for View Instances

You may find it helpful to understand other ADF features before you start working with view instances. Following are links to other functionality that may be of interest.

- For details about creating a data model consisting of view object instances, see Chapter 9, "Implementing Business Services with Application Modules."

- For a quick reference to the most common code that you will typically write, use, and override in your custom ADF Business Components classes, see Appendix D, "Most Commonly Used ADF Business Components Methods."
- For API documentation related to the `oracle.jbo` package, see the following Javadoc reference document:
  - *Oracle Fusion Middleware Java API Reference for Oracle ADF Model*

6.2 Creating an Application Module to Test View Instances

Before you can test view objects that you create in your data model project, you must create an application module where you will define instances of the view objects you want to test. The application module is the transactional component that the Oracle ADF Model Tester (or UI client) will use to work with application data. The set of view objects used by an application module defines its *data model*, in other words, the set of data that a client can display and manipulate through a user interface.

To test the view objects you added to an application module, use the Oracle ADF Model Tester, which is accessible from the Application Navigator. For details about using the Oracle ADF Model Tester, see Section 6.3, "Testing View Object Instances Using the Oracle ADF Model Tester."

6.2.1 How to Create the Application Module with Individual View Object Instances

To create an application module that will define instances of individual view objects, use the Create Application Module wizard, which is available in the New Gallery.

Before you begin:

It may be helpful to have an understanding of application modules. For more information, see Section 6.2, "Creating an Application Module to Test View Instances."

You will need to complete this task:

Create the desired view objects, as described in Section 5.2.1, "How to Create an Entity-Based View Object" and Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object."

To create an application module to test individual view object instances:

1. In the Application Navigator, right-click the project in which you want to create the application module and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Application Module**, and click **OK**.
3. In the Items list, select **Application Module** to launch the Create Application Module wizard.
4. In the Create Application Module wizard, in the Name page, provide a package name and an application module name. Click **Next**.
5. On the Data Model page, include instances of the view objects you have previously defined and edit the view object instance names to be exactly what you want clients to see. Then click **Finish**.

   Instead of accepting the default instance name shown in the Data Model page, you can change the instance name to something more meaningful (for example, instead of the default name `OrderItems1` you can rename it to `AllOrderItems`).

6.2.2 How to Create the Application Module with Master-Detail View Object Instances

You can also use the Create Application Module wizard to create a hierarchy of view objects for an application module, based on master-detail relationships that the view objects represent.

Before you begin:

It may be helpful to have an understanding of application modules. For more information, see Section 6.2, "Creating an Application Module to Test View Instances."

You will need to complete this task:

Create hierarchical relationships between view objects, as described in Section 5.6, "Working with Multiple Tables in a Master-Detail Hierarchy."

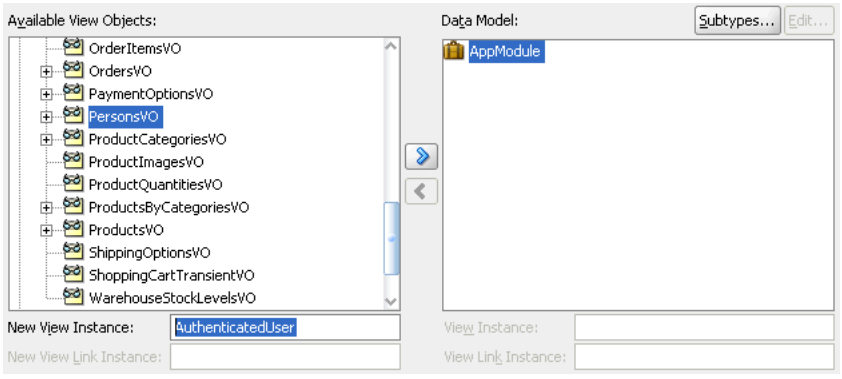
To create an application module based on view object relationships:

1. In the Application Navigator, right-click the project in which you want to create the application module and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Application Module**, and click **OK**.
3. In the Items list, select **Application Module** to launch the Create Application Module wizard.
4. In the Create Application Module wizard, select the **Data Model** node.
5. In the **Available View Objects** list on the left, select the instance of the view object that you want to be the actively coordinating master.

   The master view object will appear with a plus sign in the list indicating the available view links for this view object. The view link must exist to define a master-detail hierarchy.

   For example, Figure 6–1 shows `PersonsVO` selected and renamed `AuthenticatedUser` in the **New View Instance** field.

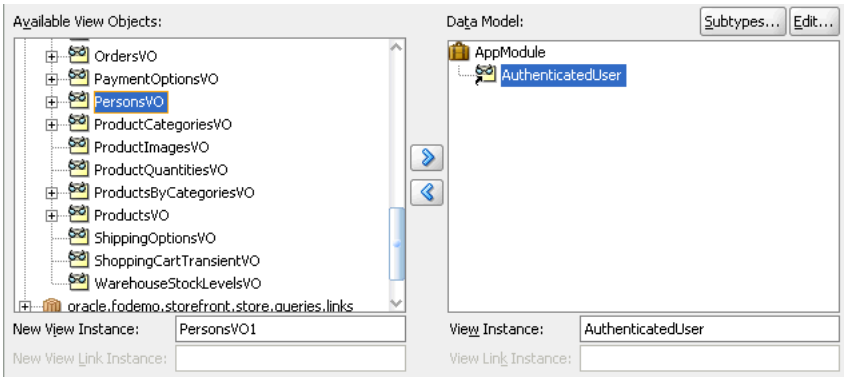
Figure 6–1 Master View Object Selected



6. Shuttle the selected master view object to the **Data Model** list

   For example, Figure 6–2 shows the newly created master view instance `AuthenticatedUser` in the **Data Model** list after you add it to the list.

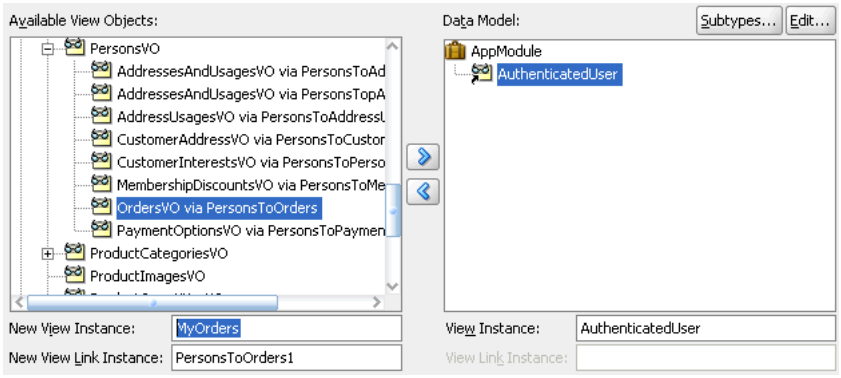
Figure 6–2 Master View Instance Created



7. In the **Data Model** list, leave the newly created master view instance selected, so that it appears highlighted. This will be the target of the detail view instance you will add. Then locate and select the detail view object beneath the master view object in the **Available View Objects** list.

   For example, Figure 6–3 shows the detail `OrdersVO` indented beneath master `PersonsVO` with the name `OrdersVO via PersonsToOrders`. The name identifies the view link `PersonsToOrders`, which defines the master-detail hierarchy between `PersonsVO` and `OrdersVO`. The detail view instance is renamed to `MyOrders`.

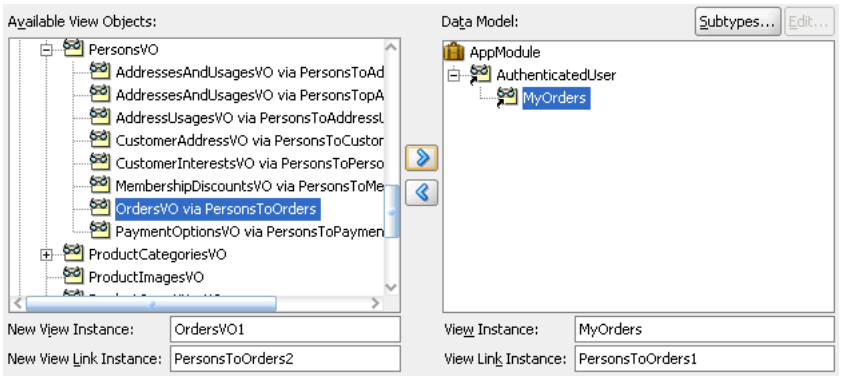
Figure 6–3 Detail View Object Selected



8. To add the detail instance to the previously added master instance, shuttle the detail view object to the **Data Model** list below the selected master view instance.

   Figure 6–4 shows the newly created detail view instance `MyOrders` is a detail of the `AuthenticatedUser` in the data model.

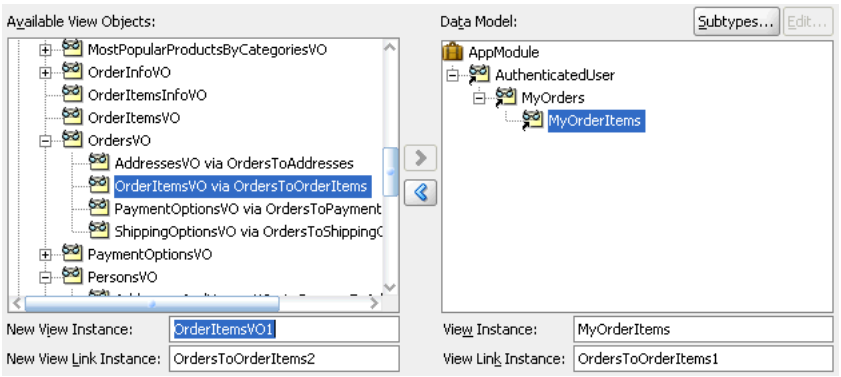
Figure 6–4 Master View Instance Created



9. To add another level of hierarchy, select the newly added detail in the **Data Model** list, then shuttle over the new detail which itself has a master-detail relationship with the previously added detail instance.

Your data model can contain as many levels of hierarchy as your view object relationships support. For example, Figure 6–5 shows the **Data Model** list with instance `AuthenticatedUser` (renamed for `PersonsVO`) as the master of `MyOrders` (renamed for `OrdersVO via PersonsToOrders`), which in turn is a master for `MyOrderItems` (renamed from `OrderItemsVO via OrdersToOrderItems`). The detail view object `MyOrderItems` is the last level of the hierarchy possible because this view object is itself not a master for another view object.

Figure 6–5 Master-Detail-Detail Hierarchy Created



6.3 Testing View Object Instances Using the Oracle ADF Model Tester

Using the Oracle ADF Model Tester, you can simulate an end user interacting with your application module data model before you have started to build any custom user interface of your own. Even *after* you have your UI pages constructed, you will come to appreciate using the Oracle ADF Model Tester to assist in diagnosing problems when they arise. You can reproduce the issues in the Oracle ADF Model Tester to discover whether the problem lies in the view or controller layers of the application, or whether there is instead a problem in the business service layer application module itself.

6.3.1 How to Run the Oracle ADF Model Tester

To test the view objects you added to an application module, use the Oracle ADF Model Tester, which is accessible from the Application Navigator.

Before you begin:

It may be helpful to have an understanding of the Oracle ADF Model Tester. For more information, see Section 6.3, "Testing View Object Instances Using the Oracle ADF Model Tester."

To test view objects in an application module configuration:

1. In the Application Navigator, expand the project containing the desired application module and view objects.
2. Right-click the application module and choose **Run**.

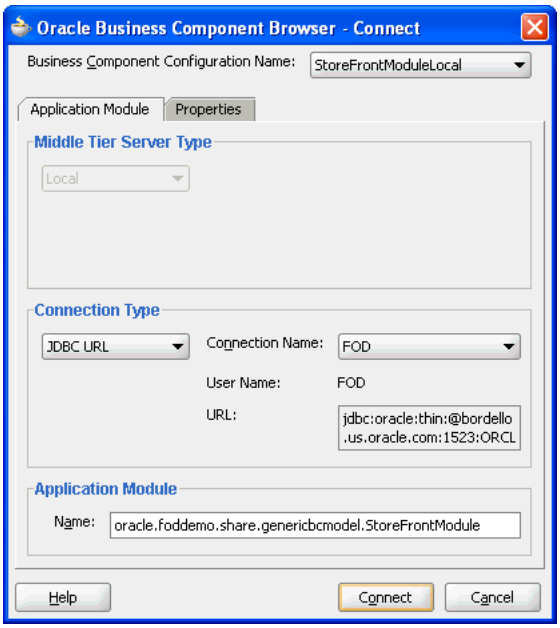
   Alternatively, choose **Debug** when you want to run the application in the Oracle ADF Model Tester with debugging enabled. JDeveloper opens the debugger process panel in the Log window and the various debugger windows. For example, when debugging using the Oracle ADF Model Tester, you can view status message and exceptions, step in and out of source code, and manage breakpoints.

   For information about receiving diagnostic messages specific to ADF Business Component debugging, see Section 6.3.8, "How to Enable ADF Business Components Debug Diagnostics".
3. In the Select Business Components Configuration dialog, choose the desired application module configuration from the **Business Component Configuration Name** list to run the Oracle ADF Model Tester.

   By default, an application module has only its default configurations, named *AppModuleName*`Local` and *AppModuleName*`Shared`. For example, Figure 6–6 shows the `StoreFrontModuleLocal` configuration used by the application module to connect to the database.

   If you have created additional configurations for your application module and want to test it using one of those instead, just select the desired configuration from the **Business Component Configuration Name** dropdown list on the Configuration dialog before clicking **Connect**.

Figure 6–6 Configuration Selection in Configuration Dialog

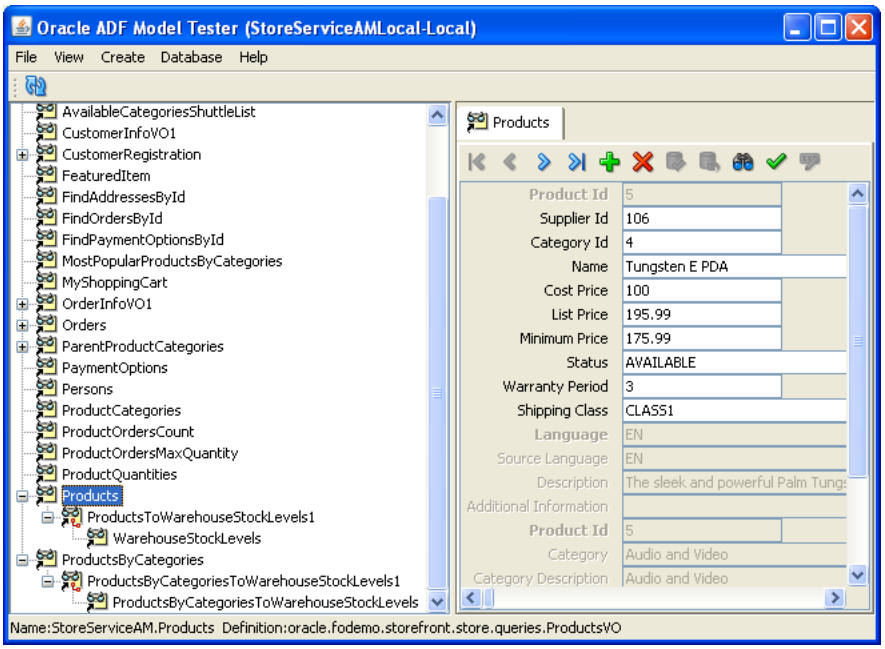


4. Click **Connect** to start the application module using the selected configuration.

5. To execute a view object in the Oracle ADF Model Tester, expand the data model tree and double-click the desired view object node.

   Note that the view object instance may already appear executed in the testing session. In this case, the Oracle ADF Model Tester data view page on the right already displays query results for the view object instance. The fields in the Oracle ADF Model Tester data view page of a read-only view object will always appear disabled since the data it represents is not editable. For example, in Figure 6–7, data for the view instance `Products` appears in the tester. Fields like **Product Id**, **Language**, and **Category** appear disabled because the attributes themselves are not editable.

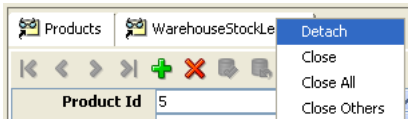
Figure 6–7 Testing the Data Model in the Oracle ADF Model Tester



6. Right-click a node in the data model tree at the left of the Oracle ADF Model Tester to display the context menu for that node. For example, on a view object node you can reexecute the query if needed, to remove the view object from the data model tree, and perform other tasks.

7. Right-click the tab of an open data viewer to display the context menu for that tab, as shown in Figure 6–8. For example, you can close the data viewer or open it in a separate window.

Figure 6–8 Context Menu for Data Viewer Tabs in the Oracle ADF Model Tester



6.3.2 How to Test Entity-Based View Objects Interactively

You test entity-based view objects interactively in the same way as read-only ones. Just add instances of the desired view objects to the data model of some application module, and then test that application module using the Oracle ADF Model Tester.

You'll find the Oracle ADF Model Tester invaluable for quickly testing and debugging your application modules. Table 6–1 gives an overview of the operations that the Oracle ADF Model Tester toolbar buttons perform when you display an entity-based view object.

Table 6–1 Oracle ADF Model Tester Toolbar Buttons

Button	Operation	Usage
	Move to ... row	Changes the current row displayed by the Oracle ADF Model Tester. Moves to the first, previous, next, or last row.

Table 6–1 (Cont.) Oracle ADF Model Tester Toolbar Buttons

Button	Operation	Usage
	Insert a new row	Creates and inserts a new row.
	Delete the current row	Deletes the current row.
	Save changes to the database	Posts and commits changes that you made in the ADF Business Components cache.
	Discard all changes since last save	Discards changes that you made in the ADF Business Components cache and restores the original values, rolling back any changes posted to the database.
	Specify view criteria	Displays the View Criteria dialog that you can use to create and apply view criteria to the master view object instance.
	Validate row	Validates the current row by applying validation rules defined for all entity object instances. Disabled unless at least one field is editable.
	Edit bind variables	Displays the Bind Variable dialog that you can use to enter values for bind parameters used in the view object query. Disabled unless the view object query uses bind parameters in the query statement.

To test the entity-based view objects you added to an application module, use the Oracle ADF Model Tester, which is accessible from the Application Navigator.

Before you begin:

It may be helpful to have an understanding of the Oracle ADF Model Tester. For more information, see Section 6.3, "Testing View Object Instances Using the Oracle ADF Model Tester."

To test entity-based view objects using an application module configuration:

1. Select the application module in the Application Navigator and choose **Run** from the context menu.
2. Click **Connect** on the Select Oracle ADF Model Tester Configuration dialog and use the desired configuration for testing.
3. To execute an entity-based view object in the Oracle ADF Model Tester, expand the data model tree and double-click the desired view object node.

   Unlike the fields of a read-only view object, the fields displayed in the data view page will appear enabled, because the data it represents is editable.
4. You can use the editable fields to update individual values and perform validation checks on the entered data.

   In the case of a view instance with referenced entities, you can change the foreign key value and observe that the referenced part changes.
5. You can use the toolbar buttons to perform row-level operations, such as navigate rows, create row, remove row, and validate the current row.

For further discussion about simulating end-user interaction in the data view page, see Section 6.3.5, "How to Simulate End-User Interaction in the Oracle ADF Model Tester".

6.3.3 How to Update the Oracle ADF Model Tester to Display Project Changes

Normally, changes that you make to the data model project will not be picked up automatically by running the Oracle ADF Model Tester. You can, however, force the Oracle ADF Model Tester to reload metadata from the data model project any time you want to synchronize the displayed data model and the data model project. This option is an alternative to quitting the Oracle ADF Model Tester, editing your project, and rerunning the Oracle ADF Model Tester to view the latest changes.

Using the **Reload Application** option saves time, especially as you work iteratively between the Oracle ADF Model Tester and JDeveloper. For example, while running the Oracle ADF Model Tester you might determine the need to modify the data model with a new view instance or you might find that a view instance is missing an LOV attribute definition. You can return to JDeveloper and use the Business Components overview editors to make the changes that alter the data model metadata. Then, after you recompile the project (a necessary step), you can return to the Oracle ADF Model Tester to reload the updated metadata from the project's class path.

Before you begin:

It may be helpful to have an understanding of the Oracle ADF Model Tester. For more information, see Section 6.3, "Testing View Object Instances Using the Oracle ADF Model Tester."

To reload the data model metadata in the running Oracle ADF Model Tester:

1. In the Application Navigator, right-click the application module and choose **Run**.
2. Test the data model and determine any changes you want to make. Do not exit the Oracle ADF Model Tester.
3. In JDeveloper, make the desired changes and recompile the data model project. (For example, you can right-click the data model project in the Application Navigator and choose **Make** to complete the recompile step.)

   Although the metadata changes that you make are not involved in compiling the project, the compile step is necessary to copy the metadata to the class path and to allow the Oracle ADF Model Tester to reload it.
4. Return to the Oracle ADF Model Tester and click the **Reload the Application Metadata** button above the data model tree. The Oracle ADF Model Tester closes all open windows.

   Alternatively, you can choose **Reload Application** from the **File** menu of the Oracle ADF Model Tester.
5. Reopen the desired windows and view your changes.

6.3.4 What Happens When You Use the Oracle ADF Model Tester

When you launch the Oracle ADF Model Tester, JDeveloper starts the tool in a separate process and the Oracle ADF Model Tester appears. The tree at the left of the dialog displays all of the view object instances in your application module's data model. After you double-click the desired view object instance, the Oracle ADF Model Tester will display a data view page to inspect the query results. For example, Figure 6–7 shows the view instance `Products` that has been double-clicked in the

expanded tree to display the data for this view instance in the data view page on the right.

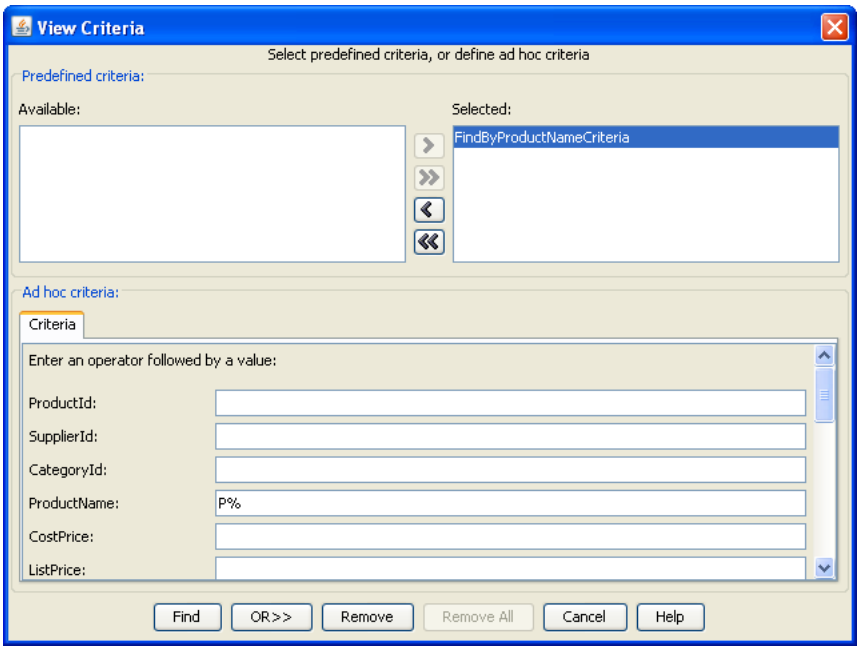
The data view page will appear disabled for any read-only view objects you display because the data is not editable. But even for a read-only view object, the tool affords some useful features:

- You can validate that the UI hints based on the Label Text hint and format masks are defined correctly.
- You can also scroll through the data using the toolbar buttons.
- You can enter Query-by-Example criteria to find a particular row whose data you want to inspect. By clicking the **Specify View Criteria** button in the toolbar, the View Criteria dialog displays the list of available Query-by-Example criteria.

  For example, as shown in Figure 6–9, you can select a view criteria like `FindByProductNameCriteria` and enter a query criteria like "`P%`" for a `ProductName` attribute and click **Find** to narrow the search to only those products with a name that begins with the letter `P`.

The Oracle ADF Model Tester becomes even more useful when you create entity-based view objects that allow you to simulate inserting, updating, and deleting rows, as described in Section 6.3.2, "How to Test Entity-Based View Objects Interactively."

Figure 6–9 Built-in Query-by-Example Functionality

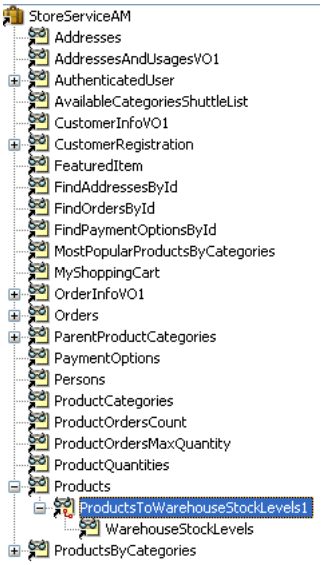


6.3.5 How to Simulate End-User Interaction in the Oracle ADF Model Tester

When you launch the Oracle ADF Model Tester, the tree at the left of the display shows the hierarchy of the view object instances that the data model of your application module defines. If the data model defines master-detail view instance relationships, the tree will display them as parent and child nodes. A node between the master-detail view instances represent the view link instance that performs the active master-detail coordination as the current row changes in the master. For example, in Figure 6–10 the tree is expanded to show the master-detail relationship

between the master `Products` view instance and the detail `WarehouseStockLevels` view instance. The selected node, `ProductsToWarehouseStockLevels1`, is the view link instance that defines the master-detail relationship.

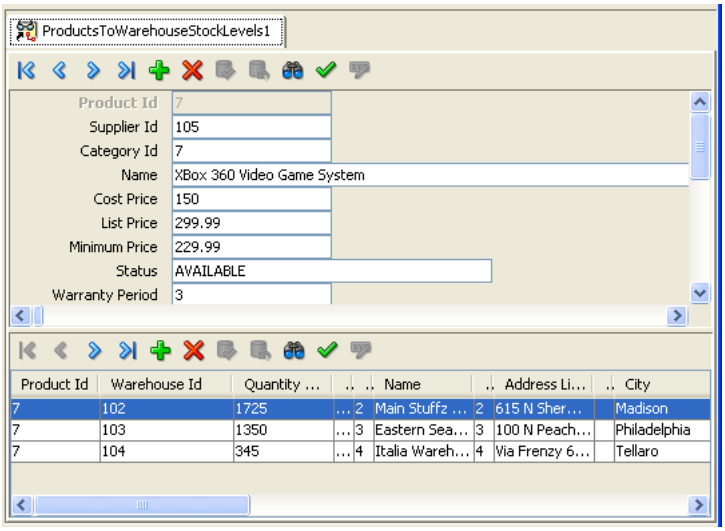
Figure 6–10 Application Module Data Model in the Oracle ADF Model Tester



Double-clicking the view link instance executes the master object and displays the master-detail data in the data view page. For example, in Figure 6–11, double-clicking the `ProductsToWarehouseStockLevels1` view link instance in the tree executes the `Products` master view instance in the top portion of the data view page and the `WarehouseStockLevels` view instance in the bottom portion of the data view page. Additional context menu items on the view object node allow you to reexecute the query if needed, remove the view object from the data model panel, and perform other tasks.

In the master-detail data view page, you can scroll through the query results. Additionally, because instance of entity-based view objects are fully editable, Instead of displaying *disabled* UI controls showing read-only data for a read-only view object, the data view page displays editable fields. You are free to experiment with creating, inserting, updating, validating, committing, and rolling back.

Figure 6–11 Master-Detail Data View Page in the Oracle ADF Model Tester



For example, you can view multiple levels of master-detail hierarchies, opening multiple data view pages at the same time. Use the **Detach** context menu item to open any tab into a separate window and visualize multiple view object's data at the same time.

Using just the master-detail data view page, you can test several functional areas of your application.

6.3.5.1 Testing Master-Detail Coordination

When you click the navigation buttons on the Oracle ADF Model Tester toolbar, you can see that the rows for the current master view object are correctly coordinated. For example, Figure 6–11 shows a master-detail hierarchy with products and warehouses. If you click the **Next Row** button in the master panel, the master panel will display the next product (identified by a product ID) and the detail panel will update to display the list of warehouses and quantities available for the product.

6.3.5.2 Testing UI Hints

The entity-based view object attributes inherit their UI hints from those on the underlying entity object attribute. The prompts displayed in the data view page help you see whether you have correctly defined a user-friendly label text UI hint for each attribute. For details on setting up the hint on your entity object, see Section 5.13, "Defining UI Hints for View Objects."

6.3.5.3 Testing Business Domain Layer Validation

Depending on the validation rules you have defined, you can try entering invalid values to trigger and verify validation exceptions. For example, when you have defined a range validation rule, enter a value outside the range and see an error similar to:

```
(oracle.jbo.AttrSetValException) Valid product codes are between 100 and 999
```

Click the rollback button in the toolbar to revert data to the previous state.

6.3.5.4 Testing Alternate Language Message Bundles and UI Hints

When your application defines alternative languages in your resource message bundles, you can configure the Oracle ADF Model Tester to recognize these languages. In the Oracle ADF Model Tester, you can then display the **Locale** menu and select among the available language choices.

To specify a default language for the Oracle ADF Model Tester:

1. Choose **Preferences** from the JDeveloper **Tools** menu.
2. Expand **ADF Business Components** in the selection panel, and select **Tester**.
3. To execute an entity-based view object in the Oracle ADF Model Tester, expand the data model tree and double-click the desired view object node.
4. In the Oracle ADF Model Tester page, add any locale for which you have created a resource message bundle to the **Selected** list.

Alternatively, you can configure the default language choice by setting ADF Business Components runtime configuration properties. These runtime properties also determine which language the Oracle ADF Model Tester will display as the default. In the Select Oracle ADF Model Tester Configuration dialog, select the **Properties** tab and enter the desired country code for the country and language. For example, to specify the Italian language, you would enter `IT` and `it` for these two properties:

- `jbo.default.country` = `IT`
- `jbo.default.language` = `it`

Testing the language message bundles in the Oracle ADF Model Tester lets you verify that the translations of the entity object UI hints are correctly located. Or, if the message bundle defines date formats for specific attributes, the tool lets you verify that date formats change (like `04/12/2007` to `12/04/2007`).

6.3.5.5 Testing View Objects That Reference Entity Usages

By scrolling through the data — or using the **Specify View Criteria** button in the Oracle ADF Model Tester toolbar to search — you can verify whether you have correctly altered the `WHERE` clause in an entity-based view object's query to use an outer join. The rows should appear as expected.

You also can try changing a primary key attribute of a master view object. This will allow you to verify that the corresponding reference information is automatically updated to reflect the new primary key value.

Use the Oracle ADF Model Tester to verify that UI hints defined at the view object level override the ones it would normally inherit from the underlying entity object. If you notice that several attributes share the same label text, you can edit the UI hint for the desired attributes at the view object level. For example, you can set the **Label Text** hint to **Member Since** for the `RegisteredDate` attribute and **Provisioned?** for the `ProvisionedFlag` attribute.

6.3.5.6 Testing Row Creation and Default Value Generation

When displaying an entity-based view object, click the **Create Row** button in the Oracle ADF Model Tester toolbar for the view object instance to create a new blank row. Any fields that have a declarative default value will appear with that value in the blank row. If the a `DBSequence`-valued attribute is used, a temporary value will appear in the new row. After entering all the required fields, click the **Commit** button to commit the transaction. The actual, trigger-assigned primary key should appear in the field after successful commit.

6.3.5.7 Testing That New Detail Rows Have Correct Foreign Keys

If you click **Create Row** in the Oracle ADF Model Tester toolbar to try adding a new row to an existing detail entity-based view object instance, you'll notice that the view link automatically ensures that the foreign key attribute value in the new row is set to the value of the current master view instance row.

6.3.6 How to Test Multiuser Scenarios in the Oracle ADF Model Tester

When view objects and entity objects cooperate at runtime, two exceptions can occur when you run the application in a multiuser environment. To anticipate these exceptions, you can simulate a multiuser environment for testing purposes using the Oracle ADF Model Tester. For example, when the application displays edit forms for view object queries, what is the expected behavior when two users attempt to modify the same attribute in their forms?

To understand the expected behavior, open two instances of the Oracle ADF Model Tester on the application module to simulate two users editing the same view object attribute. Keep both instances open and perform the following two tests to demonstrate how multiuser exceptions can arise:

- In one instance of the Oracle ADF Model Tester, modify an attribute of an existing view object and tab out of the field. Then, in the other tester instance, try to modify the same view object attribute in some way. You'll see that the second user gets the `oracle.jbo.AlreadyLockedException`.

  You can then change the value of `jbo.locking.mode` to be `pessimistic` on the Properties page of the Oracle ADF Model Tester Connect dialog and try repeating the test (the default mode is set to `pessimistic`). You'll see the error occurs for the second user immediately after changing the value instead of after committing the change.

- In one instance of the Oracle ADF Model Tester, modify an attribute of an existing view object and tab out of the field. Then, in the other tester instance, retrieve (but don't modify) the same view object attribute. Back in the first window, commit the change. If the second user then tries to modify that same attribute, you'll see that the second user gets the `oracle.jbo.RowInconsistentException`. The row has been modified and committed by another user since the second user retrieved the row into the entity cache.

6.3.7 How to Customize Configuration Options Before Running the Tester

Using the Select Business Components Configuration dialog, you can select a predefined configuration to run the tool using that named set of runtime configuration properties. The Select Configuration dialog also features a **Properties** tab that allows you to see the selected configurations settings and to override any of the configuration's settings for the current run of the tester. For example, you could alter the default language for the UI hints for a single instance of the Oracle ADF Model Tester by opening the **Properties** tab and setting the following two properties with the desired country code (in this case, `IT` for Italy):

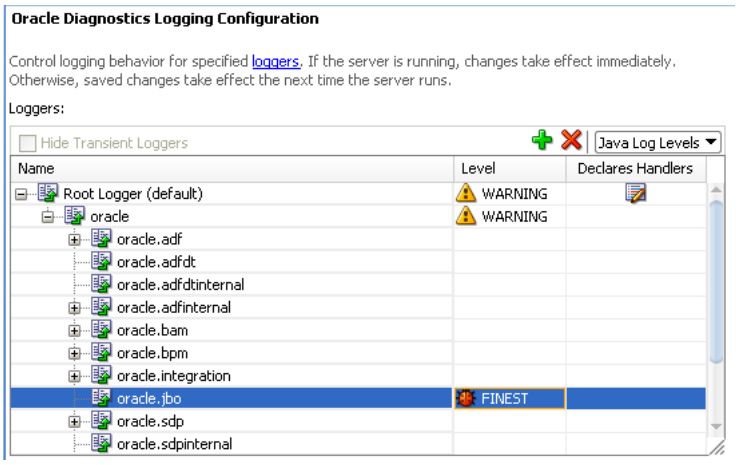
- `jbo.default.country = IT`
- `jbo.default.language = it`

Tip: If you wanted to make the changes to your configuration permanent, you could use the Configuration Manager to copy the current configuration and create a new configuration in which you set the desired properties set. For example, anytime you wanted to test in Italian you could simply choose to use the `UserServiceLocalItalian` configuration, instead of the default `UserServiceLocal`.

6.3.8 How to Enable ADF Business Components Debug Diagnostics

When launching the Oracle ADF Model Tester, if you have configured diagnostic logging for the `oracle.jbo` logger, JDeveloper will direct ADF Business Components debug diagnostics messages to the JDeveloper Log window. Figure 6–12 shows the `oracle.jbo` logger set to a log level of `FINEST` to enable ADF Business Components debug diagnostics.

Figure 6–12 Configuring a Logger for ADF Business Components Debugging



The `oracle.jbo` logger can be configured either before running the application or while the application is running in Integrated WebLogic Server. The logging will begin without the need to restart the server. You do not need to run the application in debug mode to log diagnostic messages.

With the `oracle.jbo` logger configured, the next time you run the Oracle ADF Model Tester and double-click the view object, you'll see detailed diagnostic output in the Log window, as shown in Example 6–1. Configuring the `oracle.jbo` logger with a log level `FINEST` will allow you to visualize everything the ADF Business Components framework components are doing for your application.

Example 6–1 Diagnostic Output of Oracle ADF Model Tester

```
  :
[355] Oracle SQLBuilder: Registered driver: oracle.jdbc.OracleDriver
[356] Creating a new pool resource
[357] **** DBTransactionImpl establishNewConnection
[358] Successfully logged in
[359] JDBCDriverVersion: 11.1.0.7.0-Production
[360] DatabaseProductName: Oracle
[361] DBTransactionImpl initTransaction
[362] Replacing: null with: StoreServiceAM_AddressesPageDef
[363] Replacing: null with: StoreServiceAM_MostPopularProductsByCategoriesPageDef
...
```

```
[537] Orders ViewRowSetImpl.execute caused params to be "un"changed
[538] Column count: 41
[539] ViewObject: Orders Created new QUERY statement
[540] Orders>#q computed SQLStmtBufLen: 952, actual=865, storing=895
[541] SELECT OrderEO.ORDER_ID, OrderEO.ORDER_DATE, OrderEO.ORDER_SHIPPED_DATE,
       FROM ORDERS OrderEO ORDER BY OrderEO.ORDER_DATE desc
[542] Bind params for ViewObject: Orders
```

For backward compatibility, it remains possible to enable ADF Business Components debug diagnostics on the data model project using the Java system property `jbo.debugoutput=console`. To set the property, open the Run/Debug/Profile page in the Project Properties dialog for your data model project. Click **Edit** to edit the chosen run configuration, and add `-Djbo.debugoutput=console` to the **Java Options** field in the page. Other legal values for this property are `silent` (the default, if not specified) and `file`. If you choose the file option, diagnostics are written to the system `temp` directory.

Other legal values for the `-Djbo.debugoutput` system property are `silent` (the default, if not specified) and `file`. If you enter the `file` option, diagnostics are written to the system `temp` directory.

6.3.9 What Happens at Runtime: How View Objects and Entity Objects Cooperate

On their own, view objects and entity objects simplify two important jobs that every enterprise application developer needs to do:

- Work with SQL query results
- Modify and validate rows in database tables

Entity-based view objects can query any selection of data that you want the end user to be able to view and modify. Any data the end user is allowed to change will be validated and saved by your reusable business domain layer. The key ingredients you provide as the developer are the ones that only *you* can know:

- You decide what business logic should be enforced in your business domain layer
- You decide what queries describe the data you need to put on the screen

These are the things that make *your* application unique. The built-in functionality of your entity-based view objects handles the rest of the implementation details.

> **Note:** Understanding row keys and what role the entity cache plays in the transaction are important concepts that help to clarify the nature of the entity-based view objects. These two concepts are addressed in Section 6.4.1, "ViewObject Interface Methods for Working with the View Object's Default RowSet."

6.3.9.1 What Happens at Runtime: After a View Object Executes Its Query

After adding an instance of an entity-based view object to the application module's data model, you can see what happens at runtime when you execute the query. Like a read-only view object, an entity-based view object sends its SQL query straight to the database using the standard Java Database Connectivity (JDBC) API, and the database produces a result set. In contrast to its read-only counterpart, however, as the entity-based view object retrieves each row of the database result set, it partitions the row attributes based on which entity usage they relate to. This partitioning occurs by creating an entity object row of the appropriate type for each of the view object's entity usages, populating them with the relevant attributes retrieved by the query, and

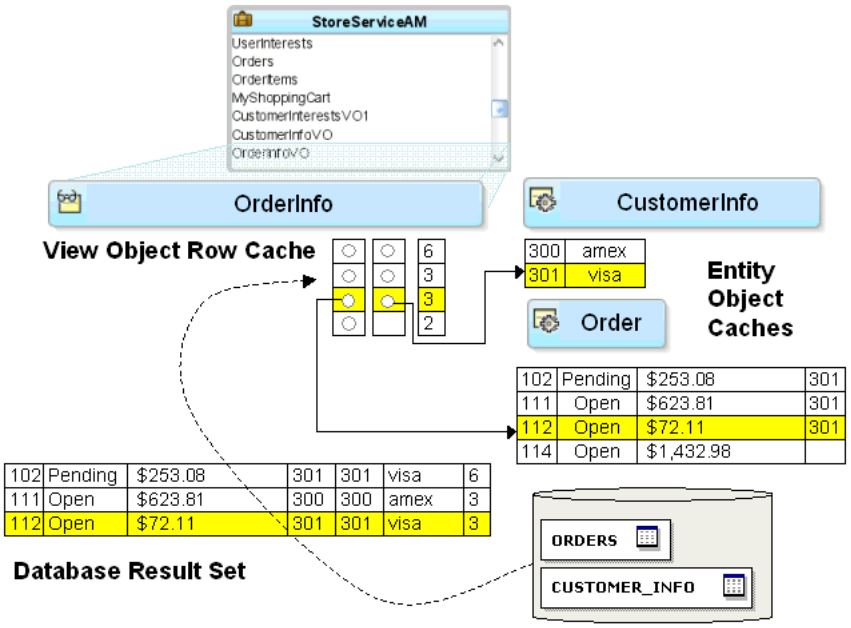
storing each of these entity rows in its respective entity cache. Then, rather than storing duplicate copies of the data, the view row simply *points* at the entity row parts that comprise it.

Figure 6–13 illustrates how the entity cache partitions the result set attributes of two entity-based view objects. In this example, the highlighted row in the database result set is partitioned into an `Order` entity row with primary key `112` and a `CustomerInfo` entity row with primary key `301`.

As described in Section 6.4.1.2, "The Role of the Entity Cache in the Transaction," the entity row that is brought into the cache using `findByPrimaryKey()` contains *all* attributes of the entity object. In contrast, an entity row created by partitioning rows from the entity-based view object's query result contains values only for attributes that appear in the query. It does *not* include the complete set of attributes. This partially populated entity row represents an important runtime performance optimization.

Since the ratio of rows retrieved to rows modified in a typical enterprise application is very high, you can save memory by bringing only the attributes into memory that you need to display instead of bringing all attributes into memory all the time.

Figure 6–13 Entity Cache Partitions View Rows into Entity Rows



By partitioning queried data this way into its underlying entity row constituent parts, the first benefit you gain is that all of the rows that include some data queried will display a consistent result when changes are made in the current transaction. In other words, if one view object allows the `PaymentType` attribute of customer `301` to be modified, then all rows in any entity-based view object showing the `PaymentType` attribute for customer `301` will update instantly to reflect the change. Since the data related to customer `301` is stored exactly once in the `CustomerInfo` entity cache in the entity row with primary key `301`, any view row that has queried the order's `PaymentType` attribute is just pointing at this single entity row.

Luckily, these implementation details are completely hidden from a client working with the rows in a view object's row set. The client works with a view row, getting and setting the attributes, and is unaware of how those attributes might be related to entity rows behind the scenes.

6.3.9.2 What Happens at Runtime: After a View Row Attribute Is Modified

When a user attempts to update the attribute of a view row, a series of steps occur to automatically coordinate this view row attribute modification with the underlying entity row. These steps ensure that a validation rule defined on the entity-mapped attribute will be triggered before the value is changed.

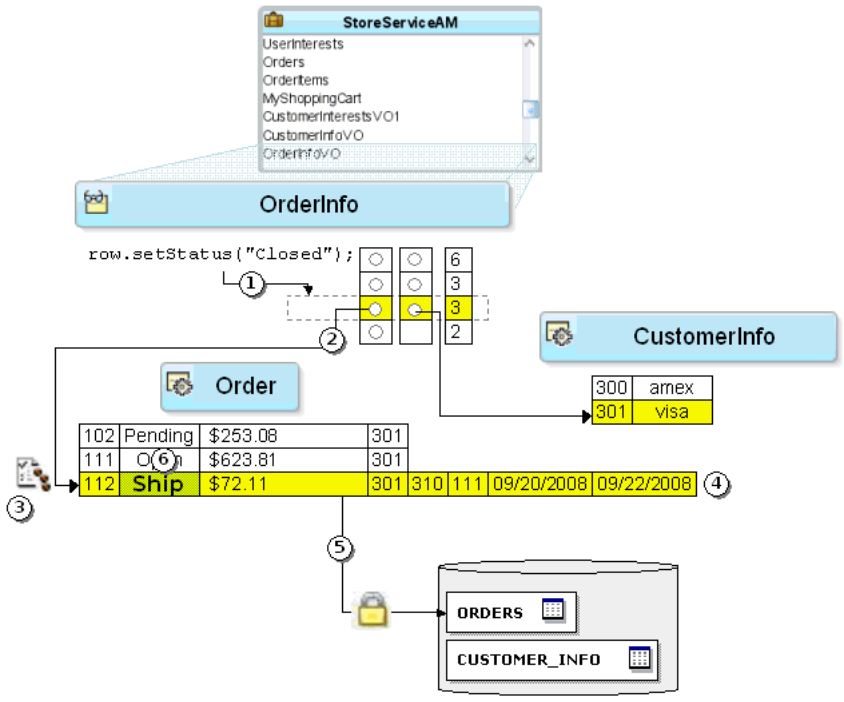
Figure 6–14 illustrates the basic steps that occur at runtime when the user attempts to update an entity-mapped attribute. In this example, the modified attribute `Status` is mapped to an entity usage where a validation rule is defined.

1. The user attempts to set the `Status` attribute to the value `Ship`.
2. Since `Status` is an entity-mapped attribute from the `Order` entity usage, the view row delegates the attribute set to the appropriate underlying entity row in the `Order` entity cache having primary key `112`.
3. Any attribute-level validation rules on the `Status` attribute of the `Order` entity object are evaluated and the modification attempt will fail if any rule does not succeed.

   Assume that some validation rule for the `Status` attribute programmatically references the `ShipDate` attribute (for example, to enforce a business rule that an `Order` cannot be shipped the same day it is placed). The `ShipDate` was not one of the `Order` attributes retrieved by the query, so it is not present in the partially populated entity row in the `Order` entity cache.
4. To ensure that business rules can always reference all attributes of the entity object, the entity object detects this situation and "faults-in" the entire set of `Order` entity object attributes for the entity row being modified using the primary key (which must be present for each entity usage that participates in the view object).
5. After the attribute-level validations all succeed, the entity object attempts to acquire a lock on the row in the `ORDERS` table before allowing the first attribute to be modified.
6. If the row can be locked, the attempt to set the `Status` attribute in the row succeeds and the value is changed in the entity row.

> **Note:** The `jbo.locking.mode` configuration property controls how rows are locked. The default value is `optimistic`. Typically, Fusion web applications will use the default setting `optimistic`, so that rows aren't locked until transaction commit time. In `pessimistic` locking mode, the row must be lockable before any change is allowed to it in the entity cache.

Figure 6–14 View Row Attribute Updates Delegate to the Entity



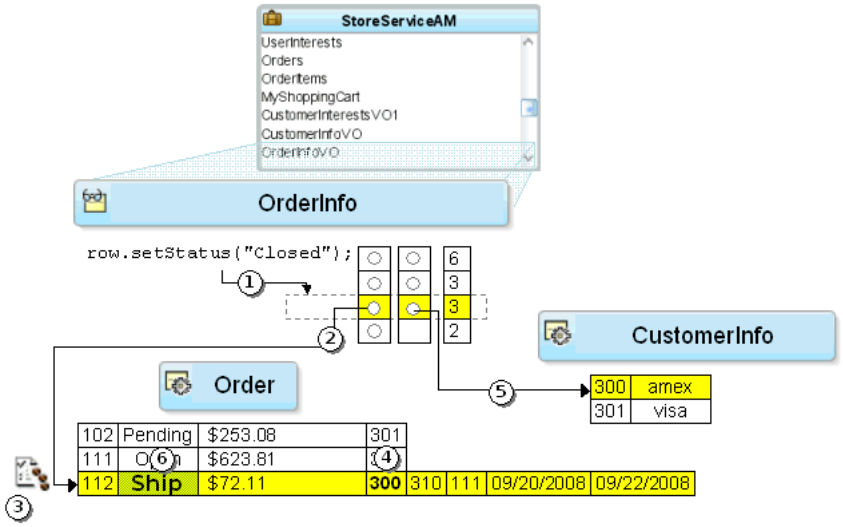
6.3.9.3 What Happens at Runtime: After a Foreign Key Attribute is Changed

When a user attempts to update a foreign key attribute, a series of steps occur to automatically coordinate this view row attribute modification with the underlying entity row. These steps ensure that a validation rule defined on the foreign key, entity-mapped attribute will be triggered before the value is changed. They also ensure that the view row for the changed foreign key attribute reflects the correct attributes of all referenced entity objects.

Figure 6–15 illustrates the basic steps that occur at runtime when the user attempts to update a foreign key, entity-mapped attribute. In this example, the modified attribute `CustomerInfoId` is mapped to an entity usage `Order` where the attribute is associated with another entity object `CustomerInfo`.

1. The user attempts to set the `CustomerInfoId` attribute to the value `300`.
2. Since `CustomerInfoId` is an entity-mapped attribute from the `Order` entity usage, the view row delegates the attribute set to the appropriate underlying entity row in the `Order` entity cache, which has primary key `112`.
3. Any attribute-level validation rules on the `CustomerInfoId` attribute of the `Order` entity object are evaluated and the modification attempt will fail if any rule does not succeed.
4. The row is already locked, so the attempt to set the `CustomerInfoId` attribute in the row succeeds and the value is changed in the entity row.
5. Since the `CustomerInfoId` attribute on the `Order` entity usage is associated with the `CustomerInfo` entity object, this change of foreign key value causes the view row to replace its current entity row part for customer `301` with the entity row corresponding to the new `CustomerInfoId = 300`. This effectively makes the view row for order `112` point to the entity row for `300`, so the value of the `PaymentType` in the view row updates to reflect the correct reference information for this newly assigned customer.

Figure 6–15 After Updating a Foreign Key, View Row Points to a New Entity



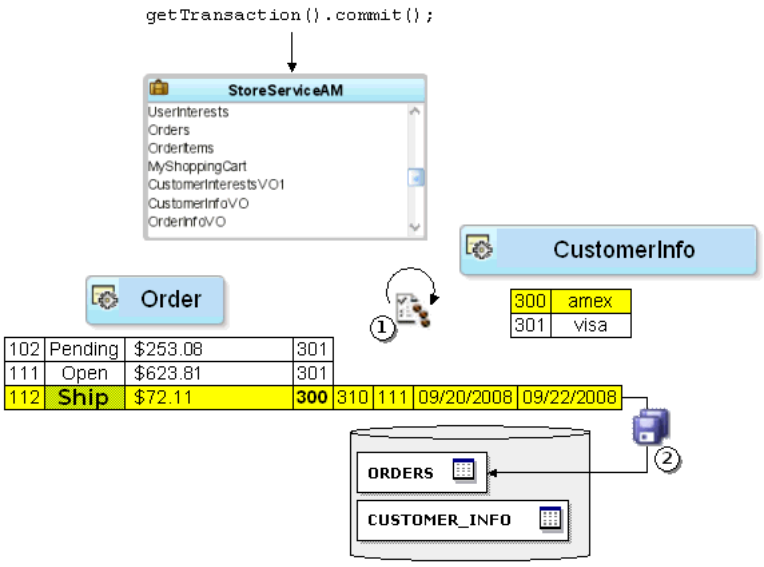
6.3.9.4 What Happens at Runtime: After a Transaction is Committed

Suppose the user is satisfied with the changes, and commits the transaction. As shown in Figure 6–16, there are two basic steps:

1. The `Transaction` object validates any invalid entity rows in its pending changes list.
2. The entity rows in the pending changes list are saved to the database.

The figure depicts a loop in Step 1 before the act of validating one modified entity object might programmatically affect changes to other entity objects. Once the transaction has processed its list of invalid entities on the pending changes list, if the list has entities, the transaction will complete another pass. It will attempt up to ten passes through the list. If by that point there are still invalid entity rows, it will throw an exception because this typically means you have an error in your business logic that needs to be investigated.

Figure 6–16 Committing the Transaction Validates Invalid Entities, Then Saves Them



6.3.9.5 What Happens at Runtime: After a View Object Requeries Data

When you reexecute a view object's query, by default the view rows in its current row set are "forgotten" in preparation for reading in a fresh result set. This view object operation does not directly affect the entity cache, however. The view object then sends the SQL to the database and the process begins again to retrieve the database result set rows and partition them into entity row parts.

Note: Typically when the view object requeries data, you expect it to retrieve the latest database information. If instead you want to avoid a database roundtrip by restricting your view object to querying only over existing entity rows in the cache, or over existing rows already in the view object's row set, see Section 42.5, "Performing In-Memory Sorting and Filtering of Row Sets."

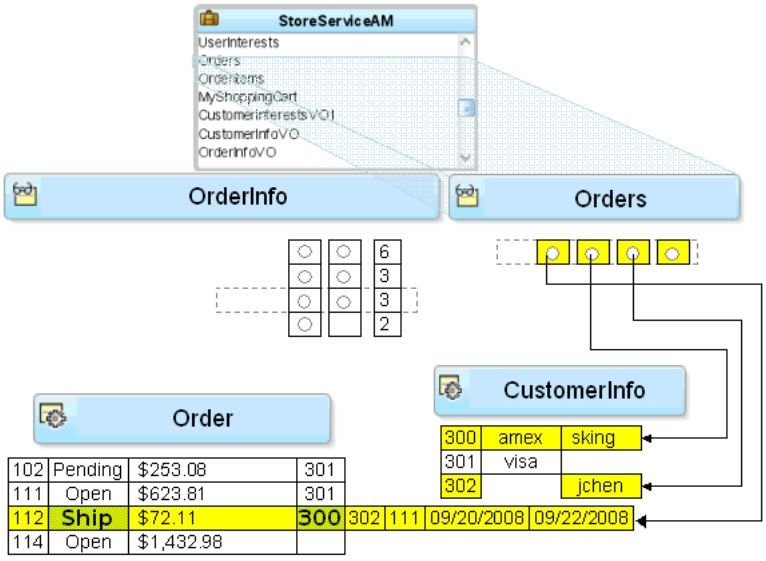
6.3.9.5.1 How Unmodified Attributes are Handled During Requery As part of the entity row partitioning process during a requery, if an attribute on the entity row is unmodified, then its value in the entity cache is updated to reflect the newly queried value.

6.3.9.5.2 How Modified Attributes are Handled During Requery However, if the value of an entity row attribute has been *modified* in the current transaction, then during a requery the entity row partitioning process does not refresh its value. Uncommitted changes in the current transaction are left intact so the end-user's logical unit of work is preserved. As with any entity attribute value, these pending modifications continue to be consistently displayed in any entity-based view object rows that reference the modified entity rows.

Note: End-user row inserts and deletes are also managed by the entity cache, which permits new rows to appear and deleted rows to be skipped during requerying. For more information about new row behavior, see Section 42.1.2, "Maintaining New Row Consistency in View Objects Based on the Same Entity."

For example, Figure 6–17 illustrates the scenario where a user "drills down" to a different page that uses the `Orders` view object instance to retrieve all details about order `112` and that this happens in the context of the current transaction's pending changes. That view object has two entity usages: a primary `Orders` usage and a reference usage for `CustomerInfo`. When its query result is partitioned into entity rows, it ends up pointing at the same `Order` entity row that the previous `OrderInfo` view row had modified. This means the end user will correctly see the pending change, that the order is assigned to `sking` in this transaction.

Figure 6–17 Entity Cache Merges Sets of Entity Attributes from Different View Objects



6.3.9.5.3 How Overlapping Subsets of Attributes are Handled During Requery Two different view objects can retrieve two *different* subsets of reference information and the results are merged whether or not they have matching sets of attributes. For example, Figure 6–17 also illustrates the situation, where the `Orders` view object queries the user's `Email`, while the `OrderInfo` view object queried the user's `PaymentOption`. The figure shows what happens at runtime: if while partitioning the retrieved row, the entity row part contains a different set of attributes than does the partially populated entity row that is already in the cache, the attributes get "merged". The result is a partially populated entity row in the cache with the *union* of the overlapping subsets of user attributes. In contrast, for `jchen` (user 302), who wasn't in the cache already, the resulting new entity row contains only the `Email` attribute, but not the `PaymentOption`.

6.3.10 What You May Need to Know About Optimizing View Object Runtime Performance

The view object provides tuning parameters that let you control how SQL is executed and how data is fetched from the database. These tuning parameters play a significant role in the runtime performance of the view object. If the fetch options are not tuned correctly for the application, then your view object may fetch an excessive amount of data and may make too many roundtrips to the database.

You can use the **Tuning** section of the General page of the overview editor to configure the fetch options shown in Table 6–2.

Table 6–2 Parameters to Tune View Object Performance

Fetch Tuning Parameters	Usage
Fetch Mode	The default fetch option is the **All Rows** option, which will be retrieved **As Needed** (`FetchMode="FETCH_AS_NEEDED"`) or **All at Once** (`FetchMode="FETCH_ALL"`), depending on which option is desired. The **As Needed** option ensures that an `executeQuery()` operation on the view object initially retrieves only as many rows as necessary to fill the first page of a display, whose number of rows is set based on the view object's range size.
Fetch Size	In conjunction with the **Fetch Mode** option, the **in Batches of** field controls the number of records fetched at one time from the database (`FetchSize` in the view object XML). The default value is 1, which will give poor performance unless only one row will be fetched. The suggested configuration is to set this value to *n*+1 where *n* is the number of rows to be displayed in the user interface.
Max Fetch Size	The default max fetch size for a view object is -1, which means that there is no limit to the number of rows the view object can fetch. In cases where the result set should contain only *n* rows of data, the option **Only up to row number** should be selected and set to *n*. The developer can alternatively call `setMaxFetchSize(n)` to set this programmatically or manually add the parameter `MaxFetchSize` to the view object XML. For view objects whose `WHERE` clause expects to retrieve a single row, set the option **At Most One Row**. This way the view object knows you don't expect any more rows and it will skip its normal test for that situation. As mentioned earlier, setting a maximum fetch size of 0 (zero) makes the view object insert-only. In this case, no select query will be issued, so no rows will be fetched. When you want to specify a global threshold for all view object queries in the application, you can configure the **Row Fetch Limit** property in the `adf-config.xml` file. Setting this property means you can avoid changing the **Max Fetch Size** for individual query operations. If you do specify a fetch limit for individual view objects, the **Row Fetch Limit** setting will be ignored in those cases. For more details about **Row Fetch Limit**, see Section 42.1.1, "Limiting the View Object Max Fetch Size to Fetch the First n Rows."

Table 6–2 (Cont.) Parameters to Tune View Object Performance

Fetch Tuning Parameters	Usage
Forward-only Mode	If a data set will only be traversed going forward, then forward-only mode can help performance when iterating through the data set. This can be configured by programmatically calling `setForwardOnly(true)` on the view object. Setting forward-only will also prevent caching previous sets of rows as the data set is traversed.

When you tune view objects, you should also consider these issues:

- Large data sets: View objects provide a mechanism to page through large data sets such that a user can jump to a specific page in the results. This is configured by calling `setRangeSize(`*n*`)` followed by `setAccessMode(RowSet.RANGE_PAGING)` on the view object where *n* is the number of rows contained within one page. When the user navigates to a specific page in the data set, the application can call `scrollToRangePage(P)` on the view object to navigate to page *P*. Range paging fetches and caches only the current page of rows in the view object row cache at the cost of another query execution to retrieve each page of data. Range paging is not appropriate where it is beneficial to have all fetched rows in the view object row cache (for example, when the application needs to read all rows in a dataset for an LOV or page back and forth in records of a small data set.
- Spillover: There is a facility to use the data source as "virtual memory" when the JVM container runs out of memory. By default, this is disabled and can be turned on as a last resort by setting `jbo.use.pers.coll=true`. Enabling spillover can have a large performance impact.
- SQL platform: If the generic SQL92 SQL platform is used to connect to generic SQL92-compliant databases, then some view object tuning options will not function correctly. The parameter that choosing the generic SQL92 SQL platform affects the most is the fetch size. When SQL92 SQL platform is used, the fetch size defaults to 10 rows regardless of what is configured for the view object. You can set the SQL platform when you define the database connection or you can define it as global project setting in the `adf-config.xml` file. By default, the SQL platform will be `Oracle`. To manually override the SQL platform, you can also pass the parameter `-Djbo.SQLBuilder="SQL92"` to the JVM upon startup.

Additionally, you have some options to tune the view objects' associated SQL for better database performance:

- Bind variables: If the query associated with the view object contains values that may change from execution to execution, use bind variables. Using bind variables in the query allows the query to reexecute without needing to reparse the query on the database. You can add bind variables to the view object in the Query page of the overview editor for the view object. For more information, see Section 5.10, "Working with Bind Variables."
- Query optimizer hints: The view object can pass hints to the database to influence which execution plan to use for the associated query. The optimizer hints can be specified in the **Retrieve from the Database** group box in the Tuning section of the overview editor for the view object. For information about optimizer hints, see Section 42.2.4.3, "Specify a Query Optimizer Hint if Necessary."

6.4 Testing View Object Instances Programmatically

When you are ready to test a working application module containing at least one view object instance, you can build a simple test client program to illustrate the basics of working programmatically with the data in the contained view object instances.

From the point of view of a client accessing your application module's data model, the API's to work with a read-only view object and an entity-based view object are identical. The key functional difference is that entity-based view objects allow the data in a view object to be fully updatable. The application module that contains the entity-based view objects defines the unit of work and manages the transaction. This section presents four simple test client programs that work with the `StoreFrontAM` application module in the Fusion Order Demo to illustrate:

- Iterating master-detail-detail hierarchy
- Finding a row and updating a foreign key value
- Creating a new order
- Retrieving the row key identifying a row

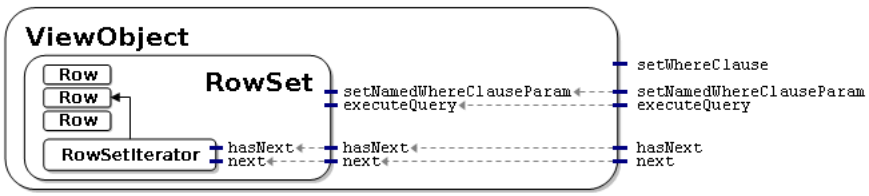
6.4.1 ViewObject Interface Methods for Working with the View Object's Default RowSet

The `ViewObject` interface in the `oracle.jbo` package provides the methods to easily perform any data-retrieval task. Some of these methods used in the example include:

- `executeQuery()`, to execute the view object's query and populate its row set of results
- `setWhereClause()`, to add a dynamic predicate at runtime to narrow a search
- `setNamedWhereClauseParam()`, to set the value of a named bind variable
- `hasNext()`, to test whether the row set iterator has reached the last row of results
- `next()`, to advance the row set iterator to the next row in the row set
- `getEstimatedRowCount()`, to count the number of rows a view object's query would return

Typically, when you work with a view object, you will work with only a single row set of results at a time. To simplify this overwhelmingly common use case, as shown in Figure 6–18, the view object contains a default `RowSet`, which, in turn, contains a default `RowSetIterator`. The default `RowSetIterator` allows you to call all of the data-retrieval methods directly on the `ViewObject` component itself, knowing that they will apply automatically to its default row set.

Figure 6–18 ViewObject Contains a Default RowSet and RowSetIterator



Note: Chapter 42, "Advanced View Object Techniques" presents situations when you might want a single view object to produce *multiple* distinct row sets of results. You can also find scenarios for creating *multiple* distinct row set iterators for a row set. Most of the time, however, you'll need only a single iterator.

The phrase "working with the rows in a view object," when used in this guide more precisely means working with the rows in the view object's default row set. Similarly, the phrase "iterate over the rows in a view object," more precisely means you will use the default row set iterator of the view object's default row set to loop over its rows.

6.4.1.1 The Role of the Key Object in a View Row or Entity Row

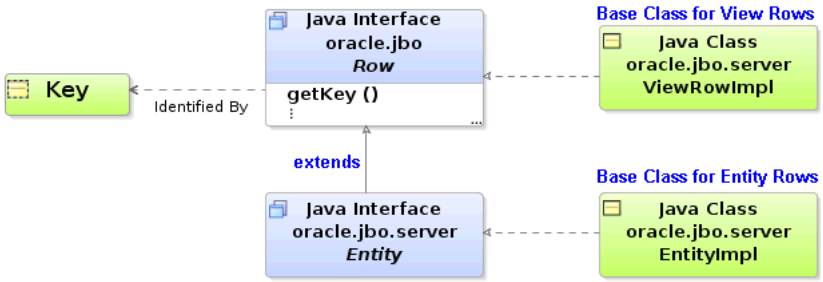
When you work with view rows you use the `Row` interface in the `oracle.jbo` package. As shown in Figure 6–19, the interface contains a method called `getKey()` that you can use to access the `Key` object that identifies any row. Notice that the `Entity` interface in the `oracle.jbo.server` package extends the `Row` interface. This relationship provides a concrete explanation of why the term *entity row* is so appropriate. Even though an entity row supports additional features for encapsulating business logic and handling database access, you can still treat any entity row as a `Row`.

An entity-based view object delegates the task of finding rows by key to its underlying entity row parts.

Recall that both view rows and entity rows support either single-attribute or multiattribute keys, so the `Key` object related to any given `Row` will encapsulate all of the attributes that comprise its key. Once you have a `Key` object, you can use the `findByKey()` method on any row set to find a row based on its `Key` object. When you use the `findByKey()` method to find a view row by key, the view row proceeds to use the entity definition's `findByPrimaryKey()` method to find each entity row contributing attributes to the view row key.

In the case of a read-only view object with no underlying entity row to which to delegate this task, the view object implementation automatically enables the `manageRowsByKey` flag when at least one primary key attribute is detected. This ensures that the `findByKey()` method is successful in the case of read-only view objects. If the `manageRowsByKey` flag is not enabled, then UI operations like setting the current row with the key, which depend on the `findByKey()` method, would not work.

Figure 6–19 Any View Row or Entity Row Supports Retrieving Its Identifying Key



Note: When you define an entity-based view object, by default the primary key attributes for all of its entity usages are marked with their **Key Attribute** property set to `true`. In any *nonupdatable* reference entity usages, you should disable the **Key Attribute** property for the key attributes. Since view object attributes related to the primary keys of *updatable* entity usages must be part of the composite view row key, their **Key Attribute** property cannot be disabled.

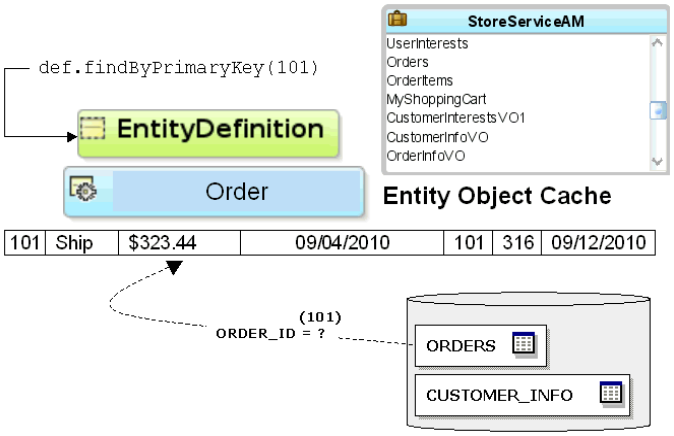
6.4.1.2 The Role of the Entity Cache in the Transaction

An application module is a transactional container for a logical unit of work. At runtime, it acquires a database connection using information from the named configuration you supply, and it delegates transaction management to a companion `Transaction` object. Since a logical unit of work may involve finding and modifying multiple entity rows of different types, the `Transaction` object provides an entity cache as a "work area" to hold entity rows involved in the current user's transaction. Each entity cache contains rows of a single entity type, so a transaction involving two or more entity objects holds the working copies of those entity rows in separate caches.

By using an entity object's related entity definition, you can write code in an application module to find and modify existing entity rows. As shown in Figure 6–20, by calling `findByPrimaryKey()` on the entity definition for the `Order` entity object, you can retrieve the row with that key. If it is not already in the entity cache, the entity object executes a query to retrieve it from the database. This query selects all of the entity object's persistent attributes from its underlying table, and finds the row using an appropriate `WHERE` clause against the column corresponding to the entity object's primary key attribute. Subsequent attempts to find the same entity row by key during the same transaction will find it in the cache, preventing the need for a trip to the database. In a given entity cache, entity rows are indexed by their primary key. This makes finding an entity row in the cache a fast operation.

When you access related entity rows using association accessor methods, they are also retrieved from the entity cache. If related entity rows are not in the cache, then they are retrieved from the database. Finally, the entity cache is also the place where new entity rows wait to be saved. In other words, when you use the `createInstance2()` method on the entity definition to create a new entity row, it is added to the entity cache.

Figure 6–20 Entity Cache Stores Entity Rows During the Transaction



When an entity row is created, modified, or removed, it is automatically enrolled in the transaction's list of pending changes. When you call `commit()` on the `Transaction` object, it processes its pending changes list, validating new or modified entity rows that might still be invalid. When the entity rows in the pending list are all valid, the `Transaction` issues a database `SAVEPOINT` and coordinates saving the entity rows to the database. If all goes successfully, it issues the final database `COMMIT` statement. If anything fails, the `Transaction` performs a `ROLLBACK TO SAVEPOINT` to allow the user to fix the error and try again.

The `Transaction` object used by an application module represents the working set of entity rows for a single end-user transaction. By design, it is *not* a shared, global cache. The database engine itself is an extremely efficient shared, global cache for multiple, simultaneous users. Rather than attempting to duplicate all the work of fine-tuning that has gone into the database's shared, global cache functionality, ADF Business Components consciously embraces it. To refresh a single entity object's data from the database at any time, you can call its `refresh()` method. You can `setClearCacheOnCommit()` or `setClearCacheOnRollback()` on the `Transaction` object to control whether entity caches are cleared at commit or rollback. The defaults are `false` and `true`, respectively. The `Transaction` object also provides a `clearEntityCache()` method you can use to programmatically clear entity rows of a given entity type (or all types). When you clear an entity cache, you allow entity rows of that type to be retrieved from the database fresh the next time they are either found by primary key or retrieved by an entity-based view object.

6.4.2 How to Create a Command-Line Java Test Client

To the create a test client program, use the Create Java Class wizard, which is accessible from the New Gallery.

6.4.2.1 Generating a Test Client with Skeleton Code

When you use the Create Java Class wizard to create the test client program, JDeveloper will open your program file in the source editor and allow you to add code from a predefined code template to complete the test client.

To generate a skeleton Java test client:

1. In the Application Navigator, right-click the project in which you want to create the test client and choose **New**.
2. In the New Gallery, expand **General**, select **Java** and then **Java Class**, and click **OK**.
3. In the Create Java Class dialog, enter a class name, like `TestClient`, a package name, like `oracle.fodemo.storefront.client`, and ensure that the **Extends** field shows `java.lang.Object`.
4. In **Optional Attributes**, deselect **Constructors from Superclass** and select **Main Method**.
5. Click **OK**.

   The `.java` file opens in the source editor to show the skeleton code, as shown in Example 6–2.

Example 6–2 Skeleton Code for TestClient.java

```
package oracle.fodemo.storefront.client;
public class TestClient {
  public static void main(String[] args) {
```

```
  }
}
```

6.4.2.2 Modifying the Skeleton Code to Create the Test Client

After you generate skeleton code for the test client, you can proceed to edit the file using the predefined `bc4jclient` code template available from JDeveloper.

To insert the bc4jclient code template:

1. Place the cursor on a blank line inside the body of the `main()` method and use the `bc4jclient` code template to create the few lines of necessary code.
2. Type the characters `bc4jclient` followed Ctrl + Enter.

   JDeveloper will expand the class file with the template as shown in Example 6–3.
3. Adjust the values of the `amDef` and`config` variables to reflect the names of the application module definition and the configuration that you want to use, respectively.

   For the Example 6–3, the changed lines look like this:

   ```
   String amDef = "oracle.fodemo.storefront.store.service.StoreServiceAM";
   String config = "StoreServiceAMLocal";
   ```
4. Finally, change the view object instance name in the call to `findViewObject()` to the one you want to work with. Specify the name exactly as it appears in the **Data Model** tree on the Data Model page of the overview editor for the application module.

   For the Example 6–3, the changed line looks like this:

   ```
   ViewObject vo = am.findViewObject("Persons");
   ```

Example 6–3 Expanded Skeleton Code for TestClient.java

```
package oracle.fodemo.storefront.client;
import oracle.jbo.client.Configuration;
import oracle.jbo.*;
import oracle.jbo.domain.Number;
import oracle.jbo.domain.*;
public class TestClient {
  public static void main(String[] args) {
    String        amDef = "test.TestModule";
    String        config = "TestModuleLocal";
    ApplicationModule am =
      Configuration.createRootApplicationModule(amDef,config);
    ViewObject vo = am.findViewObject("TestView");
    // Work with your appmodule and view object here
    Configuration.releaseRootApplicationModule(am,true);
  }
}
```

Your skeleton test client for your application module should contain source code like what you see in Example 6–4.

> **Note:** The examples throughout Section 9.10, "Working Programmatically with an Application Module's Client Interface" expand this test client sample code to illustrate calling custom application module service methods, too.

Example 6–4 Working Skeleton Code for an Application Module Test Client Program

```
package oracle.fodemo.storefront.client;
import oracle.jbo.ApplicationModule;
import oracle.jbo.Row;
import oracle.jbo.RowSet;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;

public class TestClient {

  public static void main(String[] args) {
    String  amDef = "oracle.fodemo.storefront.store.service.StoreServiceAM";
    String  config = "StoreServiceAMLocal";
    ApplicationModule am =
                      Configuration.createRootApplicationModule(amDef,config);
    // 1. Find the Persons view object instance.
    ViewObject personList = am.findViewObject("Persons");
    // Work with your appmodule and view object here
    Configuration.releaseRootApplicationModule(am,true);
  }
}
```

Replace `// Work with your appmodule and view object here`, with code that will execute the view objects you want to test. For example, to execute the view object's query, display the number of rows it will return, and loop over the result to fetch the data and print it out to the console, you can adapt the code shown in Example 6–5 for your data model project components.

Example 6–5 Looping Over Master-Detail View Objects and Printing the Results to the Console

```
    // 2. Execute the query
    personList.executeQuery();
    // 3. Iterate over the resulting rows
    while (personList.hasNext()) {
        Row person = personList.next();
        // 4. Print the person's email
        System.out.println("Person: " + person.getAttribute("Email"));
        // 5. Get related rowset of Orders using view link accessor attribute
        RowSet orders = (RowSet)person.getAttribute("Orders");
        // 6. Iterate over the Orders rows
        while (orders.hasNext()) {
            Row order = orders.next();
            // 7. Print out some order attribute values
              System.out.println(" ["+order.getAttribute("OrderStatusCode")+"] "+
                                 order.getAttribute("OrderId")+": "+
                                 order.getAttribute("OrderTotal"));
              if(!order.getAttribute("OrderStatusCode").equals("COMPLETE")) {
                  // 8. Get related rowset of OrderItems
                  RowSet items = (RowSet)order.getAttribute("OrderItems");
                  // 9. Iterate over the OrderItems rows
                  while (items.hasNext()) {
```

```
                    Row item = items.next();
                    // 10. Print out some order items attributes
                    System.out.println("  "+item.getAttribute("LineItemId")+": "+
                                       item.getAttribute("LineItemTotal"));
                }
            }
        }
    }
```

The first line calls the `executeQuery()` method to execute the view object's query. This produces a row set of zero or more rows that you can loop over using a `while` statement that iterates until the view object's `hasNext()` method returns `false`. Inside the loop, the code puts the current `Row` in a variable named `person`, then invokes the `getAttribute()` method twice on that current `Row` object to get the value of the `Email` and `Orders` attributes to print order information to the console. A second `while` statement performs the same task for the line items of the order.

6.4.3 What Happens When You Run a Test Client Program

The call to `createRootApplicationModule()` on the `Configuration` object returns an instance of the application module to work with. As you might have noticed in the debug diagnostic output, the ADF Business Components runtime classes load XML component definitions as necessary to instantiate the application module and the instance of the view object component that you've defined in its data model at design time. The `findViewObject()` method on the application module finds a view object instance by name from the application module's data model. After the loop shown in Example 6–5, the test client executes `releaseRootApplicationModule()` on the `Configuration` object. This signals that you're done using the application module and it allows the framework to clean up resources, like the database connection that was used by the application module.

6.4.4 What You May Need to Know About Running a Test Client

The `createRootApplicationModule()` and `releaseRootApplicationModule()` methods are very useful for command-line access to application module components. However, you typically won't need to write these two lines of code in the context of an ADF-based web or Swing application. The ADF Model data binding layer cooperates automatically with the ADF business services layer to acquire and release ADF application module components for you in those scenarios.

6.4.5 How to Count the Number of Rows in a Row Set

The `getEstimatedRowCount()` method is used on a `RowSet` to determine how many rows it contains:

```
long numReqs = reqs.getEstimatedRowCount();
```

The implementation of the `getEstimatedRowCount()` initially issues a `SELECT COUNT(*)` query to calculate the number of rows that the query will return. The query is formulated by "wrapping" your view object's entire query in a statement like:

```
SELECT COUNT(*) FROM ( ... your view object's SQL query here ... )
```

The `SELECT COUNT(*)` query allows you to access the count of rows for a view object without necessarily retrieving all the rows themselves. This approach permits an important optimization for working with queries that return a large number of

rows, or for testing how many rows a query *would* return before proceeding to work with the results of the query.

Once the estimated row count is calculated, subsequent calls to the method do not reexecute the `COUNT(*)` query. The value is cached until the next time the view object's query is executed, since the fresh query result set returned from the database could potentially contain more, fewer, or different rows compared with the last time the query was run. The estimated row count is automatically adjusted to account for pending changes in the current transaction, adding the number of relevant new rows and subtracting the number of removed rows from the count returned.

You can also override `getEstimatedRowCount()` to perform a custom count query that suits your application's needs.

6.4.6 How to Access a Detail Collection Using the View Link Accessor

Once you've retrieved the `RowSet` of detail rows using a view link accessor, as described in Section 5.6.6.2, "Programmatically Accessing a Detail Collection Using the View Link Accessor,", you can loop over the rows it contains using the same pattern used by the view object's row set of results, as shown in Example 6–6.

Example 6–6 Pattern Used to Access a Detail Collection

```
while (reqs.hasNext()) {
  Row curReq = reqs.next();
  System.out.println("--> (" + curReq.getAttribute("OrderId") + ") " +
                     curReq.getAttribute("OrderTotal"));
}
```

Example 6–7 shows the `main()` method sets a dynamic `WHERE` clause to restrict the `PersonList` view object instance to show only persons whose `person_type_code` has the value `CUST`. Additionally, the `executeAndShowResults()` method accesses the view link accessor attribute and prints out the request number (`PersonId`) and `Email` attribute for each one.

To access the a detail collection using a view link accessor, follow these basic steps (as illustrated in Example 6–7):

1. Find the master view object instance.
2. Execute the query.
3. Iterate over the master view object rows.
4. Get the related row set of the detail view object using the view link accessor attribute.
5. Iterate over the detail view object rows.
6. Optionally, do something with the detail row set attributes.

> **Performance Tip:** If the code you write to loop over the rows does not need to display them, then you can call the `closeRowSet()` method on the row set when you're done. This technique will make memory use more efficient. The next time you access the row set, its query will be reexecuted.

Example 6–7 Programmatically Accessing Detail Rows Using the View Link Accessor

```
package devguide.examples.readonlyvo.client;

import oracle.jbo.ApplicationModule;
```

```
import oracle.jbo.Row;
import oracle.jbo.RowSet;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;

public class TestClient2 {
  public static void main(String[] args) {
    String amDef = "devguide.examples.readonlyvo.PersonService";
    String config = "PersonServiceLocal";
    ApplicationModule am =
                      Configuration.createRootApplicationModule(amDef, config);
    // 1. Find the Persons view object instance.
    ViewObject vo = am.findViewObject("PersonList");
    // Add an extra where clause with a new named bind variable
    vo.setWhereClause("person_type_code = :ThePersonType");
    vo.defineNamedWhereClauseParam("ThePersonType", null, null);
    vo.setNamedWhereClauseParam("ThePersonType", "CUST");
    // Show results when :ThePersonType = 'CUST'
    executeAndShowResults(vo);
    Configuration.releaseRootApplicationModule(am, true);
  }
  private static void executeAndShowResults(ViewObject vo) {
    System.out.println("---");
    // 2. Execute the query
    vo.executeQuery();
    // 3. Iterate over the resulting rows of the Persons view object
    while (vo.hasNext()) {
        Row curPerson = vo.next();
        // 4. Access the row set of Orders using the view link accessor attribute
        RowSet orders =(RowSet)curPerson.getAttribute("OrdersShippedToPurchaser");
        long numOrders = orders.getEstimatedRowCount();
        System.out.println(curPerson.getAttribute("PersonId") + " " +
                           curPerson.getAttribute("Email")+" ["+
                           numOrders+" orders]");
        // 5. Iterate over the resulting detail rows
        while (orders.hasNext()) {
            Row curOrder = orders.next();
        // 6. Print out some Order attribute values
            System.out.println("--> (" + curOrder.getAttribute("OrderId") + ") " +
                        curOrder.getAttribute("OrderTotal"));
        }
    }
  }
}
```

Running `TestClient2.java` produces output in the Log window, as shown in Example 6–8. Each customer is listed, and for each customer that has some orders, the order total appears beneath their name.

Example 6–8 Results of Running TestClient.java

```
---
121 AFRIPP [0 orders]
115 AKHOO [0 orders]
109 DFAVIET [0 orders]
114 DRAPHEAL [0 orders]
118 GHIMURO [0 orders]
126 IMIKKILI [0 orders]
111 ISCIARRA [0 orders]
110 JCHEN [0 orders]
```

```
127 JLANDRY [0 orders]
112 JMURMAN [0 orders]
125 JNAYER [0 orders]
119 KCOLMENA [0 orders]
124 KMOURGOS [0 orders]
129 LBISSOT [0 orders]
113 LPOPP [1 orders]
--> (1013) 89.99
120 MWEISS [1 orders]
--> (1003) 5000
108 NGREENBE [1 orders]
--> (1002) 1249.91
122 PKAUFLIN [0 orders]
116 SBAIDA [0 orders]
128 SMARKLE [0 orders]
117 STOBIAS [0 orders]
123 SVOLLMAN [0 orders]
```

If you run `TestClient2.java` with debug diagnostics enabled, you will see the SQL queries that the view object performed. The view link `WHERE` clause predicate is used to automatically perform the filtering of the detail service request rows for the current row in the `PersonList` view object.

6.4.7 How to Iterate Over a Master-Detail-Detail Hierarchy

To iterate over a master-detail with an additional level of nesting, follow these basic steps (as illustrated in Example 6–9):

1. Find the master view object instance.
2. Executes the query.
3. Iterate over the resulting rows.
4. Optionally, do something with the attributes of the master row set.
5. Get the related row set of the detail view object using the view link accessor attribute.
6. Iterate over the detail row set rows.
7. Optionally, do something with the attributes of the detail row set.
8. Get the related row set of the second detail view object using the view link accessor attribute.
9. Iterates over the second detail row set rows.
10. Optionally, do something with the second detail row set attributes.

Other than having one additional level of nesting, Example 6–9 uses the same API's used in the `TestClient` program that was iterating over master-detail read-only view objects in Section 6.4.6, "How to Access a Detail Collection Using the View Link Accessor."

If you use JDeveloper's **Refactor > Duplicate** functionality on an existing `TestClient.java` class, you can quickly "clone" it to create a `TestClient2.java` class. For example, the `TestClient.java` class in Example 6–8 is suited to this technique.

Example 6–9 Iterating Master/Detail/Detail Hierarchy

```
package oracle.fodemo.storefront.client;
```

```
import oracle.jbo.ApplicationModule;
import oracle.jbo.Row;
import oracle.jbo.RowSet;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;

public class TestClient2 {

  public static void main(String[] args) {
    String    amDef =
                "oracle.fodemo.storefront.store.service.StoreServiceAM";
    String    config = "StoreServiceAMLocal";
    ApplicationModule am =
                Configuration.createRootApplicationModule(amDef,config);
    // 1. Find the Persons view object instance.
    ViewObject personList = am.findViewObject("Persons");
    // 2. Execute the query
    personList.executeQuery();
    // 3. Iterate over the resulting rows
    while (personList.hasNext()) {
    Row person = personList.next();
    // 4. Print the person's email
    System.out.println("Person: " + person.getAttribute("Email"));
    // 5. Get related rowset of Orders using view link accessor attribute
    RowSet orders = (RowSet)person.getAttribute("Orders");
    // 6. Iterate over the Orders rows
    while (orders.hasNext()) {
        Row order = orders.next();
        // 7. Print out some order attribute values
        System.out.println(" ["+order.getAttribute("OrderStatusCode")+"] "+
                           order.getAttribute("OrderId")+": "+
                           order.getAttribute("OrderTotal"));
        if(!order.getAttribute("OrderStatusCode").equals("COMPLETE")) {
        // 8. Get related rowset of OrderItems using view link accessor attribute
        RowSet items = (RowSet)order.getAttribute("OrderItems");
        // 9. Iterate over the OrderItems rows
        while (items.hasNext()) {
            Row item = items.next();
            // 10. Print out some order items attributes
            System.out.println("  "+item.getAttribute("LineItemId")+": "+
                               item.getAttribute("LineItemTotal"));
            }
        }
        }
    }
    Configuration.releaseRootApplicationModule(am,true);
  }
}
```

Running the program produces the output shown in Example 6–10.

Example 6–10 Results of Running TestClient2.java

```
Staff Member: David Austin
 [Open] 104: Spin cycle not draining
  1: Researching issue
Staff Member: Bruce Ernst
 [Closed] 101: Agitator does not work
 [Pending] 102: Washing Machine does not turn on
  1: Called customer to make sure washer was plugged in...
```

```
  2: We should modify the setup instructions to include...
 [Open] 108: Freezer full of frost
  1: Researching issue
Staff Member: Alexander Hunold
 [Closed] 100: I have noticed that every time I do a...
 [Closed] 105: Air in dryer not hot
   :
```

6.4.8 How to Find a Row and Update a Foreign Key Value

To find a row and update a foreign key value, follow these basic steps (as illustrated in Example 6–11):

1. Find the view object instance.
2. Construct a `Key` object to look up the row for the view instance.
3. Use `findByKey()` to find the row.
4. Optionally, do something with the row's attribute.

Example 6–11 shows the `main()` method finds and updates a foreign key value to find a row of the `Orders` view object instance. The sample then prints out the existing value of the `OrderStatusCode` attribute before changing the value on the row.

Example 6–11 Finding and Updating a Foreign Key Value

```
package oracle.fodemo.storefront.client;

import oracle.jbo.ApplicationModule;
import oracle.jbo.JboException;
import oracle.jbo.Key;
import oracle.jbo.Row;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;

public class TestFindAndUpdate {
    public static void main(String[] args) {
        String  amDef =
                    "oracle.fodemo.storefront.store.service.StoreServiceAM";
        String  config = "StoreServiceAMLocal";
        ApplicationModule am =
                    Configuration.createRootApplicationModule(amDef,config);
        // 1. Find the Orders view object instance
        ViewObject vo = am.findViewObject("Orders");
        // 2. Construct a new Key to find Order # 1011
        Key orderKey = new Key(new Object[]{1011});
        // 3. Find the row matching this key
        Row[] ordersFound = vo.findByKey(orderKey,1);
        if (ordersFound != null && ordersFound.length > 0) {
            Row order = ordersFound[0];
            // 4. Print some order information
            String orderStatus = (String)order.getAttribute("OrderStatusCode");
            System.out.println("Current status is: "+ orderStatus);
            try {
                // 5. Try setting the status to an illegal value
                order.setAttribute("OrderStatusCode","REOPENED");
            }
            catch (JboException ex) {
                System.out.println("ERROR: "+ex.getMessage());
            }
            // 6. Set the status to a legal value
```

```
            order.setAttribute("OrderStatusCode","PENDING");
            // 7. Show the value of the status was updated successfully
            System.out.println("Current status is: " +
                        order.getAttribute("OrderStatusCode"));
            // 8. Show the current value of the customer for this order
            System.out.println("Customer: " + order.getAttribute("CustomerId"));
            // 9. Reassign the order to customer # 113
            order.setAttribute("CustomerId",113); // Luis Popp
            // 10. Show the value of the reference information now
            System.out.println("Customer: "+order.getAttribute("CustomerId"));
            // 11. Rollback the transaction
            am.getTransaction().rollback();
            System.out.println("Transaction canceled");
        }
        Configuration.releaseRootApplicationModule(am,true);
    }
}
```

Running this example produces the output shown in Example 6–12.

Example 6–12 Results of Running TestFindAndUpdate.java

```
Current status is: Closed
ERROR: The status must be Open, Pending, or Closed
Current status is: Open
Assigned: bernst
Assigned: Luis Popp
Transaction canceled
```

6.4.9 How to Create a New Row for a View Object Instance

To create a new view row instance, follow these basic steps (as illustrated in Example 6–13):

1. Find the view object instance.
2. Create a new row and insert it into the row set.
3. Set the values of the required attributes in the new row.
4. Commit the transaction.

Example 6–13 shows the `main()` method finds the `Orders` view object instance and inserts a new row into the row set. Because the `Orders` view object is entity-based, the `CreatedBy` attribute derives its value from the mapped entity object attribute. The sample then sets values for the remaining attributes before committing the transaction.

Example 6–13 Creating a New Order

```
package oracle.fodemo.storefront.client;
import oracle.jbo.ApplicationModule;
import oracle.jbo.Row;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;
import oracle.jbo.domain.DBSequence;
import oracle.jbo.domain.Date;
import oracle.jbo.domain.Timestamp;

public class TestCreateOrder {
    public static void main(String[] args) throws Throwable {
        String amDef = "oracle.fodemo.storefront.store.service.StoreServiceAM";
        String config = "StoreServiceAMLocal";
```

```
        ApplicationModule am =
                Configuration.createRootApplicationModule(amDef, config);
        // 1. Find the Orders view object instance.
        ViewObject orders = am.findViewObject("Orders");
        // 2. Create a new row and insert it into the row set
        Row newOrder = orders.createRow();
        orders.insertRow(newOrder);
        // Show the entity object-related defaulting for CreatedBy attribute
        System.out.println("CreatedBy defaults to: " +
                           newOrder.getAttribute("CreatedBy"));
        // 3. Set values for some of the required attributes
        Date now = new Date(new Timestamp(System.currentTimeMillis()));
        newOrder.setAttribute("OrderDate", now);
        newOrder.setAttribute("OrderStatusCode", "PENDING");
        newOrder.setAttribute("OrderTotal", 500);
        newOrder.setAttribute("CustomerId", 110);
        newOrder.setAttribute("ShipToAddressId", 2);
        newOrder.setAttribute("ShippingOptionId", 2);
        newOrder.setAttribute("FreeShippingFlag", "N");
        newOrder.setAttribute("GiftwrapFlag", "N");
        // 4. Commit the transaction
        am.getTransaction().commit();
        // 5. Retrieve and display the trigger-assigned order id
        DBSequence id = (DBSequence)newOrder.getAttribute("OrderId");
        System.out.println("Thanks, reference number is " +
                           id.getSequenceNumber());
        Configuration.releaseRootApplicationModule(am, true);
    }
}
```

Running this example produces the results shown in Example 6–14.

Example 6–14 Results of Running TestCreateOrder.java

```
CreatedBy defaults to: Luis Popp
Thanks, reference number is 200
```

6.4.10 How to Retrieve the Row Key Identifying a Row

To retrieve a row key to identify a row, follow these basic steps (as illustrated in Example 6–15):

1. Find the view object instance.
2. Construct a key using a supplied value.
3. Find the row with this key.
4. Optionally, do something with the key of the row.

Example 6–15 shows the `main()` method finds the `Orders` view object instance and constructs a row key to find an order number. The `findByKey()` method find the `Orders` rows with the specified key. The sample then displays the key of the row, accesses the row set using the `OrderItems` view link accessor, and iterates over the rows to display the key of each `OrderItems` row.

Example 6–15 Retrieving the Row Key Identifying a Row

```
package oracle.fodemo.storefront.client;
import oracle.jbo.ApplicationModule;
import oracle.jbo.Key;
import oracle.jbo.Row;
```

```
import oracle.jbo.RowSet;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;

public class TestFindAndShowKeys {
    public static void main(String[] args) {
        String amDef = "oracle.fodemo.storefront.store.service.StoreServiceAM";
        String config = "StoreServiceAMLocal";
        ApplicationModule am =
            Configuration.createRootApplicationModule(amDef, config);
        // 1. Find the Orders view object instance
        ViewObject vo = am.findViewObject("Orders");
        // 2. Construct a key to find order number 1011
        Key orderKey = new Key(new Object[] { 1011 });
        // 3. Find the Orders row with the key
        Row[] ordersFound = vo.findByKey(orderKey, 1);
        if (ordersFound != null && ordersFound.length > 0) {
            Row order = ordersFound[0];
            // 4. Displays the key of the Orders row
            showKeyFor(order);
            // 5. Accesses row set of Orders using OrderItems view link accessor
            RowSet items = (RowSet)order.getAttribute("OrderItems");
            // 6. Iterates over the OrderItems row
            while (items.hasNext()) {
                Row itemRow = items.next();
                // 4. Displays the key of each OrderItems row
                showKeyFor(itemRow);
            }
        }
        Configuration.releaseRootApplicationModule(am, true);
    }

    private static void showKeyFor(Row r) {
        // get the key for the row passed in
        Key k = r.getKey();
        // format the key as "(val1,val2)"
        String keyAttrs = formatKeyAttributeNamesAndValues(k);
        // get the serialized string format of the key, too
        String keyStringFmt = r.getKey().toStringFormat(false);
        System.out.println("Key " + keyAttrs + " has string format " +
                           keyStringFmt);
    }
    // Build up "(val1,val2)" string for key attributes

    private static String formatKeyAttributeNamesAndValues(Key k) {
        StringBuffer sb = new StringBuffer("(");
        int attrsInKey = k.getAttributeCount();
        for (int i = 0; i < attrsInKey; i++) {
            if (i > 0)
                sb.append(",");
                sb.append(k.getAttributeValues()[i]);
        }
        sb.append(")");
        return sb.toString();
    }
}
```

Running the example produces the results shown in Example 6–16. Notice that the serialized string format of a key is a hexadecimal number that includes information in a single string that represents all the attributes in a key.

Example 6–16 Results of Running TestFindAndShowKeys.java

```
Key (1011) has string format 000100000003313031
Key (1011,1) has string format 000200000003C2020200000002C102
Key (1011,2) has string format 000200000003C2020200000002C103
```

6.4.11 How to Authenticate Test Users in the Test Client

If you have enabled ADF Security for your application and provisioned the `jazn-data.xml` file with test users, you will need to include method calls to authenticate a user before you run the test client. To authenticate a user in the test client, follow these basic steps (as illustrated in Example 6–17):

1. Create the authentication service.
2. Pass in the login credentials for a test user defined in the `jazn-data.xml` file.
3. If authentication succeeds, then test the application module.
4. Log the user out.

For details about how to run the Configure ADF Security wizard to enable ADF Security and how to create test users in JDeveloper's identity store, see Section 35.2, "ADF Security Process Overview."

Example 6–17 Passing Login Credentials in the Test Client

```
package oracle.fodemo.storefront.client;
import oracle.jbo.ApplicationModule;
import oracle.jbo.Key;
import oracle.jbo.Row;
import oracle.jbo.RowSet;
import oracle.jbo.ViewObject;
import oracle.jbo.client.Configuration;

import oracle.adf.share.ADFContext;
import oracle.adf.share.security.SecurityContext;
import oracle.adf.share.security.SecurityEnv;
import oracle.adf.share.security.authentication.JAASAuthenticationService;

import javax.security.auth.Subject;
import javax.security.jacc.PolicyContext;

public class TestAuthenticationClient {
    public static void main(String[] args) {
        String amDef = "test.TestAuthModule";
        String config = "TestAuthModuleLocal";

        // 1. Create authentication service.
        JAASAuthenticationService authService = new JAASAuthenticationService();
        try
           {
              // 2. Pass in user id and password defined in local identity store.
              authService.login("tester1", "welcome1");
           }
           catch (Exception e)
           {

           }

        // Uncomment to output authentication status
        // String userName =
```

```
                        ADFContext.getCurrent().getSecurityContext().getUserName();

            // System.out.println("*** userName : " + userName);
            // Subject subject =
                        ADFContext.getCurrent().getSecurityContext().getSubject();
            // System.out.println("Subject : " + subject);

            // 3. Test application module if authentication succeeds.
            if (ADFContext.getCurrent().getSecurityContext().isAuthenticated() {
               ApplicationModule am =
                        Configuration.createRootApplicationModule(amDef,config);
               ViewObject vo = am.findViewObject("TestView");
               // Work with your appmodule and view object here
               Configuration.releaseRootApplicationModule(am,true);

               // 4. Log out test user.
               authService.logout();

               // Uncomment to report logout success
               // boolean isAuthenticated =
                       ADFContext.getCurrent().getSecurityContext().isAuthenticated();
               // System.out.println("*** isAuthenticated : " + isAuthenticated);
            }

        }
    }
```

7

Defining Validation and Business Rules Declaratively

This chapter explains the key entity object features for implementing the most common kinds of validation rules.

This chapter includes the following sections:

- Section 7.1, "About Declarative Validation"
- Section 7.2, "Understanding the Validation Cycle"
- Section 7.3, "Adding Validation Rules to Entity Objects and Attributes"
- Section 7.4, "Using the Built-in Declarative Validation Rules"
- Section 7.5, "Using Groovy Expressions For Validation and Business Rules"
- Section 7.6, "Triggering Validation Execution"
- Section 7.7, "Creating Validation Error Messages"
- Section 7.8, "Setting the Severity Level for Validation Exceptions"
- Section 7.9, "Bulk Validation in SQL"

7.1 About Declarative Validation

The easiest way to create and manage validation rules is through *declarative validation rules*. Declarative validation rules are defined using the overview editor, and once created, are stored in the entity object's XML file. Declarative validation is different from programmatic validation (covered in Chapter 8, "Implementing Validation and Business Rules Programmatically"), which is stored in an entity object's Java file.

Oracle ADF provides built-in declarative validation rules that satisfy many of your business needs. If you have custom validation rules you want to reuse, you can code them and add them to the IDE, so that the rules are available directly from JDeveloper. Custom validation rules are an advanced topic and covered in Section 8.14, "Implementing Custom Validation Rules." You can also base validation on a Groovy expression, as described in Section 7.5, "Using Groovy Expressions For Validation and Business Rules."

When you add a validation rule, you supply an appropriate error message and can later translate it easily into other languages if needed. You can also define how validation is triggered and set the severity level.

7.1.1 Declarative Validation Use Cases and Examples

In an ADF Business Components application, most of your validation code is defined in your entity objects. Encapsulating the business logic in these shared, reusable components ensures that your business information is validated consistently in every view object or client that accesses it, and it simplifies maintenance by centralizing where the validation is stored.

Another benefit of using declarative validation (versus writing your own validation) is that the validation framework takes care of the complexities of batching validation exceptions, which frees you to concentrate on your application's specific validation rule logic.

In the model layer, ADF Model validation rules can be set for the attributes of a collection. Many of the declarative validation features available for entity objects are also available at the model layer, should your application warrant the use of model-layer validation in addition to business-layer validation. For more information, see Chapter 16, "Using Validation in the ADF Model Layer."

When you use the ADF Business Components application module data control, you do not need to use model-layer validation. Consider defining all or most of your validation rules in the centralized, reusable, and easier to maintain entity objects of your business layer. With other types of data controls, model-layer validation can be more useful.

7.1.2 Additional Functionality for Declarative Validation

You may find it helpful to understand other ADF features before you start using declarative validation. It is possible to go beyond the declarative behavior to implement more complex validation rules for your business domain layer when needed. Following are links to other functionality that may be of interest.

- Section 8.2, "Using Method Validators" explains how to use the Method validator to invoke custom validation code.
- Section 8.14, "Implementing Custom Validation Rules" details how to extend the basic set of declarative rules with custom rules of your own.

7.2 Understanding the Validation Cycle

Each entity row tracks whether or not its data is valid. When an existing entity row is retrieved from the database, the entity is assumed to be valid. When the first persistent attribute of an existing entity row is modified, or when a new entity row is created, the entity is marked invalid.

When an entity is in an invalid state, the declarative validation you have configured and the programmatic validation rules you have implemented are evaluated again before the entity can be considered valid again. You can determine whether a given entity row is valid at runtime by calling the `isValid()` method on it.

Note: Because attributes can (by default) be left blank, validations are not triggered if the attribute contains no value. For example, if a user creates a new entity row and does not enter a value for a given attribute, the validation on that attribute is not run. To force the validation to execute in this situation, set the Mandatory flag on the attribute.

7.2.1 Types of Entity Object Validation Rules

Entity object validation rules fall into two basic categories: *attribute-level* and *entity-level*.

7.2.1.1 Attribute-Level Validation Rules

Attribute-level validation rules are triggered for a particular entity object attribute when either the end user or the program code attempts to modify the attribute's value. Since you cannot determine the order in which attributes will be set, attribute-level validation rules should be used only when the success or failure of the rule depends exclusively on the candidate value of that single attribute.

The following examples are attribute-level validations:

- The value of the `OrderDate` of an order should not be a date in the past.
- The `ProductId` attribute of a product should represent an existing product.

7.2.1.2 Entity-Level Validation Rules

All other kinds of validation rules are entity-level validation rules. These are rules whose implementation requires considering two or more entity attributes, or possibly composed children entity rows, in order to determine the success or failure of the rule.

The following examples are entity-level validations:

- The value of the `OrderShippedDate` should be a date that comes after the `OrderDate`.
- The `ProductId` attribute of an order should represent an existing product.

Entity-level validation rules are triggered by calling the `validate()` method on a `Row`. This occurs when:

- You call the method explicitly on the entity object
- You call the method explicitly on a view row with an entity row part that is invalid
- A view object's iterator calls the method on the current row in the view object before allowing the current row to change
- During transaction commit, processing validates an invalid entity (in the list of pending changes) before proceeding with posting the changes to the database

As part of transaction commit processing, entity-level validation rules can fire multiple times (up to a specified limit). For more information, see Section 7.2.4, "Avoiding Infinite Validation Cycles."

7.2.2 Understanding Commit Processing and Validation

Transaction commit processing happens in three basic phases:

1. Ensure that any invalid entity rows on the pending changes list are valid.
2. Post the pending changes to the database by performing appropriate DML operations.
3. Commit the transaction.

If you have business validation logic in your entity objects that executes queries or stored procedures that depend on seeing the posted changes in the `SELECT` statements they execute, they should be coded in the `beforeCommit()` method described in Section 8.5.3, "What You May Need to Know About Row Set Access with View Accessors." This method fires after all DML statements have been applied so queries or

stored procedures invoked from that method can "see" all of the pending changes that have been saved, but not yet committed.

> **Caution:** don't use the transaction-level `postChanges()` method in web applications unless you can guarantee that the transaction will definitely be committed or rolled-back during the same HTTP request. This method exists to force the transaction to post unvalidated changes without committing them. Failure to heed this advice can lead to strange results in an environment where both application modules and database connections can be pooled and shared serially by multiple different clients.

7.2.3 Understanding the Impact of Composition on Validation Order

Because a composed child entity row is considered an integral part of its composing parent entity object, any change to composed child entity rows causes the parent entity to be marked invalid. For example, if a line item on an order were to change, the entire order would now be considered to be changed, or invalid.

Therefore, when the composing entity is validated, it causes any currently invalid composed children entities to be validated first. This behavior is recursive, drilling into deeper levels of invalid composed children if they exist.

7.2.4 Avoiding Infinite Validation Cycles

If your validation rules contain code that updates attributes of the current entity or other entities, then the act of validating the entity can cause that or other entities to become invalid. As part of the transaction commit processing phase that attempts to validate all invalid entities in the pending changes list, the transaction performs multiple passes (up to a specified limit) on the pending changes list in an attempt to reach a state where all pending entity rows are valid.

The maximum number of validation passes is specified by the transaction-level validation threshold setting. The default value of this setting is 10. You can increase the threshold count to greater than one if the entities involved contain the appropriate logic to validate themselves in the subsequent passes.

If after 10 passes, there are still invalid entities in the list, you will see the following exception:

```
JBO-28200: Validation threshold limit reached. Invalid Entities still in cache
```

This is a sign that you need to debug your validation rule code to avoid inadvertently invalidating entities in a cyclic fashion.

To change the validation threshold, use the `SetValidationThreshold()` method as shown in Example 7–1. In this example, the new threshold is 12.

Example 7–1 Changing the Validation Threshold

```
oracle.jbo.server.DBTransaction::setValidationThreshold(12)
```

7.2.5 What Happens When Validations Fail

When an entity object's validation rules throw exceptions, the exceptions are bundled and returned to the client. If the validation failures are thrown by methods you've overridden to handle events during the transaction `postChanges` processing, then

the validation failures cause the transaction to roll back any database `INSERT`, `UPDATE`, or `DELETE` statements that might have been performed already during the current `postChanges` cycle.

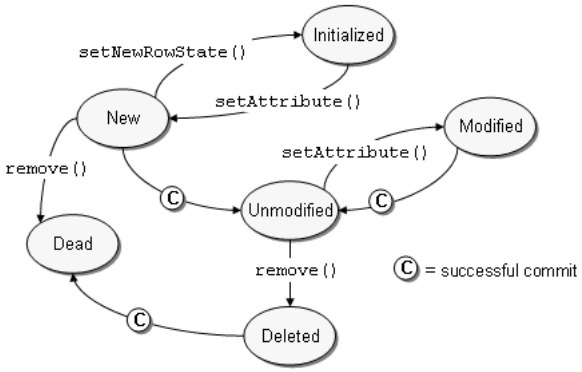
> **Note:** The bundling of exceptions is the default behavior for ADF Model-based web applications, but not for Oracle ADF Model Tester or Swing bindings. Additional configuration is required to bundle exceptions for the Oracle ADF Model Tester or Swing clients.

7.2.6 Understanding Entity Objects Row States

When an entity row is in memory, it has an entity state that reflects the logical state of the row. Figure 7–1 illustrates the different entity row states and how an entity row can transition from one state to another. When an entity row is first created, its status is `New`. You can use the `setNewRowState()` method to mark the entity as being `Initialized`, which removes it from the transaction's list of pending changes until the user sets at least one of its attributes, at which time it returns to the `New` state. This allows you to create more than one initialized row and post only those that the user modifies.

The `Unmodified` state reflects an entity that has been retrieved from the database and has not yet been modified. It is also the state that a `New` or `Modified` entity transitions to after the transaction successfully commits. During the transaction in which it is pending to be deleted, an `Unmodified` entity row transitions to the `Deleted` state. Finally, if a row that was `New` and then was removed before the transaction commits, or `Unmodified` and then successfully deleted, the row transitions to the `Dead` state.

Figure 7–1 Diagram of Entity Row States and Transitions



You can use the `getEntityState()` and `getPostState()` methods to access the current state of an entity row in your business logic code. The `getEntityState()` method returns the current state of an entity row with regard to the transaction, while the `getPostState()` method returns the current state of an entity row with regard to the database after using the `postChanges()` method to post pending changes without committing the transaction.

For example, if you start with a new row, both `getEntityState()` and `getPostState()` return `STATUS_NEW`. Then when you post the row (before commit or rollback), the row will have an entity state of `STATUS_NEW` and a post state of `STATUS_UNMODIFIED`. If you subsequently remove that row, the entity state will remain `STATUS_NEW` because for the transaction the row is still new. But the post state will be `STATUS_DEAD`.

7.2.7 Understanding Bundled Exception Mode

An application module provides a feature called bundled exception mode which allows web applications to easily present a maximal set of failed validation exceptions to the end user, instead of presenting only the *first* error that gets raised. By default, the ADF Business Components application module pool enables bundled exception mode for web applications.

You typically will not need to change this default setting. However it is important to understand that it is enabled by default since it effects how validation exceptions are thrown. Since the Java language and runtime only support throwing a single exception object, the way that bundled validation exceptions are implemented is by wrapping a set of exceptions as details of a new "parent" exception that contains them. For example, if multiple attributes in a single entity object fail attribute-level validation, then these multiple `ValidationException` objects will be wrapped in a `RowValException`. This wrapping exception contains the row key of the row that has failed validation. At transaction commit time, if multiple rows do not successfully pass the validation performed during commit, then all of the `RowValException` objects will get wrapped in an enclosing `TxnValException` object.

When writing custom error processing code, you can use the `getDetails()` method of the `JboException` base exception class to recursively process the bundled exceptions contained inside it.

Note: All the exception classes mentioned here are in the `oracle.jbo` package.

7.3 Adding Validation Rules to Entity Objects and Attributes

The process for adding a validation rule to an entity object is similar for most of the validation rules, and is done using the Add Validation Rule dialog. You can open this dialog from the overview editor by clicking the **Add** icon on the Business Rules page.

It is important to note that when you define a rule declaratively using the Add Validation Rule dialog, the rule definition you provide specifies the *valid* condition for the attribute or entity object. At runtime, the entry provided by the user is evaluated against the rule definition and an error or warning is raised if the entry fails to satisfy the specified criteria. For example, if you specify a Length validator on an attribute that requires it to be `Less Than or Equal To 12`, the validation fails if the entry is more than 12 characters, and the error or warning is raised.

7.3.1 How to Add a Validation Rule to an Entity or Attribute

To add a declarative validation rule to an entity object, use the Business Rules page of the overview editor.

Before you begin:

It may be helpful to have an understanding of the use of validation rules in entity objects and attributes. For more information, see Section 7.3, "Adding Validation Rules to Entity Objects and Attributes."

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Section 7.1.2, "Additional Functionality for Declarative Validation."

To add a validation rule:

1. In the Application Navigator, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select the object for which you want to add a validation rule, and then click the **Add** icon.
   - To add a validation rule at the entity object level, select **Entity**.
   - To add a validation rule for an attribute, expand **Attributes** and select the desired attribute.

   When you add a new validation rule, the Add Validation Rule dialog appears.
3. From the **Rule Type** dropdown list, select the desired type of validation rule.
4. Use the dialog settings to configure the new rule.

   The controls will change depending on the kind of validation rule you select.

   For more information about the different validation rules, see Section 7.4, "Using the Built-in Declarative Validation Rules."
5. You can optionally click the **Validation Execution** tab and enter criteria for the execution of the rule, such as dependent attributes and a precondition expression.

   For more information, see Section 7.6, "Triggering Validation Execution."

   Note: For Key Exists and Method entity validators, you can also use the **Validation Execution** tab to specify the validation level.
6. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails.

   For more information, see Section 7.7, "Creating Validation Error Messages."
7. Click **OK**.

7.3.2 How to View and Edit a Validation Rule on an Entity Object or Attribute

The Business Rules page of the overview editor for entity objects displays the validation rules for an entity and its attributes in a tree control. To see the validation rules that apply to the entity as a whole, expand in the **Entity** node. To see the validation rules that apply to an attribute, expand the **Attributes** node and then expand the attribute.

The validation rules that are shown on the Business Rules page of the overview editor include those that you have defined as well as database constraints, such as mandatory or precision. To open a validation rule for editing, double-click the rule or select the rule and click the **Edit** icon.

7.3.3 What Happens When You Add a Validation Rule

When you add a validation rule to an entity object, JDeveloper updates its XML component definition to include an entry describing what rule you've used and what rule properties you've entered. For example, if you add a range validation rule to the `DiscountAmount` attribute, this results in a `RangeValidationBean` entry in the XML file, as shown in Example 7–2.

Example 7–2 Range Validation Bean

```
<Attribute
```

```
  Name="DiscountAmount"
  IsNotNull="true"
  ColumnName="DISCOUNT_AMOUNT"
  . . .
  <validation:RangeValidationBean
    xmlns="http://xmlns.oracle.com/adfm/validation"
    Name="DiscountAmount_Rule_0"
    ResId="DiscountAmount_RangeError_0"
    OnAttribute="DiscountAmount"
    OperandType="LITERAL"
    Inverse="false"
    MinValue="0"
    MaxValue="40"/>
  . . .
</Attribute>
```

At runtime, the rule is enforced by the entity object based on this declarative information.

7.3.4 What You May Need to Know About Entity and Attribute Validation Rules

Declarative validation enforces both entity-level and attribute-level validation, depending on where you place the rules. Entity-level validation rules are enforced when a user tries to commit pending changes or navigates between rows. Attribute-level validation rules are enforced when the user changes the value of the related attribute.

The Unique Key validator (described in Section 7.4.1, "How to Ensure That Key Values Are Unique") can be used only at the entity level. Internally the Unique Key validator behaves like an attribute-level validator. This means that users see the validation error when they tab out of the key attribute for the key that the validator is validating. This is done because the internal cache of entities can never contain a duplicate, so it is not allowed for an attribute value to be set that would violate that. This check needs to be performed when the attribute value is being set because the cache consistency check is done during the setting of the attribute value.

> **Best Practice:** If the validity of one attribute is dependent on one or more other attributes, enforce this rule using entity validation, not attribute validation. Examples of when you would want to do this include the following:
>
> - You have a Compare validator that compares one attribute to another.
> - You have an attribute with an expression validator that examines the value in another attribute to control branching in the expression to validate the attribute differently depending on the value in this other attribute.
> - You make use of conditional execution, and your precondition expression involves an attribute other than the one that you are validating.

Entity object validators are triggered whenever the entity, as a whole, is dirty. To improve performance, you can indicate which attributes play a role in your rule and thus the rule should be triggered only if one or more of these attributes are dirty. For more information on triggering attributes, see, Section 7.6, "Triggering Validation Execution."

7.4 Using the Built-in Declarative Validation Rules

The built-in declarative validation rules can satisfy many, if not all, of your business needs. These rules are easy to implement because you don't write any code. You use the user-interface tools to choose the type of validation and how it is used.

Built-in declarative validation rules can be used to:

- Ensure that key values are unique (primary key or other unique keys)
- Determine the existence of a key value
- Make a comparison between an attribute and anything from a literal value to a SQL query
- Validate against a list of values that might be a literal list, a SQL query, or a view attribute
- Make sure that a value falls within a certain range, or that it is limited by a certain number of bytes or characters
- Validate using a regular expression or evaluate a Groovy expression
- Make sure that a value satisfies a relationship defined by an aggregate on a child entity available through an accessor
- Validate using a validation condition defined in a Java method on the entity

7.4.1 How to Ensure That Key Values Are Unique

The Unique Key validator ensures that primary key values for an entity object are always unique. The Unique Key validator can also be used for a non-primary-key attribute, as long as the attribute is defined as an alternate key. For information on how to define alternate keys, see Section 4.10.15, "How to Define Alternate Key Values."

Whenever any of the key attribute values change, this rule validates that the new key does not belong to any other entity object instance of this entity object class. (It is the business-logic tier equivalent of a unique constraint in the database.) If the key is found in one of the entity objects, a `TooManyObjectsException` is thrown. The validation check is done both in the entity cache and in the database.

There is a slight possibility that unique key validation might not be sufficient to prevent duplicate rows in the database. It is possible for two application module sessions to simultaneously attempt to create records with the same key. To prevent this from happening, create a unique index in the database for any unique constraint that you want to enforce.

Before you begin:

It may be helpful to have a general understanding of the built-in validation rules. For more information, see Section 7.4, "Using the Built-in Declarative Validation Rules."

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Section 7.1.2, "Additional Functionality for Declarative Validation."

To ensure that a key value is unique:

1. In the Application Navigator, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select the **Entity** node, and click the **Add** icon.

3. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **UniqueKey**.
4. In the **Keys** box, select the primary or alternate key.
5. You can optionally click the **Validation Execution** tab and enter criteria for the execution of the rule, such as dependent attributes and a precondition expression.

   For more information, see Section 7.6, "Triggering Validation Execution."

   > **Best Practice:** While it is possible to add a precondition for a Unique Key validator, it is not a best practice. If a Unique Key validator fails to fire, for whatever reason, the cache consistency check is still performed and an error will be returned. It is generally better to add the validator and a meaningful error message.

6. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails.

   For more information, see Section 7.7, "Creating Validation Error Messages."
7. Click **OK**.

7.4.2 What Happens When You Use a Unique Key Validator

When you use a Unique Key validator, a `<UniqueKeyValidationBean>` tag is added to the entity object's XML file. Example 7–3 shows the XML for a Unique Key validator.

Example 7–3 Unique Key Validator XML Code

```
<validation:UniqueKeyValidationBean
  Name="PersonEO_Rule_1"
  KeyName="AltKey"
  ResId="PersonEO_Rule_1">
  <validation:OnAttributes>
    <validation:Item
      Value="Email"/>
  </validation:OnAttributes>
</validation:UniqueKeyValidationBean>
```

7.4.3 How to Validate Based on a Comparison

The Compare validator performs a logical comparison between an entity attribute and a value. When you add a Compare validator, you specify an operator and something to compare with. You can compare the following:

- Literal value

  When you use a Compare validator with a literal value, the value in the attribute is compared against the specified literal value. When using this kind of comparison, it is important to consider data types and formats. The literal value must conform to the format specified by the data type of the entity attribute to which you are applying the rule. In all cases, the type corresponds to the type mapping for the entity attribute.

  For example, an attribute of column type DATE maps to the `oracle.jbo.domain.Date` class, which accepts dates and times in the same format accepted by `java.sql.TimeStamp` and `java.sql.Date`. You can use format masks to ensure that the format of the value in the attribute matches that of

the specified literal. For information about entity object attribute type mappings, see Section 4.10.1, "How to Set Database and Java Data Types for an Entity Object Attribute." For information about the expected format for a particular type, refer to the Javadoc for the type class.

- Query result

  When you use this type of validator, the SQL query is executed each time the validator is executed. The validator retrieves the first row from the query result, and it uses the value of the first column in the query (of that first row) as the value to compare. Because this query cannot have any bind variables in it, this feature should be used only when selecting one column of one row of data that does not depend on the values in the current row.

- View object attribute

  When you use this type of validator, the view object's SQL query is executed each time the validator is executed. The validator retrieves the first row from the query result, and it uses the value of the selected view object attribute from that row as the value to compare. Because you cannot associate values with the view object's named bind variables, those variables can only take on their default values. Therefore this feature should be used only for selecting an attribute of one row of data that does not depend on the values in the current row.

- View accessor attribute

  When defining the view accessor, you can assign row-specific values to the validation view object's bind variables.

- Expression

  For information on the expression option, see Section 7.5, "Using Groovy Expressions For Validation and Business Rules."

- Entity attribute

  The entity attribute option is available only for entity-level Compare validators.

Before you begin:

It may be helpful to have a general understanding of the built-in validation rules. For more information, see Section 7.4, "Using the Built-in Declarative Validation Rules."

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Section 7.1.2, "Additional Functionality for Declarative Validation."

To validate based on a comparison:

1. In the Application Navigator, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select where you want to add the validator, and click the **Add** icon.
   - To add an entity-level validator, select the **Entity** node.
   - To add an attribute-level validator, expand the **Attributes** node and select the appropriate attribute.
3. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Compare**. Note that the subordinate fields change depending on your choices.
4. Select the appropriate operator.

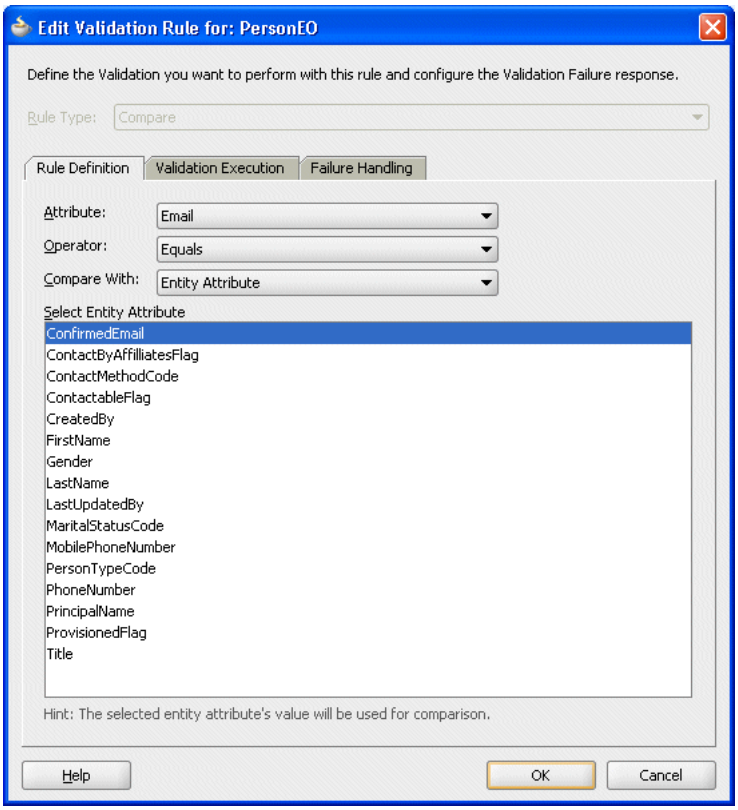
5. Select an item in the **Compare With** list, and based on your selection provide the appropriate comparison value.
6. You can optionally click the **Validation Execution** tab and enter criteria for the execution of the rule, such as dependent attributes and a precondition expression.

   For more information, see Section 7.6, "Triggering Validation Execution."
7. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails.

   For more information, see Section 7.7, "Creating Validation Error Messages."
8. Click **OK**.

Figure 7–2 shows what the dialog looks like when you use an entity-level Compare validator with an entity object attribute.

Figure 7–2 Compare Validator Using an Entity Object Attribute



7.4.4 What Happens When You Validate Based on a Comparison

When you use a Compare validator, a `<CompareValidationBean>` tag is added to an entity object's XML file. Example 7–4 shows the XML code for the `Email` attribute in the `PersonEO` entity object.

Example 7–4 Compare Validator XML Code

```
<validation:CompareValidationBean
  xmlns="http://xmlns.oracle.com/adfm/validation"
  Name="PersonEO_Rule_0"
  ResId="PersonEO_Rule_0"
  OnAttribute="Email"
```

```
OperandType="ATTR"
Inverse="false"
CompareType="EQUALTO"
CompareValue="ConfirmedEmail"/>
```

7.4.5 How to Validate Using a List of Values

The List validator compares an attribute against a list of values (LOV). When you add a List validator, you specify the type of list to choose from:

- Literal values - The validator ensures that the entity attribute is in (or not in, if specified) the list of values.
- Query result - The validator ensures that the entity attribute is in (or not in, if specified) the first column of the query's result set. The SQL query validator cannot use a bind variable, so it should be used only on a fixed, small list that you have to query from a table. All rows of the query are retrieved into memory.
- View object attribute - The validator ensures that the entity attribute is in (or not in, if specified) the view attribute. The View attribute validator cannot use a bind variable, so it should be used only on a fixed, small list that you have to query from a table. All rows of the query are retrieved into memory.
- View accessor attribute - The validator ensures that the entity attribute is in (or not in) the view accessor attribute. The view accessor is probably the most useful option, because it can take bind variables and after you've created the LOV on the user interface, a view accessor is required.

> **Best Practice:** When using a List validator, the view accessor is typically the most useful choice because you can define a view criteria on the view accessor to filter the view data when applicable; and when defining an LOV on a view attribute, you typically use a view accessor with a view criteria.

Before you begin:

It may be helpful to have a general understanding of the built-in validation rules. For more information, see Section 7.4, "Using the Built-in Declarative Validation Rules."

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Section 7.1.2, "Additional Functionality for Declarative Validation."

To validate using a list of values:

1. In the Application Navigator, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select where you want to add the validator, and click the **Add** icon.
   - To add an entity-level validator, select the **Entity** node.
   - To add an attribute-level validator, expand the **Attributes** node and select the appropriate attribute.
3. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **List**.
4. In the **Attribute** list, choose the appropriate attribute.
5. In the **Operator** field, select **In** or **NotIn**, depending on whether you want an inclusive list or exclusive.

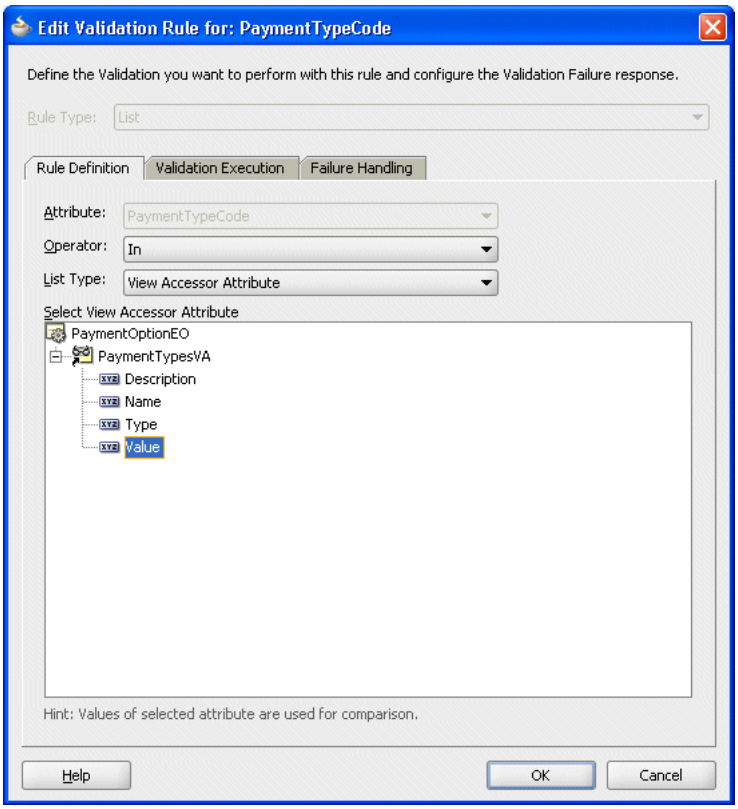
6. In the **List Type** field, select the appropriate type of list.
7. Depending on the type of list you selected, you can either enter a list of values (each value on a new line) or an SQL query, or select a view object attribute or view accessor attribute.
8. You can optionally click the **Validation Execution** tab and enter criteria for the execution of the rule, such as dependent attributes and a precondition expression.

   For more information, see Section 7.6, "Triggering Validation Execution."
9. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails.

   For more information, see Section 7.7, "Creating Validation Error Messages."
10. Click **OK**.

Figure 7–3 shows what the dialog looks like when you use a List validator with a view accessor attribute.

Figure 7–3 List Validator Using a View Accessor Attribute



7.4.6 What Happens When You Validate Using a List of Values

When you validate using a list of values, a `<ListValidationBean>` tag is added to an entity object's XML file. Example 7–5 shows the `PaymentOptionEO.PaymentTypeCode` attribute, which uses a view accessor attribute for the List validator.

Example 7–5 List Validator XML Code

```
<validation:ListValidationBean
```

```
xmlns="http://xmlns.oracle.com/adfm/validation"
Name="PaymentTypeCode_Rule_0"
ResId="PaymentTypeCode_Rule_0"
OnAttribute="PaymentTypeCode"
OperandType="VO_USAGE"
Inverse="false"
ViewAccAttrName="Value"
ViewAccName="PaymentTypesVA"/>
```

7.4.7 What You May Need to Know About the List Validator

The List validator is designed for validating an attribute against a relatively small set of values. If you select the **Query Result** or **View Object Attribute** type of list validation, keep in mind that the validator retrieves all of the rows from the query before performing an in-memory scan to validate whether the attribute value in question matches an attribute in the list. The query performed by the validator's SQL or view object query does not reference the value being validated in the `WHERE` clause of the query.

It is inefficient to use a validation rule when you need to determine whether a user-entered product code exists in a table of a large number of products. Instead, Section 8.5, "Using View Objects for Validation" explains the technique you can use to efficiently perform SQL-based validations by using a view object to perform a targeted validation query against the database. See also Section 5.12.11.2, "Using Validators to Validate Attribute Values."

Also, if the attribute you're comparing to is a key, the Key Exists validator is more efficient than validating a list of values; and if these choices need to be translatable, you should use a static view object instead of the literal choice.

7.4.8 How to Make Sure a Value Falls Within a Certain Range

The Range validator performs a logical comparison between an entity attribute and a range of values. When you add a Range validator, you specify minimum and maximum literal values. The Range validator verifies that the value of the entity attribute falls within the range (or outside the range, if specified).

If you need to dynamically calculate the minimum and maximum values, or need to reference other attributes on the entity, use the Script Expression validator and provide a Groovy expression. For more information, see Section 3.6.1, "Referencing Business Components Objects in Groovy Expressions" and Section 3.6.3, "Manipulating Business Component Attribute Values in Groovy Expressions."

Before you begin:

It may be helpful to have a general understanding of the built-in validation rules. For more information, see Section 7.4, "Using the Built-in Declarative Validation Rules."

You may also find it useful to understand additional functionality that can be added using other validation features. For more information, see Section 7.1.2, "Additional Functionality for Declarative Validation."

To validate within a certain range:

1. In the Application Navigator, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select where you want to add the validator, and click the **Add** icon.

- To add an entity-level validator, select the **Entity** node.
- To add an attribute-level validator, expand the **Attributes** node and select the appropriate attribute.

3. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Range**.
4. In the **Attribute** list, select the appropriate attribute.
5. In the **Operator** field, select **Between** or **NotBetween**.
6. In the **Minimum** and **Maximum** fields, enter appropriate values.
7. You can optionally click the **Validation Execution** tab and enter criteria for the execution of the rule, such as dependent attributes and a precondition expression.

   For more information, see Section 7.6, "Triggering Validation Execution."
8. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails.

   For more information, see Section 7.7, "Creating Validation Error Messages."
9. Click **OK**.

7.4.9 What Happens When You Use a Range Validator

When you validate against a range, a `<RangeValidationBean>` tag is added to the entity object's XML file. Example 7–6 shows the `PersonEO.CreditLimit` attribute with a minimum credit limit of zero and a maximum of 10,000.

Example 7–6 Range Validator XML Code

```
<validation:RangeValidationBean
  Name="CreditLimit_Rule_0"
  ResId="CreditLimit_Rule_0"
  OnAttribute="CreditLimit"
  OperandType="LITERAL"
  Inverse="false"
  MinValue="0"
  MaxValue="10000"/>
```

7.4.10 How to Validate Against a Number of Bytes or Characters

The Length validator validates whether the string length (in characters or bytes) of an attribute's value is less than, equal to, or greater than a specified number, or whether it lies between a pair of numbers.

Before you begin:

It may be helpful to have a general understanding of the built-in validation rules. For more information, see Section 7.4, "Using the Built-in Declarative Validation Rules."

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Section 7.1.2, "Additional Functionality for Declarative Validation."

To validate against a number of bytes or characters:

1. In the Application Navigator, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select where you want to add the validator, and click the **Add** icon.

- To add an entity-level validator, select the **Entity** node.
- To add an attribute-level validator, expand the **Attributes** node and select the appropriate attribute.

3. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Length**.
4. In the **Attribute** list, select the appropriate attribute.
5. In the **Operator** field, select how to evaluate the value.
6. In the **Comparison Type** field, select **Byte** or **Character** and enter a length.
7. You can optionally click the **Validation Execution** tab and enter criteria for the execution of the rule, such as dependent attributes and a precondition expression. For more information, see Section 7.6, "Triggering Validation Execution."
8. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see Section 7.7, "Creating Validation Error Messages."
9. Click **OK**.

7.4.11 What Happens When You Validate Against a Number of Bytes or Characters

When you validate using length, a `<LengthValidationBean>` tag is added to the entity object's XML file, as shown in Example 7–7. For example, you might have a field where the user enters a password or PIN and the application wants to validate that it is at least 6 characters long, but not longer than 10. You would use the Length validator with the Between operator and set the minimum and maximum values accordingly.

Example 7–7 Validating the Length Between Two Values

```
<validation:LengthValidationBean
  OnAttribute="pin"
  CompareType="BETWEEN"
  DataType="CHARACTER"
  MinValue="6"
  MaxValue="10"
  Inverse="false"/>
```

7.4.12 How to Validate Using a Regular Expression

The Regular Expression validator compares attribute values against a mask specified by a Java regular expression.

If you want to create expressions that can be personalized in metadata, you can use the Script Expression validator. For more information, see Section 7.5, "Using Groovy Expressions For Validation and Business Rules."

Before you begin:

It may be helpful to have a general understanding of the built-in validation rules. For more information, see Section 7.4, "Using the Built-in Declarative Validation Rules."

You may also find it useful to understand additional functionality that can be added using other validation features. For more information, see Section 7.1.2, "Additional Functionality for Declarative Validation."

To validate using a regular expression

1. In the Application Navigator, double-click the desired entity object.

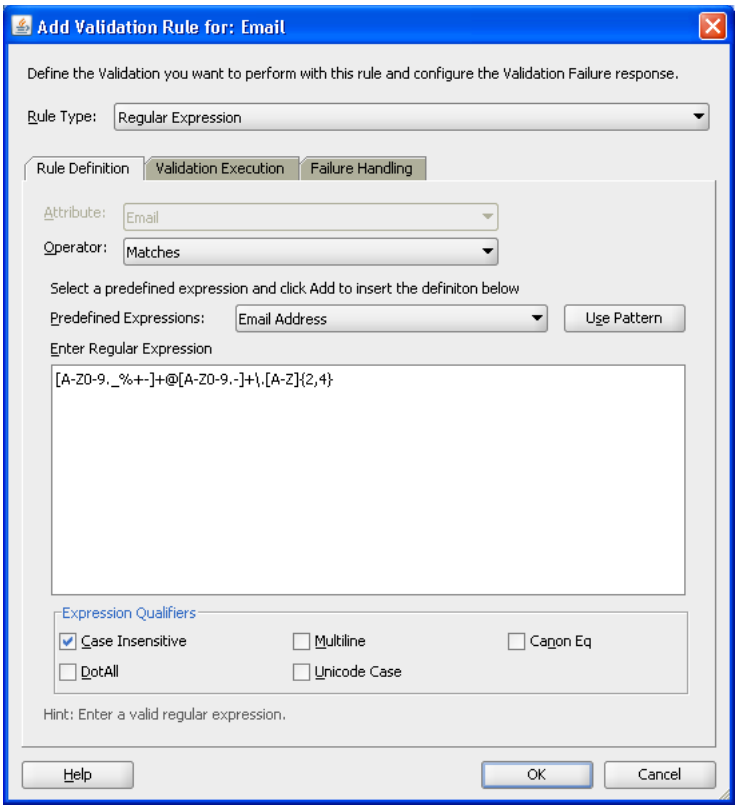
2. In the overview editor, click the **Business Rules** navigation tab, select where you want to add the validator, and click the **Add** icon.
   - To add an entity-level validator, select the **Entity** node.
   - To add an attribute-level validator, expand the **Attributes** node and select the appropriate attribute.
3. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Regular Expression**.
4. In the **Operator** field, you can select **Matches** or **Not Matches**.
5. To use a predefined expression (if available), you can select one from the dropdown list and click **Use Pattern**. Otherwise, write your own regular expression in the field provided.

   Note: You can add your own expressions to the list of predefined expressions. To add a predefined expression, add an entry in the `PredefinedRegExp.properties` file in the BC4J subdirectory of the JDeveloper system directory (for example, `C:\Documents and Settings\`*`username`*`\Application Data\JDeveloper\`*`system`*`##\o.BC4J\PredefinedRegExp.properties`).

6. You can optionally click the **Validation Execution** tab and enter criteria for the execution of the rule, such as dependent attributes and a precondition expression. For more information, see Section 7.6, "Triggering Validation Execution."
7. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see Section 7.7, "Creating Validation Error Messages."
8. Click **OK**.

Figure 7–4 shows what the dialog looks like when you select a Regular Expression validator and validate that the `Email` attribute matches a predefined **Email Address** expression.

Figure 7–4 Regular Expression Validator Matching Email Address



7.4.13 What Happens When You Validate Using a Regular Expression

When you validate using a regular expression, a `<RegExpValidationBean>` tag is added to the entity object's XML file. Example 7–8 shows an `Email` attribute that must match a regular expression.

Example 7–8 Regular Expression Validator XML Code

```
<validation:RegExpValidationBean
  Name="Email_Rule_0"
  OnAttribute="Email"
  Pattern="[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}"
  Flags="CaseInsensitive"
  Inverse="false"/>
```

7.4.14 How to Use the Average, Count, or Sum to Validate a Collection

You can use collection validation on the average, count, sum, min, or max of a collection. This validator is available only at the entity level. It is useful for validating the aggregate calculation over a collection of associated entities by way of an entity accessor to a child entity (on the many end of the association). You must select the association accessor to define the Collection validator.

Before you begin:

It may be helpful to have a general understanding of the built-in validation rules. For more information, see Section 7.4, "Using the Built-in Declarative Validation Rules."

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Section 7.1.2, "Additional Functionality for Declarative Validation."

To validate using an aggregate calculation:

1. In the Application Navigator, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select the **Entity** node and click the **Add** icon.
3. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Collection**.
4. In the **Operation** field, specify the operation (sum, average, count, min, or max) to perform on the collection for comparison.
5. Select the appropriate accessor and attribute for the validation.

   The accessor you choose must be a composition association accessor. Only accessors of this type are displayed in the dropdown list.
6. Specify the operator and the comparison type and value.
7. You can optionally click the **Validation Execution** tab and enter criteria for the execution of the rule, such as dependent attributes and a precondition expression. For more information, see Section 7.6, "Triggering Validation Execution."
8. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see Section 7.7, "Creating Validation Error Messages."
9. Click **OK**.

7.4.15 What Happens When You Use Collection Validation

When you validate using a Collection validator, a `<CollectionValidationBean>` tag is added to the entity object's XML file, as in Example 7–9.

Example 7–9 Collection Validator XML Code

```
<validation:CollectionValidationBean
    Name="OrderEO_Rule_0"
    OnAttribute="OrderTotal"
    OperandType="LITERAL"
    Inverse="false"
    CompareType="LESSTHAN"
    CompareValue="5"
    Operation="sum"/>
```

7.4.16 How to Determine Whether a Key Exists

The Key Exists validator is used to determine whether a key value (primary, foreign, or alternate key) exists.

There are a couple of benefits to using the Key Exists validator:

- The Key Exists validator has better performance because it first checks the cache and only goes to the database if necessary.
- Since the Key Exists validator uses the cache, it will find a key value that has been added in the current transaction, but not yet committed to the database. For

example, you add a new `Department` and then you want to link an `Employee` to that new department.

Before you begin:

It may be helpful to have a general understanding of the built-in validation rules. For more information, see Section 7.4, "Using the Built-in Declarative Validation Rules."

You may also find it useful to understand additional functionality that can be added using other validation features. For more information, see Section 7.1.2, "Additional Functionality for Declarative Validation."

To determine whether a value exists:

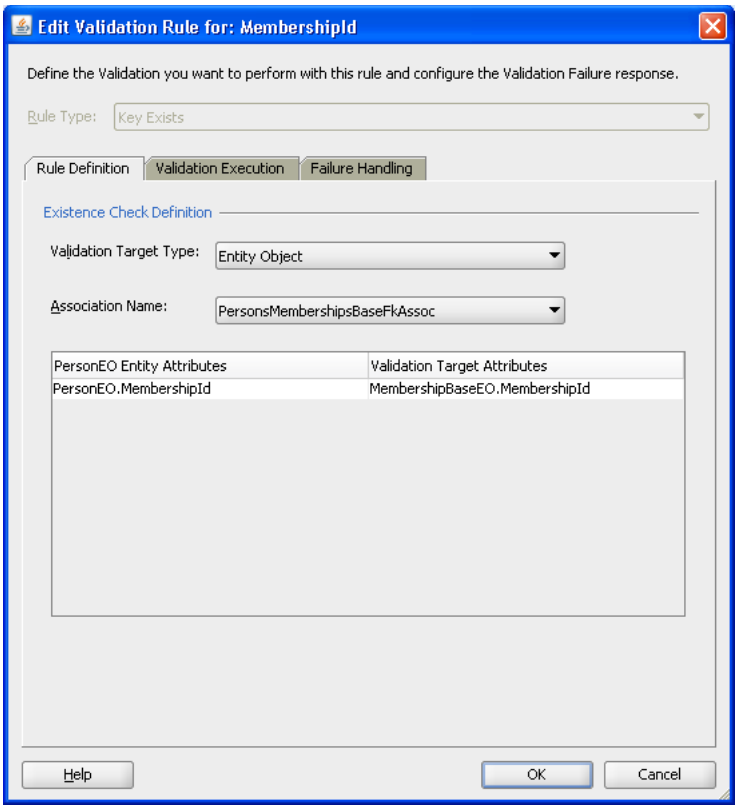
1. In the Application Navigator, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select where you want to add the validator, and click the **Add** icon.
   - To add an entity-level validator, select the **Entity** node.
   - To add an attribute-level validator, expand the **Attributes** node and select the appropriate attribute.
3. In the Add Validation Rule dialog, select **Key Exists** from the **Rule Type** list.
4. Select the type of validation target (**Entity Object**, **View Object**, or **View Accessor**).

   If you want the Key Exists validator to be used for all view objects that use this entity attribute, select **Entity Object**.
5. Depending on the validation target, you can choose either an association or a key value.

   If you are searching for an attribute that does not exist in the **Validation Target Attributes** list, it is probably not defined as a key value. To create alternate keys, see Section 4.10.15, "How to Define Alternate Key Values."
6. You can optionally click the **Validation Execution** tab and enter criteria for the execution of the rule, such as dependent attributes and the validation level (entity or transaction). For more information, see Section 7.6, "Triggering Validation Execution."
7. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see Section 7.7, "Creating Validation Error Messages."
8. Click **OK**.

Figure 7–5 shows a Key Exists validator that validates whether the `MembershipId` entered in the `PersonEO` entity object exists in the `MembershipBaseEO` entity object.

Figure 7–5 Key Exists Validator on an Entity Attribute



7.4.17 What Happens When You Use a Key Exists Validator

When you use a Key Exists validator, an `<ExistsValidationBean>` tag is created in the XML file for the entity object, as in Example 7–10.

Example 7–10 Using the Key Exists Validator With an Association

```
<validation:ExistsValidationBean
      Name="MembershipId_Rule_0"
      ResId="MembershipId_Rule_0"
      OperandType="EO"
      AssocName=
"oracle.fodemo.storefront.entities.associations.PersonsMembershipsBaseFkAssoc"/>
```

7.4.18 What You May Need to Know About Declarative Validators and View Accessors

When using declarative validators you must consider how your validation will interact with expected input. The combination of declarative validators and view accessors provides a simple yet powerful alternative to coding. But, as powerful as the combination is, you still need to consider how data composition can impact performance.

Consider a scenario where you have the following:

- A `ServiceRequestEO` entity object with `Product` and `RequestType` attributes, and a view accessor that allows it to access the `RequestTypeVO` view object
- A `RequestTypeVO` view object with a query specifying the `Product` attribute as a bind parameter

The valid list of RequestTypes varies by Product. So, to validate the RequestType attribute, you use a List validator using the view accessor.

Now lets add a set of new service requests. For the first service request (row), the List validator binds the value of the Product attribute to the view accessor and executes it. For each subsequent service request the List validator compares the new value of the Product attribute to the currently bound value.

- If the value of Product matches, the current RowSet object is retained.
- If the value of Product has changed, the new value is bound and the view accessor re-executed.

Now consider the expected composition of input data. For example, the same products could appear in the input multiple times. If you simply validate the data in the order received, you might end up with the following:

1. Dryer (initial query)
2. Washing Machine (re-execute view accessor)
3. Dish Washer (re-execute view accessor)
4. Washing Machine (re-execute view accessor)
5. Dryer (re-execute view accessor)

In this case, the validator will execute 5 queries to get 3 distinct row sets. As an alternative, you can add an ORDER BY clause to the RequestTypeVO to sort it by Product. In this case, the validator would execute the query only once each for Washing Machine and Dryer.

1. Dish Washer (initial query)
2. Dryer (re-execute view accessor)
3. Dryer
4. Washing Machine (re-execute view accessor)
5. Washing Machine

A small difference on a data set this size, but multiplied over larger data sets and many users this could easily become an issue. An ORDER BY clause is not a solution to every issue, but this example illustrates how data composition can impact performance.

7.5 Using Groovy Expressions For Validation and Business Rules

Groovy expressions are Java-like scripting code stored in the XML definition of an entity object. Because Groovy expressions are stored in XML, you can change the expression values even if you don't have access to the entity object's Java file. You can even change or specify values at runtime.

For more information about using Groovy script in your entity object business logic, see Section 3.6, "Overview of Groovy Scripting Language Support."

7.5.1 How to Reference Entity Object Methods in Groovy Validation Expressions

You can call methods on the current entity instance using the source property of the current object. The source property allows you to access to the entity instance being validated.

If the method is a non-boolean type and the method name is `getXyzAbc()` with no arguments, then you access its value as if it were a property named `XyzAbc`. For a boolean-valued property, the same holds true but the JavaBean naming pattern for the getter method changes to recognize `isXyzAbc()` instead of `getXyzAbc()`. If the method on your entity object does not match the JavaBean getter method naming pattern, or if it takes one or more arguments, then you must call it like a method using its complete name.

For example, say you have an entity object with the four methods shown in Example 7–11.

Example 7–11 Sample Entity Object Methods

```
public boolean isNewRow() {
 System.out.println("## isNewRow() accessed ##");
 return true;
}

public boolean isNewRow(int n) {
 System.out.println("## isNewRow(int n) accessed ##");
 return true;
}

public boolean testWhetherRowIsNew() {
 System.out.println("## testWhetherRowIsNew() accessed ##");
 return true;
}

public boolean testWhetherRowIsNew(int n) {
 System.out.println("## testWhetherRowIsNew(int n) accessed ##");
 return true;
}
```

Then the following Groovy validation condition would trigger them all, one of them being triggered twice, as shown in Example 7–12.

Example 7–12 Groovy Script Calling Sample Methods

```
newRow && source.newRow && source.isNewRow(5) && source.testWhetherRowIsNew() &&
source.testWhetherRowIsNew(5)
```

By running this example and forcing entity validation to occur, you would see the diagnostic output shown in Example 7–13 in the log window:

Example 7–13 Output From Sample Groovy Script

```
## isNewRow() accessed ##
## isNewRow() accessed ##
## isNewRow(int n) accessed ##
## testWhetherRowIsNew() accessed ##
## testWhetherRowIsNew(int n) accessed ##
```

Notice the slightly different syntax for the reference to a method whose name matches the JavaBeans property getter method naming pattern. Both `newRow` and `source.newRow` work to access the boolean-valued, JavaBeans getter-style method that has no arguments. But because the `testWhetherRowIsNew` method does not match the JavaBeans getter method naming pattern, and the second `isRowNew` method takes an argument, then you must call them like methods using their complete name.

7.5.2 How to Validate Using a True/False Expression

You can use a Groovy expression to return a true/false statement. The Script Expression validator *requires* that the expression either return `true` or `false`, or that it calls the `adf.error.raise/warn()` method. A common use of this feature would be to validate an attribute value, for example, to make sure that an account number is valid.

> **Note:** Using the `adf.error.raise/warn()` method (rather than simply returning `true` or `false`) allows you to define the message text to show to the user, and to associate an entity-level validator with a specific attribute. For more information, see Section 7.7.3, "How to Conditionally Raise Error Messages Using Groovy."

Before you begin:

It may be helpful to have an understanding of the use of Groovy in validation rules. For more information, see Section 7.5, "Using Groovy Expressions For Validation and Business Rules."

You may also find it useful to understand additional functionality that can be added using other validation features. For more information, see Section 7.1.2, "Additional Functionality for Declarative Validation."

To validate using a true/false expression:

1. In the Application Navigator, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select where you want to add the validator, and click the **Add** icon.
   - To add an entity-level validator, select the **Entity** node.
   - To add an attribute-level validator, expand the **Attributes** node and select the appropriate attribute.
3. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Script Expression**.
4. Enter a validation expression in the field provided.
5. You can optionally click the **Validation Execution** tab and enter criteria for the execution of the rule, such as dependent attributes and a precondition expression. For more information, see Section 7.6, "Triggering Validation Execution."
6. Click the **Failure Handling** tab and enter or select the error message that will be shown to the user if the validation rule fails. For more information, see Section 7.7, "Creating Validation Error Messages."
7. Click **OK**.

The sample code in Example 7–14 comes from the `PaymentOptionEO` entity object. The code validates account numbers based on the Luhn algorithm, a checksum formula in widespread use.

Example 7–14 Validating an Account Number Using an Expression

```
<validation:ExpressionValidationBean
  Name="AccountNumber_Rule_0"
  OperandType="EXPR"
  Inverse="false">
  <OnCondition>
```

```
    <![CDATA[PaymentTypeCode=='CC']]>
  </OnCondition>
  <MsgIds>
   <Item
    Value="PaymentOption_AccountNumber"/>
  </MsgIds>
  <TransientExpression>
   <![CDATA[
    String acctnumber = newValue;
     sumofdigits = 0;
     digit = 0;
     addend = 0;
     timesTwo = false;
     range = acctnumber.length()-1..0
      range.each {i ->
      digit = Integer.parseInt (acctnumber.substring (i, i + 1));
      if (timesTwo) {
       addend = digit * 2;
       if (addend > 9) {
        addend -= 9;
       }
      }
      else {
       addend = digit;
      }
      sumofdigits += addend;
      timesTwo = !timesTwo;
     }
     modulus = sumofdigits % 10;
     return modulus == 0;
   ]]>
  </TransientExpression>
</ExpressionValidationBean>
```

7.5.3 What Happens When You Add a Groovy Expression

When you create a Groovy expression, it is saved in the entity object's XML component. Example 7–15 shows the `RegisteredDate` attribute in the `PersonEO.xml` file. The Groovy expression is wrapped by a `<TransientExpression>` tag.

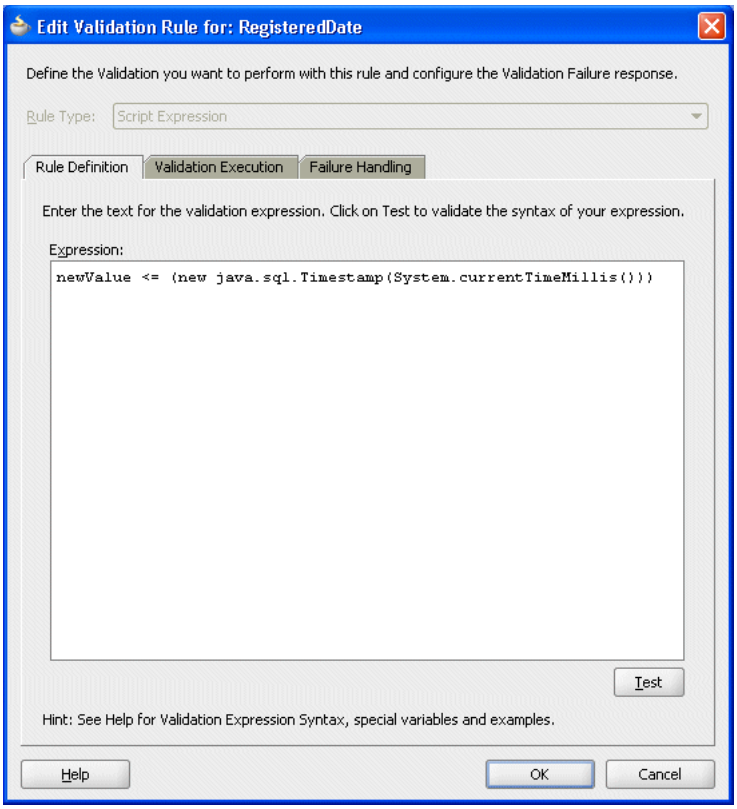
Example 7–15 XML Code for RegisteredDate Attribute on the PersonEO Entity Object

```
<Attribute
    Name="RegisteredDate"
    IsUpdateable="true"
    ColumnName="REGISTERED_DATE"
    Type="oracle.jbo.domain.Date"
    ColumnType="DATE"
    SQLType="DATE"
    TableName="PERSONS">
    <DesignTime>
      <Attr Name="_DisplaySize" Value="7"/>
    </DesignTime>
    <validation:ExpressionValidationBean
      Name="RegisteredDate_Rule_0"
      OperandType="EXPR"
      Inverse="false">
      <MsgIds>
```

```
        <Item
          Value="RegisteredDate_Rule_0"/>
      </MsgIds>
      <TransientExpression>
        <![CDATA[
newValue <= (new java.sql.Timestamp(System.currentTimeMillis()))
         ]]>
      </TransientExpression>
    </ExpressionValidationBean>
  </Attribute>
```

This tag can take one of several forms. For some Groovy expressions, the `<TransientExpression>` tag is wrapped by an `<ExpressionValidationBean>` tag as well. Figure 7–6 shows the validation expression in the Edit Validation Rule dialog.

Figure 7–6 Validation Expression for RegisteredDate Attribute on the PersonEO Entity Object



7.6 Triggering Validation Execution

JDeveloper allows you to select the attributes that trigger validation, so that validation execution happens only when one of the triggering attributes is dirty. In previous releases of JDeveloper, an entity-level validator would fire on an attribute whenever the entity as a whole was dirty. This feature is described in Section 7.6.1, "How to Specify Which Attributes Fire Validation."

JDeveloper also allows you to specify a precondition for the execution of a validator (as described in Section 7.6.3, "How to Set Preconditions for Validation") and set

transaction-level validation (described in Section 7.6.4, "How to Set Transaction-Level Validation").

7.6.1 How to Specify Which Attributes Fire Validation

When defining a validator at the entity level, you have the option of selecting one or more attributes of the entity object that, when changed, trigger execution of the validator.

> **Note:** When the validity of one attribute is dependent on the value in another attribute, the validation should be performed as entity validation, not attribute validation. You can set validation execution order on the entity level or attribute level.

If you do not specify one or more dependent attributes, the validator will fire whenever the entity is dirty. Firing execution only when required makes your application more performant.

Before you begin:

It may be helpful to have an understanding of how validation rules are triggered. For more information, see Section 7.6, "Triggering Validation Execution."

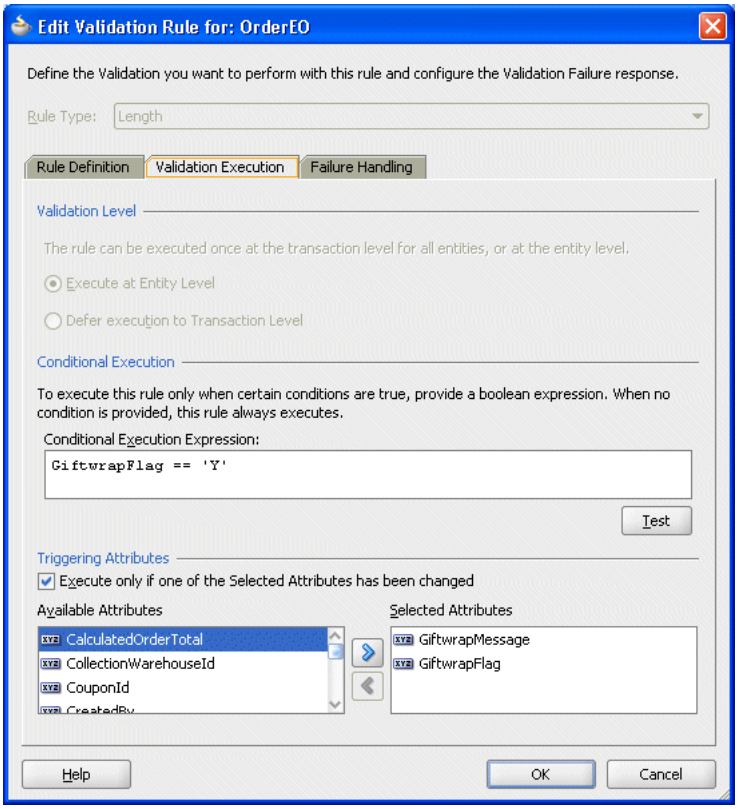
You may also find it useful to understand additional functionality that can be added using other validation features. For more information, see Section 7.1.2, "Additional Functionality for Declarative Validation."

To specify which attributes fire validation:

1. In the Application Navigator, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select the **Entity** node and click the **Edit** icon.
3. In the Edit Validation Rule dialog, click the **Validation Execution** tab.
4. Select the attributes that will fire validation.
5. Click **OK**.

For example, in the StoreFront module of the Fusion Order Demo application, the `OrderEO` entity object has an entity-level validator that constrains the length of the `GiftwrapMessage` attribute. As shown in Figure 7–7, this validator is set to be executed on the entity object only when either the `GiftwrapMessage` attribute or the `GiftwrapFlag` attribute has been changed.

Figure 7–7 Triggering attributes on the Validation Execution tab of the Edit Validation Rule dialog



7.6.2 What Happens When You Constrain Validation Execution with Triggering Attributes

When you specify triggering attributes on the **Validation Execution** tab of the Edit Validation Rule dialog, JDeveloper adds an `<OnAttributes>` tag to the validator definition in the entity object's XML file. Example 7–16 shows the XML code for the entity-level validator for the `OrderEO` entity object in the StoreFront module of the Fusion Order Demo application.

Example 7–16 OnAttributes element in XML validation code

```
<LengthValidationBean
  xmlns="http://xmlns.oracle.com/adfm/validation"
  Name="OrderEO_Rule_0"
  OnAttribute="GiftwrapMessage"
  CompareType="GREATERTHANEQUALTO"
  DataType="CHARACTER"
  CompareLength="1"
  Inverse="false"
  ResId="GiftMessage_Required_Error_0">
  <OnAttributes>
    <Item Value="GiftwrapMessage"/>
    <Item Value="GiftwrapFlag"/>
  </OnAttributes>
  <OnCondition>
    <![CDATA[GiftwrapFlag == 'Y']]>
  </OnCondition>
</LengthValidationBean>
```

7.6.3 How to Set Preconditions for Validation

The **Validation Execution** tab (on the Add/Edit Validation Rule dialog) allows you to add a Groovy expression that serves as a precondition. If you enter an expression in the **Conditional Execution Expression** box, the validator is executed only if the condition evaluates `True`.

> **Best Practice:** While it is possible to add a precondition for a Unique Key validator, it is not a best practice. If a Unique Key validator fails to fire, for whatever reason, the cache consistency check is still performed and an error will be returned. It is generally better to add the validator and a meaningful error message.

7.6.4 How to Set Transaction-Level Validation

Performing a validation during the transaction level (rather than entity level) means that the validation will be performed after all entity-level validation is performed. For this reason, it may be useful if you want to ensure that a validator is performed at the end of the process.

In addition, the Key Exists validator is more performant with bulk transactions if it is run as a transaction level validator since it will be run only once for all entities in the transaction (of the same type), rather than once per entity. This will result in improved performance if the validator has to go to the database.

Note: Transaction-level validation is only applicable to Key Exists and Method entity validators.

Before you begin:

It may be helpful to have an understanding of how validation rules are triggered. For more information, see Section 7.6, "Triggering Validation Execution."

You may also find it useful to understand additional functionality that can be added using other validation features. For more information, see Section 7.1.2, "Additional Functionality for Declarative Validation."

To specify entity-level or transaction-level validation:

1. In the Application Navigator, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select an entity-level validation rule and click the **Edit** icon.
3. In the Edit Validation Rule dialog, click the **Validation Execution** tab.
4. Select **Execute at Entity Level** or **Defer Execution to Transaction Level**.
5. Click **OK**.

7.6.5 What You May Need to Know About the Order of Validation Execution

You cannot control the order in which attributes are validated – they are always validated in the order they appear in the entity definition. You can order validations for a given attribute (or for the entity), but you cannot reorder the attributes themselves.

7.7 Creating Validation Error Messages

Validation error messages provide important information for the user: the message should convey what went wrong and how to fix it.

7.7.1 How to Create Validation Error Messages

When you create or edit a validation rule, enter text to help the user determine what caused the error.

Before you begin:

It may be helpful to have an understanding of error messages in validation rules. For more information, see Section 7.7, "Creating Validation Error Messages."

You may also find it useful to understand additional functionality that can be added using other validation features. For more information, see Section 7.1.2, "Additional Functionality for Declarative Validation."

To create validation error messages:

1. In the Application Navigator, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab, select a validation rule and click the **Edit** icon.
3. In the Edit Validation Rule dialog, click the **Failure Handling** tab.
4. In the **Message Text** field, enter your error message.

   You can also define error messages in a message bundle file. To select a previously defined error message or to define a new one in a message bundle file, click the **Select Message** icon.

   Note: The Script Expression validator allows you to enter more than one error message. This is useful if the validation script conditionally returns different error or warning messages. For more information, see Section 7.7.3, "How to Conditionally Raise Error Messages Using Groovy."

5. You can optionally include message tokens in the body of the message, and define them in the **Token Message Expressions** list.

   Figure 7–8 shows the failure message for a validation rule in the `PaymentOptionEO` entity object that contains message tokens. For more information on this feature, see Section 7.7.4, "How to Embed a Groovy Expression in an Error Message."

6. Click **OK**.

7.7.2 How to Localize Validation Messages

The error message is a translatable string and is managed in the same way as translatable UI control hints in an entity object message bundle class. To view the error message for the defined rule in the message bundle class, locate the `String` key in the message bundle that corresponds to the `ResId` property in the XML component definition entry for the validator. For example, Example 7–17 shows a message bundle where the `NAME_CANNOT_BEGIN_WITH_U` key appears with the error message for the default locale.

Example 7–17 Message Bundle Contains Validation Error Messages

```
package devguide.advanced.customerrors;
import java.util.ListResourceBundle;

public class CustomMessageBundle extends ListResourceBundle {
  private static final Object[][] sMessageStrings = new String[][] {
  // other strings here
  {"NAME_CANNOT_BEGIN_WITH_U", "The name cannot begin with the letter u!"},
  // other strings here
  };
  // etc.
}
```

Resource bundles can be created for your applications as a list resource bundle (as shown in Example 7–17), as a properties bundle, or as an XLIFF resource bundle. For more information about using translatable strings in a resource bundle, see Section 4.7, "Working with Resource Bundles."

7.7.3 How to Conditionally Raise Error Messages Using Groovy

You can use the `adf.error.raise()` and `adf.error.warn()` methods to conditionally raise one error message or another depending upon branching in the Groovy expression. For example, if an attribute value is *x*, then validate as follows, and if the validation fails, raise error messageA; whereas if the attribute value is *y*, then instead validate a different way and if validation fails, raise error messageB.

If the expression returns `false` (versus raising a specific error message using the `raise()` method), the validator calls the first error message associated with the validator.

The syntax of the `raise()` method takes one required parameter (the `msgId` to use from the message bundle), and optionally can take the `attrName` parameter. If you pass in the `AttrName`, the error is associated with that attribute even if the validation is assigned to the entity.

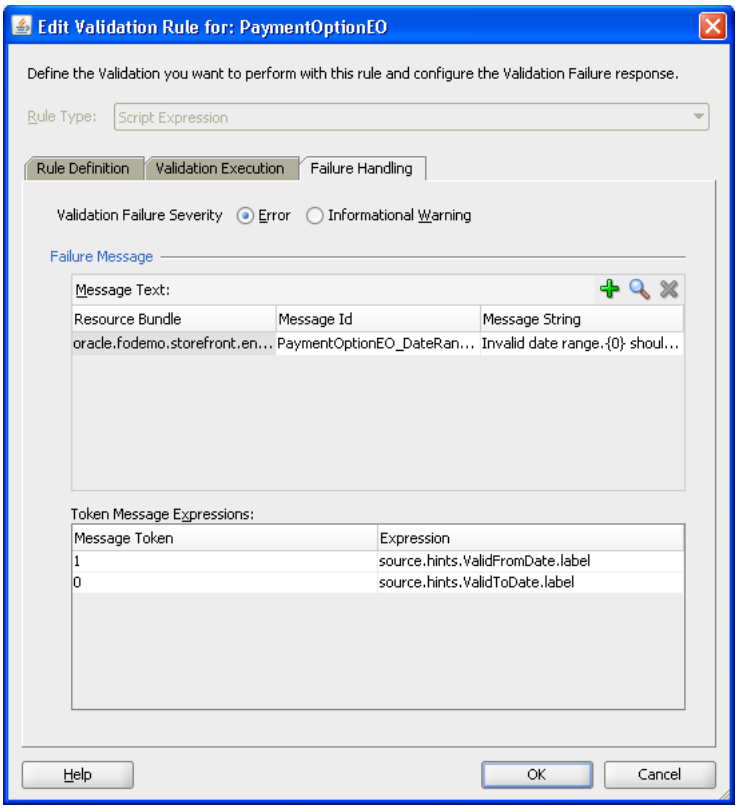
You can use either `adf.error.raise()` or `adf.error.warn()` methods, depending on whether you want to throw an exception, or whether you want processing to continue, as described in Section 7.8, "Setting the Severity Level for Validation Exceptions."

7.7.4 How to Embed a Groovy Expression in an Error Message

A validator's error message can contain embedded expressions that are resolved by the server at runtime. To access this feature, simply enter a named token delimited by curly braces (for example, `{2}` or `{errorParam}`) in the error message text where you want the result of the Groovy expression to appear.

After entering the token into the text of the error message (on the **Failure Handling** tab of the Edit Validation Rule dialog), the **Token Message Expressions** table at the bottom of the dialog displays a row that allows you to enter a Groovy expression for the token. Figure 7–8 shows the failure message for a validation rule in the `PaymentOptionEO` entity object that contains message tokens.

Figure 7–8 Using Message Tokens in a Failure Message



The expressions shown in Figure 7–8 are Groovy expressions that return the labels of the specified fields. You can also use Groovy expressions to access attribute values and other business components objects. You can use the Groovy expression `newValue` to return the entered value, as shown in the Rule validator for the `RoutingIdentifier` attribute of the `PaymentOptionEO` entity object in the StoreFront module of the Fusion Order Demo application.

The Groovy syntax to retrieve a value from a view accessor is *`accessorName`*`.currentRow.`*`AttributeName`*. For example, the Groovy expression `MyEmpAccessor.currentRow.Job` returns the value of the `Job` attribute in the current row of the `MyEmpAccessor` view accessor.

The Groovy expression can also be more complex, as in Example 7–18, which shows an expression in the error message for the List validation rule for the `OwnerTypeCode` attribute in the `AddressUsageEO` entity object.

Example 7–18 Groovy Script in the OwnerTypeCode Validation Error Message

```
def ownertypevalue = []
while ( AddressOwnerTypesVA.hasNext() ) {
AddressOwnerTypesVA.next()
  ownertypevalue.add(AddressOwnerTypesVA.currentRow.Value)
}
return ownertypevalue
```

For more information about accessing business components objects using Groovy, see Section 3.6, "Overview of Groovy Scripting Language Support."

7.8 Setting the Severity Level for Validation Exceptions

You can set the severity level for validation exceptions to two levels, Informational Warning and Error. If you set the severity level to Informational Warning, an error message will display, but processing will continue. If you set the validation level to Error, the user will not be able to proceed until you have fixed the error.

Under most circumstances you will use the Error level for validation exceptions, so this is the default setting. However, you might want to implement a Informational Warning message if the user has a certain security clearance. For example, a store manager may want to be able to make changes that would surface as an error if a clerk tried to do the same thing.

To set the severity level for validation exceptions, use the **Failure Handling** tab of the Add Validation Rule dialog.

To set the severity level of a validation exception:

1. In the Application Navigator, double-click the desired entity object.
2. On the Business Rules page, select an existing validation rule and click the **Edit** icon, or click the **Add** icon to create a new rule.
3. In the Edit/Add Validation Rule dialog, click the **Failure Handling** tab and select the option for either **Error** or **Informational Warning**.
4. Click **OK**.

7.9 Bulk Validation in SQL

To improve the performance of batch-load applications, such as data synchronization programs, the ADF framework employs bulk validation for primary keys (including alternate keys) and foreign keys.

When the Key Exists validator is configured to defer validation until the transaction commits, or when the rows are being updated or inserted through the `process`*`XXX`* methods of the ADF business components service layer, the validation cache is preloaded. This behavior uses the normal row-by-row derivation and validation logic, but uses validation logic that checks a memory cache before making queries to the database. Performance is improved by preloading the memory cache using bulk SQL operations based on the inbound data.

8

Implementing Validation and Business Rules Programmatically

This chapter explains the key entity object events and features for implementing the most common kinds of business rules.

This chapter includes the following sections:

- Section 8.1, "About Programmatic Business Rules"
- Section 8.2, "Using Method Validators"
- Section 8.3, "Assigning Programmatically Derived Attribute Values"
- Section 8.4, "Undoing Pending Changes to an Entity Using the Refresh Method"
- Section 8.5, "Using View Objects for Validation"
- Section 8.6, "Accessing Related Entity Rows Using Association Accessors"
- Section 8.7, "Referencing Information About the Authenticated User"
- Section 8.8, "Accessing Original Attribute Values"
- Section 8.9, "Storing Information About the Current User Session"
- Section 8.10, "Accessing the Current Date and Time"
- Section 8.11, "Sending Notifications Upon a Successful Commit"
- Section 8.12, "Conditionally Preventing an Entity Row from Being Removed"
- Section 8.13, "Determining Conditional Updatability for Attributes"
- Section 8.14, "Implementing Custom Validation Rules"

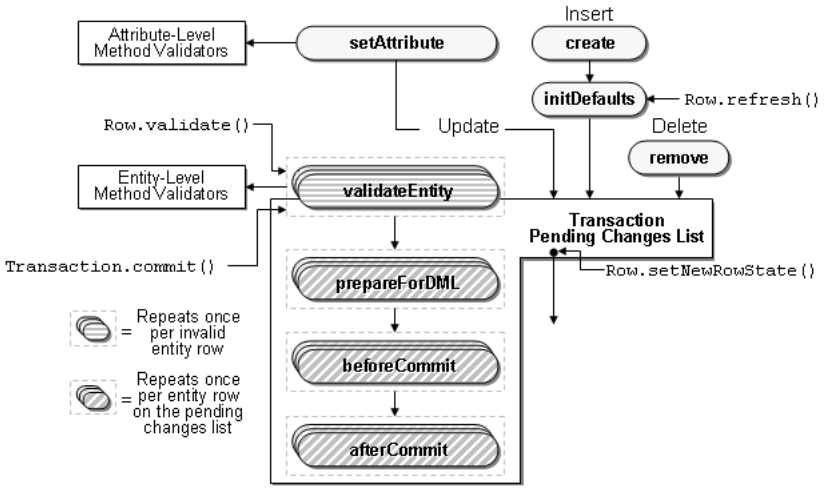
8.1 About Programmatic Business Rules

Complementing the built-in *declarative* validation features, entity objects and view objects have method validators and several events you can handle to *programmatically* implement encapsulated business logic using Java code. These concepts are illustrated in Figure 8–1.

- Attribute-level method validators trigger validation code when an attribute value changes.
- Entity-level method validators trigger validation code when an entity row is validated.
- You can override the following key methods in a custom Java class for an entity:
  - `create()`, to assign default values when a row is created

- `initDefaultExpressionAttributes()`, to assign defaults either when a row is created or when a new row is refreshed
- `remove()`, to conditionally disallow deleting
- `isAttributeUpdateable()`, to make attributes conditionally updatable
- `setAttribute()`, to trigger attribute-level method validators
- `validateEntity()`, to trigger entity-level method validators
- `prepareForDML()`, to assign attribute values before an entity row is saved
- `beforeCommit()`, to enforce rules that must consider all entity rows of a given type
- `afterCommit()`, to send notifications about a change to an entity object's state

Figure 8–1 Key Entity Objects Features and Events for Programmatic Business Logic



Note: When coding programmatic business rules, it's important to have a firm grasp of the validation cycle. For more information, see Section 7.2, "Understanding the Validation Cycle."

8.1.1 Programmatic Business Rules Use Cases and Examples

While much of your validation can be implemented using basic declarative behavior, you can implement more complex business rules for your business domain layer when needed, using the Method validator to invoke custom validation code.

Some examples of when you might want to use programmatic business rules include:

- Eagerly defaulting an attribute value from a database sequence
- Assigning values derived from complex calculations
- Undoing pending changes to an entity object
- Accessing and storing information about the current user session
- Determining conditional updatability for attributes

8.1.2 Additional Functionality for Programmatic Business Rules

You may find it helpful to understand other ADF features before you start using programmatic validation. Following are links to other functionality that may be of interest.

- Before implementing business rules programmatically, you should see if the declarative validation can handle the needs of your application. For more information, see Chapter 7, "Defining Validation and Business Rules Declaratively."
- You can use resource bundles to provide localizable validation error messages, as described in Section 4.7, "Working with Resource Bundles."
- Section 4.10.9, "How to Synchronize with Trigger-Assigned Values," explains how to use the `DBSequence` type for primary key attributes whose values need to be populated by a database sequence at *commit* time.
- For information about security features in Oracle Fusion Web Applications, including how to access information about the authenticated user, see Chapter 35, "Enabling ADF Security in a Fusion Web Application."
- For information about using Groovy script in your entity object business logic, see Section 3.6, "Overview of Groovy Scripting Language Support."

8.2 Using Method Validators

Method validators are the primary way of supplementing declarative validation rules and Groovy-scripted expressions using your own Java code. Method validators trigger Java code that you write in your own validation methods at the appropriate time during the entity object validation cycle. There are many types of validation you can code with a method validator, either on an attribute or on an entity as a whole.

You can add any number of attribute-level or entity-level method validators, provided they each trigger a distinct method name in your code. All validation method names must begin with the word `validate`; however, following that rule you are free to name them in any way that most clearly identifies the functionality. For an attribute-level validator, the method must take a single argument of the same type as the entity attribute. For an entity-level validator, the method takes no arguments. The method must also be public, and must return a boolean value. Validation will fail if the method returns `false`.

> **Note:** Although it is important to be aware of these rules, when you use JDeveloper to create method validators, JDeveloper creates the correct interface for the class.

At runtime, the Method validator passes an entity attribute to a method implemented in your entity object class.

In Example 8–1, the method accepts strings that start with a capital letter and throws an exception on null values, empty strings, and strings that do not start with a capital letter.

Example 8–1 Method That Validates If the First Letter Is a Capital

```
public boolean validateIsCapped(String text)
{
  if (text != null &&
```

```
      text.length() != 0 &&
      text[0] >= 'A' &&
      text[0] <= 'Z')
  {
    return true;
  }
}
```

8.2.1 How to Create an Attribute-Level Method Validator

The use of method validators is a programmatic approach that supplements your declarative validation rules.

Before you begin:

It may be helpful to have an understanding of what method validators are. For more information, see Section 8.2, "Using Method Validators."

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Section 8.1.2, "Additional Functionality for Programmatic Business Rules."

To create an attribute-level Method validator:

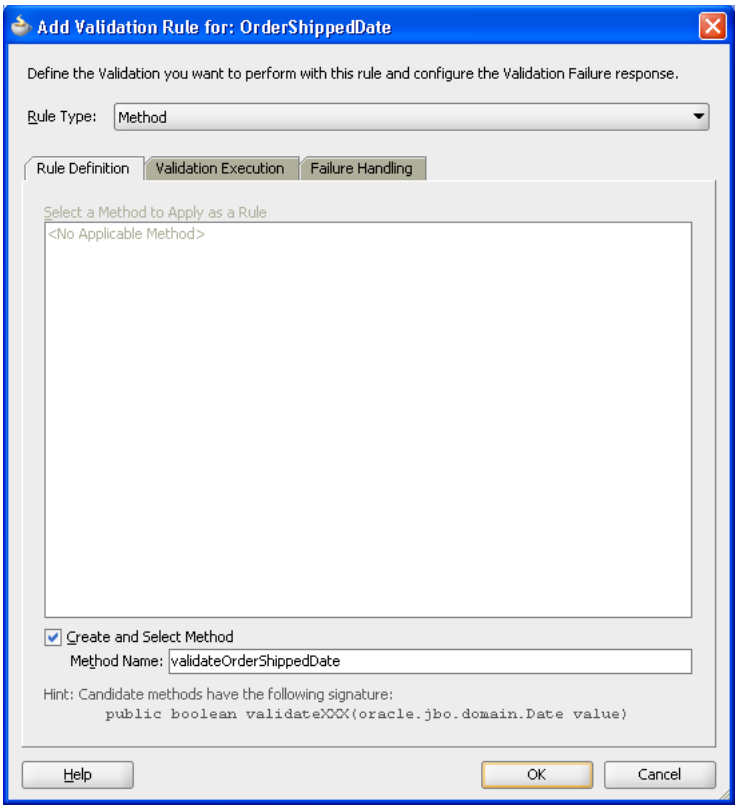
1. In the Application Navigator, double-click the desired entity object.
2. In the overview editor, click the **Java** navigation tab.

   The Java page shows the Java generation options that are currently enabled for the entity object. If your entity object does not yet have a custom entity object class, then you must generate one before you can add a Method validator. To generate the custom Java class, click the **Edit** icon, then select **Generate Entity Object Class**, and click **OK** to generate the `*.java` file.
3. Click the **Business Rules** navigation tab, and then expand the **Attributes** node, and select the attribute that you want to validate.
4. Click the **New** icon to add a validation rule.
5. Select **Method** from the **Rule Type** dropdown list.

   The Add Validation Rule dialog displays the expected method signature for an attribute-level validation method. You have two choices:

   - If you already have a method in your entity object's custom Java class of the appropriate signature, it will appear in the list and you can select it after deselecting the **Create and Select Method** checkbox.
   - If you leave the **Create and Select Method** checkbox selected (see Figure 8–2), you can enter any method name in the **Method Name** box that begins with the word `validate`. When you click **OK,** JDeveloper adds the method to your entity object's custom Java class with the appropriate signature.
6. Finally, supply the text of the error message for the default locale that the end user should see if this validation rule fails.

Figure 8–2 Adding an Attribute-Level Method Validator



8.2.2 What Happens When You Create an Attribute-Level Method Validator

When you add a new method validator, JDeveloper updates the XML component definition to reflect the new validation rule. If you asked to have the method created, the method is added to the entity object's custom Java class. Example 8–2 illustrates a simple attribute-level validation rule that ensures that the `OrderShippedDate` of an order is a date in the current month. Notice that the method accepts an argument of the same type as the corresponding attribute, and that its conditional logic is based on the value of this incoming parameter. When the attribute validator fires, the attribute value has not yet been set to the new value in question, so calling the `getOrderShippedDate()` method inside the attribute validator for the `OrderShippedDate` attribute would return the attribute's *current* value, rather than the *candidate* value that the client is attempting to set.

Example 8–2 Simple Attribute-Level Method Validator

```
public boolean validateOrderShippedDate(Date  data) {
  if (data != null && data.compareTo(getFirstDayOfCurrentMonth()) <= 0) {
    return false;
  }
  return true;
}
```

> **Note:** The return value of the `compareTo()` method is zero (`0`) if the two dates are equal, negative one (`-1`) if the first date is less than the second, or positive one (`1`) if the first date is greater than the second.

8.2.3 How to Create an Entity-Level Method Validator

Entity-level method validators are similar to attribute-level method validators, except that they have a broader scope: the entire entity rather than a single attribute.

Before you begin:

It may be helpful to have an understanding of what method validators are. For more information, see Section 8.2, "Using Method Validators."

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Section 8.1.2, "Additional Functionality for Programmatic Business Rules."

To create an entity-level method validator:

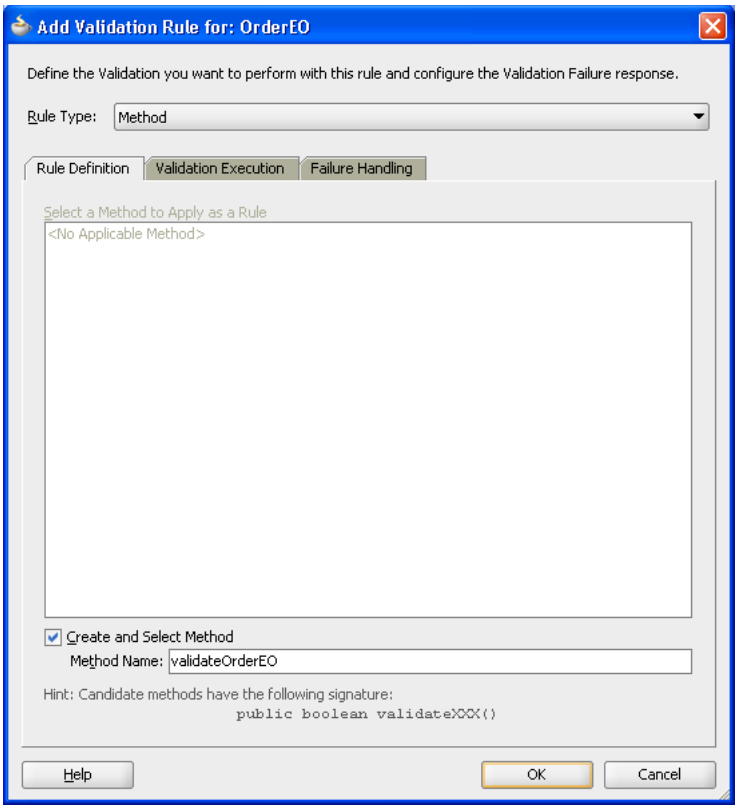
1. In the Application Navigator, double-click the desired entity object.
2. In the overview editor, click the **Java** navigation tab.

   The Java page shows the Java generation options that are currently enabled for the entity object. If your entity object does not yet have a custom entity object class, then you must generate one before you can add a Method validator. To generate the custom Java class, click the **Edit** icon, then select **Generate Entity Object Class**, and click **OK** to generate the `*.java` file.
3. Click the **Business Rules** navigation tab, and then select the **Entity** node.
4. Click the **New** icon to add a validation rule.
5. Select **Method** from the **Rule Type** dropdown list.

   The Add Validation Rule dialog displays the expected method signature for an entity-level validation method. You have two choices:

   - If you already have a method in your entity object's custom Java class of the appropriate signature, it will appear in the list and you can select it after deselecting the **Create and Select Method** checkbox.
   - If you leave the **Create and Select Method** checkbox selected (see Figure 8–3), you can enter any method name in the **Method Name** box that begins with the word `validate`. When you click **OK,** JDeveloper adds the method to your entity object's custom Java class with the appropriate signature.
6. Finally, supply the text of the error message for the default locale that the end user should see if this validation rule fails.

Figure 8–3 Adding an Entity-Level Method Validator



8.2.4 What Happens When You Create an Entity-Level Method Validator

When you add a new method validator, JDeveloper updates the XML component definition to reflect the new validation rule. If you asked to have the method created, the method is added to the entity object's custom Java class. Example 8–3 illustrates a simple entity-level validation rule that ensures that the `OrderShippedDate` of an order comes after the `OrderDate`.

Example 8–3 Simple Entity-Level Method Validator

```
public boolean validateOrderShippedDateAfterOrderDate() {
  Date orderShippedDate = getOrderShippedDate();
  Date orderDate  = getOrderDate();
  if (orderShippedDate != null && orderShippedDate.compareTo(orderDate) < 0) {
    return false;
  }
  return true;
}
```

8.2.5 What You May Need to Know About Translating Validation Rule Error Messages

Like the locale-specific UI control hints for entity object attributes, the validation rule error messages are added to the entity object's component message bundle file. These entries in the message bundle represent the strings for the default locale for your application. To provide translated versions of the validation error messages, follow the same steps as for translating the UI control hints, as described in Section 4.7, "Working with Resource Bundles."

8.3 Assigning Programmatically Derived Attribute Values

When declarative defaulting falls short of your needs, you can perform programmatic defaulting in your entity object:

- When an entity row is first created
- When the entity row is first created or when refreshed to null values
- When the entity row is saved to the database
- When an entity attribute value is set

8.3.1 How to Provide Default Values for New Rows at Create Time

The `create()` method provides the entity object event you can handle to initialize default values the first time an entity row is created. Example 8–4 shows the overridden create method of the `OrderEO` entity object in the StoreFront module of the Fusion Order Demo. It calls an attribute setter methods to populate the `OrderDate` attribute in a new order entity row.

You can also define default values using a Groovy expression. For more information, see Section 4.10.6, "How to Define a Static Default Value."

Example 8–4 Programmatically Defaulting Attribute Values for New Rows

```
// In OrderEOImpl.java in Fusion Order Demo
protected void create(AttributeList nameValuePair) {
   super.create(nameValuePair);
   this.setOrderDate(new Date());
}
```

> **Note:** Calling the `setAttribute()` method inside the overridden `create()` method does not mark the new row as being changed by the user. These programmatically assigned defaults behave like declaratively assigned defaults.

8.3.1.1 Choosing Between create() and initDefaultExpressionAttributes() Methods

You should override the `initDefaultExpressionAttributes()` method for programmatic defaulting logic that you want to fire both when the row is first created, and when it might be refreshed back to initialized status.

If an entity row has `New` status and you call the `refresh()` method on it, then the entity row is returned to an `Initialized` status if you do not supply either the `REFRESH_REMOVE_NEW_ROWS` or `REFRESH_FORGET_NEW_ROWS` flag. As part of this process, the entity object's `initDefaultExpressionAttributes()` method is invoked, but not its `create()` method again.

8.3.1.2 Eagerly Defaulting an Attribute Value from a Database Sequence

Section 4.10.9, "How to Synchronize with Trigger-Assigned Values," explains how to use the `DBSequence` type for primary key attributes whose values need to be populated by a database sequence at *commit* time. Sometimes you may want to eagerly allocate a sequence number at entity row creation time so that the user can see its value and so that this value does not change when the data is saved. To accomplish this, use the `SequenceImpl` helper class in the `oracle.jbo.server` package in an overridden `create()` method as shown in Example 8–5. It shows code from the

custom Java class of the `WarehouseEO` entity object in the StoreFront module of the Fusion Order Demo. After calling `super.create()`, it creates a new instance of the `SequenceImpl` object, passing the sequence name and the current transaction object. Then it calls the `setWarehouseId()` attribute setter method with the return value from `SequenceImpl`'s `getSequenceNumber()` method.

> **Note:** For a metadata-driven alternative to this approach, see Section 4.14.5, "Assigning the Primary Key Value Using an Oracle Sequence."

Example 8–5 Eagerly Defaulting an Attribute's Value from a Sequence at Create Time

```
// In WarehouseEOImpl.java
import oracle.jbo.server.SequenceImpl;
// Default WarehouseId value from WAREHOUSE_SEQ sequence at entity row create time
protected void create(AttributeList attributeList) {
    super.create(attributeList);
    SequenceImpl sequence = new SequenceImpl("WAREHOUSE_SEQ",getDBTransaction());
    setWarehouseId(sequence.getSequenceNumber());
}
```

8.3.2 How to Assign Derived Values Before Saving

If you want to assign programmatic defaults for entity object attribute values before a row is saved, override the `prepareForDML()` method and call the appropriate attribute setter methods to populate the derived attribute values. To perform the assignment only during `INSERT`, `UPDATE`, or `DELETE`, you can compare the value of the `operation` parameter passed to this method against the integer constants `DML_INSERT`, `DML_UPDATE`, `DML_DELETE` respectively.

Example 8–6 shows an overridden `prepareForDML()` method that assigns derived values.

Example 8–6 Assigning Derived Values Before Saving Using PrepareForDML

```
protected void prepareForDML(int operation, TransactionEvent e) {
  super.prepareForDML(operation, e);
  //Populate GL Date
  if (operation == DML_INSERT) {
    if (this.getGlDate() == null) {
      String glDateDefaultOption =
        (String)this.getInvoiceOption().getAttribute("DefaultGlDateBasis");
      if ("I".equals(glDateDefaultOption)) {
        setAttribute(GLDATE, this.getInvoiceDate());
      } else {
        setAttribute(GLDATE, this.getCurrentDBDate());
      }
    }
  }

  //Populate Exchange Rate and Base Amount if null
  if ((operation == DML_INSERT) || (operation == DML_UPDATE)) {
    BigDecimal defaultExchangeRate = new BigDecimal(1.5);
    if ("Y".equals(this.getInvoiceOption().getAttribute("UseForeignCurTrx"))) {
      if (!(this.getInvoiceCurrencyCode().equals(
                          this.getLedger().getAttribute("CurrencyCode")))) {
        if (this.getExchangeDate() == null) {
```

```
          setAttribute(EXCHANGEDATE, this.getInvoiceDate());
        }
        if (this.getExchangeRateType() == null) {
          String defaultConvRateType =
            (String)this.getInvoiceOption().getAttribute("DefaultConvRateType");
          if (defaultConvRateType != null) {
            setAttribute(EXCHANGERATETYPE, defaultConvRateType);
          } else {
            setAttribute(EXCHANGERATETYPE, "User");
          }
        }
        if (this.getExchangeRate() == null) {
          setAttribute(EXCHANGERATE, defaultExchangeRate);
        }
        if ((this.getExchangeRate() != null) &&
            (this.getInvoiceAmount() != null)) {
          setAttribute(INVAMOUNTFUNCCURR,
                       (this.getExchangeRate().multiply(this.getInvoiceAmount())));
        }
      } else {
        setAttribute(EXCHANGEDATE, null);
        setAttribute(EXCHANGERATETYPE, null);
        setAttribute(EXCHANGERATE, null);
        setAttribute(INVAMOUNTFUNCCURR, null);
      }
    }
  }
}
```

8.3.3 How to Assign Derived Values When an Attribute Value Is Set

To assign derived attribute values whenever another attribute's value is set, add code to the latter attribute's setter method. Example 8–7 shows the setter method for an `AssignedTo` attribute in an entity object.

Example 8–7 Setting the Assigned Date Whenever the AssignedTo Attribute Changes

```
public void setAssignedTo(Number value) {
  setAttributeInternal(ASSIGNEDTO, value);
  setAssignedDate(getCurrentDateWithTime());
}
```

After the call to `setAttributeInternal()` to set the value of the `AssignedTo` attribute, it uses the setter method for the `AssignedDate` attribute to set its value to the current date and time.

> **Note:** It is safe to add custom code to the generated attribute getter and setter methods as shown here. When JDeveloper modifies code in your class, it intelligently leaves your custom code in place.

8.4 Undoing Pending Changes to an Entity Using the Refresh Method

You can use the `refresh(int flag)` method on a row to refresh any pending changes it might have. The behavior of the `refresh()` method depends on the flag that you pass as a parameter. The three key flag values that control its behavior are the following constants in the `Row` interface:

- `REFRESH_WITH_DB_FORGET_CHANGES` forgets modifications made to the row in the current transaction, and the row's data is refreshed from the database. The latest data from the database replaces data in the row regardless of whether the row was modified or not.
- `REFRESH_WITH_DB_ONLY_IF_UNCHANGED` works just like `REFRESH_WITH_DB_FORGET_CHANGES`, but for *unmodified* rows. If a row was already modified by this transaction, the row is not refreshed.
- `REFRESH_UNDO_CHANGES` works the same as `REFRESH_WITH_DB_FORGET_CHANGES` for *unmodified* rows. For a modified row, this mode refreshes the row with attribute values at the beginning of this transaction. The row remains in a modified state if it had been previously posted but not committed in the current transaction prior to performing the refresh operation.

8.4.1 How to Control What Happens to New Rows During a Refresh

By default, any entity rows with `New` status that you `refresh()` are reverted back to blank rows in the `Initialized` state. Declarative defaults are reset, as well as programmatic defaults coded in the `initDefaultExpressionAttributes()` method, but the entity object's `create()` method is not invoked during this blanking-out process.

You can change this default behavior by combining one of the flags in Section 8.4 with one of the following two flags (using the bitwise-`OR` operator):

- `REFRESH_REMOVE_NEW_ROWS`, new rows are removed during refresh.
- `REFRESH_FORGET_NEW_ROWS`, new rows are marked `Dead`.

8.4.2 How to Cascade Refresh to Composed Children Entity Rows

You can cause a `refresh()` operation to cascade to composed child entity rows by combining the `REFRESH_CONTAINEES` flag (using the bitwise-`OR` operator) with any of the valid flag combinations described in Section 8.4 and Section 8.4.1. This causes the entity to invoke `refresh()` using the same mode on any composed child entities it contains.

8.5 Using View Objects for Validation

When your business logic requires performing SQL queries, the natural choice is to use a view object to perform that task. Keep in mind that the SQL statements you execute for validation will "see" pending changes in the entity cache only if they are entity-based view objects. Read-only view objects will only retrieve data that has been posted to the database.

8.5.1 How to Use View Accessors for Validation Against View Objects

Since entity objects are designed to be reused in any application scenario, they should not depend *directly* on a view object instance in any specific application module's data model. Doing so would prevent them from being reused in other application modules, which is highly undesirable.

Instead, you should use a view accessor to validate against a view object. For more information, see Section 10.4.1, "How to Create a View Accessor for an Entity Object or View Object."

Using a view accessor, your validation code can access the view object and set bind variables, as shown in Example 8–8.

Example 8–8 Using a Validation View Object in a Method Validator

```
// Sample entity-level validation method
public boolean validateSomethingUsingViewAccessor() {
  RowSet rs = getMyValidationVO();
  rs.setNamedBindParameter("Name1", value1);
  rs.setNamedBindParameter("Name2", value2);
  rs.executeQuery();
  if ( /* some condition */) {
  /*
   * code here returns true if the validation succeeds
   */
  }
  return false;
}
```

> **Best Practice:** Any time you access a row set programmatically, you should consider creating a secondary iterator for the row set. This ensures that you will not disturb the current row set of the default row set iterator that may be utilized when your expose your view objects as data controls to the user interface project. You can call `createRowSetIterator()` on the row set you are working with to create a secondary named row set iterator. When you are through with programmatic iteration, your code should call `closeRowSetIterator()` on the row set to remove the secondary iterator from memory.

As the sample code suggests, view objects used for validation typically have one or more named bind variables in them. In this example, the bind variables are set using the `setNamedBindParameter()` method. However, you can also set these variables declaratively in JDeveloper using Groovy expressions in the view accessor definition page.

Depending on the kind of data your view object retrieves, the "`/* some condition */`" expression in the example will look different. For example, if your view object's SQL query is selecting a `COUNT()` or some other aggregate, the condition will typically use the `rs.first()` method to access the first row, then use the `getAttribute()` method to access the attribute value to see what the database returned for the count.

If the validation succeeds or fails based on whether the query has returned zero or one row, the condition might simply test whether `rs.first()` returns `null` or not. If `rs.first()` returns `null`, there is no "first" row. In other words, the query retrieved no rows. In other cases, you may be iterating over one or more query results retrieved by the view object to determine whether the validation succeeds or fails.

8.5.2 How to Validate Conditions Related to All Entities of a Given Type

The `beforeCommit()` method is invoked on *each* entity row in the pending changes list after the changes have been posted to the database, but before they are committed. This can be a useful method in which to execute view object-based validations that must assert some rule over all entity rows of a given type.

> **Note:** You can also do this declaratively using a transaction-level validator (see Section 7.6.4, "How to Set Transaction-Level Validation").

If your `beforeCommit()` logic can throw a `ValidationException`, you must set the `jbo.txn.handleafterpostexc` property to `true` in your configuration to have the framework automatically handle rolling back the in-memory state of the other entity objects that may have already successfully posted to the database (but not yet been committed) during the current commit cycle.

> **Note:** The example in this section refers to the `ViewObjectValidations` project of the `AdvancedViewObjectExamples` application workspace in the `StandaloneExamples` module of the Fusion Order Demo application.

For example, consider the overridden `beforeCommit()` shown in Example 8–9. In this example, there are three view objects based on polymorphic entity objects (`Persons`, `Staff`, and `Supplier`), with the `PersonTypeCode` attribute as the discriminator. The `PersonsImpl.java` file has an overridden `beforeCommit()` method that calls a validation method. The validation method uses the fourth view object, `PersonsValidator`, to make sure that the principal name is unique across each person type. For example, there is a `PrincipalName` of `SKING` for the `Staff` view object, but there cannot be another `SKING` in this or the other person types.

Example 8–9 Overriding beforeCommit() to Validate All Entities of a Given Type

```
// from the PersonsImpl.java file
. . .
@Override
public void beforeCommit(TransactionEvent transactionEvent) throws
ValidationException {
  String principalName = getPrincipalName();
  if (!validatePrincipalNameIsUniqueUsingViewAccessor(principalName)) {
    throw new ValidationException("Principal Name must be unique across person
types");
  }
    super.beforeCommit(transactionEvent);
}

public boolean validatePrincipalNameIsUniqueUsingViewAccessor(String
principalName) {
RowSet rs = getPersonsValidatorVO();
rs.setNamedWhereClauseParam("principalName", principalName);
rs.setRangeSize(-1);
rs.executeQuery();
Row[] validatorRows = rs.getAllRowsInRange();
if (validatorRows.length > 1)
  // more than one row has the same princpalName
{
    return false;
}
rs.closeRowSetIterator();
return true;
}
```

8.5.3 What You May Need to Know About Row Set Access with View Accessors

If your entity object or view object business logic iterates over its own view accessor row set, and that view accessor is not also used by a model-defined List of Values, then there is no need to use a secondary row set iterator. For example, if an entity object has a view accessor named `AirportValidationVA` for a view object that takes one named bind parameter, it can iterate its own view accessor row set using either Groovy script or Java. Example 8–10 show a Groovy script that iterates over a view accessor row set.

Example 8–10 Using a View Accessor in Groovy Script

```
AirportValidationVA.setNamedWhereClauseParam("VarTla",newValue)
AirportValidationVA.executeQuery();
return AirportValidationVA.first() != null;
```

Example 8–11 shows a Java method validator that iterates over a view accessor row set.

Example 8–11 Using a View Accessor in a Method Validator

```
public boolean validateJob(String job) {
   getAirportValidationVA().setNamedWhereClauseParam("VarTla",job);
   getAirportValidationVA().executeQuery();
   return getAirportValidationVA().first() != null;
}
```

8.6 Accessing Related Entity Rows Using Association Accessors

To access information from related entity objects, you use an association accessor method in your entity object's custom Java class. By calling the accessor method, you can easily access any related entity row — or set of entity rows — depending on the cardinality of the association.

8.6.1 How to Access Related Entity Rows

You can use an association accessor to access related entity rows. Example 8–12 shows code from the `ControllingPostingOrder` project in the `AdvancedEntityExamples` module of the Fusion Order Demo that shows the overridden `postChanges()` method in the `ProductsBase` entity object's custom Java class. It uses the `getSupplier()` association accessor to retrieve the related supplier for the product.

Example 8–12 Accessing a Parent Entity Row In a Create Method

```
// In ProducstBaseImpl.java in the ControllingPostingOrder project
// of the Fusion Order Demo Advanced Entity Examples
@Override
public void postChanges(TransactionEvent transactionEvent) {
  /* If current entity is new or modified */
  if (getPostState() == STATUS_NEW || getPostState() == STATUS_MODIFIED) {
    /* Get the associated supplier for the product */
    SuppliersImpl supplier = getSupplier();
    /* If there is an associated product */
    if (supplier != null) {
      /* And if its post-status is NEW */
```

```
      if (supplier.getPostState() == STATUS_NEW) {
        /* Post the supplier first, before posting this entity */
        supplier.postChanges(transactionEvent);
      }
    }
  }
  super.postChanges(transactionEvent);
}
```

8.6.2 How to Access Related Entity Row Sets

If the cardinality of the association is such that multiple rows are returned, you can use the association accessor to return sets of entity rows.

Example 8–13 illustrates the code for the overridden `postChanges()` method in the `Suppliers` entity object's custom Java class. It shows the use of the `getProductsBase()` association accessor to retrieve the `RowSet` object of `ProductsBase` rows in order to update the `SupplierId` attribute in each row using the `setSupplierId()` association accessor.

Example 8–13 Accessing a Related Entity Row Set Using an Association Accessor

```
// In SuppliersImpl.java in the ControllingPostingOrder project
// of the Fusion Order Demo Advanced Entity Examples
RowSet newProductsBeforePost = null;
@Override
public void postChanges(TransactionEvent transactionEvent) {
  /* Only update references if Supplier is new */
  if (getPostState() == STATUS_NEW) {
   /*
    * Get a rowset of products related to this new supplier before calling super
    */
    newProductsBeforePost = (RowSet)getProductsBase();
  }
  super.postChanges(transactionEvent);
}

...

protected void refreshFKInNewContainees() {
  if (newProductsBeforePost != null) {
    Number newSupplierId = getSupplierId().getSequenceNumber();
    /*
     * Process the rowset of suppliers that referenced the new product prior
     * to posting, and update their ProdId attribute to reflect the refreshed
     * ProdId value that was assigned by a database sequence during posting.
     */
    while (newProductsBeforePost.hasNext()){
      ProductsBaseImpl product = (ProductsBaseImpl)newProductsBeforePost.next();
      product.setSupplierId(newSupplierId);
    }
    closeNewProductRowSet();
  }
}
```

8.7 Referencing Information About the Authenticated User

If you have run the Configure ADF Security wizard on your application to enable the ADF authentication servlet to support user login and logout, the `oracle.jbo.server.SessionImpl` object provides methods you can use to get information about the name of the authenticated user and about the roles of which they are a member. This is the implementation class for the `oracle.jbo.Session` interface that clients can access.

For information about how to access information about the authenticated user, see Section 35.11.3.3, "How to Determine the Current User Name, Enterprise Name, or Enterprise ID" and Section 35.11.3.4, "How to Determine Membership of a Java EE Security Role".

For more information about security features in Oracle Fusion Web Applications, read Chapter 35, "Enabling ADF Security in a Fusion Web Application."

8.8 Accessing Original Attribute Values

If an entity attribute's value has been changed in the current transaction, when you call the attribute getter method for it you will get the pending changed value. Sometimes you want to get the original value before it was changed. Using the `getPostedAttribute()` method, your entity object business logic can consult the original value for any attribute as it was read from the database before the entity row was modified. This method takes the attribute *index* as an argument, so pass the appropriate generated attribute index enums that JDeveloper maintains for you.

8.9 Storing Information About the Current User Session

If you need to store information related to the current user session in a way that entity object business logic can reference, you can use the user data hash table provided by the `Session` object.

8.9.1 How to Store Information About the Current User Session

When a new user accesses an application module for the first time, the `prepareSession()` method is called. As shown in Example 8–14, the application module overrides `prepareSession()` to retrieve information about the authenticated user by calling a `retrieveUserInfoForAuthenticatedUser()` method on the view object instance. Then, it calls the `setUserIdIntoUserDataHashtable()` helper method to save the user's numerical ID into the user data hash table.

Example 8–14 Overriding prepareSession() to Query User Information

```
// In the application module
protected void prepareSession(Session session) {
  super.prepareSession(session);
  /*
   * Query the correct row in the VO based on the currently logged-in
   * user, using a custom method on the view object component
   */
  getLoggedInUser().retrieveUserInfoForAuthenticatedUser();
  setUserIdIntoUserDataHashtable();
}
```

Example 8–15 shows the code for the view object's `retrieveUserInfoForAuthenticatedUser()` method. It sets its own `EmailAddress` bind variable to the name of the authenticated user from the session and then calls `executeQuery()` to retrieve the additional user information from the `USERS` table.

Example 8–15 Accessing Authenticated User Name to Retrieve Additional User Details

```
// In the view object's custom Java class
public void retrieveUserInfoForAuthenticatedUser() {
  SessionImpl session = (SessionImpl)getDBTransaction().getSession();
  setEmailAddress(session.getUserPrincipalName());
  executeQuery();
  first();
}
```

One of the pieces of information about the authenticated user that the view object retrieves is the user's numerical ID number, which that method returns as its result. For example, the user `sking` has the numeric `UserId` of `300`.

Example 8–16 shows the `setUserIdIntoUserDataHashtable()` helper method — used by the `prepareSession()` code in Example 8–14 — that stores this numerical user ID in the user data hash table, using the key provided by the string constant `CURRENT_USER_ID`.

Example 8–16 Setting Information into the UserData Hashtable for Access By Entity Objects

```
// In the application module
private void setUserIdIntoUserDataHashtable() {
  Integer userid = getUserIdForLoggedInUser();
  Hashtable userdata = getDBTransaction().getSession().getUserData();
  userdata.put(CURRENT_USER_ID, userid);
}
```

The corresponding entity objects in this example can have an overridden `create()` method that references this numerical user ID using a helper method like the one in Example 8–17 to set the `CreatedBy` attribute programmatically to the value of the currently authenticated user's numerical user ID.

Example 8–17 Referencing the Current User ID in a Helper Method

```
protected Number getCurrentUserId() {
  Hashtable userdata = getDBTransaction().getSession().getUserData();
  Integer userId = (Integer)userdata.get(CURRENT_USER_ID);
  return userdata != null ? Utils.intToNumber(userId):null;
}
```

8.9.2 How to Use Groovy to Access Information About the Current User Session

The top-level `adf` object allows you access to objects that the framework makes available to Groovy script. The `adf.userSession` object returns a reference to the ADF Business Components user session, which you can use to reference values in the `userData` hash map that is part of the session.

Example 8–18 shows the Groovy script you would use to reference a `userData` hash map key named `MyKey`.

Example 8–18 Accessing the Current User Session Using Groovy Script

```
adf.userSession.userData.MyKey
```

8.10 Accessing the Current Date and Time

You might find it useful to reference the current date and time in your entity object business logic. You can reference the current date or current date and time using the following Groovy script expressions:

- `adf.currentDate` — returns the current date (time truncated)
- `adf.currentDateTime` — returns the current date and time

For more information about using Groovy script in your entity object business logic, see Section 3.6, "Overview of Groovy Scripting Language Support."

8.11 Sending Notifications Upon a Successful Commit

The `afterCommit()` method is invoked on *each* entity row that was in the pending changes list and got successfully saved to the database. You can use this method to send a notification on a commit.

A better way to send notifications upon a successful commit is by declaring a business event. For more information on how to create a business event, see Section 4.12, "Creating Business Events."

8.12 Conditionally Preventing an Entity Row from Being Removed

Before an entity row is removed, the `remove()` method is invoked on an entity row. You can throw a `JboException` in the `remove()` method to prevent a row from being removed if the appropriate conditions are not met.

For example, you can add a test in the `remove()` method that determines the state of the entity object and allows the removal only if it is a new record. Example 8–19 demonstrates this technique.

> **Note:** This example is in the `AddressesImpl.java` file in the `ConditionalDelete` project of the `DevGuideExamples` workspace in the `StandaloneExamples` module of the Fusion Order Demo application.

Example 8–19 Overriding the remove() Method to Verify Entity Status Before Removal

```
// In the Addresses entity object custom Java class
private boolean isDeleteAllowed() {
    byte s = this.getEntityState();
  return s==STATUS_NEW;
}

/**
 * Add entity remove logic in this method.
 */
public void remove() {
  if (isDeleteAllowed())
    super.remove();
  else
```

```
      throw new JboException("Delete not allowed in this view");
}
```

Note: The entity object offers declarative prevention of deleting a master entity row that has existing, composed children rows. You configure this option on the Relationship page of the overview editor for the association.

8.13 Determining Conditional Updatability for Attributes

You can override the `isAttributeUpdateable()` method in your entity object class to programmatically determine whether a given attribute is updatable or not at runtime based on appropriate conditions.

Example 8–20 shows how an entity object can override the `isAttributeUpdateable()` method to enforce that its `PersonTypeCode` attribute is updatable only if the current authenticated user is a staff member. Notice that when the entity object fires this method, it passes in the integer attribute index whose updatability is being considered.

You can implement conditional updatability logic for a particular attribute inside an `if` or `switch` statement based on the attribute index. Here `PERSONTYPECODE` is referencing the integer attribute index enums that JDeveloper maintains in your entity object custom Java class.

Example 8–20 Conditionally Determining an Attribute's Updatability at Runtime

```
// In the entity object custom Java class
public boolean isAttributeUpdateable(int index) {
  if (index == PERSONTYPECODE) {
    if (!currentUserIsStaffMember()) {
      return super.isAttributeUpdateable(index);
    }
    return CUSTOMER_TYPE.equals(getPersonTypeCode()) ? false : true;
  }
  return super.isAttributeUpdateable(index);
}
```

Note: Entity-based view objects inherit this conditional updatability as they do everything else encapsulated in your entity objects. Should you need to implement this type of conditional updatability logic in a way that is specific to a transient view object attribute, or to enforce some condition that involves data from multiple entity objects participating in the view object, you can override this same method in a view object's view row class to achieve the desired result.

8.14 Implementing Custom Validation Rules

ADF Business Components comes with a base set of built-in declarative validation rules that you can use. However, a powerful feature of the validator architecture for entity objects is that you can create your own custom validation rules. When you notice that you or your team are writing the same kind of validation code over and over, you can build a custom validation rule class that captures this common validation "pattern" in a parameterized way.

After you've defined a custom validation rule class, you can register it in JDeveloper so that it is as simple to use as any of the built-in rules. In fact, you can even bundle your custom validation rule with a custom UI panel that JDeveloper leverages to facilitate developers' using and configuring the parameters your validation rule might require.

8.14.1 How to Create a Custom Validation Rule

To write a custom validation rule for entity objects, you need a Java class that implements the `JboValidatorInterface` in the `oracle.jbo.rules` package. You can create a skeleton class from the New Gallery.

Before you begin:

It may be helpful to have an understanding of custom validation rules. For more information, see Section 8.14, "Implementing Custom Validation Rules."

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Section 8.1.2, "Additional Functionality for Programmatic Business Rules."

To create a custom validator:

1. In the Application Navigator, right-click the project where you want to create the validator, and choose **New** from the context menu.
2. In the New Gallery, expand **Business Tier**, click **ADF Business Components**, select **Validation Rule** and click **OK**.

As shown in Example 8–21, `JBOValidatorInterface` contains one main `validate()` method, and a getter and setter method for a `Description` property.

Example 8–21 All Validation Rules Must Implement the JboValidatorInterface

```
package oracle.jbo.rules;
public interface JboValidatorInterface {
  void validate(JboValidatorContext valCtx) { }
  java.lang.String getDescription() { }
  void setDescription(String description) { }
}
```

If the behavior of your validation rule will be parameterized to make it more flexible, then add additional bean properties to your validator class for each parameter. For example, the code in Example 8–22 implements a custom validation rule called `DateMustComeAfterRule` which validates that one date attribute must come after another date attribute. To allow the developer using the rule to configure the names of the date attributes to use as the initial and later dates for validation, this class defines two properties `initialDateAttrName` and `laterDateAttrName`.

Example 8–22 shows the code that implements the custom validation rule. It extends the `AbstractValidator` to inherit support for working with the entity object's custom message bundle, where JDeveloper saves the validation error message when a developer uses the rule in an entity object.

The `validate()` method of the validation rule gets invoked at runtime whenever the rule class should perform its functionality. The code performs the following basic steps:

1. Ensures validator is correctly attached at the entity level.
2. Gets the entity row being validated.

3. Gets the values of the initial and later date attributes.
4. Validate that initial date is before later date.
5. Throws an exception if the validation fails.

Example 8–22 Custom DateMustComeAfterRule

```
// NOTE: package and imports omitted
public class DateMustComeAfterRule extends AbstractValidator
       implements JboValidatorInterface {
  /**
   * This method is invoked by the framework when the validator should do its job
   */
  public void validate(JboValidatorContext valCtx) {
    // 1. If validator is correctly attached at the entity level...
    if (validatorAttachedAtEntityLevel(valCtx)) {
      // 2. Get the entity row being validated
      EntityImpl eo = (EntityImpl)valCtx.getSource();
      // 3. Get the values of the initial and later date attributes
      Date initialDate = (Date) eo.getAttribute(getInitialDateAttrName());
      Date laterDate = (Date) eo.getAttribute(getLaterDateAttrName());
      // 4. Validate that initial date is before later date
      if (!validateValue(initialDate,laterDate)) {
        // 5. Throw the validation exception
        RulesBeanUtils.raiseException(getErrorMessageClass(),
                                      getErrorMsgId(),
                                      valCtx.getSource(),
                                      valCtx.getSourceType(),
                                      valCtx.getSourceFullName(),
                                      valCtx.getAttributeDef(),
                                      valCtx.getNewValue(),
                                      null, null);
      }
    }
    else {
      throw new RuntimeException("Rule must be at entity level");
    }
  }
  /**
   * Validate that the initialDate comes before the laterDate.
   */
  private boolean validateValue(Date initialDate, Date laterDate) {
    return (initialDate == null) || (laterDate == null) ||
    (initialDate.compareTo(laterDate) < 0);
  }
  /**
   * Return true if validator is attached to entity object
   * level at runtime.
   */
  private boolean validatorAttachedAtEntityLevel(JboValidatorContext ctx) {
    return ctx.getOldValue() instanceof EntityImpl;
  }
  // NOTE: Getter/Setter Methods omitted
  private String description;
  private String initialDateAttrName;
  private String laterDateAttrName;
}
```

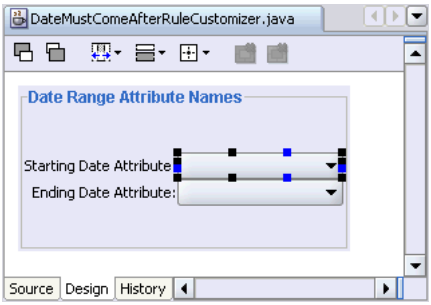
For easier reuse of your custom validation rules, you would typically package them into a JAR file for reference by applications that make use of the rules.

8.14.2 Adding a Design Time Bean Customizer for Your Rule

Since a validation rule class is a bean, you can implement a standard JavaBean customizer class to improve the design time experience of setting the bean properties. In the example of the `DateMustComeAfter` rule (in Example 8–22), the two properties that the developers must configure are the `initialDateAttrName` and `laterDateAttrName` properties.

Figure 8–4 illustrates using JDeveloper's visual designer for Swing to create a `DateMustComeAfterRuleCustomizer` using a `JPanel` with a titled border containing two `JLabel` prompts and two `JComboBox` controls for the dropdown lists. The code in the class populates the dropdown lists with the names of the Date-valued attributes of the current entity object being edited in the IDE. This will allow a developer who adds a `DateMustComeAfterRule` validation to their entity object to easily pick which date attributes should be used for the starting and ending dates for validation.

Figure 8–4 Using JDeveloper's Swing Visual Designer to Create a Validation Rule Customizer



To associate a customizer with your `DateMustComeAfterRule` Java Bean, you follow the standard practice of creating a `BeanInfo` class. As shown in Example 8–23, the `DateMustComeAfterRuleBeanInfo` returns a `BeanDescriptor` that associates the customizer class with the `DateMustComeAfter` bean class.

You would typically package your customizer class and this bean info in a separate JAR file for design-time-only use.

Example 8–23 BeanInfo to Associate a Customizer with a Custom Validation Rule

```
package oracle.fodemo...frameworkExt.rules;
import java.beans.BeanDescriptor;
import java.beans.SimpleBeanInfo;
public class DateMustComeAfterRuleBeanInfo extends SimpleBeanInfo {
  public BeanDescriptor getBeanDescriptor() {
    return new BeanDescriptor(DateMustComeAfterRule.class,
                              DateMustComeAfterRuleCustomizer.class);
  }
}
```

8.14.3 How to Register and Using a Custom Rule in JDeveloper

After you've created a custom validation rule, you can add it to the project or application level in the JDeveloper IDE so that other developers can use the rule declaratively.

8.14.3.1 Registering a Custom Validator at the Project Level

When you register a custom validation rule at the project level, you can use it within the project.

Before you begin:

It may be helpful to have an understanding of custom validation rules. For more information, see Section 8.14, "Implementing Custom Validation Rules."

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Section 8.1.2, "Additional Functionality for Programmatic Business Rules."

To register a custom validation rule, you must have already created it, as described in Section 8.14.1, "How to Create a Custom Validation Rule."

To register a custom validation rule in a project containing entity objects:

1. In the Application Navigator, right-click the desired project, and choose **Project Properties** from the context menu.
2. In the Project Properties dialog, expand **Business Components**, and select **Registered Rules**.
3. On the Registered Rules page, click **Add**.
4. In the Register Validation Rule dialog, browse to find the validation rule you have created (such as the one created in Section 8.14.1, "How to Create a Custom Validation Rule"), and click **OK**.

8.14.3.2 Registering a Custom Validator at the IDE Level

When you register a custom validation rule at the IDE level for JDeveloper, you can use it in other projects as well as your current project.

Before you begin:

It may be helpful to have an understanding of custom validation rules. For more information, see Section 8.14, "Implementing Custom Validation Rules."

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Section 8.1.2, "Additional Functionality for Programmatic Business Rules."

To register a custom validation rule, you must have already created it, as described in Section 8.14.1, "How to Create a Custom Validation Rule."

To register a custom validator at the IDE level:

1. From the **Tools** menu, choose **Preferences**.
2. In the Preferences dialog, expand **Business Components**, and select **Register Rules**.
3. From the Register Rules page, you can add a one or more validation rules.

   When adding a validation rule, provide the fully-qualified name of the validation rule class, and supply a validation rule name that will appear in JDeveloper's list of available validators.

9

Implementing Business Services with Application Modules

This chapter describes how to create application modules to encapsulate a data model using view objects. This chapter also describes how to combine business service methods with that data model to implement a complete business service.

This chapter includes the following sections:

- Section 9.1, "About Application Modules"
- Section 9.2, "Creating and Modifying an Application Module"
- Section 9.3, "Configuring Your Application Module Database Connection"
- Section 9.4, "Defining Nested Application Modules"
- Section 9.5, "Creating an Application Module Diagram for Your Business Service"
- Section 9.6, "Supporting Multipage Units of Work"
- Section 9.7, "Customizing an Application Module with Service Methods"
- Section 9.8, "Customizing Application Module Message Strings"
- Section 9.9, "Publishing Custom Service Methods to UI Clients"
- Section 9.10, "Working Programmatically with an Application Module's Client Interface"
- Section 9.11, "Overriding Built-in Framework Methods"
- Section 9.12, "Calling a Web Service from an Application Module"

9.1 About Application Modules

An *application module* is an ADF Business Components component that encapsulates the business service methods and data model for a logical unit of work related to an end-user task.

In the early phases of application development, architects and designers often use UML use case techniques to create a high-level description of the application's planned end-user functionalities. Each high-level, end-user use case identified during the design phase typically depends on:

- The domain business objects involved. To answer the question, "What core business data is relevant to the use case?"

- The user-oriented view of business data required. To answer the questions, "What subset of columns, what filtered set of rows, sorted in what way, grouped in what way, is needed to support the use case?"

The identified domain objects involved in each use case help you identify the required entity objects from your business domain layer. The user-oriented view of the required business data helps to define the right SQL queries captured as view objects and to retrieve the data in the exact way needed by the end user. For best performance, this includes retrieving the minimum required details necessary to support the use case. In addition to leveraging view object queries to shape the data, you've learned how to use view links to set up natural master-detail hierarchies in your data model to match exactly the kind of end-user experience you want to offer the user to accomplish the use case.

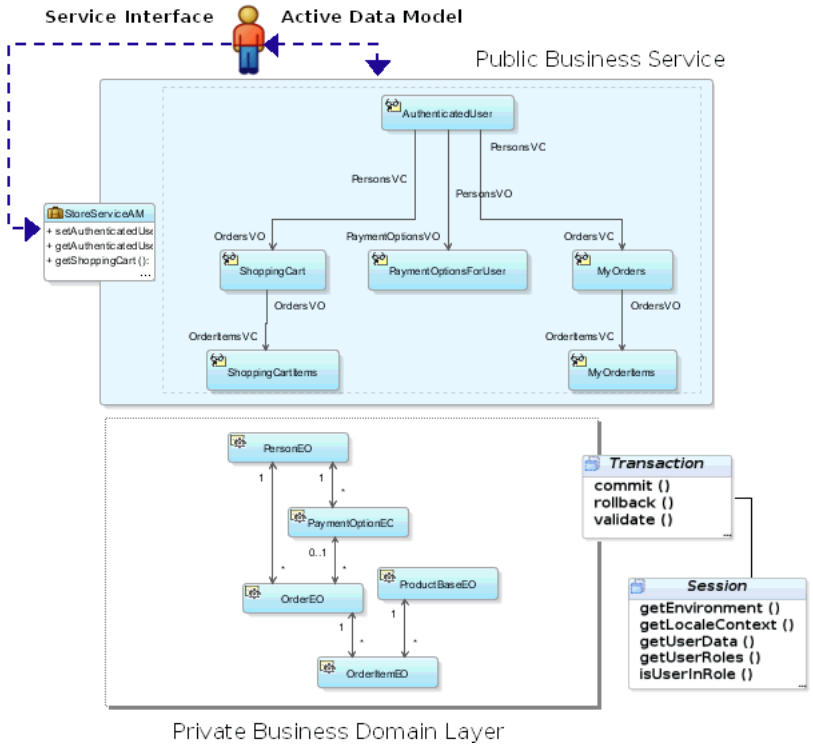
The application module is the "work unit" container that includes instances of the reusable view objects required for the use case in question, related through metadata to the underlying entity objects in your reusable business domain layer whose information the use case is presenting or modifying.

9.1.1 Application Module Use Cases and Examples

This chapter illustrates the following concepts illustrated in Figure 9–1, and more:

- You use instances of view objects in an application module to define its data model.
- You write service methods to encapsulate task-level business logic.
- You expose selected methods on the client interface for UI clients to call.
- You expose selected methods on the service interface for programmatic use in application integration scenarios.
- You use application modules from a pool during a logical transaction that can span multiple web pages.
- Your application module works with a `Transaction` object that acquires a database connection and coordinates saving or rolling back changes made to entity objects.
- The related `Session` object provides runtime information about the current application user.

Figure 9–1 Application Module Is a Business Service Component Encapsulating a Unit of Work



9.1.2 Additional Functionality for Application Modules

You may find it helpful to understand other ADF features before you start working with application modules. Following are links to other functionality that may be of interest.

- For details about creating shared application modules, see Chapter 10, "Sharing Application Module View Instances."
- For details about reusing ADF Business Components using external services, see Chapter 11, "Integrating Service-Enabled Application Modules."
- For details about how the Data Controls panel exposes the application module to UI developers, see Chapter 13, "Using ADF Model in a Fusion Web Application."
- For details about configuring application module instances to improve runtime performance, see Chapter 43, "Application State Management" and Chapter 44, "Tuning Application Module Pools and Connection Pools."
- For a quick reference to the most common code that you will typically write, use, and override in your custom application module classes, see Appendix D, "Most Commonly Used ADF Business Components Methods."
- For API documentation related to the `oracle.jbo` package, see the following Javadoc reference document:
  - *Oracle Fusion Middleware Java API Reference for Oracle ADF Model*

9.2 Creating and Modifying an Application Module

In a large application, you typically create one application module to support each coarse-grained end-user task. In a smaller-sized application, you may decide that

creating a single application module is adequate to handle the needs of the complete set of application functionality. Section 9.4, "Defining Nested Application Modules" provides additional guidance on this subject.

9.2.1 How to Create an Application Module

Any view object you create is a reusable component that can be used in the context of one or more application modules to perform the query it encapsulates in the context of that application module's transaction. The set of view objects used by an application module defines its *data model*, in other words, the set of data that a client can display and manipulate through a user interface.

To create an application module, use the Create Application Module wizard, which is available in the New Gallery.

Before you begin:

It may be helpful to have an understanding of application modules. For more information, see Section 9.2, "Creating and Modifying an Application Module."

Complete this task:

Create the desired view objects as described in Section 5.2.1, "How to Create an Entity-Based View Object," and Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object."

To create an application module:

1. In the Application Navigator, right-click the project node in which you want to create the application module and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Application Module**, and click **OK**.
3. In the Create Application Module wizard, on the Name page, provide a package name and an application module name. Click **Next**.

   Note: In Fusion web applications, the reserved words `data`, `bindings`, `security`, and `adfContext` must not be used to name your application module. Also, avoid using the "_" (underscore) at the beginning of the name. For more information, see Section 9.2.5, "How to Edit an Existing Application Module."

4. On the Data Model page, include instances of the view objects you have previously defined and edit the view object instance names to be exactly what you want clients to see. Then click **Next**.
5. On the Java page, you can optionally generate the Java files that allow you to programmatically customize the behavior of the application module or to expose methods on the application module's client interface that can be called by clients. To generate an XML-only application module component, leave the fields unselected and click **Finish**.

   Initially, you may want to generate only the application module XML definition component. After you complete the wizard, you can subsequently use the overview editor to generate the application module class files when you require programmatic access. For details about the programmatic use of the application module, see Section 9.7, "Customizing an Application Module with Service Methods."

For more step by step details, see Section 9.2.3.2, "Adding Master-Detail View Object Instances to an Application Module."

9.2.2 What Happens When You Create an Application Module

When you create an application module, JDeveloper creates the XML component definition file that represents its declarative settings and saves it in the directory that corresponds to the name of its package. For example, given an application module named `StoreServiceAM` in the `storefront.model` package, the XML file created will be `./storefront/model/StoreServiceAM.xml` under the project's source path. This XML file contains the information needed at runtime to re-create the view object instances in the application module's data model.

If you are curious to view its contents, you can see the XML file for the application module by double-clicking the **StoreServiceAM** node in the Application Navigator to open the overview editor. In the editor window, click the **Source** tab to view the XML so that you can inspect it. The Structure window shows the structure of the XML file.

When you create business components, JDeveloper automatically creates a data control that contains all the functionality of the application module. Data controls are an ADF Model abstraction layer that provides supplemental metadata to describe the application module's operations and data collections (row sets of view object instances), including information about the attributes, methods, and types involved. Developers can then use the representation of the data control displayed in JDeveloper's Data Controls panel to create UI components that are automatically bound to the application module. At runtime, the ADF Model layer reads the metadata describing the data controls and bindings from appropriate XML files and implements the two-way connection between the user interface and the business service.

For example, the `StoreServiceAMDataControl` application module implements the business service layer of the StoreFront module application. Its data model contains numerous view object instances, including several master-detail hierarchies. The view layer of the Fusion Order Demo application consists of JSF pages whose UI components are bound to data from the view object instances in the `StoreServiceAMDataControl`'s data model, and to built-in operations and service methods on its client interface. For details about how the Data Controls panel exposes the application module to UI developers, see Section 13.3, "Exposing Application Modules with Oracle ADF Data Controls."

9.2.3 How to Add a View Object to an Application Module

You can add a view object to an application module as you are creating the application module with the Create Application Module wizard, or you can add it later.

For information about using the Create Application Module wizard, see Section 9.2.1, "How to Create an Application Module."

9.2.3.1 Adding a View Object Instance to an Existing Application Module

You can add a view object to an application module that you have already created. To add a view object to an existing application module, and optionally, customize the view object instance, use the Data Model Components page of the overview editor for the application module.

Before you begin:

It may be helpful to have an understanding of application modules. For more information, see Section 9.2, "Creating and Modifying an Application Module."

Complete this task:

Create the desired view objects as described in Section 5.2.1, "How to Create an Entity-Based View Object," and Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object."

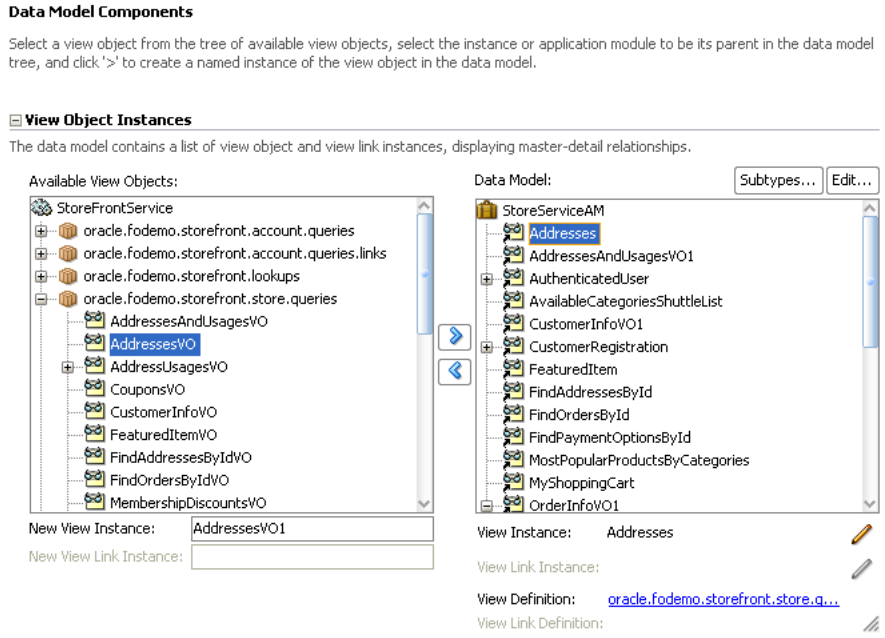
To add a view object instance to an existing application module:

1. In the Application Navigator, double-click the application module.
2. In the overview editor, click the **Data Model** navigation tab.
3. On the Data Model Components page, expand the **View Object Instances** section and, in the **Available View Objects** list, select the view instance you want to add.

   The **New View Instance** field below the list shows the name that will be used to identify the next instance of that view object that you add to the data model.
4. To change the name before adding it, enter a different name in the **New View Instance** field.
5. With the desired view object selected, shuttle the view object to the **Data Model** list.

   Figure 9–2 shows the view object `AddressVO` has been renamed to `Address` before it was shuttled to the **Data Model** list.

Figure 9–2 Data Model Displays Added View Object Instances



9.2.3.2 Adding Master-Detail View Object Instances to an Application Module

You can use the data model that the application module overview editor displays to create a hierarchy of view instances, based on existing view links that your project defines. If you have defined view links that establish more than one level of

master-detail hierarchy, then you can proceed to create as many levels of master-detail view instances as your application supports.

Before you begin:

It may be helpful to have an understanding of application modules. For more information, see Section 9.2, "Creating and Modifying an Application Module."

Complete this task:

Create hierarchical relationships between view objects as described in Section 5.6, "Working with Multiple Tables in a Master-Detail Hierarchy."

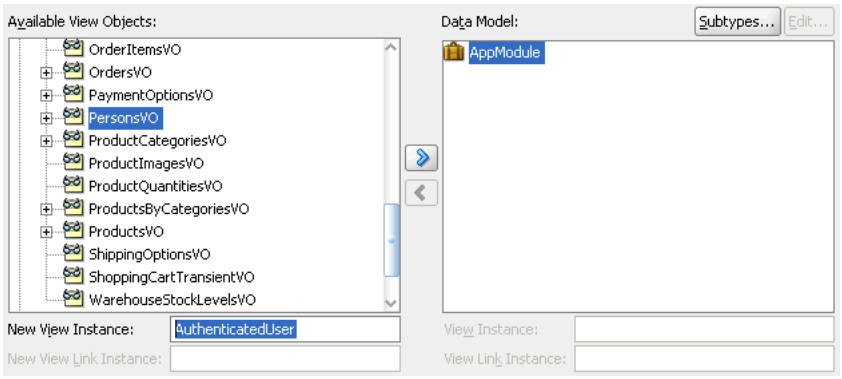
To add master-detail view object instances to a data model:

1. In the Application Navigator, double-click the application module.
2. In the overview editor, click the **Data Model** navigation tab.
3. On the Data Model Components page, expand the **View Object Instances** section and, in the **Data Model** list, select the instance of the view object that you want to be the actively coordinating master.

   The master view object will appear with a plus sign in the list indicating the available view links for this view object. The view link must exist to define a master-detail hierarchy.

   Figure 9–3 shows `PersonsVO` selected and renamed `AuthenticatedUser` in the **New View Instance** field.

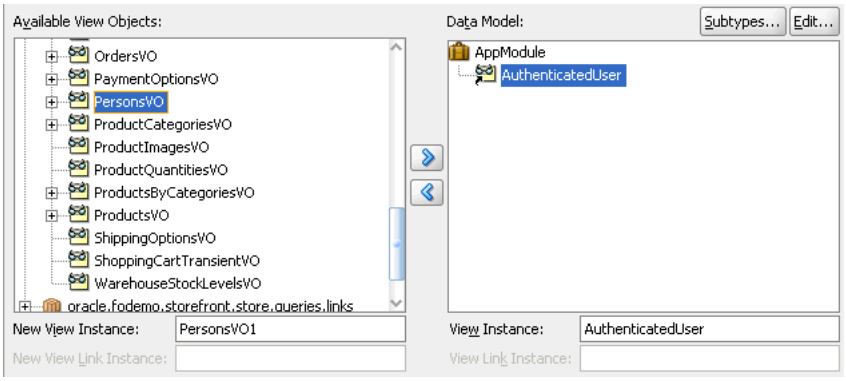
Figure 9–3 Master View Object Selected



4. Shuttle the selected master view object to the **Data Model** list.

   Figure 9–4 shows the newly created master view instance `AuthenticatedUser` in the **Data Model** list.

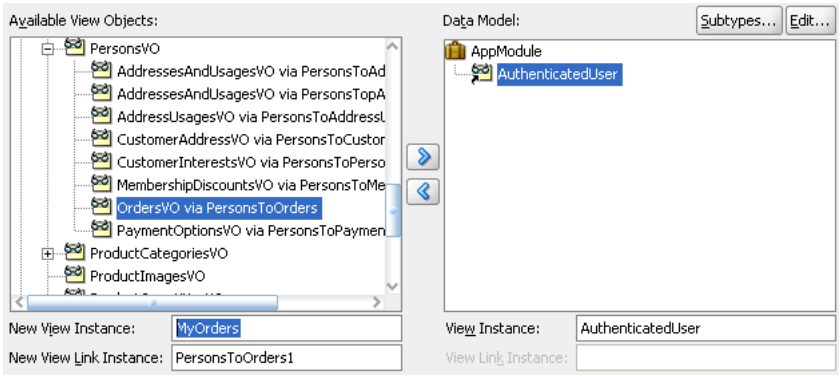
Figure 9–4 Master View Instance Created



5. In the **Data Model** list, leave the newly created master view instance selected so that it appears highlighted. This will be the target of the detail view instance you will add. Then locate and select the detail view object beneath the master view object in the **Available View Objects** list.

   Figure 9–5 shows the detail `OrdersVO` indented beneath master `PersonsVO` with the name `OrdersVO via PersonsToOrders`. The name identifies the view link `PersonsToOrders`, which defines the master-detail hierarchy between `PersonsVO` and `OrdersVO`. Notice also that the `OrdersVO` will have the view instance name `MyOrders` when added to the data model.

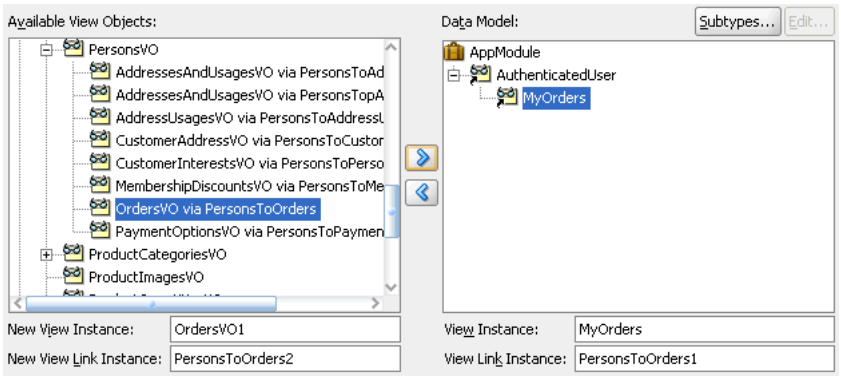
Figure 9–5 Detail View Object Selected



6. To add the detail instance to the previously added master instance, shuttle the detail view object to the **Data Model** list below the selected master view instance.

   Figure 9–6 shows the newly created detail view instance `MyOrders` as a detail of the `AuthenticatedUser` in the data model.

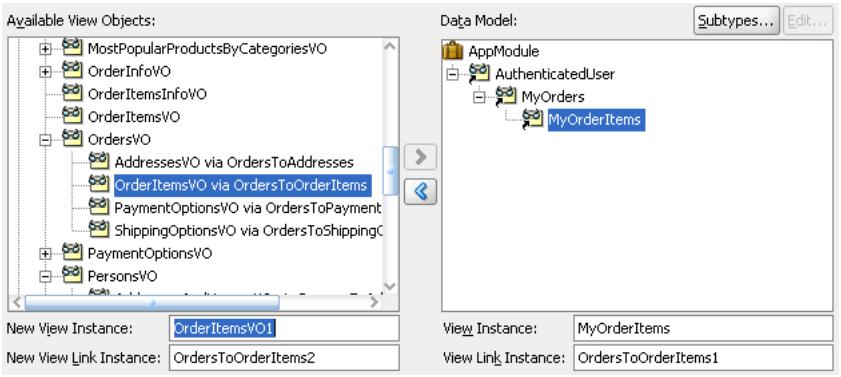
Figure 9–6 Master View Instance Created



7. To add another level of hierarchy, repeat Step 3 through Step 6, but select the newly added detail in the **Data Model** list, then shuttle over the new detail, which itself has a master-detail relationship with the previously added detail instance.

   Figure 9–7 shows the **Data Model** list with instance `AuthenticatedUser` (renamed from `PersonsVO`) as the master of `MyOrders` (renamed from `OrdersVO via PersonsToOrders`), which is, in turn, a master for `MyOrderItems` (renamed from `OrderItemsVO via OrdersToOrderItems`).

Figure 9–7 Master-Detail-Detail Hierarchy Created



9.2.3.3 Customizing a View Object Instance that You Add to an Application Module

You can optionally customize the view object instance by using the Data Model Components page of the overview editor for the application module. For example, you might want to apply a filter to set the controlling attribute for a master-detail view object relationship.

Before you begin:

It may be helpful to have an understanding of application modules. For more information, see Section 9.2, "Creating and Modifying an Application Module."

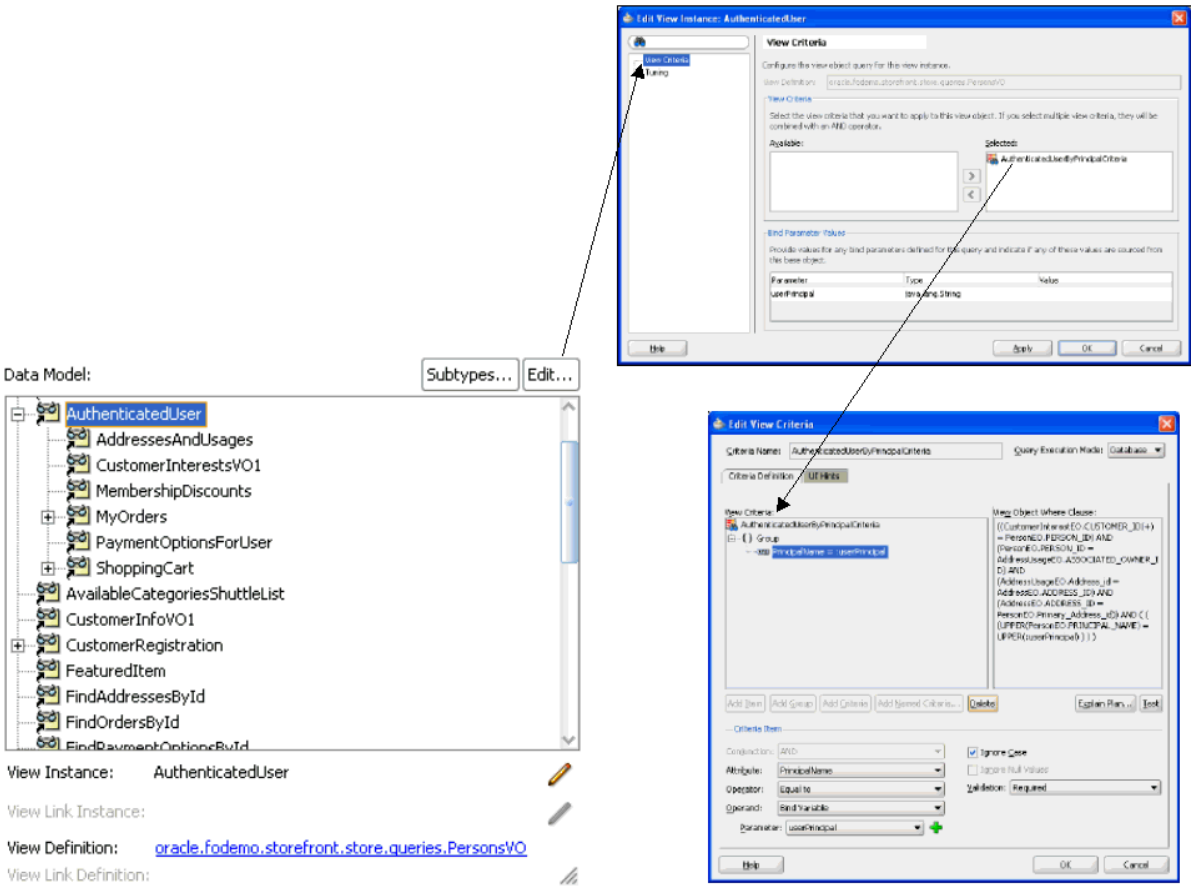
To customize a view object instance that you add to an existing application module:

1. In the Application Navigator, double-click the application module.
2. In the overview editor, click the **Data Model** navigation tab.

3. On the Data Model Components page, expand the **View Object Instances** section and, in the **Data Model** list, select the view object instance you want to customize and click the **Edit** button.
4. In the Edit View Instance dialog, perform any of the following steps, and then click **OK**.
   - In the **View Criteria** group box, select one or more view criteria that you want to apply to the view object instance. The view criteria will be appended as a `WHERE` clause to the instance query. For details about defining view criteria, see Section 5.11, "Working with Named View Criteria."
   - In the **Bind Parameters Values** group box, enter any values that you wish the instance to use when applying the defined view criteria. For more information about defining bind variables, see Section 5.10, "Working with Bind Variables."

   Figure 9–8 shows the Edit View Instance dialog opened for the `AuthenticatedUser` view usage with the `AuthenticatedUserByPrincipalCriteria` selected. No default value is supplied for the bind variable `userPrincipal` since the value will be provided at runtime through the evaluation of a Groovy expression that obtains the current user from the ADF security context. The data model for the project defines a master-detail relationship with the `PrincipalName` attribute as the controlling attribute for `AuthenticatedUser` view usage. The controlling attribute, when set by the view criteria filter, provides a way to retrieve only the view rows for the current user.

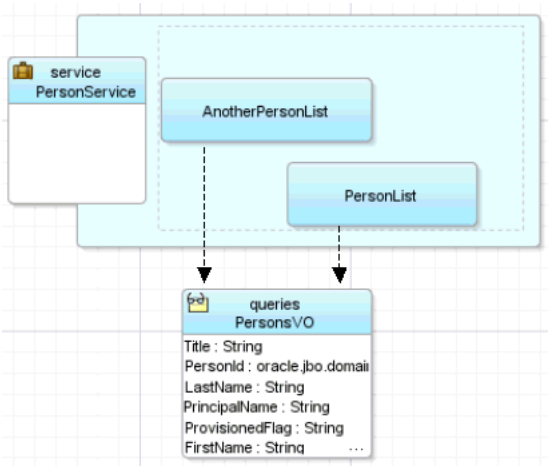
Figure 9–8 Customized View Object Instance Using a View Criteria FIlter



9.2.4 What Happens When You Add a View Object to an Application Module

While adding a view object to an application module, you use instances of a view object component to define its data model. Figure 9–9 shows a JDeveloper business components diagram of a `PersonService` application module.

Figure 9–9 Application Module Containing Two Instances of a View Object Component



The sample application module contains two instances of the `Persons` view object component, with member names of `PersonList` and `AnotherPersonList` to distinguish them. At runtime, both instances share the same `PersonsVO` view object component definition—ensure that they have the same attribute structure and view object behavior—however, each might be used independently to retrieve data about different users. For example, some of the runtime properties, like an additional filtering `WHERE` clause or the value of a bind variable, might be different on the two distinct instances.

Example 9–1 shows how the `PersonService` application module defines its member view object instances in its XML component definition file.

Example 9–1 Member View Object Instances Defined in XML

```
<AppModule Name="PersonService">
   <ViewUsage
      Name="PersonList"
      ViewObjectName="oracle.fodemo.storefront.store.queries.PersonsVO"/>
   <ViewUsage
      Name="AnotherPersonList"
      ViewObjectName="oracle.fodemo.storefront.store.queries.PersonsVO"/>
</AppModule>
```

9.2.5 How to Edit an Existing Application Module

After you've created a new application module, you can edit any of its settings by using the Edit Application Module dialog. To launch the editor, in the Application Navigator, right-click the application module node and choose **Open**, or double-click the application module. By visiting the different pages of the editor, you can adjust the data model to determine whether or not to reference nested application modules, specify Java generation settings, client interface methods, remote deployment options, runtime instantiation behavior, and custom properties.

If you edit the name of your application module, choose a name that is not among the reserved words that Oracle Application Development Framework (Oracle ADF) defines. In particular, reserved words are not valid for a data control usage name which JDeveloper automatically assigns based on your application module's name. In Fusion web applications, these reserved words consist of `data`, `bindings`, `security`, and `adfContext`. For example, you should not name an application module `data`. If JDeveloper creates a data control usage with an ID that collides with a reserved word, your application may not reliably access your data control objects at runtime and may fail with a runtime `ClassCastException`.

Do not name the application module with an initial underscore (_) character to prevent a potential name collision with a wider list of reserved words that begin with the underscore.

Application module names that incorporate a reserved word into their name (or that change the case of the reserved word) will not conflict. For example, `Product_Data`, `Product_data`, or just `Data` are all valid application module names since the whole name does not match the reserved word `data`.

9.2.6 How to Change the Data Control Name Before You Begin Building Pages

By default, an application module will appear in the Data Controls panel as a data control named *`AppModuleName`*`DataControl`. The user interface designer uses the Data Controls panel to bind data from the application module to the application's web pages. For example, if the application module is named `StoreServiceAM`, the Data Controls panel will display the data control with the name `StoreServiceAMDataControl`. You can change the default data control name to make it shorter or to supply a more preferable name.

When the user interface designer works with the data control, they will see the data control name for your application module in the `DataBindings.cpx` file in the user interface project and in each data binding page definition XML file. In addition, you might refer to the data control name in code when needing to work programmatically with the application module service interface. For this reason, if you plan to change the name of your application module, do this change before you begin building your view layer.

For complete information about the application module data control, see Chapter 13, "Using ADF Model in a Fusion Web Application."

Note: If you decide to change the application module's data control name after you have already referenced it in one or more pages, you will need to open the page definition files and `DataBindings.cpx` file where it is referenced and update the old name to the new name manually.

To change the application module data control name:

1. In the Application Navigator, double-click the application module.
2. Open the Property Inspector and expand the **Other** section.
3. Enter your preferred data control name in the **Data Control Name** field.

9.2.7 What You May Need to Know About Application Module Granularity

A common question related to application modules is, "How big should my application module be?" In other words, "Should I build one big application module to

contain the entire data model for my enterprise application, or many smaller application modules?" The answer depends on your situation.

In general, application modules should be as big as necessary to support the specific use case you have in mind for them to accomplish. They can be assembled from finer-grained application module components using a nesting feature, as described in Section 9.4, "Defining Nested Application Modules." Since a complex business application is not really a single use case, a complex business application implemented using Oracle ADF will typically not be just a single application module.

In actual practice, you may choose any granularity you wish. For example, in a small application with one main use case and a "backend" supporting use case, you could create two application modules. However, for the sake of simplicity you can combine both use cases, rather than create a second application module that contains just a couple of view objects.

9.2.8 What You May Need to Know About View Object Components and View Object Instances

While designing an application module, you use instances of a view object component to define its data model. Just as the user interface may contain two instances of a `Button` component with member names of `myButton` and `anotherButton` to distinguish them, your application module contains two instances of the `Persons` view object component, with member names of `PersonList` and `AnotherPersonList` to distinguish them.

9.3 Configuring Your Application Module Database Connection

When you initialize your data model project to use ADF Business Components, JDeveloper prompts you to supply database connection details. You must specify a database connection before you can run ADF Business Components wizards and work with the database's tables and views to create business components. After you specify the connection details, JDeveloper also creates a Java Database Connectivity (JDBC) data source that will be used as the default runtime connection for all application modules created in the current project. You can use the Edit Business Components Configuration dialog to change the JDBC data source definition for each application module individually. In cases, where you do not want to use a data source, you also use the dialog to replace the connection type for individual application modules with a JDBC URL connection type.

You can use either the JDBC data source or the JDBC URL connection type to run the application module in any context where Java can run. Your application is not restricted to running inside a Java Enterprise Edition (Java EE) application server. For example, although the Oracle ADF Model Tester is a standalone Java tool and does not run within the context of a Java EE application server, you can use either connection type to test your business components in the Oracle ADF Model Tester. You can use the Edit Business Components Configuration dialog to select among both connection types for the existing application resource connections that appear in the Database Navigator

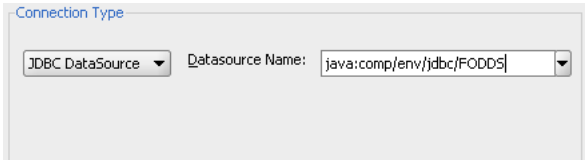
When you make no connection type selection, JDeveloper configures the application module by default to use the data source connection type. A JDBC data source is a vendor-independent encapsulation of a database server connection. The JDBC data source offers advantages that the JDBC URL connection type does not. When you define a connection type based on a data source, you reconfigure the data source without changing the deployed application. The data source is also centrally defined at the application server level, whereas JDBC URL connections are not. In cases, where

a JDBC data source is not viable, at runtime, ADF Business Components will construct a default JDBC URL based on the data source connection information.

9.3.1 How to Use a JDBC Data Source Connection Type

The type of connection all default application module configurations use is a JDBC data source. You define a JDBC data source as part of your application server configuration information, and then the application module looks up the resource at runtime using a logical name. When you use the JDBC data source as the connection type, your application resource connection details may change, and you will not need to change the deployed application module configuration. For this reason the JDBC data source is the recommended choice for all application module configurations. Figure 9–10 shows how the default selection appears in the Edit Business Components Configuration dialog.

Figure 9–10 JDBC DataSource Connection Type Setting in Edit Business Components Configuration Dialog



Example 9–2 shows the `<resource-ref>` tags in the `web.xml` file of a Fusion web application. These define two logical data sources named `jdbc/FODemoDS` and `jdbc/FODemoCoreDS`. The Edit Business Components Configuration dialog references this logical connection name after the prefix `java:comp/env` in the **JDBC Datasource Name** field. For example, the JDBC data source name for the same Fusion web application would display the value `java:comp/env/jdbc/FODemoDS` that you can select. Therefore the **Datasource Name** field is prepopulated with the JNDI name for all available application resources connection names.

Example 9–2 Logical Data Source Resource Names Defined in web.xml

```
<!-- In web.xml -->
<resource-ref>
  <res-ref-name>jdbc/FODemoDS</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
<resource-ref>
  <res-ref-name>jdbc/FODemoCoreDS</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

You can directly edit the **Datasource Name** field when you want to specify a connection name for a global data source that is required to run the application on a target standalone application server. When you deploy to Oracle WebLogic Server, by default, the application-specific data source is not packaged with the application and Oracle WebLogic Server is configured to find a global data source named `jdbc/`*`applicationConnectName`*`DS` using the look up `java:comp/env/jdbc/`*`applicationConnectName`*`DS`. Therefore, by following this naming convention, you enable a single data source connection name to work correctly when running the application in JDeveloper using an application-specific

data source or when running on the deployed standalone server using a global data source.

Note: When configuring the ADF application module to access a highly available database system, such as redundant databases or Oracle Real Application Clusters (Oracle RAC) as the backend, the data source must be container-defined. In this scenario, the application module will use a multi data source; however, from the standpoint of the application module configuration, the naming convention for the multi data source is the same as it is for an non-multi data source. This ensures that the correct data source will be used at runtime. For details about configuring multi data sources for high availability applications, see the *Oracle Fusion Middleware High Availability Guide*.

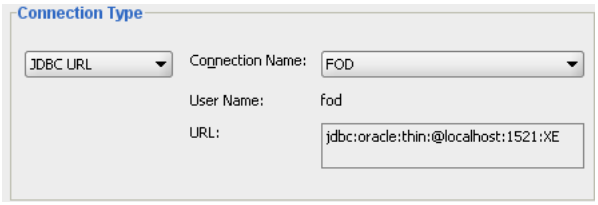
9.3.2 How to Use a JDBC URL Connection Type

The other type of connection type you can use for an application module configuration is a JDBC URL connection. It is based on the named connection definition set on the Business Components page of the Project Properties dialog for the project containing your application module. Figure 9–11 shows what this section would look like in a configuration using a JDBC URL connection.

Note: If you are using a non-Oracle JDBC driver, make sure that you set the appropriate properties for the driver on the **Properties** page of the Edit Business Components Configuration dialog to avoid runtime exceptions.

For example, if you are using a Sybase JDBC driver, you must set the `jbo.sql92.LockTrailer` property (which is set to `FOR UPDATE` by default) to a value appropriate for the Sybase database. Not setting this property would generate a `SybSQLException` runtime exception.

Figure 9–11 JDBC URL Connection Type Setting in Edit Business Components Configuration Dialog



Note: See Section 44.1.1.2, "Database Connection Pools," and Section 44.2.9, "What You May Need to Know About Database Connection Pool Parameters" for more information on how database connection pools are used and how you can tune them.

9.3.3 What Happens When You Create an Application Module Database Connection

When you select connection type in the Edit Business Components Configuration dialog, JDeveloper updates the application module configuration file, `bc4j.xcfg` in

the ./common subdirectory relative to the application module's XML component definition. The file defines configurations for all of the application modules in a single Java package. For example, if you look at the bc4j.xcfg file in the ./classes/oracle/fodemo/storefront/store/service/common directory of the Fusion Order Demo application's StoreFront project, you will see the three named configurations for its StoreServiceAM application module.

The configurations defined by the bc4j.xcfg file allow the Fusion web application to interact with specific, deployed application modules. In addition to the connection type for the application module, the bc4j.xcfg file contains metadata information about application module names and it contains the runtime parameters that are configured for the application module. The application resource database connection details for the named connection type are defined in the Connections folder of the Application Navigator and saved in the application's connection.xml file.

Example 9–3 displays a sample bc4j.xcfg file from the Fusion Order Demo application. The configurations StoreServiceAMLocal and StoreServiceAMLocalWeb both reference a data source (named FODDS) in the JDBCDataSource attribute. The configuration StoreFrontService references a data source that will eventually be defined on the target application server. The JDBCDataSource attribute in each configuration specifies the JNDI name for the application resources connection name in the form of java:comp/env/jdbc/*applicationConnectName*DS, where *applicationConnectName* is the name of the application resources database connection defined in JDeveloper (in this case, FOD). This JNDI naming convention (with the application-specific name space java:comp/env/jdbc/ and DS appended to the application resources database connection name) ensures that a deployed Fusion web application will run on Oracle WebLogic Server using the application's global data source and no changes will be required. The global data source is typically defined by the application server administrator using the Oracle WebLogic Server Administration Console.

Example 9–3 Application Module Database Configurations in the bc4j.xcfg File

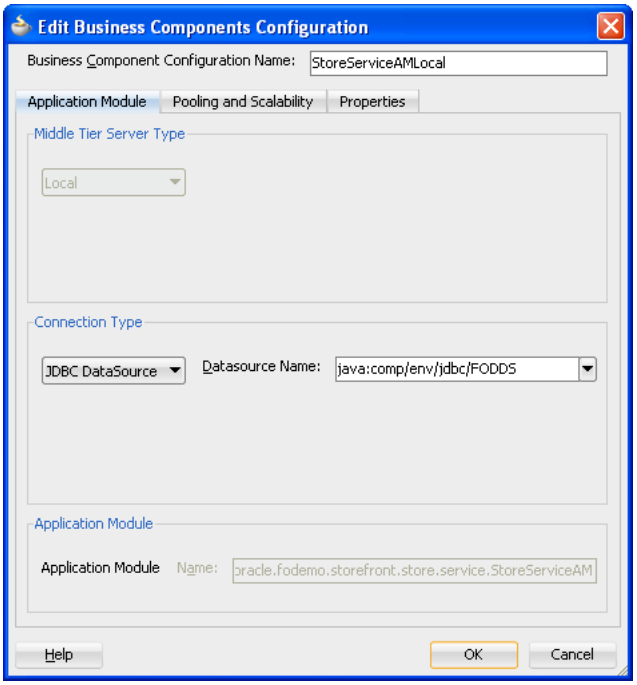
```
<BC4JConfig version="11.1" xmlns="http://xmlns.oracle.com/bc4j/configuration">
   <AppModuleConfigBag ApplicationName="oracle.fodemo.storefront.store.service.StoreServiceAM">
      <AppModuleConfig
            DeployPlatform="LOCAL"
            jbo.project="StoreFrontService"
            name="StoreServiceAMLocal"
            ApplicationName="oracle.fodemo.storefront.store.service.StoreServiceAM">
         <Database jbo.locking.mode="optimistic"/>
         <Security AppModuleJndiName="oracle.fodemo.storefront.store.service.StoreServiceAM"/>
         <Custom JDBCDataSource="java:comp/env/jdbc/FODDS"/>
      </AppModuleConfig>
      <AppModuleConfig
            DeployPlatform="LOCAL"
            jbo.project="StoreFrontService"
            name="StoreServiceAMLocalWeb"
            ApplicationName="oracle.fodemo.storefront.store.service.StoreServiceAM">
         <AM-Pooling jbo.ampool.initpoolsize="1"/>
         <Database jbo.locking.mode="optimistic"/>
         <Security AppModuleJndiName="oracle.fodemo.storefront.store.service.StoreServiceAM"/>
         <Custom JDBCDataSource="java:comp/env/jdbc/FODDS" fod.application.issoaenabled="false"/>
      </AppModuleConfig>
      <AppModuleConfig
            DeployPlatform="SI"
            jbo.project="StoreFrontService"
            name="StoreFrontService"
```

```
            ApplicationName="oracle.fodemo.storefront.store.service.StoreServiceAM">
          <Database jbo.TypeMapEntries="Java" jbo.locking.mode="optimistic"/>
          <Security AppModuleJndiName="oracle.fodemo.storefront.store.service.StoreServiceAM"/>
          <Custom JDBCDataSource="java:comp/env/jdbc/FODDS"/>
        </AppModuleConfig>
    </AppModuleConfigBag>
</BC4JConfig>
```

9.3.4 How to Change Your Application Module's Runtime Configuration

In addition to creating the application module XML component definition, JDeveloper also adds a default configuration named *appModuleName*`Local` to the `bc4j.xcfg` file in the subdirectory named `common`, relative to the directory containing the application module XML component definition file. The `bc4j.xcfg` file does not appear in the Application Navigator. To view the default settings or to change the application module's runtime configuration settings, you can use the Edit Business Components Configuration dialog shown in Figure 9–12.

Figure 9–12 bc4j.xcfg File Configurations Displayed by Edit Business Components Configuration Dialog



Before you begin:

It may be helpful to have an understanding of application module database connections. For more information, see Section 9.3, "Configuring Your Application Module Database Connection."

To manage your application module's configuration:

1. In the Application Navigator, double-click the application module.
2. In the overview editor, click the **Configurations** navigation tab.
3. In the Configurations page, double-click the configuration you want to edit. JDeveloper may display these three default configurations:

- *appModuleName*`Local`: One local configuration is automatically defined when you create the application module. Create a new local configuration, for example, when you want to change the runtime properties for the application module for testing purposes.
- *appModuleName*`Shared`: The shared application module configuration is present only when a named instance of a shared application module has been defined in the Project Properties dialog for the data model project. A shared application module is one that groups view instances when you want to reuse lists of static data across the application. For example, you can define a shared application module to group view instances that access lookup data, such as a list of countries.
- *appModuleName*`Service` (type SI for service interface): The service interface configuration is present only when the service interface (web service implementation wrapper) has been defined in the Service Interface page of the overview editor for the application module. An application module exposed as a web service allows your application module to participate in a composite application to provide data access and method calls to web service clients. The same application module can support interactive web user interfaces using ADF data controls and web service clients.

4. In the Edit Business Components Configuration dialog, edit the desired runtime properties and click **OK** to save the changes for your configuration.

9.3.5 How to Change the Database Connection for Your Project

When you are developing applications, you may have a number of different users or schemas that you want to switch between. You can do this by changing the database connection properties of the project that contains the business components. The selection you make will automatically update the connection string for each configuration in your project's `bc4j.xcfg` file that specifies a JDBC URL type connection.

Before you begin:

It may be helpful to have an understanding of application module database connections. For more information, see Section 9.3, "Configuring Your Application Module Database Connection."

To change the connection used by your application module's configuration:

1. In the Application Navigator, right-click the project that contains the application module and choose **Project Properties**.
2. In the Project Properties dialog, select **Business Components** to display the Business Components page, which shows details of the current database connection.
3. Click **Edit**, and in the Edit Database Connection dialog, make the appropriate changes.
4. Click **OK**.

9.4 Defining Nested Application Modules

Application modules support the ability to create software components that mimic the modularity of your use cases, for which your higher-level functions might reuse a "subfunction" that is common to several business work flows. You can implement this

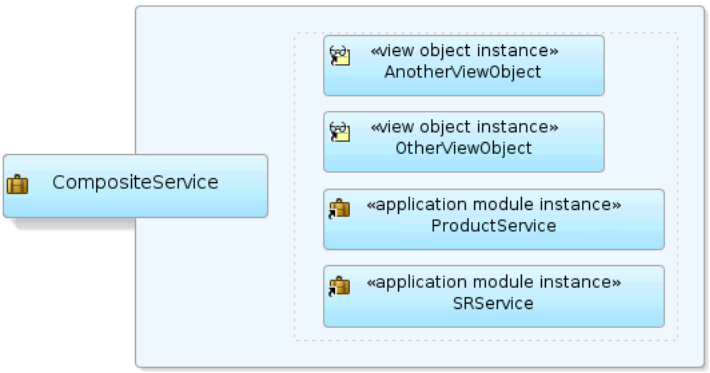
modularity by defining composite application modules that you assemble using instances of other application modules. This task is referred to as *application module nesting*. That is, an application module can contain (logically) one or more other application modules, as well as view objects. The outermost containing application module is referred to as the *root application module*.

Declarative support for defining nested application modules is available through the overview editor for the application module, as shown in Figure 9–14. The API for application modules also supports nesting of application modules at runtime.

When you nest an instance of one application module inside another, you aggregate not only the view objects in its data model, but also any custom service methods it defines. This feature of "nesting," or reusing, an instance of one application module inside of another is one of the most powerful design aspects of ADF Business Components for implementing larger-scale, real-world application systems.

Using the basic logic that an application module represents an end-user use case or work flow, you can build application modules that cater to the data required by some shared, modular use case, and then reuse those application modules inside of other more complicated application modules that are designed to support a more complex use case. For example, imagine that after creating the application modules `StoreServiceAM` and `ProductService`, you later need to build an application that uses both of these services as an integral part of a new `CompositeService` application module. Figure 9–13 illustrates what this `CompositeService` would look like in a JDeveloper business components diagram. Notice that an application module like `CompositeService` can contain a combination of view object instances and application module instances.

Figure 9–13 Application Module Instances Can Be Reused to Assemble Composite Services



9.4.1 How to Define a Nested Application Module

To specify a composite root application module that nests an instance of an existing application module, use the overview editor for the application module. All of the nested component instances (contained by the application module instance) share the same transaction and entity object caches as the root application module that reuses an instance of them.

> **Tip:** If you leverage nested application modules in your application, be sure to read Section 13.3.1.4, "How Nested Application Modules Appear in the Data Controls Panel" to avoid common pitfalls when performing data binding involving them.

Before you begin:

It may be helpful to have an understanding of nested application modules. For more information, see Section 9.4, "Defining Nested Application Modules."

Complete this task:

Create the desired application modules as described in Section 9.2.1, "How to Create an Application Module."

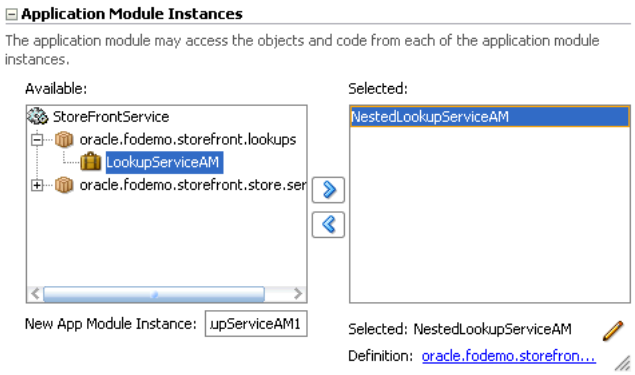
To define a nested application module:

1. In the Application Navigator, double-click the root application module.
2. In overview editor, click the **Data Model** navigation tab.
3. In the Data Model Components page, expand the **Application Module Instances** section and, in the **Available** list, select the application module that you want to add to the data model.

   The **New App Module Instance** field below the list shows the name that will be used to identify the nested application module that you add to the data model.
4. To change the name before adding it, type a different name in the **New App Module Instance** field.
5. With the desired application module selected, shuttle the application module to the **Selected** list.

   Figure 9–14 shows the application module `LookupServiceAM` has been renamed to `NestedLookupServiceAM` before it was shuttled to the **Selected** list.

Figure 9–14 Data Model Displays Added Application Module Instances



9.4.2 What You May Need to Know About Root Application Modules Versus Nested Application Module Usages

At runtime, your application works with a *main* — or what's known as a *root* — application module. Any application module can be used as a root application module; however, in practice the application modules that are used as root application modules are the ones that map to more complex end-user use cases, assuming you're not just building a straightforward CRUD application. When a root application module contains other nested application modules, they all participate in the root application module's transaction and share the same database connection and a single set of entity caches. This sharing is handled for you automatically by the root application module and its `Transaction` object.

Additionally, when you construct an application using an ADF bounded task flow, to declaratively manage the transactional boundaries, Oracle ADF will automatically nest application modules used by the task flow at runtime. For details about bounded task flows and transactions, see Section 22.2, "Managing Transactions in Task Flows."

9.5 Creating an Application Module Diagram for Your Business Service

As you develop the business service's data model, it is often convenient to be able to visualize it using a UML model. JDeveloper supports easily creating a diagram for your application module that other developers can use for reference.

You can perform a number of tasks directly on the diagram, such as editing the application module, controlling display options, filtering methods names, showing related objects and files, publishing the application, and launching the Oracle ADF Model Tester.

9.5.1 How to Create an Application Module Diagram

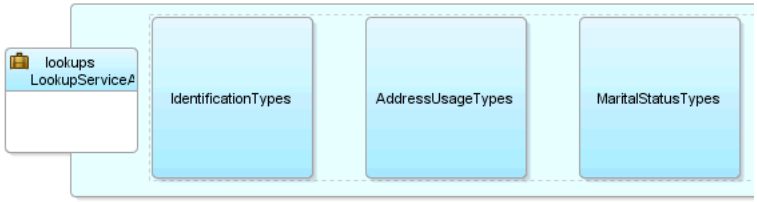
To create an application module diagram, use the Create Business Components Diagram dialog, which is available in the New Gallery.

To create a diagram of your application module:

1. In the Application Navigator, right-click the project in which you want to create the diagram and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **Business Components Diagram**, and click **OK**.
3. In the Create Business Components dialog, enter a diagram name and a package name in which the diagram will be created.
4. Click **OK** to create the empty diagram and open the diagrammer.
5. To add your existing application module to the open diagram, select the desired application module in the Application Navigator and drop it onto the diagram surface.
6. Use the Property Inspector to:
   - Hide the package name
   - Change the font
   - Turn off the grid and page breaks
   - Turn off the display of the end names on the view link connectors (`"Master"/"Detail"`)

After completing these steps, the diagram looks similar to the diagram shown in Figure 9–15.

Figure 9–15 Partial UML Diagram of Application Module



9.5.2 What Happens When You Create an Application Module Diagram

When you create a business components diagram, JDeveloper creates a `.adfbc_diagram` file to represents the diagram in a subdirectory of the project's model path that matches the package name in which the diagram resides.

By default, the Application Navigator unifies the display of the project content's paths so that ADF components and Java files in the source path appear in the same package tree as the UML model artifacts in the project model path. You can use the **Navigator Display Options > Show Directories** toolbar option in the Application Navigator to switch between the unified directory view and a more distinct directory path view of the project content.

9.5.3 How to Use the Diagram to Edit the Application Module

The UML diagram of business components is not just a static picture that reflects the point in time when you dropped the application module onto the diagram. Rather, it is a UML-based rendering of the current component definitions, so it will always reflect the current state of affairs. The UML diagram is both a visualization aid and a visual navigation and editing tool.

You can bring up the overview editor for any application module in a diagram by choosing **Properties** from the context menu (or by double-clicking the application module).

You can also perform some application module editing tasks directly on the diagram, tasks such as renaming view object instances, dropping view object definitions from the Application Navigator onto the data model to create a new view object instance, and removing view object instances by pressing the Delete key.

Note: Deleting components from the diagram only removes their visual representation on the diagram surface. The components and classes remain on the file system and in the Application Navigator.

9.5.4 How to Control Diagram Display Options

After you display the application module in the diagram, you can use the Property Inspector to control its display options.

In the **Display Options** category, toggle properties like the following:

- **Show Methods** — to display service methods
- **Show Package** — to display the package name
- **Show Stereotype** — to display the type of object (for example "`<<application module>>`")
- **Show Usages** — to display all usages by other application modules in the current project

---

Note: The term *operation* is a more generic, UML name for methods.

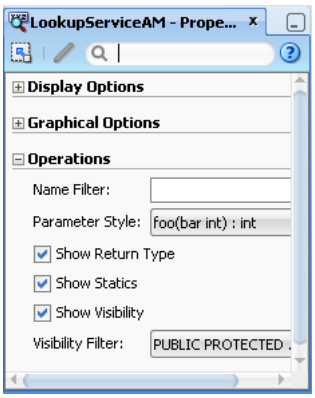
In the **Operations** category, consider changing the following properties depending on the amount of detail you want to provide in the diagram:

- **Parameter Style**
- **Show Return Type**

- **Show Visibility** (public, private, etc.)

By default, all operations of the application module are fully displayed, as shown by the Property Inspector settings in Figure 9–16.

Figure 9–16 Property Inspector with Default Diagrammer Options



On the context menu of the diagram, you can also select to **View As**:

- **Compact** — to show only the icon and the name
- **Symbolic** — to show service operations
- **Expanded** — to show operations and data model (*default*)

9.5.5 How to Filtering Method Names Displayed in the Diagram

Initially, if you show the operations for the application module, the diagram displays all the methods. Any method it recognizes as an overridden framework method displays in the **<<Framework>>** operations category. The rest display in the **<<Business>>** methods category.

The **Name Filter** property in the **Operations** category of the Property Inspector is a regular expression that you can use to filter out methods you don't want to display on the diagram. For example, by setting the **Name Filter** property to:

```
findLoggedInUser.*|retrieveOrder.*|get.*
```

you can filter out all of the following application module methods:

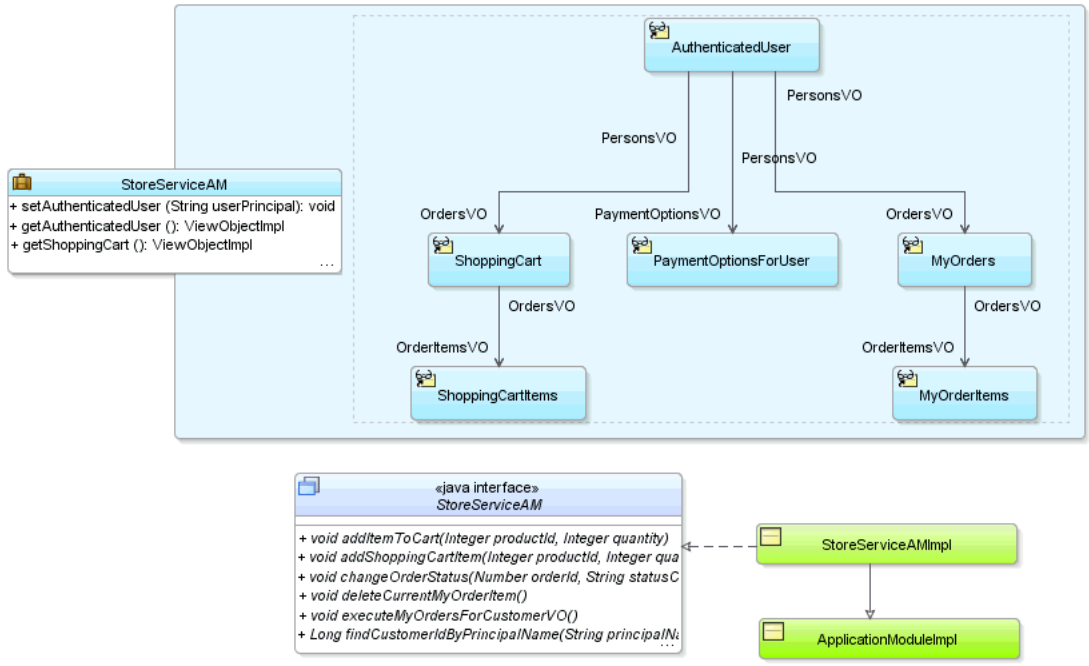
- `findLoggedInUserByEmail`
- `retrieveOrderById`
- All the generated view object getter methods

9.5.6 How to Show Related Objects and Implementation Files in the Diagram

After selecting the application module on the diagram — or any set of individual view object instances in its data model — you can choose **Show > Related Elements** from the context menu to display related component definitions on the diagram. In a similar fashion, choosing **Show > Implementation Files** will include the files that implement the application module on the diagram. You can repeat these options on the additional diagram elements that appear until the diagram includes the level of detail you want to convey.

Figure 9–17 illustrates how the diagram displays the implementation files for an application module. You will see the related elements for the application module's implementation class (`StoreServiceAMImpl`). The diagram also draws an additional dependency line between the application module and the implementation class. If you have cast the application module instance to a specific custom interface, the diagram will also show that.

Figure 9–17 Adding Detail to a Diagram Using Show Related Elements and Show Implementation Files



9.5.7 How to Publish the Application Module Diagram

To publish the diagram to `PNG`, `JPG`, `SVG`, or compressed `SVG` format, choose **Publish Diagram** from the context menu on the diagram surface.

9.5.8 How to Test the Application Module from the Diagram

To launch the Oracle ADF Model Tester for an application module in the diagram, choose **Run** from the context menu.

9.6 Supporting Multipage Units of Work

While interacting with your Fusion web application, end users might:

- Visit the same pages multiple times, expecting fast response times
- Perform a logical unit of work that requires visiting many different pages to complete
- Need to perform a partial "rollback" of a pending set of changes they've made but haven't saved yet.
- Unwittingly be the victim of an application server failure in a server farm before saving pending changes

The application module pooling and state management features simplify implementing scalable, well-performing applications to address these requirements.

> **Note:** ADF bounded task flows can represent a transactional unit of work. You can specify options on the task flow to determine how to handle the transaction. For details about the declarative capabilities of ADF bounded task flows, see Section 22.2, "Managing Transactions in Task Flows."

9.6.1 How to Simulate State Management in the Oracle ADF Model Tester

To simulate what the state management functionality does, you can launch two instances of Oracle ADF Model Tester on an application module in the Application Navigator.

Before you begin:

It may be helpful to have an understanding of state management. For more information, see Section 9.6, "Supporting Multipage Units of Work."

To simulate transaction state passivation using the Oracle ADF Model Tester:

1. Run the Oracle ADF Model Tester and double-click a view object instance to query its data.
2. Make a note of the current values of a several attributes for a few rows.
3. Update those rows to have a different value those attributes, but do not commit the changes.
4. Choose **File > Save Transaction State** from the Oracle ADF Model Tester main menu.

   A Passivated Transaction State dialog appears, indicating a numerical transaction ID number. Make a note of this number.
5. Exit out of the Oracle ADF Model Tester completely.
6. Restart the Oracle ADF Model Tester and double-click the same view object instance to query its data.
7. Notice that the data is *not* changed. The queried data from the data reflects the current state of the database without your changes.
8. Choose **File > Restore Transaction State** from the Oracle ADF Model Tester main menu, and enter the transaction ID you noted in Step 4.

At this point, you'll see that your pending changes are reflected again in the rows you modified. If you commit the transaction now, your changes are permanently saved to the database.

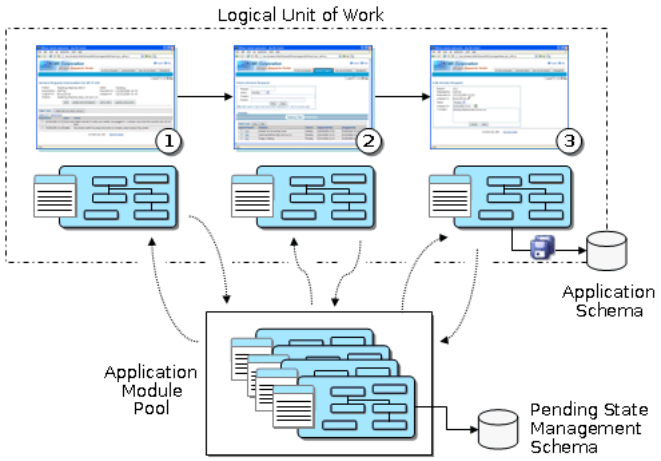
9.6.2 What Happens at Runtime: How the Application Uses Application Module Pooling and State Management

Applications you build that leverage an application module as their business service take advantage of an automatic application module pooling feature. This facility manages a configurable set of application module instances that grows and shrinks as the end-user load on your application changes during the day. Due to the natural "think time" inherent in the end user's interaction with your application user interface,

the number of application module instances in the pool can be smaller than the overall number of active users using the system.

As shown in Figure 9–18, as a given end user visits multiple pages in your application to accomplish a logical task, with each page request an application module instance in the pool is acquired automatically from the pool for the lifetime of that one request. At the end of the request, the instance is automatically returned to the pool for use by another user session. In order to protect the end user's work against application server failure, the application module supports the ability to freeze the set of pending changes in its entity caches to a persistent store by saving an XML snapshot describing the change set. For scalability reasons, this state snapshot is typically saved in a state management schema that is a different database schema than the one containing the application data.

Figure 9–18 Using Pooled Application Modules Throughout a Multipage, Logical Unit of Work



The pooling algorithm affords a tunable optimization whereby a certain number of application module instances will attempt to stay "sticky" to the last user session that returned them to the pool. The optimization is not a guarantee, but when a user can benefit from the optimization, they continue to work with the same application module instance from the pool as long as system load allows. When load is too high, the pooling algorithm uses any available instance in the pool to service the user's request and the frozen snapshot of their logical unit of work is reconstituted from the persistent store to allow the new instance of the application module to continue where the last one left off. The end user continues to work in this way until they commit or roll back their changes.

Using these facilities, the application module delivers the productivity of a stateful development paradigm that can easily handle multipage work flows, in an architecture that delivers the runtime performance near that of a completely stateless application. You will learn more about these application module features in Chapter 43, "Application State Management" and about how to tune them in Chapter 44, "Tuning Application Module Pools and Connection Pools."

Note: This application module pooling and state management is also available for thin-client, desktop-fidelity Swing applications and web-style user interfaces.

9.7 Customizing an Application Module with Service Methods

An application module can expose its data model of view objects to clients without requiring any custom Java code. This allows client code to use the `ApplicationModule`, `ViewObject`, `RowSet`, and `Row` interfaces in the `oracle.jbo` package to work directly with any view object in the data model. However, just because you *can* programmatically manipulate view objects any way you want to in client code doesn't mean that doing so is always a best practice.

Whenever the programmatic code that manipulates view objects is a logical aspect of implementing your complete business service functionality, you should encapsulate the details by writing a custom method in your application module's Java class. This includes code that:

- Configures view object properties to query the correct data to display
- Iterates over view object rows to return an aggregate calculation
- Performs any kind of multistep procedural logic with one or more view objects

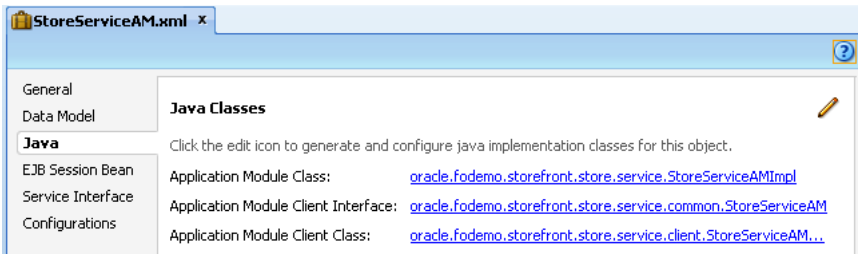
By centralizing these implementation details in your application module, you gain the following benefits:

- You make the intent of your code more clear to clients.
- You allow multiple client pages to easily call the same code if needed.
- You simplify regression-testing of your complete business service functionality.
- You keep the option open to improve your implementation without affecting clients.
- You enable *declarative* invocation of logical business functionality in your pages.

9.7.1 How to Generate a Custom Class for an Application Module

To add a custom service method to your application module, you must first enable a custom Java class for it. If you have configured your IDE-level Business Components Java generation preferences to automatically generate an application module class, a custom class will be present. As Figure 9–19 shows, if you're not sure whether your application module has a custom Java class, open the overview editor for the application module node in the Application Navigator. The Java Classes page of the editor displays the complete list of classes generated for the application module in the project. If the file exists because someone created it already, then the Java Classes page will display a linked file name identified as the **Application Module Class**. To open an existing file in the source editor, click the corresponding file name link.

Figure 9–19 Application Module's Custom Java Class in Overview Editor



If no Java class exists in your project, you can generate one using the Java Classes page of the overview editor for the application module.

Before you begin:

It may be helpful to have an understanding of application module service methods. For more information, see Section 9.7, "Customizing an Application Module with Service Methods."

Complete this task:

Create the desired application modules as described in Section 9.2.1, "How to Create an Application Module."

To generate a Java file for your application module class:

1. In the Application Navigator, double-click the application module.
2. In the overview editor, click the **Java** navigation tab and click the **Edit java options** button.
3. In the Select Java Options dialog, select **Generate Application Module Class**.
4. Click **OK**.

   The new `.java` file will appear in the Java Classes page.

9.7.2 What Happens When You Generate a Custom Class for an Application Module

When you generate a custom class for an application module, JDeveloper creates the file in the same directory as the component's XML component definition file. The default name for its custom Java file will be *AppModuleName*`Impl.java`.

The Java generation option choices you made for the application module persist on the Java Classes page on subsequent visits to the overview editor for the application module. Just as with the XML definition file, JDeveloper keeps the generated code in your custom Java classes up to date with any changes you make in the editor. If later you decide you do not require a custom Java file, from the Java Classes page open the Select Java Options dialog and deselect **Generate Application Module Class** to remove the custom Java file from the project.

9.7.3 What You May Need to Know About Default Code Generation

By default, the application module Java class will look similar to what you see in Example 9–4 when you've first enabled it. Of interest, it contains:

- Getter methods for each view object instance in the data model
- A `main()` method allowing you to debug the application module using the Oracle ADF Model Tester

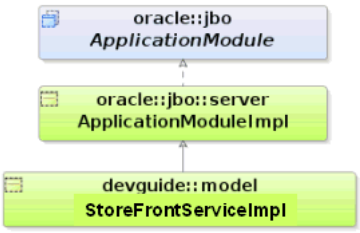
Example 9–4 Default Application Module Generated Code

```
package devguide.model;
import devguide.model.common.StoreServiceAM;
import oracle.jbo.server.ApplicationModuleImpl;
import oracle.jbo.server.ViewLinkImpl;
import oracle.jbo.server.ViewObjectImpl;
// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---    Custom code may be added to this class.
// ---    Warning: Do not modify method signatures of generated methods.
// ---------------------------------------------------------------------
public class StoreServiceAMImpl extends ApplicationModuleImpl {
  /** This is the default constructor (do not remove) */
  public StoreServiceImpl() { }
```

```
  /** Container's getter for YourViewObjectInstance1 */
  public ViewObjectImpl getYourViewObjectInstance1() {
    return (ViewObjectImpl)findViewObject("YourViewObjectInstance1");
  }

  // ... Additional ViewObjectImpl getters for each view object instance

  // ... ViewLink getters for view link instances here
}
```

As shown in Figure 9–20, your application module class extends the base ADF `ApplicationModuleImpl` class to inherit all the default behavior before adding your custom code.

Figure 9–20 Your Custom Application Module Class Extends ApplicationModuleImpl



9.7.4 How to Add a Custom Service Method to an Application Module

To add a custom service method to an application module, simply navigate to the application module's custom class and enter the Java code for a new method into the application module's Java implementation class. Use the following guidelines to decide on the appropriate visibility for the method:

- If you will use the method only inside this component's implementation as a helper method, make the method `private`.
- If you want to allow eventual subclasses of your application module to be able to invoke or override the method, make it `protected`.
- If you need clients to be able to invoke it, it must be `public`.

> **Note:** The `StoreServiceAM` application module examples in this chapter use the strongly typed, custom entity object classes that you saw illustrated in the `StoreServiceAMImpl2.java` example at the end of Chapter 4, "Creating a Business Domain Layer Using Entity Objects."

Example 9–5 shows a `private retrieveOrderById()` helper method in the `StoreServiceAMImpl.java` class for the `StoreServiceAM` application module. It uses the `static getDefinition()` method of the `OrdersEOImpl` entity object class to access its related entity definition, it uses the `createPrimaryKey()` method on the entity object class to create an appropriate Key object to look up the order, and then it uses the `findByPrimaryKey()` method on the entity definition to find the entity row in the entity cache. It returns an instance of the strongly typed `OrdersEOImpl` class, the custom Java class for the `OrderEO` entity object.

Example 9–5 Private Helper Method in Custom Application Module Class

```
// In devguide.model.StoreServiceAMImpl class
/*
 * Helper method to return a Order by Id
 */
private OrdersEOImpl retrieveOrderById(long orderId) {
  EntityDefImpl orderDef = OrdersEOImpl.getDefinitionObject();
  Key orderKey =
    OrdersEOImpl.createPrimaryKey(new DBSequence(orderId));
  return (OrdersEOImpl)orderDef.findByPrimaryKey(getDBTransaction(),
                                                 orderKey);
}
```

Example 9–6 shows a `public createProduct()` method that allows the caller to pass in a name and description of a product to be created. It uses the `getDefinition()` method of the `ProductImpl` entity object class to access its related entity definition, and it uses the `createInstance2()` method to create a new `ProductImpl` entity row, whose `Name` and `Description` attributes it populates with the parameter values passed in before committing the transaction.

Example 9–6 Public Method in Custom Application Module Class

```
/*
 * Create a new Product and Return its new id
 */
public long createProduct(String name, String description) {
  EntityDefImpl productDef = ProductImpl.getDefinitionObject();
  ProductImpl newProduct =
    (ProductImpl)productDef.createInstance2(getDBTransaction(),null);
  newProduct.setName(name);
  newProduct.setDescription(description);
  try {
    getDBTransaction().commit();
  }
  catch (JboException ex) {
    getDBTransaction().rollback();
    throw ex;
  }
  DBSequence newIdAssigned = newProduct.getProdId();
  return newIdAssigned.getSequenceNumber().longValue();
}
```

9.7.5 How to Test the Custom Application Module Using a Static Main Method

When you are ready to test the methods of your custom application module, you can use JDeveloper to generate JUnit test cases. With JUnit, you can use any of the programmatic APIs available in the `oracle.jbo package` to work with the application module and invoke the custom methods. For details about using JUnit with ADF Business Components, see Section 36.11, "Regression Testing with JUnit."

As an alternative to JUnit test cases, a common technique to test your custom application module methods is to write a simple test case. For example, you could build the testing code into an object and include that code in the `static main()` method. Example 9–7 shows a sample `main()` method you could add to your custom application module class to test the sample methods you will write. You'll make use of a `Configuration` object (see Section 6.4.2, "How to Create a Command-Line Java Test Client") to instantiate and work with the application module for testing.

> **Note:** The fact that this `Configuration` object resides in the `oracle.jbo.client` package suggests that it is used for accessing an application module as an application *client*. Because a `main()` method is a kind of programmatic, command-line client, so this is an acceptable practice. Furthermore, even though you typically would not cast the return value of `createRootApplicationModule()` directly to an application module's implementation class, it is legal to do so in this one situation since despite being a client to the application module, the `main()` method's code resides right inside the application module implementation class itself.

A glance through the code in Example 9–7 shows that it exercises the four methods created in the previous examples to:

1. Retrieve the total for order 1011.
2. Retrieve the name of the customer for order 1011.
3. Set the status of order 1011 to the value "CANCEL".
4. Create a new product supplying a null product name.
5. Create a new product with a product name and display its newly assigned product ID.

Example 9–7 Sample Main Method to Test a Custom Application Module from the Inside

```
// Main method in StoreServiceAMImpl.java
  public static void main(String[] args) {
    String        amDef = "devguide.model.StoreFrontService";
    String        config = "StoreServiceLocal";
    /*
     * This is the correct way to use application custom methods
     * from the client, by using the application module's automatically-
     * maintained custom service interface.
     */
    // 1. Acquire instance of application module, cast to client interface
    ApplicationModule am =
                          Configuration.createRootApplicationModule(amDef,config);
/*
* NOTE: This cast to use the StoreFrontServiceImpl class is OK since this
*       code is inside a business tier *Impl.java file and not in a
*       client class that is accessing the business tier from "outside".
*/
    StoreFrontServiceImpl service = (StoreFrontServiceImpl)am;
    String total = service.findOrderTotal(1011);
    System.out.println("Total for Order # 1011 = " + total);
    String customerName = service.findOrderCustomer(1011);
    System.out.println("Customer for Order # 1011 = " + customerName);
    try {
       service.updateOrderStatus(1011,"CANCEL");
    }
    catch (JboException ex) {
       System.out.println("ERROR: "+ex.getMessage());
    }
    long id = 0;
    try {
       id = service.createProduct(null, "NEW", "CLASS1");
    }
```

```
    catch (JboException ex) {
       System.out.println("ERROR: "+ex.getMessage());
    }
    id = service.createProduct("Canon PowerShot G9", "NEW", "CLASS1");
    System.out.println("New product created successfully with id = "+id);
    Configuration.releaseRootApplicationModule(am,true);
  }
```

Running the custom application module class calls the `main()` method in Example 9–7, and shows the following output:

```
Total for Order # 1011 = 99.99
Customer for Order # 1011 = John Chen
ERROR: JBO-27014: Attribute ProductName in ProductsBaseEO is required.
New product created successfully with id = 133
```

Notice that the first attempt to call `createProduct()` with a null for the product name raises an exception due to the built-in mandatory validation on the `Name` attribute of the `Product` entity object.

> **Note:** For an explanation of how you can use the client application to invoke the custom service methods that you create in your custom application module, see Section 9.9, "Publishing Custom Service Methods to UI Clients."

9.7.6 What You May Need to Know About Programmatic Row Set Iteration

Any time your application logic accesses a row set to perform programmatic iteration, you should use a secondary row set iterator when working with view object instances in an application module's data model, or view link accessor row sets of these view object instances, since they may be bound to user interface components. To create a secondary iterator, use the `createRowSetIterator()` method on the row set you are working with. When you are done using it, call the `closeRowSetIterator()` method on the row set to remove the secondary iterator from memory. Example 9–8 shows a typical application module custom method that correctly uses a secondary row set iterator for programmatic iteration, because the `EmpView1` view object instance in its data model may be bound to a user interface (either now or at a later time).

Example 9–8 Using a Secondary Row Set Iterator in an Application Module Custom Method

```
// Custom method in an application module implementation class
public void doSomeCustomProcessing() {
    ViewObject vo = getEmpView1();
    // create secondary row set iterator with system-assigned name
    RowSetIterator iter = vo.createRowSetIterator(null);
    while (iter.hasNext()) {
        Row r = iter.next();
        // Do something with the current row.
    }
    // close secondary row set iterator
    iter.closeRowSetIterator();
}
```

> **Note:** The same recommendation holds for custom code in a view object's implementation class that iterates its own default row set using that row set's default row set iterator.

There are two important reasons to follow this recommendation. Failing to do so can lead to confusion for the end user when the current row unexpectedly changes or it can introduce subtle business logic errors because the first or last row, or both rows get skipped.

- Confusing the end user by changing the current row unexpectedly

  The iterator bindings determine what row the end-user sees as the current row in the row set. If your own programmatic logic iterates through the row set using the same default row set iterator that the iterator binding uses, you may inadvertently change the current row the user has selected, leaving the user confused.

- Introducing subtle business logic errors by inadvertently skipping the first or last row

  Iterator bindings force their row set iterator to be on a valid row to guarantee that UI components display data when the row set is not empty. This has the side-effect of preventing your custom logic from navigating to the slot either before the first row or to the slot after the last row (when it is using the same row set iterator as an iterator binding). In concrete terms, this means that a typical `while (rowset.hasNext())` iteration loop will either be skipped or start by processing the second row instead of the first as shown in Example 9–9.

Example 9–9 Consequences of Using the Default Row Set Iterator

```
// Reset the default row set iterator to the slot before the first row
rowset.reset();
// If an iterator binding is bound to the same default row set iterator,
// then it has already forced it to navigate to the first row here instead
// of being on the slot before the first row.
//
// If the row set has only one row, the following will then return false
while (rowset.hasNext()) {
  // If the row set has more than one row, the first time through the loop
  // this call to next() will return the second row rather than the first
  // row as expected.
  Row curRow = rowset.next();
  // Do something with current row
}
```

9.8 Customizing Application Module Message Strings

The ADF application module does not have a resource bundle of its own and there is no design time in JDeveloper to associate one with the application module. However, if you do want to register a `.properties` file that contains your custom message strings, you can set a resource bundle definition in the definition class file that you generate for the application module.

9.8.1 How to Add a Resource Bundle to an Application Module

To generate the custom definition class file for the application module, use the Select Java Options dialog, which you open for the application module on the Java Classes

page of the application module overview editor. You can use this file to override the built-in framework method `finishedLoading()`.

Before you begin:

1. Create the desired application module, as described in Section 9.2.1, "How to Create an Application Module."
2. Create the `.properties` file and add the message key and message, as described in .Section 4.7, "Working with Resource Bundles."

   The `.properties` file you create can reference attribute properties by their fully-qualified package name and custom method exception messages. For example, you might define message keys and strings as follows:

   ```
   test.Order.Orderno_LABEL=Order Number
   INVALID=You have called the method foo in an invalid way.
   ```

3. If your resource bundle defines a message for a method exception message, the custom method should appear in the application module client interface, as described in Section 9.9.1, "How to Publish a Custom Method on the Application Module's Client Interface."

   For example, if you defined a message for the method `foo()` that to replace the exception message `INVALID`, your interface might define this method to invoke the message from the resource bundle as:

   ```
   public void foo() {
       ResourceBundleDef r = getResourceBundleDef();
       throw new JboException(r,"INVALID",null);
   }
   ```

To generate the definition class and override the finishedLoading method:

1. In the Application Navigator, double-click the application module.
2. In the overview editor, click the **Java** navigation tab.
3. On the Java Classes page, click the **Edit java options** button.
4. In the Select Java Options dialog, select **Generate Application Module Definition Class** and click **OK**.
5. In the overview editor, on the Java Classes page, click the linked file name of the application module definition class that you want to customize. JDeveloper opens the class file in the source editor.
6. From the main menu, choose **Source > Override Methods**.

   If the Source menu is not displayed in the main menu, be sure the definition class file is open and the source editor is visible.
7. In the Override Methods dialog, scroll the list to locate the finishedLoading() methods, select it, and click **OK**.
8. In the source editor for the definition class file, add the code to invoke your message bundle .properties file.

   For example, if your file is `MyAMBundle` in the `test` package, your code would look like:

   ```
   @Override
   protected void finishedLoading() {
       super.finishedLoading();
       PropertiesBundleDef pbd = new PropertiesBundleDef(this);
   ```

```
    pbd.setPropertiesFile("test.MyAMBundle");
    setResourceBundleDef(pbd);
}
```

In this example, the `finishedLoading()` method creates a message bundle definition and then sets the custom message bundle on the definition.

9. Save the file.

9.8.2 What Happens When You Add a Resource Bundle to an Application Module

When you generate a custom definition class for an application module, JDeveloper creates the file in the same directory as the component's XML component definition file. The default name for its custom Java file will be *AppModuleName*`DefImpl.java`.

Because you override the built-in `finishedLoading()` method in the definition class file, after the application is loaded at runtime, the framework will invoke the method and automatically load the named `.properties` file.

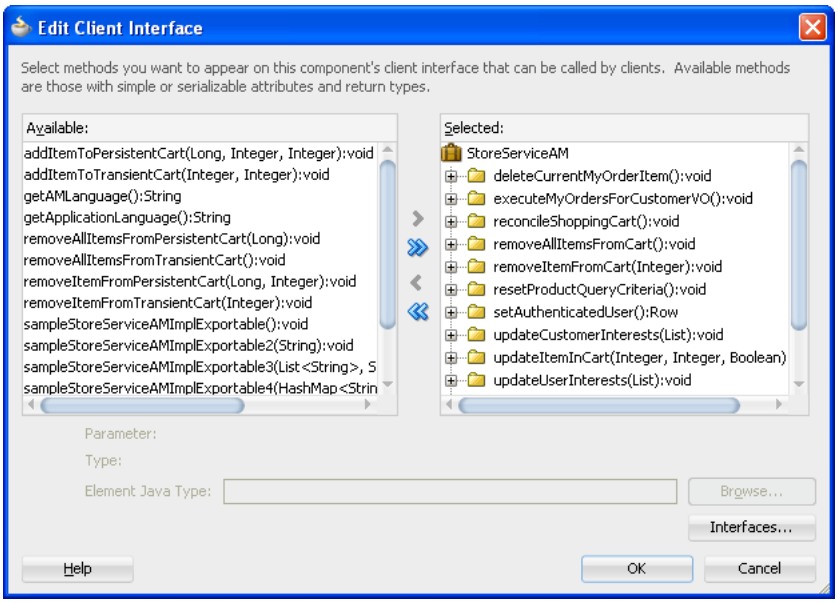
9.9 Publishing Custom Service Methods to UI Clients

When you add a `public` custom method to your application module class, if you want your application's UI to be able to invoke it, you need to include the method on the application module's UI client interface.

9.9.1 How to Publish a Custom Method on the Application Module's Client Interface

To include a public method from your application module's custom Java class on the client interface, use the Java Classes page of the overview editor for the application module, and then click the **Edit** icon in the **Client Interface** section of the page to display the Edit Client Interface dialog. Select one or more desired methods from the **Available** list and click the **Add** button to shuttle them into the **Selected** list. Then click **OK** to close the editor. Figure 9–21 shows multiple public methods added to the client interface.

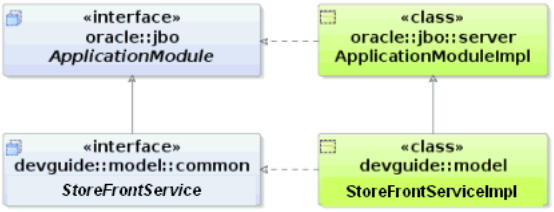
Figure 9–21 Public Methods Added to an Application Module's Client Interface



9.9.2 What Happens When You Publish Custom Service Methods

When you publish custom service methods on the client interface, as shown in Figure 9–22, JDeveloper creates a Java interface with the same name as the application module in the `common` subpackage of the package in which your application module resides. For an application module named `StoreServiceAM` in the `fodemo.model` package, this interface will be named `StoreServiceAM` and reside in the `fodemo.model.common` package. The interface extends the base `ApplicationModule` interface in the `oracle.jbo` package, reflecting that a client can access all of the base functionality that your application module inherits from the `ApplicationModuleImpl` class.

Figure 9–22 Custom Client Interface Extends the Base ApplicationModule Interface



As shown in Example 9–10, the `StoreServiceAM` interface includes the method signatures of all of the methods you've selected to be on the client interface of your application module.

Example 9–10 Custom Client Interface Based on Methods Selected in the Client Interface Panel

```
package fodemo.model.common;
import oracle.jbo.ApplicationModule;
// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---------------------------------------------------------------------
public interface StoreServiceAM extends ApplicationModule {
```

```
  void deleteCurrentMyOrderItem();
  void executeMyOrdersForCustomerVO();
  void userRegistrationCreate(String userType);
  void updateUserInterests(List pCategoryIds);
  void userRegistrationCreateAddress();
}
```

Each time you add or remove methods in the **Client Interface** section, the corresponding client interface file is updated automatically. JDeveloper also generates a companion client proxy class that is used when you deploy your application module for access by a remote client. For the `StoreServiceAM` application module in this example, the client proxy file is called `StoreServiceAMClient` and it is created in the `devguide.model.client` subpackage.

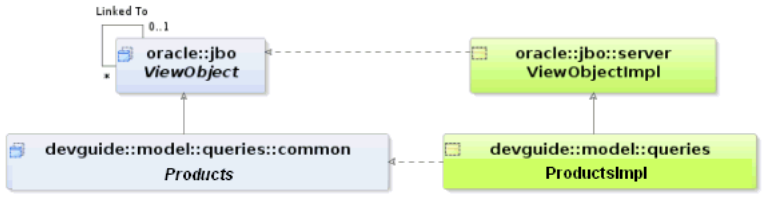
> **Note:** After adding new custom methods to the client interface, if your new custom methods do not appear to be available when you use JDeveloper's code insight context-sensitive statement completion, try recompiling the generated client interface. To do this, select the application module in the Application Navigator, select the source file for the interface of the same name in the Structure window, and choose **Rebuild** from the context menu. Consider this tip for new custom methods added to view objects and view rows as well.

9.9.3 How to Generate Client Interfaces for View Objects and View Rows

In addition to generating a client interface for your application module, it is also possible to generate strongly typed client interfaces for working with the other key client objects that you can customize. For example, you can open Java page in the overview editor for a view object, you can then expand the **Client Interface** section and the **Client Row Interface** section and add custom methods to the view object client interface and the view row client interface, respectively.

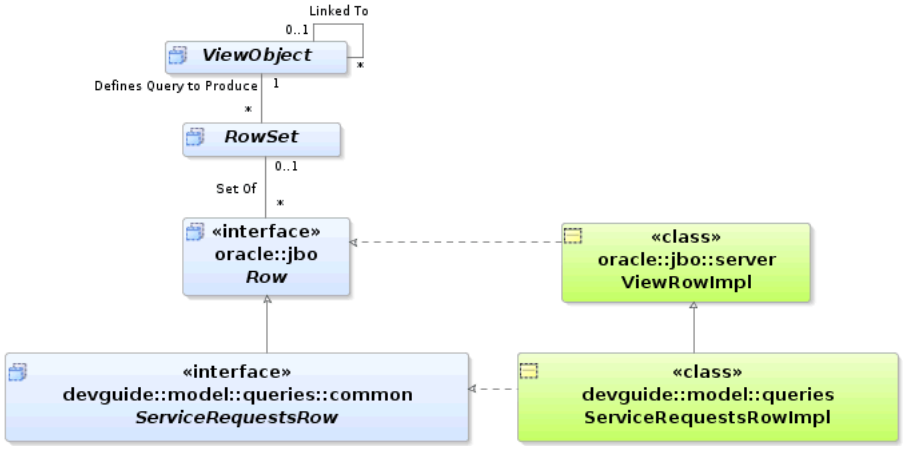
If for the `Products` view object in the `devguide.model.queries` package you were to enable the generation of a custom view object Java class and add one or more custom methods to the view object client interface, JDeveloper would generate the `ProductsImpl` class and `Products` interface, as shown in Figure 9–23. As with the application module custom interface, notice that it gets generated in the `common` subpackage.

Figure 9–23 Custom View Object Interface Extends the Base ViewObject Interface



Likewise, if for the same view object you were to enable the generation of a custom view row Java class and add one or more custom methods to the view row client interface, JDeveloper would generate the `ProductsRowImpl` class and `ProductsRow` interface, as shown in Figure 9–24.

Figure 9–24 Custom View Row Interface Extends the Base Row Interface



9.9.4 How to Test Custom Service Methods Using the Oracle ADF Model Tester

You can test the methods of your custom application module in the Oracle ADF Model Tester after you have published them on the client interface, as described in Section 9.9, "Publishing Custom Service Methods to UI Clients."

To test the service methods that you have published:

1. In the Application Navigator, expand the project containing the desired application module and view objects.
2. Right-click the application module and choose **Run**.

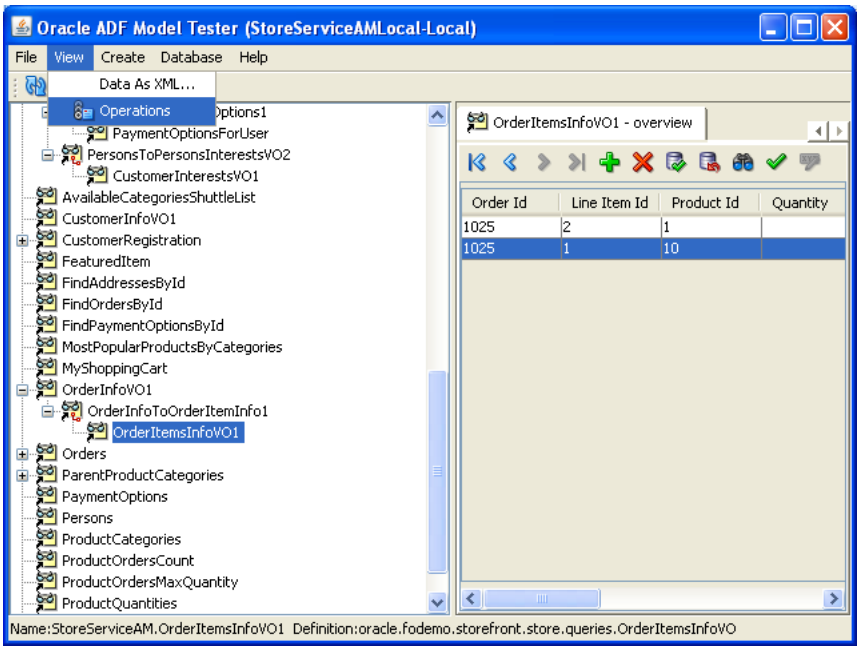
   Alternatively, choose **Debug** when you want to run the application in the Oracle ADF Model Tester with debugging enabled. JDeveloper opens the debugger process panel in the Log window and the various debugger windows. When debugging using the Oracle ADF Model Tester, you can use these windows to view status message and exceptions, step in and out of source code, and manage breakpoints.

   For information about receiving diagnostic messages specific to ADF Business Component debugging, see Section 6.3.8, "How to Enable ADF Business Components Debug Diagnostics."
3. To open the method testing panel for a method defined by a client interface, do one of the following:
   - In the data model tree, select the application module node and from the main menu choose **View - Operations** when you want to execute a method you published for the application module client interface. You can also double-click the application module node to display the method testing panel.
   - In the data model tree, select the desired view object node and from the main menu choose **View - Operations** when you want to execute a method you published on the client interface for a view object. You can also right-click the view object node and choose **Operations**.
4. To open the method testing panel for a method defined by a client row interface for a view object row, expand the data model tree, right-click the desired view object node and choose **Show Table**. Then in the overview panel for the view instance, select the desired row, and from the main menu, choose **View - Operations**.

Do not select a master view instance in the data model tree since view row operations are not permitted on master view objects. Always select a detail view instance or a view instance that is not specified in a master-detail hierarchy, as shown in Figure 9–25.

> **Tip:** In the case of a detail view instance, you can open the master view instance to navigate to the detail with the desired row. The Oracle ADF Model Tester automatically synchronizes the data displayed in the open overview panel with the master view instance that you navigate to.

Figure 9–25 Menu Selection for View Row Operations in the Oracle ADF Model Tester



5. In the method panel, select the desired service method from the dropdown list, enter values to pass as method parameters, and click **Execute**.

   Notice that the method testing panel displays the parameter names to help you identify where to enter the values to pass. This is particularly useful when the method signature defines multiple parameters of the same data type.

   You can view the return value (if any) and test result. The result displayed in the Oracle ADF Model Tester will indicate whether or not the method executed successfully.

9.9.5 What You May Need to Know About Method Signatures on the Client Interface

You can include any custom method in the client interface that obeys these implementation rules:

- If the method has a non-`void` return type, the type must be serializable.
- If the method accepts any parameters, all their types must be serializable.
- If the method signature includes a throws clause, the exception must be an instance of `JboException` in the `oracle.jbo` package.

In other words, all the types in its method signature must implement the `java.io.Serializable` interface, and any checked exceptions must be

`JboException` or its subclass. Your method can throw any unchecked exception — `java.lang.RuntimeException` or a subclass of it — without disqualifying the method from appearing on the application module's client interface.

Note that method signatures of type `java.util.List` are allowed as long as the implementing class for the interface is serializable. For example, `java.util.ArrayList` and `java.util.LinkedList` are both serializable implementing classes. The same requirement applies to element types within the collection. The ADF Business Components runtime will produce an error if you instantiate a class that implements the interface yet does not implement the `java.io.Serializable` interface.

Note: If the method you've added to the application module class doesn't appear in the **Available** list, first verify that it doesn't violate any of the method implementation rules. If it seems like it should be a legal method, try recompiling the application module class before visiting the overview editor for the application module again.

9.9.6 What You May Need to Know About Passing Information from the Data Model

The private implementation of an application module custom method can easily refer to any view object instance in the data model using the generated accessor methods. By calling the `getCurrentRow()` method on any view object, it can access the same current row for any view object that the client user interface sees as the current row. As a result, while writing application module business service methods, you may not need to pass in parameters from the client. This is true if you would be passing in values only from the current rows of other view object instances in the same application module's data model.

For example, the custom application module method in Example 9–11 accepts no parameters. Internally, the `createOrderItem()` method calls `getGlobals().getCurrentRow()` to access the current row of the `Globals` view object instance. Then it uses the strongly typed accessor methods on the row to access the values of the `Description` and `LineItemId` attributes to set them as the values of corresponding attributes in a newly created `OrderItem` entity object row.

Example 9–11 Using View Object Accessor Methods to Access a Current Row

```
// In StoreServiceAMImpl.java, createOrderItem() method
GlobalsRowImpl globalsRow = (GlobalsRowImpl)getGlobals().getCurrentRow();
newReq.setDescription(globalsRow.getDescription());
newReq.setLineItemId(globalsRow.getLineItemId());
```

9.10 Working Programmatically with an Application Module's Client Interface

After publishing methods on your application module's client interface, you can invoke those methods from a client.

9.10.1 How to Work Programmatically with an Application Module's Client Interface

To work programmatically with an application module's client interface, do the following:

- Cast `ApplicationModule` to the more specific client interface.

- Call any method on the interface.

> **Note:** For simplicity, this section focuses on working only with the custom application module interface; however, the same downcasting approach works on the client to use a `ViewObject` interface as a view object interface like `Orders` or a `Row` interface as a custom view row interface like `OrdersRow`.

Example 9–12 illustrates a `TestClientCustomInterface` class that puts these two steps into practice. You could also use the `main()` method of this class to test application module methods, as described in Section 9.7.5, "How to Test the Custom Application Module Using a Static Main Method." Here you use it to call all of the same methods from the client using the `StoreFrontService` client interface.

> **Note:** If you work with your application module using the default `ApplicationModule` interface in the `oracle.jbo` package, you won't have access to your custom methods. Make sure to cast the application module instance to your more specific custom interface like the `StoreFrontService` interface in this example.

The basic logic of Example 9–12 follows these steps:

1. Retrieve the total for order 1011.
2. Retrieve the name of the customer for order 1011.
3. Set the status of order 1011 to the value "CANCEL".
4. Create a new product supplying a null product name.
5. Create a new product with a product name and display its newly assigned product ID.

Example 9–12 Using the Custom Interface of an Application Module from the Client

```
package devguide.examples.appmodules;

import devguide.examples.appmodules.common.StoreFrontService;
import devguide.examples.entities.PersonsEOImpl;
import devguide.examples.entities.ProductsBaseEOImpl;

import oracle.jbo.ApplicationModule;
import oracle.jbo.JboException;
import oracle.jbo.Key;
import oracle.jbo.client.Configuration;
import oracle.jbo.domain.DBSequence;
import oracle.jbo.domain.Number;
import oracle.jbo.server.ApplicationModuleImpl;
import oracle.jbo.server.EntityDefImpl;

public class TestClientCustomInterface {
  public static void main(String[] args) {
    String        amDef = "devguide.model.StoreFrontService";
    String        config = "StoreFrontServiceLocal";
    /*
     * This is the correct way to use application custom methods
     * from the client, by using the application module's automatically-
     * maintained custom service interface.
```

```
     */
    //  Acquire instance of application module, cast to client interface
StoreFrontService service =
     (StoreFrontService)Configuration.createRootApplicationModule(amDef,config);
    String total = service.findOrderTotal(1011);
    System.out.println("Status of Order # 1011 = " + total);
    String customerName = service.findOrderCustomer(1011);
    System.out.println("Customer for Order # 1011 = " + customerName);
    try {
       service.updateOrderStatus(1011,"CANCEL");
    }
    catch (JboException ex) {
       System.out.println("ERROR: "+ex.getMessage());
    }
    long id = 0;
    try {
       id = service.createProduct(null, "NEW", "CLASS1");
    }
    catch (JboException ex) {
       System.out.println("ERROR: "+ex.getMessage());
    }
    id = service.createProduct("Canon PowerShot G9", "NEW", "CLASS1");
    System.out.println("New product created successfully with id = "+id);
    Configuration.releaseRootApplicationModule(am,true);
  }
}
```

Running the test client in Example 9–12 calls the custom methods of the client interface, and shows the following output:

```
Total for Order # 1011 = 99.99
Customer for Order # 1011 = John Chen
ERROR: JBO-27014: Attribute ProductName in ProductsBaseEO is required.
New product created successfully with id = 133
```

Notice that the first attempt to call `createProduct()` with a null for the product name raises an exception due to the built-in mandatory validation on the `Name` attribute of the `Product` entity object

9.10.2 What Happens at Runtime: How the Application Module's Client Interface is Accessed

If the client layer accessing your application module will be located in the same tier of the Java EE architecture, the application module will be deployed in what is known as *local mode*. In local mode, the client interface is implemented directly by your custom application module Java class. Typically, you access an application module in local mode when you need to:

- Access the application module in the web tier of a JavaServer Faces application
- Access the application module in the web tier of a JSP/Struts application
- Access the application module in the client tier (two-tier client-server style) for a Swing application

In contrast, when the client layer accessing your application module is located in a *different* tier of the Java EE architecture, the application module will be deployed in what is known as *remote mode*. In remote mode, the generated client proxy class implements your application module client interface on the client side, and the class handles all of the communications details of working with the remotely deployed application module service. You typically access the application module in remote

mode only when a thin-client Swing application needs to access the application module on a remote application server.

A unique feature of ADF Business Components is that by adhering to the interface-only approach for working with client service methods, you can be sure your client code works unchanged regardless of your chosen deployment mode. Even if you plan to work only in local mode, the Java EE development community favors the interface-based approach to working with services. Using application modules, it's extremely easy to follow this approach in your applications.

Note: Whether you plan to deploy your application modules in local mode or remote mode, as described in Section 9.9.5, "What You May Need to Know About Method Signatures on the Client Interface," the JDeveloper design time ensures that your custom interface methods will use serializable types. This allows you to switch at any time between local mode or remote mode, or to support both at the same time, with no code changes.

9.10.3 How to Access an Application Module Client Interface in a Fusion Web Application

The `Configuration` class in the `oracle.jbo.client` package makes it very easy to get an instance of an application module for *testing*. This eases writing test client programs like the test client program described in Section 36.11, "Regression Testing with JUnit" as part of the JUnit regression testing fixture.

Because it is easy, it is tempting for developers to use the class `createRootApplicationModule()` and `releaseApplicationModule()` methods *anywhere* to access an application module. However, for Fusion web applications you should resist this temptation because there is an even easier way.

When working with Fusion web applications using the ADF Model layer for data binding, JDeveloper configures a servlet filter in your user interface project called the `ADFBindingFilter`. It orchestrates the automatic acquisition and release of an appropriate application module instance based on declarative binding metadata, and ensures that the service is available to be looked up as a data control using a known action binding or iterator binding, specified by any page definition file in the user interface project. You may eventually want to read about the ADF binding container, data controls, page definition files, and bindings, as described in Chapter 13, "Using ADF Model in a Fusion Web Application." For now, it is enough to realize that you can access the application module's client interface from this `DCBindingContainer` by naming an ADF action binding or an ADF iterator binding. You can reference the binding context and call methods on the custom client interface in a JSF managed bean, as shown in Example 9–13 for an action binding and Example 9–14 for an iterator binding.

To access the custom interface of your application module using an action binding, follow these basic steps (as illustrated in Example 9–13):

1. Access the ADF binding container.
2. Find a named action binding. (Use the name of any available action binding in the page definition files of the user interface project.)
3. Get the data control by name from the action binding.
4. Access the application module data provider from the data control.
5. Cast the application module to its client interface.

6. Call any method on the client interface.

Example 9–13 Accessing the Application Module Client Interface in a JSF Backing Bean Using a Named Action Binding

```
package demo.view;
import oracle.fodemo.storefront.store.service.common.StoreServiceAM;
import oracle.adf.model.binding.DCBindingContainer;
import oracle.adf.model.binding.DCDataControl;
import oracle.jbo.ApplicationModule;
import oracle.jbo.uicli.binding.JUCtrlActionBinding;
public class YourBackingBean {
  public String commandButton_action() {
    // Example using an action binding to get the data control
    public String commandButton_action() {
    // 1. Access the binding container
    DCBindingContainer bc = (DCBindingContainer)getBindings();
    // 2. Find a named action binding
    JUCtrlActionBinding action =
                   (JUCtrlActionBinding)bc.findCtrlBinding("SomeActionBinding");
    // 3. Get the data control from the iterator binding (or method binding)
    DCDataControl dc  = action.getDataControl();
    // 4. Access the data control's application module data provider
    ApplicationModule am = (ApplicationModule)dc.getDataProvider();
    // 5. Cast the AM to call methods on the custom client interface
    StoreServiceAM service = (StoreServiceAM)am;
    // 6. Call a method on the client interface
    service.doSomethingInteresting();
    return "SomeNavigationRule";
  }
}
```

To access the custom interface of your application module using an iterator binding, follow these basic steps (as illustrated in Example 9–14):

1. Access the ADF binding container.
2. Find a named iterator binding. (Use the name of any iterator binding in the page definition files of the user interface project.)
3. Get the data control by name from the iterator binding.
4. Access the application module data provider from the data control.
5. Cast the application module to its client interface.
6. Call any method on the client interface.

Example 9–14 Accessing the Application Module Client Interface in a JSF Backing Bean Using a Named Iterator Binding

```
package demo.view;
import oracle.fodemo.storefront.store.service.common.StoreServiceAM;
import oracle.adf.model.binding.DCBindingContainer;
import oracle.adf.model.binding.DCDataControl;
import oracle.adf.model.binding.DCIteratorBinding;
import oracle.jbo.ApplicationModule;
public class YourBackingBean {
  public String commandButton_action() {
    // Example using an iterator binding to get the data control
    public String commandButton_action() {
    // 1. Access the binding container
    DCBindingContainer bc = (DCBindingContainer)getBindings();
```

```
    // 2. Find a named iterator binding
    DCIteratorBinding iter = bc.findIteratorBinding("SomeIteratorBinding");
    // 3. Get the data control from the iterator binding
    DCDataControl dc  = iter.getDataControl();
    // 4. Access the data control's application module data provider
    ApplicationModule am = (ApplicationModule)dc.getDataProvider();
    // 5. Cast the AM to call methods on the custom client interface
    StoreServiceAM service = (StoreServiceAM)am;
    // 6. Call a method on the client interface
    service.doSomethingInteresting();
    return "SomeNavigationRule";
  }
}
```

These backing bean examples depend on the helper method shown in Example 9–15.

Example 9–15 Helper Method for Backing Bean Class

```
public BindingContainer getBindings() {
        if (this.bindings == null) {
            FacesContext fc = FacesContext.getCurrentInstance();
            this.bindings =
                (BindingContainer)fc.getApplication().evaluateExpressionGet(fc,
                            "#{bindings}", BindingContainer.class);
        }
        return this.bindings;
    }
```

If you create the backing bean class by overriding a button that is declaratively bound to an ADF action, then JDeveloper will automatically generate this method in your class. Otherwise, you will need to add the helper method to your class yourself.

9.11 Overriding Built-in Framework Methods

The `ApplicationModuleImpl` base class provides a number of built-in methods that implement its functionality. While Appendix D, "Most Commonly Used ADF Business Components Methods" provides a quick reference to the most common code that you will typically write, use, and override in your custom application module classes, this section focuses on helping you understand the basic steps to override one of these built-in framework methods to augment the default behavior.

9.11.1 How to Override a Built-in Framework Method

To override a built-in framework method for an application module, use the Override Methods dialog, which you select for the application module Java class from the main menu.

Before you begin:

It may be helpful to have an understanding of the application module base class. For more information, see Section 9.11, "Overriding Built-in Framework Methods."

Complete this task:

Create the desired application modules as described in Section 9.2.1, "How to Create an Application Module."

To override an application module framework method:

1. In the Application Navigator, double-click the application module.

2. In the overview editor, click the **Java** navigation tab.
3. On the Java Classes page, click the linked file name of the application module Java class that you want to customize.

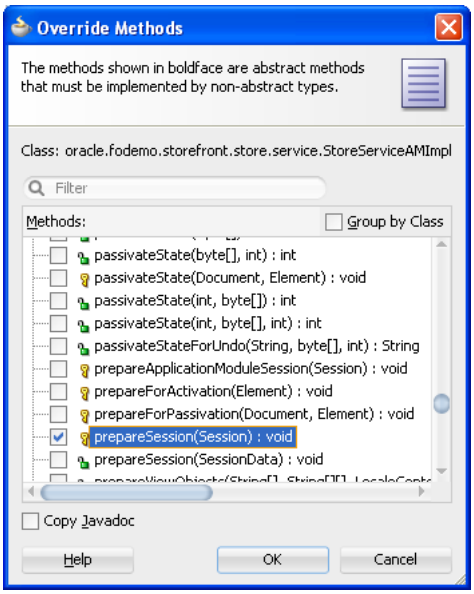
   JDeveloper opens the class file in the source editor.
4. From the main menu, choose **Source > Override Methods**.

   If the **Source** menu is not displayed, be sure that the desired Java class file is open and that the source editor is visible.
5. In the Override Methods dialog, scroll the list to locate the desired methods or type the first few letters of the method name to perform an incremental search.
6. Select one or more methods.

   The Override Methods dialog allows you to select any number of methods to override simultaneously.

   For example, if you wanted to override the application module's `prepareSession()` method to augment the default functionality when a new user session begins working with an application module service component for the first time, you would select the checkbox next to the `prepareSession(Session)` method, as shown in Figure 9–26.

Figure 9–26 Overriding a Built-in Framework Method

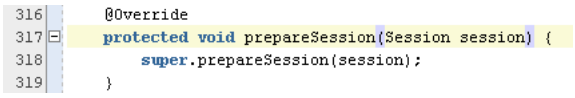


7. Click **OK**.

9.11.2 What Happens When You Override a Built-in Framework Method

When you dismiss the Override Methods dialog, you return to the source editor with the cursor focus on the overridden method, as shown in Figure 9–27. Notice that the method appears with a single line that calls `super.prepareSession()`. This is the syntax in Java for invoking the default behavior that the base class would have normally performed for this method. By adding code before or after this line in the custom application module class, you can augment the default behavior before or after the default functionality.

Figure 9–27 Source Editor Margin Gives Visual Feedback About Overridden Methods



Also notice that when you override a method using the Override Methods dialog, the source editor inserts the JDK `@Override` annotation just before the overridden method. This causes the compiler to generate a compile-time error if the method in the application module class does not match the signature of any method in the superclass.

Be careful when you add method names to your class to override a method in the superclass; you must have the signature *exactly* the same as the base class method you want to override. Be sure to add the `@Override` annotation just before the method. This way, if your method does not match the signature of any method in the superclass, the compiler will generate a compile-time error. Also, when you write code for a method instead of calling the superclass implementation, you should have a thorough understanding of what built-in code you are suppressing or replacing.

9.11.3 How to Override prepareSession() to Set Up an Application Module for a New User Session

Since the `prepareSession()` method is invoked by the application module when it is used for the first time by a new user session, it's a useful method to override in your custom application module class to perform setup tasks that are specific to each new user that uses your application module. Example 9–16 illustrates an overridden `prepareSession()` method in the `oracle.fodemo.storefront.adfextensions.FODApplicationModuleImpl` class that invokes a `setCurrentUserLanguage()` helper method to initialize the language used by the application.

Example 9–16 Initializing the Language to Use for Current User Session

```
public class FODApplicationModuleImpl extends ApplicationModuleImpl {

public static String preferredLanguage;
public static boolean isWebUser=false;
private String[] supportedLanguages = {"EN","JA","EL","FR","DE"};

    /**
     * @param session
     */
    @Override
    protected void prepareSession(Session session) {
        super.prepareSession(session);
        setCurrentUserLanguage();
    }

    private void setCurrentUserLanguage() {
        DBTransactionImpl dbti = (DBTransactionImpl)getDBTransaction();
        CallableStatement statement =
            dbti.createCallableStatement(("BEGIN " +
                                          "user_context_pkg.set_app_user_lang(?); " +
                                          "END;"), 0);
        try {
            statement.setString(1, getApplicationLanguage());
            statement.execute();
```

```
            } catch (SQLException sqlerr) {
                throw new JboException(sqlerr);
            } finally {
                try {
                    if (statement != null) {
                        statement.close();
                    }
                } catch (SQLException closeerr) {
                    throw new JboException(closeerr);
                }
            }
        }

        /**
         * @return
         */
        public String getApplicationLanguage(){
            String appLanguage = "EN";
            if (isWebUser){
            for (int index=0; index<supportedLanguages.length; index++){
                if (preferredLanguage.equals(supportedLanguages[index])){
                    appLanguage = preferredLanguage;
                break;
                }
            }
            } else{
                appLanguage = getAMLanguage();
            }
                return appLanguage;
        }

        /**
         * @return
         */
        public String getAMLanguage(){
            PropertyMetadata langProperty =
                              PropertyMetadata.findProperty("jbo.default.language");
            String amLanguage = langProperty.getProperty();
            return amLanguage.toUpperCase();
        }
    }
```

9.12 Calling a Web Service from an Application Module

In a service-oriented architecture, your Oracle ADF application module may need to take advantage of functionality offered by a web service that is not based on an application module. A web service can be implemented in any programming language and can reside on any server on the network. Each web service identifies the methods in its API by describing them in a standard, language-neutral XML format. This XML document, whose syntax adheres to the Web Services Description Language (WSDL), enables JDeveloper to understand the names of the web service's methods, as well as the data types of the parameters they might expect and their eventual return value.

> **Note:** Application modules can also be exposed as web services so that they can be consumed across modules of the deployed Fusion web application. For details about reusing ADF Business Components using external services, see Chapter 11, "Integrating Service-Enabled Application Modules."

JDeveloper's built-in web services wizards make this an easy task. Create a web service proxy class using the wizard, then call the service using method calls you add to a local Java object.

9.12.1 How to Call an External Service Programmatically

To call a web service from an application module, you create a web service proxy class for the service you want to invoke. A web service proxy is a generated Java class that represents the web service inside your application. It encapsulates the service URL of the web service and handles the lower-level details of making the call.

To work with a web service, you need to know the URL that identifies its WSDL document. If you have received the WSDL document as an email attachment, for example, and saved it to your local hard drive, the URL could be similar to:

```
file:///D:/temp/SomeService.wsdl
```

Alternatively, the URL could be an HTTP-based URL like:

```
http://someserver.somecompany.com/SomeService/SomeService.wsdl
```

Some web services make their WSDL document available by using a special parameter to modify the service URL. For example, a web service that expects to receive requests at the HTTP address of `http://someserver.somecompany.com/SomeService` might publish the corresponding WSDL document using the same URL with an additional parameter on the end, like this:

```
http://someserver.somecompany.com/SomeService?WSDL
```

Since there is no established standard, you will just need to know what the correct URL to the WSDL document is. With the URL information, you can then create a web service proxy class to call the service.

ADF Business Components services have URLs to the service of the following formats:

- On Integrated WebLogic Server, the URL has the format *http://host:port/EJB-context-root/@WebService-name?WSDL*, for example:

  ```
  http://localhost:8888/EJB-StoreFrontService/StoreFrontService?WSDL
  ```

- On Oracle Application Server, the URL has the format *http://host:port/context-root/@WebService-name?WSDL*, for example:

  ```
  http://localhost:8888/StoreFrontService/StoreFrontService?WSDL
  ```

The web service proxy class presents a set of Java methods that correspond to the web service's public API. By using the web service proxy class, you can call any method in the web service in the same way as you work with the methods of any other local Java class.

To call a web service from an application module using a proxy class, you perform the following tasks:

1. Create a web service proxy class for the web service. To create a web service proxy class for a web service that you need to call, use the Create Web Service Proxy wizard.
2. Implement the methods in the proxy class to access the desired web services.
3. Create an instance of the web service proxy class in your application module and invoke one or more methods on the web service proxy object.

9.12.1.1 Creating a Web Service Proxy Class to Programmatically Access the Service

To create a web service proxy class for a web service you need to call, use the Create Web Service Proxy wizard.

To create a web service proxy class to programmatically access the service:

1. In the Application Navigator, right-click the project in which you want to create the web service proxy, and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **Web Services** and then **Web Service Proxy Client**, and click **OK**.
3. On the Select Web Service Description page of the wizard, enter or choose a Java package name for the generated web service proxy class.
4. Enter the URL for the WSDL of the service you want to call in your application, and then tab out of the field.

   If the **Next** button does not enable, click **Why Not?** to understand what problem JDeveloper encountered when trying to read the WSDL document. If necessary, fix the problem after verifying the URL and repeat this step.
5. When the wizard displays **Next** enabled, then JDeveloper has recognized and validated the WSDL document. You can click **Next** and continue.
6. Continue through the pages of the wizard to specify details about the web service proxy. For more information about each page of the wizard, press F1 or click **Help**.
7. Click **Finish**.

9.12.1.2 Calling the Web Service Proxy Template to Invoke the Service

After you create the web service proxy, you must implement the methods in the proxy class to access the desired web services.

To call the web service proxy template to invoke the service:

1. Open the proxy client class, called *port_name*`Client.java`, in the source editor, and locate the comment `// Add your own code to call the desired methods`, which is in the main method.
2. Add the appropriate code to invoke the web service.
3. Deploy the full set of client module classes that JDeveloper has generated, and reference this class in your application.

9.12.1.3 Calling a Web Service Method Using the Proxy Class in an Application Module

After you've generated the web service proxy class, you can use it inside a custom method of your application module, as shown in Example 9–17. The method creates an instance of the web service proxy class and calls the web service method from the web service proxy class for the result.

Example 9–17 Web Service Proxy Class Calls Web Service Method

```
// In YourModuleImpl.java
public void performSomeApplicationTask(String symbol) throws Exception {
  // application-specific code here
   :
  // Create an instance of the web service proxy class
  StockQuoteServiceSoapHttpPortClient svc =
            new StockQuoteServiceSoapHttpPortClient();
  // Call a method on the web service proxy class and get the result
  QuoteInfo quote = svc.quoteForSymbol(symbol);
  float currentPrice = quote.getPrice();
  // more application-specific code here
}
```

9.12.2 What Happens When You Create the Web Service Proxy

JDeveloper generates the web service proxy class in the package you've indicated with a name that reflects the name of the web service port it discovered in the WSDL document. The web service port name might be a human-readable name like `StockQuoteService`, or could be a less-friendly name like `StockQuoteServiceSoapHttpPort`. The port name is decided by the developer that published the web service you are using. If the port name of the service were `StockQuoteServiceSoapHttpPort`, for example, JDeveloper would generate a web proxy class named `StockQuoteServiceSoapHttpPortClient`.

The web service proxy displays in the Application Navigator as a single, logical node called *`WebServiceName`*`Proxy`. For example, the node for the `StockQuoteService` web service would appear in the navigator with the name `StockQuoteServiceProxy`. As part of generating the proxy class, in addition to the main web service proxy class that you use to invoke the server, JDeveloper generates a number of auxiliary classes and interfaces. You can see these files in the Application Navigator under the *`WebServiceName`*`Proxy` node. The generated files are used as part of the lower-level implementation of invoking the web service.

The only auxiliary generated classes you need to reference are those created to hold structured web service parameters or return types. For example, imagine that the `StockQuoteService` web service has a `quoteForSymbol()` method that accepts one `String` parameter and returns a floating-point value indicating the current price of the stock. If the designer of the web service chose to return a simple floating-point number, then the web service proxy class would have a corresponding method like this:

```
public float quoteForSymbol(String symbol)
```

If instead the designer of the web service thought it useful to return multiple pieces of information as the result, then the service's WSDL file would include a named structure definition describing the multiple elements it contains. For example, assume that the service returns both the symbol name and the current price as a result. To contain these two data elements, the WSDL file might define a structure named `QuoteInfo` with an element named `symbol` of string type and an element named

price of floating-point type. In this situation, when JDeveloper generates the web service proxy class, the Java method signature would instead look like this:

```
public QuoteInfo quoteForSymbol(String symbol)
```

The QuoteInfo return type references one of the auxiliary classes that comprises the web service proxy implementation. It is a simple bean whose properties reflect the names and types of the structure defined in the WSDL document. In a similar way, if the web service accepts parameters whose values are structures or arrays of structures, then you will work with these structures in your Java code using the corresponding generated beans.

9.12.3 What Happens at Runtime: When You Call a Web Service Using a Web Service Proxy Class

When you invoke a web service from an application module, the web service proxy class handles the lower-level details of using the XML-based web services protocol described in SOAP. In particular, it does the following:

- Creates an XML document to represent the method invocation
- Packages any method arguments in XML
- Sends the XML document to the service URL using an HTTP POST request
- Unpackages the XML-encoded response from the web service

If the method you invoke has a return value, your code receives it as an appropriately typed object to work with in your application module code.

9.12.4 What You May Need to Know About Web Service Proxies

When you are implementing web service proxies in an application, you might want to use a try-catch block to handle web service exceptions or invoke an application module with a web service proxy class. The following sections contain additional information you might need to know about these and other features with regard to web service proxies.

9.12.4.1 Using a Try-Catch Block to Handle Web Service Exceptions

By using the generated web service proxy class, invoking a remote web service becomes as easy as calling a method in a local Java class. The only distinction to be aware of is that the web service method call could fail if there is a problem with the HTTP request involved. The method calls that you perform against a web service proxy should anticipate the possibility that the request might fail by wrapping the call with an appropriate try...catch block. Example 9–18 improves on the simpler example (shown in Section 9.12.1.3, "Calling a Web Service Method Using the Proxy Class in an Application Module") by catching the web service exception. In this case, it simply rethrows the error as a JboException, but you could implement more appropriate error handling in your own application.

Example 9–18 Wrapping Web Service Method Calls with a Try-Catch Block

```
// In YourModuleImpl.java
public void performSomeApplicationTask(String symbol) {
  // application-specific code here
  // :
  QuoteInfo quote = null;
  try {
    // Create an instance of the web service proxy class
```

```
    StockQuoteServiceSoapHttpPortClient svc =
              new StockQuoteServiceSoapHttpPortClient();
    // Call a method on the web service proxy class and get the result
    quote = svc.quoteForSymbol(symbol);
  }
  catch (Exception ex) {
    throw new JboException(ex);
  }
  float currentPrice = quote.getPrice();
  // more application-specific code here
}
```

9.12.4.2 Separating Application Module and Web Services Transactions

You will use some web services to access reference information. However, other services you call may modify data. This data modification might be in your own company's database if the service was written by a member of your own team or another team in your company. If the web service is outside your firewall, of course the database being modified will be managed by another company.

In either of these situations, it is important to understand that any data modifications performed by a web service you invoke will occur in their own distinct transaction, unrelated to the application module's current unit of work. For example, if you have invoked a web service that modifies data and then you later call `rollback()` to cancel the pending changes in the application module's current unit of work, this has no effect on the changes performed by the web service you called in the process. You may need to invoke a corresponding web service method to perform a compensating change to account for your rollback of the application module's transaction.

9.12.4.3 Setting Browser Proxy Information

If the web service you need to call resides outside your corporate firewall, you need to ensure that you have set the appropriate Java system properties to configure the use of an HTTP proxy server. The Java system properties to configure are:

- `http.proxyHost` — Set this to the name of the proxy server.
- `http.proxyPort` — Set this to the HTTP port number of the proxy server (often 80).
- `http.nonProxyHosts` — Optionally set this to a vertical-bar-separated list of servers not requiring the user of a proxy server (for example, `localhost|127.0.0.1|*.`*`yourcompany`*`.com`).

Within JDeveloper, you can configure an HTTP proxy server on the Web Browser and Proxy page of the Preferences dialog. When you run your application, JDeveloper includes appropriate `-D` command-line options to set these three system properties based on the settings you've indicated in this dialog.

9.12.4.4 Invoking Application Modules with a Web Service Proxy Class

If you use a web service proxy class to invoke an Oracle ADF service-based application module, you lose the ability to optimize the call when the calling component and the service you are calling are colocated. As an alternative, you can use the service interface approach described in Chapter 11, "Integrating Service-Enabled Application Modules."

10

Sharing Application Module View Instances

This chapter describes how to organize your data model project to most efficiently utilize read-only data accessed from lookup tables or other static data source, such as a flat file.

This chapter includes the following sections:

- Section 10.1, "About Shared Application Modules"
- Section 10.2, "Sharing an Application Module Instance"
- Section 10.3, "Defining a Base View Object for Use with Lookup Tables"
- Section 10.4, "Accessing View Instances of the Shared Service"
- Section 10.5, "Testing View Object Instances in a Shared Application Module"

10.1 About Shared Application Modules

Web applications often utilize data that is required across sessions and does not change very frequently. An example of this type of *static data* might be displayed in the application user interface in a lookup list. Each time your application accesses the static data, you could incur an unnecessary overhead when the static data caches are repopulated from the database for each application session on every request. In order to optimize performance, a common practice when working with ADF Business Components is to cache the shared static data for reuse across sessions and requests.

10.1.1 Shared Application Module Use Cases and Examples

View accessors in ADF Business Components are value accessor objects that point from an entity object attribute (or view object) to a destination view object or shared view instance in the same application workspace.

This ability to access view objects in different application modules makes view accessors particularly useful for:

- Validation rules that you set on the attributes of an entity object. For example, when the end user fills out a registration form, individual validation rules can verify the title, marital status, and contact code against lookup table data queried by view instances of the shared application module.
- List of Value (LOV) that you enable for the attribute of any view object. For example, to display a list of values to the end user at runtime.

10.1.2 Additional Functionality for Shared Application Modules

You may find it helpful to understand other ADF features before you start working with shared application modules. Following are links to other functionality that may be of interest.

- For details about configuring application module instances to improve runtime performance, see Chapter 43, "Application State Management" and Chapter 44, "Tuning Application Module Pools and Connection Pools."
- For a quick reference to the most common code that you will typically write, use, and override in your custom application module classes, see Appendix D, "Most Commonly Used ADF Business Components Methods."
- For API documentation related to the `oracle.jbo` package, see the following Javadoc reference document:
  - *Oracle Fusion Middleware Java API Reference for Oracle ADF Model*

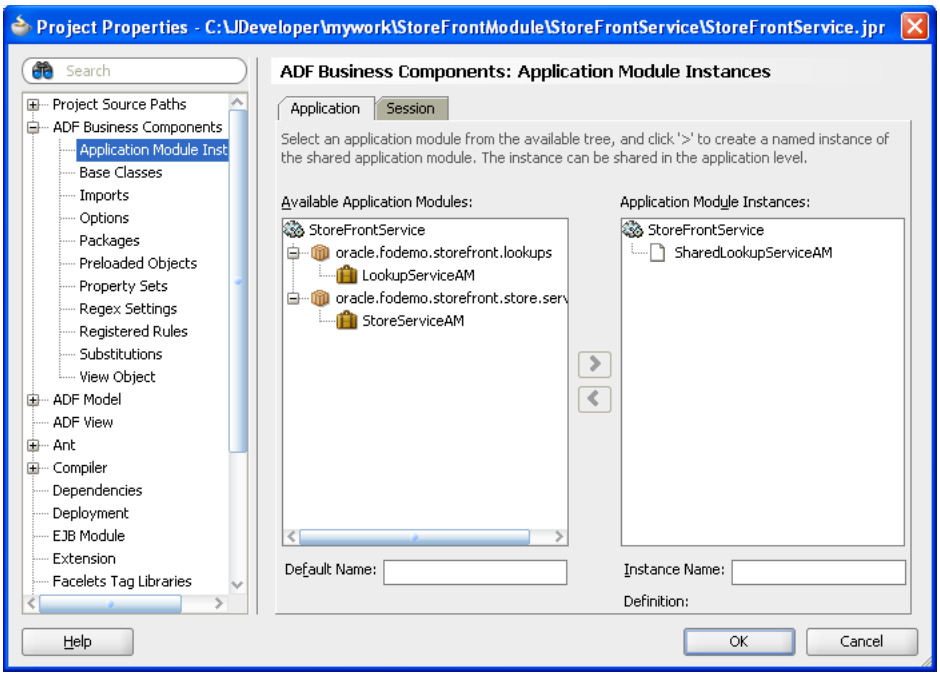
10.2 Sharing an Application Module Instance

Declarative support for shared data caches is available in JDeveloper through the Project Properties dialog. Creating a shared application module allows requests from multiple sessions to share a single application module instance which is managed by an application pool for the lifetime of the web server virtual machine.

> **Best Practice:** Use a shared application module to group view instances when you want to reuse lists of static data across the application. The shared application module can be configured to allow any user session to access the data or it can be configured to restrict access to just the UI components of a single user session. For example, you can use a shared application module to group view instances that access lookup data, such as a list of countries. The use of a shared application module allows all shared resources to be managed in a single place and does not require a scoped managed bean for this purpose.

As shown in Figure 10–1, the Project Properties dialog lets you specify application-level or session-level sharing of the application module's data model. In the case of *application-level sharing*, any HTTP user session will be able to access the same view instances contained in the shared application module. In contrast, the lifecycle of the *session-level shared* application module extends to an application module session (`SessionImpl`) that is in use by a single HTTP user session and applies to a single root application module. In this case, each distinct *root* application module used by a given HTTP user session will get its own distinct instance of a session-scoped shared application module. In other words, distinct root application modules used by the same HTTP session do not share data in a session-scoped shared application module.

Figure 10–1 Project Properties Dialog Defines Shared Application Module Instance



When you create the data model for the application module that you intend to share, be sure that the data in cached row sets will not need to be changed either at the application level or session level. For example, in the application-level shared application module, view instances should query only static data such as state codes or currency types. If a view object instance queries data that depends on the current user, then the query can be cached at the session level and shared by all components that reference the row-set cache. For example, the session-level shared application module might contain a view instance with data security that takes a manager as the current user to return the list of direct reports. In this case, the cache of direct reports would exist for the duration of the manager's HTTP user session. The ADF Business Components application module pool will recreate the session-scoped application module should an HTTP user session be assigned a recycled application module from the pool. This ensures that the duration of the session-scoped application module is tied to the HTTP session for as long as the HTTP session is able to continue to use the same root application module instance. Note that the cache of direct reports of the session-level shared application module cannot be accessed across distinct root application modules.

10.2.1 How to Create a Shared Application Module Instance

To create a shared application module instance, use the Project Properties dialog. You define a logical name for a distinct, separate root application module that will hold your application's read-only data.

Before you begin:

Create the application module that you will share as described in Section 9.2.1, "How to Create an Application Module."

To create a shared application module instance:

1. In the Application Navigator, right-click the project in which you want to create the shared application module and choose **Project Properties**.

2. In the Project Properties dialog, expand **ADF Business Components** and select **Application Module Instances**.
3. On the ADF Business Components: Application Module Instances page, select one of these tabs:
   - When you want to define the shared application module for the context of the application, select the **Application** tab.
   - When you want to define the shared application module for the context of the current user session, select the **Session** tab
4. In the **Available Application Modules** list, select the desired application module and shuttle it to the **Application Module Instances** list.
5. Assign the application module a unique instance name.

   The shared application module instance (of either scope) must have a unique instance name. Supplying a meaningful name will also help to clarify which shared application module instance a given usage is referencing.
6. Click **OK**.

10.2.2 What Happens When You Define a Shared Application Module

JDeveloper automatically creates the *AppModuleName*`Shared` configuration when you create an application module. The presence of this configuration in the `bc4j.xcfg` file informs JDeveloper that the application module is a candidate to be shared, and allows JDeveloper to display the application module in the **Available Application Modules** list of the Project Properties dialog's Application Module Usage page.

The *AppModuleName*`Shared` configuration sets these properties on the application module to enable sharing and help to maintain efficient use of the shared resource at runtime:

- `jbo.ampool.isuseexclusive` is set to `false` to specify that requests from multiple sessions can share a single instance of the application module, which is managed by the application pool for the lifetime of the web server virtual machine. When you do not enable application module sharing, JDeveloper sets the value `true` to repopulate the data caches from the database for each application session on every request.
- `jbo.ampool.maxpoolsize` is set to `1` (one) to specify that only a single application module instance will be created for the ADF Business Components application module pool. This setting enforces the efficient use of the shared application module resource and prevents unneeded multiple instances of the shared application module from being created at runtime.

You can view the shared application module's configuration by choosing **Configurations** from the context menu on the application module in the Application Navigator. JDeveloper saves the `bc4j.xcfg` file in the `./common` subdirectory relative to the application module's XML component definition. If you remove the configuration or modify the values of the `jbo.ampool` runtime properties (`isuseexclusive`, `maxpoolsize`), the application module will not be available to use as a shared application module instance.

For example, if you look at the `bc4j.xcfg` file in the `./src/oracle/fodemo/storefront/lookups/common` directory of the Fusion Order Demo application's `StoreFrontService` project, you will see the two named configurations for the `LookupServiceAM` application module, as shown in

Example 10–1. Specifically, the `LookupServiceAMShared` configuration sets the `jbo.ampool` runtime properties on the shared application module instance. For more information about the ADF Business Components application module pooling and runtime configuration of application modules, see Chapter 44, "Tuning Application Module Pools and Connection Pools."

Example 10–1 LookupServiceAMShared Configuration in the bc4j.xcfg File

```
<BC4JConfig version="11.1" xmlns="http://xmlns.oracle.com/bc4j/configuration">
   <AppModuleConfigBag
      ApplicationName="oracle.fodemo.storefront.lookups.LookupServiceAM">
      <AppModuleConfig DeployPlatform="LOCAL"
           JDBCName="FOD"
           jbo.project="StoreFrontService"
           name="LookupServiceAMLocal"
           ApplicationName="oracle.fodemo.storefront.lookups.LookupServiceAM">
           <Database jbo.locking.mode="optimistic"/>
           <Security
            AppModuleJndiName="oracle.fodemo.storefront.lookups.LookupServiceAM"/>
       </AppModuleConfig>
       <AppModuleConfig DeployPlatform="LOCAL"
           JDBCName="FOD"
           jbo.project="StoreFrontService"
           name="LookupServiceAMShared"
           ApplicationName="oracle.fodemo.storefront.lookups.LookupServiceAM">
           <AM-Pooling jbo.ampool.dynamicjdbccredentials="false"
                       jbo.ampool.isuseexclusive="false"
                       jbo.ampool.maxpoolsize="1"
                       jbo.ampool.resetnontransactionalstate="false"/>
           <Database jbo.locking.mode="optimistic"/>
           <Security
            AppModuleJndiName="oracle.fodemo.storefront.lookups.LookupServiceAM"/>
       </AppModuleConfig>
   </AppModuleConfigBag>
</BC4JConfig>
```

Because the shared application module can be accessed by any data model project (based on ADF Business Components) in the same application workspace, JDeveloper maintains the scope of the shared application module in the ADF Business Components project configuration file (`.jpx`). This file is saved in the `src` directory of the project. For example, if you look at the `StoreFrontService.jpx` file in the `./src` directory of the Fusion Order Demo application's `StoreFrontService` project, you will see that the `SharedLookupService` application module's usage definition specifies `SharedScope = 2`, corresponding to application-level sharing, as shown in Example 10–2. An application module that you set to session-level sharing will show `SharedScope = 1`.

Example 10–2 Application Module Usage Configuration in the .jpx File

```
<JboProject
   xmlns="http://xmlns.oracle.com/bc4j"
   Name="StoreFrontService"
   SeparateXMLFiles="true"
   PackageName="">
   . . .
   <AppModuleUsage
      Name="SharedLookupService"
      FullName="oracle.fodemo.storefront.lookups.LookupServiceAM"
      ConfigurationName="oracle.fodemo.storefront.lookups.LookupServiceAMShared"
```

```
      SharedScope="2"/>
</JboProject>
```

10.2.3 What You May Need to Know About Design Time Scope of the Shared Application Module

Defining the shared application module in the Project Properties dialog makes the application module's data model available to other data model projects of the same application workspace only. When you want to make the data model available beyond the application workspace, you can publish the data model as an ADF Library, as described in Chapter 38, "Reusing Application Components."

When viewing a data control usage from the `DataBindings.cpx` file in the Structure window, do not set the **Configuration** property to a shared application module configuration. By default, for an application module named ***AppModuleName***, the Property Inspector will list the configurations named ***AppModuleName*Shared** and ***AppModuleName*Local**. At runtime, Oracle Application Development Framework (Oracle ADF) uses the shared configuration automatically when you configure an application as a shared application module, but the configuration is not designed to be used by an application module data control usage. For more information about data control usage, see Section 13.5, "Working with the DataBindings.cpx File."

10.2.4 What You May Need to Know About the Design Time Scope of View Instances of the Shared Application Module

You define view accessors on the business component definition for the data model project that will permit access to view instances of the shared application module. The view accessor lets you point from an entity object or view object definition in one data model project to a view object definition or view instance in a shared application module. For details about creating view accessors for this purpose, see Section 10.4, "Accessing View Instances of the Shared Service."

10.2.5 What You May Need to Know About Managing the Number of Shared Query Collections

Similar to the way application module pooling works in ADF Business Components, shared query collections are stored in a query collection pool. To manage the query collection pool, the ADF Business Components framework removes query collections based on a maximum idle time setting. This behavior limits the growth of the cache and prevents rarely-used query collections from occupying memory space.

As in application module and connection pooling, a query collection pool monitor wakes up after a user-specified sleep interval and then initiates the cleanup operation. Any query collection that exceeds the maximum idle time (length of time since it was last used), will be removed from the pool.

You can change the default values for the maximum idle time for the shared query collection (default is 900000 ms/15 min) and the sleep period for its pool monitor (default is 1800000 ms/30 min). To configure these values, open the Edit Business Components Configuration dialog, select the ***AppModuleName*Shared** configuration, and set these properties in the Properties page of the editor:

- `jbo.qcpool.monitorsleepinterval` the time (ms) that the shared query collection pool monitor should sleep between pool checks.

- `jbo.qcpool.maxinactiveage` the maximum amount of time (ms) that a shared query collection may remain unused before it is removed from the pool.

10.2.6 What You May Need to Know About Shared Application Modules and Connection Pooling

The default connection behavior for all application modules is to allow each root application module to have its own database connection. When your application defines more than one shared application module, you can change the default to optimize database connection usage by defining a named transaction for each root application module to use. The transaction name is an arbitrary string that you set on the `jbo.shared.txn` property in the Properties page of the editor for the `bc4j.xcfg` file of the root application module. At runtime, the root application modules with the same `jbo.shared.txn` property setting (identified by the string you supply) will share the same database connection and entity cache. This optimization can reduce the database resources that the application uses and is particularly useful in shared application modules cases because they are read only and have longer life than transactional application modules.

Currently, the application module configuration parameter `jbo.doconnectionpooling=true` is not supported for use with shared application modules. This feature is available to configure non-shared application modules when it is desirable to release JDBC connection objects to the database connection pool.

This feature is intentionally not supported for shared application modules to prevent decreases in performance that would result from managing state for shared access. Instead, the default use of `jbo.doconnectionpooling=false` is enforced.

The default connection pooling configuration ensures that each shared application module instance holds onto the JDBC connection object that it acquires from the pool until the application module instance is removed from the application module pool. For more information about the `jbo.doconnectionpooling` parameter and connection pool behavior, see Section 44.2.6, "What You May Need to Know About How Database and Application Module Pools Cooperate."

10.3 Defining a Base View Object for Use with Lookup Tables

When your application needs to display static data, you can define a shared application module with view instances that most likely will access lookup tables. A lookup table is a static, translated list of data to which the application refers. Lookup table data can be organized in the database in various ways. While it is possible to store related lookup data in separate tables, it is often convenient to combine all of the lookup information for your application within a single table. For example, a column `LOOKUP_TYPE` created for the `ORDERS_LOOKUPS` table would serve to partition one table that might contain diverse codes such as `FWK_TBX_YES_NO` for the values yes and no, `FWK_TBX_COUNTRY` for country names, and `FWK_TBK_CURRENCY` for the names of national currencies.

When your database schema organizes lookup data in a single database table, you want to avoid creating individual queries for each set of data. Instead, you will use the overview editor to define a single, base view object that maps the desired columns of the lookup table to the view object attributes you define. Since only the value of the `LOOKUP_TYPE` column will need to change in the query statement, you can add view criteria on the view object definition to specify a `WHERE` clause that will set the `LOOKUP_TYPE` value. In this way, your application encapsulates access to the lookup table data in a single view object definition that will be easy to maintain when a

LOOKUP_TYPE value changes or your application needs to query additional lookup types.

10.3.1 How to Create a Base View Object Definition for a Lookup Table

The base view object that queries columns of the lookup table will be a read-only view object, since you do not need to handle updating data or require any of the benefits provided by entity-based view objects. (For a description of those benefits, see Section 5.1, "About View Objects.")

> **Note:** While read-only view objects you create to access lookup tables are ideal for inclusion in a shared application module, if you intend to share the view object in a shared application module instance, you must create the view object in the same package as the shared application module.

To create a read-only view object, use the Create View Object wizard, which is available from the New Gallery.

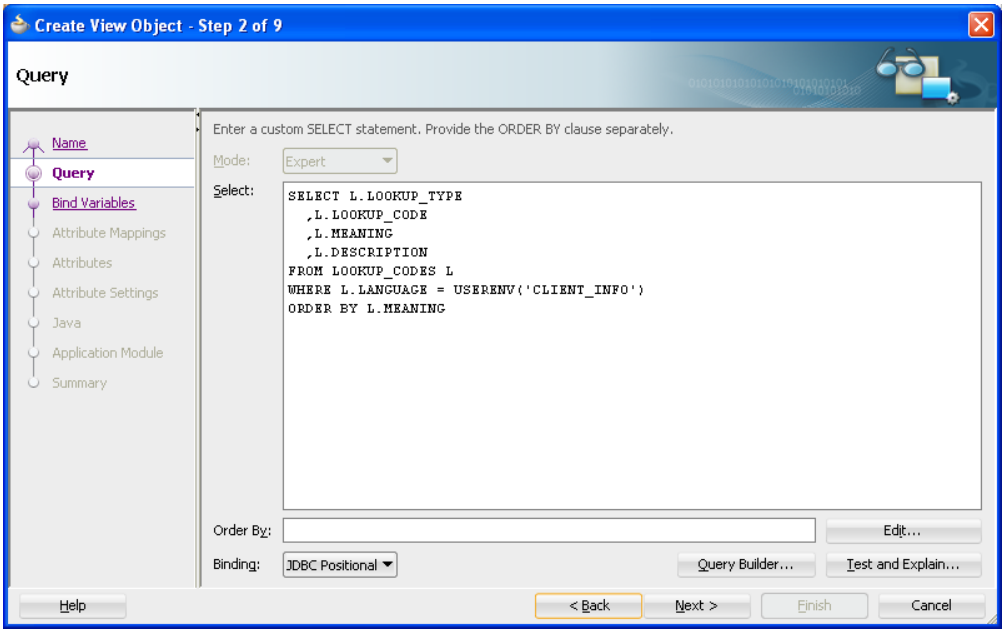
To create a base view object for a lookup table:

1. In the Application Navigator, locate the shared application module you created in which you want to create the view object, right-click its package node, and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then select **View Object**, and click **OK**.
3. In the Create View Object wizard, on the Name page, enter a package name and a view object name.

   When naming the package, consider creating a separate package for the lookup.
4. Select **SQL query** to indicate that you want this view object to manage data with read-only access and click **Next**.
5. On the Query page, enter your SQL statement directly into the **Query Statement** box.

   Your query names the columns of the lookup table, similar to the SQL statement shown in Figure 10–2 to query the LOOKUP_CODE, MEANING, and DESCRIPTION columns in the LOOKUP_CODES table.

Figure 10–2 Create View Object Wizard, SQL Query for Lookup Table

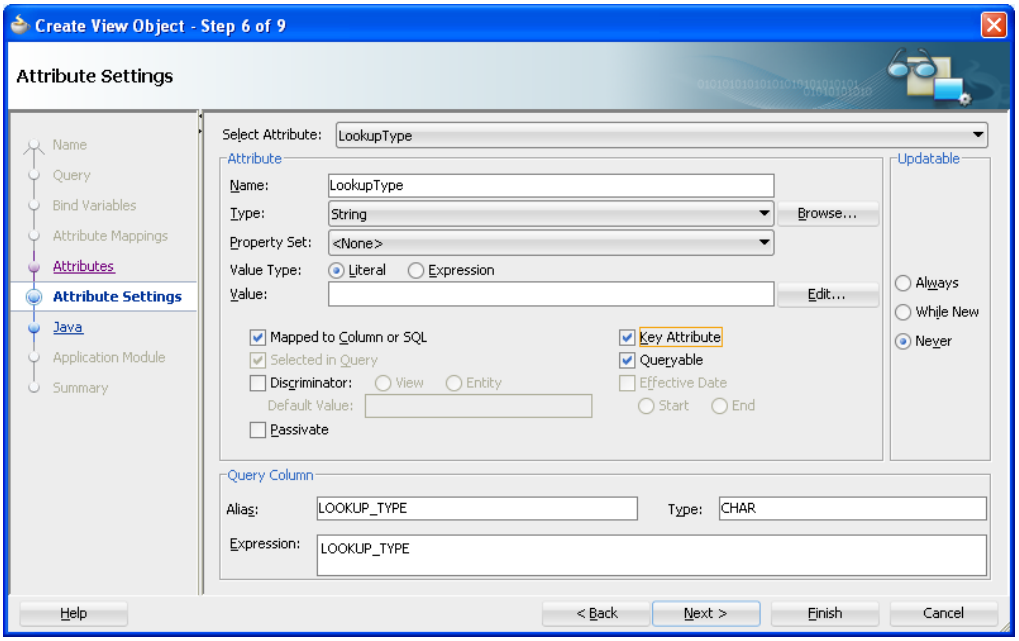


6. After entering the query statement, no other changes are required. Click **Next**.
7. On the Bind Variables page, click **Next**.
8. On the Attribute Mappings page, note the mapped view object attribute names displayed and click **Next**.

   By default, the wizard creates Java-friendly view object attribute names that correspond to the `SELECT` list column names.
9. On the Attribute Settings page, from the **Select Attribute** dropdown, select the attribute that corresponds to the primary key of the queried table and then enable the **Key Attribute** checkbox.

   Because the read-only view object is not based on an entity object, the Create View Object wizard does not define a key attribute by default. Failure to define the key attribute can result in unexpected runtime behavior for ADF Faces components with a data control based on the read-only view object collection. In the case of read-only view objects, define the key attribute, as shown in Figure 10–3.

Figure 10–3 Create View Object Wizard, Attribute Settings Page



10. If you want to rename individual attributes to use names that might be more appropriate, from the **Select Attributes** dropdown, choose the attribute and enter the desired name in the **Name** field. When you are finished, click **Next**.

    For example, the Fusion Order Demo application renames the default attributes `LookupType` and `LookupCode` to `Type` and `Value` respectively. Changes you make to the view object definition will not change the underlying query.

11. On the Java page, click **Next**.

12. On the Application Module page, do not add an instance of the view object to the application module data model. Click **Finish**.

    The shared application module data model will include view instances based on view criteria that you add to the base view object definition. In this way, you do not need to create an individual view object to query each `LOOKUP_TYPE` value. For details about adding the view object instances to the data model, see Section 9.2.3.2, "Adding Master-Detail View Object Instances to an Application Module."

10.3.2 What Happens When You Create a Base View Object

When you create the view object definition for the lookup table, JDeveloper first describes the query to infer the following from the columns in the `SELECT` list:

- The Java-friendly view attribute names (for example, `LookupType` instead of `LOOKUP_TYPE`)

  By default, the wizard creates Java-friendly view object attribute names that correspond to the `SELECT` list column names.

- The SQL and Java data types of each attribute

JDeveloper then creates the XML component definition file that represents the view objects's declarative settings and saves it in the directory that corresponds to the name of its package. For example, the XML file created for a view object named

LookupsBaseVO in the lookups package is ./lookups/LookupsBaseVO.xml under the project's source path.

To view the view object settings, expand the desired view object in the Application Navigator, select the XML file under the expanded view object, and open the Structure Window. The Structure window displays the list of definitions, including the SQL query and the properties of each attribute. To open the file in the editor, double-click the corresponding **.xml** node. As shown in Example 10–3, the LookupsBaseVO.xml file defines one <SQLQuery> definition and one <ViewAttribute> definition for each mapped column. Without a view criteria to filter the query results, the view object query returns the LOOKUP_CODE, LOOKUP_MEANING, and LOOKUP_DESCRIPTION and maps them to view instance attribute values for Value, Name, and Description respectively. Key attributes are defined to ensure proper row set navigation when the base view object collection is bound to an ADF Faces component.

Example 10–3 LookupsBaseVO SQL Query and Attribute Mapping Definition

```
<ViewObject
    xmlns="http://xmlns.oracle.com/bc4j"
    Name="LookupsBaseVO"
    BindingStyle="OracleName"
    CustomQuery="true"
    PageIterMode="Full"
    UseGlueCode="false"
    FetchMode="FETCH_AS_NEEDED"
    FetchSize="500">
  <SQLQuery>
    <![CDATA[SELECT L.LOOKUP_TYPE
     ,L.LOOKUP_CODE
     ,L.MEANING
     ,L.DESCRIPTION
     FROM LOOKUP_CODES L
     WHERE L.LANGUAGE = USERENV('CLIENT_INFO')
     ORDER BY L.MEANING]]>
  </SQLQuery>
  <DesignTime>
    <Attr Name="_codeGenFlag2" Value="Access|VarAccess"/>
    <Attr Name="_isExpertMode" Value="true"/>
  </DesignTime>
  <ViewAttribute
    Name="Type"
    IsUpdateable="false"
    IsPersistent="false"
    IsNotNull="true"
    PrecisionRule="true"
    Precision="255"
    Type="java.lang.String"
    ColumnType="VARCHAR2"
    AliasName="LOOKUP_TYPE"
    Expression="LOOKUP_TYPE"
    SQLType="VARCHAR">
    <DesignTime>
      <Attr Name="_DisplaySize" Value="30"/>
    </DesignTime>
    ...
  </ViewAttribute>
  <ViewAttribute
    Name="Value"
    IsUpdateable="false"
    IsPersistent="false"
```

```
    IsNotNull="true"
    PrecisionRule="true"
    Precision="30"
    Type="java.lang.String"
    ColumnType="VARCHAR2"
    AliasName="LOOKUP_CODE"
    Expression="LOOKUP_CODE"
    SQLType="VARCHAR">
    <DesignTime>
      <Attr Name="_DisplaySize" Value="30"/>
    </DesignTime>
    ...
  </ViewAttribute>
  <ViewAttribute
    Name="Name"
    IsUpdateable="false"
    IsPersistent="false"
    IsNotNull="true"
    PrecisionRule="true"
    Precision="80"
    Type="java.lang.String"
    ColumnType="VARCHAR2"
    AliasName="MEANING"
    Expression="MEANING"
    SQLType="VARCHAR">
    <DesignTime>
      <Attr Name="_DisplaySize" Value="80"/>
    </DesignTime>
    ...
  </ViewAttribute>
  <ViewAttribute
    Name="Description"
    IsUpdateable="false"
    IsPersistent="false"
    PrecisionRule="true"
    Precision="240"
    Type="java.lang.String"
    ColumnType="VARCHAR2"
    AliasName="DESCRIPTION"
    Passivate="true"
    Expression="DESCRIPTION"
    SQLType="VARCHAR">
    <DesignTime>
      <Attr Name="_DisplaySize" Value="240"/>
    </DesignTime>
    ...
  </ViewAttribute>
  <AttrArray Name="KeyAttributes">
    <Item Value="Type"/>
    <Item Value="Value"/>
  </AttrArray>
. . .
</ViewObject>
```

10.3.3 How to Define the WHERE Clause of the Lookup View Object Using View Criteria

You create named view criteria definitions in the data model project when you need to filter view object results. View criteria that you define at design time can participate in UI scenarios that require filtering of data.

Use the Edit View Criteria dialog to create the view criteria definition for the lookup base view object you defined to query the lookup table. The editor lets you build a `WHERE` clause using attribute name instead of the target view object's corresponding SQL column names. The resulting definition will include:

- One view criteria row consisting of one view criteria group, with a single view criteria item used to define the lookup view object's `Type` attribute.
- The view criteria item will consist of an `Type` attribute name, the Equal operator, and the value of the `LOOKUP_TYPE` that will filter the query results.

Because a single view criteria is defined, no logical conjunctions are needed to bracket the `WHERE` clause conditions.

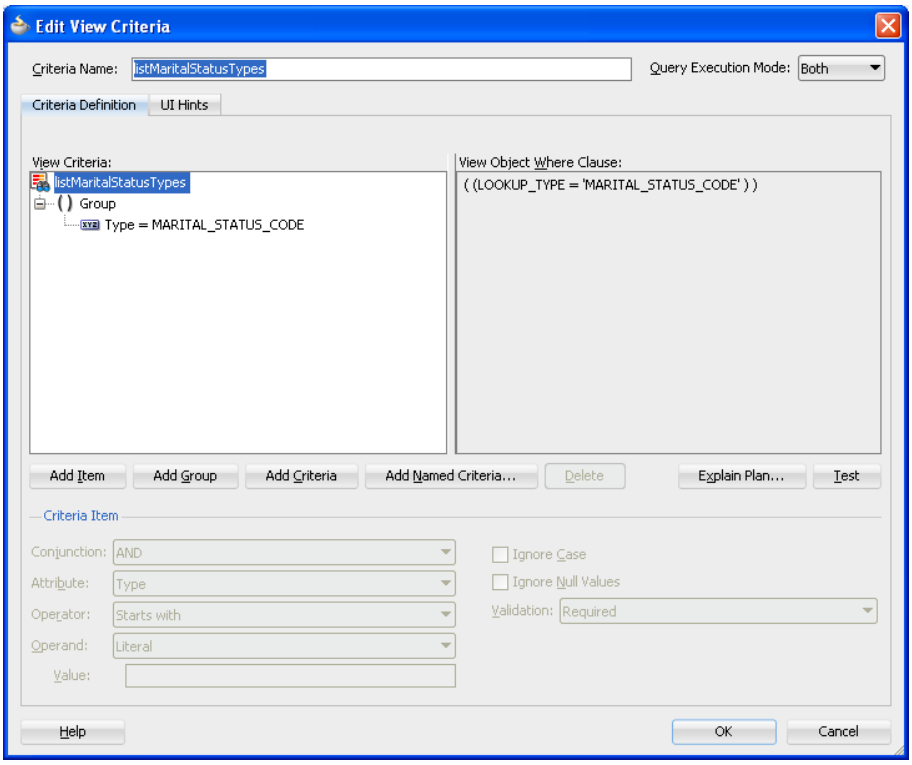
To create LOOKUP_TYPE view criteria for the lookup view object:

1. In the Application Navigator, double-click the lookup base view object you defined.
2. In the overview editor, click the **Query** navigation tab.
3. In the Query page, expand the **View Criteria** section, and click the **Create new view criteria** button.
4. In the Create View Criteria dialog, on the View Criteria page, click the **Add Item** button to add a single criteria item to the view criteria group.
5. In the **Criteria Item** panel, define the criteria item as follows:
   - Choose **Type** as the attribute (or other name that you defined for the attribute the view object maps to the `LOOKUP_TYPE` column).
   - Choose **equal to** as the operator.
   - Keep **Literal** as the operand choice and enter the value name that defines the desired type. For example, to query the marital status codes, you might enter the value `MARITAL_STATUS_CODE` corresponding to the `LOOKUP_TYPE` column.

   Leave all other settings unchanged.

   The view object `WHERE` clause shown in the editor should display a simple criteria similar to the one shown in Figure 10–4, where the value `MARITAL_STATUS_CODE` is set to filter the `LOOKUP_TYPE` column.
6. Click **OK**.
7. Repeat this procedure to define one view criteria for each `LOOKUP_TYPE` that you wish to query.

Figure 10–4 Edit View Criteria Dialog with Lookup View Object View Criteria Specified



10.3.4 What Happens When You Create a View Criteria with the Editor

The Create View Criteria dialog in JDeveloper lets you easily create view criteria and save them as named definitions. These named view criteria definitions add metadata to the target view object's own definition. Once defined, named view criteria appear by name in the Query page of the overview editor for the view object.

JDeveloper then creates the XML component definition file that represents the view objects's declarative settings and saves it in the directory that corresponds to the name of its package. For example, the XML file created for a view object named `LookupsBaseVO` in the `lookups` package is `./lookups/LookupsBaseVO.xml` under the project's source path.

To view the view criteria, expand the desired view object in the Application Navigator, select the XML file under the expanded view object, open the Structure window, and expand the View Criteria node. As shown in Example 10–4, the `LookupsBaseVO.xml` file specifies the `<ViewCriteria>` definition that allows the `LookupsBaseVO` to return only the marital types. Other view criteria added to the `LookupsBaseVO` are omitted from this example for brevity.

Example 10–4 listMaritalStatusTypes View Criteria in the Lookup View Object Definition

```
<ViewObject
    xmlns="http://xmlns.oracle.com/bc4j"
    Name="LookupsBaseVO"
    BindingStyle="OracleName"
    CustomQuery="true"
    PageIterMode="Full"
    UseGlueCode="false">
  <SQLQuery>
    <![CDATA[SELECT L.LOOKUP_TYPE
```

```
    ,L.LOOKUP_CODE
    ,L.MEANING
    ,L.DESCRIPTION
    FROM LOOKUP_CODES L
    WHERE L.LANGUAGE = SYS_CONTEXT('USERENV','LANG')
    ORDER BY L.MEANING]]>
  </SQLQuery>
     ...
  <ViewCriteria
    Name="listMaritalStatusTypes"
    ViewObjectName="oracle.fodemo.storefront.lookups.LookupsBaseVO"
    Conjunction="AND"
    Mode="3"
    <Properties>
      <CustomProperties>
       <Property
         Name="autoExecute"
         Value="false"/>
       <Property
         Name="showInList"
         Value="true"/>
       <Property
         Name="mode"
         Value="Basic"/>
      </CustomProperties>
    </Properties>
    <ViewCriteriaRow
      Name="vcrow24">
      <ViewCriteriaItem
        Name="Type"
        ViewAttribute="Type"
        Operator="="
        Conjunction="AND"
        Value="MARITAL_STATUS_CODE"
        Required="Optional"/>
    </ViewCriteriaRow>
</ViewCriteria>
```

10.3.5 What Happens at Runtime: How a View Instance Accesses Lookup Data

When you create a view instance based on a view criteria, the next time the view instance is executed it augments its SQL query with an additional `WHERE` clause predicate corresponding to the view criteria that you've populated in the view criteria rows.

10.4 Accessing View Instances of the Shared Service

View accessors in ADF Business Components are value accessor objects that point from an entity object attribute (or view object) to a destination view object or shared view instance in the same application workspace. The view accessor returns a row set that by default contains all rows from the destination view object. You can optionally filter this row set by applying view criteria to the view accessor. The base entity object or view object on which you create the view accessor and the destination view object need not be in the same project or application module, but they must be in the same application workspace.

Because view accessors give you the flexibility to reach across application modules to access the queried data, they are ideally suited for accessing view instances of shared

application modules. For details about creating a data model of view instances for a shared application module, see Section 10.2.1, "How to Create a Shared Application Module Instance."

This ability to access view objects in different application modules makes view accessors particularly useful for:

- Validation rules that you set on the attributes of an entity object. In this case, the view accessor derives the validation rule's values from lookup data corresponding to a view instance attribute in the shared application module.
- List of Value (LOV) that you enable for the attribute of any view object. In this case, the view accessor derives the list of values from lookup data corresponding to a view instance attribute in the shared application module.

Validation rules with accessors are useful when you do not want the UI to display a list of values to the user, but you still need to restrict the list of valid values. Alternatively, consider defining an LOV for view object attributes to simplify the task of working with list controls in the user interface. Because you define the LOV on the individual attributes of business components, you can customize the LOV usage for an attribute once and expect to see the list control in the form wherever the attribute appears.

10.4.1 How to Create a View Accessor for an Entity Object or View Object

Entity-based view objects inherit view accessors that you define on their base entity objects. Thus, defining the view accessor once on the entity object itself allows you to reuse the same view accessor, whether you want to define validation rules for entity object attributes or to create LOV-enabled attributes for that entity object's view object. However, when you do not anticipate using view accessors for validation rules, you can add the view accessor directly to the view object that defines the LOV-enabled attribute.

For example, in the `StoreFrontModule` package of the Fusion Order Demo application, the `AddressEO` entity object defines the `Shared_CountriesVA` view accessor and the `AddressesVO` view object inherits this view accessor. In this case, defining the view accessor on the entity object is useful: the accessor for `AddressEO` defines a validation rule on the `CountryId` attribute and the same accessor for `AddressesVO` enables an LOV on its `CountryId` attribute.

When you create a view accessor that accesses a view instance from a shared application module, you may want to use a prefix like `Shared_` to name the view accessor. This naming convention will help you identify the view accessor when you need to select it for the entity object or view object.

You can further refine the list returned by a view accessor by applying view criteria that you define on the view object. To create view criteria for use with a view accessor, see Section 10.3.3, "How to Define the WHERE Clause of the Lookup View Object Using View Criteria."

To create the view accessor:

1. In the Application Navigator, double-click the entity object or view object on which you want to define the view accessor.

   Whether you create the view accessor on the entity object or on the view object will depend on the view accessor's intended usage. Generally, creating view accessors on the entity object ensures the widest possible usage.

2. In the overview editor, click the **View Accessors** navigation tab and click the **Create new view accessors** button to add the accessor to the entity object or view object definition you are currently editing.

3. In the View Accessors dialog, select the view instance name you created for your lookup table from the shared application module node and shuttle it to the view accessors list.

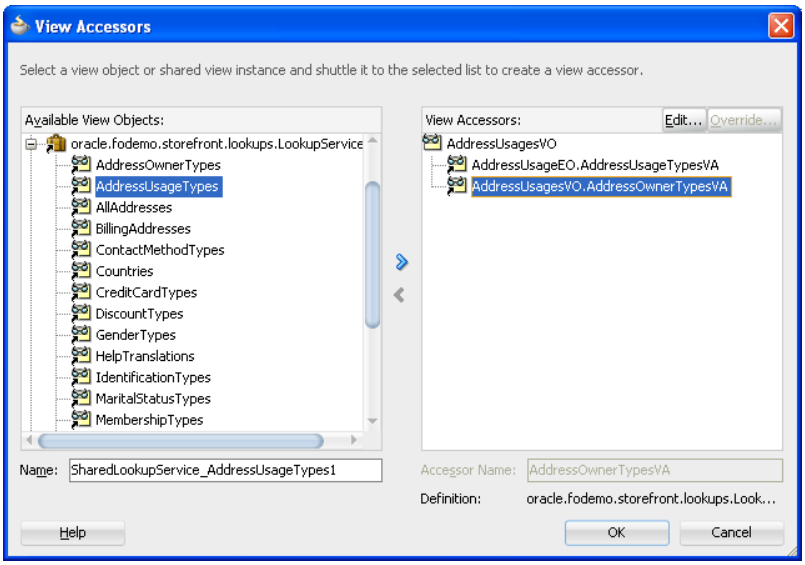
   For example, the View Accessors dialog in the Fusion Order Demo application shows the shared application module `LookupServiceAM` with the list of view instances, as shown in Figure 10–5.

   The dialog will display all view objects and view instances from your application. If you have not yet enabled application module sharing, you must do so before selecting the view instance. For details, see Section 10.2.1, "How to Create a Shared Application Module Instance."

   By default, the view accessor you create will display the same name as the view object instance (or will have an integer appended when it is necessary to distinguish it from a child view object of the same name). You can edit **Accessor Name** to give it a unique name.

   For example, the View Accessors dialog in Figure 10–5 shows the view accessor `AddressUsageTypesVA` for the `AddressUsageTypes` view instance selection in the shared application module `LookupServiceAM`. This view accessor is created on the base entity object `AddressUsagesEO` and accesses the row set of the `AddressUsageTypes` view instance.

Figure 10–5 Defining a View Accessor on an Entity Object



4. Optionally, if you want to apply an existing view criteria to filter the accessor, with the view accessor selected in the overview editor, click the **Edit** icon.

   In the Edit View Accessor dialog, click **Edit** and perform the following steps to apply the view criteria:

   a. Select the view criteria that you want to apply and shuttle it to the **Selected** list.

      You can add additional view criteria to apply multiple filters (a logical AND operation will be performed at runtime).

b. Enter the attribute name for the bind variable that defines the controlling attribute for the view accessor row set.

   Unlike view criteria that you set directly on a view object (to create a view instance, for example), the controlling attribute of the view accessor's view criteria derives the value from the view accessor's base view object.

c. Click **OK** to return to the View Accessors dialog.

5. Click **OK**.

10.4.2 How to Validate Against the Attribute Values Specified by a View Accessor

View accessors that you create to access the view rows of a destination view object may be used to verify data that your application solicits from the end user at runtime. For example, when the end user fills out a registration form, individual validation rules can verify the title, marital status, and contact code against lookup table data queried by view instances of the shared application module.

You can apply view accessors you have defined on the entity object to these built-in declarative validation rules:

- The Compare validator performs a logical comparison between an entity attribute and a value. When you specify a view accessor to determine the possible values, the compare validator applies the `Equals`, `NotEquals`, `GreaterThan`, `LessThan`, `LessOrEqualTo`, `GreaterOrEqualTo` operator you select to compare against the values returned by the view accessor.
- The List validator compares an entity attribute against a list of values. When you specify a view accessor to determine the valid list values, the List validator applies an `In` or `NotIn` operator you select against the values returned by the view accessor.
- The Collection validator performs a logical comparison between an operation performed on a collection attribute and a value. When you specify a view accessor to determine the possible values, the Collection validator applies the Sum, Average, Count, Min, Max operation on the selected collection attribute to compare against the values returned by the view accessor.

Validation rules that you define to allow runtime validation of data for entity-based view objects are always defined on the attributes of the entity object. You use the editor for the entity object to define the validation rule on individual attributes. Any view object that you later define that derives from an entity object with validation rules defined will automatically receive attribute value validation.

Before you begin:

Create the desired entity objects as described in Section 4.2.1, "How to Create Multiple Entity Objects and Associations from Existing Tables."

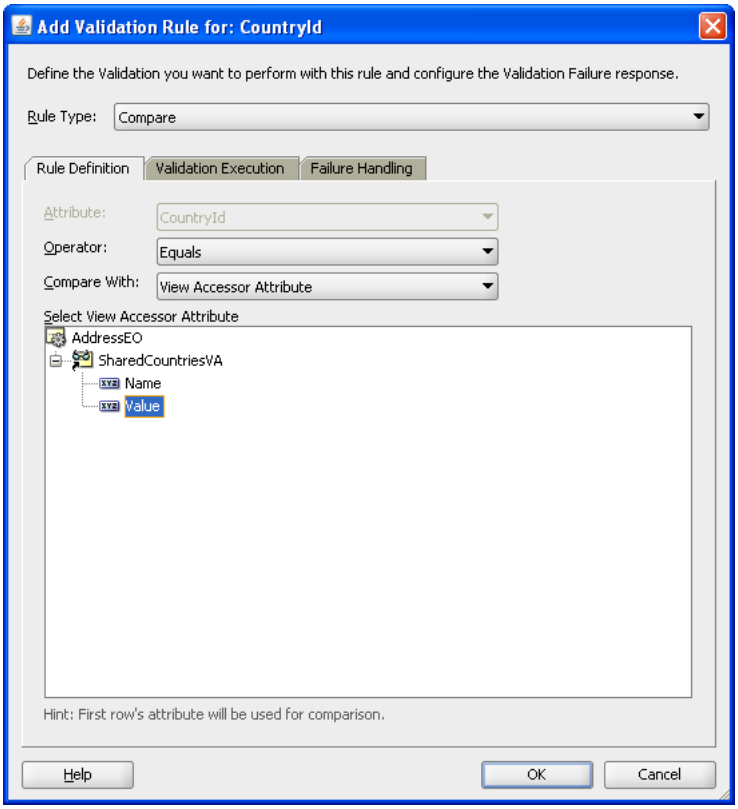
To validate against a view accessor comparison, list, or collection type:

1. In the Application Navigator, double-click the desired entity object.
2. In the overview editor, click the **Business Rules** navigation tab.
3. In the Business Rules page, expand the entity object, select the attribute to be validated and then click the **Create new validator** button to add the validation rule to the entity object attribute.
4. In the Add Validation Rule dialog, in the **Rule Type** dropdown list, select **Compare**, **List**, or **Collection**.

5. Make the selections required by the validator selection.
6. In the **Compare With** or **List Type** dropdown list, select **View Accessor Attribute**.
7. In the **Select View Accessor Attribute** group box, expand the desired view accessor from the shared service and select the attribute you want to provide as validation.

   Figure 10–6 shows what the dialog looks like when you use a List validator to select a view accessor attribute.

Figure 10–6 List Validator Using a View Accessor



8. Click the **Failure Handling** tab and enter a message that will be shown to the user if the validation rule fails.
9. Click **OK**.

10.4.3 What Happens When You Define a View Accessor Validator

When you use a List validator, a `<ListValidationBean>` tag is added to an entity object's XML file. Example 10–5 shows the XML code for the `CountryId` attribute in the `Address` entity object. A List validator has been used to validate the user's entry against the list of country ID values as retrieved by the view accessor from the `Countries` view instance.

Example 10–5 List Validator with View Accessor List Type XML Code

```
<Attribute
   Name="CountryId"
   IsNotNull="true"
   Precision="2"
```

```
   ColumnName="COUNTRY_ID"
   Type="java.lang.String"
   ColumnType="CHAR"
   SQLType="VARCHAR"
   TableName="ADDRESSES">
   RetrievedOnUpdate="true"
   RetrievedOnInsert="true">
   <DesignTime>
         <Attr Name="_DisplaySize" Value="2"/>
   </DesignTime>
   <ListValidationBean
         xmlns="http://xmlns.oracle.com/adfm/validation"
         Name="CountryId_Rule_1"
         ResId="CountryId_Rule_0"
         OnAttribute="CountryId"
         OperandType="VO_USAGE"
         Inverse="false"
         ViewAccAttrName="Value"
         ViewAccName="SharedCountriesVA">
         <ResExpressions>
            <Expression
               Name="0"><![CDATA[SharedCountriesVA.Value]]>
            </Expression>
         </ResExpressions>
   </ListValidationBean>
   <Properties>
      <SchemaBasedProperties>
        <LABEL
          ResId="CountryId_LABEL"/>
      </SchemaBasedProperties>
   </Properties>
</Attribute>
```

10.4.4 What You May Need to Know About Dynamic Filtering with View Accessors

The View Object API `setWhereClause()` method allows you to add a dynamic `WHERE` clause to a view instance whose view accessor may already have view criteria specified at design time. At runtime, when your view accessor and its view criteria is applied to the view instance and you call the `setWhereClause()` method on the view instance to add an extra `WHERE` clause, the programmatically set `WHERE` clause is `AND`-ed with the `WHERE` clause of any already applied view criteria.

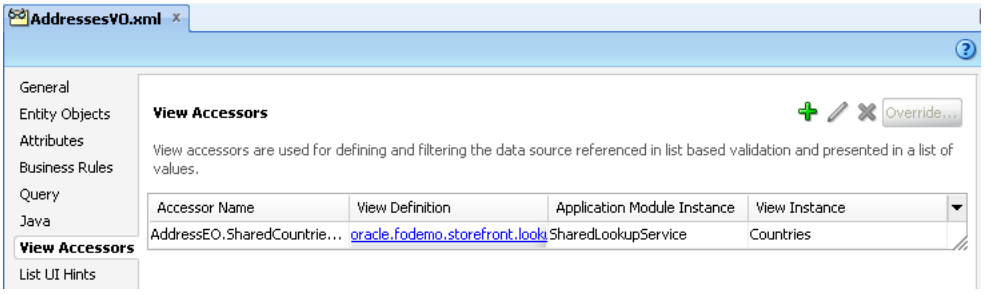
10.4.5 How to Create an LOV Based on a Lookup Table

View accessors that you create to access the view rows of a destination view object may be used to display a list of values to the end user at runtime. You first create a view accessor with the desired view instance as its data source, and then you can add the view accessor to an LOV-enabled attribute of the displaying view object. You will edit the view accessor definition for the LOV-enabled attribute so that it points to the specific lookup attribute of the view instance. Because you want to populate the row set cache for the query with static data, you would locate the destination view instance in a shared application module.

While the list usage is defined on the attribute of a view object bound to a UI list control, the view accessor definition exists on either the view object or the view object's base entity object. If you choose to create the view accessor on the view object's entity object, the View Accessors page of the overview editor for the view object will display the inherited view accessor, as shown in Figure 10–7. Alternatively,

if you choose to create the view accessor on the attribute's view object, you can accomplish this from either the editor for the LOV definition or from the View Accessors page of the overview editor.

Figure 10–7 View Accessors Page of the Overview Editor



For additional examples of how to work with LOV-enabled attributes, see Section 5.12, "Working with List of Values (LOV) in View Object Attributes."

Before you begin:

Create the desired view objects as described in Section 5.2.1, "How to Create an Entity-Based View Object," and Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object."

To create an LOV that displays values from a lookup table:

1. In the Application Navigator, right-click the view object that contains the desired attribute and choose **Open** ***ViewObjectName***.
2. In the overview editor, click the **View Accessors** navigation tab.
3. In the View Accessors page, check to see whether the view object inherited the desired view accessor from its base entity object. If no view accessor is present, either create the view accessor on the desired entity object or click the **Create new view accessors** icon to add the accessor to the view object you are currently editing.

   Validation rules that you define are always defined on the attributes of the view object's base entity object. It may therefore be convenient to define view accessors at the level of the base entity objects when you know that you will also validate entity object attributes using a view accessor list.

   For details about creating a view accessor, see Section 10.4.1, "How to Create a View Accessor for an Entity Object or View Object."
4. In the overview editor, click the **Attributes** navigation tab.
5. In the Attributes page, select the attribute that is to display the LOV, and then click the **List of Values** tab and click the **Add list of values** icon.
6. In the Create List of Values dialog, select the view accessor from the **List Data Source** dropdown list.

   The view accessor you select, will be the one created for the lookup table view object instances to use as the data source.
7. Select the attribute from this view accessor from the **List Attribute** dropdown list that will return the list of values for the attribute you are currently editing.

   The editor creates a default mapping between the view object attribute and the LOV-enabled attribute. In this use case, the attributes are the same. For example,

the attribute `OrderId` from the `OrdersView` view object would map to the attribute `OrderId` from the `Shared_OrdersVA` view accessor.

8. Optionally, when you want to specify supplemental values that your list returns to the base view object, click **Add** icon in **List Return Values** and map the desired view object attributes to the same attributes accessed by the view accessor. Supplemental attribute return values are useful when you do not require the user to make a list selection for the attributes, yet you want those attributes values, as determined by the current row, to participate in the update.

   For example, to map the attribute `StartDate` from the `OrdersView` view object, you would choose the attribute `StartDate` from the `Shared_OrdersVA` view accessor. Do not remove the default attribute mapping for the attribute for which the list is defined.

9. Click **OK**.

10.4.6 What Happens When You Define an LOV for a View Object Attribute

When you add an LOV to a view object attribute, JDeveloper updates the view object's XML file with an `LOVName` property in the `<ViewAttribute>` element. The definition of the LOV appears in a new `<ListBinding>` element. The metadata in Example 10–6 shows that the `MaritalStatusCode` attribute refers to the `MaritalStatusLOV` LOV and sets the `choice` control type to display the LOV. The LOV definition for `MaritalStatusLOV` appears in the `<ListBinding>` element.

Example 10–6 View Object with LOV List Binding XML Code

```
<ViewObject
  xmlns="http://xmlns.oracle.com/bc4j"
  Name="CustomerRegistrationVO"
  ...
   <ViewAttribute Name="MaritalStatusCode" IsNotNull="true" PrecisionRule="true"
                  EntityAttrName="MaritalStatusCode" EntityUsage="PersonEO"
                  AliasName="MARITAL_STATUS_CODE"
                  LOVName="MaritalStatusCodeLOV">
      <Properties>
         <SchemaBasedProperties>
            <CONTROLTYPE Value="choice"/>
         </SchemaBasedProperties>
      </Properties>
   </ViewAttribute>
  ...
  <ListBinding
    Name="MaritalStatusLOV"
    ListVOName="PersonEO.MaritalStatusVA"
    ListRangeSize="-1"
    NullValueFlag="start"
    NullValueId="LOVUIHints_NullValueId"
    MRUCount="0">
    <AttrArray Name="AttrNames">
      <Item Value="MaritalStatusCode"/>
    </AttrArray>
    <AttrArray Name="ListAttrNames">
      <Item Value="Value"/>
    </AttrArray>
    <AttrArray Name="ListDisplayAttrNames">
      <Item Value="Name"/>
    </AttrArray>
    <DisplayCriteria/>
```

```
    <AttrArray Name="DerivedAttrNames"/>
  </ListBinding>
  ...
</ViewObject>
```

10.4.7 How to Automatically Refresh the View Object of the View Accessor

If you need to ensure that your view accessor always queries the latest data from the lookup table, you can set the **Auto Refresh** property on the destination view object. This property allows the view object instance to refresh itself after a change in the database. The typical use case is when you define a view accessor for the destination view object.

Because the auto-refresh feature relies on the database change notification feature, observe these restrictions when enabling auto-refresh for your view object:

- The view objects should query as few read-only tables as possible. This will ensure the best performance and prevent the database invalidation queue from becoming too large.
- The database user must have database notification privileges. For example, to accomplish this with a SQL*Plus command use `grant change notification to <user name>`.

When these restrictions are observed, the refresh is accomplished through the Oracle database change notification feature. Prior to executing the view object query, the framework will use the JDBC API to register the query for database notifications. When a notification arrives, the row sets of the corresponding view object instance are marked for refresh during the next checkout of the application module. Because the shared application module waits until the next checkout, the row set currency of the current transaction is maintained and the end user is not hampered by the update.

For example, assume that an LOV displays a list of zip codes that is managed in read-only fashion by a database administrator. After the administrator adds a new zip code as a row to the database, the shared application module detects a time when there are no outstanding requests and determines that a pending notification exists for the view instance that access the list of zip codes; at that point, the view object refreshes the data and all future requests will see the new zip code.

To enable auto-refresh for a view instance of a shared application module:

1. In the Application Navigator, double-click the view object that you want to receive database change notifications.
2. In the overview editor, click the **General** navigation tab.
3. In the Property Inspector expand the **Tuning** section, and select **True** for the **Auto Refresh** property.

10.4.8 What Happens at Runtime: How the Attribute Displays the List of Values

The ADF Business Components runtime adds functionality in the attribute setters of the view row and entity object to facilitate the LOV-enabled attribute behavior. In order to display the LOV-enabled attribute values in the user interface, the LOV facility fetches the data source, and finds the relevant row attributes and mapped target attributes.

10.4.9 What You May Need to Know About Displaying List of Values From a Lookup Table

Unlike entity-based view objects, read-only view objects that you create in expert mode, will not define a key attribute by default. While it is possible to create a read-only view object without defining its key attribute, in expert mode it is a best practice to select the attribute that corresponds to the queried table's primary key and mark it as the key attribute. The presence of a key attribute ensure the correct runtime behavior for row set navigation. For example, the user interface developer may create a LOV component based on the read-only view object collection. Without a key attribute to specify the row key value, the LOV may not behave properly and a runtime error can result.

10.4.10 What You May Need to Know About Programmatically Invoking Database Change Notifications

When you create a databound UI component in a web page, you can enable the auto-refresh feature on the corresponding view object, as described in Section 10.4.7, "How to Automatically Refresh the View Object of the View Accessor." In this case, the ADF Model layer and ADF Business Components will handle processing database change notifications at runtime and the shared application module cache will be updated for you. However, when you create a method to programmatically refresh a view instance of a shared application module, your method needs to invoke the `processChangeNotification()` method on the shared application module before you refresh the view instance. Example 10–7 shows how to use the `processChangeNotification()` method to ensure the shared application module cache gets updated if the corresponding queried data has changed in the database.

Example 10–7 Programmatically Invoking a View Instance and Processing the Database Change Notification

```
public String getProduct(Long id) {
       ApplicationModuleHandle handle = null;
       try {
            handle = Configuration.createRootApplicationModuleHandle
                       ("oracle.apps.mlss.model.applicationModule.ProductAM", "ProductAMShared");
            ApplicationModule sam = handle.useApplicationModule();
            sam.processChangeNotifications();

            ViewObject svo = sam.findViewObject("Product");
            svo.setNamedWhereClauseParam("bindProductId", id);
            svo.setNamedWhereClauseParam("bindLang", "US");
            ((ViewObjectImpl)svo).refreshCollection(null, false, false);
            svo.reset();
            while (svo.hasNext()) {
                Row r = svo.next();
                return (String)r.getAttribute("Name");
           }
       } finally {
            if (handle != null)
                Configuration.releaseRootApplicationModuleHandle(handle, false);
       }

       return null;
}
```

10.4.11 What You May Need to Know About Inheritance of AttributeDef Properties

When one view object extends another, you can create the LOV-enabled attribute on the base object. Then when you define the child view object in the overview editor, the LOV definition will be visible on the corresponding view object attribute. This inheritance mechanism allows you to define an LOV-enabled attribute once and apply it later across multiple view objects instances for the same attribute. For details about extending a view object from another view object definition, see Section 12.9.2, "How To Extend a Component After Creation."

You can also use the overview editor to extend the inherited LOV definition. For example, you may add extra attributes already defined by the base view object's query to display in selection list. Alternatively, you can create a view object instance that uses a custom `WHERE` clause to query the supplemental attributes not already queried by the base view object. For information about customizing entity-based view objects, see Section 5.10, "Working with Bind Variables."

10.4.12 What You May Need to Know About Using Validators

If you have created an LOV-enabled attribute for a view object, there is no need to validate the attribute using a List validator. You use an attribute validator only when you do not want the list to display in the user interface but still need to restrict the list of valid values. A List validator may be a simple static list or it may be a list of possible values obtained through a view accessor you define. Alternatively, you might prefer to use a Key Exists validator when the attribute displayed in the UI is one that references a key value (such as a primary, foreign, or alternate key). For information about declarative validation in ADF Business Components, see Chapter 7, "Defining Validation and Business Rules Declaratively."

10.5 Testing View Object Instances in a Shared Application Module

JDeveloper includes an interactive application module testing tool that you can use to test all aspects of its data model without having to use your application user interface or write a test client program. Running the Oracle ADF Model Tester can often be the quickest way of exercising the data functionality of your business service during development.

10.5.1 How to Test the Base View Object Using the Oracle ADF Model Tester

The application module is the transactional component that the Oracle ADF Model Tester (or UI client) will use to work with application data. The set of view objects used by an application module defines its data model, in other words, the set of data that a client can display and manipulate through a user interface. You can use the Oracle ADF Model Tester to test that the accessors you defined yield the expected validation result and that they display the correct LOV attribute values.

To create an application module, use the Create Application Module wizard, which is available in the New Gallery. For more information, see Section 9.2, "Creating and Modifying an Application Module."

To test the view objects you added to an application module, use the Oracle ADF Model Tester, which is accessible from the Application Navigator.

To test view objects in an application module configuration:

1. In the Application Navigator, expand the project containing the desired application module and view objects.

2. Right-click the application module and choose **Run**.

   Alternatively, choose **Debug** when you want to run the application in the Oracle ADF Model Tester with debugging enabled. For example, when debugging using the Oracle ADF Model Tester, you can view status message and exceptions, step in and out of source code, and manage breakpoints. JDeveloper opens the debugger process panel in the Log window and the various debugger windows.

   For details about receiving diagnostic messages specific to ADF Business Components debugging, see Section 6.3.8, "How to Enable ADF Business Components Debug Diagnostics."

3. In the Select Business Components Configuration dialog, choose the desired application module configuration from the **Business Component Configuration Name** list to run the Oracle ADF Model Tester.

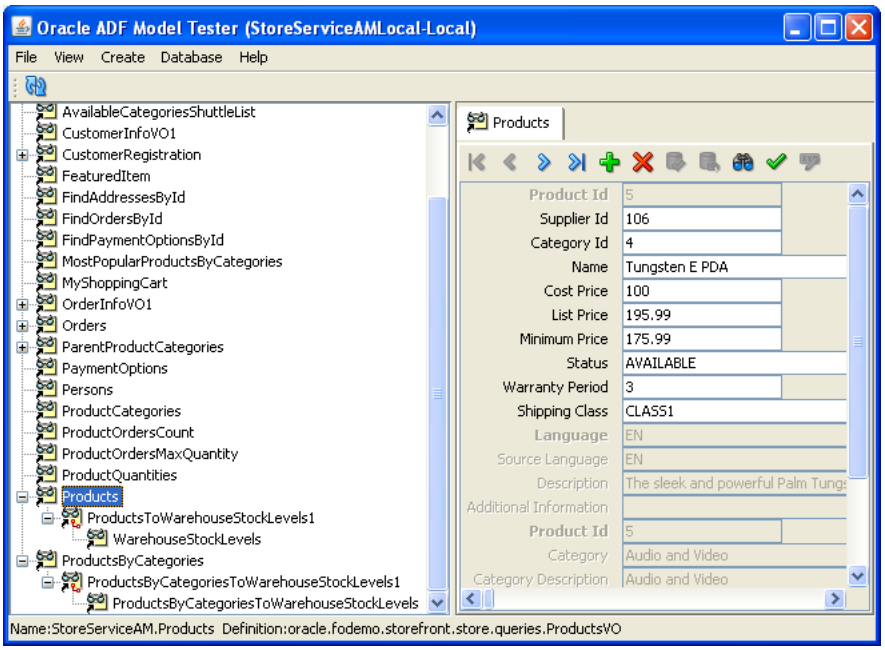
   By default, an application module has only its default configurations, named *AppModuleName*`Local` and *AppModuleName*`Shared`. If you have created additional configurations for your application module and want to test it using one of those instead, just select the desired configuration from the **Business Components Configuration** dropdown list on the Connect dialog before clicking **Connect**.

4. Click **Connect** to start the application module using the selected configuration.

5. To execute a view object in the Oracle ADF Model Tester, expand the tree list and double-click the desired view object node.

   Note that the view object instance may already appear executed in the testing session. In this case, the tester panel on the right already displays query results for the view object instance, as shown in Figure 10–8. The fields in the tester panel of a read-only view object will always appear disabled since the data it represents is not editable.

Figure 10–8 Testing the Data Model in the Oracle ADF Model Tester



10.5.2 How to Test LOV-Enabled Attributes Using the Oracle ADF Model Tester

To test the LOV you created for a view object attribute, use the Oracle ADF Model Tester, which is accessible from the Application Navigator. For details about displaying the tester and the supported control types, see Section 5.12.8, "How to Test LOV-Enabled Attributes Using the Oracle ADF Model Tester."

10.5.3 What Happens When You Use the Oracle ADF Model Tester

When you launch the Oracle ADF Model Tester, JDeveloper starts the tester tool in a separate process and the Oracle ADF Model Tester appears. The tree at the left of the dialog displays all of the view object instances in your application module's data model. Figure 10–8 shows just one instance in the expanded tree, called `ProductImages`. After you double-click the desired view object instance, the Oracle ADF Model Tester will display a panel to inspect the query results, as shown in Figure 10–8.

The test panel will appear disabled for any read-only view objects you display because the data is not editable. But even for the read-only view objects, the tool affords some useful features:

- You can validate that the UI hints based on the Label Text control hint and format masks are defined correctly.
- You can also scroll through the data using the toolbar buttons.

The Oracle ADF Model Tester becomes even more useful when you create entity-based view objects that allow you to simulate inserting, updating, and deleting rows, as described in Section 6.3.2, "How to Test Entity-Based View Objects Interactively."

10.5.4 What Happens at Runtime: How Another Service Accesses the Shared Application Module Cache

When a shared application module with application scope is requested by an LOV, then the ADF Business Components runtime will create an `ApplicationPool` object for that usage. There is only one `ApplicationPool` created for each shared usage that has been defined in the ADF Business Components project configuration file (`.jpx`). The runtime will then use that `ApplicationPool` to acquire an application module instance that will be used like a user application module instance, to acquire data. The reference to the shared application module instance will be released once the application-scoped application module is reset. The module reference is released whenever you perform an unmanaged release or upon session timeout.

Since multiple threads will be accessing the data caches of the shared application module, it is necessary to partition the iterator space to prevent race conditions between the iterators of different sessions. This will help ensure that the next request from one session does not change the state of the iterator that is being used by another session. The runtime uses ADF Business Components support for multiple iterators on top of a single `RowSet` to prevent these race conditions. So, the runtime will instantiate as many iterators as there are active sessions for each `RowSet`.

An application-scoped shared application module lifecycle is similar to the lifecycle of any application module that is managed by the `ApplicationPool` object. For example, once all active sessions have released their shared application module, then the application module may be garbage-collected by the `ApplicationPool` object. The shared pool may be tuned to meet specific application requirements.

Session-scoped shared application modules are simply created as nested application module instances within the data model of the root, user application module. For details about nested application modules, Section 9.4, "Defining Nested Application Modules."

11

Integrating Service-Enabled Application Modules

This chapter describes how to publish ADF application modules to make them available as external services in a Fusion web application, and how to incorporate the published application module as an external service in the Fusion web application.

This chapter includes the following sections:

- Section 11.1, "About Service-Enabled Application Modules"
- Section 11.2, "Publishing Service-Enabled Application Modules"
- Section 11.3, "Accessing Remote Data Over the Service-Enabled Application Module"

11.1 About Service-Enabled Application Modules

Service-enabled application modules are ADF application modules that you advertise through a service interface to service consumers. There are three scenarios for service consumers to consume a published service-enabled application module: web service access, Service Component Architecture (SCA) composite access, and access by another ADF application module.

Service Component Architecture (SCA) provides an open, technology-neutral model for implementing remotable services that are defined in terms of business functionality and that make middleware functions more accessible to application developers. ADF Business Components supports an SCA-compliant solution through application modules you can publish with a service interface.

When you service-enable your application module, JDeveloper generates the necessary artifacts comprising: 1) The Java interface defining the service, 2) an EJB 3.0 session bean that implements this Java interface, 3) a WSDL file that describes the service's operations, and (4) an XML Schema Document (XSD) that defines the service's data structures. The service interface is described for Fusion web application clients in a language-neutral way by the combination of WSDL and XSD.

SCA defines two kinds of service:

- Remoteable services, typically coarse-grained and designed to be published remotely in a loosely coupled SOA architecture
- Local services, typically fine-grained and designed to be used locally by other implementations that are deployed concurrently in a tightly coupled architecture

ADF Business Components services fall into the first category, and should only be used as remoteable services.

ADF Business Components services, including data access and method calls, defined by the remote application modules are interoperable with any other application module. This means the same application module can support interactive web user interfaces using ADF data controls and web service clients.

11.1.1 Service-Enabled Application Module Use Cases and Examples

Any development team can publish a service-enabled application module to contribute to the Fusion web application. The Fusion web application assembled from remote services also does not require the participating services to run on a single application server.

Although the web applications may run on separate application servers, the appearance that SCA provides is one of a unified application. Consuming client projects use the ADF service factory lookup mechanism to access the data and any business methods encapsulated by the service-enabled application module. At runtime, the calling client and the ADF web service may or may not participate in the same transaction, depending on the protocol used to invoke the service (either SOAP or RMI). Only the RMI protocol and a Java Transaction API (JTA) managed transaction support the option to call the service in the same transaction as the calling client. By default, to support the RMI protocol, the ADF web service is configured to participate in the same transaction.

11.1.2 Additional Functionality for Service-Enabled Application Modules

You may find it helpful to understand other ADF features before you start working with ADF Business Components services. Following are links to other functionality that may be of interest.

- For information about the SCA and service data object (SDO) standards, see the Open SOA web site at `http://www.osoa.org`.
- For further background about web services and Oracle WebLogic Server support for web services, see *Oracle Fusion Middleware Introducing Web Services*.
- For details about how to delegate data operations to the SDO data provider through the use of the Business Process Execution Language (BPEL) entity variable, see the *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*.
- For local service support, use the ApplicationModule interface and ViewObject interface support described in Section 9.10, "Working Programmatically with an Application Module's Client Interface."
- For API documentation related to the `oracle.jbo` package, see the following Javadoc reference document:
  - *Oracle Fusion Middleware Java API Reference for Oracle ADF Model*

11.2 Publishing Service-Enabled Application Modules

The application module is ADF Business Components framework component that encapsulates business logic as a set of related business functions. Application modules are mapped to services. You use the overview editor for your application module to enable a web service interface and publish rows of view object data as service data object (SDO) components.

The SDO framework upon which the SDO components are based abstracts the data of the view object and standardizes the way that data structures are passed between Java and XML. This data abstraction simplifies working with heterogeneous data sources in

a service-oriented architecture (SOA) and lets you selectively service-enable view objects using the same view object to support interactive web user interfaces and web service clients.

JDeveloper allows you to expose application modules as web services which use SDO components based on view instance that your application module defines to standardize the way that data structures are passed between Java and XML. JDeveloper also generates the WSDL service description that is used by the web service client in the consuming application.

> **Note:** JDeveloper only supports generating SDO components for view objects of a service-enabled application module. Currently, no other Oracle ADF business components may be defined as SDO components.

The service-enabled application module exposes the view objects, custom methods, built-in data manipulation operations, and specialized find methods based on named view criteria to be used by the client. Once you have enabled the application module service interface, you will need to create an ADF Business Components Service Interface deployment profile and deploy it to the target application server.

You can also expose the view instance data manipulation operations for use with a Business Process Execution Language (BPEL) process service component. BPEL is a language for composing multiple services into an end-to-end business process. For details about how to delegate data operations to the SDO data provider through the use of the BPEL entity variable, see the *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*.

> **Note:** It is important to note that you don't implement methods with SDO parameters directly. The SDO framework is used to wrap the view row types during runtime only.

11.2.1 How to Enable the Application Module Service Interface

You edit the application module in JDeveloper to create a web service interface that exposes the top-level view objects and defines the available service operations it supports. The top-level view objects that you select are service-enabled automatically and will be accessible by the service client.

The primary purpose of the standard service operations is to expose data manipulation operations on the view objects. Any business logic that you have defined on the underlying framework objects (for example, business rule validation) will be applied when you invoke a standard service operation. Table 11–1 shows the list of standard operations that service view instances support.

Table 11–1 Standard View Instance Data Manipulation Operations

Operation	Method Name	Operation Description
`Create`	`create<`*`VOName`*`>`	Creates a single ADF Business Components view row.
`Update`	`update<`*`VOName`*`>`	Updates a single ADF Business Components view row.
`Delete`	`delete<`*`VOName`*`>`	Deletes a single ADF Business Components view row.

Table 11–1 (Cont.) Standard View Instance Data Manipulation Operations

Operation	Method Name	Operation Description
`Merge`	`merge<VOName>`	Updates a ADF Business Components view row if one exists; otherwise, creates a new one.
`GetByKey`	`get<VOName>`	Gets a single ADF Business Components view row by primary key.
`Find` (by view object query statement)	`find<VOName>`	Finds and returns a list of ADF Business Components view rows based on the selected view object's query statement. Note that the query must not specify a bind variable defined as required for the query to execute. The service interface does not expose required bind variables at runtime. For details about creating a find method for this scenario, see Section 11.2.8, "How to Expose a Declarative Find Operation Filtered By a Required Bind Variable."
`Find` (by view criteria)	`find<VOName><VCName>`	Finds and returns a list of single ADF Business Components view rows by SDO-based view criteria. This is the preferred way to filter the ADF Business Components view rows that rely on a required bind variable.
`Process`	`process<VOName>`	Performs a Create, Update, Delete, or Merge operation on a list of ADF Business Components view rows. The specified operation is applied to all objects in the given list.
`ProcessChangeSummary`	`processCS<VOName>`	Performs a Create, Update, or Delete operation on a list of ADF Business Components view rows. Different operations may be applied to different objects, depending on what is specified in the `ChangeSummary` object.

Note: When you generate a SDO class for a parent view object, JDeveloper automatically generates SDO classes for view instances that extend the parent in a polymorphic collection. For details about polymorphic view objects, see Section 42.6.5, "Working with Polymorphic View Rows."

For information on how to create SDO classes for child view objects, see Section 11.2.4, "How to Service-Enable Individual View Objects."

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF application modules and enables web service clients to access rows of data and perform service operations. For more information, see Section 11.2, "Publishing Service-Enabled Application Modules."

When you create the service-enabled application module, do not create it in a project that already contains a standard web service (a Java class or interface with the `@WebService` annotation). JDeveloper deployment profiles do not support deploying

a standard web service and an ADF Business Service web service from the same project. If you attempt to deploy the ADF Business Service web service from the same project as a standard web service, the deployment will fail with an Oracle WebLogic Server exception error.

Complete these tasks:

- Create the desired view objects, as described in Section 5.2.1, "How to Create an Entity-Based View Object," and Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object."
- Create the desired application module as described in Section 9.2.1, "How to Create an Application Module."
- Optionally, set JDeveloper preferences to specify a default suffix for the names of generated SDO classes, modify the default subpackage where the service common interface and classes reside, and set the default namespace prefix for the generated SDO schema and web service as described in Section 11.2.13, "How to Set Preferences for Generating the ADF Web Service Interface."
- If you want to expose custom find operations on the service, create declarative view criteria to specify the custom query as described in Section 5.11.1, "How to Create Named View Criteria Declaratively."

To create the web service interface:

1. In the Application Navigator, double-click the application module.
2. In the overview editor, click the **Service Interface** navigation tab and click the **Enable support for Service Interface** button.

   Use the Create Service Interface wizard to configure the desired options.
3. In Create Service Interface wizard, on the Service Interface page, enter the name and target namespace for the web service.

   The target namespace is a URI for the service that you can assign to group similar services together by entering the same URI.
4. To generate a method that will return the static control hints (UI hints) defined on the view instances that you service enable, select **Generate Control Hints Operation**.

   When you enable this option, the wizard adds the `getDfltCtrlHints()` method to the service interface. The service interface client can invoke this method to resolve UI hints on the server without requiring a database roundtrip. The method takes the view object name and a locale and returns the base UI hints for that locale.
5. To expose the methods of the application module as asynchronous service methods and enable both synchronous and asynchronous operations on the web service, select **Generate Asynchronous Web Service Methods**.

   By default, the web service supports synchronous service methods. This forces the invoking client application to wait for the response to return before it can continue with its work. In cases where the response returns immediately, this method of invoking the web service is common. However, because request processing can be delayed, it is often useful for the client application to continue its work and to handle the response later on.
6. On the Service Custom Methods page, add the custom methods you want to expose in the service interface and define the data types of each method's parameters and return value.

The parameters and non-void return value of the custom service methods you enable must be one of the service-interface supported data types, such as a Java primitive type, or a list of the service-interface supported data types (including `oracle.jbo.server.ViewRowImpl`, `java.util.List<`*`ViewRowImpl`*`>`, `oracle.jbo.AttributeList`, `java.util.List<`*`AttributeList`*`>`, or `java.util.List<`*`PrimitiveType`*`>`).

Note that although both `ViewRowImpl` and `AttributeList` data types expose the identical row structure to the web service client, at runtime there will be a fundamental difference. For a description of the supported data types, see Section 11.2.3, "What You May Need to Know About Method Signatures on the ADF Web Service Interface."

After selecting a qualifying custom method to appear in the service interface, for each parameter and return value using the `ViewRowImpl` or `AttributeList` data type, you must in turn select the name of the view object instance corresponding to the row structure:

a. In the Selected list, expand **return** or **parameters** and select the item.

b. Enter the Java element data type in **Element Java Type**.

c. In the case where the Java element type is `ViewRowImpl` or `AttributeList`, enter the view object instance name to identify the row structure in **Element View Object**.

For example, if you define a custom method to return a single row of the `CustomerInfo` view object instance, you would need a custom method signature like this:

```
public ViewRowImpl findCustomerInfo(int id)
```

Then, after selecting the `findCustomerInfo()` custom method to appear in the service interface, you would select its return value in the tree and configure its View Object property to be `CustomerInfo`, the view instance name whose row structure should be used at runtime.

7. To expose service information messages such as warnings and exceptions for a custom method or warnings for process operation methods, select **Include Warnings in Return Service Data Object**. For example, you might want display an informational message when a method returns the total employee compensation and the total is outside of the desired range.

If **Include Warnings** is not selected, no informational messages will be returned with the service response.

This option is only enabled when the method does not return a view row or a list of view rows. When the method returns view rows, the underlying view object determines whether the method supports warnings, as described Section 11.2.4, "How to Service-Enable Individual View Objects."

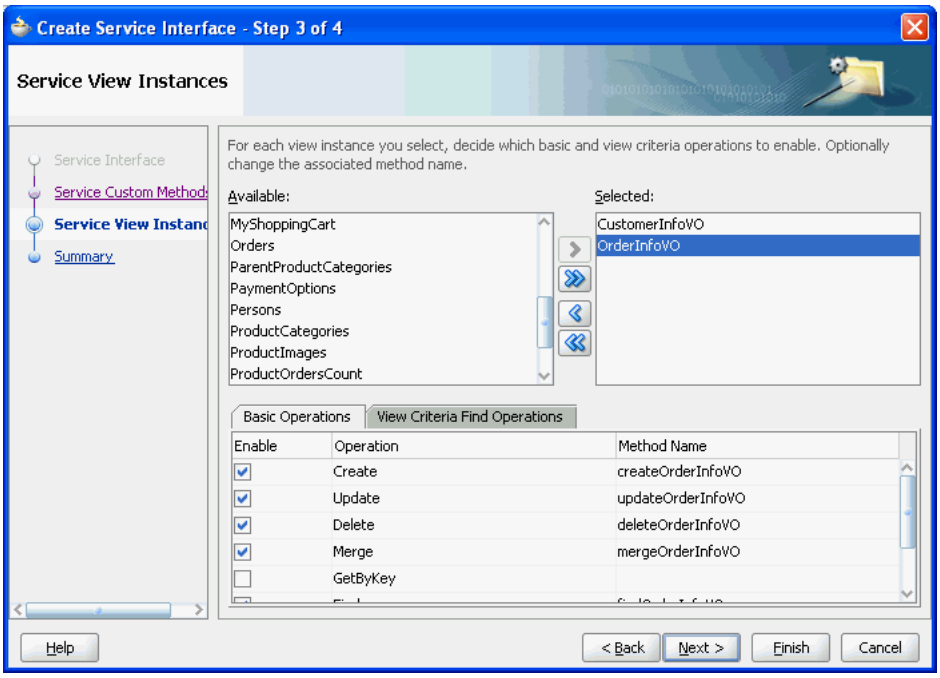
The informational messages (and warnings) are reported as part of the return object. JDeveloper generates appropriate wrappers as the return objects, and the wrappers contain the actual method return and the informational messages.

8. On the Service View Instances page, select the top-level view instances in the application module that you want to expose in the service interface.

View object subtypes of the top-level view instance will automatically be service-enabled.

Also, on this page, you can enable the available data manipulation operations supported on the exposed methods, as shown in Figure 11–1.

Figure 11–1 View Instances and CRUD Operation Selection



9. In the Basic Operations tab, select the data manipulation operations for the currently selected view instance and in the **Method Name** field, change the names of the selected service operations to the names that you prefer to expose in the service interface.

   The primary purpose of the standard service operations is to expose data manipulation operations on the view objects. Any business logic that you have defined on the underlying framework objects (for example, business rule validation) will be applied when you invoke the service operations. For a description of the operations that service view instances support, see Table 11–1.

   In the case of the find method operation that you can select, the find method must not reference a required bind variable in the view object's query statement. A required bind variable is one that makes the query execution dependent on the availability of a valid value for the bind variable. The service interface does not expose required bind variables at runtime. For details about defining a find operation for this scenario, see Section 11.2.8, "How to Expose a Declarative Find Operation Filtered By a Required Bind Variable."

10. To expose declarative find operations, select the **View Criteria Find Operations** tab and click the **Add View Criteria** icon.

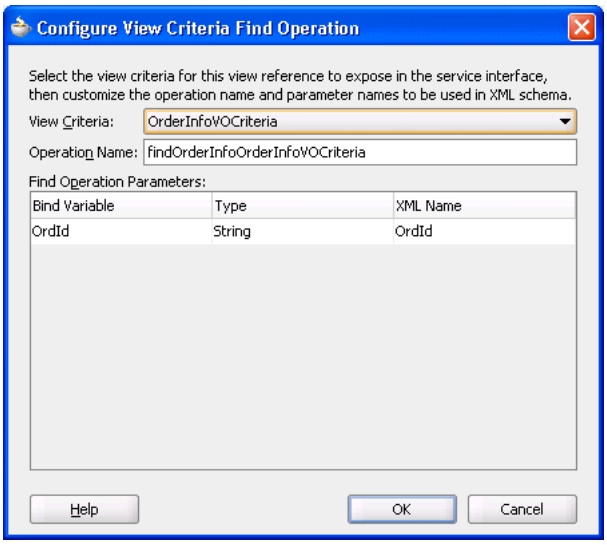
    You can define custom find operations when you want the service to support executing a predefined query. For information about defining a named view criteria, see Section 5.11, "Working with Named View Criteria."

Caution: The service interface find operations are based on specific view criteria that your project defines. This means that that the bind variables of the view criteria must match the parameters of the corresponding find operation method. If you change the number or order of the bind variables after the find operation is defined and the service interface generated, the corresponding method will not execute at runtime. Therefore, after changing the underlying view criteria, you must regenerate the service interface.

a. In the Configure View Criteria Find Operation dialog, choose the named view criteria for the find operation.

   The dialog displays the list of view criteria exposed by the referenced view object. For example, `OrderInfoVO` defines `OrderInfoVOCriteria` with a bind variable `OrdId` that specifies the order ID, as shown in Figure 11–2.

Figure 11–2 Specialized Find Methods Based on Named View Criteria



b. If the view criteria uses a bind variable, you can double-click the XML name to customize the name as it will appear in the XML definition for the service.

11. Click **Next** to review the custom methods that your service view instances will expose.

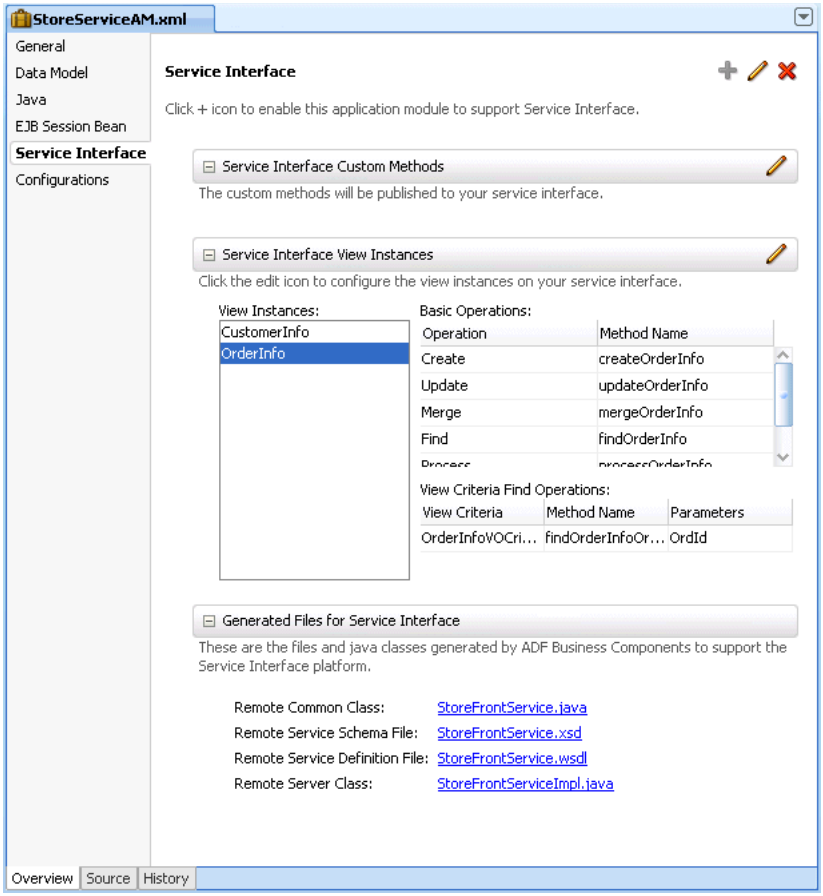
12. Click **Finish**.

11.2.2 What Happens When You Create an Application Module Service Interface

JDeveloper generates the service interface class and enables any view instance options you have chosen, as shown in Figure 11–3.

Do not modify the generated files for the service-enabled application module. The generated files implement required methods of the service interface. An exception to this are use cases that require adding Java annotations to the remote service implementation class. For example, annotations that you add in the remote common interface implementation class let you attach security policies, as described in Section 11.2.14, "How to Secure the ADF Web Service for Access By SOAP Clients."

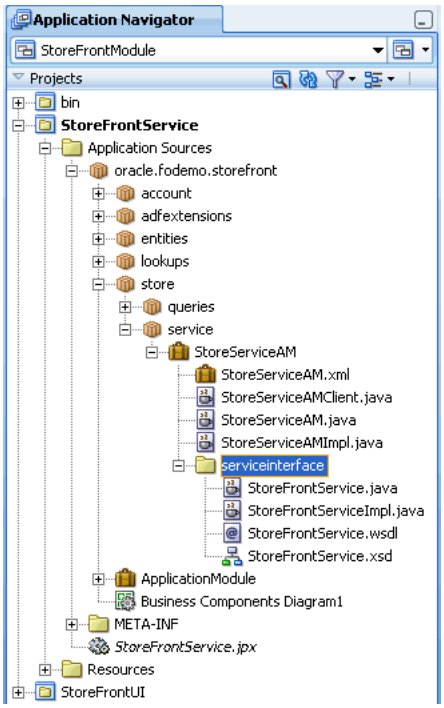
Figure 11–3 Service Interface Page of the Overview Editor for an Application Module



The following types of files are generated and are listed in the Application Navigator in the Projects panel, under the application module's **serviceinterface** node, as shown in Figure 11–4.

- Remote common interface, for example, `StoreFrontService.java`
- Remote service schema file, for example, `StoreFrontService.xsd`
- Remote service definition file, for example, `StoreFrontService.wsdl`
- Remote server class, for example, `StoreFrontServiceImpl.java`

Figure 11–4 Service Interface Files Appear Below Application Module



In addition, the `connections.xml` file is created when you first create an ADF Business Components service. Although this file is is only used by the web service client (the consuming application), it is generated with the service-enabled application module as a convenience. This file appears in the Application Navigator in the Application Resources panel, under the **Descriptors** and **ADF META-INF** nodes.

11.2.2.1 Remote Common Interface

The remote common interface uses metadata annotations specified by the web service specification (JSR-181) to indicate how the interface should be exposed as a web service. This example shows part of `StoreFrontService.java`, which is the remote common interface class for the `StoreServiceAM` application module in the StoreFront module of the Fusion Order Demo.

Example 11–1 Remote Common Interface in Fusion Order Demo

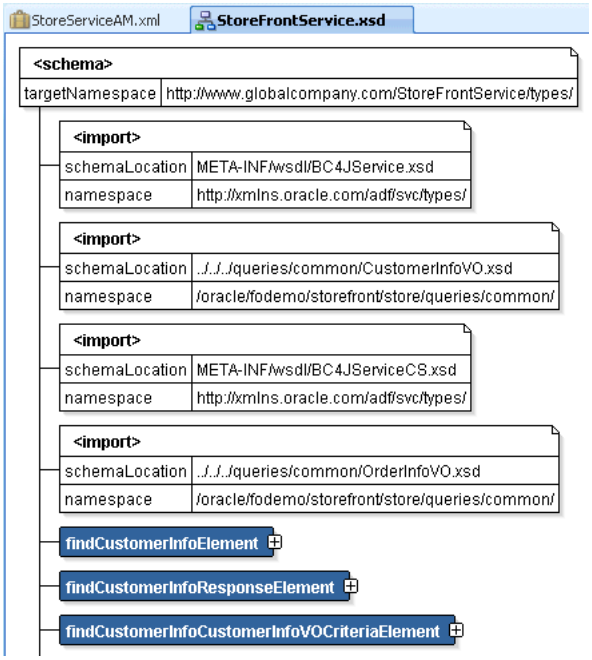
```
package oracle.fodemo.storefront.store.service.common.serviceinterface;
...
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.soap.SOAPBinding;
...
import oracle.fodemo.storefront.store.queries.common.CustomerInfoVOSDO;
import oracle.fodemo.storefront.store.queries.common.OrderInfoVOSDO;
...
import oracle.webservices.annotations.PortableWebService;
import oracle.webservices.annotations.SDODatabinding;
...
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.WRAPPED, style=SOAPBinding.Style.DOCUMENT)
@PortableWebService(targetNamespace="http://www.globalcompany.com/StoreFrontService",
    name="StoreFrontService",
    wsdlLocation=
```

```
            "oracle/fodemo/storefront/store/service/common/serviceinteface/StoreFrontService.wsdl")
@SDODatabinding(schemaLocation=
            "oracle/fodemo/storefront/store/service/common/serviceinteface/StoreFrontService.xsd")
public interface StoreFrontService
{
    public static final String NAME = ("http://www.globalcompany.com/StoreFrontService")

    /**
     * @param orderId
     * @return
     * @throws ServiceException
     */
    @WebMethod(action="www.globalcompany.example.com/getOrderInfoVOSDO",
        operationName="getOrderInfoVOSDO")
    @RequestWrapper(targetNamespace="www.globalcompany.example.com/types/",
        localName="getOrderInfoVOSDO")
    @ResponseWrapper(targetNamespace="www.globalcompany.example.com/types/",
        localName="getOrderInfoVOSDOResponse")
    @WebResult(name="result")
    OrderInfoVOSDO getOrderInfoVOSDO(@WebParam(mode = WebParam.Mode.IN, name="orderId")
        BigInteger orderId) throws ServiceException;
    ...
}
```

11.2.2.2 Remote Service Schema File

The remote service schema file is an XML schema file which represents the web service schema, as shown in Figure 11–5.

Figure 11–5 Remote Service Schema File



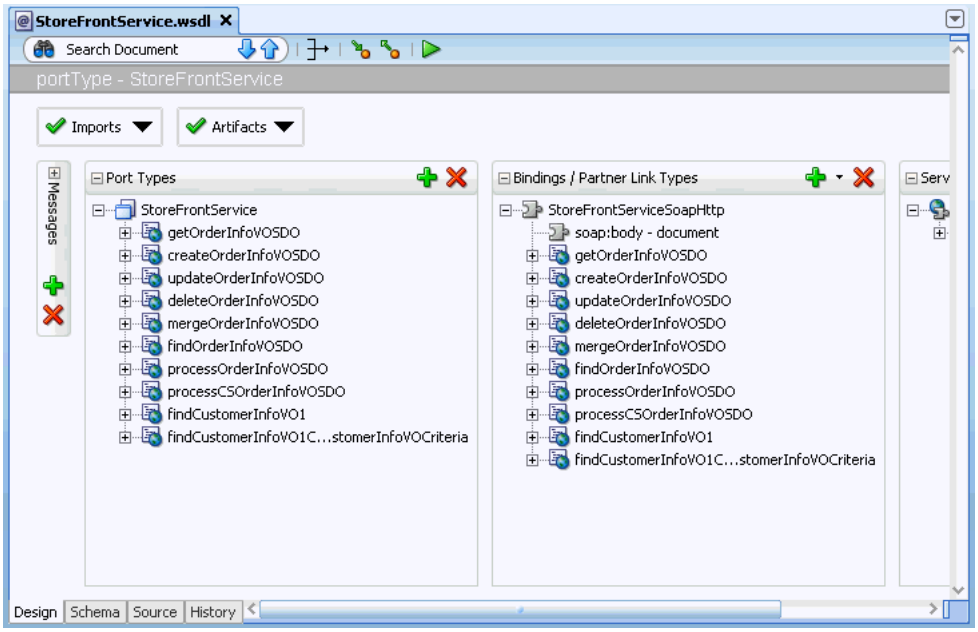
11.2.2.3 Remote Service Definition File

The remote service definition file is a XML-structured document file that conforms to the Web Service Definition Language (WSDL) specification that describes the generated web service as a collection of endpoints, or ports. A port is defined by

associating a network address with a reusable binding. The client application that connects to the web service reads the WSDL to determine what functions are available on the server. The WSDL also specifies the endpoint for the service itself, which you can use to locate and test your deployed service.

Figure 11–6 shows the WSDL for the web service generated for the `StoreServiceAM` application module in the WSDL visual editor. You can see the WSDL as an XML document by selecting the **Source** tab.

Figure 11–6 WSDL Document



11.2.2.4 Remote Server Class

The remote server class is an EJB 3.0 stateless session bean that implements the remote common interface and extends the `ServiceImpl` class, the generic service engine for ADF Business Components. Example 11–2 shows part of `StoreFrontServiceImpl.java`, which is the remote server class for the `StoreServiceAM` application module in the StoreFront module of the Fusion Order Demo.

Example 11–2 Remote Server Class Implements the Remote Common Interface

```
package oracle.fodemo.storefront.store.service.server.serviceinteface;
...
import oracle.fodemo.storefront.store.queries.common.CustomerInfoVOSDO;
import oracle.fodemo.storefront.store.queries.common.OrderInfoVOSDO;
import oracle.fodemo.storefront.store.service.common.serviceinterface.StoreFrontService;

...
import oracle.webservices.annotations.PortableWebService;
import weblogic.javaee.CallByReference;
...
@Stateless(name="oracle.fodemo.storefront.store.service.common.StoreFrontServiceBean")
@Remote(StoreFrontService.class)
@PortableWebService(targetNamespace="http://www.globalcompany.com/StoreFrontService",
    serviceName="StoreFrontService", portName="StoreFrontServiceSoapHttpPort",
    endpointInterface="oracle.fodemo.storefront.service.common.serviceinteface.StoreFrontService")
```

```
@CallByReference
public class StoreFrontServiceImpl extends ServiceImpl implements StoreFrontService
{
    ...
    /**
     * findCustomerInfoVO1: generated method. Do not modify.
     * @param findCriteria
     * @param findControl
     * @return
     * @throws ServiceException
     */
    public List<CustomerInfoVOSDO> findCustomerInfoVO1(FindCriteria findCriteria,
                                       FindControl findControl) throws ServiceException {
        return (List<CustomerInfoVOSDO>)find(findCriteria, findControl,
                                         "CustomerInfoVO1",
                                         CustomerInfoVOSDO.class);
    }
...
}
```

11.2.2.5 connections.xml

The `connections.xml` file is used by a web service client to look up the service. The ADF Business Components service factory provides the mechanism that allows the service client to look up the service. The service factory relies on ADF connection architecture and the `connections.xml` file to manage service endpoint locations. The `connections.xml` file is created when you first create an ADF Business Components service. This file appears in the Application Navigator in the Application Resources panel, under the **Descriptors** and **ADF META-INF** nodes.

Example 11–3 shows the initial `connections.xml` entry created by JDeveloper when you first create an ADF Business Components service.

Example 11–3 connections.xml File Generated by JDeveloper

```
<Reference name="{http://www.globalcompany.com}StoreFrontService"
                     className="oracle.jbo.client.svc.Service" xmlns="">
   <Factory className="oracle.jbo.client.svc.ServiceFactory"/>
   <RefAddresses>
      <StringRefAddr addrType="serviceInterfaceName">
         <Contents>oracle.fodemo.storefront.store.service.common.serviceinterface.
                           StoreFrontService</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="serviceEndpointProvider">
         <Contents>ADFBC</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="jndiName">
         <Contents>oracle.fodemo.storefront.store.service.common.
                           StoreFrontServiceBean</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="serviceSchemaName">
         <Contents>StoreFrontService.xsd</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="serviceSchemaLocation">
         <Contents>oracle/fodemo/storefront/store/service/common/
                                       serviceinterface/</Contents>
      </StringRefAddr>
   </RefAddresses>
</Reference>
```

11.2.3 What You May Need to Know About Method Signatures on the ADF Web Service Interface

You can define two different kinds of interfaces for an application module: the client interface and the service interface. The client interface is used by the ADF Model layer for UI clients. The service interface is for application integration and is used by external web services or other application services (either programmatically or automatically using the service-enabled entity feature).

An application module can support no interface at all, only client interfaces, only service interfaces, or both client interfaces and service interfaces combined. However, be aware that the two kinds of interfaces differ in the data types that are supported for the parameters and/or return values of your custom methods that you define for the respective interfaces. The types supported on the client interface are described in Section 9.9.5, "What You May Need to Know About Method Signatures on the Client Interface."

The service interface, in contrast to the client interface, supports a more narrow set of data types for custom method parameters and return values and is limited to:

- Java primitive types and their object wrapper types (for example, `int`, `Integer`, and `Long`)
- `java.lang.String`
- `java.math.BigDecimal`
- `java.math.BigInteger`
- `java.sql.Date`
- `java.sql.Time`
- `java.sql.Timestamp`
- `java.util.Date`
- `oracle.jbo.AttributeList`
- `oracle.jbo.domain.BlobDomain`
- `oracle.jbo.domain.Char`
- `oracle.jbo.domain.ClobDomain`
- `oracle.jbo.domain.DBSequence`
- `oracle.jbo.domain.Date`
- `oracle.jbo.domain.NClobDomain`
- `oracle.jbo.domain.Number`
- `oracle.jbo.domain.Timestamp`
- `oracle.jbo.domain.TimestampLTZ`
- `oracle.jbo.domain.TimestampTZ`
- `oracle.jbo.server.ViewRowImpl` or any subtype
- `java.util.List<`*`aType`*`>`, where *`aType`* is any of the service-interface supported data types, including Java primitive type

You can define a custom method that returns a type of `AttributeList` when you want the client developer to work with the list of service-enabled entity object or view object attributes to perform custom operations without the need to involve framework

behavior before running the custom method. As an alternative, when the client developer wants the framework to manage rows (create, find, and populate), define custom methods that return `ViewRowImpl` instead. In summary, if your method signature defines `ViewRowImpl` as the data type, then the application automatically:

1. Looks up the row in the corresponding view object instance by primary key and/or alternate key
2. If the row is not found, then creates a new row
3. Applies the attribute changes in the found or new row

Whereas, if your method signature defines the `AttributeList` data type, then no automatic behavior is provided, and the actions performed and data modified by the custom method will be limited to your custom method's code.

11.2.4 How to Service-Enable Individual View Objects

As a result of enabling the web service interface using the overview editor for the application module, JDeveloper automatically enables your parent view instance selections as Service Data Object (SDO) components. The generated SDO components for each view instance will reference the same namespace and will be configured with the same settings for options such as whether or not warnings are supported. You can use the Java page of the overview editor to customize the SDO definition of these existing service-enabled view objects. You can also use the Java page to service-enable view objects that were not added already to the service interface. For example, if you selected a parent view object that represents the master in a master-detail relationship, the child view object will not be automatically service-enabled. You can use the Java page of the overview editor for the child view object to individually add it to the service interface.

You use the Java page of the overview editor for the view object to configure the SDO name and namespace for a view object, or to selectively service-enable child view objects.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Section 11.2, "Publishing Service-Enabled Application Modules."

Complete these tasks:

1. Create the desired view objects, as described in Section 5.2.1, "How to Create an Entity-Based View Object," and Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object."
2. Optionally, set JDeveloper preferences to specify a default suffix for the names of generated SDO classes and set the default namespace prefix for the generated SDO schema and web service as described in Section 11.2.13, "How to Set Preferences for Generating the ADF Web Service Interface."

To set the SDO name and namespace for a view object:

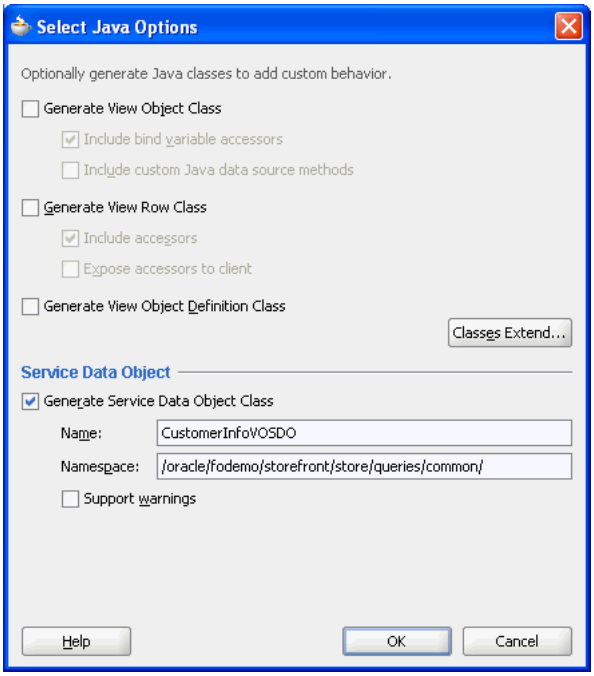
1. In the Application Navigator, double-click the view object.
2. In the overview editor, click the **Java** navigation tab and click the **Edit java options** button.

3. In the Select Java Options dialog, in the **Service Data Object** section, select **Generate Service Data Object Class**. Enter a value for the service data object name and the service data object's target namespace.

   The target namespace is a URI for the SDO component that you can assign to group similar SDO components together by entering the same URI.

   A default SDO namespace is created for you based on the SDO component's package name with periods replaced by "/". If you have defined a prefix for the namespace in the View Objects page of the Preferences dialog, the prefix will be added at runtime to the beginning of the namespace. For example, Figure 11–7 shows the default namespace based on the package name.

Figure 11–7 Service Data Object Name and Namespace Options



4. Optionally, select **Support Warnings** when you want to be able to extract warnings associated with the view rows of the service interface object. For example, your view object might have a range validator defined on the `Salary` attribute and failure handling for the validator is specified as Informational Warning.

   If **Support Warnings** is not selected, no informational messages will be returned with the service response.

   When enabled and a warning is generated from the underlying ADF business component object for one of the standard service operations or custom operations, the warning information will be captured by the response object. You can use the methods generated for the service object result class to extract the messages from the view rows, as described in Section 11.2.7.4, "Service Data Object Result Class and Interface."

5. Click **OK**.

11.2.5 How to Customize the SDO Properties of Service-Enabled View Objects

You can use the overview editor for the view object to customize the SDO component definition of the service-enabled view object. By default, all attributes of the service-enabled view object will be exposed as SDO properties. By customizing the view object definition, you can exclude individual SDO properties from participating in the service interface. In the case of SDO properties that define numeric values, you can associate two properties so they appear as a single complex type in the service interface. For example, you can associate one property that defines a currency code or unit of measure with another property that displays the numeric value. Currently, only the complex service types `AmountType` (a currency code) and `MeasureType` (a unit of measure) are supported.

11.2.5.1 Excluding Individual SDO Properties in a Generated SDO Component

As a result of enabling the web service interface using the overview editor for the application module, JDeveloper automatically enables your parent view instance selections as SDO components. Additionally, you can selectively service-enable individual child view objects and generate SDO components. By default, generated SDO components expose all attributes of their base view object definition as SDO properties. You can hide any attribute that you do not want the service interface to return as an SDO property.

You use the Attributes page of the overview editor to select the view object attribute that you want to exclude from the service interface. You then use the Details tab in the overview editor for the view object to hide the selected attribute from the SDO component.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Section 11.2, "Publishing Service-Enabled Application Modules."

You will need to complete these tasks:

1. Create the desired view objects, as described in Section 5.2.1, "How to Create an Entity-Based View Object," and Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object."
2. Service-enable the desired view object, as described in Section 11.2.4, "How to Service-Enable Individual View Objects."

To exclude an SDO property from a service-enabled view object:

1. In the Application Navigator, double-click the view object.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page of the overview editor, select the attribute corresponding to the property that you want to exclude and then click the **Details** tab and deselect **SDO Property**.

11.2.5.2 Associating Related SDO Properties Using Complex Data Types

As a result of service-enabling the view object, JDeveloper automatically exposes SDO properties as XSD-defined service types that correspond to the data types of the underlying view object's attributes. In the case of attributes that define numeric values, you can change the SDO property's service type to associate a related property using one of these predefined service types:

- `AmountType` service type, for use with any property that defines a currency code
- `MeasureType` service type, for use with any property that defines a unit of measure

When you change the service type of an SDO property to either of these complex types, the service interface associates the two properties together and returns them as a single XML element. Both properties of the SDO component must be defined by attributes in the base service-enabled view object.

For example, suppose that your view object defines the `OrderTotal` attribute and a `CurrencyCode` attribute to specify the currency code of allowed countries. By default, the service interface exposes these attributes as SDO properties and returns each property as a separate XML element:

```
<OrderTotal>100.00</Price>
<CurrencyCode>USD</CurrencyCode>
```

If you change the type of the `OrderTotal` property (assume that the XSD file defines this property as a `decimal` type) to the complex type `AmountType` and then associate the `CurrencyCode` property, the service interface will return them as one XML element:

```
<OrderTotal CurrencyCode="USD">123.00</OrderTotal>
```

Also, when you generate a web service proxy, as described in Section 9.12.1.3, "Calling a Web Service Method Using the Proxy Class in an Application Module," the class treats the two values as one object:

```
AmountType price;
...
price.setValue(123.00);
price.setCurrencyCode("USD");
```

You use the Attributes page of the overview editor to select the view object attribute whose service type you want to customize. You use the Edit Attribute dialog that you display from the Attributes page of the overview editor to associate SDO properties for the selected attribute and select the predefined complex service type.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Section 11.2, "Publishing Service-Enabled Application Modules."

Complete these tasks:

1. Create the desired view objects, as described in Section 5.2.1, "How to Create an Entity-Based View Object," and Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object."
2. Service-enable the desired view object, as described in Section 11.2.4, "How to Service-Enable Individual View Objects."

To associate SDO properties in a service-enabled view object:

1. In the Application Navigator, double-click the view object.
2. In the overview editor, click the **Attributes** navigation tab.

3. In the Attributes page, select the attribute corresponding to the property that you will associate with another SDO property, and then click the **Edit selected attribute** button.

   The attribute you select must define a numeric type. For example, to associate a currency code with the attribute that displays the amount paid by a customer, you might select the `OrderTotal` attribute in the `Orders` service-enabled view object.

4. In the Edit Attribute dialog, in the View Attribute page, choose the desired XSD Type.

   If the **XSD Type** field is not enabled, return to the overview editor and select an attribute of type numeric. Attributes whose values are not a numeric type cannot be associated with the available complex service types.

   The SDO framework supports the complex service types `AmountType` and `MeasureType`. Choose **AmountType** when the property you want to associate specifies currency information. Choose **MeasureType** when the property you want to associate specifies a unit of measure.

5. In the **currencyCode** or **unitCode** dropdown list, select the view object attribute to define the complex type.

   The dialog changes to display the dropdown list appropriate to the XSD type selection. You can choose any numeric attribute that the view object defines.

6. Click **OK**.

11.2.6 How to Support Nested Processing in Service-Enabled Master-Detail View Objects

When your data model defines master-detail relationships between parent and child view objects, the service operations that you enable for the master view object may not automatically be executed on the detail view object. Post operations on the detail view object are supported by default when the primary entity object of the master view object is composed with the primary entity object of the detail view object. This master-detail relationship is known as a composition association and is the default type associate created in JDeveloper, as described in Section 4.3.6, "What You May Need to Know About Composition Associations."

To support create/merge/update/process methods that post child details along with the parent, you you have created a view link to define the master-detail relationship according to one of these scenarios:

- The view link uses the default composition association, then post operations on the detail view object are supported by default.
- The view link is based on an association, and the association has the destination accessor generated, and the association has a custom property `SERVICE_PROCESS_CHILDREN=true` defined.
- The view link is not based on an association but has a custom property `SERVICE_PROCESS_CHILDREN=true` defined.

The custom property provides an alternative to using a composition association that makes it convenient to support nested processing for any view objects with a view link defined. You can define `SERVICE_PROCESS_CHILDREN` as a custom property in the overview editor for either the view link or the view link's association (when present).

To support get and find methods that retrieve child details along with the parent, the view link between the master and detail view objects must have the destination

accessor generated. The destination accessor permits traversal of the hierarchy from the master to the detail view object.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Section 11.2, "Publishing Service-Enabled Application Modules."

You may also find it helpful to have an understanding of how master-detail relationships are defined using view links or entity associations:

- For more information about view links, see Section 5.6, "Working with Multiple Tables in a Master-Detail Hierarchy."
- For more information about associations, see Section 4.3, "Creating and Configuring Associations."

Complete these tasks:

1. Create the desired view objects and service-enable the child view object in the master-detail hierarchy, as described in Section 11.2.4, "How to Service-Enable Individual View Objects."
2. If the view link is not based on an association, confirm that a destination accessor exists for the view link by opening the view link in the overview editor and viewing the Relationship page. To generate the accessor so it appears in the Relationship page, click the **Edit Accessors** button, and then, in the View Link Properties dialog, select **Generate Accessor in View Object** for the destination accessor.

   If the view link is based on an association, then the destination accessor must exist for the association's destination entity object. To generate one, use the Relationship page of the overview editor for the association.

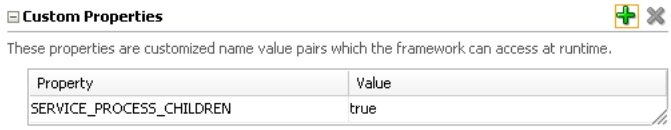
To support nested processing in a master-detail hierarchy:

1. If the view link for the master-detail hierarchy is not based on an association, then in the Application Navigator, double-click the view link; otherwise, if the view is based on an association, then in the Application Navigator, double-click the association.

   You can confirm how the view link was created in the Relationship page of the overview editor. The Attributes section names the source and destination attributes. When the view link is based on an association, the attribute hyperlinks will contain the names of the association. Otherwise, the hyperlinks will contain the names of the base entity objects.
2. In the overview editor, click the **General** navigation tab.

   The overview editor for the view link and the association display similar selections.
3. In the General page, expand the Custom Properties section, and then click the **Add Custom Property** icon and enter `SERVICE_PROCESS_CHILDREN` for the property and enter `true` for the property value, as shown in Figure 11–8.

Figure 11–8 Custom Property to Support Nested Processing



11.2.7 What Happens When You Create SDO Classes

When you create SDO classes, the following files are generated and appear in the Application Navigator under the owning view object:

- Service data object interface
- Service data object class
- Service data object schema file
- Service data object result class and Interface, generated when **Support Warnings** is enabled in the Select Java Options dialog

Do not modify the files generated for service-enabled view objects. The generated files implement required methods of the view object SDO interface.

11.2.7.1 Service Data Object Interface

The view object SDO interface contains strongly typed accessors for the SDO properties, as shown in Example 11–4.

Example 11–4 SDO Interface Contains Strongly Typed Accessors for SDO Properties

```
package oracle.fodemo.storefront.store.queries.common;
  public interface AddressesVOSDO {
  public java.math.BigInteger getAddressId();
  public void setAddressId(java.math.BigInteger value);
...}
```

11.2.7.2 Service Data Object Class

The view object SDO class implements the view object SDO interface and extends the `SDODataObject` class, which is Oracle's implementation of the SDO specification.

At runtime an instance of an SDO object represents a row in memory.

The SDO class is similar to the view row class, as shown in Example 11–5.

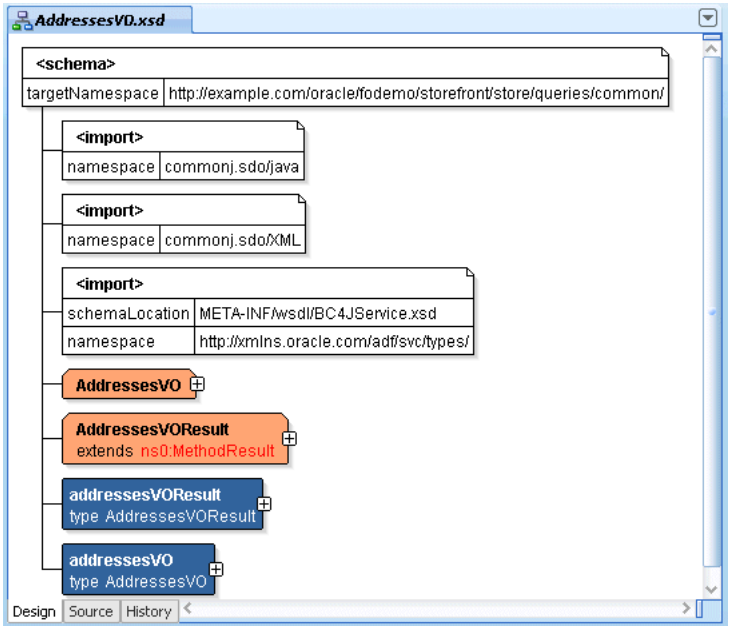
Example 11–5 SDO Class Implements View Object SDO Interface

```
package oracle.fodemo.storefront.store.queries.common;
import commonj.sdo.Type;
import oracle.sdo.SDODataObject;
public class AddressesVOSDOImpl extends SDODataObject implements AddressesVOSDO
  {
  ...
  }
```

11.2.7.3 Service Data Object Schema File

The view object SDO schema file, as shown in Figure 11–15, is an XML Schema file which represents the SDO schema.

Figure 11–9 Generated SDO Schema



11.2.7.4 Service Data Object Result Class and Interface

The view object SDO result class is a container object that allows a service operation to return a list of view rows (wrapped in service data objects) and a list of warnings associated with these view rows. Specifically, the service get operation returns the original object, while the create/update/merge/find/process operations return a wrapper object that contains a list of the original object and a list of information messages, and the delete operation returns only the informational message. If you have enabled the **Support Warnings** option for the service-enabled view object, you can use the generated method result interface to extract warnings.

The view object SDO result class, as shown in Example 11–6, is similar to the view row class.

Example 11–6 SDO Result Class Defines Methods to Get Warnings from List

```
package oracle.fodemo.storefront.store.queries.common;
import oracle.sdo.SDODataObject;
public class OrderInfoVOSDOResultImpl extends
   oracle.jbo.common.service.types.MethodResultImpl implements OrderInfoVOResult {

   public static final int START_PROPERTY_INDEX =
          oracle.jbo.common.service.types.MethodResultImpl.END_PROPERTY_INDEX + 1;
   public static final int END_PROPERTY_INDEX = START_PROPERTY_INDEX + 0;
   public OrderInfoVOResultImpl() {}
   public java.util.List getValue() {
      return getList(START_PROPERTY_INDEX + 0);
   }

   public void setValue(java.util.List value) {
      set(START_PROPERTY_INDEX + 0 , value);
   }
}
```

11.2.8 How to Expose a Declarative Find Operation Filtered By a Required Bind Variable

The ADF service interface framework allows you to expose declarative find operations to execute the query define by a view object you select. However, when that query uses a bind variable to filter the query results, the bind variable must not be specified as **Required** and **Updatable**. Because the service interface does not expose required, updatable bind variables, a find operation that you execute for such a view object would fail to return any result.

When you want to filter a query result using bind parameters, use the view criteria and expose it as a find operation on the service interface. A service interface find operation based on a view criteria that you create can specify required bind variables.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Section 11.2, "Publishing Service-Enabled Application Modules."

You may also find it helpful to have an understanding of how to filter view object queries using bind variables. For more information, see Section 5.10, "Working with Bind Variables."

Complete these tasks:

1. Create the desired application module as described in Section 9.2.1, "How to Create an Application Module."
2. Create the view criteria as described in Section 5.11.1, "How to Create Named View Criteria Declaratively." In the Edit View Criteria dialog, set the criteria item as a bind variable and set the **Validation** field to **Required**. This selection ensures that the query will not execute without a valid value.

To expose a find operation for a view criteria with required bind variable:

1. In the Application Navigator, double-click the application module.
2. In the overview editor, click the **Service Interface** navigation tab and click the **Edit attributes of Service Interface** icon.

   Alternatively, you can select **Edit Service Custom Methods** if you have already defined the service interface.
3. In the Edit Service Interface dialog, select **Service View Instances** from the navigation list and add the view object that you want to filter with its named view criteria to the **Selected** list.
4. To expose the find operation, select the view instance, click the **View Criteria Find Operations** tab, and then click the **Add View Criteria** button.
5. In the Configure View Criteria Find Operation dialog, choose the named view criteria for the find operation.

   The dialog displays the bind variable for the selected view criteria.
6. You can double-click the XML name to customize the bind variable name shown in the XML definition for the service.
7. Click **OK**.

11.2.9 How to Expose a Custom Find Method Filtered By a Required Bind Variable

As an alternative to exposing a declarative find operation that relies on a view criteria, you can define a service method in your data model project's application module implementation class. The class you create for this purpose allows you to encapsulate business service functionality into a single method that you implement. For details about the purpose of the custom application module implementation class, see Section 9.7, "Customizing an Application Module with Service Methods."

Example 11–7 shows a custom find method implemented in the *AppModuleName*`Impl.java` file to set the bind variable and execute the view object instance query. It uses `setNamedWhereClauseParam()` on the view object instance to set the bind variable. Before executing the query, the find method sets the view object in forward-only mode to prevent caching the view rows that the find method iterates over.

Example 11–7 Find Method Added to Application Module Implementation Class

```
public class AppModuleImpl extends ApplicationModuleImpl
{
   public List<ViewRowImpl> findProducts(String location)
   {
      List<ViewRowImpl> result = new ArrayList<ViewRowImpl>();
      ViewObjectImpl vo = getProductsView1();
      vo.setNamedWhereClauseParam("TheLocation", location);
      vo.setForwardOnly(true);
      vo.executeQuery();
      while (vo.hasNext()) {
        result.add((ViewRowImpl)vo.next());
      }
      return result;
   }
}
```

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled view objects and enables web service clients to access rows of data and perform service operations. For more information, see Section 11.2, "Publishing Service-Enabled Application Modules."

You may also find it helpful to have an understanding of how to filter view object queries using bind variables. For more information, see Section 5.10, "Working with Bind Variables."

Complete these tasks:

1. Create the custom application module class as described in Section 9.7.1, "How to Generate a Custom Class for an Application Module."
2. Create the custom find method to programmatically filter a query result and set the required bind variable as described in Section 5.10.5, "How to Add a WHERE Clause with Named Bind Variables at Runtime."

To expose a find method that sets a required bind variable:

1. In the Application Navigator, double-click the application module.
2. In the overview editor, click the **Service Interface** navigation tab and click the **Edit attributes of Service Interface** icon.

Alternatively, you can click the **Edit Service Custom Methods** icon if you have already defined the service interface.

3. In the Edit Service Interface dialog, select **Service View Instances** from the navigation list and add the find method that you defined to the **Selected** list.
4. Click **OK**.

11.2.10 How to Generate Asynchronous ADF Web Service Methods

By default, the web service supports synchronous service methods. This forces the invoking client application to wait for the response to return before it can continue with its work. In cases where the response returns immediately, this method of invoking the web service is common. However, because request processing can be delayed, it is often useful for the client application to continue its work and to handle the response later on.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled view objects and enables web service clients to access rows of data and perform service operations. For more information, see Section 11.2, "Publishing Service-Enabled Application Modules."

You may also find it helpful to have an understanding of invoking web services using asynchronous request-response. For more information, see *Oracle Fusion Middleware Concepts Guide for Oracle Infrastructure Web Services*.

Complete this task:

Before you can deploy an asynchronous web service, you must configure the queues used to store the request and response. For information about configuring the request and response queues, see the *Oracle Fusion Middleware Concepts Guide for Oracle Infrastructure Web Services*.

To expose asynchronous web service methods:

1. In the Application Navigator, double-click the application module.
2. In the overview editor, click the **Service Interface** navigation tab and click the **Edit attributes of Service Interface** icon.
3. In the Edit Service Interface dialog, select **Generate Asynchronous Web Service Methods**.
4. Click **OK**.

11.2.11 What Happens When You Generate Asynchronous ADF Web Service Methods

JDeveloper generates the remote common interface for the service and enables the asynchronous service operation. As shown in Example 11–8, the class annotation `@AsyncWebService` declares the `EmpService` service interface asynchronous and for each synchronous method in the interface, the service exposes an asynchronous method with the same method name and "`Async`" appended.

Exposing both synchronous and asynchronous methods in the same interface allows the web service client developer to decide how to invoke the operation through a web service proxy: by calling the appropriately named method. Note that developers should not invoke asynchronous methods through the ADF Business Components service proxy directly.

In this example, because the `EmpService` service is enabled for asynchronous operation, the interface exposes the `getEmployeeAsync()` method and declares the `getEmployee()` method synchronous using the method annotation `@CallbackMethod(exclude=true)` to override the default operation (it is the `exclude=true` part that declares a method in the asynchronous service as synchronous). No annotation is required to declare the asynchronous service methods when the class annotation `@AsyncWebService` is present.

Example 11–8 Remote Common Interface with Asynchronous Service Methods

```
import javax.xml.ws.Action;
...
import oracle.webservices.annotations.async.AsyncWebService;
import oracle.webservices.annotations.async.CallbackMethod;

@SOAPBinding(parameterStyle=SOAPBinding.ParameterStyle.WRAPPED, style=SOAPBinding.Style.DOCUMENT)
@PortableWebService(targetNamespace="http://xmlns.example.com/apps/service/", name="EmpService",
                    wsdlLocation="oracle/apps/service/EmpService.wsdl")
@SDODatabinding(schemaLocation="oracle/apps/service/EmpService.xsd")
@AsyncWebService
public interface EmpService
{
   ...

   @WebMethod(action="http://xmlns.example.com/apps/service/getEmployee",
                       operationName="getEmployee")
   @RequestWrapper(targetNamespace="http://xmlns.example.com/apps/service/types/",
                       localName="getEmployee")
   @ResponseWrapper(targetNamespace="http://xmlns.example.com/apps/service/types/",
                       localName="getEmployeeResponse")
   @WebResult(name="result")
   @CallbackMethod(exclude=true)
   Emp getEmployee(@WebParam(mode = WebParam.Mode.IN, name="empno") Integer empno)
      throws ServiceException;

   @WebMethod(action="http://xmlns.example.com/apps/service/getEmployeeAsync",
                        operationName="getEmployeeAsync")
   @RequestWrapper(targetNamespace="http://xmlns.example.com/apps/service/types/",
                        localName="getEmployeeAsync")
   @ResponseWrapper(targetNamespace="http://xmlns.example.com/apps/service/types/",
                        localName="getEmployeeAsyncResponse")
   @WebResult(name="result")
   @Action(input="http://xmlns.example.com/apps/service/getEmployeeAsync",
           output="http://xmlns.example.com/apps/service/getEmployeeAsyncResponse")
   Emp getEmployeeAsync(@WebParam(mode = WebParam.Mode.IN, name="empno") Integer empno);
}
```

The duplicate asynchronous methods delegate to the synchronous methods in the service implementation, as shown in Example 11–9. This ensures that the underlying business logic is the same for operations declared as either synchronous or asynchronous.

Example 11–9 Remote Server Class Implements Asynchronous Service Methods

```
...
import oracle.webservices.annotations.async.AsyncWebService;

@Stateless(name="oracle.apps.service.EmpServiceBean", mappedName="EmpServiceBean")
@Remote(EmpService.class)
@PortableWebService(targetNamespace="http://xmlns.oracle.com/apps/service/",
```

```
                    serviceName="EmpService", portName="EmpServiceSoapHttpPort",
                    endpointInterface="oracle.apps.service.EmpService")
@Interceptors(ServiceContextInterceptor.class)
@AsyncWebService
public class EmpServiceImpl extends ServiceImpl implements EmpService
{
   ...

   /**
    * getEmployee: generated method. Do not modify.
    */
   public Emp getEmployee(Integer empno)
      throws ServiceException
   {
      return (Emp) get(new Object[] { empno }, "Employee", Emp.class);
   }

   /**
    * getEmployeeAsync: generated method. Do not modify.
    */
   public Emp getEmployeeAsync(Integer empno)
      throws ServiceException
   {
      return getEmployee(empno);
   }
}
```

11.2.12 What Happens at Runtime: How the Asynchronous Call Is Made

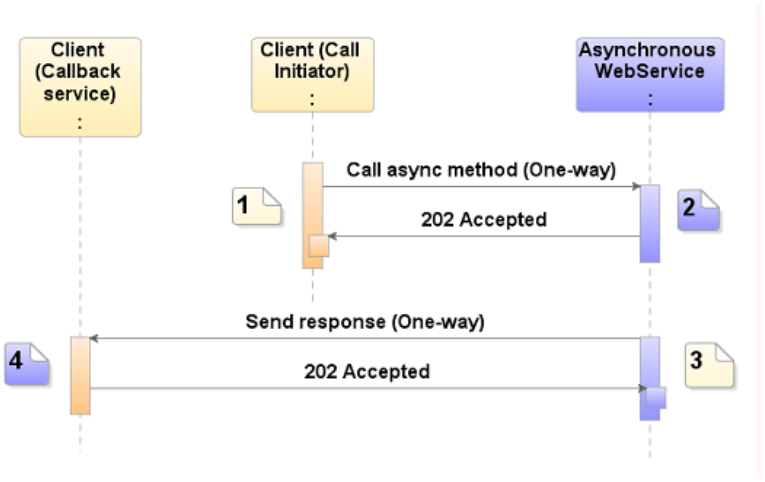
From the client's point of view, an asynchronous call consists of two one-way message exchanges. The sequence diagram in Figure 11–10 depicts the following flow:

1. The client calls for the asynchronous operation. (In the figure, Step 1.)
2. The asynchronous service receives the request and returns the HTTP acknowledgement back to the client without actually processing the request. (In the figure, step 2)
3. Eventually the asynchronous operation will complete and the module on the server side will send the response to the client side. (In the figure, step 3.)

   To receive the response at the client side, the client must have some kind of HTTP listener, for example, a servlet or a web service.
4. The client side-generated web service (the Callback Service) receives the asynchronous responses. (In the figure, step 4.)

   The module in step 3 on the server side acts like a client to the callback service and so is referred as the callback client.

Figure 11–10 Asynchronous Call Sequence



11.2.13 How to Set Preferences for Generating the ADF Web Service Interface

You have additional control of the service generated by JDeveloper. You can set JDeveloper preferences to use a default suffix for the names of generated SDO classes, modifying the default subpackage where the service common interface and classes go.

To set the SDO class name suffix:

1. From the main menu, choose **Tools** > **Preferences**.
2. In the Preferences dialog, expand **ADF Business Components** and choose **Class Naming**.
3. In the **View Object** suffix list, enter a suffix for **SDO**, for example, `SDO`.

To set the default subpackage for the generated service interface:

1. From the main menu, choose **Tools** > **Preferences**.
2. In the Preferences dialog, expand **ADF Business Components** and choose **Packages**.
3. In the **Relative Package Specification for Classes** list, specify the default package names:
   - To set the Service Interface package name, enter a value for the **Client Interface**. (The **Service Interface** displays the same package name you specify for the client interface). The default package name is `common`.
   - Enter a value for the **Service Interface Subpackage** of the **Service Interface**. The default subpackage name is `serviceinterface`.

   For example, if you enter `common` for **Service Interface** and `serviceinterface` for **Service Interface Subpackage** (the defaults), service interfaces for data model components in the data model package `oracle.storefront.store.service` will be placed in the subpackage `oracle.storefront.store.service.common.serviceinterface`.

To set the default namespace prefix for the generated SDO schema and web service:

1. From the main menu, choose **Tools** > **Preferences**.

2. In the Preferences dialog, expand **ADF Business Components** and choose **View Objects**.
3. Enter a value for the **Service Data Object Namespace Prefix** to be added to the beginning of the target namespace of the generated SDO schema and web service. For example, `http://example.com/`.

11.2.14 How to Secure the ADF Web Service for Access By SOAP Clients

At runtime, the web service client will invoke the service-enable methods of the application module through the SOAP protocol. You can configure a Oracle Web Service Manager (Oracle WSM) security policy to enable authentication and authorization on the service. The security policy that you select will require the SOAP client call to provide credential information (or SAML token) as part of the SOAP header. You can also configure other policies to enable message protection (integrity and confidentiality) for inbound SOAP requests, for instance.

To secure the web service for SOAP clients:

1. Configure an Oracle WSM authentication policy.
2. Configure an Oracle WSM authorization policy.

You can enable authentication to require users to supply credentials before they have access to the service methods on the service interface. The type of authentication required is configured on the remote server class using an Oracle WSM authentication policy annotation.

You can enable permission checking to enable only users with sufficient privileges to invoke a service method on the service interface. Permission checking is configured on the remote server class using this Oracle WSM authorization annotation:

- `binding_permission_authorization_policy`

  This policy provides simple permission-based authorization for the request based on the authenticated Subject at the SOAP binding level. This policy ensures that the Subject has permission to perform the operation. This policy should follow an authentication policy where the Subject is established and can be attached to any SOAP-based endpoint.

As an alternative to the permission checking policy, you can configure one of these role-based Oracle WSM security policies:

- `binding_authorization_denyall_policy`

  This policy provides simple role-based authorization for the request based on the authenticated Subject at the SOAP binding level. This policy denies all users with any roles. It should follow an authentication policy where the Subject is established and can be attached to any SOAP-based endpoint.

- `binding_authorization_permitall_policy`

  This policy provides a simple role-based authorization for the request based on the authenticated Subject at the SOAP binding level. This policy permits all users with any roles. It should follow an authentication policy where the Subject is established and can be attached to any SOAP-based endpoint.

For further details about the authorization policies, see the *Oracle Fusion Middleware Security and Administrator's Guide for Web Services*.

ADF service interface framework also provides an EJB interceptor `ServicePermissionCheckInterceptor` which does the same permission check as `oracle/binding_permission_authorization_policy`, and additionally

covers RMI invocation to services. If your services will always be invoked through SOAP, you should attach the Oracle WSM `oracle/binding_permission_authorization_policy` authorization policy. If your services will also be invoked through RMI, you should attach the EJB interceptor `ServicePermissionCheckInterceptor`.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Section 11.2, "Publishing Service-Enabled Application Modules."

You may also find it helpful to have an understanding of the predefined authentication policies supported by Oracle WSM. For more information, see the *Oracle Fusion Middleware Security and Administrator's Guide for Web Services*.

Complete this task:

- Grant users access to the service, as described in Section 11.2.16, "How to Grant Test Users Access to the Service."

To configure an authentication and authorization:

1. In the Application Navigator, expand the application module, expand the **serviceinterface** node, and then double-click the remote server class (***AppModule*ServiceImpl.java** file).

   In the web service generated from the `StoreServiceAM` application module in the `StoreFrontModule` application of the Fusion Order Demo, the remote server class is `StoreFrontServiceImpl.java`.

2. In the source for the remote server class, place your cursor on the `@PortableWebService` annotation.

   For example, `StoreFrontServiceImpl.java` shows the annotation for the service as follows:

   ```
   ...
   @PortableWebService(targetNamespace="www.globalcompany.example.com",
       serviceName="StoreFrontService", portName="StoreFrontServiceSoapHttpPort",
       endpointInterface=
          "oracle.fodemo.storefront...common.serviceinterface.StoreFrontService")
   ...
   ```

3. Open the Property Inspector and expand the **Web Services Extension** section, and then click the ellipsis button next to the **Security** field for Oracle WSM Policies.

4. In the Edit Property: Security dialog, select the desired authentication security policy and click **OK**.

   For details about the available security policies supported by Oracle WSM, see the *Oracle Fusion Middleware Security and Administrator's Guide for Web Services*.

5. Return to the source file and note that new annotation `@SecurityPolicy` has been added.

   The `@SecurityPolicy` annotation that you define for the remote server class specifies the security requirements to potential clients. For example, if you had selected **oracle/wss11_saml_or_username_token_with_message_protection_service_policy** in the dialog, the following `@SecurityPolicy` annotation and `@CallbackSecurityPolicy` (for asynchronous service) would appear below the `@PortableWebService` annotation:

```
...
@PortableWebService(targetNamespace="www.globalcompany.example.com",
    serviceName="StoreFrontService",
    portName="StoreFrontServiceSoapHttpPort",
    endpointInterface=
    "oracle.fodemo.storefront.store.service.common.
                                serviceinterface.StoreFrontService")
@SecurityPolicy( { "oracle/wss11_saml_or_username_token_with_message_protec
tion_service_policy" })
@CallSecurityPolicy("oracle/wss11_saml_or_username_token_with_message_protec
tion_service_policy")
...
```

6. If your services will always be invoked exclusively through SOAP (and not RMI), you should use an Oracle WSM authorization policy as follows:

   a. Open the Property Inspector and expand the **Web Services Extension** section, and then click the ellipsis button next to the **Security** field for Oracle WSM Policies.

   b. In the Edit Property: Security dialog, select the desired security policy and click **OK**.

      For details about the security policies supported by Oracle WSM, see the *Oracle Fusion Middleware Security and Administrator's Guide for Web Services*.

   c. Return to the source file and note that annotation `@SecurityPolicy` is configured.

      The `@SecurityPolicy` annotation that you define for the remote server class specifies the security requirements to potential clients. In this example, the annotation shows both the permission-checking authorization policy (`oracle/binding_permission_authorization_policy`) and an authentication policy:

      ```
      ...
      @PortableWebService(targetNamespace="www.globalcompany.example.com",
          serviceName="StoreFrontService",
          portName="StoreFrontServiceSoapHttpPort",
          endpointInterface=

      "oracle.fodemo.storefront...common.serviceinterface.StoreFrontService")
             @SecurityPolicy( { "oracle/binding_permission_authorization_policy",
                         "oracle/wss_username_token_service_policy" })
      ```

   If your services will be invoked through SOAP and RMI, you should use `ServicePermissionCheckInterceptor` only (there is no need to use an Oracle WSM authorization policy) as follows:

   a. In the source editor, place your cursor in the `@Interceptors` annotation and add `ServicePermissionCheckInterceptor.class` to enable permission checking at runtime.

      ```
      ...
      @SecurityPolicy( { "oracle/wss11_saml_or_username_token_with_message_
      protection_service_policy" })
      @CallSecurityPolicy("oracle/wss11_saml_or_username_token_with_message_
      protection_service_policy")
      @Interceptors({ServiceContextInterceptor.class,
                              ServicePermissionCheckInterceptor.class})
      ```

7. Save the remote server class file.

11.2.15 How to Secure the ADF Web Service for Access By RMI Clients

Because the ADF web service is implemented as an EJB and deployed on Oracle WebLogic Server as Oracle Web Service's `PortableWebService`, the client application can invoke the service-enable methods of the application module through the RMI protocol.

To secure the web service for RMI clients:

1. Configure JNDI context properties to enable authentication.
2. Enable permission checking to configure an authorization policy.

Before you begin:

It may be helpful to have an understanding of the predefined authorization policies supported by Oracle WSM. For more information, see the *Oracle Fusion Middleware Security and Administrator's Guide for Web Services*.

11.2.15.1 Enabling Authentication for RMI Clients

When the ADF web service is invoked through RMI, authentication is handled with the common JAAS login module. The login module can be passed the principal and credential as part of the JNDI initial context for the EJB in the calling application. If you do not define the JNDI context properties, the login module will attempt to obtain the caller's current security context.

When you choose to define remote JNDI context information, then these four JNDI context properties need to be added to the `connections.xml` file.

> **Note:** When you intend to test the service in JDeveloper using Integrated WebLogic Server, before deploying the service you can edit the JNDI context properties in the `connections.xml` file directly. However, when you deploy the service to standalone Oracle WebLogic Server, you will use Oracle Enterprise Manager to configure the JNDI context properties.

- `jndiFactoryInitial` should be set to `weblogic.jndi.WLInitialContextFactory`.
- `jndiProviderURL` is the JNDI provider URL that indicates the location of the JNDI server. The URL should be composed as `t3://<hostname>:<server port>`.

  When you test the service in JDeveloper, and your service is deployed to Integrated WebLogic Server, specify the JNDI provider URL of Integrated WebLogic Server: `t3://<hostname>`:7101.

  When you deploy the service to remote Oracle WebLogic Server, specify a URL like: `t3://localhost:8888`, where `t3` is the Oracle WebLogic protocol, `localhost` is the host name that the remote Oracle WebLogic Server instance runs in, `8888` is the port number.
- `jndiSecurityPrincipal` specifies the principal (user name) with permission to access the remote JNDI.

  As Example 11–10 shows, when you test the service in JDeveloper Integrated WebLogic Server, you should omit this context property since no security is configured for the JNDI server on Integrated WebLogic Server.

As Example 11–11 and Example 11–12 show, when you deploy the service to standalone Oracle WebLogic Server, the user name can be read from the file.

- `jndiSecurityCredentials` specifies the credentials (password) to be used for the security principal.

  As Example 11–10 shows, when you test the service in JDeveloper Integrated WebLogic Server, you should omit this context property since no security is configured for the JNDI server on Integrated WebLogic Server.

  As Example 11–11 shows, when you deploy the service to standalone Oracle WebLogic Server in a test environment, you can specify credentials in plain text for the JNDI provider. For example, you can specify `weblogic/weblogic1` which are the default administrator user name/password credentials with sufficient privileges to access JNDI provider for Oracle WebLogic Server.

  When you deploy the service to a production environment, you must remove the plain text password to avoid creating a security vulnerability. As Example 11–12 shows, the `connections.xml` file must contain `<SecureRefAddr addrType="jndiSecurityCredentials"/>` with no password. To configure the service password for standalone Oracle WebLogic Server, you must use Oracle Enterprise Manager, which will store the encrypted password in Oracle's credential store.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Section 11.2, "Publishing Service-Enabled Application Modules."

To configure JNDI context properties to handle authentication:

1. In the Application Navigator in the Application Resources panel, expand the **Descriptors** and **ADF META-INF** nodes, and then double-click the `connections.xml` file.
2. If the source editor, use the JNDI context properties to specify the principal and credentials.

   If you are testing the service in JDeveloper's Integrated WebLogic Server, you only need to specify the `jndiProviderURL` property, as shown in Example 11–10.

Example 11–10 JNDI Properties for JDeveloper Integrated WebLogic Server

```
<References xmlns="http://xmlns.oracle.com/adf/jndi">
  <Reference name="{www.globalcompany.com}StoreFrontService"
             className="oracle.jbo.client.svc.Service" xmlns="">
    <Factory className="oracle.jbo.client.svc.ServiceFactory"/>
    <RefAddresses>
      ...
      <StringRefAddr addrType="jndiFactoryInitial">
         <Contents>weblogic.jndi.WLInitialContextFactory</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="jndiProviderURL">
          <Contents>t3://a_hostname:7101</Contents>
      </StringRefAddr>
      ...
    </RefAddresses>
  </Reference>
  ...
```

```
</References>
```

If you are deploying the service for testing purposes to standalone Oracle WebLogic Server, you can use the `connections.xml` file to specify credentials for the JNDI provider. For example, as shown in Example 11–11, you can specify `weblogic/weblogic1` which are the default administrator user name/password credentials with sufficient privileges to access JNDI provider for Oracle WebLogic Server.

Example 11–11 JNDI Properties for a Test Environment

```
<References xmlns="http://xmlns.oracle.com/adf/jndi">
  <Reference name="{www.globalcompany.com}StoreFrontService"
             className="oracle.jbo.client.svc.Service" xmlns="">
    <Factory className="oracle.jbo.client.svc.ServiceFactory"/>
    <RefAddresses>
      ...
      <StringRefAddr addrType="jndiFactoryInitial">
         <Contents>weblogic.jndi.WLInitialContextFactory</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="jndiProviderURL">
          <Contents>t3://localhost:8888</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="jndiSecurityPrincipal">
          <Contents>weblogic</Contents>
      </StringRefAddr>
      <SecureRefAddr addrType="jndiSecurityCredentials">
          <Contents>weblogic1</Contents>
      </SecureRefAddr>
      ...
    </RefAddresses>
  </Reference>
  ...
</References>
```

If you are deploying the service to production Oracle WebLogic Server, you can use the `connections.xml` file to specify the user name. As shown in Example 11–12, you must not specify the password.

Example 11–12 JNDI Properties for a Production Environment

```
<References xmlns="http://xmlns.oracle.com/adf/jndi">
  <Reference name="{www.globalcompany.com}StoreFrontService"
             className="oracle.jbo.client.svc.Service" xmlns="">
    <Factory className="oracle.jbo.client.svc.ServiceFactory"/>
    <RefAddresses>
      ...
      <StringRefAddr addrType="jndiFactoryInitial">
         <Contents>weblogic.jndi.WLInitialContextFactory</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="jndiProviderURL">
          <Contents>t3://localhost:8888</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="jndiSecurityPrincipal">
          <Contents>a_username</Contents>
      </StringRefAddr>
      <SecureRefAddr addrType="jndiSecurityCredentials/">
      ...
    </RefAddresses>
  </Reference>
```

```
  ...
</References>
```

3. Save the file.

11.2.15.2 Enabling Authorization for RMI Clients

You can enable permission checking to allow only users with sufficient privileges to invoke a service method on the service interface. In order to enable permission checking, the ADF service interface framework provides an EJB interceptor named `ServicePermissionCheckInterceptor`. This EJB interceptor ensures permission checking is enforced at runtime. Currently, the interceptor is configured to use the Oracle Web Services Manager (Oracle WSM) authorization policy `binding_permission_authorization_policy`.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Section 11.2, "Publishing Service-Enabled Application Modules."

You may also find it helpful to have an understanding of the `binding_permission_authorization_policy` authorization policy supported by Oracle WSM. For more information, see the *Oracle Fusion Middleware Security and Administrator's Guide for Web Services*.

Complete these tasks:

1. Configure the authentication policy for the service in the `connections.xml` file of the client application (the one invoking the service), as described in Section 11.2.15.1, "Enabling Authentication for RMI Clients."
2. Grant users access to the service, as described in Section 11.2.16, "How to Grant Test Users Access to the Service."

To configure a permission-based authorization policy:

1. In the Application Navigator, expand the **META-INF** node of the web service project and double-click the **ejb-jar.xml** node.
2. In the source editor, add the following `JpsInterceptor` definition required by the EJB for application roles evaluation.

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<ejb-jar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/j2ee/ejb-jar_3_0.xsd" version="3.0"
         xmlns="http://java.sun.com/xml/ns/javaee">
  <enterprise-beans>
    ...
  </enterprise-beans>
  <interceptors>
     <interceptor>
        <interceptor-class>
            oracle.security.jps.ee.ejb.JpsInterceptor
        </interceptor-class>
           <env-entry>
              <env-entry-name>application.name</env-entry-name>
              <env-entry-type>java.lang.String</env-entry-type>
              <env-entry-value>ApplicationName</env-entry-value>
              <injection-target>
```

```
                    <injection-target-class>
                        oracle.security.jps.ee.ejb.JpsInterceptor
                    </injection-target-class>
                    <injection-target-name>
                        application_name
                    </injection-target-name>
                </injection-target>
            </env-entry>
        </interceptor>
        ...
    <interceptors>
    <assembly-descriptor>
        <interceptor-binding>
            <ejb-name>*</ejb-name>
            <interceptor-class>
                 oracle.security.jps.ee.ejb.JpsInterceptor
            </interceptor-class>
        </interceptor-binding>
    </assembly-descriptor>
</ejb-jar>
```

3. In the Application Navigator, expand the application module, expand the **serviceinterface** node, and then double-click the remote server class (***AppModule*ServiceImpl.java**) node.
4. In the source editor, place your cursor at the end of your annotations section and add the annotation named `ServicePermissionCheckInterceptor` to enable permission checking at runtime.

```
...
@Stateless(name="oracle.fodemo.storefront.store.service.common.
                                          StoreFrontServiceBean")
@Remote(StoreFrontService.class)
@PortableWebService(targetNamespace="http://www.globalcompany.com/
                      StoreFrontService", serviceName="StoreFrontService",
                      portName="StoreFrontServiceSoapHttpPort",
                      endpointInterface="oracle.fodemo.storefront.
                            service.common.serviceinteface.StoreFrontService")
@CallByReference
@Interceptors({ServiceContextInterceptor.class,
                       ServicePermissionCheckInterceptor.class})
```

5. Save the files.

11.2.16 How to Grant Test Users Access to the Service

After you have have configured the authorization policy for the service, you must configure the Oracle Platform Security Services (OPSS) security provider to specify which users can invoke method on the service. At design time, you perform this task by editing the `jazn-data.xml` configuration file to create application roles and make an invoke permission grant to the desired application roles. Then when you deploy the service, the administrator for the target Oracle WebLogic Server will associate enterprise users with the application roles you specify. This allows you to confer the right to invoke a service method to any user who is a member of that application role. Users who are members of a role that has not been granted the invoke permission, will denied access to the service method.

The invoke permission for Oracle Web Services is defined by the `oracle.wsm.security.WSFunctionPermission` class. You can grant the invoke

permission to the application roles you define for all the methods of the service or just to individual methods.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Section 11.2, "Publishing Service-Enabled Application Modules."

You may also find it helpful to have an understanding of application roles and the OPSS security provides. For more information, see the *Oracle Fusion Middleware Security Guide*.

Complete these tasks:

1. Configure authentication and authorization policies for your service, as described in Section 11.2.14, "How to Secure the ADF Web Service for Access By SOAP Clients," or Section 11.2.15, "How to Secure the ADF Web Service for Access By RMI Clients."
2. Add the OPSS security provider configuration file to your project by creating a `jazn-data.xml` deployment descriptor. For more information about deployment descriptors, see the "Deploying Applications" chapter in the *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.

   Note that in JDeveloper you open the New Gallery, expand **General**, select **Deployment Descriptors** and then **Oracle Deployment Descriptors**, and click **OK**.
3. Create the desired application roles that you want to make grants to, as described in Section 35.4, "Creating Application Roles."
4. For the purpose of testing your web service in JDeveloper using Integrated WebLogic Server, you can populate the application roles with test users, as described in Section 35.6, "Creating Test Users."

To grant the web service permission to application roles in the jazn-data.xml file:

1. From the Application menu, choose **Secure > Resource Grants** or choose **Secure > Entitlement Grants**.

   You can grant multiple privileges to an application role as an entitlement set or you can grant individual privileges to a resource. Create an entitlement grant to aggregate privileges that together secure a specific end user duty. For details about entitlement grants, see Section 35.5.11, "How to Aggregate Resource Grants as Entitlement Grants."
2. In the editor window for the `jazn-data.xml` file, select the **Source** tab.
3. In the source for the `jazn-data.xml` file, expand the `<policy-store>` element to view all ADF security policies that you already defined for your application.

   Currently, this release does not provide an editor to create an application security policy; you will need to manually create the policy in the source for the `jazn-data.xml` file.
4. Inside the `<jazn-policy>` element, create a `<grant>` element that defines the `<grantee>` with the desired application role and the `<permission>` with the fully qualified class name of the Oracle WSM permission class (`oracle.wsm.security.WSFunctionPermission`), the permission target name that uniquely identifies the service method, and the invoke method action that you want to grant to the application role principal.

Your finished source should look similar to this:

```
<grant>
  <grantee>
    <principals>
      <principal>
         <class>oracle.security.jps.service.policystore.ApplicationRole</class>
         <name>customers</name>
      </principal>
     </principals>
  </grantee>
  <permissions>
     <permission>
       <class>oracle.wsm.security.WSFunctionPermission</class>
      <name>www.globalcompany.example.com/
                                        StoreFrontService#CreateAccount</name>
       <actions>invoke</actions>
     </permission>
   </permissions>
</grant>
```

The `<principal>` element is defined by the application role class name `oracle.security.jps.service.policystore.ApplicationRole` and an application role name that you already created. For example, if you created an application role `customers` and you want to grant invoke service method permission to the members of that role, then enter `customers`.

The `<permission>` element is defined by the Oracle WSM class name `oracle.wsm.security.WSFunctionPermission` and the permission target name. The permission target name is formed by appending `/`*`serviceInterfaceName`* and `#`*`serviceMethodName`* (or wildcard character) to the service target namespace.

> **Tip:** You can find the target namespace and service name from the WSDL definition file for the service. In the Application Navigator, double-click the WSDL file under the **serviceinterface** node to view the `name` and `targetNamespace` definitions.

For example, in the Fusion Order Demo, the WSDL definition file defines the following name and namespace:

```
<wsdl:definitions
     name="StoreFrontService"
     targetNamespace="www.globalcompany.example.com"
```

Assume that you want to grant a permission to allow authorized users to invoke a `CreateAccount` service method on the service interface with these Fusion Order Demo name and namespace, you would enter the target name like this:

```
www.globalcompany.example.com/StoreFrontService#CreateAccount
```

Alternatively, you can enter the target name using the wildcard character * to grant all operations of the service interface in a single permission:

```
www.globalcompany.example.com/StoreFrontService#*
```

The actions that you can enter are defined by the permission class. In this case, `oracle.wsm.security.WSFunctionPermission` defines the single action `invoke`.

5. Save the changes to the `jazn-data.xml` file.

11.2.17 How to Enable Support for Binary Attachments for SOAP Clients

The ADF service interface framework supports using Message Transmission Optimization Mechanism (MTOM) to handle sending binary data in any service method that operates on a `ViewRow` with a `BlobDomain/ClobDomain` attribute. This permits binary data to accompany XML messages, for example when images are required to document an insurance claim. The SDO data objects of the service-enabled application module maps `BlobDomain/ClobDomain` to `javax.activation.DataHandler`. These `DataHandler` properties could be passed as attachments during SDO data object marshalling/unmarshalling when the web service is called using the SOAP protocol.

To enable MTOM support for your SOAP protocol, you must add the `@MTOM` annotation to the service interface implementation class (for example, `StoreFrontServiceImpl.java`) and your method must operate on a `ViewRow` with `BlobDomain/ClobDomain` attribute.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Section 11.2, "Publishing Service-Enabled Application Modules."

To enable support for sending binary data attachments:

1. In the Application Navigator, expand the application module, expand the **serviceinterface** node, and then double-click the remote server class (***AppModule*ServiceImpl.java**) file.

   In the web service generated from the `StoreServiceAM` application module in the `StoreFrontModule` application of the Fusion Order Demo, the remote server class is `StoreFrontServiceImpl.java`.

2. In the source for the remote server class, place your cursor anywhere in the annotation section.

   For example, in the `StoreFrontServiceImpl.java` the annotation section for the service is:

   ```
   ...
   @Stateless(name="oracle.fodemo.storefront....common.StoreFrontServiceBean",
       mappedName="StoreFrontServiceBean")
   @Remote(StoreFrontService.class)
   @PortableWebService(targetNamespace="www.globalcompany.example.com",
       serviceName="StoreFrontService", portName="StoreFrontServiceSoapHttpPort",
       endpointInterface=
          "oracle.fodemo.storefront...common.serviceinterface.StoreFrontService")
   ...
   ```

3. Open the Property Inspector and expand the **Web Services** section, then select **Enable MTOM**.

   JDeveloper adds the `@MTOM` annotation to the annotations section of the file.

   ```
   ...
   @Stateless(name="oracle.fodemo.storefront....common.StoreFrontServiceBean",
       mappedName="StoreFrontServiceBean")
   @Remote(StoreFrontService.class)
   ```

```
@PortableWebService(targetNamespace="www.globalcompany.example.com",
    serviceName="StoreFrontService", portName="StoreFrontServiceSoapHttpPort",
    endpointInterface=
       "oracle.fodemo.storefront...common.serviceinterface.StoreFrontService")
@MTOM
...
```

11.2.18 How to Test the Web Service Using Integrated WebLogic Server

You can run the web service in JDeveloper using Integrated WebLogic Server. You can also deploy the web service to Oracle WebLogic Server to test the service.

To run and test using Integrated WebLogic Server:

1. In the Application Navigator, expand the application module, expand the **serviceinterface** node, and then select the remote server class (***AppModule*ServiceImpl.java**) file.

   In the web service generated from the `StoreServiceAM` application module in the `StoreFrontModule` application of the Fusion Order Demo, the remote server class is `StoreFrontServiceImpl.java`.

2. Right-click the remote server class file, and choose **Run** or **Debug**.

   The Configure Default Domain dialog displays the first time you run the application and start a new domain in Integrated WebLogic Server. Use the dialog to define an administrator password for the new domain. Passwords you enter can be eight characters or more and must have a numeric character.

   JDeveloper initializes the server instance, and then deploys the application and starts the web service. During this time, the output from these processes is displayed in the **Running** tab of the Log window. After the web service has started, the target URL is also displayed in the Log window.

   > **Tip:** In the Log window, you can click the target URL link to launch the HTTP Analyzer. This is a convenient shortcut for testing with JDeveloper Integrated WebLogic Server. For more information about the HTTP Analyzer, see the "Auditing and Profiling Applications" chapter in the *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.

3. Copy the target URL (beginning with `http://`) from the Log window.

   For example, if the Log window displays:

   `http://<`*`ipaddress`*`>/ADFServiceDemo-ADFModel-context-root`

   Integrated WebLogic Server will display the service endpoint URL. So there is no need to append the service name.

4. Launch a web browser, paste the target URL you copied from the Log window into the browser address field, and submit the HTTP request.

5. In the test page, choose the operation you want to invoke from the **Operations** dropdown list and enter sample data in its parameter fields.

6. When you are ready, press **Invoke** to submit the operation and view the results for the operation in the Test Results page.

   The Test Results page displays the XML Soap format of the information returned by the operation.

11.2.19 How to Prevent Custom Service Methods from Timing Out

When you test the web service you may find that some of your custom methods exceed the established timeout limitation established by the Java Transaction API (JTA). The JTA timeout setting establishes an execution boundary for service methods that by default may not exceed 30 seconds. You can use the Administration Console for Oracle WebLogic Server to increase the JTA timeout setting. If you still receive a timeout exception or you anticipate that the custom methods of the service interface may be long running, you can specify an EJB transaction attribute for the stateless session bean to prevent the EJB from executing those methods in a JTA transaction.

To make a custom method exempt from timing out, you set `TransactionAttributeType.NOT_SUPPORTED` in the Property Inspector specifically for that method. JDeveloper updates the method by adding the annotation `@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)`.

Example 11–13 NOT_SUPPORTED Annotation in Custom Service Method

```
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public void updateCustomerInterests(List<String> pCategoryIds) throws
                                                          ServiceException {
        invokeCustom((Method) _map.get("updateCustomerInterests"),
             new Object[] { pCategoryIds }, new String[] { null }, false);
}
```

Because a method with this transaction attribute setting will not be executed in the JTA transaction, it is your responsibility to enforce control over the transaction using the ADF Business Components methods of the `oracle.jbo.ApplicationModule` and `oracle.jbo.Transaction` interfaces. For instance, as Example 11–14 shows, the methods of the implementation class of the application module that you service-enabled, will need to call `am.getDBTransaction().commit()` or `rollback()` in order to complete the transaction.

Example 11–14 Custom Code to Handle Transaction Execution in Implementation Method

```
public void updateCustomerInterests(List pCategoryIds) {
     try
     {
        if (pCategoryIds != null && pCategoryIds.size() > 0) {
            List<Integer> copyOfCategoryIds = (List<Integer>)
                                        this.cloneList(pCategoryIds);
            ViewObject selectedCategories =
                                        this.getSelectedCategoriesShuttleList();
            RowSetIterator rsi = selectedCategories.createRowSetIterator(null);
            // remove rows for the current user not in the list of product keys
            while (rsi.hasNext()) {
                Row r = rsi.next();
                Number interestId = (Number)r.getAttribute("CategoryId");
                // existing row is in the list, we're ok, so remove from list.
                if (copyOfCategoryIds.contains(interestId)) {
                    copyOfCategoryIds.remove(interestId);
                }
                // if the existing row is in not list, remove it.
                else {
                    r.remove();
                }
            }
            rsi.closeRowSetIterator();
            // at this point, add new rows for the keys that are left
```

```
                    for (int i =0 ;i < copyOfCategoryIds.size(); i++ )  {
                        Row newRow = selectedCategories.createRow();
                        selectedCategories.insertRow(newRow);
                        newRow.setAttribute("CategoryId", (String)
                                        copyOfCategoryIds.get(i).toString());
                    }
            this.getTransaction().commit();
            }
        }
        catch (JboException e)
        {
            this.getTransaction().rollback();
            throw e;
        }
}
```

You should not change the default transaction attribute setting for the standard service methods generated for the service interface (see Table 11–1). The standard methods will execute within the default execution boundary set for the JTA transaction.

Before you begin:

It may be helpful to have an understanding of how the SDO framework supports service-enabled ADF view objects and enables web service clients to access rows of data and perform service operations. For more information, see Section 11.2, "Publishing Service-Enabled Application Modules."

To prevent custom methods from executing in a JTA transaction:

1. In the Application Navigator, expand the application module, expand the **serviceinterface** node, then double-click the remote server class (***AppModule*ServiceImpl.java**) file.

   In the web service generated from the `StoreServiceAM` application module in the `StoreFrontModule` application of the Fusion Order Demo, the remote server class is `StoreFrontServiceImpl.java`.

2. In the source editor, locate the custom method that you want to prevent from timing out and place your cursor on the method.

3. In the Property Inspector, expand **Stateless Session Bean** and choose **TransactionAttributeType.NOT_SUPPORTED** from the **TransactionAttribute** dropdown list.

   As shown in `updateCustomerInterests()` in Example 11–13, JDeveloper updates the custom service method with the annotation.

4. Save the remote server class.

5. In the Application Navigator, double-click the application module implementation class (*`AppModule`*`Impl.java`) file.

   The implementation class defines the custom methods that the you exposed through the service interface. In the Fusion Order Demo, the application module implementation class is `StoreServiceAMImpl.java`.

6. In the source editor, in the custom method that implements the service method that you previously set the **TransactionAttribute** property on, add the custom code that will commit and rollback the transaction.

   As Example 11–14 shows, if you configured the **TransactionAttribute** property on the service method named `updateCustomerInterests()`, then you would

open the implementation class for the application module, locate the custom method `updateCustomerInterests()`, and add `am.getDBTransaction().commit()` and `rollback()` as part of the method's `try` and `catch` statements.

7. Save the application module implementation class.

11.2.20 How to Deploy Web Services to Oracle WebLogic Server

You can deploy the web service to Oracle WebLogic Server, for example to perform a second stage of testing the service.

Before you begin:

1. If your project contains a standard web service (a Java class or interface with the `@WebService` annotation), you must remove the `@WebService` annotation from the Java class before you can deploy the Business Components web service. If you attempt to deploy the Business Components web service from the same project as the standard web service, deployment will fail with an an Oracle WebLogic Server exception error. It is therefore necessary to create standard web services in a separate project from your ADF Business Components service.
2. If you created an asynchronous web service, before you can deploy the service you must configure the queues used to store the request and response. For information about configuring the request and response queues, see the *Oracle Fusion Middleware Concepts Guide for Oracle Infrastructure Web Services*.
3. If you configured authorization for the web service as described in Section 11.2.14, "How to Secure the ADF Web Service for Access By SOAP Clients," edit the `weblogic-application.xml` file to define application ID parameters. This file appears in the Application Navigator in the Application Resources panel, under the **Descriptors** and **META-INF** nodes.

   Add the following `<application-param>` definition as the first element:

```
<weblogic-application
xmlns="http://www.bea.com/ns/weblogic/weblogic-application"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation=

"http://www.bea.com/ns/weblogic/weblogic-application.xsd">
   <application-param>
      <param-name>jps.policystore.applicationid</param-name>
      <param-value>ApplicationName</param-value>
   </application-param>
   ...
</weblogic-application>
```

   Note that the *`ApplicationName`* that you enter must match the name identified in the `jazn-data.xml` policy store definition:

```
<jazn-data>
   <policy-store>
        <applications>
            <application>
                <name>ApplicationName</name>
                <app-roles>
                    ...
                </app-roles>
                <jazn-policy>
                    ...
```

```
                </jazn-policy>
            </application>
        </applications>
    </policy-store>
</jazn-data>
```

4. Edit the `ejb-jar.xml` file to add the following `JpsInterceptor` definition required by the EJB for application roles evaluation. This file appears in the Application Navigator under the **META-INF** node of the web service project.

```
<?xml version = '1.0' encoding = 'windows-1252'?>
<ejb-jar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
         http://java.sun.com/xml/ns/j2ee/ejb-jar_3_0.xsd" version="3.0"
         xmlns="http://java.sun.com/xml/ns/javaee">
  <enterprise-beans>
    ...
  </enterprise-beans>
  <interceptors>
     <interceptor>
        <interceptor-class>
            oracle.security.jps.ee.ejb.JpsInterceptor
        </interceptor-class>
           <env-entry>
              <env-entry-name>application.name</env-entry-name>
              <env-entry-type>java.lang.String</env-entry-type>
              <env-entry-value>ApplicationName</env-entry-value>
              <injection-target>
                  <injection-target-class>
                      oracle.security.jps.ee.ejb.JpsInterceptor
                  </injection-target-class>
                  <injection-target-name>
                      application_name
                  </injection-target-name>
              </injection-target>
           </env-entry>
     </interceptor>
     ...
  <interceptors>
  <assembly-descriptor>
     <interceptor-binding>
        <ejb-name>*</ejb-name>
        <interceptor-class>
             oracle.security.jps.ee.ejb.JpsInterceptor
        </interceptor-class>
     </interceptor-binding>
  </assembly-descriptor>
</ejb-jar>
```

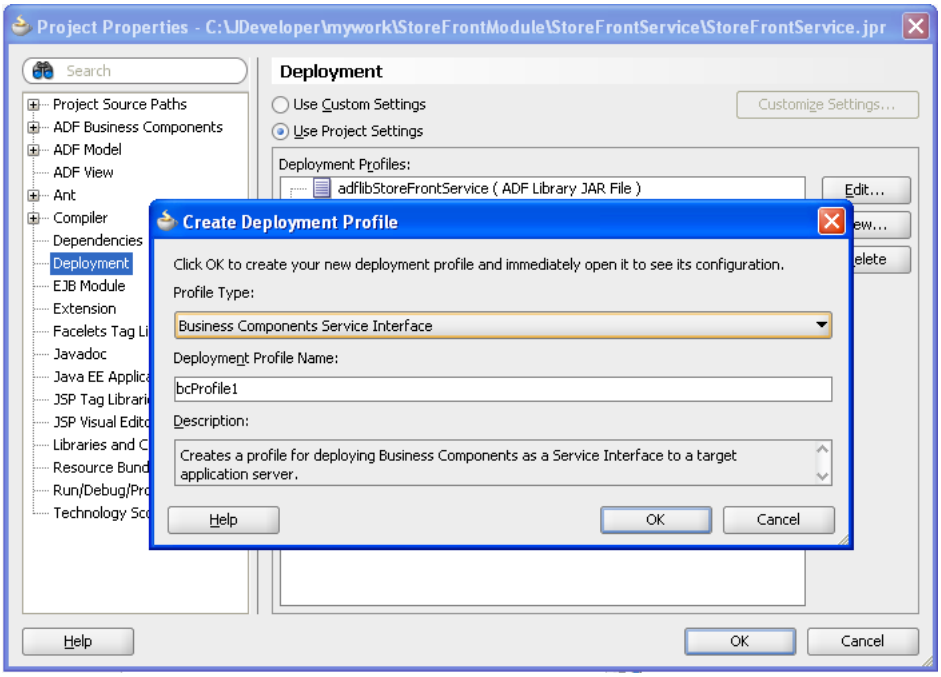
   Note that *`ApplicationName`* must also match the application name identified in the `jazn-data.xml` policy store definition.

To deploy to Oracle WebLogic Server:

1. Create an application server connection to Oracle WebLogic Server:
   a. From the main menu, choose **View** > **Application Server Navigator**.
   b. In the Application Server Navigator, right-click the **Application Servers** node and choose **New Application Server**, and complete the Create Application Server Connection wizard.

2. Create a service deployment profile:
   a. In the Application Navigator, right-click the project contain the web service and choose **Project Properties**.
   b. In the Project Properties dialog, open the Deployment page and click **New**.
   c. In the Create Deployment Profile dialog, choose **Business Components Service Interface** as the archive type, as shown in Figure 11–11.

Figure 11–11 Dialog for Creating a Business Components Service Deployment Profile



3. From the **Application** menu, choose **Deploy** > *deployment profile* for the deployment profile you created.
4. In the Deploy wizard, on the Deployment Action page, select **Deploy to Application Server** and click **Next**.
5. On the **Select Server** page, select the application server connection.
6. Click **OK**.

11.3 Accessing Remote Data Over the Service-Enabled Application Module

ADF Business Components application modules offer built-in support for web services and for publishing rows of view object data as service data objects (SDOs). Entity objects that you create in your local data model project can utilize the SDO services that the service-enabled application module exposes on its service interface. By creating service-backed entity objects in your local project, you avoid having to work directly with the service proxy and SDOs programmatically for all common remote service data access tasks.

The Create Entity Object wizard makes it easy for you to choose between a local database and the remote ADF Business Components service when you create the entity object, as described in Section 11.3.1, "How to Use Service-Enabled Entity

Objects and View Objects." In this way, service-enabled application modules provide an alternative way to access data that is not available locally in the database.

Once you create the service-backed entity object, you will be able to create view objects, view links, and view criteria to filter the data at runtime. You will also be able to utilize these view objects in your data model as though you were working with locally available data accessed from database tables.

The following sections describe how to augment your data model project using a service-enabled ADF application module.

11.3.1 How to Use Service-Enabled Entity Objects and View Objects

You will want to use the service-backed components as part of your application design strategy when one of the following conditions is true:

- The client data model project needs to work with data from a service-enabled application that is part of a separate business process.
- The client data model project needs to work with data from a pluggable, external service.

In the first case, you provide both sides of the service. In the second case, you may not know what the external service looks like and you may need to perform the following:

1. Even though you might not need a Fusion implementation of the service, since the service-enabled application module's service interface is the supported unit of pluggability and the supported way of creating service-backed entity objects and view objects, you create an application module with a service interface that describes the shape you want your "canonical" pluggable service to have.
2. After generating the service-enabled application module as described in Section 11.2.1, "How to Enable the Application Module Service Interface," you then build service-back entity objects and view objects.
3. Finally, you can create an EJB session bean (or an SCA composite) that supports the same service interface as the "canonical" service-enabled application module that you created in Step 2 and configure the `connections.xml` file of the client project containing the service-enabled business components based on this service interface to use this "plugged" version instead.

For details about how to expose an application module as a web service, see Section 11.2, "Publishing Service-Enabled Application Modules."

11.3.1.1 Creating Entity Objects Backed by SDO Services

You create the service-backed entity object using the Create Entity Object wizard by specifying the URL for the WSDL document that describes the deployed service already running on an application server. The wizard uses the WSDL service description to display the list of available service view instances. In the wizard, you select among the displayed view instances to specify the entity object's data source. At the time you run the wizard, the service endpoint must be accessible in order to locate the WSDL document.

To create the entity object that uses a service view instance as its data source:

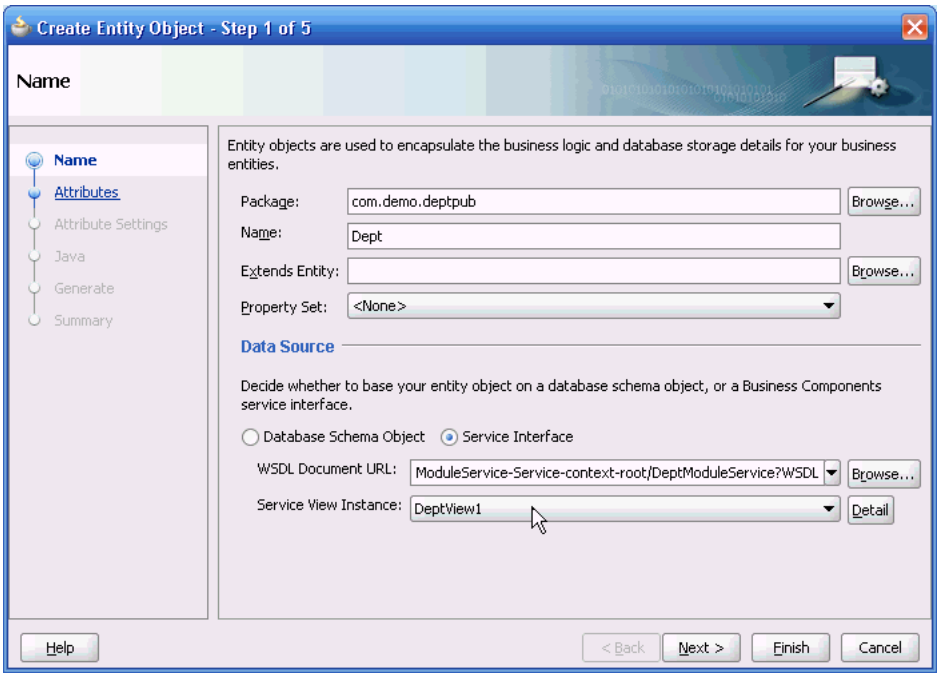
1. In the Application Navigator, right-click the project in which you want to create the entity object and choose **New**.
2. In the New Gallery, expand the **Business Tier** node, select **ADF Business Components** and then **Entity Object**, and click **OK**.

3. On the Name page of the Create Entity Object wizard, do the following to create the entity object:

   - Enter the package name in which the entity object will be created and enter the entity object name.
   - Select **Service Interface** as the data source for which you want to create the entity object.
   - Enter the WSDL document URL for the published web service or click **Browse** to use the Find Web Services wizard to locate the remote service from the UDDI registry.
   - Select **Service View Instance** from the dropdown list, as shown in Figure 11–12.

     The wizard will attempt to connect with the service endpoint and populate the list from the WSDL service description. If the endpoint is unavailable, the list will remain empty.

Figure 11–12 Service Interface as Data Source in the Create Entity Object Wizard



4. Click **Next** and modify the attributes of the entity object.

   For example, on the Attributes page, you can remove an attribute when you do not want the service-backed entity object to reference it.

5. Click **Next** and modify the attributes settings of the entity object before you complete the wizard.

   For example, on the Attributes Settings page, you can enable the **Refresh After Insert** and **Refresh After Update** options for attributes that you anticipate will be modified whenever the entity is modified. Typical candidates include a version number column or an updated date column in the row.

6. Click **Finish**.

11.3.1.2 Creating View Objects Backed by SDO Services

After you add the service-backed entity object to your project, you can create service-backed view objects to query and optionally filter the data from the remote service for use in the user interface. A service-backed view object is a view object whose single entity usage references an entity object that is backed an SDO service. You cannot make existing view objects service-backed. Instead, when you create the view object, the new view object will automatically be service-backed if its entity usage is a service-backed entity object.

Before you begin:

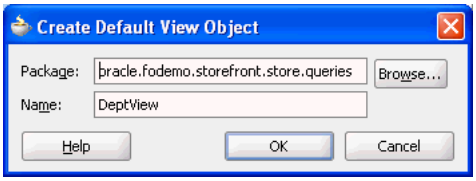
Create the service-backed entity object as described in Section 11.3.1.1, "Creating Entity Objects Backed by SDO Services."

To create a view object from the service-backed entity object:

1. In the Application Navigator, right-click the service-backed entity object and choose **New Default View Object**.
2. In the Create Default View Object dialog, enter the package name in which the view object will be created and enter the view object name, as shown in Figure 11–13.

   The generated view object will contain the same attributes as the entity object. You can optionally edit the view object in the overview editor to customize the query. You can also define view criteria for the view object when you want to filter the data from the remote service. For details about filtering query results, see Section 5.11, "Working with Named View Criteria."

Figure 11–13 View Object Can Be Created from Service-Backed Entity Object



11.3.2 What Happens When You Create Service-Backed Business Components

The service-backed entity object is an entity object that encapsulates the details of accessing and, if necessary, modifying a row of data from a remote ADF Business Components service. After you use the Create Entity Object wizard to create the service-backed entity object, JDeveloper saves additional service-related metadata in the `<Datasource>` element of the entity component definition, as shown in Example 11–15.

The service-backed view object references the single, service-backed entity object in its metadata just as any entity-based view object does. You can use the service-backed view object just as you would use any other view object. For details about working with view objects, see Chapter 5, "Defining SQL Queries Using View Objects." The ADF runtime handles the interaction with the remote ADF Business Components service.

Example 11–15 Entity Object Metadata Shows Service View as Data Source

```
<Entity
   xmlns="http://xmlns.example.com/bc4j"
   Name="CustomerSEO"
```

```
   DBObjectType="table"
   AliasName="CustomerSEO"
   BindingStyle="OracleName"
   UseGlueCode="false">
   <DataSource
      DataSourceClass="oracle.jbo.datasource.svc.SIEODataSourceImpl"
      Type="ServiceInterface">
      <ServiceInterface
         ServiceName="{http://www.globalcompany.com/oesvc/}OrderEntryService"
         SDOName="{http://www.globalcompany.com/oesvc/}CustomersSVO"
         SVIName="{http://www.globalcompany.com/oesvc/}CustomersSVO"
         CreateOpName="createCustomer"
         UpdateOpName="updateCustomer"
         DeleteOpName="deleteCustomer"
         GetOpName="getCustomer"
         FindOpName="findCustomers"
         ProcessOpName="processCustomers"/>
   </DataSource>
   <Attribute
      Name="CustomerId"
      IsNotNull="true"
      Precision="6"
      Scale="0"
      ColumnName="CustomerId"
      Type="oracle.jbo.domain.Number"
      ColumnType="NUMBER"
      SQLType="NUMERIC"
      PrimaryKey="true">
   </Attribute>
   <!-- ... Other Attribute elements here ... -->
</Entity>
```

11.3.3 How to Update the Data Model for Service-Backed Business Components

Because the service interface exposes individual view instances, you are responsible for defining hierarchical relationships between service-backed entity objects (through associations) and service-backed view objects (through view links) in your consuming project. View links and associations are not automatically created when you create the service-backed business component. For example, if the application module of the published ADF Business Components service defines a master-detail relationship that you want to utilize, then you must define a view link for the corresponding view objects in your own project to preserve this hierarchy.

Furthermore, while you can create view links between view objects that query their data locally and service-backed view objects (and the other way around), once you define the view link, you will not be able to create entity-based view objects with the following entity object usages:

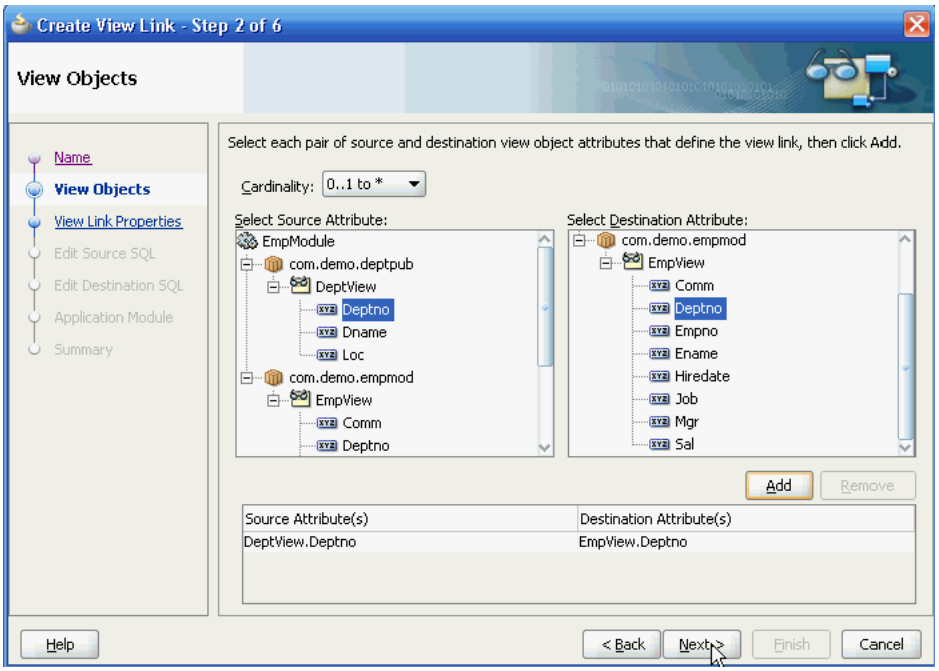
- The view object will not be able to reference a secondary entity usage that is a service-backed entity object.
- The view object will not be able to reference a primary entity usage that is a service-backed entity object with secondary entity usages.

The same restrictions apply to associations in the client project between regular entity objects and service-backed entity objects: while you can create the associations, you will not be able to create view objects.

You use the Create View Link wizard to specify relationships between the view objects that your project defines, as shown in Figure 11–14. For details about creating view

links, see Section 5.6.2, "How to Create a Master-Detail Hierarchy for Entity-Based View Objects."

Figure 11–14 One to Many Relationship Defined in Create View Link Wizard



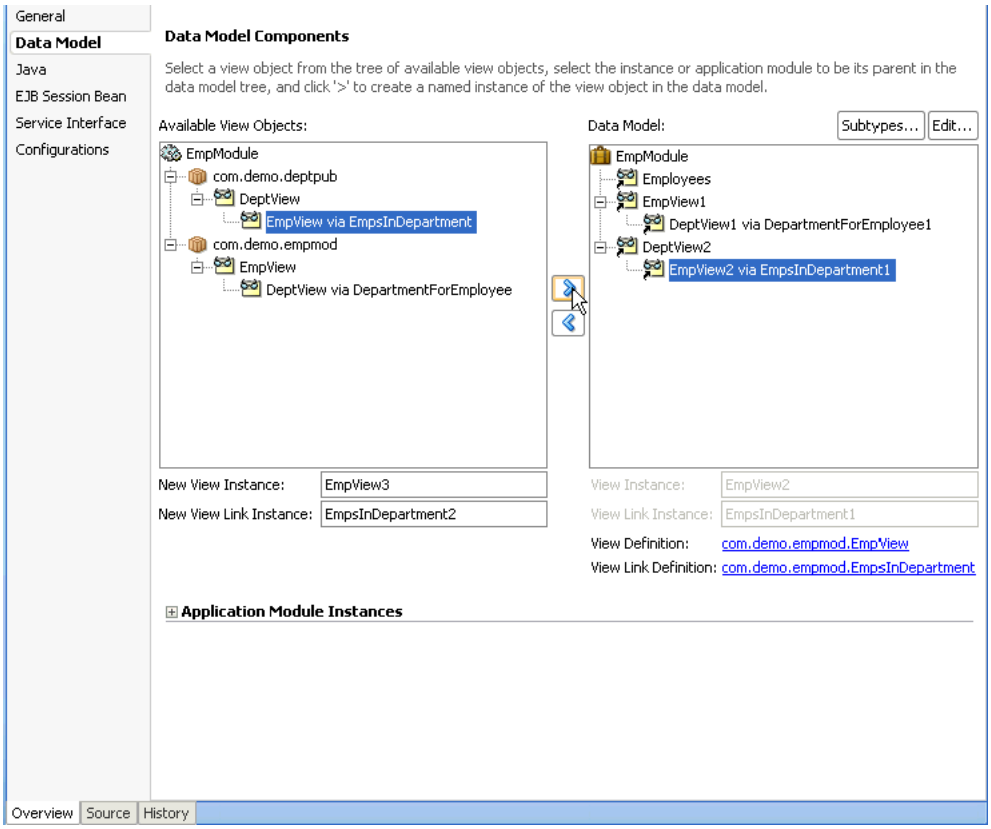
View links you create may define relationships between service-backed view objects and view objects that query locally accessed database tables. For example, you might choose to drive a database-derived detail view object with a service-backed master view object. You can create view links with the combinations shown in Table 11–2.

Table 11–2 Supported View Link Combinations Involving Service-Backed View Objects

Use Case	Master View Object Type	View Linked Detail View Object Type	View Link Cardinality
Local master rows with remote details	Query-based	Service-backed	One-to-many
Remote master rows with local details	Service-backed	Query-based	One-to-many
Local master rows with remote reference information	Query-based	Service-backed	Many-to-one
Remote master rows with local reference information	Service-backed	Query-based	Many-to-one

Once you have defined the desired view hierarchy, using the Create View Link wizard, you use the overview editor for your project's application module to define new view instances on the data model, as shown in Figure 11–15. The updated data model allows you to expose the view objects as ADF data controls that enable databinding with the user interface components of the Fusion web application. For details about updating the data model, see Section 9.2.3.2, "Adding Master-Detail View Object Instances to an Application Module."

Figure 11–15 Data Model Contains Service View Instances



11.3.4 How to Configure the Service-Backed Business Components Runtime

Before you can run your application and interact with the published service-enabled ADF application module to invoke service operations, you need to describe the published service, including the service's endpoint provider type and other configuration information. The ADF Business Components `ServiceFactory` class (`oracle.jbo.client.svc.ServiceFactory`) returns a proxy for the service, then uses the service proxy to invoke the service operations. The service factory can return proxies for three different service endpoint providers, to support these transport protocols:

- When the service endpoint provider is ADF Business Component, the transport protocol is EJB RMI.
- When the service endpoint provider is SOA Fabric, the transport protocol is SOA Fabric SDO binding.
- When the service endpoint provider is SOAP (for JAX-WS clients), the transport protocol is SOAP.

To configure the consuming application to invoke published service operations:

1. Add the *`bcProfileName`*`_Common.jar` file for the SDO's generated classes to the client project's classpath.
2. Update the `connections.xml` file in the client project's `.adf/META-INF` folder to describe the published ADF Business Components service.

The updates you make to the file will depend on the transport protocol your application uses: EJB RMI protocol, SOA Fabric SDO binding, or SOAP protocol (for JAX-WS clients).

11.3.4.1 Adding the SDO Client Library to the Classpath

Before your application can access the published service, the service consuming project must have access to the generated SDO classes and their schema definitions. These files are packaged in the *bcProfileName*_Common.jar file generated by the development team responsible for publishing the service.

To make the SDO classes and their schema definitions available to your application, obtain the *bcProfileName*_Common.jar file from the service-provider team and place this JAR file in a folder of your local project. For example, you may copy the JAR file into your project's deploy folder. You can then use JDeveloper to add the JAR file to your project's classpath with a SDO client library you create. For steps to generate the SDO classes JAR file, see Section 11.2.20, "How to Deploy Web Services to Oracle WebLogic Server."

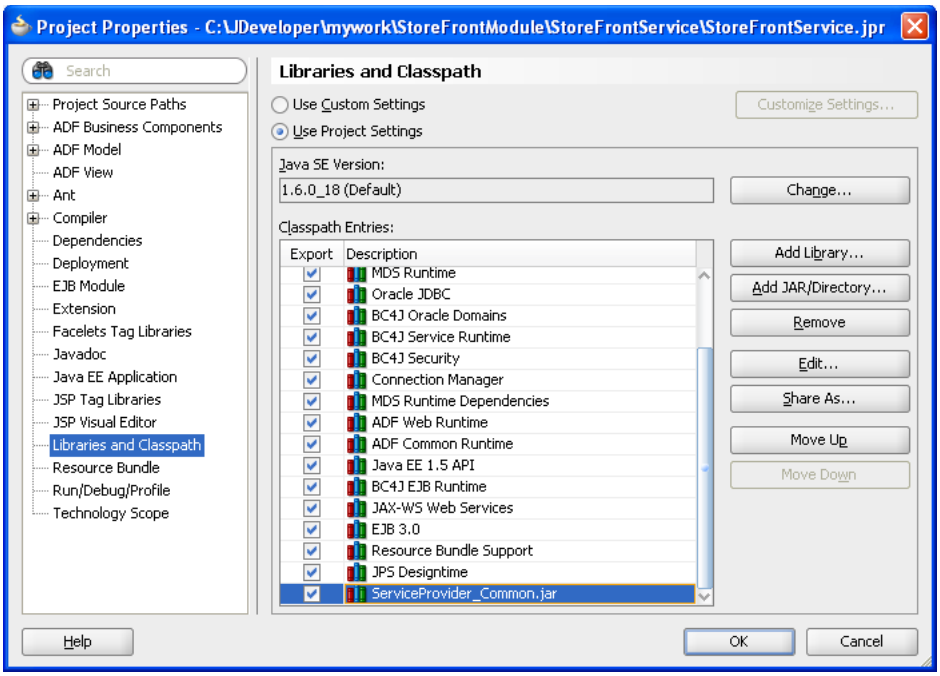
To add the SDO client library to the classpath:

1. In the Application Navigator, right-click your project node and choose **Project Properties**.
2. In the Project Properties dialog, select **Libraries and Classpath** and click **Add Libraries**.
3. In the Add Library dialog, click **New** to create the SDO client library.
4. In the Create Library dialog, click **Add Entry** to add a classpath entry.
5. In the Select Path Entry dialog, browse to the folder that contains the *bcProfileName*_Common.jar file and select the file to view it in the Create Library dialog.

   The Select Path Entry dialog lets you browse the file system or local area network to locate the JAR file. If you cannot browse the deploy folder of the service-provider's application workspace to obtain the JAR file, you must obtain the file and copy it into your own project's folder. For example, you may have copied the JAR file into your project's deploy folder.
6. Click **OK** in the dialogs to display the Project Properties dialog with the SDO client library selected. Click **OK** to add the library to the classpath.

   Figure 11–16 shows the SDO client library with the name ServiceProvider_Common.jar selected. In this case, the library name is the same as the JAR file name. Optionally, you can edit the library name in the Create Library dialog.

Figure 11–16 SDO Client Library Classpath Entry



11.3.4.2 Registering the ADF Business Components Service in the Consuming Application's connections.xml for the EJB RMI Protocol

When the service endpoint provider is ADF Business Components, the service factory will return an EJB object proxy bound to a stateless session bean running in the EJB container. You must provide the JNDI context information to allow the consuming application to look up the published service.

Lookup information that you provide to register the published ADF Business Components service appears in the consuming Fusion web application's `connections.xml` file, located in the `.adf/META-INF` folder relative to the application. The ADF connection architecture uses this file to encapsulate the details of the service endpoint provider.

The JNDI lookup information you provide will depend on whether the published service runs locally (in the same JVM) with the consuming application or runs remotely on a separate server from the consuming application. Typically, the ADF Business Components service is in a different application from the consuming application and is therefore run remotely.

To register the published service with your client application, update the `connections.xml` file following the example in Example 11–16. When the ADF Business Components service runs local to the consuming application (as occurs when you run within JDeveloper), the service factory needs only the JNDI name to look up the service.

Note: When you deploy the calling application to standalone Oracle WebLogic Server, you will use Oracle Enterprise Manager to configure the JNDI context properties instead of editing the `connections.xml` file. For instructions, refer to the online documentation in Oracle Enterprise Manager.

Example 11–16 Client connections.xml File Registers the Local EJB ADF Business Components Service

```
<References xmlns="http://xmlns.oracle.com/adf/jndi">
  <Reference name="{www.globalcompany.com}StoreFrontService"
             className="oracle.jbo.client.svc.Service" xmlns="">
     <Factory className="oracle.jbo.client.svc.ServiceFactory"/>
     <RefAddresses>
       <StringRefAddr addrType="serviceInterfaceName">
        <Contents>oracle.foddemo.storefront.store.service.common.
                                              serviceinterface.StoreFrontService</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="serviceEndpointProvider">
          <Contents>ADFBC</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="jndiName">
          <Contents>StoreFrontServiceBean#oracle.fodemo.storefront.store.service.common.
                                               serviceinterface.StoreFrontService</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="serviceSchemaName">
          <Contents>StoreFrontAMService.xsd</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="serviceSchemaLocation">
          <Contents>oracle/fodemo/storefront/store/service/common/serviceinterface/</Contents>
       </StringRefAddr>
     </RefAddresses>
  </Reference>
  ...
</References>
```

When the ADF Business Components service runs remotely to the calling client, then remote JNDI context information needs to be added to the `connections.xml` file. You can edit these JNDI context properties in the `connections.xml` file, as shown in Example 11–17:

- `jndiFactoryInitial` should be set to `weblogic.jndi.WLInitialContextFactory`.
- `jndiProviderURL` is the JNDI provider URL that indicates the location of the JNDI server. The URL should be composed as `t3://`*`<hostname>`*`:`*`<server port>`*.

  For example, specify a URL like: `t3://localhost:8888`, where `t3` is the Oracle WebLogic protocol, `localhost` is the host name that the remote Oracle WebLogic Server instance runs in, `8888` is the port number.
- `jndiSecurityPrincipal` specifies the principal (user name) with permission to access the remote JNDI.

  When you deploy the service to standalone Oracle WebLogic Server, the user name can be read from the file.
- `jndiSecurityCredentials` specifies the credentials (password) to be used for the security principal.

  As Example 11–11 shows, when you deploy the service to standalone Oracle WebLogic Server in a test environment, you can specify credentials in plain text for the JNDI provider. For example, you can specify `weblogic/weblogic1` which are the default administrator user name/password credentials with sufficient privileges to access JNDI provider for Oracle WebLogic Server.

  When you deploy the service to a production environment, you must remove the plain text password to avoid creating a security vulnerability. As Example 11–17 shows, the `connections.xml` file must contain `<SecureRefAddr`

addrType="jndiSecurityCredentials"/> with no password. To configure the service password for standalone Oracle WebLogic Server, you must use Oracle Enterprise Manager, which will store the encrypted password in Oracle's credential store.

Example 11–17 Client connections.xml File Registers the Remote EJB ADF Business Components Service

```
<References xmlns="http://xmlns.oracle.com/adf/jndi">
  <Reference name="{www.globalcompany.com}StoreFrontService"
             className="oracle.jbo.client.svc.Service" xmlns="">
     <Factory className="oracle.jbo.client.svc.ServiceFactory"/>
     <RefAddresses>
       <StringRefAddr addrType="serviceInterfaceName">
        <Contents>oracle.foddemo.storefront.store.service.common.
                                              serviceinterface.StoreFrontService</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="serviceEndpointProvider">
          <Contents>ADFBC</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="jndiName">
          <Contents>StoreFrontServiceBean#oracle.fodemo.storefront.store.service.common.
                                               serviceinterface.StoreFrontService</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="jndiFactoryInitial">
          <Contents>weblogic.jndi.WLInitialContextFactory</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="jndiProviderURL">
          <Contents>t3://localhost:8888</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="jndiSecurityPrincipal">
          <Contents>a_username</Contents>
       </StringRefAddr>
       <SecureRefAddr addrType="jndiSecurityCredentials"/>
       <StringRefAddr addrType="serviceSchemaName">
          <Contents>StoreFrontAMService.xsd</Contents>
       </StringRefAddr>
       <StringRefAddr addrType="serviceSchemaLocation">
          <Contents>oracle/fodemo/storefront/store/service/common/serviceinterface/</Contents>
       </StringRefAddr>
     </RefAddresses>
  </Reference>
  ...
</References>
```

11.3.4.3 Registering the ADF Business Components Service in the Consuming Application's connections.xml for the SOAP Protocol

When the service endpoint provider is SOAP, the service factory will create a dynamic JAX-WS client proxy. You must provide the WSDL URL and port name to allow the consuming application to look up the published service. Additionally, for the SOAP client, Oracle Web Service Manager (Oracle WSM) client security policy can be attached as part of the SOAP header.

Lookup information that you provide to register the published ADF Business Components service appears in the consuming Fusion web application's `connections.xml` file, located in the `.adf/META-INF` folder relative to the application. The ADF connection architecture uses this file to encapsulate the details of the service endpoint provider.

> **Note:** When you deploy the calling application to standalone Oracle WebLogic Server, you will use Oracle Enterprise Manager to configure the JNDI context properties instead of editing the `connections.xml` file. For instructions, refer to the online documentation in Oracle Enterprise Manager.

To register the published service with your client application for the SOAP protocol, depending on whether your application uses identity propagation or identity switching, update the `connections.xml` file following the example in either Example 11–18 or Example 11–19. Identity propagation and switching are similar in that each process involves propagating an identity. In Fusion web applications, identity propagation involves propagating the identity that is currently executing code. Identity switching, on the other hand, involves propagating an application identity that is different from that currently executing code.

To register the published service with your client application so the the user identity will be switched based on the credential key, specify the clientside policy `oracle/wss11_username_token_with_message_protection_client_policy` in the `connections.xml` file following the example in Example 11–18.

> **Note:** The `connections.xml` file supports Oracle WSM security policy client overrides. When the security policy is `oracle/wss11_username_token_with_message_protection_client_policy`, the `csf-key` property can be overridden to specify the consuming application's credentials.

Example 11–18 Client connections.xml File Registers the Remote Business Components Service for the SOAP Protocol Using Identify Switching

```
<Reference name="{http://xmlns.oracle.com/apps/sample/hrService/}HrService"
className="oracle.jbo.client.svc.Service" xmlns="">
   <Factory className="oracle.jbo.client.svc.ServiceFactory"/>
   <RefAddresses>
      <StringRefAddr addrType="serviceInterfaceName">
            <Contents>oracle.apps.sample.hrService.HrService</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="serviceEndpointProvider">
         <Contents>SOAP</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="webServiceConnectionName">
         <Contents>HrServiceConnection</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="serviceSchemaName">
         <Contents>HrService.xsd</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="serviceSchemaLocation">
         <Contents>oracle/apps/sample/hrService/</Contents>
      </StringRefAddr>
   </RefAddresses>
</Reference>
<Reference name="HrServiceConnection"
         className="oracle.adf.model.connection.webservice.impl.WebServiceConnectionImpl" xmlns="">
   <Factory
         className="oracle.adf.model.connection.webservice.api.WebServiceConnectionFactory"/>
   <RefAddresses>
      <XmlRefAddr addrType="WebServiceConnection">
```

```
        <Contents>
           <wsconnection
              description="http://rws65094fwks:7202/MySampleSvc/HrService?WSDL"
              service="{http://xmlns.oracle.com/apps/sample/hrService/}HrService">
              <model
                 name="{http://xmlns.oracle.com/apps/sample/hrService/}HrService"
                 xmlns="http://oracle.com/ws/model">
                 <service name="{http://xmlns.oracle.com/apps/sample/hrService/}HrService">
                    <port name="HrServiceSoapHttpPort"
                       binding="{http://xmlns.oracle.com/apps/sample/hrService/}HrServiceSoapHttp"
                       portType="http://xmlns.oracle.com/apps/sample/hrService/}HrService">
                       <call-properties xmlns="http://oracle.com/adf">
                          <call-property id="csf-key" xmlns="">
                              <name>csf-key</name>
                              <value>meuser.credentials</value>
                           </call-property>
                       </call-properties>
                       <policy-references xmlns="http://oracle.com/adf">
                          <policy-reference category="security"
                           uri="oracle/wss11_username_token_with_message_protection_client_policy"
                           enabled="true"
                           id="oracle/wss11_username_token_with_message_protection_client_policy"
                           xmlns=""/>
                       </policy-references>
                       <soapaddressUrl="http://rws65094fwks:7202/MySampleSvc/HrService"
                           xmlns="http://schemas.xmlsoap.org/wsdl/soap/"/>
                    </port>
                 </service>
              </model>
           </wsconnection>
        </Contents>
     </XmlRefAddr>
  </RefAddresses>
  </Reference>
```

To register the published service with your client application so the the user identity will be propagated to the caller, specify the clientside policy `oracle/wss11_saml_token_with_message_protection_client_policy` in the `connections.xml` file following the example in Example 11–19.

Example 11–19 Client connections.xml File Registers the Remote Business Components Service for the SOAP Protocol Using Identify Propagation

```
<Reference name="{http://xmlns.oracle.com/apps/sample/hrService/}HrService"
className="oracle.jbo.client.svc.Service" xmlns="">
   <Factory className="oracle.jbo.client.svc.ServiceFactory"/>
   <RefAddresses>
      <StringRefAddr addrType="serviceInterfaceName">
            <Contents>oracle.apps.sample.hrService.HrService</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="serviceEndpointProvider">
         <Contents>SOAP</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="webServiceConnectionName">
         <Contents>HrServiceConnection</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="serviceSchemaName">
         <Contents>HrService.xsd</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="serviceSchemaLocation">
```

```
         <Contents>oracle/apps/sample/hrService/</Contents>
      </StringRefAddr>
   </RefAddresses>
</Reference>
<Reference name="HrServiceConnection"
         className="oracle.adf.model.connection.webservice.impl.WebServiceConnectionImpl" xmlns="">
   <Factory
         className="oracle.adf.model.connection.webservice.api.WebServiceConnectionFactory"/>
   <RefAddresses>
      <XmlRefAddr addrType="WebServiceConnection">
         <Contents>
            <wsconnection
               description="http://rws65094fwks:7202/MySampleSvc/HrService?WSDL"
               service="{http://xmlns.oracle.com/apps/sample/hrService/}HrService">
               <model
                  name="{http://xmlns.oracle.com/apps/sample/hrService/}HrService"
                  xmlns="http://oracle.com/ws/model">
                  <service
                     name="{http://xmlns.oracle.com/apps/sample/hrService/}HrService">
                     <port name="HrServiceSoapHttpPort"
                        binding="{http://xmlns.oracle.com/apps/sample/hrService/}HrServiceSoapHttp"
                        portType="http://xmlns.oracle.com/apps/sample/hrService/}HrService">
                        <policy-references xmlns="http://oracle.com/adf">
                           <policy-reference category="security"
                                uri="oracle/wss11_saml_token_with_message_protection_client_policy"
                                enabled="true"
                                id="oracle/wss11_saml_token_with_message_protection_client_policy"
                                xmlns=""/>
                        </policy-references>
                        <soap addressUrl="http://rws65094fwks:7202/MySampleSvc/HrService"
                              xmlns="http://schemas.xmlsoap.org/wsdl/soap/"/>
                     </port>
                  </service>
               </model>
            </wsconnection>
         </Contents>
      </XmlRefAddr>
   </RefAddresses>
</Reference>
```

11.3.4.4 Registering the ADF Business Components Service in the Consuming Application's connections.xml for Fabric SDO Binding

When the service endpoint provider is Fabric, the service factory will return a SOA Fabric composite proxy and call the service running inside a Fabric composite through Fabric's SDO binding. You must provide the name of the Fabric composite to allow the consuming application to look up the published service.

Lookup information that you provide to register the published ADF Business Components service appears in the consuming Fusion web application's `connections.xml` file, located in the `.adf/META-INF` folder relative to the application. The ADF connection architecture uses this file to encapsulate the details of the service endpoint provider.

Note: When you deploy the calling application to standalone Oracle WebLogic Server, you will use Oracle Enterprise Manager to configure the JNDI context properties instead of editing the `connections.xml` file. For instructions, refer to the online documentation in Oracle Enterprise Manager.

To register the published service with your client application for the Fabric protocol, update the `connections.xml` file following the example in Example 11–20, where `fabricAddress` is the name of the Fabric composite for the published service.

Example 11–20 Client connections.xml File Registers the Remote Business Components Service for the SOA Fabric SDO Binding

```
<References xmlns="http://xmlns.oracle.com/adf/jndi">
  <Reference name="{www.globalcompany.com}StoreFrontService"
             className="oracle.jbo.client.svc.Service" xmlns="">
    <Factory className="oracle.jbo.client.svc.ServiceFactory"/>
    <RefAddresses>
      <StringRefAddr addrType="serviceInterfaceName">
       <Contents>oracle.foddemo.storefront.store.service.common.
                                          serviceinterface.StoreFrontService</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="serviceEndpointProvider">
         <Contents>Fabric</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="fabricAddress">
        <Contents>fabric_StoreFrontService</Contents>
      <StringRefAddr addrType="serviceSchemaName">
         <Contents>StoreFrontAMService.xsd</Contents>
      </StringRefAddr>
      <StringRefAddr addrType="serviceSchemaLocation">
         <Contents>oracle/fodemo/storefront/store/service/common/serviceinterface/</Contents>
      </StringRefAddr>
    </RefAddresses>
  </Reference>
  ...
</References>
```

11.3.5 How to Test the Service-Backed Components in the Oracle ADF Model Tester

Before you can launch the Oracle ADF Model Tester, your project must meet the runtime requirements as described in Section 11.3.4, "How to Configure the Service-Backed Business Components Runtime." The Oracle ADF Model Tester will display the view objects you create from the remote service and allow you to interact with the service to perform standard CRUD operations.

Because the application module that you run can access locally queried data and remotely queried data together, service-backed view objects and database-derived view objects will display in the same tester. If the endpoint is unavailable at the time you select the service-backed view object in the Oracle ADF Model Tester, you will get a runtime exception.

For details about running the Oracle ADF Model Tester, see Section 6.3.1, "How to Run the Oracle ADF Model Tester."

11.3.6 How to Invoke Operations of the Service-Backed Components in the Consuming Application

The ADF Business Components service interface requires that you return a service proxy to ensure that operations you invoke use the transport protocol specified by the published service.

Before you begin:

You need to ensure that the consuming application has the correct libraries on the classpath. In the Application Navigator, double-click the project and in the Project Properties dialog, select **Libraries and Classpath** and confirm the following libraries appear:

- **Java EE 1.5**
- **Oracle XML Parser v2**
- **BC4J Service Client**
- **JAX-WS Client**
- The service's common JAR file

As Example 11–21 shows, when you invoke the operation, you perform the following tasks:

1. Import the `oracle.jbo.client.svc.ServiceFactory` class and published service class.
2. Call `getServiceProxy()` on the service factory object and pass in the service name in the form *`<serviceName>`*`.NAME`. The ADF service factory embeds a `SDOHelperContext` ID in the service proxy object returned by this method to ensure delivery of the latest ADF Business Component service schema metadata to the SDO.

   The schema (`.xsd` files) for the service object may be stored in MDS and may have been extended for example to add more business component attributes, extend existing types, or define new types. The local helper context allows customization of individual service's schema definitions without affecting other service's SDO metadata or requiring restarting the application.
3. Call `create()` on a data factory object, where the proxy object is obtained from the `getServiceProxy()` call.
4. Invoke the operation on the proxy object and return a data object.
5. Save the data object return as XML.

Example 11–21 Obtaining and Invoking a Service Proxy in the Consuming Application

```
import commonj.sdo.DataObject;
import commonj.sdo.helper.DataFactory;
import commonj.sdo.helper.XMLHelper;
import hr.common.Dept;
import hr.common.serviceinterface.HRAppService;
import oracle.jbo.client.svc.ServiceFactory;
   ...
   {
      HRAppService proxy = (HRAppService) ServiceFactory.getServiceProxy(HRAppService.NAME);

      Dept dept = (Dept)
               ServiceFactory.getDataFactory(proxy).create("http://example.com/hr/common/" "Dept");
      dept.setDname("ENGINEERING");
```

```
    ...
    dept = proxy.createDept(dept);
    String xml = ServiceFactory.getXMLHelper(proxy).save((DataObject) dept,
                                "http://example.com/hr/common/", "dept");
    out.print(xml);
}
```

11.3.7 What You May Need to Know About Creating Service Data Objects in the Consuming Application

The ADF service interface framework defines a set of common data objects, such as `oracle.jbo.common.service.types.FindCriteria`, `oracle.jbo.common.service.types.FindControl`, `oracle.jbo.common.service.types.ProcessControl`, and so on. Those data objects are used in standard `findXxx()` method and `processXxx()` method calls. You can create these data objects in the consuming application and pass them to the standard service methods. As Example 11–22 shows, you can need to call the data factory class' `create()` method to construct those data objects before passing them into the `findXxx()` or `processXxx()` methods.

Example 11–22 Creating Data Objects in the Consuming Application

```
FindCriteria fc = (FindCriteria) ServiceFactory.getDataFactory(proxy).create
                        ("http://xmlns.oracle.com/adf/svc/types/", "FindCriteria");
```

11.3.8 What Happens at Runtime: How the Application Accesses the Published Application Module

The ADF runtime obtains the data source information from the service-backed entity object XML definition to automate interactions with the service interface methods as needed. By using the service-backed entity object, you avoid having to work directly with the service proxy and service data objects programmatically for all common remote service data access tasks. The ADF service factory looks up the service and then uses the service interface you specified in the `connections.xml` to invoke the service methods.

When your application accesses a remote ADF Business Components service, each remote call is stateless, and the remote service will not participate in the same transaction as the business component that uses a service-enabled application module's service interface.

In the majority of the cases, calls to remote services will be informational in nature and will not make changes to remote objects. However, if you must use a remote service to make changes, then keep these points in mind:

- An exception thrown by the remote service will cause the local transaction to fail.
- If you successfully call a remote service that results in modifying data, and then subsequently your local transaction fails for any reason, then it is the responsibility of your error handling code to perform a compensating transaction against the remote service to "undo" the previous change made.

11.3.9 What You May Need to Know About Service-Backed Entity Objects and View Objects

You will use some web services to access reference information. However, other services you call may modify data. This data modification might be in your own company's database if the service was written by a member of your own or another

team in your company. If the web service is outside your firewall, of course the database being modified will be managed by another company. In either of these situations, it is important to understand that any data modifications performed by a web service you invoke will occur in its own distinct transaction unrelated to the service-enabled application module's current unit of work. For example, if you have invoked a web service that modifies data and then you later call `rollback()` to cancel the pending changes in the application module's current unit of work, rolling back the changes has no effect on the changes performed by the web service you called in the process. You may need to invoke a corresponding web service method to perform a compensating change to account for your rollback of the application module's transaction.

At runtime, ADF handles the interaction with the remote ADF Business Components service. However, you should be aware that service-backed business components have the following design time restrictions that may restrict your application's runtime behavior. For more details about how these restrictions apply at design time, see Section 11.3.3, "How to Update the Data Model for Service-Backed Business Components."

- View objects that you create cannot reference a service-backed entity object as a secondary entity object usage.
- View objects that you create cannot produce a flattened join from two or more related entity objects when at least one of those entity objects is a service-backed entity object.
- Service-backed view objects that you create from service-backed entity objects will not reference secondary entity usages.

For more details about how these restrictions apply at design time, see Section 11.3.3, "How to Update the Data Model for Service-Backed Business Components."

12

Extending Business Components Functionality

This chapter describes techniques for extending all types of ADF Business Components objects.

This chapter includes the following sections:

- Section 12.1, "About Extending Business Components Functionality"
- Section 12.2, "Globally Extending ADF Business Components Functionality"
- Section 12.3, "Creating a Layer of Framework Extensions"
- Section 12.4, "Customizing Framework Behavior with Extension Classes"
- Section 12.5, "Creating Generic Extension Interfaces"
- Section 12.6, "Invoking Stored Procedures and Functions"
- Section 12.7, "Accessing the Current Database Transaction"
- Section 12.8, "Customizing Business Components Error Messages"
- Section 12.9, "Creating Extended Components Using Inheritance"
- Section 12.10, "Substituting Extended Components in a Delivered Application"

12.1 About Extending Business Components Functionality

When you need to add custom code to extend the base functionality of the ADF Business Components framework, you can enable a custom Java class for any of the key types of ADF business components you create. You enable the generation of custom classes for a component on the Java page of its respective overview editor in JDeveloper. When you enable this option, JDeveloper creates a Java source file for a custom class related to the component whose name follows a configurable naming standard. This class, whose name is recorded in the component's XML component definition, provides a place where you can write the custom Java code required by that component.

When used without customization, your component is completely defined by its XML component definition and it will be fully functional without custom Java code or even a Java class file for the component. If you have no need to extend the built-in functionality of a component in ADF Business Components, and no need to write any custom code to handle its built-in events, you can use the component in this XML-only fashion.

12.1.1 Extending Business Components Use Cases and Examples

An ADF Business Components framework extension class is Java class you write that extends one of the framework's base classes to:

- Augment a built-in feature works with additional, generic functionality
- Change how a built-in feature works, or even to
- Workaround a bug you encounter in a generic way

12.1.2 Additional Functionality for Extending Business Components

You may find it helpful to understand other ADF features before you start working with the ADF Business Components framework. Following are links to other functionality that may be of interest.

- For details about creating a reusable library to make your framework extension layer classes easier to package, see Chapter 38, "Reusing Application Components."
- For details about using the customization features provided by Oracle Metadata Services (MDS) to create applications that can be customized and subsequently deployed by a customer, see Chapter 39, "Customizing Applications with MDS."
- For a quick reference to the most common code that you will typically write, use, and override in your custom classes, see Appendix D, "Most Commonly Used ADF Business Components Methods."
- For API documentation related to the `oracle.jbo` package, see the following Javadoc reference document:
  - *Oracle Fusion Middleware Java API Reference for Oracle ADF Model*

12.2 Globally Extending ADF Business Components Functionality

One of the powerful features of framework-based development is the ability to extend the base framework to change a built-in feature to behave differently or to add a new feature that can be used by all of your applications.

Note: To experiment with the examples in this chapter, use the `AdvancedExamples` workspace in the `StandaloneExamples` module of the Fusion Order Demo application, as described in Section 2.4.3, "Standalone Applications in the AdvancedExamples Application Workspace." For information about how to obtain and install the Fusion Order Demo, see Section 2.2, "Setting Up the Fusion Order Demo Application."

12.2.1 How To Create a Framework Extension Class

An ADF Business Components framework extension class is Java class you write that extends one of the framework's base classes to:

- Augment a built-in feature works with additional, generic functionality
- Change how a built-in feature works, or even to
- Workaround a bug you encounter in a generic way

Once you've created a framework extension class, any new ADF components you create can be based on your *customized* framework class instead of the base one. Of

course, you can also update the definitions of existing components to use the new framework extension class as well.

To create a framework extension class:

1. Identify a project to contain the framework extension class.

   You can create it in the same project as your business service components if you believe it will only be used by components in that project. Alternatively, if you believe you might like to reuse the framework extension class across multiple Fusion web applications, create a separate `FrameworkExtensions` project to contain the framework extension classes.

2. Ensure the **BC4J Runtime** library is in the project's libraries list.

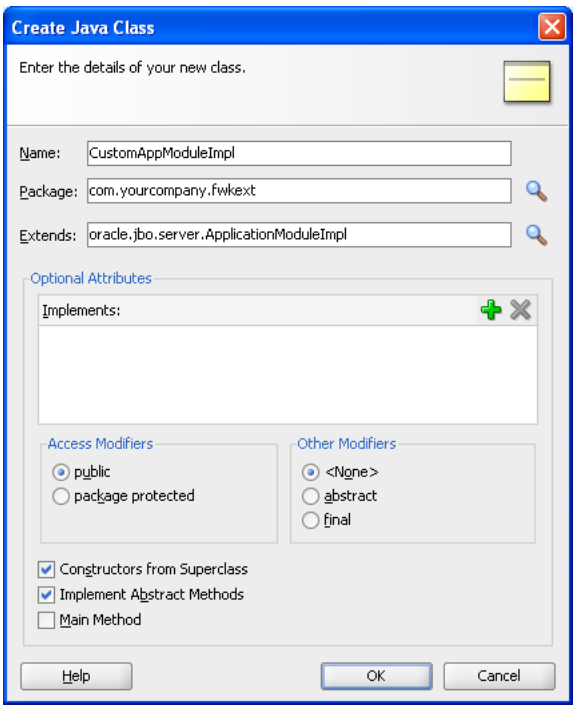
   Use the **Libraries** page of the **Project Properties** dialog to verify this and to add the library if missing.

3. In the **New Gallery**, in the **General** category, select **Java Class** to create the new class.

4. In the Create Java Class dialog, specify the appropriate framework base class from the `oracle.jbo.server` package in the **Extends** field.

   Figure 12–1 illustrates what it would look like to create a custom framework extension class named `CustomAppModuleImpl` in the `com.yourcompany.fwkext` package to customize the functionality of the base application module component. To quickly find the base class you're looking for, use the **Browse** button next to the **Extends** field that launches the JDeveloper **Class Browser**. Using its **Search** tab, you can type in part of the class name (including using * as a wildcard) to quickly subset the list of classes to find the one you're looking for.

Figure 12–1 Creating a Framework Extension Class for an Application Module



When you click **OK**, JDeveloper creates the custom framework extension class for you in the directory of the project's source path corresponding to the package name you've chosen.

Note: Some ADF Business Components component classes exist in both a server-side and a remote-client version. For example, if you use the JDeveloper **Class Browser** and type `ApplicationModuleImpl` into the **Match Class Name** field on the **Search** tab, the list will show two ApplicationModuleImpl classes: one in the `oracle.jbo.server` package and the other in the `oracle.jbo.client.remote` package. When creating framework extension classes, use the base ADF classes in the `oracle.jbo.server` package.

12.2.2 What Happens When You Create a Framework Extension Class

After creating a new framework extension class, it will not automatically be used by your application. You must decide which components in your project should make use of it. The following sections describe the available approaches for basing your ADF components on your own framework extension classes.

12.2.3 How to Base an ADF Component on a Framework Extension Class

You can set the base classes for any ADF component using the **Java** page of any ADF Business Components wizard or editor.

Before you begin:

- Create the framework extension class, as described in Section 12.2.1, "How To Create a Framework Extension Class."
- If you created your framework extension classes in a separate project, visit the **Dependencies** page of the Project Properties dialog for the project containing your business components and mark the `FrameworkExtension` project as a project dependency.
- If you have packaged your framework extension classes in a Java archive (JAR) file, create a named library definition to reference its JAR file and also list that library in the library list of the project containing your business components. To create a library if missing, use the Manage Libraries dialog available from the **Tools > Manage Libraries** main menu item. To verify or adjust the project's library list, use the **Libraries** page of the Project Properties dialog.

After you ensure the framework classes are available to reference, you can create the ADF component. Every ADF Business Components wizard and editor displays the same **Class Extends** button on the **Java** page so you can use the technique to choose your desired framework extension base class(es) both for new components or existing ones.

There is no fixed limit on how many levels of framework extension classes you create. For example, after creating a company-level `CustomAppModuleImpl` to use for all application modules in all Fusion web applications that your company creates, some later project team might encounter the need to further customize that framework extension class. That team could create a `SomeProjectCustomAppModuleImpl` class that extends the `CustomAppModuleImpl` and then include the project-specific custom application module code in there as shown in Example 12–1.

Example 12–1 Extending a Custom Class

```
public class SomeProjectCustomAppModuleImpl
       extends CustomAppModuleImpl {
  /*
   * Custom application module code specific to the
   * "SomeProject" project goes here.
   */
}
```

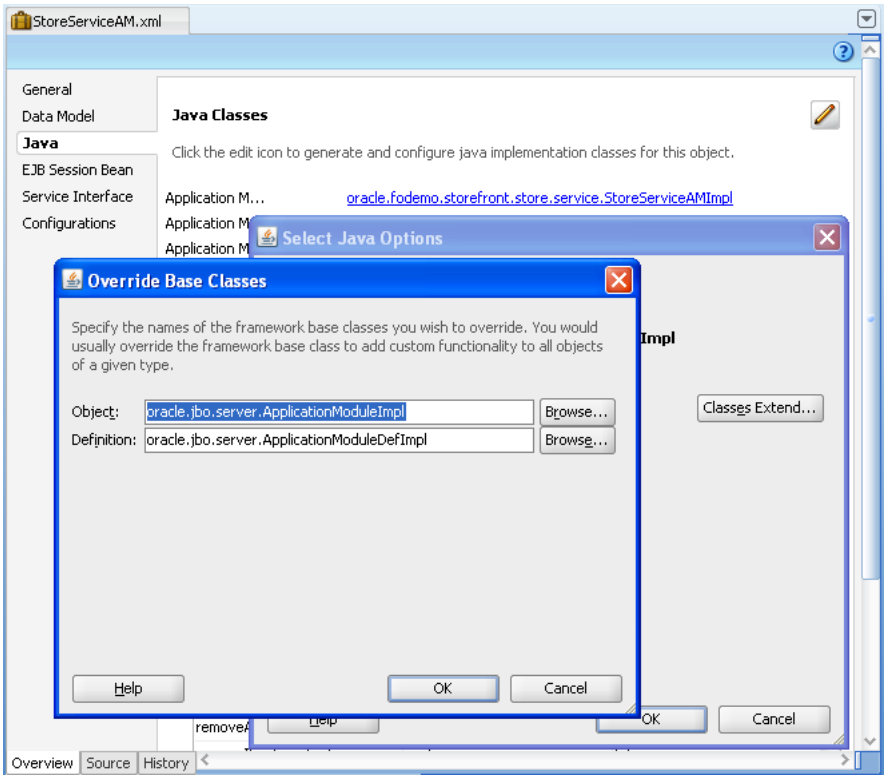
Then, any application modules created as part of the implementation of this specific project can use the `SomeProjectCustomAppModuleImpl` as their base class instead of the `CustomAppModuleImpl`.

To create an ADF component based on a framework extension class:

1. In the Application Navigator, double-click the desired component.
2. In the overview editor, click the **Java** navigation tab and click the **Edit Java options** button.
3. In the Select Java Options dialog, click **Classes Extends**.
4. In the Override Base Classes dialog, enter the fully-qualified name of the framework base classes you wish to override. You can also use the **Browse** button to use the JDeveloper Class Browser to find the classes quickly.

   When you use the Class Browser to select a custom base class for the component, the list of available classes is automatically filtered to show only classes that are appropriate. For example, when clicking **Browse** in Figure 12–2 to select an application module **Object** base class, the list will only show classes available in the current project's library list which extend the `oracle.jbo.server.ApplicationModule` class either directly or indirectly. If you don't see the class you're looking for, either you extended the incorrect base class or you have chosen the wrong component class name to override.

Figure 12–2 Specifying a Custom Base Class for a New Application Module



12.2.4 How to Define Framework Extension Classes for All New Components

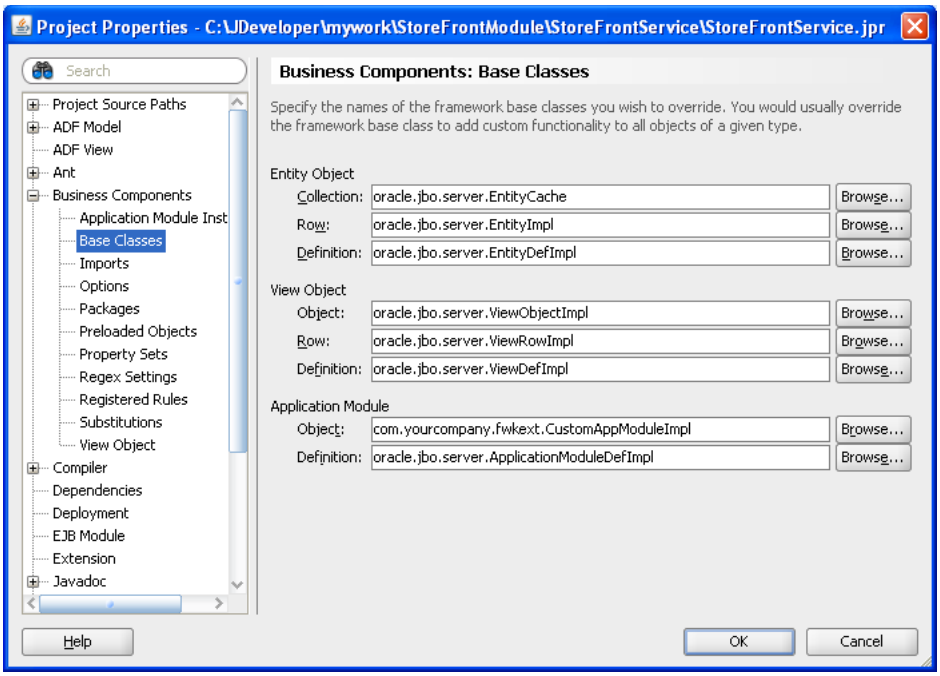
If you decide to use a specific set of framework extension classes as a standard for a given project, you can use the Project Properties dialog to define your preferred base classes for each component type. Setting these preferences for base classes does not affect any existing components in the project, but the component wizards will use the preferences for any new components created.

To define project-level preferences for framework extension classes:

1. In the Application Navigator, right-click the data model project that will contain the extension classes and choose **Project Properties**.
2. In the Project Properties dialog, expand **Business Components > Base Classes** in the tree.
3. On the Business Components page, enter the fully-qualified name of that class in the **Application Module Object** class name field.

   For example, to indicate that any new application modules created in the project should use the `CustomAppModuleImpl` class by default, enter the fully-qualified name of that class in the ***componentName* Object** class name field as shown in Figure 12–3.

Figure 12–3 Setting Project-Level Preferences for ADF Component Base Classes



12.2.5 How to Define Framework Extension Classes for All New Projects

When you want to apply the same base class preferences to each new project that you create in JDeveloper, you can define the preferences at a global level using the Preferences dialog. Base classes that you specify at the global level will not alter your existing projects containing ADF components.

To define global preferences for framework extension classes:

1. In the Tools menu, choose Preferences.
2. In the Preferences dialog, expand **Business Components > Base Classes** in the tree.
3. On the Business Components page, enter the fully-qualified name of that class in ***componentName* Object** class name field.

   The page displays the same options for specifying the preferred base classes for each component type as shown in Figure 12–3.

12.2.6 What Happens When You Base a Component on a Framework Extension Class

When an ADF component you create extends a custom framework extension class, JDeveloper updates its XML component definition to reflect the custom class name you've chosen.

12.2.6.1 XML-Only Components

For example, assume you've created the `YourService` application module in the `com.yourcompany.yourapp` package, with a custom application module base class of `CustomAppModuleImpl`. If you have opted to leave the component as an XML-only component with no custom Java file, its XML component definition (YourService.xml) will look like what you see in Example 12–2. The value of the

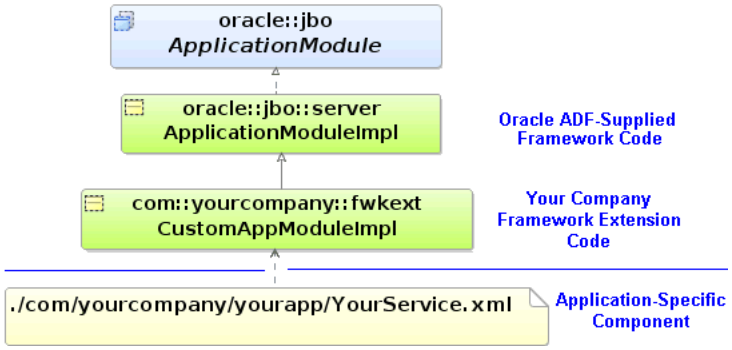
`ComponentClass` attribute of the <AppModule> tag is read at runtime to identify the Java class to use to represent the component.

Example 12–2 Custom Base Class Names Are Recorded in XML Component Definition

```
<AppModule
   Name="YourService"
   ComponentClass="com.yourcompany.fwkext.CustomAppModuleImpl" >
  <!-- etc. -->
</AppModule>
```

Figure 12–4 illustrates how the XML-only `YourService` application module relates to your custom extension class. At runtime, it uses the `CustomAppModuleImpl` class which inherits its base behavior from the `ApplicationModuleImpl` class.

Figure 12–4 XML-Only Component Reference an Extended Framework Base Class



12.2.6.2 Components with Custom Java Classes

If your component requires a custom Java class, as you've seen in previous chapters you open the **Java** page of the component editor and check the appropriate checkbox to enable it. For example, when you enable a custom application module class for the `YourServer` application module, JDeveloper creates the appropriate `YourServiceImpl.java` class. As shown in Example 12–3, it also updates the component's XML component definition to reflect the name of the custom component class.

Example 12–3 Custom Component Class Recorded in XML Component Definition

```
<AppModule
   Name="YourService"
   ComponentClass="com.yourcompany.yourapp.YourServiceImpl" >
  <!-- etc. -->
</AppModule>
```

JDeveloper also updates the component's custom Java class to modify its `extends` clause to reflect the new custom framework base class, as shown in Example 12–4.

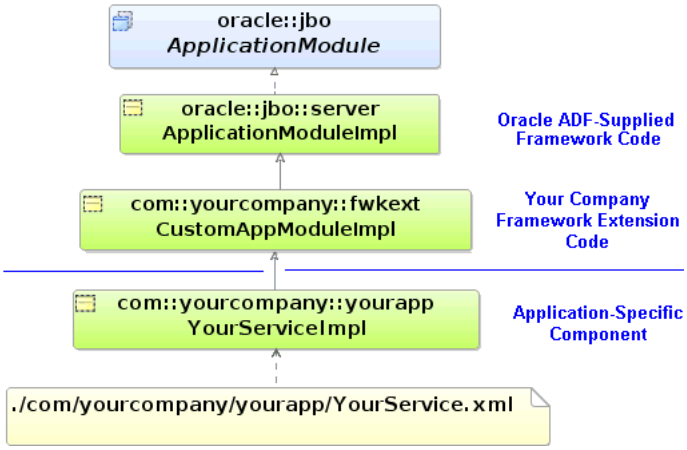
Example 12–4 Component's Custom Java Class Updates to Reflect New Base Class

```
package com.yourcompany.yourapp;
import com.yourcompany.fwkext.CustomAppModuleImpl;
// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---    Custom code may be added to this class.
// ---    Warning: Do not modify method signatures of generated methods.
// ---------------------------------------------------------------------
```

```
public class YourServiceImpl extends CustomAppModuleImpl {
  /**This is the default constructor (do not remove)   */
  public YourServiceImpl() {}
  // etc.
}
```

Figure 12–5 illustrates how the `YourService` application module with its custom `YourServiceImpl` class is related to your framework extension class. At runtime, it uses the `YourServiceImpl` class which inherits its base behavior from the `CustomAppModuleImpl` framework extension class which, in turn, extends the base `ApplicationModuleImpl` class.

Figure 12–5 Component with Custom Java Extending Customized Framework Base Class



12.2.7 What You May Need to Know About Updating the Extends Clause in Custom Component Java Files

If you have an ADF component with a custom Java class and later decide to base the component on a framework extension class, use the **Class Extends** button on the **Java** page of the component's overview editor to change the component's base class. Doing this updates the component's XML component definition to reflect the new base class, and *also* modifies the `extends` clause in the component's custom Java class. If you manually update the `extends` clause without using the component editor, the component's XML component definition will not reflect the new inheritance and the next time you open the editor, your manually modified `extends` clause will be overwritten with what the component editor believes is the correct component base class.

12.3 Creating a Layer of Framework Extensions

Before you begin to develop application-specific business components, Oracle recommends that you consider creating a complete layer of framework extension classes and setting up your project-level preferences to use that layer by default. You might not have any custom code in mind to put in these framework extension classes yet, but you will be glad you heeded this recommendation the first time you encounter a need to:

- Add a generic feature that all your company's application modules require
- Augment a built-in feature with some custom, generic processing

- Workaround a bug you encounter in a generic way

Failure to set up these preferences at the outset can present your team with a substantial inconvenience if you discover mid-project that all of your entity objects, for example, require a new generic feature, augmented built-in feature, or a generic bug workaround. Putting a complete layer of framework classes in place at the start of your project is an insurance policy against this inconvenience and the wasted time related to dealing with it later in the project. JDeveloper will automatically use framework classes when you create them.

12.3.1 How to Create Your Layer of Framework Extension Layer Classes

A common set of customized framework base classes in a package name of your own choosing like *com.yourcompany.adfextensions*, each importing the `oracle.jbo.server.*` package, would consist of the following classes:

- `public class CustomEntityImpl extends EntityImpl`
- `public class CustomEntityDefImpl extends EntityDefImpl`
- `public class CustomViewObjectImpl extends ViewObjectImpl`
- `public class CustomViewRowImpl extends ViewRowImpl`
- `public class CustomApplicationModuleImpl extends ApplicationModuleImpl`
- `public class CustomDBTransactionImpl extends DBTransactionImpl2`
- `public class CustomDatabaseTransactionFactory extends DatabaseTransactionFactory`

To make your framework extension layer classes easier to package as a reusable library, create them in a separate project from the projects that use them.

For details about using the custom `DBTransactionImpl` class, see Section 12.8.5.2, "Configuring an Application Module to Use a Custom Database Transaction Class."

> **Note:** For your convenience, the `FrameworkExtensions` project of the `AdvancedExamples` workspace in the `StandaloneExamples` module of the Fusion Order Demo application contains a set of these classes. You can select the `com.yourcompany.adfextensions` package in the Application Navigator and choose the **Refactor > Rename** option from the context menu to change the package name of all the classes to a name you prefer.

For completeness, you may also want to create customized framework classes for the following classes as well, note however that overriding anything in these classes would be a fairly rare requirement.

- `public class CustomViewDefImpl extends ViewDefImpl`
- `public class CustomEntityCache extends EntityCache`
- `public class CustomApplicationModuleDefImpl extends ApplicationModuleDefImpl`

12.3.2 How to Package Your Framework Extension Layer in a JAR File

Use the **Create Deployment Profile: JAR File** dialog to create a JAR file containing the classes in your framework extension layer. This is available in the **New Gallery** in the **General > Deployment Files** category.

Give the deployment profile a name like `FrameworkExtensions` and click **OK**. By default the JAR file will include all class files in the project. Since this is exactly what you want, when the **JAR Deployment Profile Properties** dialog appears, you can just click **OK** to finish.

> **Note:** Do not use the **ADF Library JAR** archive type to package your framework extension layer. You create the ADF Library JAR file when you want to package reusable components to share in the JDeveloper Resource Catalog. For details about working with ADF components and the ADF Library JAR archive type, see Section 38.3, "Packaging a Reusable ADF Component into an ADF Library."

Finally, to create the JAR file, right-click the project node in the Application Navigator and choose **Deploy - *YourProfileName* - to JAR File** on the context menu. A **Deployment** tab appears in the JDeveloper **Log window** that should display feedback like:

```
----  Deployment started.  ----    Feb 14, 2008 1:42:39 PM
Running dependency analysis...
Wrote JAR file to ...\FrameworkExtensions\deploy\FrameworkExtensions.jar
Elapsed time for deployment:  2 seconds
----  Deployment finished.  ----    Reb 14, 2008 1:42:41 PM
```

12.3.3 How to Create a Library Definition for Your Framework Extension JAR File

JDeveloper uses named libraries as a convenient way to organize the one or more JAR files that comprise reusable component libraries.

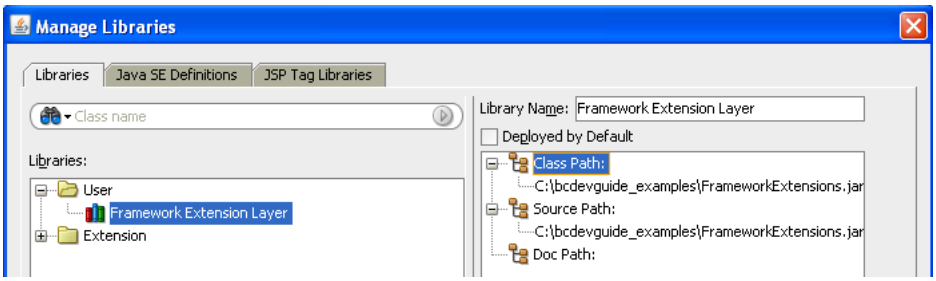
To define a library for your framework extensions JAR file:

1. Choose **Tools > Manage Libraries** from the main menu.
2. In the **Manage Libraries** dialog, select the **Libraries** tab.
3. Select the **User** node in the tree and click the **New** button.
4. In the **Create Library** dialog that appears, name the library "Framework Extension Layer" and select the **Class Path** node and click **Add Entry**.
5. Use the **Select Path Entry** dialog that appears to select the `FrameworkExtensions.jar` file that contains the class files for the framework extension components, then click **Select**.
6. Select the **Source Path** node and click **Add Entry.**
7. Use the **Select Path Entry** dialog that appears to select the `..\FrameworkExtensions\src` directory where the source files for the framework extension classes reside, then click **Select**.
8. Click **OK** to dismiss the **Create Library** dialog and define the new library.

When finished, you will see your new "Framework Extension Layer" user-defined library, as shown in Figure 12–6. You can then add this library to the library list of any project where you will be building business services, and your custom framework

extension classes will be available to reference as the preferred component base classes.

Figure 12–6 New User-Defined Library for Your Framework Extensions Layer



12.4 Customizing Framework Behavior with Extension Classes

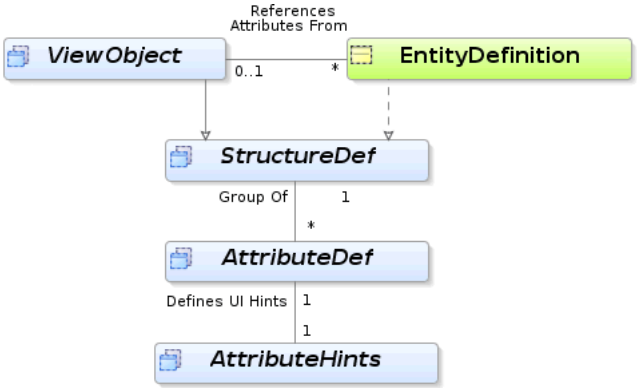
One of the common tasks you'll perform in your framework extension classes is implementing custom application functionality. Since framework extension code is written to be used by all components of a specific type, the code you write in these classes often needs to work with component attributes in a generic way. To address this need, ADF provides API's that allow you to access component metadata at runtime. It also provides the ability to associate custom metadata properties with any component or attribute. You can write your generic framework extension code to leverage runtime metadata and custom properties to build generic functionality, which if necessary, only is used in the presence of certain custom properties.

Note: The example in this section refers to the `ProgrammaticallySetProperties` project of the `AdvancedExamples` application workspace in the `StandaloneExamples` module of the Fusion Order Demo application.

12.4.1 How to Access Runtime Metadata For View Objects and Entity Objects

Figure 12–7 illustrates the three primary interfaces ADF provides for accessing runtime metadata about view objects and entity objects. The `ViewObject` interface extends the `StructureDef` interface. The class representing the entity definition (`EntityDefImpl`) also implements this interface. As its name implies, the `StructureDef` defines the structure and the component and provides access to a collection of `AttributeDef` objects that offer runtime metadata about each attribute in the view object row or entity row. Using an `AttributeDef`, you can access its companion `AttributeHints` object to reference hints like the display label, format mask, tooltip, etc.

Figure 12–7 Runtime Metadata Available for View Objects and Entity Objects



12.4.2 How to Implement Generic Functionality Using Runtime Metadata

In Section 6.4.1, "ViewObject Interface Methods for Working with the View Object's Default RowSet" you learned that for read-only view objects the `findByKey()` method and the `setCurrentRowWithKey` builtin operation only work if you override the `create()` method on the view object to call `setManageRowsByKey(true)`. This can be a tedious detail to remember if you create a lot of read-only view objects, so it is a great candidate for automating in a framework extension class for view objects.

Assume a `FrameworkExtensions` project contains a `FODViewObjectImpl` class that is the base class for all view objects in the application. This framework extension class for view objects extends the base `ViewObjectImpl` class and overrides the `create()` method as shown in Example 12–5 to automate this task. After calling the `super.create()` to perform the default framework functionality when a view object instance is created at runtime, the code tests whether the view object is a read-only view object with at least one attribute marked as a key attribute. If this is the case, it invokes `setManageRowsByKey(true)`.

The `isReadOnlyNonEntitySQLViewWithAtLeastOneKeyAttribute()` helper method determines whether the view object is read-only by testing the combination of the following conditions:

- `isFullSql()` is `true`

  This method returns true if the view object's SQL query is completely specified by the developer, as opposed to having the select list derived automatically based on the participating entity usages.

- `getEntityDefs()` is `null`

  This method returns an array of `EntityDefImpl` objects representing the view object's entity usages. If it returns `null`, then the view object has no entity usages.

It goes on to determine whether the view object has any key attributes by looping over the `AttributeDef` array returned by the `getAttributeDefs()` method. If the `isPrimaryKey()` method returns true for any attribute definition in the list, then you know the view object has a key.

Example 12–5 Automating Setting Manage Rows By Key

```
public class FODViewObjectImpl extends ViewObjectImpl {
  protected void create() {
    super.create();
    if (isReadOnlyNonEntitySQLViewWithAtLeastOneKeyAttribute()) {
```

```
      setManageRowsByKey(true);
    }
  }
  boolean isReadOnlyNonEntitySQLViewWithAtLeastOneKeyAttribute() {
    if (getViewDef().isFullSql() && getEntityDefs() == null) {
      for (AttributeDef attrDef : getAttributeDefs()) {
        if (attrDef.isPrimaryKey()) {
          return true;
        }
      }
    }
    return false;
  }
  // etc.
}
```

12.4.3 How to Implement Generic Functionality Driven by Custom Properties

When you create application modules, view objects, and entity objects you can select the **General** navigation tab in the overview editor for these business components and expand the **Custom Properties** section to define custom metadata properties for any component. These are name/value pairs that you can use to communicate additional declarative information about the component to the generic code that you write in framework extension classes. You can use the `getProperty()` method in your code to conditionalize generic functionality based on the presence of, or the specific value of, one of these custom metadata properties.

For example, the `FODViewObjectImpl` framework extension class overrides the view object's `insertRow()` method as shown in Example 12–6 to conditionally force a row to be inserted and to appear as the last row in the row set. If any view object extending this framework extension class defines a custom metadata property named `InsertNewRowsAtEnd`, then this generic code executes to insert new rows at the end. If a view object does not define this property, it will have the default `insertRow()` behavior.

Example 12–6 Conditionally Inserting New Rows at the End of a View Object's Default RowSet

```
public class FODViewObjectImpl extends ViewObjectImpl {
  private static final String INSERT_NEW_ROWS_AT_END = "InsertNewRowsAtEnd";
  public void insertRow(Row row) {
    super.insertRow(row);
    if (getProperty(INSERT_NEW_ROWS_AT_END) != null) {
      row.removeAndRetain();
      last();
      next();
      getDefaultRowSet().insertRow(row);
    }
  }
  // etc.
}
```

In addition to defining component-level custom properties, you can also define properties on view object attributes, entity object attributes, and domains. At runtime, you access them using the `getProperty()` method on the `AttributeDef` interface for a given attribute.

12.4.4 What You May Need to Know About the Kinds of Attributes

In addition to providing information about an attribute's name, Java type, SQL type, and many other useful pieces of information, the `AttributeDef` interface contains the `getAttributeKind()` method that you can use to determine the kind of attribute it represents. This method returns a `byte` value corresponding to one of the public constants in the `AttributeDef` interface listed in Table 12–1.

Table 12–1 Entity Object and View Object Attribute Kinds

Public `AttributeDef` Constant	**Attribute Kind Description**
`ATTR_PERSISTENT`	Persistent attribute
`ATTR_TRANSIENT`	Transient attribute
`ATTR_ENTITY_DERIVED`	View object attribute mapped to an entity-level transient attribute
`ATTR_SQL_DERIVED`	SQL-Calculated attribute
`ATTR_DYNAMIC`	Dynamic attribute
`ATTR_ASSOCIATED_ROWITERATOR`	Accessor attribute returning a `RowSet` of set of zero or more `Rows`
`ATTR_ASSOCIATED_ROW`	Accessor attribute returning a single `Row`

12.4.5 What You May Need to Know About Custom Properties

You may find it handy to programmatically set custom property values at runtime. While the `setProperty()` API to perform this function is by design not available to clients on the `ViewObject`, `ApplicationModule`, or `AttributeDef` interfaces in the `oracle.jbo` package, code you write *inside* your ADF components' custom Java classes can use it.

12.5 Creating Generic Extension Interfaces

In addition to creating framework extension classes, you can create custom interfaces that all of your components can implement by default. The client interface is very useful for exposing methods from your application module that might be invoked by UI clients, for example. This section considers an example for an application module, however, the same functionality is possible for a custom extended view object and view row interface as well. For more information about client interfaces, see also Section 9.9, "Publishing Custom Service Methods to UI Clients" and Section 9.10, "Working Programmatically with an Application Module's Client Interface."

Assume that you have a `CustomApplicationModuleImpl` class that extends `ApplicationModuleImpl` and that you want to expose two custom methods like this:

```
public void doFeatureOne(String arg);
public int anotherFeature(String arg);
```

Perform the following steps to create a custom extension interface `CustomApplicationModule` and have your `CustomApplicationModuleImpl` class implement it.

1. Create a custom interface that contains the methods you would like to expose globally on your application module components. For this scenario, that interface would look like this:

   ```
   package devguide.advanced.customintf.fwkext;
   ```

```
/**
 * NOTE: This does not extend the
 * ====  oracle.jbo.ApplicationModule interface.
 */
public interface CustomApplicationModule  {
  public void doFeatureOne(String arg);
  public int anotherFeature(String arg);
}
```

Notice that the interface does *not* extend the `oracle.jbo.ApplicationModule` interface.

2. Modify your `CustomApplicationModuleImpl` application module framework extension class to implement this new `CustomApplicationModule` interface.

```
package devguide.advanced.customintf.fwkext;
import oracle.jbo.server.ApplicationModuleImpl;
public class CustomApplicationModuleImpl
       extends ApplicationModuleImpl
       implements CustomApplicationModule {
  public void doFeatureOne(String arg) {
    System.out.println(arg);
  }
  public int anotherFeature(String arg) {
    return arg == null ? 0 : arg.length();
  }
}
```

3. Rebuild your project.

   The ADF wizards will only "see" your interfaces after they have been successfully compiled.

After your have implemented your `CustomApplicationModuleImpl` class, you can create a new application module which exposes the global extension interface and is based on your custom framework extension class. For this purpose you use the overview editor for application modules.

To create a custom application module interface:

1. In the Application Navigator, double-click the application module for which you want to create the custom interface.

   For example, you might create a new `ProductModule` application module which exposes the global extension interface `CustomApplicationModule` and is based on the `CustomApplicationModuleImpl` framework extension class.

2. In the overview editor, select the **Java** navigation tab and click the **Edit Java options** icon.

   The Java Classes page should show an existing Java class for the application module identified as **Application Module Class**.

   By default, JDeveloper generates the Java class for application modules you create. However, if you disabled this feature, click the **Edit Java options** button in the Java Classes section and select **Generate Application Module Class**. Click **OK** to add a Java class to the project from which you will create the custom interface.

3. In the Select Java Options dialog, click **Class Extends**.

4. In the Override Base Classes dialog, specify the name of the framework base class you want to override and click **OK**.

For example, you might select `CustomApplicationModuleImpl` as the base class for the application module.

5. In the Java Classes page of the overview editor, expand the **Client Interface** section and click the **Edit application module client interface** button.
6. In the Edit Client Interface dialog, click the **Interfaces** button.
7. In the Select Interfaces to Extend dialog, select the desired custom application module interface from the available list and click **OK**.

   For example, you might shuttle the `CustomApplicationModule` interface to the **Selected** list to be one of the custom interfaces that clients can use with your component.
8. In the Edit Client Interfaces dialog, ensure that at least one method appears in the **Selected** list.

> **Note:** You need to select at least one method in the **Selected** list in the Edit Client Interfaces dialog, even if it means redundantly selecting one of the methods on the global extension interface. Any method will do in order to get JDeveloper to generate the custom interface.

9. Click **OK**.

   The Java Classes page displays the new custom interface for the application module identified as **Application Module Client Interface**.

When you dismiss the Edit Client Interfaces dialog and return to the application module overview editor, JDeveloper generates the application module custom interface. For example, the custom interface `ProductModule` automatically extends *both* the base `ApplicationModule` interface and your `CustomApplicationModule` extension interface like this:

```
package devguide.advanced.customintf.common;
import devguide.advanced.customintf.fwkext.CustomApplicationModule;

import oracle.jbo.ApplicationModule;
// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---------------------------------------------------------------------
public interface ProductModule
       extends CustomApplicationModule, ApplicationModule {
  void doSomethingProductRelated();
}
```

Once you've done this, then client code can cast your `ProductModule` application module to a `CustomApplicationModule` interface and invoke the generic extension methods it contains in a strongly-typed way.

> **Note:** The basic steps are the same for exposing methods on a `ViewObjectImpl` framework extension class, as well as for a `ViewRowImpl` extension class.

12.6 Invoking Stored Procedures and Functions

You can write code in the custom Java classes for your business components to invoke database stored procedures and functions. Here you'll consider some simple examples based on procedures and functions in a PL/SQL package; however, using the same techniques, you also can invoke procedures and functions that are not part of a package.

Consider the PL/SQL package shown in Example 12–7.

Example 12–7 PL/SQL Package with Example Procedures

```
create or replace package devguidepkg as
  procedure proc_with_no_args;
  procedure proc_with_three_args(n number, d date, v varchar2);
  function  func_with_three_args(n number, d date, v varchar2) return varchar2;
  procedure proc_with_out_args(n number, d out date, v in out varchar2);
end devguidepkg;
```

The following sections explain how to invoke each of the example procedures and functions in this package.

> **Note:** The example in this section refers to the `StoredProcedureInvocation` project of the `AdvancedExamples` application workspace in the `StandaloneExamples` module of the Fusion Order Demo application.

12.6.1 How to Invoke Stored Procedures with No Arguments

If you need to invoke a stored procedure that takes no arguments, you can use the `executeCommand()` method on the `DBTransaction` interface (in the `oracle.jbo.server` package as shown in Example 12–8.

Example 12–8 Executing a Stored Procedure with No Arguments

```
// In StoredProcTestModuleImpl.java
public void callProcWithNoArgs() {
  getDBTransaction().executeCommand(
    "begin devguidepkg.proc_with_no_args; end;");
}
```

12.6.2 How to Invoke Stored Procedure with Only IN Arguments

Invoking stored procedures that accept only `IN`-mode arguments — which is the default PL/SQL parameter mode if not specified — requires using a JDBC `PreparedStatement` object. The `DBTransaction` interface provides a `createPreparedStatement()` method to create this object for you in the context of the current database connection. You could use a helper method like the one shown in Example 12–9 to simplify the job of invoking a stored procedure of this kind using a `PreparedStatement`. Importantly, by using a helper method, you can encapsulate the code that closes the JDBC `PreparedStatement` after executing it. The code performs the following basic tasks:

1. Creates a JDBC `PreparedStatement` for the statement passed in, wrapping it in a PL/SQL `begin...end` block.
2. Loops over values for the bind variables passed in, if any.
3. Sets the value of each bind variable in the statement.

Notice that since JDBC bind variable API's use one-based numbering, the code adds one to the zero-based for loop index variable to account for this.

4. Executes the statement.
5. Closes the statement.

Example 12–9 Helper Method to Simplify Invoking Stored Procedures with Only IN Arguments

```
protected void callStoredProcedure(String stmt, Object[] bindVars) {
  PreparedStatement st = null;
  try {
    // 1. Create a JDBC PreparedStatement for
    st = getDBTransaction().createPreparedStatement("begin "+stmt+";end;",0);
    if (bindVars != null) {
      // 2. Loop over values for the bind variables passed in, if any
      for (int z = 0; z < bindVars.length; z++) {
        // 3. Set the value of each bind variable in the statement
        st.setObject(z + 1, bindVars[z]);
      }
    }
    // 4. Execute the statement
    st.executeUpdate();
  }
  catch (SQLException e) {
    throw new JboException(e);
  }
  finally {
    if (st != null) {
      try {
        // 5. Close the statement
        st.close();
      }
      catch (SQLException e) {}
    }
  }
}
```

With a helper method like this in place, calling the `proc_with_three_args` procedure shown in Example 12–7 would look like this:

```
// In StoredProcTestModuleImpl.java
public void callProcWithThreeArgs(Number n, Date d, String v) {
  callStoredProcedure("devguidepkg.proc_with_three_args(?,?,?)",
                      new Object[]{n,d,v});
}
```

Notice the question marks used as JDBC bind variable placeholders for the arguments passed to the function. JDBC also supports using named bind variables, but using these simpler positional bind variables is also fine since the helper method is just setting the bind variable values positionally.

12.6.3 How to Invoke Stored Function with Only IN Arguments

Invoking stored functions that accept only `IN`-mode arguments requires using a JDBC `CallableStatement` object in order to access the value of the function result after executing the statement. The `DBTransaction` interface provides a `createCallableStatement()` method to create this object for you in the context of the current database connection. You could use a helper method like the one shown in

Example 12–10 to simplify the job of invoking a stored function of this kind using a `CallableStatement`. The helper method encapsulates both the creation and clean up of the JDBC statement being used.

The code performs the following basic tasks:

1. Creates a JDBC `CallableStatement` for the statement passed in, wrapping it in a PL/SQL `begin...end` block.
2. Registers the first bind variable for the function return value.
3. Loops over values for the bind variables passed in, if any.
4. Sets the value of each bind user-supplied bind variable in the statement.

   Notice that since JDBC bind variable API's use one-based numbering, and since the function return value is already the first bind variable in the statement, the code adds *two* to the zero-based for loop index variable to account for these.
5. Executes the statement.
6. Returns the value of the first bind variable.
7. Closes the statement.

Example 12–10 Helper Method to Simplify Invoking Stored Functions with Only IN Arguments

```
// Some constants
public static int NUMBER = Types.NUMERIC;
public static int DATE = Types.DATE;
public static int VARCHAR2 = Types.VARCHAR;

protected Object callStoredFunction(int sqlReturnType, String stmt,
                                    Object[] bindVars) {
  CallableStatement st = null;
  try {
    // 1. Create a JDBC CallabledStatement
    st = getDBTransaction().createCallableStatement(
           "begin ? := "+stmt+";end;",0);
    // 2. Register the first bind variable for the return value
    st.registerOutParameter(1, sqlReturnType);
    if (bindVars != null) {
      // 3. Loop over values for the bind variables passed in, if any
      for (int z = 0; z < bindVars.length; z++) {
        // 4. Set the value of user-supplied bind vars in the stmt
        st.setObject(z + 2, bindVars[z]);
      }
    }
    // 5. Set the value of user-supplied bind vars in the stmt
    st.executeUpdate();
    // 6. Return the value of the first bind variable
    return st.getObject(1);
  }
  catch (SQLException e) {
    throw new JboException(e);
  }
  finally {
    if (st != null) {
      try {
        // 7. Close the statement
        st.close();
      }
```

```
      catch (SQLException e) {}
    }
  }
}
```

With a helper method like this in place, calling the `func_with_three_args` procedure shown in Example 12–7 would look like this:

```
// In StoredProcTestModuleImpl.java
public String callFuncWithThreeArgs(Number n, Date d, String v) {
  return (String)callStoredFunction(VARCHAR2,
                            "devguidepkg.func_with_three_args(?,?,?)",
                            new Object[]{n,d,v});
}
```

Notice the question marks are used as JDBC bind variable placeholders for the arguments passed to the function. JDBC also supports using named bind variables, but using these simpler positional bind variables is also fine since the helper method is just setting the bind variable values positionally.

12.6.4 How to Call Other Types of Stored Procedures

Calling a stored procedure or function like `devguidepkg.proc_with_out_args` that includes arguments of `OUT` or `IN OUT` mode requires using a `CallableStatement` as in the previous section, but is a little more challenging to generalize into a helper method. Example 12–11 illustrates the JDBC code necessary to invoke the `devguidepkg.proc_with_out_args` procedure.

The code performs the following basic tasks:

1. Defines a PL/SQL block for the statement to invoke.
2. Creates the `CallableStatement` for the PL/SQL block.
3. Registers the positions and types of the `OUT` parameters.
4. Sets the bind values of the `IN` parameters.
5. Executes the statement.
6. Creates a JavaBean to hold the multiple return values

   The `DateAndStringBean` class contains bean properties named `dateVal` and `stringVal`.
7. Sets the value of its `dateVal` property using the first `OUT` param.
8. Sets value of its `stringVal` property using second `OUT` param.
9. Returns the result.
10. Closes the JDBC `CallableStatement`.

Example 12–11 Calling a Stored Procedure with Multiple OUT Arguments

```
public Date callProcWithOutArgs(Number n, String v) {
  CallableStatement st = null;
  try  {
    // 1. Define the PL/SQL block for the statement to invoke
    String stmt = "begin devguidepkg.proc_with_out_args(?,?,?); end;";
    // 2. Create the CallableStatement for the PL/SQL block
    st = getDBTransaction().createCallableStatement(stmt,0);
    // 3. Register the positions and types of the OUT parameters
    st.registerOutParameter(2,Types.DATE);
```

```
    st.registerOutParameter(3,Types.VARCHAR);
    // 4. Set the bind values of the IN parameters
    st.setObject(1,n);
    st.setObject(3,v);
    // 5. Execute the statement
    st.executeUpdate();
    // 6. Create a bean to hold the multiple return values
    DateAndStringBean result = new DateAndStringBean();
    // 7. Set value of dateValue property using first OUT param
    result.setDateVal(new Date(st.getDate(2)));
    // 8. Set value of stringValue property using 2nd OUT param
    result.setStringVal(st.getString(3));
    // 9. Return the result
    return result;
  } catch (SQLException e)  {
    throw new JboException(e);
  } finally  {
    if (st != null) {
      try {
        // 10. Close the JDBC CallableStatement
        st.close();
      }
      catch (SQLException e) {}
    }
  }
}
```

The `DateAndString` bean used in Example 12–11 is a simple JavaBean with two bean properties like this:

```
package devguide.advanced.storedproc;
import java.io.Serializable;
import oracle.jbo.domain.Date;
public class DateAndStringBean implements Serializable {
  Date dateVal;
  String stringVal;
  public void setDateVal(Date dateVal) {this.dateVal=dateVal;}
  public Date getDateVal() {return dateVal;}
  public void setStringVal(String stringVal) {this.stringVal=stringVal;}
  public String getStringVal() {return stringVal;}
}
```

> **Note:** In order to allow the custom method to be a legal candidate for inclusion in an application module's custom service interface (if desired), the bean needs to implement the `java.io.Serializable`. interface. Since this is a "marker" interface, this involves simply adding the `implements Serializable` keywords without needing to code the implementation of any interface methods.

12.7 Accessing the Current Database Transaction

Since the ADF Business Components components abstract all of the lower-level database programming details for you, you typically won't need *direct* access to the JDBC `Connection` object. Unless you use the reserved release mode described in Section 43.2.2.3.3, "About Reserved Release Level," there is no guarantee at runtime that your application will use the exact same application module instance or JDBC `Connection` instance across different web page requests. Since inadvertently holding

a reference to the JDBC Connection object in this type of pooled services environment can cause unpredictable behavior at runtime, by design, ADF Business Components has no direct API to obtain the JDBC `Connection`. This is an intentional attempt to discourage its direct use and inadvertent abuse.

However, on occasion it may come in handy when you're trying to integrate third-party code with ADF Business Components, so you can use a helper method like the one shown in Example 12–12 to access the connection.

Example 12–12 Helper Method to Access the Current JDBC Connection

```
/**
 * Put this method in your XXXImpl.java class where you need
 * to access the current JDBC connection
 */
private Connection getCurrentConnection() throws SQLException {
 /* Note that we never execute this statement, so no commit really happens */
 PreparedStatement st = getDBTransaction().createPreparedStatement("commit",1);
 Connection conn = st.getConnection();
 st.close();
 return conn;
}
```

Caution: Never cache the JDBC connection obtained using the helper method from Example 12–12 in your own code anywhere. Instead, call the helper method each time you need it to avoid inadvertently holding a reference to a JDBC Connection that might be used in another request by another user at a later time do to the pooled services nature of the ADF runtime environment.

12.8 Customizing Business Components Error Messages

Note: The example in this section refers to the `CustomizedErrorMessages` project of the `AdvancedExamples` application workspace in the `StandaloneExamples` module of the Fusion Order Demo application.

You can customize any of the builtin ADF Business Components error messages by providing an alternative message string for the error code in a custom message bundle.

Note: For your convenience, the `CustomizedErrorMessages` project in the `AdvancedExamples` workspace of the Fusion Order Demo lets you test the custom message bundle sample described in this section. Before you run the `ProductModule`, you will need to run the `addProductsTableConstraint.sql` script to set up the sample with the corresponding database constraint. When you run the `ProductModule` application module in the Oracle ADF Model Tester, you can blank out the **ProductId** attribute and click the **Validate** button. This will show the custom error message for the JBO error. Rollback the change and then enter `undefined` in the **ProductName** attribute. You can view the custom error message for the database error by clicking **Commit** (clicking **Validate** won't perform a commit to the database, so the database error is not generated in that case).

12.8.1 How to Customize Base ADF Business Components Error Messages

Assume you want to customize the builtin error message:

```
JBO-27014: Attribute Name is Product is required
```

If you have requested the Oracle Application Development Framework (Oracle ADF) source code from Oracle Worldwide Support, you can look in the `CSMessageBundle.java` file in the `oracle.jbo` package to see that this error message is related to the combination of the following lines in that message bundle file:

```
public class CSMessageBundle extends CheckedListResourceBundle {
  // etc.
  public static final String EXC_VAL_ATTR_MANDATORY            = "27014";
  // etc.
  private static final Object[][] sMessageStrings = {
    // etc.
    {EXC_VAL_ATTR_MANDATORY, "Attribute {2} in {1} is required"},
    // etc.
  }
}
```

The numbered tokens `{2}` and `{1}` are error message placeholders. In this example the `{1}` is replaced at runtime with the name of the entity object and the `{2}` with the name of the attribute.

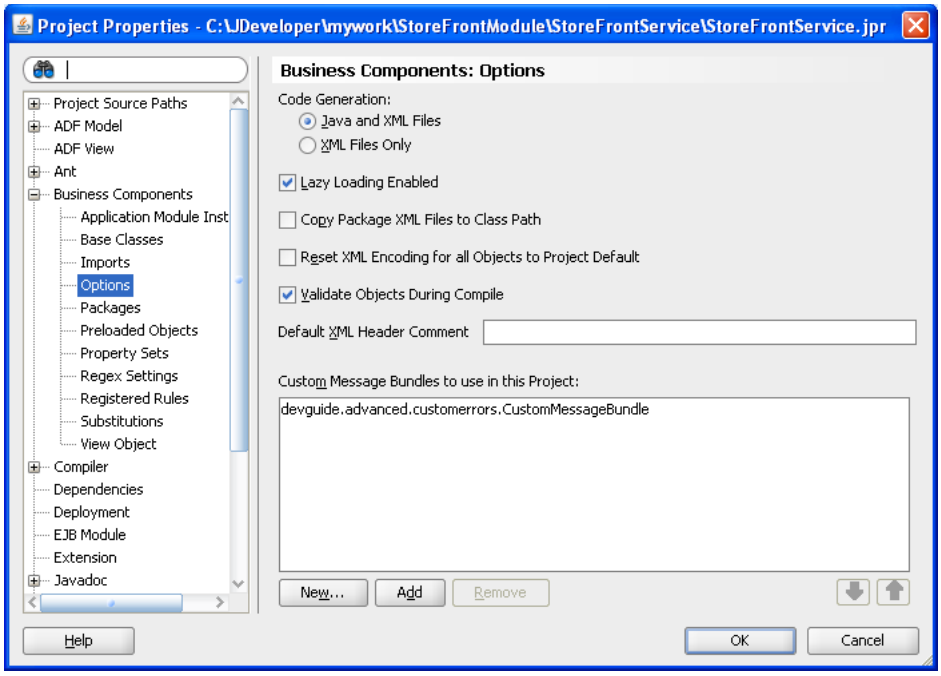
To create a custom message bundle file:

1. In the Application Navigator, right-click the data model project that you want to add the message bundle file to and choose **Project Properties**.
2. In the Project Properties dialog, select **Business Components > Options**.

   The **Custom Message Bundles to use in this Project** list displays at the bottom of the dialog.
3. Click **New**.
4. Enter a name and package for the custom message bundle in the **Create MessageBundle class** dialog and click **OK**.

Note: If the fully-qualified name of your custom message bundle file does not appear in the **Custom Message Bundles to use in this Project** list, click the **Remove** button, then click the **Add** button to add the new message bundle file created. When the custom message bundle file is correctly registered, its fully-qualified class name should appear in the list, as shown in Figure 12–8.

Figure 12–8 Project Properties Displays Message Resource Bundles



5. Click **OK** to dismiss the **Project Properties** dialog and open the new custom message bundle class in the source editor.

6. Edit the two-dimensional `String` array in the custom message bundle class to contain any customized messages you'd like to use.

   Example 12–13 illustrates a custom message bundle class that overrides the error message string for the `JBO-27014` error.

Example 12–13 Custom ADF Business Components Message Bundle

```
package devguide.advanced.customerrors;
import java.util.ListResourceBundle;
public class CustomMessageBundle extends ListResourceBundle {
  private static final Object[][] sMessageStrings
    = new String[][] {
       {"27014","You must provide a value for {2}"}
       };
  protected Object[][] getContents() {
    return sMessageStrings;
  }
}
```

12.8.2 What Happens When You Customize Base ADF Business Components Error Messages

After adding this message to your custom message bundle file, if you test the application using the Oracle ADF Model Tester and try to blank out the value of a mandatory attribute, you'll now see your custom error message instead of the default one:

```
JBO-27014: You must provide a value for Name
```

You can add as many messages to the message bundle as you want. Any message whose error code key matches one of the built-in error message codes will be used at runtime instead of the default one in the `oracle.jbo.CSMessageBundle` message bundle.

12.8.3 How to Display Customize Error Messages as Nested Exceptions

When you customize ADF Business Components error messages, you will also need to customize the display of nested error messages. To accomplish this, you must create and register a custom error handler class.

When your business method throws an error, the ADF binding layer intercepts the error and invokes the registered custom error handler class. In general, the error handler class is responsible for formatting the exception to be readable. During this process, the default error handler `DCErrorHandlerImpl` normally skips the top-level `JboException`, as this object is a wrapper over other business exceptions and does not have any business significance.

Although skipping the top-level exception is the desired behavior in the case of ADF Business Components errors, the default behavior will result in skipping the custom message you set for replacing the `SQLException`. To avoid this situation, While displaying each item in a nested exception, your custom error handler class must override `DCErrorHandlerImpl::skipException(Exception` *`ex`*`)` to decide whether to display the corresponding exception to the user in the final list or not.

Before you begin:

It may be helpful to have an understanding of application modules. For more information, see Section 12.10, "Substituting Extended Components in a Delivered Application."

You may also find it helpful to understand functionality that can be added using other Oracle ADF features. For more information, see Section 12.1.2, "Additional Functionality for Extending Business Components."

You will need to complete this task:

Create the error message in the resource bundle, as described in Section 12.8.1, "How to Customize Base ADF Business Components Error Messages."

To provide custom messages for SQLExceptions in your project:

1. Create an error handler class that extends the default error handler `DCErrorHandlerImpl` interface provided by the ADF binding layer.
2. In the error handler class, override the default error handler behavior for the `DCErrorHandlerImpl::skipException(Exception` *`ex`*`)` method, as shown in Example 12–14.

   This overridden method is necessary to display each item in a nested exception, such as the ones returned for database-level error messages. You must implement

logic to check for specifics exception types and, based on the business scenario, determine whether to display it in the list.

3. You can then register the custom error handler in your project's `DataBindings.cpx` file, as described in Section 16.3, "Customizing Error Handling."

Example 12–14 shows a custom implementation of the error handler that skips the `SQLIntegrityConstraintViolationException` from displaying in the error final list displayed to the user. You can choose to skip other database-level error message resulting from errors, such as unique constraint violations or foreign key constraint violations.

Example 12–14 DDL Statement Specifies Constraint Name

```
package view;

import java.sql.SQLIntegrityConstraintViolationException;

import oracle.adf.model.BindingContext;
import oracle.adf.model.RegionBinding;
import oracle.adf.model.binding.DCBindingContainer;
import oracle.adf.model.binding.DCErrorHandlerImpl;

import oracle.adf.model.binding.DCErrorMessage;

import oracle.jbo.DMLConstraintException;
import oracle.jbo.JboException;

public class CustomErrorHandler extends DCErrorHandlerImpl {

    public CustomErrorHandler() {
        super(false);
    }

    public CustomErrorHandler(boolean b) {
        super(b);
    }

    /**
     * If an exception is a RowValException or a TxnValException
     * and they have nested exceptions, then do not display
     * it.
     */
    @Override
    protected boolean skipException(Exception ex) {

        if (ex instanceof DMLConstraintException) {
            return false;
        } else if (ex instanceof SQLIntegrityConstraintViolationException) {
            return true;
        }
        return super.skipException(ex);
    }

)
```

12.8.4 How to Customize Error Messages for Database Constraint Violations

If you enforce constraints in the database, you might want to provide a custom error message in your Fusion web application to display to the end user when one of those constraints is violated. For example, assume a constraint called `NAME_CANNOT_BEGIN_WITH_U` gets added to the application's `PRODUCTS_BASE` table using the following DDL statement shown in Example 12–15.

Example 12–15 DDL Statement Specifies Constraint Name

```
alter table products_base add (
  constraint name_cannot_begin_with_u
      check (upper(substr(product_name,1,1)) != 'U')
);
```

To define a custom error message in your application, you add a message to a custom message bundle with the constraint name as the message key. Example 12–16 shows the `CustomMessageBundle.java` class when it defines a message with the key `NAME_CANNOT_BEGIN_WITH_U` which matches the name of the database constraint name defined in Example 12–15.

Example 12–16 Customizing Error Message for Database Constraint Violation

```
package devguide.advanced.customerrors;
import java.util.ListResourceBundle;
public class CustomMessageBundle extends ListResourceBundle {
  private static final Object[][] sMessageStrings
    = new String[][] {
       {"NAME_CANNOT_BEGIN_WITH_U",
        "The name cannot begin with the letter u!"}
     };
  protected Object[][] getContents() {
    return sMessageStrings;
  }
}
```

12.8.5 How to Implement a Custom Constraint Error Handling Routine

If the default facility for assigning a custom message to a database constraint violation does not meet your needs, you can implement your own custom constraint error handling routine. Doing this requires creating a custom framework extension class for the ADF transaction class, which you then configure your application module to use at runtime.

12.8.5.1 Creating a Custom Database Transaction Framework Extension Class

To write a custom framework extension class for the ADF transaction, create a class like the `CustomDBTransactionImpl` shown in Example 12–17. This example overrides the transaction object's `postChanges()` method to wrap the call to `super.postChanges()` with a `try/catch` block in order to perform custom processing on any `DMLConstraintException` errors that might be thrown. In this simple example, the only custom processing being performed is a call to `ex.setExceptions(null)` to clear out any nested detail exceptions that the `DMLConstraintException` might have. Instead of this, you could perform any other kind of custom exception processing required by your application, including throwing a *custom* exception, provided your custom exception extends `JboException` directly or indirectly.

Example 12–17 Custom Database Transaction Framework Extension Class

```
package devguide.advanced.customerrors;
import oracle.jbo.DMLConstraintException;
import oracle.jbo.server.DBTransactionImpl2;
import oracle.jbo.server.TransactionEvent;
public class CustomDBTransactionImpl  extends DBTransactionImpl2 {
  public void postChanges(TransactionEvent te) {
    try {
      super.postChanges(te);
    }
    /*
     * Catch the DML constraint exception
     * and perform custom error handling here
     */
    catch (DMLConstraintException ex) {
      ex.setExceptions(null);
      throw ex;
    }
  }
}
```

12.8.5.2 Configuring an Application Module to Use a Custom Database Transaction Class

In order for your application module to use a custom database transaction class at runtime, you must:

1. Provide a custom implementation of the `DatabaseTransactionFactory` class that overrides the `create()` method to return an instance of the customized transaction class.
2. Configure the value of the `TransactionFactory` property to be the fully-qualified name of this custom transaction factory class.

Example 12–18 shows a custom database transaction factory class that does this. It returns a new instance of the `CustomDBTransactionImpl` class when the framework calls the `create()` method on the database transaction factory.

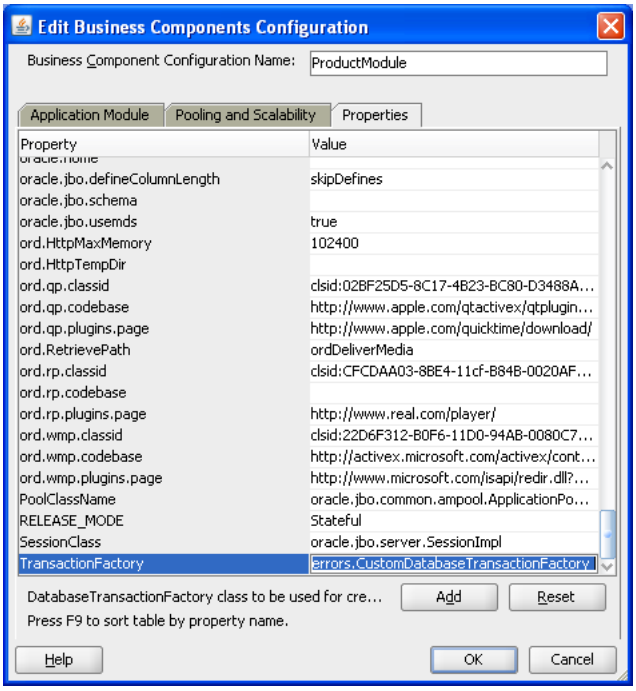
Example 12–18 Custom Database Transaction Factory Class

```
package devguide.advanced.customerrors;
import oracle.jbo.server.DBTransactionImpl2;
import oracle.jbo.server.DatabaseTransactionFactory;
public class CustomDatabaseTransactionFactory
       extends DatabaseTransactionFactory {
  public CustomDatabaseTransactionFactory() {
  }
  /**
   * Return an instance of our custom ToyStoreDBTransactionImpl class
   * instead of the default implementation.
   *
   * @return instance of custom CustomDBTransactionImpl implementation.
   */
  public DBTransactionImpl2 create() {
    return new CustomDBTransactionImpl();
  }
}
```

To complete the job, use the Properties tab of the Create Business Components Configuration dialog to assign the value

devguide.advanced.customerrors.CustomDatabaseTransactionFactory to the TransactionFactory property, as shown in Figure 12–9. You can open the Create Business Components Configuration dialog from the Configuration page of the overview editor for the application module by clicking the **Create new configuration objects** button. When you run the application using this configuration, your custom transaction class will be used.

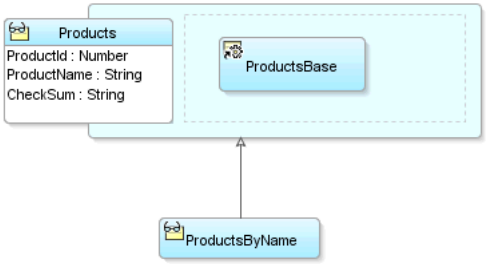
Figure 12–9 ADF Business Components Can Use Custom Database Transaction Class



12.9 Creating Extended Components Using Inheritance

Whenever you create a new business component, if necessary, you can extend an existing one to create a customized version of the original. As shown in Figure 12–10, the ProductsByName view object extends the Products view object to add a named bind variable named TheStatus and to customize the WHERE clause to reference that bind variable.

Figure 12–10 ADF Business Components Can Extend Another Component



While the figure shows a view object example, this component inheritance facility is available for all component types. When one component extends another, the extended component inherits all of the metadata and behavior from the parent it extends. In the

extended component, you can add new features or customize existing features of its parent component both through metadata and Java code.

Note: The example in this section refers to the `BaseProject` project of the `AdvancedExamples` workspace in the `StandaloneExamples` module of the Fusion Order Demo application.

12.9.1 How To Create a Component That Extends Another

To create an extended component, use the component wizard in the **New Gallery** for the type of component you want to create. For example, to create an extended view object, you use the Create View Object wizard. On the **Name** page of the wizard — in addition to specifying a name and a package for the new component — provide the fully-qualified name of the component that you want to extend in the **Extends** field. To pick the component name from a list, use the **Browse** button next to the **Extends** field. Then, continue to create the extended component in the normal way using the remaining panels of the wizard.

12.9.2 How To Extend a Component After Creation

After you define an extended component, JDeveloper lets you change the parent component from which an extended component inherits. You can use the overview editor for the component to accomplish this.

To change the parent component after creation:

1. Double-click the component.
2. In the overview editor for the component, click the **General** navigation tab and click the **Refactor object extends** button next to the **Extends** field.
3. In the Select Parent dialog, choose the desired component to extend from the package list.

To change the extended component to not inherit from any parent, select the **None** checkbox in the Select Parent dialog. This has the same effect as if you deleted the component and recreated to accomplish this.

12.9.3 What Happens When You Create a Component That Extends Another

ADF business components you create are comprised of an XML component definition and an optional Java class. When you create a component that extends another, JDeveloper reflects this component inheritance in both the XML component definition and in any generated Java code for the extended component.

12.9.3.1 Understanding an Extended Component's XML Descriptor

JDeveloper notes the name of the parent component in the new component's XML component definition by adding an `Extends` attribute to the root component element. Any new declarative features you add or any aspects of the parent component's definition you've overridden appear in the extended component's XML component definition. In contrast, metadata that is purely inherited from the parent component is not repeated for the extended component.

Example 12–19 shows what the `ProductsByName.xml` XML component definition for the `ProductsByName` view object looks like. Notice the `Extends` attribute on the <ViewObject> element, the <Variable> element related to the additional bind variable added in the extended view object, and the overridden value of the `Where` attribute

for the `WHERE` clause that was modified to reference the `theProductName` bind variable.

Example 12–19 Extended Component Reflects Parent in Its XML Descriptor

```
<ViewObject
  xmlns="http://xmlns.oracle.com/bc4j"
  Name="ProductsByName"
  Extends="devguide.advanced.baseproject.extsub.Products"
  Where="UPPER(PRODUCT_NAME) LIKE UPPER(:theProductName)||'%'"
  BindingStyle="OracleName"
  CustomQuery="false"
  RowClass="devguide.advanced.baseproject.extsub.ProductsByNameRowImpl"
  ComponentClass="devguide.advanced.baseproject.extsub.ProductsByNameImpl"
...
  <Variable
     Name="theProductName"
     Kind="where"
     Type="java.lang.String"/>
...
</ViewObject>
```

12.9.3.2 Understanding Java Code Generation for an Extended Component

If you enable custom Java code for an extended component, JDeveloper automatically generates the Java classes to extend the respective Java classes of its parent component. In this way, the extended component can override any aspect of the parent component's programmatic behavior as necessary. If the parent component is an XML-only component with no custom Java class of its own, the extended component's Java class extends whatever base Java class the parent would use at runtime. This could be the default ADF Business Components framework class in the `oracle.jbo.server` package, or could be your own framework extension class if you have specified that in the **Extends** dialog of the parent component.

In addition, if the extended component is an application module or view object and you enable client interfaces on it, JDeveloper automatically generates the extended component's client interfaces to extend the respective client interfaces of the parent component. If the respective client interface of the parent component does not exist, then the extended component's client interface directly extends the appropriate base ADF Business Components interface in the `oracle.jbo` package.

12.9.4 What You May Need to Know

12.9.4.1 You Can Use Parent Classes and Interfaces to Work with Extended Components

Since an extended component is a customized version of its parent, code you write that works with the *parent* component's Java classes or its client interfaces works without incident for either the parent component *or* any customized version of that parent component.

For example, assume you have a base `Products` view object with custom Java classes and client interfaces like:

- class `ProductsImpl`
- row class `ProductsRowImpl`

- interface `Products`
- row interface `ProductsRow`

If you create a `ProductsByName` view object that extends `Products`, then you can use the base component's classes and interface to work both with `Products` and `ProductsByName`.

Example 12–20 illustrates a test client program that works with the `Products`, `ProductsRow`, `ProductsByName`, and `ProductsByNameRow` client interfaces. A few interesting things to note about the example are the following:

1. You can use parent `Products` interface for working with the `ProductsByName` view object that extends it.
2. Alternatively, you can cast an instance of the `ProductsByName` view object to its own more specific `ProductsByName` client interface.
3. You can test if row `ProductsRow` is actually an instance of the more specific `ProductsByNameRow` before casting it and invoking a method specific to the `ProductsByNameRow` interface.

Example 12–20 Working with Parent and Extended Components

```
package devguide.advanced.baseproject.extsub;
/* imports omitted */
public class TestClient {
  public static void main(String[] args) {
    String        amDef = "devguide.advanced.baseproject.extsub.ProductModule";
    String        config = "ProductModuleLocal";
    ApplicationModule am =
    Configuration.createRootApplicationModule(amDef,config);
    Products products = (Products)am.findViewObject("Products");
    products.executeQuery();
    ProductsRow product = (ProductsRow)products.first();
    printAllAttributes(products,product);
    testSomethingOnProductsRow(product);
    // 1. You can use parent Products interface for ProductsByName
    products = (Products)am.findViewObject("ProductsById");
    // 2. Or cast it to its more specific ProductsByName interface
    ProductsByName productsById = (ProductsByName)products;
    productsById.setProductName("Ice");
    productsById.executeQuery();
    product = (ProductsRow)productsById.first();
    printAllAttributes(productsById,product);
    testSomethingOnProductsRow(product);
    am.getTransaction().rollback();
    Configuration.releaseRootApplicationModule(am,true);
  }
  private static void testSomethingOnProductsRow(ProductsRow product) {
    try {
      // 3. Test if row is a ProductsByNameRow before casting
      if (product instanceof ProductsByNameRow) {
        ProductsByNameRow productByName = (ProductsByNameRow)product;
        productByName.someExtraFeature("Test");
      }
      product.setName("Q");
      System.out.println("Setting the Name attribute to 'Q' succeeded.");
    }
    catch (ValidationException v) {
      System.out.println(v.getLocalizedMessage());
    }
```

```
  }
  private static void printAllAttributes(ViewObject vo, Row r) {
    String viewObjName = vo.getName();
    System.out.println("Printing attribute for a row in VO '"+
                       viewObjName+"'");
    StructureDef def = r.getStructureDef();
    StringBuilder sb = new StringBuilder();
    int numAttrs = def.getAttributeCount();
    AttributeDef[] attrDefs = def.getAttributeDefs();
    for (int z = 0; z < numAttrs; z++) {
      Object value = r.getAttribute(z);
      sb.append(z > 0 ? "  " : "")
        .append(attrDefs[z].getName())
        .append("=")
        .append(value == null ? "<null>" : value)
        .append(z < numAttrs - 1 ? "\n" : "");
    }
    System.out.println(sb.toString());
  }
}
```

Running the test client produces the results shown in Example 12–21.

Example 12–21 Results of Running TestClient.java

```
Printing attribute for a row in VO 'Products'
ProdId=100
  Name=Washing Machine W001
  Checksum=I am the Product Class
Setting the Name attribute to 'Q' succeeded.
Printing attribute for a row in VO 'ProductsById'
ProdId=119
  Name=Ice Maker I012
  Checksum=I am the Product Class
  SomeExtraAttr=SomeExtraAttrValue
## Called someExtraFeature of ProductsByNameRowImpl
Setting the Name attribute to 'Q' succeeded.
```

> **Note:** In this example, Products is an entity-based view object based on the `Product` entity object. The `Product` entity object includes a transient `Checksum` attribute that returns the string "I am the Product class". You'll learn more about why this was included in the example in Section 12.10, "Substituting Extended Components in a Delivered Application."

12.9.4.2 Class Extends is Disabled for Extended Components

When you create an extended component, the **Class Extends** button on the Java page of the extended component's wizard is disabled. Additionally, in the application module editor's Java page, when you click **Edit java options**, the **Class Extends** button in the Java dialog appears disabled. This is due to the fact that JDeveloper automatically extends the appropriate class of its parent component, so it does not make sense to allow you to select a different class.

12.9.4.3 Interesting Aspects You Can Extend for Key Component Types

Entity Objects
When you create an extended entity object, you can introduce new attributes, new associations, new validators, and new custom code. You can override certain declarative aspects of existing attributes as well as overriding any method from the parent component's class.

View Objects
When you create an extended view object, you can introduce new attributes, new view links, new bind variables, and new custom code. You can override certain declarative aspects of existing attributes as well as overriding any method from the parent component's class.

Application Modules
When you create an extended application module, you can introduce new view object instances or new nested application module instance and new custom code. You can also override any method from the parent component's class.

12.9.4.4 Extended Components Have Attribute Indices Relative to Parent

If you add new attributes in an extended entity object or view object, the attribute index numbers are computed relative to the parent component. For example, consider the `Products` view object mentioned in Section 12.9.4.1, "You Can Use Parent Classes and Interfaces to Work with Extended Components." If you enable a custom view row class, it might have attribute index constants defined in the `ProductsRowImpl.java` class like this:

```
public class ProductsRowImpl extends ViewRowImpl
                             implements ProductsRow {
  public static final int PRODID = 0;
  public static final int NAME = 1;
  public static final int CHECKSUM = 2;
  //etc.
}
```

When you create an extended view object like `ProductsByName`, if that view object adds an addition attribute like `SomeExtraAttr` and has a custom view row class enabled, then its attribute constants will be computed relative to the maximum value of the attribute constants in the parent component:

```
public class ProductsByNameRowImpl extends ProductsRowImpl
                                   implements ProductsByNameRow {
  public static final int MAXATTRCONST =
    ViewDefImpl.getMaxAttrConst("devguide.advanced.baseproject.extsub.Products");
  public static final int SOMEEXTRAATTR = MAXATTRCONST;
```

Additional attributes would have index values of `MAXATTRCONST+1`, `MAXATTRCONST+2`, etc.

12.10 Substituting Extended Components in a Delivered Application

If you deliver packaged applications that can require on-site customization for each potential client of your solution, ADF Business Components offers a useful feature to simplify that task.

> **Note:** The example in this section refers to the `BaseProject` project and the `ExtendAndSubstitute` project of the `AdvancedExamples` workspace in the `StandaloneExamples` module of the Fusion Order Demo application.

All too often, on-site application customization is performed by making direct changes to the source code of the delivered application. This approach demonstrates its weaknesses whenever you deliver patches or new feature releases of your original application to your clients. Any customizations they had been applied to the base application's source code need to be painstakingly re-applied to the patched or updated version of the base application. Not only does this render the application customization a costly, ongoing maintenance expense, it can introduce subtle bugs due to human errors that occur when reapplying previous customizations to new releases.

ADF Business Components offers a superior, component-based approach to support application customization that doesn't require changing — or even having access to — the base application's source code. To customize your delivered application, your customers can:

1. Import one or more packages of components from the base application into a new project.
2. Create new components to effect the application customization, extending appropriate parent components from the base application as necessary.
3. Define a list of global component substitutions, naming their customized components to substitute for your base application's appropriate parent components.

When the customer runs your delivered application with a global component substitution list defined, their customized application components are used by your delivered application without changing any of its code. When you deliver a patched or updated version of the original application, their component customizations apply to the updated version the next time they restart the application without needing to re-apply any customizations.

12.10.1 How To Substitute an Extended Component

To define global component substitutions, use the Project Properties dialog in the project where you have created extended components based on the imported components from the base application.

> **Note:** You can only substitute a component in the base application with an extended component that inherits directly or indirectly from the base one.

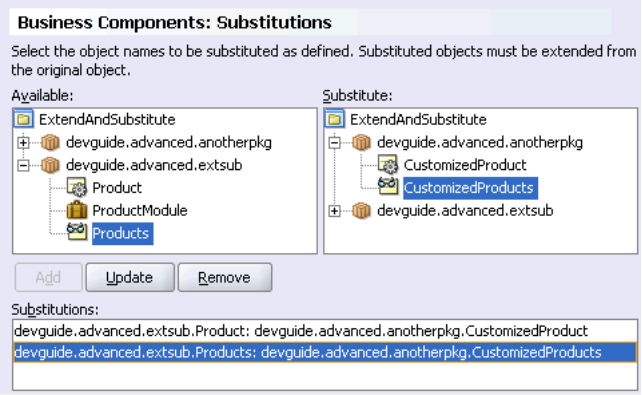
To substitute an extended component:

1. In the Application Navigator, right-click the data model project that you want to add the extended component to and choose **Project Properties**.
2. In the Project Properties dialog, select **Business Components > Substitutions**.
3. In the Substitutions page, select the base application's component in the **Available** list.
4. Select the customized, extended component to substitute in the **Substitute** list.

5. Click **Add**.

   For example, assume that you have created the view object `CustomizedProducts` in a package that extends the base view object `Products`. To substitute the `CustomizedProducts` view object for the legacy `Products` view object, you would select these view objects as shown in Figure 12–11 to define the component substitution.

Figure 12–11 Defining Business Components Substitutions



12.10.2 What Happens When You Substitute

When you define a list of global component substitutions in a project named *`YourExtendsAndSubstitutesProject`*, the substitution list is saved in the *`YourExtendsAndSubstitutesProject`*`.jpx` in the root directory of the source path.

The file will contain <Substitute> elements as shown in Example 12–22, one for each component to be substituted.

Example 12–22 Component Substitution List Saved in the Project's JPX File

```
<JboProject
   Name="ExtendAndSubstitute"
   SeparateXMLFiles="true"
   PackageName="" >
   <Containee
      Name="anotherpkg"
      FullName="devguide.advanced.anotherpkg.anotherpkg"
      ObjectType="JboPackage" >
   </Containee>
   <Containee
      Name="extsub"
      FullName="devguide.advanced.extsub"
      ObjectType="JboPackage" >
      <DesignTime>
         <Attr Name="_LocationURL"
               Value="../../BaseProject/deploy/BaseProjectCSMT.jar" />
      </DesignTime>
   </Containee>
   <Substitutes>
      <Substitute OldName="devguide.advanced.extsub.Product"
                  NewName="devguide.advanced.anotherpkg.CustomizedProduct" />
      <Substitute OldName="devguide.advanced.extsub.Products"
                  NewName="devguide.advanced.anotherpkg.CustomizedProducts" />
```

```
   </Substitutes>
</JboProject>
```

12.10.3 How to Enable the Substituted Components in the Base Application

To have the original application use the set of substituted components, define the Java system property `Factory-Substitution-List` and set its value to the name of the project whose `*.jpx` file contains the substitution list. The value should be just the project name without any `*.jpr` or `*.jpx` extension.

Consider a simple example that customizes the `Product` entity object and the `Products` view object described in Section 12.9.4.1, "You Can Use Parent Classes and Interfaces to Work with Extended Components." To perform the customization, assume you create new project named `ExtendsAndSubstitutes` that:

- Defines a library for the JAR file containing the base components
- Imports the package containing `Product` and `Products`
- Creates new extended components in a distinct package name called `CustomizedProduct` and `CustomizedProducts`
- Defines a component substitution list to use the extended components.

When creating the extended components, assume that you:

- Added an extra view attribute named `ExtraViewAttribute` to the CustomizedProducts view object.
- Added a new validation rule to the `CustomizedProduct` entity object to enforce that the product name cannot be the letter "`Q`".
- Overrode the `getChecksum()` method in the `CustomizedProduct.java` class to return "**I am the CustomizedProduct Class**".

If you define the `Factory-Substitution-List` Java system property set to the value `ExtendsAndSubstitutes`, then when you run the exact same test client class shown in Example 12–20 the output of the sample will change to reflect the use of the substituted components as shown in Example 12–23.

Example 12–23 Results of Running TestClient.java with System Property Set

```
Printing attribute for a row in VO 'Products'
ProdId=100
  Name=Washing Machine W001
  Checksum=I am the CustomizedProduct Class
  ExtraViewAttribute=Extra Attr Value
The name cannot be Q!
Printing attribute for a row in VO 'ProductsById'
ProdId=119
  Name=Ice Maker I012
  Checksum=I am the CustomizedProduct Class
  SomeExtraAttr=SomeExtraAttrValue
## Called someExtraFeature of ProductsByNameRowImpl
The name cannot be Q!
```

Compared to the output from Example 12–20, notice that in the presence of the factory substitution list, the `Products` view object in the original test program now has the additional `ExtraViewAttribute`, now reports a `Checksum` attribute value of "I am the CustomizedProduct Class", and now disallows the assignment of the product name to have the value "Q". These component behavior changes were performed

without needing to modify the original Java or XML source code of the delivered components.

Part III

Using the ADF Model Layer

Part III contains the following chapters:

- Chapter 13, "Using ADF Model in a Fusion Web Application"
- Chapter 14, "Exposing Web Services Using the ADF Model Layer"
- Chapter 15, "Exposing URL Services Using the ADF Model Layer"
- Chapter 16, "Using Validation in the ADF Model Layer"
- Chapter 17, "Designing a Page Using Placeholder Data Controls"

13

Using ADF Model in a Fusion Web Application

This chapter describes how an application module's data model and business service interface methods appear at design time for drag and drop data binding, how they are accessible at runtime by the ADF Model data binding layer using the application module data control, and how developers can use the Data Controls panel to create databound pages.

This chapter includes the following sections:

- Section 13.1, "About ADF Data Binding"
- Section 13.2, "Additional Functionality"
- Section 13.3, "Exposing Application Modules with Oracle ADF Data Controls"
- Section 13.4, "Using the Data Controls Panel"
- Section 13.5, "Working with the DataBindings.cpx File"
- Section 13.6, "Configuring the ADF Binding Filter"
- Section 13.7, "Working with Page Definition Files"
- Section 13.8, "Creating ADF Data Binding EL Expressions"
- Section 13.9, "Using Simple UI First Development"

13.1 About ADF Data Binding

ADF Model implements concepts that enable decoupling the user interface technology from the business service implementation: *data controls* and *declarative bindings*.

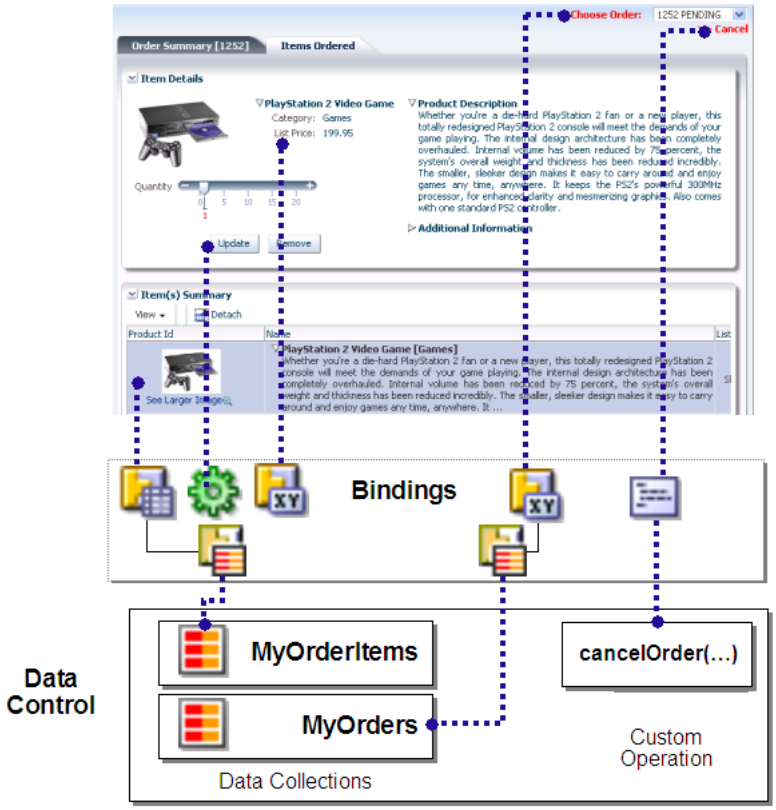
Data controls abstract the implementation technology of a business service by using standard metadata interfaces to describe the service's operations and data collections, including information about the properties, methods, and types involved. In an application that uses business components, a data control is automatically created when you create an application module, and it contains all the functionality of the application module. Developers can then use the representation of the data control displayed in JDeveloper's Data Controls panel to create UI components that are automatically bound to the application module. At runtime, the ADF Model layer reads the information describing the data controls and bindings from the appropriate XML files and then implements the two-way connection between the user interface and the business service.

Declarative bindings abstract the details of accessing data from data collections in a data control and of invoking its operations. There are three basic kinds of declarative binding objects:

- Executable bindings: Included in executable bindings are iterator bindings, which simplify the building of user interfaces that allow scrolling and paging through collections of data and drilling-down from summary to detail information. Executable bindings also include bindings that allow searching and nesting a series of pages within another page, as well as bindings that cause operations to occur immediately.
- Value bindings: Used by UI components that display data. Value bindings range from the most basic variety that work with a simple text field to more sophisticated list and tree bindings that support the additional needs of list, table, and tree UI controls.
- Action bindings: Used by UI components like hyperlinks or buttons to invoke built-in or custom operations on data collections or a data control without writing code.

Figure 13–1 shows how bindings connect UI components to data control collections and methods.

Figure 13–1 Bindings Connect UI Components to Data Controls



The group of bindings supporting the UI components on a page are described in a page-specific XML file called the *page definition file*. The ADF Model layer uses this file at runtime to instantiate the page's bindings. These bindings are held in a request-scoped map called the *binding container*, accessible during each page request

using the EL expression `#{bindings}`. This expression always evaluates to the binding container for the current page.

> **Tip:** The current binding container is also available from `AdfContext` for programmatic access.

You can design a databound user interface by dragging an item from the Data Controls panel and dropping it on a page as a specific UI component. When you use data controls to create a UI component, JDeveloper automatically creates the various code and objects needed to bind the component to the data control you selected.

> **Note:** Using the ADF Model layer to perform business service access ensures that the view and the business service stay in sync. For example, while you could.call a method on an application module by class-casting the data control reference to the application module instance and then calling the method directly, doing so would bypass the model layer and it would then become unaware of any changes.

13.2 Additional Functionality

You may find it helpful to understand other ADF features before you configure or use the ADF Model layer. Additionally, you may want to read about what you can do with your model layer configurations. Following are links to other functionality that may be of interest.

- ADF model and data binding rely on the business layer. In most cases, you will be working with representations of your view objects. For more information, see Chapter 5, "Defining SQL Queries Using View Objects." You may also want to be familiar with advanced functionality discussed in Chapter 42, "Advanced View Object Techniques."
- You can use the ADF Model layer to create your view pages. After reading this chapter for the basic information on how data binding in the view layer works, you should refer to the chapters in Part V, "Creating a Databound Web User Interface" for information about using data binding to create specific view functionality.
- For information about how ADF Model works with the page lifecycle, see Chapter 25, "Understanding the Fusion Page Lifecycle."
- For a complete list of configuration parameters that affect the model layer, see Section A.7, "DataBindings.cpx" and Section A.8, "pageNamePageDef.xml."
- For a complete list of binding properties, see Appendix B, "Oracle ADF Binding Properties."

13.3 Exposing Application Modules with Oracle ADF Data Controls

The application module data control is a thin adapter over an application module pool that automatically acquires an available application module instance at the beginning of the request. During the current request, the application module data control holds a reference to the application module instance on behalf of the current user session. At the end of the request, the data control releases the instance back to the pool. Importantly, the application module component directly implements the interfaces that the binding objects expect for data collections, built-in operations, and service

methods. This optimized interaction allows the bindings to work *directly* with the application module instances in its data model in the following ways:

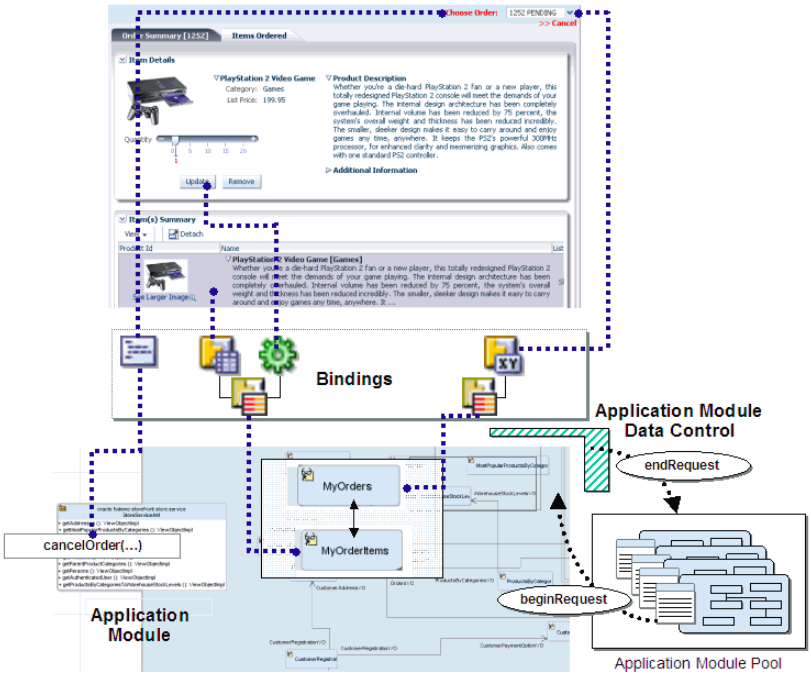
- Iterator bindings directly bind to the default row set iterator of the default row set of any view object instance. The row set iterator manages the current object and current range information.

  Tip: You can also use the iterator binding to bind to a secondary named row set that you have created. To bind to a secondary row set iterator, you need to use the `RSIName`. For more information about the difference between the default row set and secondary row sets and how to create them, see Section 42.1.10, "Working with Multiple Row Sets and Row Set Iterators."

- Action bindings directly bind to either:
  - Custom methods on the data control client interface
  - Built-in operations of the application module and view objects

Figure 13–2 illustrates the pool management role the application module data control plays and highlights the direct link between the bindings and the application module instance.

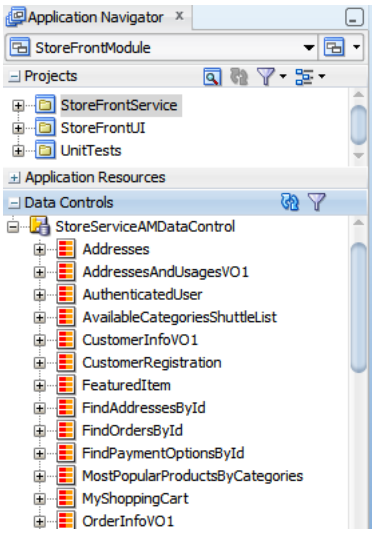
Figure 13–2 Bindings Connect Directly to View Objects and Methods of an Application Module from a Pool



13.3.1 How an Application Module Data Control Appears in the Data Controls Panel

You use the Data Controls panel to create databound UI components by dragging and dropping icons from the panel onto the visual editor for a page. Figure 13–3 shows the Data Controls panel displaying the data controls for the StoreFront module.

Figure 13–3 Data Controls Panel in JDeveloper



The Data Controls panel lists all the data controls that have been created for the application's business services and exposes all the collections (row sets of data objects), methods, and built-in operations that are available for binding to UI components.

Note: If you've configured JDeveloper to expose them, any view link accessor returns are also displayed. For more information, see Section 5.6, "Working with Multiple Tables in a Master-Detail Hierarchy." To view the accessor methods:

1. From the JDeveloper main menu, choose **Tools > Preferences**.
2. Select the **Data Controls Panel** node.
3. Select **Show Underlying Accessor Nodes** to activate the checkbox.

For example, in an application that uses ADF Business Components to define the business services, each data control on the Data Controls panel represents a specific application module, and exposes the view object instances in that application's data model. The hierarchy of objects in the data control is defined by the view links between view objects that have specifically been added to the application module data model. For information about creating view objects and view links, see Chapter 5, "Defining SQL Queries Using View Objects." For information about adding view links to the data model, see Section 5.6.4, "How to Enable Active Master-Detail Coordination in the Data Model."

Tip: You can open the overview editor for a view object by right-clicking the associated data control object and choosing **Edit Definition**.

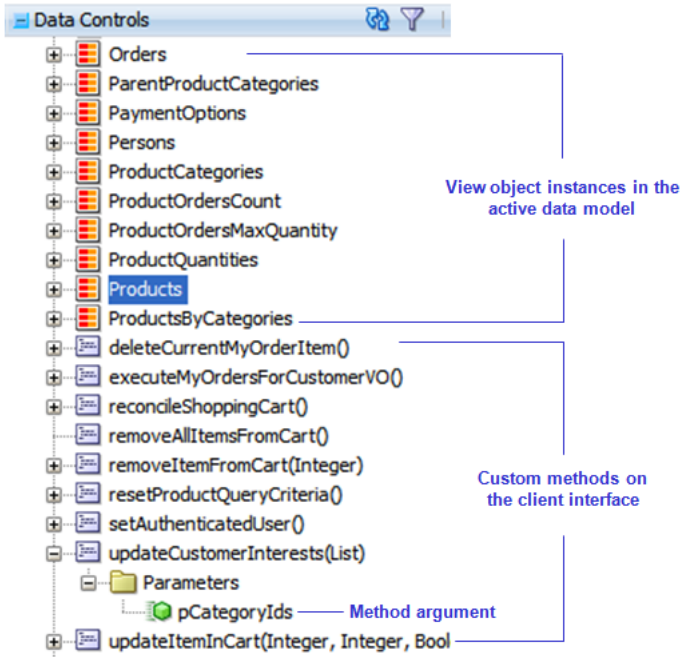
For example, the `StoreServiceAMDataControl` application module implements the business service layer of the StoreFront module application. Its data model contains numerous view object instances, including several master-detail hierarchies. The view layer of the ADF sample application consists of JSF pages whose UI components are bound to data from the view object instances in the `StoreServiceAMDataControl`'s data model, and to built-in operations and service methods on its client interface.

13.3.1.1 How the Data Model and Service Methods Appear in the Data Controls Panel

Each view object instance appears as a named data collection whose name matches the view object instance name. Figure 13–4 illustrates how the Data Controls panel displays the view object instances in the `StoreServiceAMDataControl`'s data model (note that for viewing simplicity, the figure omits some details in the tree that appear for each view object). The Data Controls panel reflects the master-detail hierarchies in your application module data model by displaying detail data collections nested under their master data collection.

The Data Controls panel also displays each custom method on the application module's client interface as a named data control custom operation whose name matches the method name. If a method accepts arguments, they appear in a Parameters node as operation parameters nested inside the operation's node.

Figure 13–4 How the Data Model Appears in the Data Controls Panel

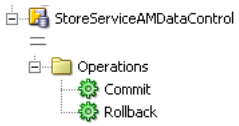


13.3.1.2 How Transaction Control Operations Appear in the Data Controls Panel

The application module data control exposes two data control built-in operations named `Commit` and `Rollback`, as shown in Figure 13–5 (note that the Operations node in the data controls tree omits all of the data collections and custom operations for a more streamlined view). At runtime, when these operations are invoked by the data binding layer, they delegate to the `commit()` and `rollback()` methods of the `Transaction` object associated with the current application module instance.

Note: In an application module with many view object instances and custom methods, you may need to scroll the Data Controls panel display to find the Operations node that is the direct child node of the data control. This node is the one that contains these built-in operations.

Figure 13–5 How Transaction Control Operations Appear in the Data Controls panel

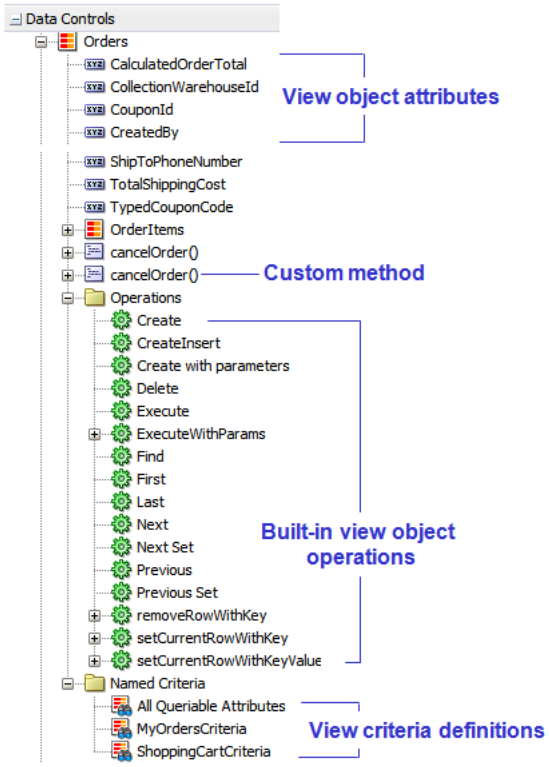


13.3.1.3 How View Objects Appear in the Data Controls Panel

The view object attributes are displayed as immediate child nodes of the corresponding data collection, as are any custom methods you've created. Figure 13–6 shows how each view object instance in the application module's data model appears in the Data Controls panel. If you have selected any custom methods to appear on the view object's client interface, they appear as custom methods immediately following the view object attributes at the same level. If the method accepts arguments, these appear in a nested Parameters node as operation parameters.

By default, implicit view criteria are created for each attribute that is able to be queried on a view object. They appear as the All Queriable Attributes node under the Named Criteria node, as shown in Figure 13–6. If any named view criteria were created for the view object, they appear under the Named Criteria node. The View Criteria expressions (both implicit and named) appear as method returns. The conjunction used in the query, along with the criteria items and if applicable, any nested criteria, are shown as children. These items are used to create quick search forms, as detailed in Chapter 31, "Creating ADF Databound Search Forms."

Figure 13–6 How View Objects Appear in the Data Controls Panel



As shown in Figure 13–6, the Operations node under the data collection displays all its available built-in operations. If an operation accepts one or more parameters, then

those parameters appear in a nested Parameters node. At runtime, when one of these data collection operations is invoked by name by the data binding layer, the application module data control delegates the call to an appropriate method on the `ViewObject` interface to handle the built-in functionality. The built-in operations fall into three categories: operations that affect the current row, operations that refresh the data collection, and all other operations.

Operations that affect the current row:

- `Create`: Creates a new row that becomes the current row, but does not insert it.
- `CreateInsert`: Creates a new row that becomes the current row, and inserts the new blank row into the data source.
- `Create with Parameters`: Creates a new row taking parameter values. The passed parameters can supply the create-time value for the following:
  - A discriminator for a polymorphic view object
  - A composing parent's foreign key attribute needed for the creation of a polymorphic view object
  - A composed child view object row when it is not created in the context of a parent row

  For more information about polymorphic view objects, see Section 42.6, "Using View Objects to Work with Multiple Row Types."
- `Delete`: Deletes the current row.
- `First`: Sets the current row to be the first row in the row set.
- `Last`: Sets the current row to be the last row in the row set.
- `Next`: Sets the row to be the next row in the row set.
- `Next Set`: Navigates backward one full set of rows.
- `Previous`: Sets the current row to be the previous row in the row set.
- `Previous Set`: Navigates forward one full set of rows.
- `setCurrentRowWithKey`: Tries to finds a row using the serialized string representation of row key passed as a parameter. If found, that row becomes the current row.
- `setCurrentRowWithKeyValue`: Tries to finds a row using the primary key attribute value passed as a parameter. If found, that row becomes the current row.

Operations that refresh the data collection:

- `Execute`: Refreshes the data collection by executing or reexecuting the view object's query, leaving any bind parameters at their current values.
- `ExecuteWithParams`: Refreshes the data collection by first assigning new values to the named bind variables passed as parameters, then executing or reexecuting the view object's query.

> **Note:** The `executeWithParams` operation appears only for view objects that have defined one or more named bind variables at design time.

All other operations:

- `removeRowWithKey`: Tries to finds a row using the serialized string representation of row key passed as a parameter. If found, the row is removed.
- `Find`: Toggles "Find Mode" on and off for the data collection.

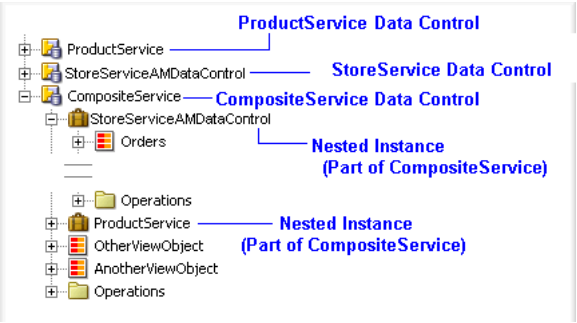
13.3.1.4 How Nested Application Modules Appear in the Data Controls Panel

If you build composite application modules by including nested instances of other application modules, the Data Controls panel reflects this component assembly in the tree hierarchy. For example, assume that, in addition to the `StoreServiceAMDataControl` application module, you have also created the following application modules in the same package:

- An application module named `ProductService`, and renamed its data control to `ProductService`
- An application module named `CompositeService`, and renamed its data control to `CompositeService`

Then assume that you've added two view object instances named `OtherViewObject` and `AnotherViewObject` to the data model of `CompositeService` and that on the Application Modules page of the Edit Application Module dialog you have added an instance of the `StoreServiceAMDataControl` application module and an instance of the `ProductService` application module to reuse them as part of `CompositeService`. Figure 13–7 illustrates how your `CompositeService` would appear in the Data Controls panel (note that much of the structure of the nested `StoreServiceAMDataControl` has been omitted for clarity). The nested instances of `StoreServiceAMDataControl` and `ProductService` appear in the panel tree display nested inside of the `CompositeService` data control. The entire data model and set of client methods that the nested application module instances expose to clients are automatically available as part of the `CompositeService` that reuses them.

Figure 13–7 How Nested Application Modules Appear in the Data Controls Panel



One possibly confusing point is that even though you have reused nested instances of `StoreServiceAMDataControl` and `ProductService` inside of `CompositeService`, the `StoreServiceAMDataControl` and `ProductService` application modules also appear themselves as top-level data control nodes in the panel tree. JDeveloper assumes that you might want to sometimes use `StoreServiceAMDataControl` or `ProductService` on their own as separate data controls from `CompositeService`, so it displays all three of them. You need to be careful to perform your drag-and-drop data binding from the correct data control. If you want your page to use a view object instance from the nested `StoreServiceAMDataControl` instance's data model that is an aggregated part of the `CompositeService` data control, then ensure that you select the data collection that appears as part of the `CompositeService` data control node in the panel.

It is important to do the drag -and-drop operation that corresponds to your intended usage. When you drop a data collection from the *top-level* `StoreServiceAMDataControl` data control node in the panel, at runtime your page will use an instance of the `StoreServiceAMDataControl` application module acquired from a pool of `StoreServiceAMDataControl` components. When you drop a data collection from the *nested* instance of `StoreServiceAMDataControl` that is part of `CompositeService`, at runtime your page will use an instance of the `CompositeService` application module acquired from a pool of `CompositeService` components. Since different types of application module data controls will have distinct transactions and database connections, inadvertently mixing and matching data collections from both a nested application module and a top-level data control will lead to unexpected runtime behavior.

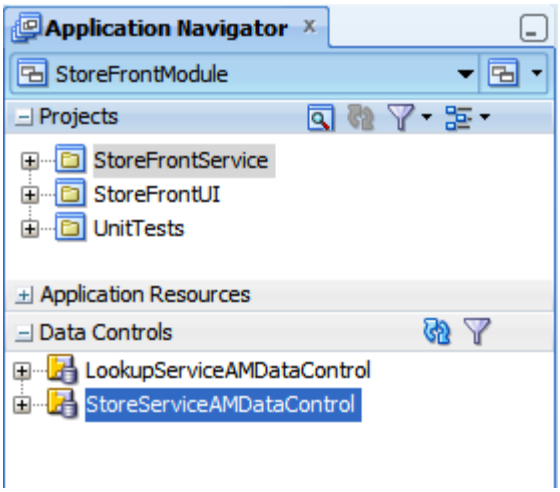
13.3.2 How to Open the Data Controls Panel

The Data Controls panel is a panel within the Application Navigator, located at the top left of JDeveloper. To view its contents, click the panel header to expand the panel. If you do not see the panel header, then the Application Navigator may not be displaying.

To open the Application Navigator and Data Controls panel:

1. From the main menu, choose **View > Application Navigator**.
2. To open the Data Controls accordion panel, click the expand icon in the Data Controls header, as shown in Figure 13–8.

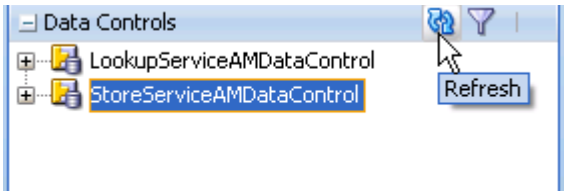
Figure 13–8 Data Controls Panel in the Application Navigator



13.3.3 How to Refresh the Data Controls Panel

Any time changes are made to the application module or underlying services, you need to manually refresh the data control in order to view the changes. To refresh the application module data control, click the **Refresh** icon in the header of the Data Controls panel, as shown in Figure 13–9.

Figure 13–9 Refresh Icon on Data Controls Panel



When you click **Refresh**, the Data Controls panel looks for all available data controls, and therefore will now reflect any structural changes made to the data control.

13.3.4 Packaging a Data Control for Use in Another Project

You can package up data controls so that they can be used in another project. For example, one development group might be tasked with creating the services and data controls, while another development group might be tasked with creating the UI. The first group would create the services and data controls, and then package them up as an Oracle ADF Library and send it to the second group. The second group can then add the data controls to their project using the Resource Palette. For more information, see Chapter 38, "Reusing Application Components."

13.4 Using the Data Controls Panel

You can design a databound user interface by dragging an item from the Data Controls panel and dropping it on a page as a specific UI component. When you use data controls to create a UI component, JDeveloper automatically creates the various code and objects needed to bind the component to the data control you selected.

In the Data Controls panel, each data control object is represented by a specific icon. Table 13–1 describes what each icon represents, where it appears in the Data Controls panel hierarchy, and what components it can be used to create.

Table 13–1 Data Controls Panel Icons and Object Hierarchy

Icon	Name	Description	Used to Create...
	Data Control	Represents a data control. You cannot use the data control itself to create UI components, but you can use any of the child objects listed under it. Depending on how your business services were defined, there may be more than one data control. Usually, there is one data control for each application module. However, you may have additional data controls that were created for other types of business services (for example, for web services). For information about creating data controls for web services, see Chapter 14, "Exposing Web Services Using the ADF Model Layer."	Serves as a container for the other object and is not used to create anything.
	Collection	Represents a named data collection. A *data collection* represents a set of data objects (also known as a *row set*) in the data model. Each object in a data collection represents a specific structured data item (also known as a *row*) in the data model. Throughout this guide, *data collection* and *collection* are used interchangeably. For application modules, the data collection is the default row set contained in a view object instance. The name of the collection matches the view object instance name. A view link creates a master-detail relationship between two view objects. If you explicitly add an instance of a detail view object (resulting from a view link) to the application module data model, the collection contained in that detail view object appears as a child of the collection contained in the master view object. For information about adding detail view objects to the data model, see Section 5.6.4, "How to Enable Active Master-Detail Coordination in the Data Model." The children under a collection may be attributes of the collection, other collections that are related by a view link, custom methods that return a value from the collection, or built-in operations that can be performed on the collection. If you've configured JDeveloper to display viewlink accessor returns, then those are displayed as well.	Forms, tables, graphs, trees, range navigation components, and master-detail components. For more information about using collections on a data control to create forms, see Chapter 26, "Creating a Basic Databound Page." For more information about using collections to create tables, see Chapter 27, "Creating ADF Databound Tables." For more information about using master-detail relationships to create UI components, see Chapter 29, "Displaying Master-Detail Data." For information about creating graphs, charts, and other visualization UI components, see Chapter 33, "Creating Databound ADF Data Visualization Components."
	Attribute	Represents a discrete data element in an object (for example, an attribute in a row). Attributes appear as children under the collections or method returns to which they belong. Only the attributes that were included in the view object are shown under a collection. If a view object joins one or more entity objects, that view object's collection will contain selected attributes from all of the underlying entity objects.	Label, text field, date, list of values, and selection list components. For information about using attributes to create fields on a page, see Section 26.2, "Using Attributes to Create Text Fields." For information about creating lists, see Chapter 30, "Creating Databound Selection Lists and Shuttles."

Table 13–1 (Cont.) Data Controls Panel Icons and Object Hierarchy

Icon	Name	Description	Used to Create...
	Structured Attribute	Represents a returned object that is neither a Java primitive type (represented as an attribute) nor a collection of any type. An example of a structured attribute would be a domain, which is a developer-created data type used to simplify application maintenance. For more information about domains, see Section 4.15, "Creating Custom, Validated Data Types Using Domains."	Label, text field, date, list of values, and selection list components.
	Method	Represents an operation in the data control or one of its exposed structures that may accept parameters, perform some business logic and optionally return single value, a structure, or a collection. In application module data controls, custom methods are defined in the application module itself and usually return either nothing or a single scalar value. For more information about creating custom methods, see Chapter 9, "Implementing Business Services with Application Modules."	Command components For methods that accept parameters: command components and parameterized forms.
	Method Return	Represents an object that is returned by a custom method. The returned object can be a single value or a collection. If a custom method defined in the application module returns anything at all, it is usually a single scalar value. Application module methods do not need to return a set of data to the view layer, because displaying the latest changes to the data is handled by the view objects in the data model (for more information, see Section 3.4, "Overview of the Oracle ADF Active Data Model"). However, custom methods in non-application module data controls (for example, a data control for a CSV file) can return collections to the view layer. A method return appears as a child under the method that returns it. The objects that appear as children under a method return can be attributes of the collection, other methods that perform actions related to the parent collection, or operations that can be performed on the parent collection.	The same components as for collections and attributes. For named criteria: query or quick query forms. For more information, see Chapter 31, "Creating ADF Databound Search Forms." When a single-value method return is dropped, the method is not invoked automatically by the framework. You need either to create an invoke action as an excecutable, or to drop the corresponding method as a button to invoke the method. For more information about executables, see Section 13.7.2.2, "Executable Binding Objects Defined in the Page Definition File."
	Operation	Represents a built-in data control operation that performs actions on the parent object. Data control operations are located in an Operations node under collections or method returns, and also under the root data control node. The operations that are children of a particular collection or method return operate on those objects only, while operations under the data control node operate on all the objects in the data control. If an operation requires one or more parameters, they are listed in a Parameters node under the operation.	UI command components, such as buttons, links, and menus. For more information, see Section 26.4, "Incorporating Range Navigation into Forms," and Section 26.5, "Creating a Form to Edit an Existing Record."
	Parameter	Represents a parameter value that is declared by the method or operation under which it appears. Parameters appear in the Parameters node under a method or operation.	Label, text, and selection list components.

13.4.1 How to Use the Data Controls Panel

JDeveloper provides you with a predefined set of UI components from which to choose for each data control item you can drop.

Before you begin:

It may be helpful to have an understanding of the different objects in the Data Controls panel. For more information, see Section 13.4, "Using the Data Controls Panel."

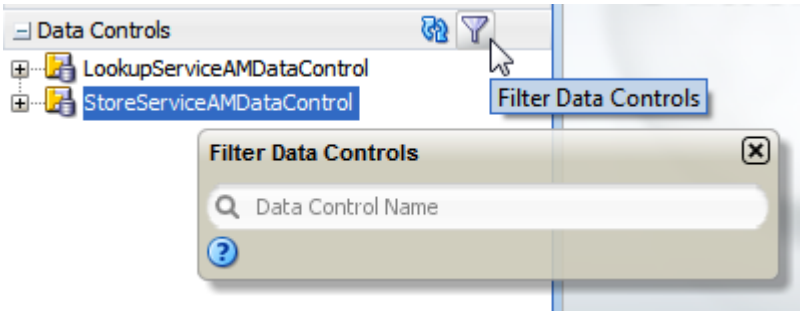
To use the Data Controls panel to create UI components:

1. Select an item in the Data Controls panel and drag it onto the visual editor for your page. For a definition of each item in the panel, see Table 13–1.

   Tip: If you need to drop an operation or method onto a method activity in a task flow, you can simply drag and drop it onto the activity in the diagram.

   Tip: You can use the Filter icon in the Data Controls Panel header to search for a specific item, as shown in Figure 13–10.

Figure 13–10 Filtering the Data Controls Panel

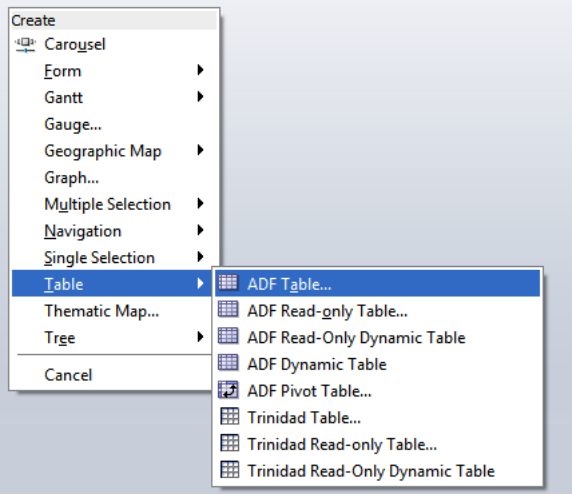


2. From the ensuing context menu, select a UI component.

   When you drag an item from the Data Controls panel and drop it on a page, JDeveloper displays a context menu of all the default UI components available for the item you dropped. The components displayed are based on the libraries in your project.

   Figure 13–11 shows the context menu displayed when a data collection from the Data Controls panel is dropped on a page.

Figure 13–11 Data Controls Panel Context Menu



Depending on the component you select from the context menu, JDeveloper may display a dialog that enables you to define how you want the component to look. For example, if you select **ADF Read-only Table** from the context menu, the Edit Table Columns dialog launches. This dialog enables you to define which attributes you want to display in the table columns, what the column labels are, what types of text fields you want use for each column, and what functionality you want to include, such as row selection or column sorting. For more information about creating tables, see Chapter 27, "Creating ADF Databound Tables."

The UI components selected by default are determined first by any UI control hints set on the corresponding business object. If no control hints have been set, then JDeveloper uses input components for standard forms and tables, and output components for read-only forms and tables. Components for lists are determined based on the type of list you chose when dropping the data control object.

Once you select a component, JDeveloper inserts the UI component on the page in the visual editor. For example, if you drag a collection from the Data Controls panel and choose **ADF Read-only Table** from the context menu, a read-only table appears in the visual editor, as shown in Figure 13–12.

Figure 13–12 Databound UI Component: ADF Read-Only Table

Product Id	Supplier Id	Category Id	Name
#{...ProductId}	#{...SupplierId}	#{...CategoryId}	#{...ProductName}
#{...ProductId}	#{...SupplierId}	#{...CategoryId}	#{...ProductName}
#{...ProductId}	#{...SupplierId}	#{...CategoryId}	#{...ProductName}

By default, the UI components created when you use the Data Controls panel use ADF Faces components, are bound to attributes in the ADF data control, and may have one or more built-in features, including:

- Databound labels
- Tooltips
- Formatting
- Basic navigation buttons

- Validation, if validation rules are attached to a particular attribute. For more information, see Chapter 7, "Defining Validation and Business Rules Declaratively."

The default components are fully functional without any further modifications. However, you can modify them to suit your particular needs. Each component and its various features are discussed further in Part V, "Creating a Databound Web User Interface".

> **Tip:** If you want to change the type of ADF databound component used on a page, the easiest method is to use either the visual editor or the structure window to delete the component, and then drag and drop a new one from the Data Controls panel. When you use the visual editor or the structure window to delete a databound component from a page, if the related binding objects in the page definition file are not referenced by any other component, JDeveloper automatically deletes those binding objects for you (automatic deletion of binding objects will not happen if you use the source editor).

13.4.2 What Happens When You Use the Data Controls Panel

When an Oracle ADF web application is built using the JSF framework, it requires a few additional application object definitions to render and process a page containing ADF databound UI components. If you do not use the Data Controls panel, you will have to manually configure these various files yourself. However, when you use the Data Controls panel, JDeveloper does all of the following required steps:

- Creates a `DataBindings.cpx` file in the default package for the project (if one does not already exist), and adds an entry for the page.

  `DataBindings.cpx` files define the *binding context* for the application. The binding context is a container object that holds a list of available data controls and data binding objects. For more information, see Section 13.4.3, "What Happens at Runtime: How the Binding Context Works." Each `DataBindings.cpx` file maps individual pages to the binding definitions in the page definition file and registers the data controls used by those pages. For more information, see Section 13.5, "Working with the DataBindings.cpx File."

- Creates the `adfm.xml` file in the META-INF directory. This file creates a registry for the `DataBindings.cpx` file, which allows the application to locate it at runtime so that the binding context can be created.

- Registers the ADF binding filter in the `web.xml` file.

  The ADF binding filter preprocesses any HTTP requests that may require access to the binding context. For more information about the binding filter configuration, see Section 13.6, "Configuring the ADF Binding Filter."

- Adds the following libraries to the view project:
  - ADF Faces Databinding Runtime
  - Oracle XML Parser v2
  - JDeveloper Runtime
  - ADF Model Runtime
  - BC4J Runtime
  - Oracle JDBC

  - Connection Manager
  - BC4J Oracle Domains
- Adds a page definition file (if one does not already exist for the page) to the page definition subpackage. The default subpackage is `view.pageDefs` in the `adfmsrc` directory.

  Tip: You can set the package configuration (such as name and location) in the ADF Model settings page of the Project Properties dialog (accessible by double-clicking the project folder).

  The page definition file (*pageName*`PageDef.xml`) defines the ADF binding container for each page in an application's view layer. The binding container provides runtime access to all the ADF binding objects for a page. In later chapters, you will see how the page definition files are used to define and edit the binding object definitions for specific UI components. For more information about the page definition file, see Section 13.7, "Working with Page Definition Files."
- Configures the page definition file, which includes adding definitions of the binding objects referenced by the page.
- Adds ADF Faces components to the JSF page.

  These prebuilt components include ADF data binding expression language (EL) expressions that reference the binding objects in the page definition file. For more information, see Section 13.8, "Creating ADF Data Binding EL Expressions."
- Adds all the libraries, files, and configuration elements required by ADF Faces components. For more information, see the "ADF Faces Configuration" appendix in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

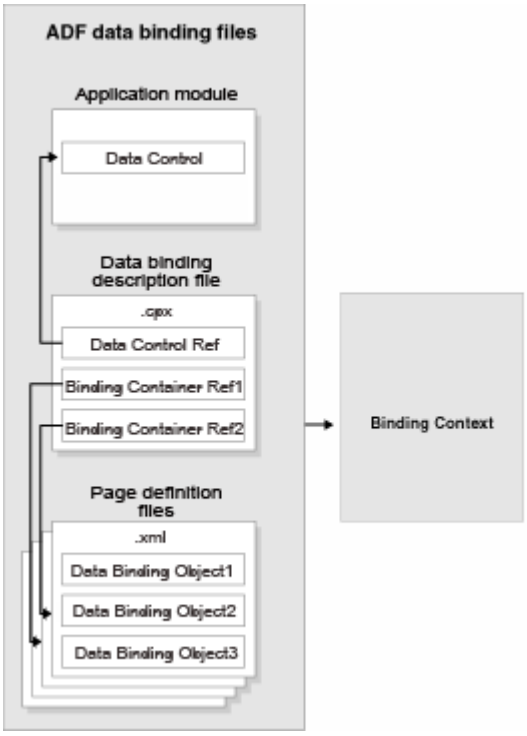
13.4.3 What Happens at Runtime: How the Binding Context Works

When a page contains ADF bindings, at runtime the interaction with the business services initiated from the client or controller is managed by the application through a single object known as the *binding context*. The binding context is a runtime map (named *data* and accessible through the EL expression `#{data}`) of all data controls and page definitions within the application.

The ADF lifecycle creates the Oracle ADF binding context from the application module, `DataBindings.cpx`, and page definition files, as shown in Figure 13–13. The union of all the `DataControls.dcx` files and any application modules in the workspace define the available data controls at design time, but the `DataBindings.cpx` files define what data controls are available to the application at runtime. A `DataBindings.cpx` file lists all the data controls that are being used by pages in the application and maps the binding containers, which contain the binding objects defined in the page definition files, to web page URLs. The page definition files define the binding objects used by the application pages. There is one page definition file for each page.

The binding context does not contain real live instances of these objects. Instead, the map contains references that become data control or binding container objects on demand. When the object (such as a page definition) is released from the application, for example when a task flow ends or when the binding container or data control is released at the end of the request, data controls and binding containers turn back into reference objects. For information about the ADF lifecycle, see Chapter 25, "Understanding the Fusion Page Lifecycle."

Figure 13–13 ADF File Binding Runtime Usage



13.5 Working with the DataBindings.cpx File

The `DataBindings.cpx` files define the binding context for the entire application and provide the metadata from which the Oracle ADF binding objects are created at runtime. An application may have more than one `DataBindings.cpx` file if a component, for example a region, was created outside of the project and then imported. These files map individual pages to page definition files and declare which data controls are being used by the application. At runtime, only the data controls listed in the `DataBindings.cpx` files are available to the current application.

13.5.1 How JDeveloper Creates a DataBindings.cpx File

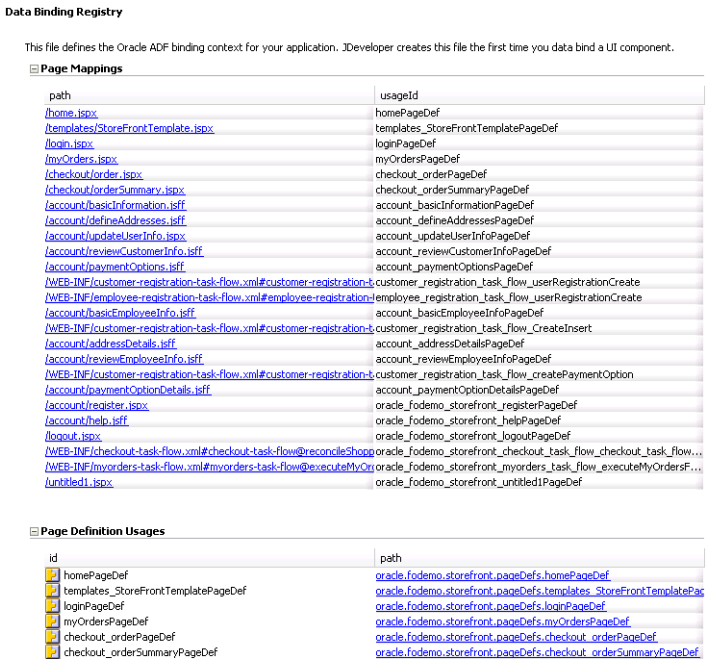
The first time you use the Data Controls panel to add a component to a page or an operation to an activity, JDeveloper automatically creates a `DataBindings.cpx` file in the default package of the view project. It resides in the `adfmsrc` directory for the project. Once the `DataBindings.cpx` file is created, JDeveloper adds an entry for the first page or task flow activity. Each subsequent time you use the Data Controls panel, JDeveloper adds an entry to the `DataBindings.cpx` for that page or activity, if one does not already exist.

> **Tip:** JDeveloper supports refactoring. That is, you can safely rename or move many of the objects referenced in the `DataBindings.cpx` file, and the references will be updated. For more information, see Chapter 37, "Refactoring a Fusion Web Application."

13.5.2 What Happens When JDeveloper Creates a DataBindings.cpx File

Once JDeveloper creates a `DataBindings.cpx` file, you can open it in the overview editor. Figure 13–14 shows the `DataBindings.cpx` file from the StoreFront module application, as viewed in the overview editor (note that it's been truncated).

Figure 13–14 DataBindings.cpx File in the Overview Editor



Example 13–1 shows an excerpt from the .cpx file in the StoreFront module application.

Example 13–1

```
<Application xmlns="http://xmlns.oracle.com/adfm/application"
             version="11.1.1.44.61" id="DataBindings" SeparateXMLFiles="false"
             Package="oracle.fodemo.storefront" ClientType="Generic"
             ErrorHandlerClass="oracle.fodemo.frmwkext.FODCustomErrorHandler">
  <definitionFactories>
    <factory nameSpace="http://xmlns.oracle.com/adf/controller/binding"
             className="oracle.adf.controller.internal.binding.
                                                TaskFlowBindingDefFactoryImpl"/>
    <factory nameSpace="http://xmlns.oracle.com/adfm/dvt"
             className="oracle.adfinternal.view.faces.dvt.model.binding.
                                                          FacesBindingFactory"/>
    <dtfactory className="oracle.adf.controller.internal.dtrt.binding.
                                                       BindingDTObjectFactory"/>
  </definitionFactories>
  <pageMap>
    <page path="/home.jspx" usageId="homePageDef"/>
    <page path="/templates/StoreFrontTemplate.jspx"
          usageId="templates_StoreFrontTemplatePageDef"/>
    <page path="/login.jspx" usageId="loginPageDef"/>
    <page path="/myOrders.jspx" usageId="myOrdersPageDef"/>
.
.
.
  </pageMap>
  <pageDefinitionUsages>
    <page id="homePageDef"
          path="oracle.fodemo.storefront.pageDefs.homePageDef"/>
    <page id="templates_StoreFrontTemplatePageDef"
          path="oracle.fodemo.storefront.pageDefs.
                                templates_StoreFrontTemplatePageDef"/>
```

```
    <page id="loginPageDef"
          path="oracle.fodemo.storefront.pageDefs.loginPageDef"/>
    <page id="myOrdersPageDef"
          path="oracle.fodemo.storefront.pageDefs.myOrdersPageDef"/>
    <page id="cart_cartSummaryPageDef"
          path="oracle.fodemo.storefront.pageDefs.cart_cartSummaryPageDef"/>
    <page id="checkout_orderPageDef"
          path="oracle.fodemo.storefront.pageDefs.checkout_orderPageDef"/>
.
.
.
  </pageDefinitionUsages>
  <dataControlUsages>
    <BC4JDataControl id="StoreServiceAMDataControl"
                     Package="oracle.fodemo.storefront.store.service"
                     FactoryClass="oracle.adf.model.bc4j.DataControlFactoryImpl"
                     SupportsTransactions="true" SupportsFindMode="true"
                     SupportsRangesize="true" SupportsResetState="true"
                     SupportsSortCollection="true"
                     Configuration="StoreServiceAMLocalWeb" syncMode="Immediate"
                     xmlns="http://xmlns.oracle.com/adfm/datacontrol"/>
    <BC4JDataControl id="LookupServiceAMDataControl"
                     Package="oracle.fodemo.storefront.lookups"
                     FactoryClass="oracle.adf.model.bc4j.DataControlFactoryImpl"
                     SupportsTransactions="true" SupportsFindMode="true"
                     SupportsRangesize="true" SupportsResetState="true"
                     SupportsSortCollection="true"
                     Configuration="LookupServiceAMLocal" syncMode="Immediate"
                     xmlns="http://xmlns.oracle.com/adfm/datacontrol"/>
  </dataControlUsages>
</Application>
```

The **Page Mappings** section of the editor maps each JSF page or task flow activity to its corresponding page definition file using an ID. The **Page Definition Usages** section maps the page definition ID to the absolute path for page definition file in the application. The **Data Control Usages** section identifies the data controls being used by the binding objects defined in the page definition files. These mappings allow the binding container to be initialized when the page is invoked.

You can use the overview editor to change the ID name for page definition files or data controls by double-clicking the current ID name and editing inline. Doing so will update all references in the application. Note, however, that JDeveloper updates only the ID name, it does not update the file name. Be sure that you do not change a data control name to a reserved word. For more information, see Section 9.2.5, "How to Edit an Existing Application Module."

You can also click an element in the Structure window and then use the Property Inspector to change property values. For more information about the elements and attributes in the `DataBindings.cpx` file, see Section A.7, "DataBindings.cpx."

13.6 Configuring the ADF Binding Filter

The ADF binding filter is a servlet filter that is an instance of the `oracle.adf.model.servlet.ADFBindingFilter` class. ADF web applications use the ADF binding filter to preprocess any HTTP requests that may require access to the binding context. To do this, the ADF binding filter must be aware of all `DataBindings.cpx` files that exist for an application.

13.6.1 How JDeveloper Configures the ADF Binding Filter

The first time you add a databound component to a page using the Data Controls panel, JDeveloper automatically configures the filter for you in the application's `web.xml` file.

13.6.2 What Happens When JDeveloper Configures an ADF Binding Filter

To configure the binding filter, JDeveloper adds the following elements to the `web.xml` file:

- An ADF binding filter class: Specifies the name of the binding filter object, which implements the `javax.servlet.Filter` interface.

  The ADF binding filter is defined in the `web.xml` file, as shown in Example 13–2. The `filter-name` element must contain the value `adfBindings`, and the `filter-class` element must contain the fully qualified name of the binding filter class, which is `oracle.adf.model.servlet.ADFBindingFilter`.

Example 13–2 Binding Filter Class Defined in the web.xml File

```
<filter>
   <filter-name>adfBindings</filter-name>
   <filter-class>oracle.adf.model.servlet.ADFBindingFilter</filter-class>
</filter>
```

- Filter mappings: Link filters to static resources or servlets in the web application.

  At runtime, when a mapped resource is requested, a filter is invoked. Filter mappings are defined in the `web.xml` file, as shown in Example 13–3. The `filter-name` element must contain the value `adfBindings`.

Example 13–3 Filter Mapping Defined in the web.xml File

```
<filter-mapping>
  <filter-name>adfBindings</filter-name>
  <servlet-name>Faces Servlet</servlet-name>
  <dispatcher>FORWARD</dispatcher>
  <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

> **Tip:** If you have multiple filters defined in the `web.xml` file, be sure to list them in the order in which you want them to run. At runtime, the filters are executed in the sequence in which they appear in the `web.xml` file.
>
> The `adfBindings` filter should appear after the Trinidad filter and before any filters that depend on the ADF context to be initialized.

13.6.3 What Happens at Runtime: How the ADF Binding Filter Works

At runtime, the ADF binding filter performs the following functions:

- Overrides the character encoding when the filter is initialized with the name specified as a filter parameter in the `web.xml` file. The parameter name of the filter `init-param` element is `encoding`.
- Instantiates the `ADFContext` object, which is the execution context for a Fusion web application and contains context information about ADF, including the security context and the environment class that contains the request and response object.

- Initializes the binding context for a user's HTTP session. To do this, it first loads the bindings as defined in the `DataBindings.cpx` file in the current project's `adfmsrc` directory. If the application contains `DataBindings.cpx` files that were imported from another project, those files are present in the application's class path. The filter additively loads any auxiliary `.cpx` files found in the class path of the application.
- Serializes incoming HTTP requests from the same browser (for example, from frame sets) to prevent multithreading problems.
- Notifies data control instances that they are about to receive a request, allowing them to do any necessary per-request setup.
- Notifies data control instances after the response has been sent to the client, allowing them to do any necessary per-request cleanup.

13.7 Working with Page Definition Files

Page definition files define the binding objects that populate the data in UI components at runtime. For every page that has ADF bindings, there must be a corresponding page definition file that defines the binding objects used by that page. Page definition files provide design time access to all the ADF bindings. At runtime, the binding objects defined by a page definition file are instantiated in a binding container, which is the runtime instance of the page definition file.

Note: When multiple windows are open to the same page, the ADF Controller assigns each window its own `DataControlFrame`. This ensures that each window has its own binding container.

13.7.1 How JDeveloper Creates a Page Definition File

The first time you use the Data Controls panel, JDeveloper automatically creates a page definition file for that page and adds definitions for each binding object referenced by the component. For each subsequent databound component you add to the page, JDeveloper automatically adds the necessary binding object definitions to the page definition file.

By default, the page definition files are located in the `view.PageDefs` package in the Application Sources directory of the view project. If the corresponding JSF page is saved to a directory other than the default (`public_html`), or to a subdirectory of the default, then the page definition will also be saved to a package of the same name. For example, if you save your JSF file to the `public_html\myDirectory` directory, the page definition will be saved to the `myDirectory` package. You can change the location of the page definition files using the ADF Model Settings page of the Project Properties dialog.

JDeveloper names the page definition files using the following convention:

`pageName``PageDef.xml`

where *`pageName`* is the name of the JSF page. For example, if the JSF page is named `home.jsp`, the default page definition file name is `homePageDef.xml`. If you organize your pages into subdirectories, JDeveloper prefixes the directory name to the page definition file name using the following convention:

`directoryName_pageName``PageDef.xml`

For example, in the StoreFront module, the name of the page definition file for the `updateUserInfo` page, which is in the `account` subdirectory of the Web Content node is `account_updateUserInfoPageDef.xml`.

> **Tip:** Page definitions for task flows follow the same naming convention.

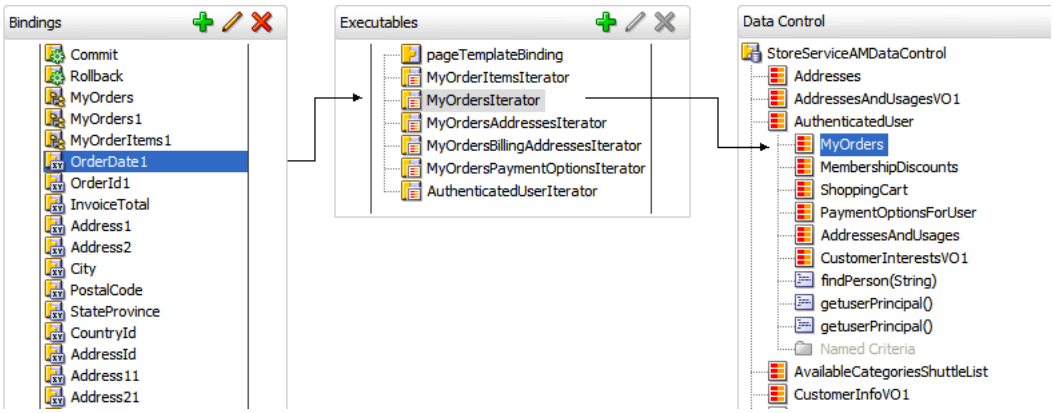
To open a page definition file, you can right-click directly on the page or activity in the visual editor, and choose **Go to Page Definition**, or for a JSF page, you can click the **Bindings** tab of the editor and click the Page Definition File link.

> **Tip:** While JDeveloper automatically creates a page definition for a JSF page when you create components using the Data Controls panel, or for a task flow when you drop an item onto an activity, it does not delete the page definition when you delete the associated JSF page or task flow activity (this is to allow bindings to remain when they are needed without a JSF page, for example when using desktop integration). If you no longer want the page definition, you need to delete the page definition and all references to it manually. Note however, that as long as a corresponding page or activity is never called, the page definition will never be used to create a binding context. It is therefore not imperative to remove any unused page definition files from your application.

13.7.2 What Happens When JDeveloper Creates a Page Definition File

When JDeveloper creates a paged definition file, it is displayed in the overview editor. Figure 13–15 shows the page definition file in the overview editor that was created for the `myOrders.jspx` page in the StoreFront module application.

Figure 13–15 Page Definition File in the Overview Editor



The overview editor contains the following tabs, which allow you to view and configure bindings, contextual events, and parameters for a page:

- Bindings and Executables: The Bindings and Executables tab of the page definition overview editor shows three different types of objects: bindings, executables, and the associated data controls (note that the data controls do not display unless you select a binding or executable). For example, in Figure 13–15, you can see that the binding for the `OrderDate1` attribute uses the `MyOrdersIterator` iterator to get its value. The iterator accesses the `MyOrders` collection on the

`StoreServiceAMDataControl` data control. For more information, see Section 13.7.2.2, "Executable Binding Objects Defined in the Page Definition File."

By default, the model binding objects are named after the data control object that was used to create them. If a data control object is used more than once on a page, JDeveloper adds a number to the default binding object names to keep them unique. In Section 13.8, "Creating ADF Data Binding EL Expressions," you will see how the ADF data binding EL expressions reference the binding object names.

Table 13–2 shows the icons for each of the binding objects, as displayed in the overview editor (note that while parameter objects are shown in the **Parameter** section of the editor, they are also considered binding objects).

Table 13–2 Binding Object Icons

Binding Object Type	Icon	Description
Parameter		Represents a parameter binding object.
Bindings		Represents an attribute value binding object.
		Represents a list value binding object.
		Represents a tree value binding object.
		Represents a method action binding object
Bindings/ Executables		Represents an action binding object. Also represents an invoke action executable binding object and an event.
Executables		Represents an iterator binding object.
		Represents a task flow executable binding object.

- Contextual Events: You can create contextual events that artifacts in an application can subscribe to. For example, in the StoreFront module, contextual events are used in the customer registration page to display the appropriate informational topic. The `register.jspx` page contains two regions. One region contains the customer registration task flow, and the other contains the informational topic task flow. A contextual event is passed from the customer registration region to the informational topic region so that the informational topic task flow can display the correct information topic. At design time, the event name, producer region, consumer region, consumer handler, and other information is stored in the event map section of the page definition file.
- Parameters: Parameter binding objects declare the parameters that the page evaluates at the beginning of a request. (For more information about the ADF lifecycle, see Chapter 25, "Understanding the Fusion Page Lifecycle.") You can define the value of a parameter in the page definition file using static values, or EL expressions that assign a static value.

  Example 13–4 shows how parameter binding objects can be defined in a page definition file.

Example 13–4 parameters Element of a Page Definition File

```
<parameters>
  <parameter id="filedBy"
             value="${bindings.userId}"/>
  <parameter id="status"
             value="${param.status != null ?  param.status : 'Open'}"/>
</parameters>
```

The value of the `filedBy` parameter is defined by a binding on the `userID` data attribute, which would be an attribute binding defined later in the `bindings` element. The value of the `status` parameter is defined by an EL expression, which assigns a static value.

> **Tip:** The EL expression for the parameter values uses the dollar sign ($) because these expressions need to be resolved eagerly, so that the result is returned immediately, as the page is rendered. Most EL expressions in a JSF application use the hash sign (#), which defers the expression evaluation so that the model is prepared before the values are accessed.

When you click an item in the overview editor (or the associated node in the Structure window), you can use the Property Inspector to view and edit the attribute values for the item, or you can edit the XML source directly by clicking the **Source** tab. Example 13–5 shows abbreviated XML code for the page definition file shown in Figure 13–15.

Example 13–5 Page Definition File

```
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="11.1.1.44.61" id="myOrdersPageDef"
                Package="oracle.fodemo.storefront.pageDefs"
                EnableTokenValidation="false">
  <parameters/>
  <executables>
    <page path="oracle.fodemo.storefront.pageDefs.
                                        templates_StoreFrontTemplatePageDef"
          id="pageTemplateBinding"/>
    <iterator Binds="MyOrderItems" RangeSize="25"
              DataControl="StoreServiceAMDataControl"
              id="MyOrderItemsIterator"/>
    <iterator Binds="MyOrders" RangeSize="-1"
              DataControl="StoreServiceAMDataControl" id="MyOrdersIterator"/>
.
.
.
  </executables>
  <bindings>
    <action id="Commit" InstanceName="StoreServiceAMDataControl"
            DataControl="StoreServiceAMDataControl" RequiresUpdateModel="true"
            Action="commitTransaction"/>
    <action id="Rollback" InstanceName="StoreServiceAMDataControl"
            DataControl="StoreServiceAMDataControl" RequiresUpdateModel="false"
            Action="rollbackTransaction"/>
    <methodAction id="executeMyOrdersForCustomerVO" RequiresUpdateModel="true"
                  Action="invokeMethod"
                  MethodName="executeMyOrdersForCustomerVO"
                  IsViewObjectMethod="false"
                  DataControl="StoreServiceAMDataControl"
                  InstanceName="StoreServiceAMDataControl.dataProvider"/>
```

```
    <tree IterBinding="MyOrdersIterator" id="MyOrders">
      <nodeDefinition DefName="oracle.fodemo.storefront.store.queries.OrdersVO">
        <AttrNames>
          <Item Value="OrderId"/>
          <Item Value="OrderDate"/>
          <Item Value="OrderShippedDate"/>
          <Item Value="OrderStatusCode"/>
          <Item Value="OrderTotal"/>
          <Item Value="CustomerId"/>
          <Item Value="ShipToName"/>
          <Item Value="ShipToAddressId"/>
          <Item Value="ShipToPhoneNumber"/>
          <Item Value="ShippingOptionId"/>
          <Item Value="PaymentOptionId"/>
          <Item Value="CalculatedOrderTotal"/>
          <Item Value="TotalShippingCost"/>
          <Item Value="DiscountAmount"/>
          <Item Value="InvoiceTotal"/>
          <Item Value="LastUpdateDate"/>
          <Item Value="TypedCouponCode"/>
        </AttrNames>
      </nodeDefinition>
    </tree>
    <attributeValues IterBinding="MyOrdersIterator" id="OrderDate1">
      <AttrNames>
        <Item Value="OrderDate"/>
      </AttrNames>
    </attributeValues>
    <attributeValues IterBinding="MyOrdersIterator" id="OrderId1">
      <AttrNames>
        <Item Value="OrderId"/>
      </AttrNames>
    </attributeValues>
.
.
.
  </bindings>
</pageDefinition>
```

In later chapters, you will see how the page definition file is used to define and edit the bindings for specific UI components. For a description of all the possible elements and attributes in the page definition file, see Section A.8, "pageNamePageDef.xml."

13.7.2.1 Bindings Binding Objects Defined in the Page Definition File

There are three types of Bindings binding objects used to bind UI components to objects on the data control:

- Value: Displays data in UI components by referencing an iterator binding. Each discrete UI component on a page that will display data from the data control is bound to a value binding object. Value binding objects include:
  - Attribute Values: Binds text fields to a specific attribute in an object (also referred to as an *attribute binding* object.)
  - List: Binds the list items to all values of an attribute in a data collection.
  - Tree: Binds an entire table to a data collection and can also bind the root node of a tree to a data collection.
  - Button (boolean): Binds a checkbox to a boolean value for an attribute.

  - Graph: Binds a graph directly to the source data.

- Method Action: Binds command components, such as buttons or links, to custom methods on the data control. A method action binding object encapsulates the details about how to invoke a method and what parameters (if any) the method is expecting.
- Action: Binds command components, such as buttons or links, to built-in data control operations (such as, Commit or Rollback) or to built-in collection-level operations (such as, Create, Delete, Next, or Previous).

Collectively, the binding objects are referred to as *control* binding objects, because they work with the UI controls on a page.

Example 13–6 shows a sample `bindings` element, which defines one action binding called `Commit`, one attribute binding for a text field called `PaymentOptionID1`, and one list binding called `PaymentTypeCode`.

Example 13–6 bindings Element of a Page Definition File

```
<bindings>
  <action id="Commit" InstanceName="StoreServiceAMDataControl"
          DataControl="StoreServiceAMDataControl" RequiresUpdateModel="true"
          Action="commitTransaction"/>
  <attributeValues IterBinding="PaymentOptionsForUserIterator"
                   id="PaymentOptionId1">
    <AttrNames>
      <Item Value="PaymentOptionId"/>
    </AttrNames>
  </attributeValues>
  <list IterBinding="PaymentOptionsForUserIterator" id="PaymentTypeCode"
        Uses="LOV_PaymentTypeCode" StaticList="false">
    <AttrNames>
      <Item Value="PaymentTypeCode"/>
    </AttrNames>
  </list>
</bindings>
```

The binding object defined in the action element encapsulates the information needed to invoke the built-in `commit` operation on the `StoreServiceAMDataControl` data control. The value of `true` in the `RequiresUpdateModel` attribute specifies that the model layer needs to be updated before the operation is executed.

If this operation also raised a contextual event, an event definition would appears well. If the page contained bindings that consumed an event, the event mapping would also appear.

The `attributeValues` element defines the value bindings for the text fields on the page. In the example, the `PaymentOptionId1` attribute binding will display the value of the `PaymentOptionId`, which is defined in the `AttrNames` element. The `IterBinding` attribute references the iterator binding that manages the data to be displayed in the text field (for more information, see Section 13.7.2.2, "Executable Binding Objects Defined in the Page Definition File").

The `PaymentTypeCode` element defines the list binding used to display the list of payment type codes by accessing the LOV created on the `PaymentOptions` view object. For more information about creating lists using LOVs on view objects, see Chapter 30, "Creating Databound Selection Lists and Shuttles."

13.7.2.2 Executable Binding Objects Defined in the Page Definition File

There are seven types of executable binding objects:

- Iterator: Binds to an iterator that iterates over view object collections. There is one iterator binding for each collection used on the page. All of the value bindings on the page must refer to an iterator binding in order for the component values to be populated with data at runtime.

  When you drop a collection or an attribute of a collection on the page, an iterator binding is automatically added as an executable. Iterator binding objects bind to an underlying ADF `RowSetIterator` object, which manages the current object and current range information. The iterator binding exposes the current object and range state to the other binding objects used by the page.

  By default, iterator bindings are configured so that any submitted data changes are cached until the data is committed back to the data source. When a data change is submitted, any components on the page whose bindings are associated with the iterator are refreshed to show the changed data. For more information, see Section 25.2.1, "What You May Need to Know About Partial Page Rendering and Iterator Bindings."

  The iterator *range* represents the current set of objects to be displayed on the page. The maximum number of objects in the current range is defined in the `rangeSize` attribute of the iterator. For example, if a collection in the data control contains products and the iterator range size is 25, the first 25 products in the collection are displayed on the page. If the user scrolls down, the next set of 25 is displayed, and so on. If the user scrolls up, the previous set of 25 is displayed. If your view object uses range paging, then you can configure the iterator binding to return a set of ranges at one time. For more information, see Section 42.1.5, "Efficiently Scrolling Through Large Result Sets Using Range Paging."

  > **Note:** If you have two pages each with an iterator binding bound to the iterator on the same view object (which you will if you drop the same collection, for example, on two different pages), then you should ensure that the `rangeSize` attribute is the same for both pages' iterator bindings. If not, the page with a smaller range size may cause the iterator to reexecute, causing unexpected results on the other page.

- Method Iterator: Binds to an iterator that iterates over the collections returned by custom methods in the data control.

  A method iterator binding is always related to a method action binding object. The method action binding encapsulates the details about how to invoke the method and what parameters (if any) the method is expecting. The method action binding is itself bound to the method iterator, which provides the data.

  You will see method iterator executable binding objects only if you drop a method return collection or an attribute of a method return collection from a custom method on the data control. If you are using only application module data controls, you will see only iterator binding objects.

- Variable Iterator: Binds to an iterator that exposes all the variables in the binding container to the other bindings. While there is an iterator binding for each collection, there is only one variable iterator binding for all variables used on the page. (The variable iterator is like an iterator pointing to a collection that contains only one data object whose attributes are the binding container variables.)

Page variables are local to the binding container and exist only while the binding container object exists. When you use a data control method (or an operation) that requires a parameter that is to be collected from the page, JDeveloper automatically defines a variable for the parameter in the page definition file. Attribute bindings can reference the page variables.

A variable iterator can contain one of two types of variables: `variable` and `variableUsage`. A `variable` type variable is a simple value holder, while a `variableUsage` type variable is a value holder that is related to a view object's named bind parameter. Defining a variable as a `variableUsage` type allows it to inherit the default value and UI control hints from the view object named bind variable to which it is bound.

- Invoke Action: Binds to a method that invokes the operations or methods defined in action or method action bindings during any phase of the page lifecycle.

  Tip: If you know you want a method to execute before the page is rendered, you should use a method call activity in the task flow to invoke the method, rather than an invoke action in the page definition file. Using the method call activity makes invoking page logic easier, and allows you to show more information on the task flow, making the diagram more readable and useful to anyone else who might be using it. However, if you need the method to be executed in more than one phase of the page's lifecycle, or if you plan to reuse the page and page definition file and want the method to be tied to the page, or if your application does not use ADFc, then you should use an invoke action to invoke the method.

- Page: Binds to the template's page definition file (if a template is used). For more information about how this works with templates, see Section 24.2, "Using Page Templates."

  Note: You can also use the page element to bind to another page definition file. However, at runtime, only the current incoming page's (or if the rendered page is different from the incoming, the rendered page's) binding container is automatically prepared by the framework during the current request. Therefore, to successfully access a bound value in another page from the current page, you must programmatically prepare that page's binding container in the current request (for example, using a backing bean). Otherwise, the bound values in that page may not be available or valid in the current request.

- Search Region: Binds named criteria to the iterator, so that the search can be executed.
- Task Flow: Instantiates the binding container for a region's task flow.
- Multi Task Flow: instantiates the binding container for a region's task flow from an array of task flows. This is useful when you have a page that contains an unknown number of regions, for example a a `panelTabbed` component where each tab is a region, and users can add and delete tabs at runtime. For more information, see Section 21.10, "Configuring a Page To Render an Unknown Number of Regions."

At runtime, executable bindings are refreshed based on the value of their `Refresh` attribute. Refreshing an iterator binding reconnects it with its underlying `RowSetIterator` object. Refreshing an invoke action binding invokes the action. Before refreshing any bindings, the ADF runtime evaluates any `Refresh` and `RefreshCondition` attributes specified in the executables. The `Refresh` attribute specifies the ADF lifecycle phase within which the executable should be invoked. The `RefreshCondition` attribute specifies the conditions under which the executable should be invoked. You can specify the `RefreshCondition` value using a boolean EL expression. If you leave the `RefreshCondition` attribute blank, it evaluates to `true`.

By default, the `Refresh` value is set to `deferred`. This means the binding will not be executed unless its value is accessed (for example by an EL expression on a JSF page). Once called, it will not reexecute unless any parameter values for the binding have changed, or if the binding itself has changed.

For more information about how bindings are refreshed and how to set the `Refresh` and `RefreshCondition` attributes, see Section 25.2, "About the JSF and ADF Page Lifecycles."

Example 13–7 shows an example of executable binding objects.

Example 13–7 executable Binding Objects in a Page Definition File

```
<executables>
  <page path="oracle.fodemo.storefront.pageDefs.
                                        templates_StoreFrontTemplatePageDef"
        id="pageTemplateBinding"/>
  <iterator Binds="MyOrderItems" RangeSize="25"
            DataControl="StoreServiceAMDataControl"
            id="MyOrderItemsIterator"/>
  <iterator Binds="MyOrders" RangeSize="-1"
            DataControl="StoreServiceAMDataControl" id="MyOrdersIterator"/>
```

The iterator binding named `MyOrderItems` was created by dropping the `MyOrderItems` collection on the page as a table. The iterator binding named `MyOrders` was created by dropping the `MyOrders` collection, which has a master-detail relationship with the `MyOrderItems` collection. For more information, see Chapter 29, "Displaying Master-Detail Data."

The `Binds` attribute of the `iterator` element defines the collection the iterator will iterate over. The `RangeSize` attribute defines the number of objects the iterator is to display on the page at one time. A `RangeSize` value of `-1` causes the iterator to display *all* the objects from the collection.

> **Tip:** Normally, an iterator binding's default range size is 25. However, when an iterator binding is created from the Edit List Binding dialog, the range size defaults to `-1` so that all choices display in the list, not just the first 25.

> **Performance Tip:** When you want to reduce the number of roundtrips the iterator requires to fetch the data objects from the view object in the ADF Business Components layer, you can set the `rangeSize` attribute to `-1`, and the objects will be fetched in a single round trip to the server, rather than in multiple trips as the user navigates through the objects.

13.8 Creating ADF Data Binding EL Expressions

To display data from the data model, web page UI components are bound to binding objects using JSF Expression Language (EL) expressions. These EL expressions reference a specific binding object in a binding container. At runtime, the JSF runtime evaluates an EL expression and pulls the value from the binding object to populate the component with data when the page is displayed. If the user updates data in the UI component, the JSF runtime pushes the value back into the corresponding binding object based on the same EL expression.

> **Tip:** There may be cases when you need to use EL expressions within managed beans. For information on working with EL expressions within managed beans, see the "Creating EL Expressions" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

13.8.1 How to Create an ADF Data Binding EL Expression

When you use the Data Controls panel to create a component, the ADF data binding expressions are created for you. The expressions are added to every component attribute that will either display data from or reference properties of a binding object. Each prebuilt expression references the appropriate binding objects defined in the page definition file. You can edit these binding expressions or create your own, as long as you adhere to the basic ADF binding expression syntax. ADF data binding expressions can be added to any component attribute that you want to populate with data from a binding object.

In JSF pages, a typical ADF data binding EL expression uses the following syntax to reference any of the different types of binding objects in the binding container:

```
#{bindings.BindingObject.propertyName}
```

where:

- `bindings` is a variable that identifies that the binding object being referenced by the expression is located in the binding container of the current page. All ADF data binding EL expressions must start with the `bindings` variable.
- *`BindingObject`* is the ID, or for attributes the name, of the binding object as it is defined in the page definition file. The binding `objectID` or name is unique to that page definition file. An EL expression can reference any binding object in the page definition file, including parameters, executables, or value bindings.
- *`propertyName`* is a variable that determines the default display characteristics of each databound UI component and sets properties for the binding object at runtime. There are different binding properties for each type of binding object. For more information about binding properties, see Section 13.8.2, "What You May Need to Know About ADF Binding Properties."

For example, in the following expression that might appear on a JSF page:

```
#{bindings.ProductName.inputValue}
```

the `bindings` variable references a bound value in the current page's binding container. The binding object being referenced is `ProductName`, which is an attribute binding object. The binding property is `inputValue`, which returns the value of the first `ProductName` attribute.

Tip: While the binding expressions in the page definition file can use either a dollar sign ($) or hash sign (#) prefix, the EL expressions in JSF pages can use only the hash sign (#) prefix.

As stated previously, when you use the Data Controls panel to create UI components, these expressions are built for you. However, you can also manually create them if you need to. The JDeveloper Expression Builder is a dialog that helps you build EL expressions by providing lists of binding objects defined in the page definition files, as well as other valid objects to which a UI component may be bound. It is particularly useful when creating or editing ADF databound expressions because it provides a hierarchical list of ADF binding objects and their most commonly used properties. For information about binding properties, see Section 13.8.2, "What You May Need to Know About ADF Binding Properties."

13.8.1.1 Opening the Expression Builder from the Property Inspector

You can select an item in the visual editor, and then create EL expressions for specific attributes using the Property Inspector.

To open the Expression Builder from the Property Inspector:

1. Select a UI component in the Structure window or the visual editor.
2. In the Property Inspector, click the dropdown list next to a field, and choose **Expression Builder**.

13.8.1.2 Using the Expression Builder

Once the Expression Builder is open, you can use it to create EL expressions.

Before you begin:

It may be helpful to have an understanding of how to create EL expressions when using the ADF Model layer. For more information, see Section 13.8, "Creating ADF Data Binding EL Expressions."

To use the Expression Builder:

1. Open the Expression Builder dialog.
2. Use the Expression Builder to edit or create ADF binding expressions using the following features:
   - Use the **Variables** tree to select items that you want to include in the binding expression. The tree contains a hierarchical representation of the binding objects. Each icon in the tree represents various types of binding objects that you can use in an expression (see Table 13–3 for a description of each icon).

     To narrow down the tree, you can either use the dropdown filter or enter search criteria in the search field. Double-click an item in the tree to move it to the **Expression** box.
   - Use the operator buttons to add logical or mathematical operators to the expression.

     Tip: You can also type the expression directly in the **Expression** box.

Table 13–3 Icons Under the ADF Bindings Node of the Expression Builder

Icon	Description
	Represents the `bindings` container variable, which references the binding container of the current page. Opening the `bindings` node exposes all the binding objects for the current page.
	Represents the `data` binding variable, which references the entire binding context (created from all the `.cpx` files in the application). Opening the `data` node exposes all the page definition files in the application.
	Represents an action binding object. Opening a node that uses this icon exposes a list of valid action binding properties.
	Represents an iterator binding object. Opening a node that uses this icon exposes a list of valid iterator binding properties.
	Represents an attribute binding object. Opening a node that uses this icon exposes a list of valid attribute binding properties.
	Represents a list binding object. Opening a node that uses this icon exposes a list of valid list binding properties.
	Represents a table or tree binding object. Opening a node that uses this icon exposes a list of valid table and tree binding properties.
	Represents an ADF binding object property. For more information about ADF properties, see Section 13.8.2, "What You May Need to Know About ADF Binding Properties."
	Represents a parameter binding object.
	Represents a JavaBean.
	Represents a method.

13.8.2 What You May Need to Know About ADF Binding Properties

When you create a databound component using the Expression Builder, the EL expression might reference specific ADF binding properties. At runtime, these binding properties can define such things as the default display characteristics of a databound UI component or specific parameters for iterator bindings. The ADF binding properties are defined by Oracle APIs. For a full list of the available properties for each binding type, see Appendix B, "Oracle ADF Binding Properties."

Values assigned to certain properties are defined in the page definition file. For example, iterator bindings have a property called `RangeSize`, which specifies the number of rows the iterator should display at one time. The value assigned to `RangeSize` is specified in the page definition file, as shown in Example 13–8.

Example 13–8 Iterator Binding Object with the RangeSize Property

```
<iterator Binds="ProductsByCategory1" RangeSize="25"
              DataControl="StoreFrontModuleDataControl"
              id="Products2Iterator"/>
```

13.9 Using Simple UI First Development

While the Data Controls panel enables you to design and create bound components in a single drag-and-drop action, in some cases, it may be preferable to create the basic UI components first and add the bindings later. For example, if your page will use declarative components, you will first need to drop the declarative component, and then bind it to the correct ADF control. Declarative components are reusable, composite UI components that are made up of other ADF Faces components. Once imported into a project, declarative components can be dropped onto a page from the Component Palette, similar to standard ADF Faces components. While the entire declarative component cannot use ADF data binding, you can use ADF data binding on the individual components that make up the declarative component, once the declarative component is dropped on the page. For more information about declarative components, see the "Using Declarative Components" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

> **Note:** If you know the UI components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see Chapter 17, "Designing a Page Using Placeholder Data Controls."

When designing web pages, keep in mind that ADF bindings can be added only to certain ADF Faces tags or their equivalent JSF HTML tags. Table 13–4 lists the ADF Faces and JSF tags to which you can later add ADF bindings.

> **Tip:** To enable the use of JSF Reference Implementation UI component tags with ADF bindings, you must choose the **Include JSF HTML Widgets for JSF Databinding** option in the **ADF View Settings** of the project properties. However, using ADF Faces tags, especially with ADF bindings, provides greater functionality than does using the reference implementation JSF tags.

Table 13–4 Tags That Can Be Used for ADF Bindings

ADF Faces Tags Used in ADF Bindings	Equivalent JSF HTML Tags
Text Fields	
af:inputText	h:inputText
af:outputText	h:outputText
af:outputLabel	h:outputLabel
af:inputDate	n/a
Tables	
af:table	h:dataTable
Actions	
af:commandButton	h:commandButton

Table 13–4 (Cont.) Tags That Can Be Used for ADF Bindings

ADF Faces Tags Used in ADF Bindings	Equivalent JSF HTML Tags
af:commandLink	h:commandLink
af:commandMenuItem	n/a
af:commandToolbarButton	n/a
Selection Lists	
af:inputListOfValues	n/a
af:selectOneChoice	h:selectOneMenu
af:selectOneListbox	h:selectOneListbox
af:selecOneRadio	h:selectOneRadio
af:selectBooleanCheckbox	h:selectBooleanCheckbox
Queries	
af:query	n/a
af:quickQuery	n/a
Trees	
af:tree	n/a
af:treeTable	n/a

Before adding binding to the UI components, ensure that you follow these guidelines:

- When creating the JSF page using the Create JSF JSP wizard, choose the **Do not Automatically Expose UI Components in a Managed Bean** option on the **Managed Bean** tab.

  This option turns off JDeveloper's auto-binding feature, which automatically associates every UI component in the page to a corresponding property in the backing bean for eventual programmatic manipulation. If you intend to add ADF bindings to a page, do not use the auto-binding feature. If you use the auto-binding feature, you will have to remove the managed bean bindings later, after you have added the ADF bindings. The managed bean UI component property bindings do not affect the ADF bindings, but their presence may be confusing in the JSF code. For information about managed beans, see Section 24.4, "Using a Managed Bean in a Fusion Web Application."

- Add the ADF Faces tag libraries.

  While you can add ADF bindings to JSF components, the ADF Faces components provide greater functionality, especially when combined with ADF bindings.

13.9.1 How to Apply ADF Model Data Binding to Existing UI Components

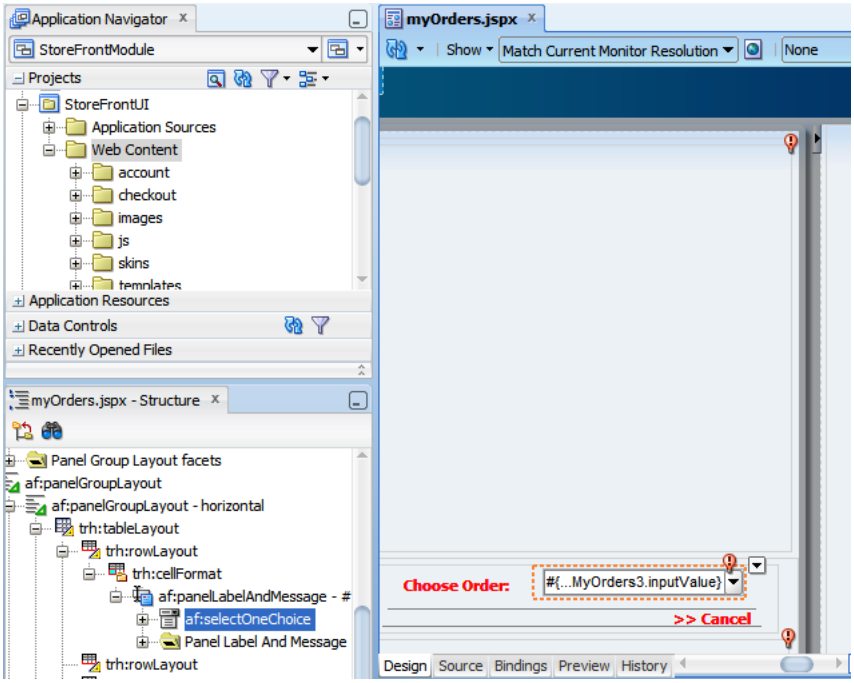
You apply ADF model binding to components using the Structure window.

To apply ADF Model data binding:

1. In the Design page of the visual editor, select the UI component to which you want to add ADF bindings.

   The component must be one of the tags listed in Table 13–4. When you select a component in the visual editor, JDeveloper simultaneously selects that component tag in the Structure window, as shown in Figure 13–16.

Figure 13–16 The Structure Window in JDeveloper



2. In the Structure window, right-click the UI component, and from the context menu, choose **Bind to ADF Control**.

> **Note:** Your project must already contain data controls for the **Bind to ADF Control** menu option to appear. If yours does not, you should consider using placeholder data controls, as described in Chapter 17, "Designing a Page Using Placeholder Data Controls."

3. In the Bind to ADF Control dialog, select the data control to which you want the UI component bound. JDeveloper will notify you if you choose a control that is not compatible with the selected UI component.

13.9.2 What Happens When You Apply ADF Model Data Binding to UI Components

When you use the Data Controls panel all of the required ADF objects are automatically created for you, as described in Section 13.4.2, "What Happens When You Use the Data Controls Panel."

14

Exposing Web Services Using the ADF Model Layer

This chapter describes how to call a third-party web service and work directly with the service proxy and service data objects (SDOs) programmatically for all common remote service data access tasks. It also describes how to create ADF data controls for third-party web services when you want to work with the web service in the user interface.

This chapter includes the following sections:

- Section 14.1, "About Web Services in Fusion Web Applications"
- Section 14.2, "Creating Web Service Data Controls"
- Section 14.3, "Creating a New Web Service Connection"
- Section 14.4, "Securing Web Service Data Controls"

14.1 About Web Services in Fusion Web Applications

Web services allow enterprises to expose business functionality irrespective of the platform or language of the originating application because the business functionality is exposed in such a way that it is abstracted to a message composed of standard XML constructs that can be recognized and used by other applications.

Web services are modular business services that can be easily integrated and reused, and it is this that makes them ideally suited as components within SOA. JDeveloper helps you to create top-down web services (services created starting from a WSDL), bottom-up web services (created from the underlying implementation such as a Java class or a PL/SQL stored procedure in a database), and services created from existing functionality, such as exposing an application module as a service.

14.1.1 Web Services Use Cases and Examples

You can consume web services in web applications, and common reasons for wanting to do so are:

- To add functionality which would be time-consuming to develop with the application, but which is readily available as a web service
- To access an application that runs on different architecture
- To access an application that is owned by another team when their application must be independently installed, upgraded, and maintained, especially when its data is not replicated locally (for example, when other methods of accessing their application cannot be used).

14.1.2 Additional Functionality for Web Services in Fusion Applications

You may find it helpful to understand other ADF features before you start working with web services. Following are links to other functionality that may be of interest.

- You can design a databound user interface by dragging an item from the Data Controls panel and dropping it on a page as a specific UI component. For more information, see Section 13.4.1, "How to Use the Data Controls Panel."
- If you are working behind a firewall and you want to use a web service that is outside the firewall, you must configure the web browser and proxy settings in JDeveloper, as described in Section 9.12.4.3, "Setting Browser Proxy Information."
- For information about creating data controls for application modules, see Chapter 13, "Using ADF Model in a Fusion Web Application." For information about other types of data controls, see Appendix H, "Data Controls in Fusion Web Applications."

The following chapters provide information about specific objects you can use in data controls:

- For information about using collections on a data control to create forms, see Chapter 26, "Creating a Basic Databound Page."
- For information about using collections to create tables, see Chapter 27, "Creating ADF Databound Tables."
- For information about using master-detail relationships to create UI components, see Chapter 29, "Displaying Master-Detail Data."
- For information about creating lists, see Chapter 30, "Creating Databound Selection Lists and Shuttles."
- For information about creating graphs, charts, and other visualization UI components, see Chapter 33, "Creating Databound ADF Data Visualization Components."

14.2 Creating Web Service Data Controls

The most common way of using web services in an application developed using Oracle ADF is to create a data control for an external web service. A typical reason for doing this is to add functionality that is readily available as a web service, but which would be time consuming to develop with the application, or to access an application that runs on a different architecture.

Additionally, you can reuse components created by Oracle ADF to make them available as web services for other applications to access.

14.2.1 How to Create a Web Service Data Control

JDeveloper allows you to create a data control for an existing web service using just the WSDL for the service. You can browse to a WSDL on the local file system, locate one in a UDDI registry, or enter the WSDL URL directly.

Note: If you are working behind a firewall and you want to use a web service that is outside the firewall, you must configure the web browser and proxy settings in JDeveloper. For more information, see Section 9.12.4.3, "Setting Browser Proxy Information."

Before you begin:

It may be helpful to have an understanding of how web service data controls are used in ADF applications. For more information, see Section 14.2, "Creating Web Service Data Controls."

You may also find it helpful to understand additional functionality that can be added using other web services features. For more information, see Section 14.1.2, "Additional Functionality for Web Services in Fusion Applications."

To create a web service data control:

1. In the Application Navigator, right-click an application and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **Web Services** and then **Web Service Data Control**, and click **OK**.
3. In the Create Web Service Data Control wizard, on the Data Source page, specify a name for the data control, a WSDL URL, and the specific web service to be accessed by the data control.
4. On the Data Control Operations page, select the operations you want the data control to support.
5. On the Response Format page, specify the format of the SOAP response.
6. On the Endpoint Authentication page, specify the authentication details for the endpoint URL, and click **Finish**.

14.2.2 How to Adjust the Endpoint for a Web Service Data Control

After developing a web service data control, you can modify the endpoint. This is useful, for example, when you migrate the application from a test environment to production.

Before you begin:

It may be helpful to have an understanding of how web service data controls are used in ADF applications. For more information, see Section 14.2, "Creating Web Service Data Controls."

You may also find it helpful to understand additional functionality that can be added using other web services features. For more information, see Section 14.1.2, "Additional Functionality for Web Services in Fusion Applications."

To change the endpoint for a web service data control:

1. In the Application Navigator, select the `.dcx` file for the web service data control.
2. In the Structure window, right-click the web service data control and choose **Edit Web Service Connection** from the context menu.
3. In the Edit Web Service Connection dialog, make the necessary changes to the endpoint URL and port name.
4. Click **OK**.

14.2.3 How to Refresh a Web Service Data Control

After creating a web service data control, you might find that a web service operation has changed in its method signature, return type, or structure. When this happens, you can update the data control without having to re-create it.

Before you begin:

It may be helpful to have an understanding of how web service data controls are used in ADF applications. For more information, see Section 14.2, "Creating Web Service Data Controls."

You may also find it helpful to understand additional functionality that can be added using other web services features. For more information, see Section 14.1.2, "Additional Functionality for Web Services in Fusion Applications."

To refresh an operation in a web service data control:

1. In the Application Navigator, select the `.dcx` file for the web service data control.
2. In the Structure window, right-click the desired web service operation and choose **Update** from the context menu.

JDeveloper queries the web service and updates the web service data control to reflect the current state of the selected operation.

14.2.4 What You May Need to Know About Web Service Data Controls

As with other kinds of data controls, you can design a databound user interface by dragging an item from the Data Controls panel and dropping it on a page as a specific UI component. For more information, see Section 13.4.1, "How to Use the Data Controls Panel."

In the Data Controls panel, each data control object is represented by an icon. Table 14–1 describes what each icon represents, where it appears in the Data Controls panel hierarchy, and what components it can be used to create.

Table 14–1 Data Controls Panel Icons and Object Hierarchy for Web Services

Icon	Name	Description	Used to Create...
	Data Control	Represents a data control. You cannot use the data control itself to create UI components, but you can use any of the child objects listed under it. Depending on how your web services are defined, there may be more than one data control. Typically, there is one data control for each web service. However, you may have additional data controls that were created for other types of business services (for example, application modules). For information about creating data controls for application modules, see Chapter 13, "Using ADF Model in a Fusion Web Application."	Serves as a container for other objects and is not used to create anything
	Collection	Represents a named data collection. A *data collection* represents a set of data objects (also known as a *row set*) in the data model. Each object in a data collection represents a specific structured data item (also known as a *row*) in the data model. Throughout this guide, *data collection* and *collection* are used interchangeably. For more information about using collections on a data control to create forms, see Chapter 26, "Creating a Basic Databound Page." For more information about using collections to create tables, see Chapter 27, "Creating ADF Databound Tables." For more information about using master-detail relationships to create UI components, see Chapter 29, "Displaying Master-Detail Data." For information about creating graphs, charts, and other visualization UI components, see Chapter 33, "Creating Databound ADF Data Visualization Components."	Forms, tables, graphs, trees, range navigation components, and master-detail components.
	Attribute	Represents a discrete data element in an object (for example, an attribute in a row). Attributes appear as children under the collections or method returns to which they belong. For information about using attributes to create fields on a page, see Section 26.2, "Using Attributes to Create Text Fields." For information about creating lists, see Chapter 30, "Creating Databound Selection Lists and Shuttles."	Label, text field, date, list of values, and selection list components.
	Structured Attribute	Represents a returned object that is not one of the Java primitive types (which are represented as attributes) and is also not a collection of any type. An example of a structured attribute would be a domain, which is a developer-created data type used to simplify application maintenance. For more information about domains, see Section 4.15, "Creating Custom, Validated Data Types Using Domains."	Label, text field, date, list of values, and selection list components
	Method	Represents an operation in the data control or one of its exposed structures that may accept parameters, perform some business logic and optionally return single value, a structure or a collection of those.	Command components For methods that accept parameters: command components and parameterized forms

Table 14–1 (Cont.) Data Controls Panel Icons and Object Hierarchy for Web Services

Icon	Name	Description	Used to Create...
	Method Return	Represents an object that is returned by a web service method. The returned object can be a single value or a collection. If a custom method returns anything at all, it is usually a single scalar value. However, some custom methods can return collections. A method return appears as a child under the method that returns it. The objects that appear as children under a method return can be attributes of the collection, other methods that perform actions related to the parent collection, and operations that can be performed on the parent collection. When a single-value method return is dropped, the method is not invoked automatically by the framework. You should either drop the corresponding method as a button to invoke the method, or if working with task flows you can create a method activity for it. For more information about executables, see Section 13.7.2.2, "Executable Binding Objects Defined in the Page Definition File."	The same components as for collections and attributes.
	Operation	Represents a built-in data control operation that performs actions on the parent object. Data control operations are located in an **Operations** node under collections or method returns, and also under the root data control node. The operations that are children of a particular collection or method return operate on those objects only, while operations under the data control node operate on all the objects in the data control. If an operation requires one or more parameters, they are listed in a **Parameters** node under the operation. The standard operations supported by the web service data control are for form navigation: First, Last, Next, and Previous. Because the web service data control is not an updateable data control, you cannot use built-in operations like commit, rollback, and execute.	UI command components, such as buttons, links, and menus. For more information, see Section 26.4, "Incorporating Range Navigation into Forms," and Section 26.5, "Creating a Form to Edit an Existing Record."
	Parameter	Represents a parameter value that is declared by the method or operation under which it appears. Parameters appear in the **Parameters** node under a method or operation. Array and structured parameters are exposed as updateable structured attributes and collections under the data control, which can be dropped as an ADF form or an updateable table on the UI. You can use the UI to build a parameter that is an array or a complex object (not a standard Java type).	Label, text, and selection list components.

14.3 Creating a New Web Service Connection

After developing a web service proxy, you can generate additional connections for the proxy that you can use in testing and deployment situations. For example, you might want to create a connection that includes user name and password for testing purposes.

14.3.1 How to Create a New Web Service Connection

The connection information is stored in the `connections.xml` file along with the other connections in your application. This abstraction of the endpoint URL also allows you to edit the connection after deployment using Enterprise Manager without requiring modification to the client code.

Before you begin:

It may be helpful to have an understanding of how web service connections are used in ADF applications. For more information, see Section 14.3, "Creating a New Web Service Connection."

You may also find it helpful to understand additional functionality that can be added using other data controls and web services features. For more information, see Section 14.1.2, "Additional Functionality for Web Services in Fusion Applications."

To create a new web service connection:

1. In the Application Navigator, right-click a web service proxy and choose **Create ADF Web Service Connection**.

   The New ADF Web Service Connection dialog displays the default settings for a connection associated with the selected proxy.

2. Modify the connection information as necessary, and click **OK**.

> **WARNING: If you create a new web service connection with the same name as an existing connection, the existing connection will be overwritten with the new information.**

After you create a new web service connection, you can modify your client to use this connection. You could use code similar to that shown in Example 14–1 to access the connection from your client.

Example 14–1 Accessing a Web Service Connection from a Client

```
Context ctx = ADFContext.getCurrent().getConnectionsContext();
WebServiceConnection wsc = (WebServiceConnection) ctx.lookup("MyAppModuleService");
MyAppModuleService proxy = wsc.getJaxWSPort(MyAppModuleService.class);
```

The argument that you pass to the `lookup()` method is the name that you gave to the web service connection. In this example, it is `MyAppModuleService`.

14.4 Securing Web Service Data Controls

Web services allow applications to exchange data and information through defined application programming interfaces. SSL (Secure Sockets Layer) provides secure data transfer over unreliable networks, but SSL only works point to point. Once the data reaches the other end, the SSL security is removed and the data becomes accessible in its raw format. A complex web service transaction can have data in multiple messages being sent to different systems, and SSL cannot provide the end-to-end security that would keep the data invulnerable to eavesdropping.

Any form of security for web services has to address the following issues:

- The authenticity and integrity of data
- Data privacy and confidentiality
- Authentication and authorization
- Non-repudiation
- Denial of service attacks

Throughout this section the "client" is the web service data control, which sends SOAP messages to a deployed web service. The deployed web service may be:

- A web service running on Oracle Application Server
- A web service running anywhere in the world that is accessible through the Internet

14.4.1 WS-Security Specification

The WS-Security specification unifies multiple security technologies to make secure web services interoperable between systems and platforms.

WS-Security addresses the following aspects of web services security issues:

- Authentication and authorization

  The identity of the sender of the data is verified, and the security system ensures that the sender has privileges to perform the data transaction.

  The type of authentication can be a basic username/password pair transmitted in plain text, or trusted X509 certificate chains. SAML assertion tokens can also be used to allow the client to authenticate against the service, or allow it to participate in a federated SSO environment, where authenticated details are shared between domains in a vendor-independent manner.

- Data authenticity, integrity, and non-repudation

  XML digital signatures, which use industry-standard messages, digest algorithms to digitally sign the SOAP message.

- Data privacy

  XML encryption that uses industry-standard encryption algorithms to encrypt the message.

- Denial of service attacks

  Defines XML structures to time-stamp the SOAP message. The server uses the time stamp to invalidate the SOAP message after a defined interval.

14.4.2 Using Key Stores

A web service data control can be configured for message-level security using key stores. For more information about creating and using key stores for message protection, see the section about managing keystores, wallets, and certificates in the *Oracle Fusion Middleware Administrator's Guide,* and the section about configuring policies in the *Oracle Application Server Web Services Security Guide*.

14.4.3 How to Define Web Service Data Control Security

After you create a web services data control in a JDeveloper project, you can define security for the data control using the Edit Data Control Policies dialog.

Before you begin:

It may be helpful to have an understanding of how security is used in web service data controls. For more information, see Section 14.4, "Securing Web Service Data Controls."

You may also find it helpful to understand additional functionality that can be added using other web services features. For more information, see Section 14.1.2, "Additional Functionality for Web Services in Fusion Applications."

To define security for a web service data control:

1. In the Application Navigator, select the web service data control `.dcx` file.

2. In the Structure window, right-click the web service data control, and choose **Define Web Service Security**.

   JDeveloper displays the Edit Data Control Policies dialog, which shows the Policy Store location. To select an alternative policy store, use the **WS Policy Store** page of the Preferences dialog.

3. From the **Ports** dropdown list, select the port to which you want the specified policies are applied.

4. From the **MTOM** dropdown list, select the MTOM (message transmission optimization mechanism) policy you want to use. If you leave this field blank, no MTOM policy is used.

5. From the **Reliability** dropdown list, select the reliability policy you want to use. If you leave this field blank, no reliability policy is used.

6. From the **Addressing** dropdown list, select the addressing policy you want to use. If you leave this field blank, no addressing policy is used.

7. In the **Security** list, you can optionally specify additional security policies to apply. To add a policy, click the Add security policy icon.

8. In the **Management** list, you can optionally specify additional management policies to apply. To add a policy, click the Add management policy icon.

9. If necessary, you can also remove policies from the **Security** list and the **Management** list by selecting the appropriate policy and clicking the corresponding delete icon.

10. You can optionally override properties for the policies in the **Security** list and the **Management** list by clicking **Override Properties**.

11. After you have selected the appropriate policies for your web service data control, click **OK** to apply your selections and close the dialog.

For more information about predefined policies and configuring policies and their properties, see the *Oracle Application Server Web Services Security Guide*.

15

Exposing URL Services Using the ADF Model Layer

This chapter explains how to expose URL services using the ADF Model layer.

This chapter includes the following sections:

- Section 15.1, "About Using ADF Model with URL Services"
- Section 15.2, "Exposing URL Services with ADF Data Controls"
- Section 15.3, "Using URL Service Data Controls"

15.1 About Using ADF Model with URL Services

A URL service can be simply a URL against which a query is posted, so that the URL can be exposed as an ADF form. For example, say you have a URL service that allows you to access employee data for your company. This data can be retrieved using a URL as shown in Example 15–1.

Example 15–1 Sample URL that Accesses a URL Service

```
http://example.com/getEmployee?empId=20+deptId=10
```

This simple URL that accesses employee data can become an ADF data control with a method (`getEmployee`) and two parameters (`empId` and `deptId`), that can then be dropped on a page as a form.

URL services can also utilize representational state transfer (REST) actions. REST services are web services that can be accessed using a simple HTTP URL, rather than a more complex SOAP protocol. The HTTP actions (`GET`, `PUT`, `POST`, `DELETE`) are mapped to service operations that access and manipulate data at the service implementation. The response data can be returned in delimiter separated value and XML formats, and you can specify an XSD URL to define the input format for the `PUT` and `POST` actions.

15.1.1 URL Services Use Cases and Examples

The REST architecture simplifies web service invocation by representing a web service as an HTTP resource, so that web service methods and operations look like a resources on the server that can be accessed through an HTTP URL.

For example, say a web service has a method called `getEmployee(int EmpID)`. Using the REST architecture, this can become `http://mywebservice.com/myService/getEmployee?EmpID=20`. When

represented as a plain HTTP URL, it is easy to use the URL service data control to quickly create a databound page that accesses this service.

15.1.2 Additional Functionality for URL Services

You may find it helpful to understand other data access features before you start working with URL services. Following are links to other functionality that may be of interest.

- You can design a databound user interface by dragging an item from the Data Controls panel and dropping it on a page as a specific UI component. For more information, see Section 13.4.1, "How to Use the Data Controls Panel."
- For information about creating data controls for application modules, see Chapter 13, "Using ADF Model in a Fusion Web Application."
- For information about creating web service data controls, see Chapter 14, "Exposing Web Services Using the ADF Model Layer."
- For information about other types of data controls, see Appendix H, "Data Controls in Fusion Web Applications."

15.2 Exposing URL Services with ADF Data Controls

The URL service data control enables you to access and consume the data stream from a specified URL. Each URL service data control represents a single operation of the service. For example, the `GET` operation and the `PUT` operation for a given URL service are represented by two different data controls. Typically, all of the actions for a given URL service will share a common URL connection.

15.2.1 How to Create a URL Connection

You use the Create URL Connection dialog to create a URL connection by supplying a name and the connection details required to access a URL endpoint.

Before you begin:

It may be helpful to have an understanding of how URL service data controls are used in ADF applications. For more information, see Section 15.1, "About Using ADF Model with URL Services."

You may also find it helpful to understand additional functionality that can be added using other URL services features. For more information, see Section 15.1.2, "Additional Functionality for URL Services."

You must have access to the URL service that the data control will access, and launch JDeveloper.

To create a URL connection:

1. From the main menu, choose **File > New**.
2. In the New Gallery, expand **General**, select **Connections** and then **URL Connection**, and click **OK**.
3. In the Create URL Connection dialog, select where to create the connection.

   Select **Application Resources** if you want the URL connection to be available only within the application. Select **Resource Palette** if you want the URL connection to be available from the Resource Palette for use in other applications.

4. Enter a name for the connection.
5. In the **URL Endpoint** field, enter the URL of the desired data stream.

   Typically, this includes just the host and port. Do not include any URL parameters (or example, `http://service.example.com:7101/`)
6. Select the level of authentication from the **Authentication Type** dropdown.

   None is the default authentication type and disables authentication. Use **Digest** when security is desired. In this way, the password will be transmitted across the network as an MD5 digest of the user's password and cannot be determined by sniffing network traffic. **Basic** authentication is primarily only useful when service access over the network does not require high security.
7. If digest or basic authentication is selected, specify the user name and password required to access the web site.
8. If the URL is associated with a protected area of the overall web site, enter the authentication realm.
9. After you have entered the name and endpoint, you can click Test Connection to verify URL connection is valid.
10. Click **OK** to create the connection.

15.2.2 How to Create a URL Service Data Control

You can create a URL Service data control using the Create URL Service Data Control wizard, which is available from the New Gallery. When you create URL service data controls, you use the wizard to create each operation, one at a time.

Before you begin:

It may be helpful to have an understanding of how URL service data controls are used in ADF applications. For more information, see Section 15.1, "About Using ADF Model with URL Services."

You may also find it helpful to understand additional functionality that can be added using other URL services features. For more information, see Section 15.1.2, "Additional Functionality for URL Services."

You must have access to the URL service that the data control will access.

You should have already created the URL connection you will use, as described in Section 15.2.1, "How to Create a URL Connection."

You must launch JDeveloper, and open the application to which you want to add the URL service data control.

To create a URL service data control:

1. In the Application Navigator, right-click the data model project and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **Data Controls** and then **URL Service Data Control**, and click **OK**.

Note: If you don't see the **URL Service Data Control** item in the New Gallery, click the **All Features** tab.

---

3. In the Create URL Service Data Control wizard (on the Data Source page), provide a name for the data control.

   Because each data control represents a single operation for the URL service, the name should reflect both the service and the operation. For example, you might name the data control for the `GET` operation of the `EmployeeData` URL service `emp_get`.

4. Select the URL connection for the data control to use.

   - If you have not yet created a URL connection, select **Create New Connection** from the **Connection** dropdown. You will specify the connection details and the action (HTTP method) for this data control on the connection page.
   - If you have already created a URL connection, select the connection you want to use from the **Connection** dropdown.

5. From the HTTP Method dropdown, choose the action for this data control.

6. Enter the source for the URL service (for example, `servicepath/servicename`), and click **Next**.

   If this operation requires a parameter, you can use the format `?symbol=##`*`ParamName`*`##` to specify it (for example, `servicepath/servicename?symbol=##id##`). You can also use a parameter to provide dynamic input for the source path (for example, `servicepath/##servicename##?symbol=##id##`).

7. On the Parameters page, supply default values for any parameters you specified, and click **Next**.

   For the `PUT` and `POST` operations, you must also provide a URL that points to the XML schema definition that defines the format of the input.

8. On the Data Format page, select the data format of the data source and set the associated properties, then click **Next**.

   You can choose either XML format (for which you provide URLs for the XSD and XSL) or delimiter separated value (for which you specify the delimiter, text qualifier, and encoding for the data).

9. On the Finish page, you can click **Test URL Connection** to verify that the URL data connection is valid, and click **Finish**.

15.2.3 What You May Need to Know About URL Service Data Controls

Because the URL Service data control is not updatable, there are limitations on some of the objects in the Data Controls panel. For example, the Create Method object is not available and the only operations available are retrieval and navigation. Also, in a URL Service data control, the parameter object is the parameter that a user passes in the URL For more information, see Table 15–1.

15.3 Using URL Service Data Controls

As with other kinds of data controls, you can design a databound user interface by dragging an item from the Data Controls panel and dropping it on a page as a specific UI component. For more information, see Section 13.4.1, "How to Use the Data Controls Panel."

In the Data Controls panel, each data control object is represented by an icon. Table 15–1 describes what each icon represents, where it appears in the Data Controls panel hierarchy, and what components it can be used to create.

Table 15–1 Data Controls Panel Icons and Object Hierarchy for the URL Service Data Control

Icon	Name	Description	Used to Create...
	Data Control	Represents a data control. You cannot use the data control itself to create UI components, but you can use the child objects listed under the data control. There may be more than one data control, each representing a logical grouping of data functions. Typically, there is one data control for a given source. However, you may have additional data controls that were created for other types of objects (for example, application modules or web services).	Serves as a container for the other objects. Not used to create anything.
	Method	Represents a custom method on the data control that may accept parameters, perform some action or business logic, and return data or data collections. If the method is a `get` method of a `map` and returns a data collection, a method return icon appears as a child under it. If a method requires a parameter, a folder appears under the method, which lists the required parameters. The URL Service data control exposes only one method `loadData()` which retrieves the contents of the URL.	UI actions such as buttons or links.
	Method Return	Represents a data collection that is returned by a custom method. A method return appears as a child under the method that returns it. The objects that appear as children under a method return may be attributes of the collection, accessor returns that represent collections related to the parent collection, other methods that perform actions related to the parent collection, and operations that can be performed on the parent collection.	Forms, tables, trees, and range navigation components.
	Accessor Return	Represents an object returned by a bean-style accessor method on the business service. An accessor method is used when the objects returned are JavaBeans. Accessor returns appear as children under method returns, other accessor returns, or in the **Attributes** node under built-in create methods. Accessor returns are objects that are related to the current object in the parent collection. This relationship is usually based on a common unique attribute in both objects. For example, if a method returns a collection of users, an accessor return that is a child of that collection might be a collection of service requests that are assigned to a particular user. In ADF, the relationship between parent and child collections is called a master-detail relationship. For more information about master-detail objects, see Chapter 29, "Displaying Master-Detail Data." Accessor returns can be either collections or single objects. For example, if a method returns a collection of service requests, one accessor return under that method might be a collection of service history details for the current service request, while another accessor return might be a single user assigned to the current service request. By default, when data controls are created from session beans over POJOs, the names of accessors that return collections end in `Collection` (for example, `OrderCollection`). The UI components available from the Data Controls panel context menu differ depending on whether the accessor return is a collection or a single object. The children under an accessor return may be attributes of the collection or object, other accessor returns, custom methods that return a value from the collection or object, and operations that can be performed on the collection or object. The accessor returns under a built-in create method are always a single object and never have any children.	For collections: Forms, tables, trees, range navigation components, and master-detail widgets. For single objects: Forms, master-detail widgets, and selection lists. For single objects under a constructor: selection lists only.

Table 15–1 (Cont.) Data Controls Panel Icons and Object Hierarchy for the URL Service Data Control

Icon	Name	Description	Used to Create...
	Attribute	Represents a discrete data element in an object. Attributes appear as children under method returns or accessor returns.	Label, text field, and selection list components.
	Operation	Represents a built-in data control operation that performs actions on the parent object. If an operation requires a parameter, a folder appears under the method, which lists the required parameters. Data control operations are located in an **Operations** node under method returns or accessor returns and under the root data control node. The operations that are children of a particular method or accessor return operate on that return object only, while operations under the data control node operate on all the objects represented by the data control. Because the URL Service data control is not updateable, only retrieval and navigation operations are available.	UI actions such as buttons or links.
	Parameter	Represents a parameter value that is declared by the method or operation under which it appears. Parameters appear in a folder under a method or operation. The parameter for a URL Service data control is the parameter that a user passes in the URL. These show up as a parameters to the `loadData()` method when the URL Service data control is created. For example, say you create a data control to the URL `http://www.example.org?id=##param##`. On the Data Controls panel, you would see that the `loadData()` method has one parameter with the name `param`. The value supplied to this parameter is substituted in the URL when the invocation occurs.	Label, text, and selection list components.

16

Using Validation in the ADF Model Layer

This chapter explains how to use validation rules in the model layer of your applications.

This chapter contains the following sections:

- Section 16.1, "About ADF Model Layer Validation"
- Section 16.2, "Defining Validation Rules in the ADF Model Layer"
- Section 16.3, "Customizing Error Handling"

16.1 About ADF Model Layer Validation

In the model layer, ADF Model validation rules can be set for a binding's attribute on a particular page. When a user edits or enters data in a field and submits the form, the bound data is validated against any set rules and conditions. If validation fails, the application displays an error message.

Note that you don't need to add additional ADF Model validation if you have already set validation rules in the business domain layer of your entity objects. In an ADF Business Components-based Fusion web application, you won't need to use ADF Model validation unless you use data controls other than your application module data controls.

16.1.1 ADF Model Layer Validation Use Cases and Examples

If your application uses data controls that are not based on ADF Business Components, you can use ADF Model layer validation to ensure the quality of user-entered data.

Best Practice: Use business layer validation whenever possible. However, the following are examples of when you might need to use model layer validation:

- When your business layer does not support declarative validation.
- When you want to validate before an expensive roundtrip begins (for example, for a binding to a web service).
- When you need to validate a form created using parameters (for example, using the `executeWithParams` operation).

You can also use client-side validation for cases where you want to validate before a roundtrip to the server. For more information, see the "Adding Validation" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Many of the declarative validation features available for ADF Business Components objects are also available at the model layer, should your application warrant the use of model-layer validation in addition to business-layer validation.

16.1.2 Additional Functionality for ADF Model Layer Validation

You may find it helpful to understand other ADF validation features before you use model layer validation. Following are links to other functionality that may be of interest.

- When you use the ADF Business Components application module data control, you do not need to use model-layer validation. Consider defining all or most of your validation rules in the centralized, reusable, and easier to maintain entity objects of your business layer. For more information, see Chapter 7, "Defining Validation and Business Rules Declaratively."
- Many of the declarative validation features available at the page level are also available at the bean level, which you would implement on the data control structure file. This can be very useful, because validation rules on the data control structure file apply to all usages of the data control. For information about implementing validation rules on the data control structure file, see the "Adding Business Logic to Data Controls" chapter in the *Oracle Fusion Middleware Java EE Developer's Guide for Oracle Application Development Framework*.

16.2 Defining Validation Rules in the ADF Model Layer

You can configure ADF Model validation for a binding's attributes on the page definition file. Validation rules in the model layer are executed for a binding's attribute on a particular page when the containing form is submitted.

You can optionally set the `skipValidation` property to `true` to bypass the ADF Model validation. You can set `skipValidation` to `skipDataControls` to validate the bound objects without validating the transaction. For example, set `skipValidation` to `skipDataControls` if you have a table action that opens a popup window to accept data entries and you want to allow the view layer to validate those entries before the commit on the table. The `skipValidation` property can be found in the Property Inspector after you have selected the root node of the page definition file in the Structure window.

16.2.1 How to Add Validation

You set ADF Model validation on the page definition file. You define the validation rule, and set an error message to display when the rule is broken.

Table 16–1 describes the ADF Model validation rules that you can configure for a binding's attributes.

Table 16–1 ADF Model Validation Rules

Validator Rule Name	Description
Compare	Compares the attribute's value with a literal value
List	Validates whether or not the value is in a list of values
Range	Validates whether or not the value is within a range of values
Length	Validates the value's character or byte size against a size and operand (such as greater than or equal to)

Table 16–1 (Cont.) ADF Model Validation Rules

Validator Rule Name	Description
Regular Expression	Validates the data using Java regular expression syntax
Required	Validates whether or not a value exists for the attribute

Before you begin:

It may be helpful to have an understanding of the use of validation rules in the model layer. For more information, see Section 16.2, "Defining Validation Rules in the ADF Model Layer."

You may also find it helpful to understand additional functionality that can be added using other validation features. For more information, see Section 16.1.2, "Additional Functionality for ADF Model Layer Validation."

You will need to complete this task:

- Create a component on the page. The component must have binding attributes.

To create an ADF Model validation rule:

1. Open the page definition that contains the binding for which you want to create a rule.
2. In the Structure window, select the attribute, list, or table binding.
3. In the Property Inspector, select **More** and then **Edit Validation Rule**.
4. In the Edit Validation Rules dialog, expand the binding node, select the attribute name, and click **New**.
5. In the Add Validation Rule dialog, select a validation rule and configure the rule as needed.
6. Select the **Failure Handling** tab and configure the message to display when the rule is broken.

16.2.2 What Happens at Runtime: Model Validation Rules

When a user submits data, as long as the submitted value is a non-null value or a string value of at least one character, then all validators on a component are called one at a time. Because the `f:validator` tag on the component is bound to the `validator` property on the binding, any validation routines set on the model are accessed and executed.

The process then continues to the next component. If all validations are successful, the Update Model Values phase starts and a local value is used to update the model. If any validation fails, the current page is redisplayed along with an error message.

16.3 Customizing Error Handling

You can report errors using a custom error handler that extends the default `DCErrorHandlerImpl` class. You are not required to write any code to register your custom exception handler class. Instead, you select the root node of the `DataBindings.cpx` file in the Structure window, and then use the Property Inspector to set the `ErrorHandlerClass` property to the fully qualified name of the error handler you want it to use.

Your custom error handler can contain the following overridable methods:

- `reportException()`: Called to report any exception that occurs. It can be overridden to analyze reported exceptions.
- `getDisplayMessage()`: Returns the message that will be reported to JSF for each error that occurs. Returning `null` is the way your custom error handler signals that a given exception should not be reported to the client.
- `getDetailedDisplayMessage()`: Returns the detail portion of the message as a `String` object or HTML that will be reported to JSF for each error that occurs. Returning `null` is the way your custom error handler signals that a given exception should not be reported to the client.
- `skipException()`: Returns a boolean depending on whether you want to display each item from the nested exception in the final error list displayed to the user. This method override lets you implement logic to check for specifics exception types and, based on the business scenario, determine whether to display it in the list.

Example 16–1 illustrates a custom error handler that extends the `DCErrorHandlerImpl` class and shows the override for the `skipException()` method that is needed to skip exceptions that should not appear in the list displayed to the user.

Example 16–1 Custom Error Handler

```
package view.controller.fwkext;

import java.sql.SQLIntegrityConstraintViolationException;

import java.util.ArrayList;
import java.util.List;

import oracle.adf.model.binding.DCBindingContainer;
import oracle.adf.model.binding.DCErrorHandlerImpl;

import oracle.jbo.CSMessageBundle;
import oracle.jbo.DMLConstraintException;
import oracle.jbo.JboException;

public class CustomErrorHandler extends DCErrorHandlerImpl {

   List<ExceptionMapper> exceptionMapperList = new ArrayList<ExceptionMapper>();
   public CustomErrorHandler() {
     this(true);
   }

   public CustomErrorHandler(boolean setToThrow) {
     super(setToThrow);
     exceptionMapperList.add(new DisableJboExceptionCodesMapper());
   }

   public void reportException(DCBindingContainer bc, Exception ex) {
     for (ExceptionMapper mapper : exceptionMapperList) {
       if (mapper.canMapException(ex)) {
         ex = mapper.mapException(ex);
       }
     }
     super.reportException(bc, ex);
   }

   /**
```

```
    * If an exception is a RowValException or a TxnValException and they
    * have nested exceptions, then do not display it. This example shows
    * an implementation that skips the SQLIntegrityConstraintViolationException
    * from displaying in the error final list displayed to the user.
    */
   @Override
   protected boolean skipException(Exception ex) {

      if (ex instanceof DMLConstraintException) {
            return false;
        } else if (ex instanceof SQLIntegrityConstraintViolationException) {
            return true;
        }
        return super.skipException(ex);
    }

}
```

You must change the constructor to `MyErrorHandler()`. The exception error handler must have a default constructor, as shown in Example 16–2.

Example 16–2 Default Constructor

```
ErrorHandlerClass="viewcontroller.MyErrorHandler"
 public MyErrorHandler()
 {
   super(true);
 }
```

16.3.1 How to Customize the Detail Portion of a Message

If you plan to customize and use the detail portion of a message, you can create a custom error handler and implement the `getDetailedDisplayMessage` method to retrieve and process that message. The finalized message will be passed to the view layer to be integrated with other messages.

To customize the detail portion of a message:

1. Create a custom error handler class that extends the default `DCErrorHandlerImpl` class.
2. In that class, override the `getDetailedDisplayMessage` method that returns a `DCErrorMessage` object.

   Example 16–3 shows an implementation of the `getDetailedDisplayMessage` method in the custom error handler class.

Example 16–3 Custom Error Handler Class with getDetailDisplayMessage Method

```
public final class MyErrorMessageHandler extends DCErrorHandlerImpl {
    public MyErrorMessageHandler (){
        super(false);
    }
    public DCErrorMessage getDetailedDisplayMessage(BindingContext ctx,
                                                    RegionBinding ctr,
                                                    Exception ex) {

        ...
        return new MyDCErrorMesssage(ctr, ex);
    }
}
```

3. Create a custom class that implements the `DCErrorMessage` interface. The class must implement the `getHTMLText` method and the `getText` method.

   You will add code to the `getHTMLText` method to perform the actual processing, but you must also implement `getText` to satisfy the interface requirements.

4. In the `getHTMLText` implementation, add code to create and process the error message.

   `getHTMLText` of `getDetailedDisplayMessage` should return the finalized version of the error message as an HTML fragment that will be inserted into the HTML code of the page. For this reason, you should perform all necessary preprocessing on the text message before the message is returned by `getDetailedDisplayMessage`. For instance, you may want to retrieve the localized version of the message or change the right-to-left ordering of the message before it is returned.

   Example 16–4 shows an implementation of this interface.

Example 16–4 Implementing the DCErrorMessage Interface

```
public final class MyDCErrorMesssage implements DCErrorMessage {
    RegionBinding m_regionBinding;
    Exception m_ex;
       public MyDCErrorMesssage(RegionBinding ctr, Exception ex) {
        super();
        this.m_regionBinding = ctr;
        this.m_ex = ex;
    }
    public String getText() {
        ...
        return "Message String";
    }
    public String getHTMLText() {
        ...
        /* Add code to process the message, including localization */
        /* and right-to-left directional requirements. */
        /* Return the message as the finalized HTML fragment.*/
        return "<b>error</b> message details";
    }
}
```

16.3.2 How to Write an Error Handler to Deal with Multiple Threads

Oracle ADF constructs an instance of the custom error handler for each `BindingContext` object that is created. Because Oracle ADF serializes simultaneous web requests from the same logical end-user session, multiple threads generally will not use the same error handler at the same time. However, to guarantee a thread-safe custom error handler, use the `setProperty()` API on `JboException`. This method stores in the exception objects themselves any hints you might need later during the phase when exceptions are translated to JSF `FacesMessage` objects for display.

17

Designing a Page Using Placeholder Data Controls

This chapter describes how to create and use placeholder data controls. It shows you how to create placeholder data types, including master-detail relationships. It also describes how to create and import sample data.

This chapter includes the following sections:

- Section 17.1, "About Placeholder Data Controls"
- Section 17.2, "Creating Placeholder Data Controls"
- Section 17.3, "Creating Placeholder Data Types"
- Section 17.4, "Using Placeholder Data Controls"

17.1 About Placeholder Data Controls

Application development is typically divided into two separate processes: technical implementation and user interface design. More often than not, they are undertaken by separate teams with very different skill sets. The two teams can work together either in a *data-first* approach or a *UI-first* approach, or with some overlap between the two. With either approach, the teams usually work together iteratively, refining the application with each cycle.

In a data-first approach, the model, or data control is built first. Then the designer creates the layout and page flow by dragging and dropping the data controls onto pages as UI components. The model data is automatically bound to the components. This approach requires the data model to be available before the designer can proceed.

In a UI-first approach, the designer creates the layout using components from the Component Palette. When the data controls do become available, UI components are then bound to them. With this approach, you should be able to see most of the layout and page flows to make a development evaluation. However, until the data controls are available and bound to components, the application may not fully convey the intent of its design. For instance, an application that has a master-detail relationship is best reviewed when there is actual data that dynamically drives that relationship.

Placeholder data controls are easy-to-create, yet fully functional, stand-in data controls that can efficiently speed up the design-development process. UI designers can use placeholder data controls to create page layouts and page flows without the need to have real data controls available. These placeholder controls can be loaded with sample data to realistically simulate application execution for design evaluations. When the real data controls are ready, the UI components can be easily rebound to complete the application.

Creating placeholder data controls is a purely declarative process and does not require coding. It does not require in-depth knowledge of the underlying model, data source technology, actual database schema, or any of the complex relationships in an actual production data control. Placeholder data controls do not require an existing data source or a database connection. You can define multiple data types with multiple attributes. You can also define nested master-detail hierarchies between data types. Placeholder data controls have the same built-in operations such as Execute, Next, and Create. An implicitly created named criteria item allows the user to create search forms as if view objects and view criteria were available.

17.1.1 Placeholder Data Controls Use Cases and Examples

For many complex applications, the UI design may actually drive the development of the model, or data source. In this UI-first scenario, having placeholder data controls with sample data is essential to properly model the behavior of the application. In some cases, even if production data controls are available, UI designers may opt to use placeholder data controls because of their flexibility and ease of use.

Placeholder data controls can be used in many situations. In addition to being used for design review and development, they can be used to develop realistic runtime mock-ups for usability studies, or for proof-of-concept requirements. They can be used to create demos when the data model is not yet ready.

17.1.2 Additional Functionality for Placeholder Data Controls

You may find it helpful to understand some data access features before you start working with placeholder data controls. Following are links to other functionality that may be of interest.

- After your initial design with placeholder data controls, when the final data controls are available, you can simply rebind the components. For more information about rebinding components, see Chapter 26, "Creating a Basic Databound Page" and Chapter 27, "Creating ADF Databound Tables."
- A placeholder data type attribute can be configured to be a list of values (LOV). For more information about LOVs, see Section 5.12, "Working with List of Values (LOV) in View Object Attributes."
- You can create master-detail relationships between placeholder data types, similar to the master-detail data collections in a standard data control. For more information on master-detail forms and tables, see Chapter 29, "Displaying Master-Detail Data."
- When you define placeholder data types in a master-detail hierarchy, JDeveloper creates view links that define that relationship. For more information about view links, see Section 5.6, "Working with Multiple Tables in a Master-Detail Hierarchy."
- Placeholder data controls can be used for the development of search forms. For more information about query search forms, see Chapter 31, "Creating ADF Databound Search Forms."
- You might want to package placeholder data controls into reusable components as ADF Library JARs. For more information about reusable components and the ADF Library, see Chapter 38, "Reusing Application Components."

17.2 Creating Placeholder Data Controls

You add placeholder data controls to a project using the New Gallery. After the placeholder data control has been created, it appears as a node in the Data Controls panel. It has a different icon than do standard data controls. Instead of an Operations node, the placeholder data control has a Built-in Operations node. Although the Built-in Operations node contains Commit and Rollback operations, these operations do not perform commits or rollbacks because there is not an actual data source for the data.

When a data control is initially created, it does not have any data types associated with it. You will need to manually create the data types as described in section Section 17.3, "Creating Placeholder Data Types."

17.2.1 How to Create a Placeholder Data Control

Placeholder data controls are defined at the project level in JDeveloper. You must already have created a project before you can create placeholder data controls.

Before you begin:

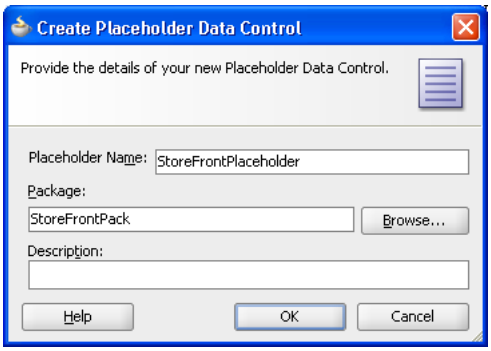
It may be helpful to have an understanding of the options you have for creating placeholder data controls. For more information, see Section 17.2, "Creating Placeholder Data Controls."

You may also find it helpful to understand additional functionality that can be added after using placeholder data controls. For more information, see Section 17.1.2, "Additional Functionality for Placeholder Data Controls."

To create a placeholder data control:

1. In the Application Navigator, right-click the project and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **Data Controls** and then **Placeholder Data Control**, and click **OK**.
3. In the Placeholder Data Control dialog, as shown in Figure 17–1, enter:
   - **Placeholder Name**: The name of the placeholder data control.
   - **Directory Name**: The package name that will be used to reference the placeholder data control.
   - **Description**: Optional description of the placeholder data control.

Figure 17–1 New Placeholder Data Control



4. Click **OK**.

17.2.2 What Happens When You Create a Placeholder Data Control

When you create a placeholder data control, the package you selected to contain the data control appears under the project node in the Application Navigator. A data control XML file *PlaceholderDataControl*.xml appears under the package, where *PlaceholderDataControl* is the name of the placeholder data control. Example 17–1 shows a sample file called StoreFrontPlaceHolder.xml, which was created when the StoreFrontPlaceHolder data control was created.

Example 17–1 Sample placeholderdatacontrol.xml

```
<?xml version='1.0' encoding='windows-1252' ?>
<AppModule
   xmlns="http://xmlns.oracle.com/placeholder"
   Name="StoreFrontPlaceholder" >
</AppModule>
```

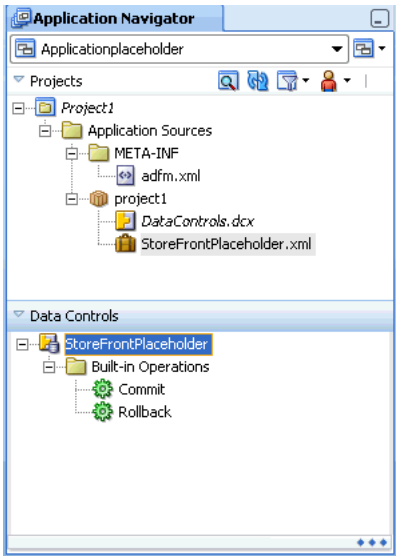
JDeveloper also creates a DataControls.dcx file if it has not yet been defined, and adds entries for the placeholder data control, as shown in Example 17–2.

Example 17–2 Placeholder Data Control entry in DataControls.dcx

```
<?xml version="1.0" encoding="UTF-8" ?>
<DataControlConfigs xmlns="http://xmlns.oracle.com/adfm/configuration"
                    version="11.1.1.44.30" id="DataControls"
                    Package="storefront">
  <PlaceholderDataControl SupportsTransactions="true" SupportsFindMode="true"
                          SupportsResetState="true" SupportsRangesize="true"
                          SupportsSortCollection="true"
                          FactoryClass=
                            "oracle.adf.model.placeholder.DataControlFactoryImpl"
                          id="StoreFrontPlaceholder"
                          xmlns="http://xmlns.oracle.com/adfm/datacontrol"
                          Definition="storefront.StoreFrontPlaceholder"
                          Package="storefront"/>
</DataControlConfigs>
```

In the Data Controls panel, the placeholder data control appears alongside other data controls in the root tree. A placeholder data control that does not yet have data types defined will have only the Commit and Rollback built-in operations available, as shown in Figure 17–2.

Figure 17–2 Application Navigator and Data Controls Panel



17.3 Creating Placeholder Data Types

A standard data control obtains its data collections and attributes from its underlying data source in the model or business service layer. For example, an application module data control obtains its data collections from the view objects and associated database tables.

For a placeholder data control, instead of data collections, it has placeholder data types. A *placeholder data type* is analogous to a data collection. It can be dropped onto a page to create complex components such as forms, tables, and trees. It also has a set of attributes that can be dropped onto pages as individual components such as input text, output text, and select choice. Some attributes may be defined as LOVs.

When you first create a placeholder data control, it is devoid of any data types because there are no underlying database tables for the placeholder data control to reference. You must declaratively create one or more placeholder data types. For each data type, you specify attribute names, types, default UI components, and other options. You can create multiple data types for a data control, similar to the multiple data collections in an application module.

After you have created a placeholder data type, it appears as a child node of the placeholder data control. It also has a Built-in operations node with the standard set of operations. It has a Named Criteria node that contains an All Queriable Attributes item that is analogous to the named view criteria of a view object in a standard data control. You can drag and drop the All Queriable Attributes item onto a page to create a query or quick query search form. In a standard data control, you can create multiple view criteria on a view object. Because there is no real view object in a placeholder data type, only one All Queriable Attributes item is available. For more information about query search forms, see Chapter 31, "Creating ADF Databound Search Forms."

You can create master-detail relationships between placeholder data types, similar to the master-detail data collections in a standard data control. You can drop master-detail data types onto pages to create master-detail forms and tables. For more information on master-detail forms and tables, see Chapter 29, "Displaying Master-Detail Data."

JDeveloper allows you to reuse placeholder data types created for other placeholder data controls in the same project. When you are creating a data type, you can select an option to load existing data types from another placeholder data control. If you select the **Copy Data Type** option, the attributes from the imported data type will be added to the list of existing attributes.

You can also select an option to copy the sample data associated with the imported data type when the attributes are added.

Note: Although you do not need sample data until you run the application, you should add sample data to provide a consistent design time rendering of the components.

17.3.1 How to Create a Placeholder Data Type

After you have created a placeholder data control, you can proceed to create data types. You define a name for the data type, and define each of its individual attributes. For each attribute, you then define its type, format, default UI component, and whether it should be an LOV.

In order to simplify the process of creating placeholder data types, you can select from a list of four of the most common types: `String`, `Boolean`, `Date`, and `Number`. Because placeholder attributes are typed, you can create column labels and include UI control hints in the design.

If you have a sample data file in comma-separated value or CSV format, you can automatically create all the attributes and load sample data using the sample data file import function. You do not need to create the attributes. JDeveloper will automatically create them for you from the format of the CSV file, which should be a comma-separated value list of column headings. The attributes default to type `String`. You can manually reset each attribute to another type as required. For instructions to import sample data, see Section 17.3.6, "How to Add Sample Data."

Before you begin:

It may be helpful to have an understanding of the options you have for creating placeholder data types. For more information, see Section 17.3, "Creating Placeholder Data Types."

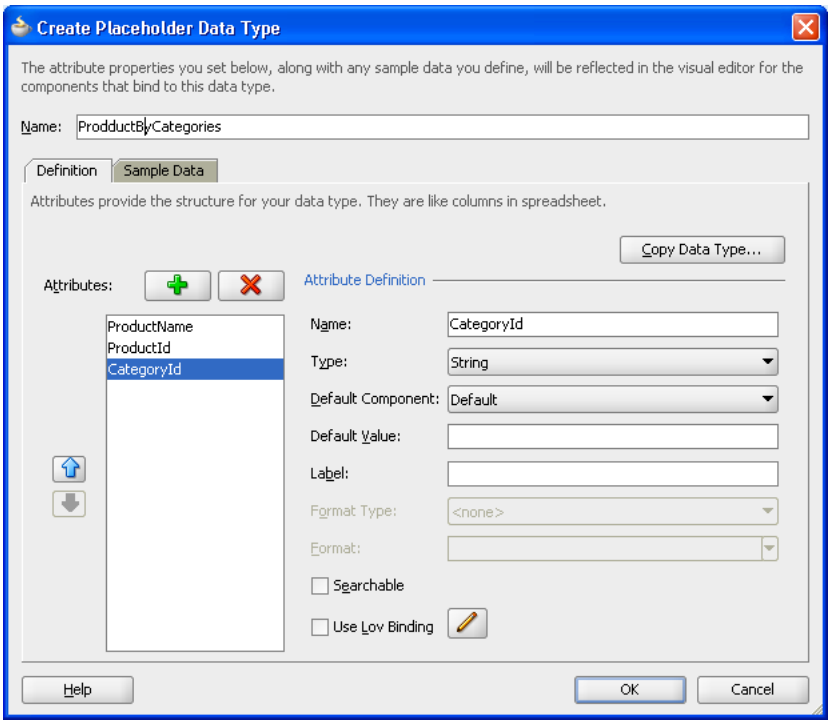
You may also find it helpful to understand additional functionality that can be added after using placeholder data controls. For more information, see Section 17.1.2, "Additional Functionality for Placeholder Data Controls."

To create a placeholder data type manually:

1. In the Data Controls panel, right-click the placeholder data control and choose **Create Placeholder Data Type**.

   Figure 17–3 shows the Create Placeholder Data Type dialog.

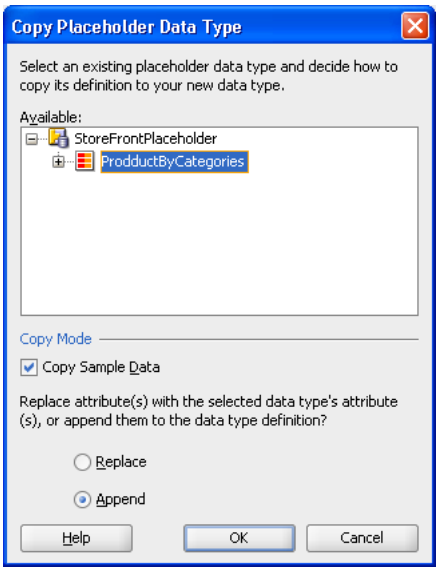
Figure 17–3 Create Placeholder Data Type Dialog



Note: If you had already added placeholder data types previously and want only to add or edit them, choose **Edit Placeholder Data Type** from the context menu instead. The dialog that appears will be the Edit Placeholder Data Type dialog. It has the same options as the Create Placeholder Data Type dialog.

2. If you already have data types defined for another placeholder data control, and you want to copy or append them, click **Copy Data Type**.
   a. In the Copy Placeholder Data Type dialog, as shown in Figure 17–4, select the placeholder data type you want. Select **Replace** to replace the current attributes with the attributes from the file, or select **Append** to add the attributes from the file to the list of current attributes.
   b. Select the **Copy Sample Data** checkbox to load sample data from the file.
   c. Click **OK**.

Figure 17–4 Copy Placeholder Data Type



3. In the Create Placeholder Data Type dialog, enter a name for the placeholder data type, and then in the **Attributes Definition** section, enter:

   - **Name**: Enter a name for the attribute.
   - **Type**: Select a type for the attribute from the dropdown list. The supported types are `String`, `Boolean`, `Date`, and `Number`.
   - **Default Component**: Select a default component for the attribute from the dropdown list. For an LOV, select **Combo Box List of Values**.
   - **Default Value**: Enter the initial value for the attribute.
   - **Label**: Enter a label for the attribute. The label will be used when the component is displayed.
   - **Format Type**: This field is enabled only when the type is `Date` or `Number`. Select a format type from the dropdown list.
   - **Format**: This field is enabled only when a format mask has been defined for that format type.
   - **Searchable**: Select this checkbox to make the attribute searchable.
   - **Use Lov Binding**: Select this checkbox if you want the attribute to be an LOV. To configure the attribute, see Section 17.3.3, "How to Configure a Placeholder Data Type Attribute to Be an LOV."

   Click the **Add** icon to add more attributes.

4. To add data, use the **Sample Data** tab. For that procedure, see Section 17.3.6, "How to Add Sample Data."

   You need sample data for runtime and for a consistent design time.

5. Click **OK**.

17.3.2 What Happens When You Create a Placeholder Data Type

When you create a placeholder data type, JDeveloper creates a *PlaceholderDataType*.xml file, where *PlaceholderDataType* is the name of the placeholder data type you had specified.

The *PlaceholderDataType*.xml file has the same format as a view object XML file. It includes the name of the view object and the name and values of each placeholder attribute that was defined.

Example 17–3 shows a *PlaceholderDataType*.xml for the Supplier data type. Two attributes were declarative defined: Supplier_Id and Supplier_Name.

Example 17–3 Sample Placeholder Data Type Suppliers.xml file

```
<?xml version='1.0' encoding='windows-1252' ?>
<ViewObject
   xmlns="http://xmlns.oracle.com/placeholder"
   Name="Suppliers"
   BindingStyle="OracleName"
   CustomQuery="true"
   ComponentClass="oracle.adf.model.placeholder.PlaceholderVOImpl"
   UseGlueCode="false" >
   <ViewAttribute
      Name="Supplier_Id"
      Type="oracle.jbo.domain.Number"
      PrimaryKey="true" >
   </ViewAttribute>
   <ViewAttribute
      Name="Supplier_Name"
      Type="java.lang.String" >
   </ViewAttribute>
   <StaticList
      Name="Suppliers"
      Rows="2"
      Columns="2" >
   </StaticList>
   <ResourceBundle >
      <PropertiesBundle
         xmlns="http://xmlns.oracle.com/bc4j"
         PropertiesFile="storefrontproject.StoreFrontProjectBundle" >
      </PropertiesBundle>
   </ResourceBundle>
</ViewObject>
```

Since a data type is similar to a data collection and is based on a view object, each data type will have a corresponding *PlaceholderDataType*.xml file.

JDeveloper also adds entries for each placeholder data type to the *PlaceholderDataControl.xml* file. For example, after the Suppliers data type has been created, the StoreFrontPlaceholder.xml file includes a new ViewUsage entry for the Suppliers data type, as shown in Example 17–4.

Example 17–4 Sample PlaceholderDataControl.xml File After Addition of Placeholder Data Type

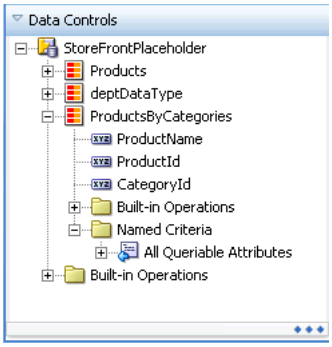
```
<?xml version='1.0' encoding='windows-1252' ?>
<AppModule
   xmlns="http://xmlns.oracle.com/placeholder"
```

```
   Name="StoreFrontPlaceHolder" >
   <ViewUsage
      Name="Suppliers"
      ViewObjectName="storefrontproject.Suppliers" >
   </ViewUsage>
</AppModule>
```

In the Data Controls panel, a placeholder data type node appears under the placeholder data control. Expanding the node reveals the presence of each of the attributes, the Built-in Operations node, and the Named Criteria node.

Figure 17–5 shows a placeholder data control as it appears in the Data Controls panel.

Figure 17–5 Data Controls Panel Showing Placeholder Data Control



17.3.3 How to Configure a Placeholder Data Type Attribute to Be an LOV

A placeholder data type attribute can be configured to be a list of values (LOV). An LOV-formatted attribute binds to UI components that display dropdown lists or list picker dialogs. For more information about LOVs, see Section 5.12, "Working with List of Values (LOV) in View Object Attributes."

When you are creating a placeholder data type, you can select an option to bring up a dialog to configure that attribute to be an LOV.

If you have only one data source, you can only create a fixed LOV. To create a dynamic LOV, there must be more than one placeholder data type available to be the source.

17.3.3.1 Configuring an Attribute to Be a Fixed LOV

Before you begin, you should determine which attribute you want to be a fixed LOV and which values should be in the fixed list.

Before you begin:

It may be helpful to have an understanding of the options you have for creating placeholder data types. For more information, see Section 17.3, "Creating Placeholder Data Types."

You may also find it helpful to understand additional functionality that can be added after using placeholder data controls. For more information, see Section 17.1.2, "Additional Functionality for Placeholder Data Controls."

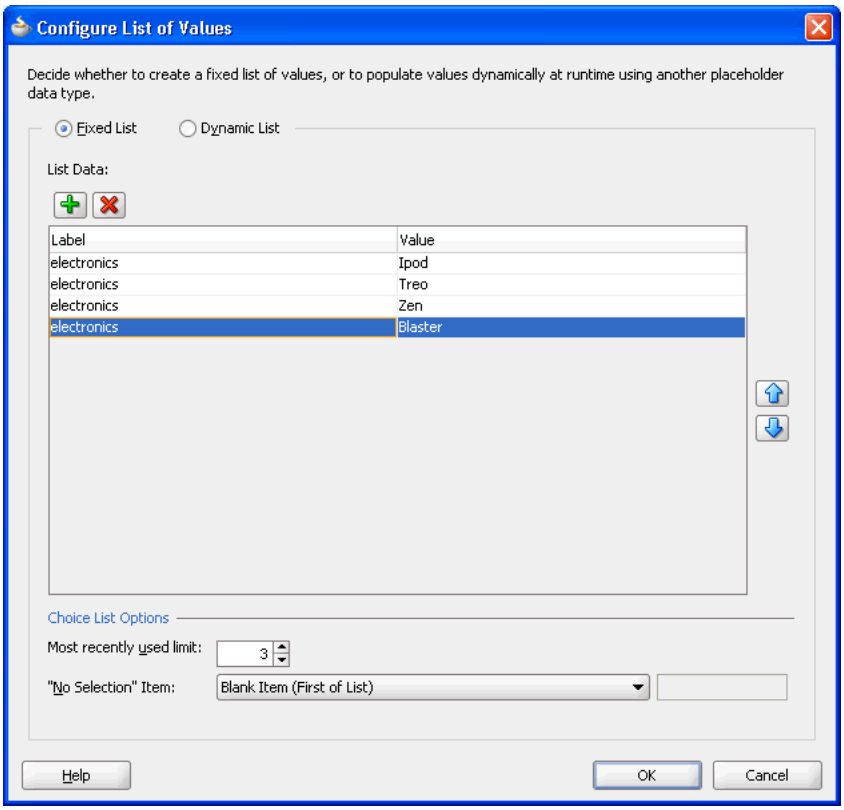
To configure an attribute to be a fixed LOV:

1. In the Data Controls panel, right-click the placeholder data control and choose **Create Placeholder Data Type** or **Edit Placeholder Data Type**.

2. In the Create Placeholder Data type or Edit Placeholder Datatype dialog, select the **Use Lov Binding** checkbox.

   The Configure List of Values dialog appears, as shown in Figure 17–6.

Figure 17–6 Configure List of Values Dialog for a Fixed LOV



3. In the dialog, select **Fixed List**.
4. Click the **Add** icon to enable adding an entry to the list of values.
5. For each entry, enter a label and a value.

   When the user selects an item from the list of values, the value entry will be entered into the input field.
6. Select the maximum number of the most recently used items that will be displayed in the dropdown list.
7. From the **No Selection Item** dropdown list, select an option for how you want the "no selection" item to be displayed.

   For instance, selecting **Blank Item (First of List)** will display the "no selection" item as a blank at the beginning of the list.
8. Click **OK**.

17.3.3.2 Configuring an Attribute to Be a Dynamic LOV

Using a placeholder data type to serve as the source, you can configure an attribute to be a dynamic LOV.

Before you begin:

It may be helpful to have an understanding of the options you have for creating placeholder data types. For more information, see Section 17.3, "Creating Placeholder Data Types."

You may also find it helpful to understand additional functionality that can be added after using placeholder data controls. For more information, see Section 17.1.2, "Additional Functionality for Placeholder Data Controls."

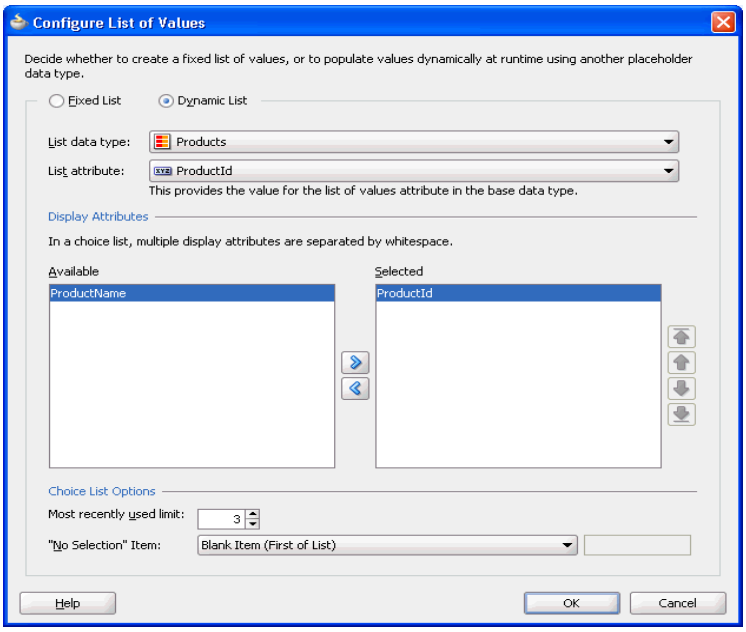
Also, you should have already created another placeholder data type to serve as the source of the dynamic LOV.

To configure an attribute to be a dynamic LOV:

1. In the Data Controls panel, right-click the placeholder data control and choose **Create Placeholder Data Type** or **Edit Placeholder Data Type**.
2. In the Create Placeholder Data type or Edit Placeholder Data type dialog, select the **Use Lov Binding** checkbox.

   The Configure List of Values dialog appears, as shown in Figure 17–7.

Figure 17–7 Configure List of Values Dialog for a Dynamic LOV



3. In the dialog, select **Dynamic List**.
4. Select the list data type with the source attribute. You must have a source placeholder data type for this selection to be available.
5. Select the list attribute.
6. Shuttle the attribute from the **Available** list to the **Selected** list.
7. Select the maximum number of the most recently used items that will be displayed in the dropdown list.
8. From the **No Selection Item** dropdown list, select an option for how you want the "no selection" item to be displayed. For instance, selecting **Blank Item (First of List)** will display the "no selection" item as a blank at the beginning of the list.

9. Click **OK**.

17.3.4 How to Create Master-Detail Data Types

You create master-detail relationships between data types in the same way you create master-detail hierarchies between tables. In a standard data control, you can use view links to define source and target view objects that would become the master and the detail objects. For more information about master-detail relationships, see Chapter 29, "Displaying Master-Detail Data."

You first create a master data type and its attributes. Then you create a detail data type as a child of the master data type. You define the source attribute in the master data type that defines the relationship to the detail data type.

Before you begin:

It may be helpful to have an understanding of the options you have for creating placeholder data types. For more information, see Section 17.3, "Creating Placeholder Data Types."

You may also find it helpful to understand additional functionality that can be added after using placeholder data controls. For more information, see Section 17.1.2, "Additional Functionality for Placeholder Data Controls."

Before you create a master-detail hierarchy, you must determine the data structure of the master data type and the data structure of the detail data type. You must also determine which attribute in the master will be the source for the detail data type.

To create master-detail hierarchical data types:

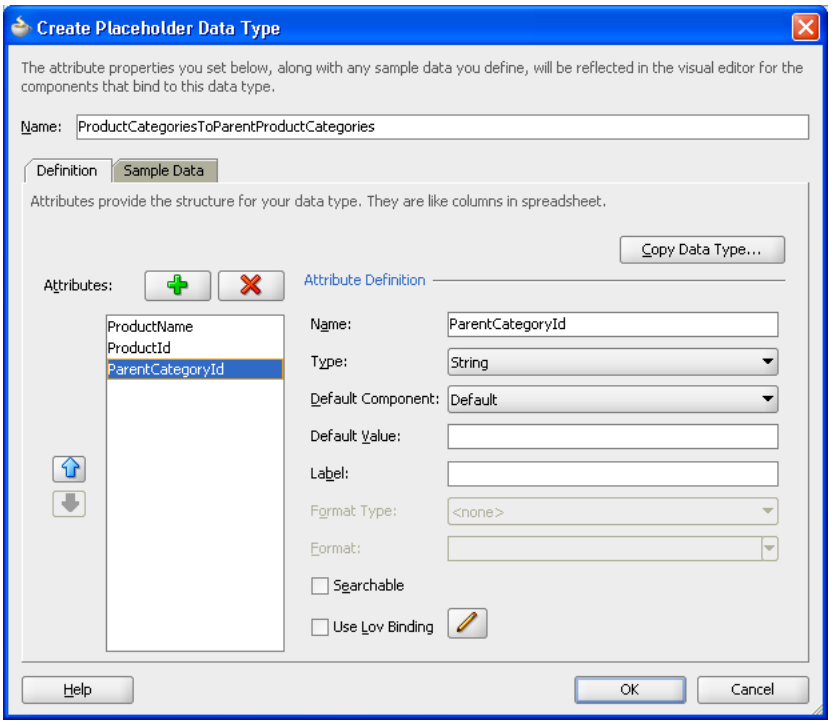
1. Create a placeholder data type to be the master as described in Section 17.3.1, "How to Create a Placeholder Data Type," or select an existing data type to be the master.

   For example, you could use the `ProductsByCategories` data type as the master.

2. In the Data Controls panel, right-click the master placeholder data type and choose **Create Placeholder Data Type**.

   Figure 17–8 shows the Create Placeholder Data Type dialog for entering detail data type attributes.

Figure 17–8 Create Placeholder Data Type Dialog

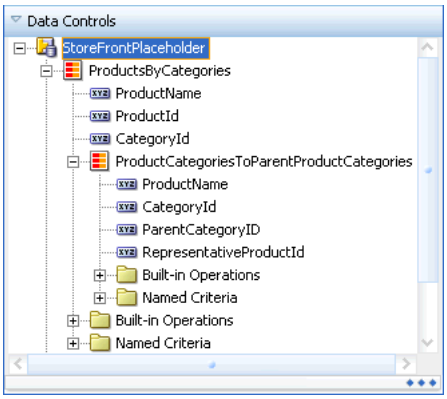


3. In the Create Placeholder Data Type dialog, enter a name for the detail data type.
4. The first attribute in the master data type appears in the **Attributes** section. This attribute provides the foreign key relationship.

   Add attributes for the detail data type, copy data type attributes from another data type, or create attributes automatically by importing sample data from a CSV file. For the procedure to add attributes, see Section 17.3.1, "How to Create a Placeholder Data Type."
5. Click **OK**.

The Data Controls panel should display the detail data type as a child of the master data type, as shown in Figure 17–9.

Figure 17–9 Master Detail Hierarchy in Placeholder Data Control



17.3.5 What Happens When You Create a Master-Detail Data Type

A master-detail relationship is implemented in the same way as is a standard master-detail relationship, using view object and view links. When you define placeholder data types in a master-detail hierarchy, JDeveloper creates a `DTLink.xml` file that contains metadata entries for view links that define that relationship. For more information about view links, see Section 5.6, "Working with Multiple Tables in a Master-Detail Hierarchy." For example, in the relationship between the master data type `Video` and the detail data type `Brand` associated with a key `dvdplayer`, JDeveloper creates a DTLink.xml file in the form of a view link file to define that relationship, as shown in Example 17–5.

Example 17–5 DTLink.xml file for Master-Detail Data Type Relationships

```
<?xml version='1.0' encoding='windows-1252' ?>
<ViewLink
   xmlns="http://xmlns.oracle.com/placeholder"
   Name="DTLink" >
   <ViewLinkDefEnd
      Name="sourceEnd"
      Cardinality="1"
      Owner="project1.ProductsByCategories"
      Source="true" >
      <AttrArray Name="Attributes">
         <Item Value="project1.ProductsByCategories.ProductName" />
      </AttrArray>
   </ViewLinkDefEnd>
   <ViewLinkDefEnd
      Name="destEnd"
      Cardinality="-1"
      Owner="project1.ProductCategoriesToParentProductCategories" >
      <AttrArray Name="Attributes">
         <Item
Value="project1.ProductCategoriesToParentProductCategories.ProductName" />
      </AttrArray>
   </ViewLinkDefEnd>
</ViewLink>
```

17.3.6 How to Add Sample Data

If you intend to run an application using the placeholder data control, you will need to add sample data for execution. You can add sample data to the placeholder data type attributes manually or by importing the data from a CSV file. Although having sample data is mandatory only at runtime, you should add sample data for a consistent design time rendering of the components.

Before you begin to add sample data to a placeholder data type, you should have already created a placeholder data control and a placeholder data type. If you are entering the data manually, you should have the data ready. If you are loading the data from a CSV file, you need to have the location of the file.

17.3.6.1 Adding Sample Data Manually

You can use the Sample Data page of the Edit Placeholder Data Type dialog to manually enter sample data for your placeholder data control.

Before you begin:

It may be helpful to have an understanding of the options you have for creating placeholder data types. For more information, see Section 17.3, "Creating Placeholder Data Types."

You may also find it helpful to understand additional functionality that can be added after using placeholder data controls. For more information, see Section 17.1.2, "Additional Functionality for Placeholder Data Controls."

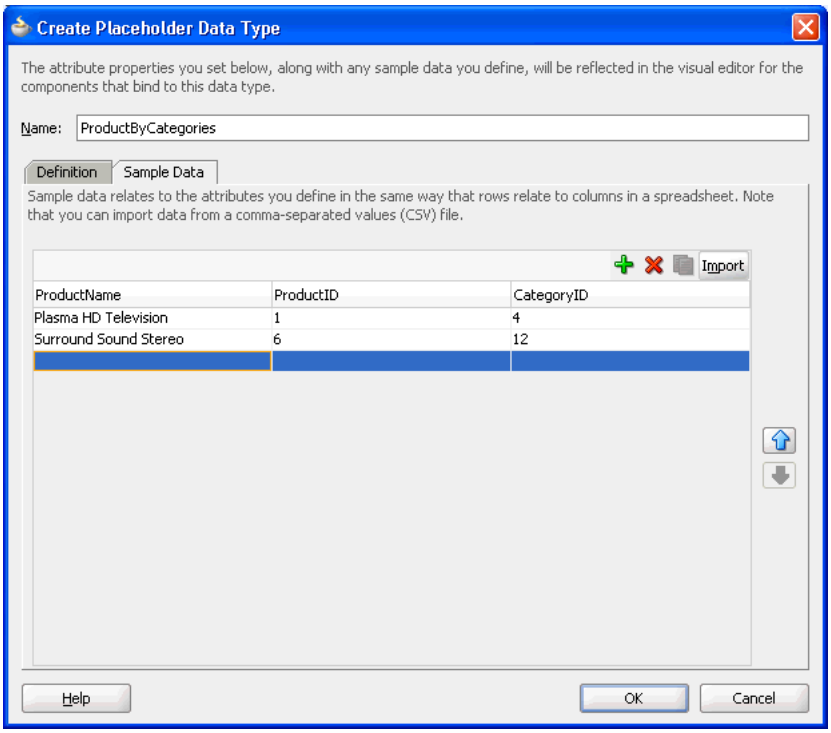
Also, you should have already prepared the sample data.

To add sample data to placeholder data types manually:

1. In the Data Controls panel, right-click the placeholder data control and choose **Create Placeholder Data Type** or **Edit Placeholder Data Type**.
2. In the Create Placeholder Data Type or Edit Placeholder Data Type dialog, click the **Sample Data** tab.
3. For each row of data, enter a value for each attribute that was defined. Click the **Add** icon to create each row.

   For example, in Figure 17–10, for the first row, `Plasma HD Television` was entered for the `ProductName` attribute, `1` was entered for the `ProductID` attribute, and `4` was entered for the `CategoryID` attribute.

Figure 17–10 Adding Sample Data



4. Click **OK**.

17.3.6.2 Importing Sample Data

You can use the Sample Data page of the Edit Placeholder Data Type dialog to import sample data for your placeholder data control from a file.

Before you begin:

It may be helpful to have an understanding of the options you have for creating placeholder data types. For more information, see Section 17.3, "Creating Placeholder Data Types."

You may also find it helpful to understand additional functionality that can be added after using placeholder data controls. For more information, see Section 17.1.2, "Additional Functionality for Placeholder Data Controls."

Additionally, you should have the sample data file available on your file system so it can be found in the Select File dialog.

To import sample data from CSV files into placeholder data types:

1. In the Data Controls panel, right-click the placeholder data control and choose **Create Placeholder Data Type** or **Edit Placeholder Data Type**.
2. In the Create Placeholder Datatype or Edit Placeholder Datatype dialog, click the **Sample Data** tab.

   > **Tip:** If you already have a CSV file for importing, you do not need to manually create the attributes for each column. JDeveloper automatically creates the attributes from the first row of the CSV file. For more information, see Section 17.3.1, "How to Create a Placeholder Data Type."

3. If you are also importing attributes, you must delete the default "attribute" in the first row.

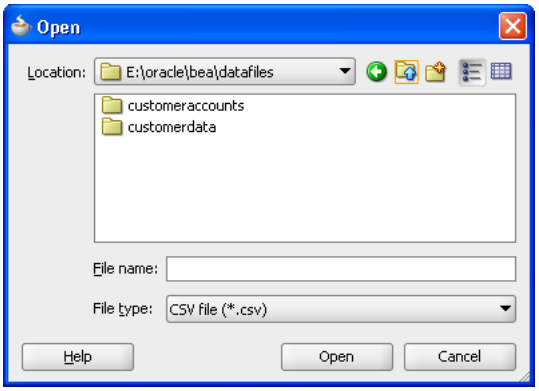
   This default attribute appears when you first navigate to the **Sample Data** tab. If you do not remove this default attribute, JDeveloper will assume that this is a declaratively created attribute and will not import any other columns except for the first column.

4. Click **Import**.

   In the Open dialog, navigate to and select the import file, and click **Open**, as shown in Figure 17–11.

   The data from the CSV file, including column heading and values, should appear as sample data.

Figure 17–11 Importing Placeholder Sample Data



5. Click **OK**.

17.3.7 What Happens When You Add Sample Data

Placeholder sample data, whether manually added using the dialog or from an imported CSV file, is stored in message bundle properties files within the placeholder data control packages. JDeveloper creates a text-based file for each data type that has sample data. The properties file name is *placeholderdatatypename*`MsgBundle.properties`. Example 17–6 shows a sample data properties file for the `Televisions` data type that has three attributes of `brand`, `size`, and `type`.

Example 17–6 Sample Data Properties File TelevisionMsgBundle.properties

```
SL_0_0=sony
SL_0_1=42
SL_0_2=lcd
SL_1_0=panasonic
SL_1_1=50
SL_1_2=plasma
SL_2_0=mitsubishi
SL_2_1=60
SL_2_2=projection
```

17.4 Using Placeholder Data Controls

You use placeholder data controls in the same way you would use standard data controls. You can drag data types onto pages and use the context menus to drop the data types as forms, tables, trees, graphs, and other components. You can drop individual attributes onto pages as text, lists of values, single selections, and other components. You can use any of the built-in operations such as Create, Execute, and Next by dropping them as buttons, command links, and menu items.

You can work in several ways to take advantage of placeholder data controls:

- Build a page using the placeholder data controls and rebind to real data controls later.
- Build a page using components, bind them to placeholder data controls, and rebind to real data controls later.
- Build a page using some combination of components from the Component Palette, components from the placeholder data controls, and then bind or rebind to the real data controls later.

17.4.1 Limitations of Placeholder Data Controls

You can use placeholder data controls in your application development in many situations. For most UI design evaluations, placeholder data controls should be able to fully substitute for real data controls.

There are a few limitations:

- Because data types are not tied to an underlying data source, the Commit and Rollback operations do not perform real transactions, nor do they update the cache.
- Placeholder data controls can only be created declaratively. You cannot create custom methods like you can with a real application module or data control.

- Placeholder data controls cannot be used when there is a need either for custom data or for filtering or custom code to fetch data. Placeholder data controls will disable those operations.

17.4.2 Creating Layout

Use the drag-and-drop feature to create the page using the placeholder data controls, any available real data controls, and components from the Component Palette. If you intend to run a page or application that requires real data, enter sample data for your placeholder data types. If you have a large amount of sample data, you may be able to create CSV files from the data source and load them into the data type. You may also use spreadsheets and other tools to create CSV sample data files.

17.4.3 Creating a Search Form

In a standard data control, you can create view criteria on view objects to modify the query. These view criteria are also used for drag-and-drop creation of query and quick query search forms. The named view criteria items appear under the Named Criteria node for a data collection. For more information about query and quick query search forms, see Chapter 31, "Creating ADF Databound Search Forms."

For placeholder data controls, there is also a Named Criteria node under each data type node. An automatically created All Queriable Attributes item appears under this node and can be used to drag and drop onto pages to create the query or quick query search forms.

17.4.4 Binding Components

Instead of building the page using the data controls, for instance, if you are unsure of the shape of your data, you can lay out the page first using the Component Palette and later bind it to the data types, attributes, or operations of the placeholder data controls.

17.4.5 Rebinding Components

After the final data controls are available, you can simply rebind the components. You can select the component in the Structure window and use the context menu to open the relevant rebind dialog. You can also drag and drop the data control item onto the UI component to initiate a rebinding editor. The rebinding procedures are the same whether the component was originally bound to a placeholder data control or a standard data control.

For more information about rebinding components, see Chapter 26, "Creating a Basic Databound Page" and Chapter 27, "Creating ADF Databound Tables."

17.4.6 Packaging Placeholder Data Controls to ADF Library JARs

A useful feature of placeholder data controls is that they allow parallel development and division of labor among developers and designers. You may be able to leverage that further by packaging placeholder data controls into reusable components as ADF Library JARs. ADF Libraries are JARs that have been packaged to contain all the necessary artifacts of an ADF component. For more information about reusable components and the ADF Library, see Chapter 38, "Reusing Application Components." You can create libraries of placeholder data controls and distribute them to multiple designers working on the same UI project. Because they are lightweight, you can even use them in place of available real data controls for the earlier phases of UI design.

Part IV

Creating ADF Task Flows

Part III contains the following chapters:

- Chapter 18, "Getting Started with ADF Task Flows"
- Chapter 19, "Working with Task Flow Activities"
- Chapter 20, "Using Parameters in Task Flows"
- Chapter 21, "Using Task Flows as Regions"
- Chapter 22, "Creating Complex Task Flows"
- Chapter 23, "Using Dialogs in Your Application"

18

Getting Started with ADF Task Flows

This chapter introduces ADF task flows. ADF task flows enable navigation, encapsulation, reuse, managed bean lifecycles, and transactions within a Fusion web application. The chapter also includes the basic steps for creating a task flow by adding activities and control flows to the task flow diagram, plus testing and running the finished task flow.

This chapter includes the following sections:

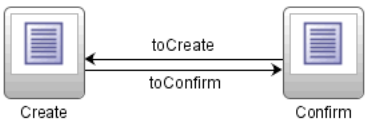
- Section 18.1, "About ADF Task Flows"
- Section 18.2, "Creating a Task Flow"
- Section 18.3, "Adding Activities to a Task Flow"
- Section 18.4, "Adding Control Flow Rules to Task Flows."
- Section 18.5, "Testing Task Flows"
- Section 18.6, "Refactoring to Create New Task Flows and Task Flow Templates"
- Section 18.7, "What You Should Know About Task Flow Constraints"

18.1 About ADF Task Flows

ADF task flows provide a modular approach for defining control flow in a Fusion web application. Instead of representing an application as a single large JSF page flow, you can break it up into a collection of reusable task flows. Each task flow contains a portion of the application's navigational graph. The nodes in the task flows are activities. An *activity node* represents a simple logical operation such as displaying a page, executing application logic, or calling another task flow. The transitions between the activities are called *control flow cases*.

Figure 18–1 shows two view activities called `Create` and `Confirm`. These view activities are similar to page nodes within a JSF page flow.

Figure 18–1 ADF Task Flow



The are two types of ADF task flow:

- Unbounded task flow: A set of activities, control flow rules, and managed beans that interact to allow a user to complete a task. The unbounded task flow consists of all activities and control flows in an application that are not included within a bounded task flow.
- Bounded task flow: A specialized form of task flow that, in contrast to the unbounded task flow, has a single entry point (an entry point is a view activity that can be directly requested by a browser) and zero or more exit points. It contains its own set of private control flow rules, activities, and managed beans. A bounded task flow allows reuse, parameters, transaction management, reentry, and can render within an ADF region in a JSF page.

  For a description of the activity types that you can add to an unbounded or bounded task flow see Chapter 19, "Working with Task Flow Activities".

  By default, JDeveloper proposes the following filename for the source file of a bounded task flow:

  `task-flow-definition`*`N`*`.xml`

  where *N* is a number that increments each time that you create a new bounded task flow. You can choose a filename for the bounded task flow. For example, you might use `checkout-task-flow.xml` if the purpose of the bounded task flow is to allow customers check out from a shopping application.

  The source file contains the metadata for the bounded task flow. Multiple bounded task flows can be included within the same source file using the following metadata:

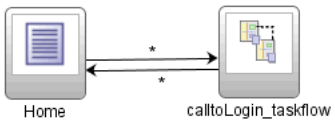
```
<task-flow-definition id="TaskFlowID1">
    ...
  </task-flow-definition>
<task-flow-definition id="TaskFlowID2">
   ...
</task-flow-definition>
```

  However, as JDeveloper's visual editors show one bounded task flow per file, we do not recommend that you include more than one bounded task flow within the same source file.

A typical application is a combination of the unbounded task flow and one or more bounded task flows. For example, JDeveloper, by default, creates an empty unbounded task flow (source file name is `adfc-config.xml`) when you create an application using the Fusion Web Application template. At runtime, the Fusion web application can call bounded task flows from activities that you added to the unbounded task flow.

As shown in Figure 18–2, the first activity to execute in an application is often a view activity within the unbounded task flow. A view activity represents a JSF page that displays as part of the application. The activity shown in Figure 18–2 starts with the `Home` view activity and then calls a bounded task flow. The `calltoLogin_taskFlow` activity calls a bounded task flow that enables a user to log into the application.

Figure 18–2 Unbounded Task Flow Calling a Bounded Task Flow



You can also design an application in which all application activities reside within the unbounded flow. This mimics a JSF application, but does not take advantage of bounded task flow functionality. To take full advantage of task flow functionality, use bounded task flows.

Table 18–1 describes the advantages that the control flow provided by ADF task flows offers over the control flow offered by JSF page flow. In certain scenarios you may need to use JSF page flow. For example, using phase listeners, as described in Section 25.4, "Customizing the ADF Page Lifecycle," involves using the JSF page flow's `faces-config.xml` configuration file. In general, it is not recommended that you mix JSF page flow and ADF task flows in your application.

Table 18–1 ADF Task Flow Advantages

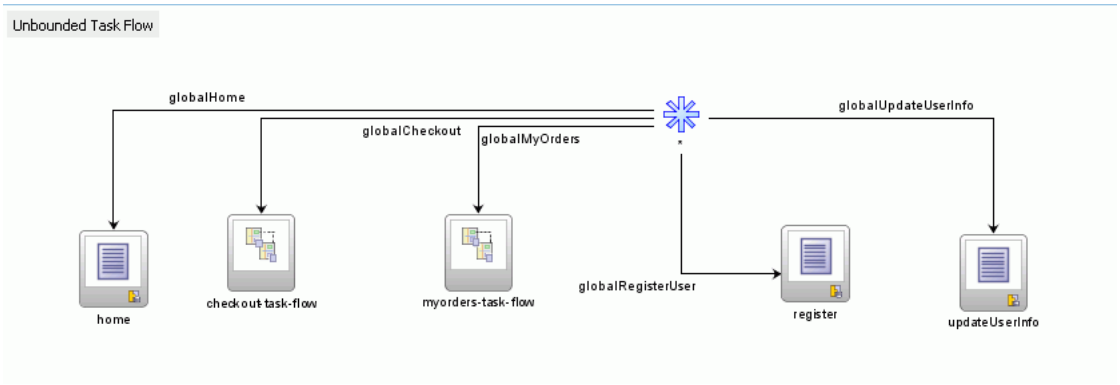
JSF Page Flow	ADF Task Flow
The entire application must be represented in a single page navigation file (`faces-config.xml`). Although you can have multiple copies of `faces-config.xml` in a project, the application loads these files as one at runtime.	The application can be broken up into a series of modular flows that call one another.
All nodes within a JSF page flow must be JSF pages. No other types of objects can exist within the JSF page flow.	You can add to the task flow diagram nodes such as views, method calls, and calls to other task flows.
Navigation is only between pages.	Navigation is between pages as well as other activities, including routers. For more information, see Section 19.4, "Using Router Activities".
Application fragments cannot be reused.	ADF task flows are reusable within the same or an entirely different application. After you break up your application into task flows, you may decide to reuse task flows containing common functionality. For more information see Chapter 38, "Reusing Application Components".
There is no shared memory scope between multiple requests except for session scope.	Shared memory scope (for example, page flow scope) enables data to be passed between activities within the task flow. Page flow scope defines a unique storage area for each instance of a bounded task flow.

18.1.1 About Unbounded Task Flows

A Fusion web application always contains an *ADF unbounded task flow*, which contains the entry point or points to the application. An entry point is a view activity that can be directly requested by a browser. A Fusion web application only has one unbounded task flow. By default, the source file for the unbounded task flow is the

`adfc-config.xml` file. Although you can create additional source files for unbounded task flows, the application combines all source files at runtime into the `adfc-config.xml` file. Figure 18–3 displays the diagram for the unbounded task flow from the Fusion Order Demo Application. This task flow contains a number of view activities that are all entry points to the application.

Figure 18–3 Unbounded Task Flow in Fusion Order Demo Application



You typically use the unbounded task flow instead of a bounded task flow if:

- You want to take advantage of ADF Controller features not offered by bounded task flows, such as bookmarkable view activities. For more information, see Section 19.2.2, "Bookmarking View Activities".
- The task flow will not be called by another task flow.
- The application has multiple points of entry. In Figure 18–3, the task flow can be entered through any of the pages represented by the view activity icons on the unbounded task flows.
- You want to bookmark more than one activity on the task flow. See Section 19.2.2, "Bookmarking View Activities" for more information.

The unbounded task flow cannot declaratively specify parameters. In addition, it cannot contain a *default activity*, an activity designated as the first to run in the unbounded task flow. This is because the unbounded task flow does not have a single point of entry. To perform any of these requires a bounded task flow.

In order to take advantage of completely declarative ADF Controller transaction and reentry support, use a bounded task flow rather than the unbounded task flow.

18.1.2 About Bounded Task Flows

An *ADF bounded task flow* is used to encapsulate a reusable portion of an application. A bounded task flow is similar to a Java method in that it:

- Has a single entry point
- May accept input parameters
- May generate return values
- Has its own collection of activities and control flow rules
- Has its own memory scope and managed bean lifespan (a page flow scope instance)

For more information about memory scopes, see Section 18.2.4, "What You May Need to Know About Memory Scope for Task Flows."

- Can determine whether or not it has an isolated data control frame

  For more information about access to data control frames, see Section 20.4, "Sharing Data Controls Between Task Flows."

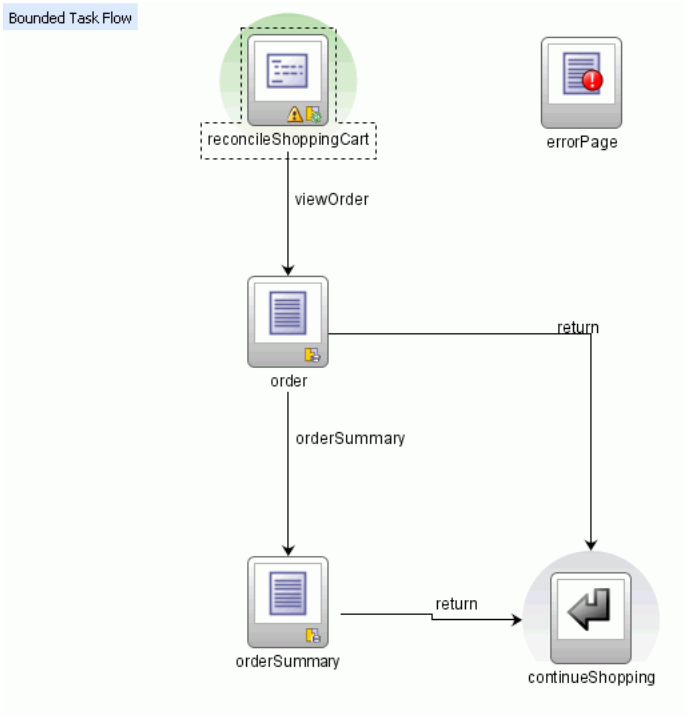
- Can define an activity as an exception handling activity

  For more information about exception handling in bounded task flows, see Section 22.4, "Handling Exceptions in Task Flows."

The `checkout-task-flow` activity in Figure 18–3 is a call to a bounded task flow. The unbounded task flow can call a bounded task flow, but cannot be called by another task flow. A bounded task flow can call another bounded task flow, which can call another and so on. There is no limit to the depth of the calls.

The checkout process is created as a separate bounded task flow, as shown in Figure 18–4.

Figure 18–4 Checkout Bounded Task Flow in Fusion Order Demo Application



The reasons for creating the `checkout-task-flow` activity as a called bounded task flow are:

- The bounded task flow always specifies a default activity, a single point of entry that must execute immediately upon entry of the bounded task flow.

  In the checkout task flow, the activity labeled `reconcileShoppingCart` invokes a method that returns a list of items that an anonymous user (one who has not yet logged in to the application) may have chosen to purchase. Any items chosen before authentication are included in the shopping cart after the user has logged in. Because it is the default activity, the method is always invoked before the shopping cart order page displays.

- `checkout-task-flow` is reusable. For example, it can be included in other applications requiring an item checkout process. The bounded task flow can also be reused within the same application.
- Any managed beans you decide to use within `checkout-task-flow` can be specified in page flow scope, so are isolated from the rest of the application. These managed beans (with page flow scope) are automatically released when the task flow completes.

The main features of bounded task flows are summarized in Table 18–2.

Table 18–2 Bounded Task Flow Features

Feature	Description
Well-defined boundary	A bounded task flow consists of its own set of private control flow rules, activities, and managed beans. A caller requires no internal knowledge of such things as page names, method calls, child bounded task flows, managed beans, and control flow rules within the bounded task flow boundary. Input parameters can be passed into the bounded task flow, and return values can be passed out on exit of the bounded task flow. Data controls can be shared between task flows.
Single point of entry	A bounded task flow has a single point of entry, a default activity that executes before all other activities in the task flow. For more information, see Section 18.2.3, "What You May Need to Know About the Default Activity in a Bounded Task Flow".
Page flow memory scope	You can specify page flow scope as the memory scope for passing data between activities within the bounded task flow. Page flow scope defines a unique storage area for each instance of a bounded task flow. Its lifespan is the bounded task flow, which is longer than request scope and shorter than session scope. For more information, see Section 18.2.4, "What You May Need to Know About Memory Scope for Task Flows".
Addressable	You can access a bounded task flow by specifying its unique identifier within the XML source file for the bounded task flow and the file name of the XML source file. For more information, see Section 19.6.8, "What Happens When You Add a Task Flow Call Activity".
Reuse	You can identify an entire group of activities as a single entity, a bounded task flow, and reuse the bounded task flow in another application within an ADF region. For example, the Hot Items and Start Shopping tabs on the home page of the Fusion Order Demo application reuse the same task flow embedded in a region. Different parameters are passed to each region to determine the lists of products that display. For more information, see Section 21.3, "Specifying Parameters for an ADF Region." You can also reuse an existing bounded task flow simply by calling it. For example, one task flow can call another bounded task flow using a task flow call activity or a URL. In addition, you can use task flow templates to capture common behaviors for reuse across different bounded task flows. For more information, see Section 22.8, "Creating Task Flow Templates".
Parameters and return values	A caller can pass input parameters to a bounded task flow and accept return values from it. For more information, see Section 20.3, "Passing Parameters to a Bounded Task Flow". In addition, you can share data controls between bounded task flows. For more information, see Section 20.4, "Sharing Data Controls Between Task Flows".

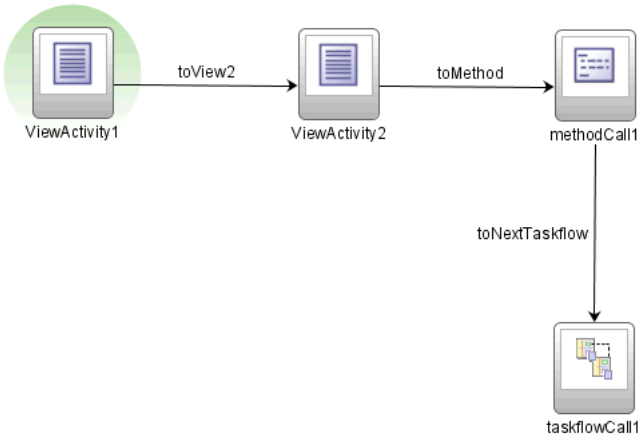
Table 18–2 (Cont.) Bounded Task Flow Features

Feature	Description
Transaction management	A bounded task flow can represent a transactional unit of work. You can declaratively specify options on the bounded task flow that determine whether, when entering the task flow, the task flow creates a new transaction, joins an existing one or is not part of the existing transaction. For more information, see Section 22.2, "Managing Transactions in Task Flows".
Reentry	You can specify options on the bounded task flow that determine whether or not it can be reentered. For more information, see Section 22.3, "Reentering Bounded Task Flows".
On-demand loading of metadata	Bounded task flow metadata is loaded on demand when entering a bounded task flow.
Security	You can secure a bounded task flow by defining the privileges that are required for someone to use it. For more information, see Section 35, "Enabling ADF Security in a Fusion Web Application."

18.1.3 About Control Flows

A task flow consists of activities and control flow cases that define the transitions between activities. Figure 18–5 shows a control flow rule, labeled `toView2`, that defines the transition between the `ViewActivity1` and `ViewActivity2` view activities. The `ViewActivity1` view activity displays before the ViewActivity2 view activity when the task flow in Figure 18–5 executes.

Figure 18–5 Task Flow with Activities and Control Flow Cases



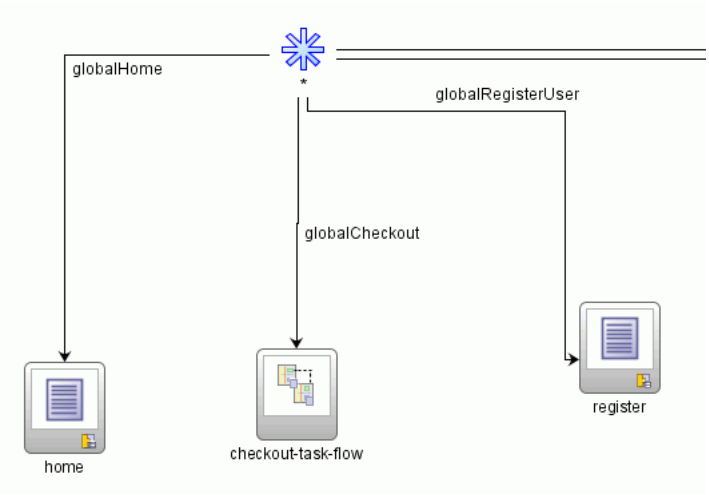
The task flow in Figure 18–5 also contains a method call activity (`methodCall1`) that invokes after the `ViewActivity2` view activity and before the `taskflowCall1` task flow call activity. In a task flow, you invoke an activity such as a method call activity before or after a page renders. Invoking a method call activity outside of a particular page can facilitate reuse because you can reuse the page in other contexts that don't require the method (for example, a different task flow). For more information about control flow rules, see Section 18.4, "Adding Control Flow Rules to Task Flows."

A wildcard control flow rule represents a control flow `from-activity-id` that contains a trailing wildcard (`foo*`) or a single wildcard character (`*`). Use the single wildcard character when you want to pass control from any activity in the task flow to

the wildcard control flow rule. Alternatively, use a trailing wildcard when you want to constrain the activities that can pass control to the wildcard control flow rule.

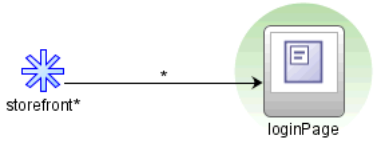
In Figure 18–6, the wildcard control flow rule contains a single wildcard character, indicating that control can pass to the activities connected to it in the task flow diagram from any activity within the task flow.

Figure 18–6 Wildcard Control Flow Rule With Single Wildcard



The trailing wildcard in Figure 18–7 indicates that control flow can pass to the `loginPage` view from any valid source activity whose `activity-id` begins with the characters `storefront`.

Figure 18–7 Wildcard Control Flow Rule with Trailing Wildcard



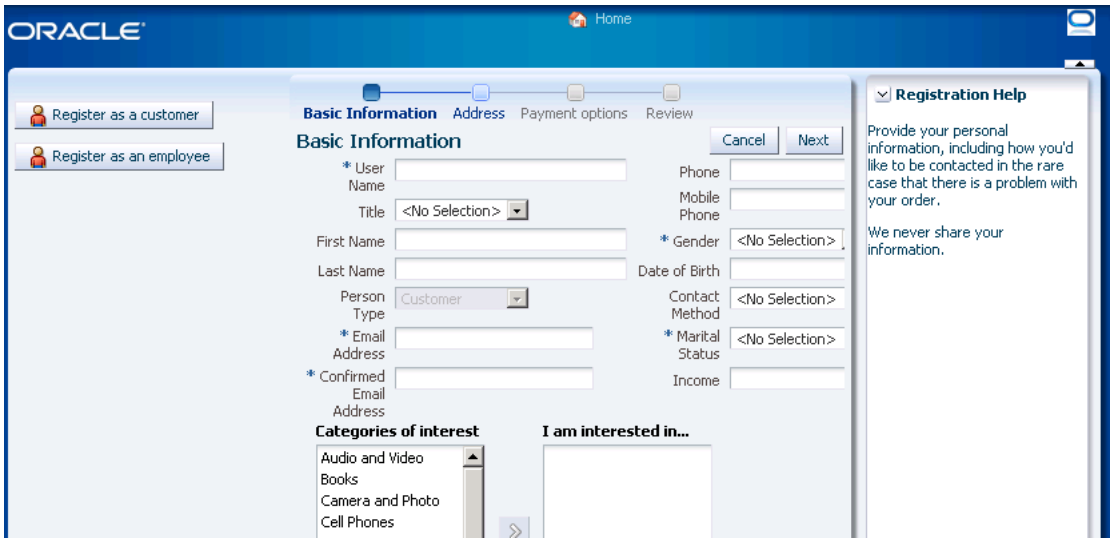
For more information about wildcard control flow rules, see Section 18.4.2, "How to Add a Wildcard Control Flow Rule."

18.1.4 ADF Task Flow Use Cases and Examples

Figure 18–8 shows a screen from the Fusion Order Demo application described in Chapter 2, "Introduction to the ADF Sample Application." End users can invoke a number of bounded task flows from this screen. The **Register as a customer** command button invokes the customer registration task flow (`customer-registration-task-flow.xml`) that allows a potential customer to register to use the application. This task flow renders screens where the user enters data such as name, email address, and so on. The **Register as an employee** command button invokes the employee registration task flow (`employee-registration-task-flow.xml`) that enables employees to register. Where appropriate, the customer registration and employee registration task flows reuse the same view activities. A third task flow in Figure 18–8 is the `help-task-flow.xml` task flow that renders in the **Registration Help** area. This task

flow provides assistance to the users as they navigate the tasks in the customer and employee registration task flows.

Figure 18–8 Task Flows for Registration and User Assistance



Task flows can also be used to secure your Fusion web application by reducing the number of access points that you expose to end users. For example, configure an unbounded task flow to display one page that provides navigation to the remaining pages in your application. Use bounded task flows for the remaining pages in the application. By setting the URL Invoke property of these bounded task flows to `url-invoke-disallowed`, your application has one access point (the page on the unbounded task flow). For more information about the URL Invoke property, see Section 19.6.4, "How to Call a Bounded Task Flow Using a URL." For more information about securing your Fusion web application, see Chapter 35, "Enabling ADF Security in a Fusion Web Application."

18.1.5 Additional Functionality for ADF Task Flows

You may find it helpful to understand other ADF features before you configure or use ADF task flows. Additionally, you may want to read about what you can do with your task flow configurations. Following are links to other functionality that may be of interest.

- Task flows can invoke managed beans. For more information about defining managed beans for use with a task flow, the supported memory scopes, and other related information, see Section 18.2.4, "What You May Need to Know About Memory Scope for Task Flows" and Section 24.4, "Using a Managed Bean in a Fusion Web Application".
- Task flows are reusable. For more information about reusing functionality in your application, Chapter 38, "Reusing Application Components."
- Task flows can be secured by defining the privileges that are required for someone to use it. For more information, see Chapter 35, "Enabling ADF Security in a Fusion Web Application."
- You can extend the functionality that your task flows implement by writing custom code. For example, you can write a custom exception handler that a task flow passes control to when a task flow activity raises an exception. For more

information, see Section 22.4.3, "How to Designate Custom Code as an Exception Handler."

Make sure when you write custom code that you do not import packages that are marked internal, as in the following example:

```
import oracle.adfinternal.controller.*;
```

For information about the APIs that you can use to write custom code, see the following reference documents:

- *Oracle Fusion Middleware Java API Reference for Oracle ADF Controller*
- *Oracle Fusion Middleware Java API Reference for Oracle ADF Faces*

18.2 Creating a Task Flow

A task flow is made up of the task flow itself, plus a number of activities with control flow rules between those activities. In most cases, the majority of the activities are view activities which represent the different pages in the flow. When some method or operation needs to be called, for example before a page is rendered, you use a method call activity with a control flow case from that activity to the appropriate next activity. When you want to call another task flow, you use a task flow call activity. If the flow requires some sort of branching, you use a router activity. At the end of a bounded task flow, you use a return activity which allows the flow to exit and control is sent back to the flow that called this bounded task flow.

For more detailed information and procedures regarding the individual components of a task flow, including the metadata created for each and additional configuration that you can set, see Section 18.3, "Adding Activities to a Task Flow."

18.2.1 How to Create a Task Flow

The processes for creating bounded and unbounded task flows are similar. The main difference is that you select the **Create as Bounded Task Flow** checkbox in the Create Task Flow dialog to create a bounded task flow.

Note: When you create the project, you may not need to create the unbounded task flow for it. If **ADF Page Flow** is specified as a selected technology on a Technology Scope page of the Project Properties dialog, the new `adfc-config.xml` source file is automatically created within the project. The `adfc-config.xml` source file is the main source file for the unbounded task flow.

Before you begin:

It may be helpful to have an understanding of what constitutes a task flow. For more information, see Section 18.2, "Creating a Task Flow."

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Section 18.1.5, "Additional Functionality for ADF Task Flows."

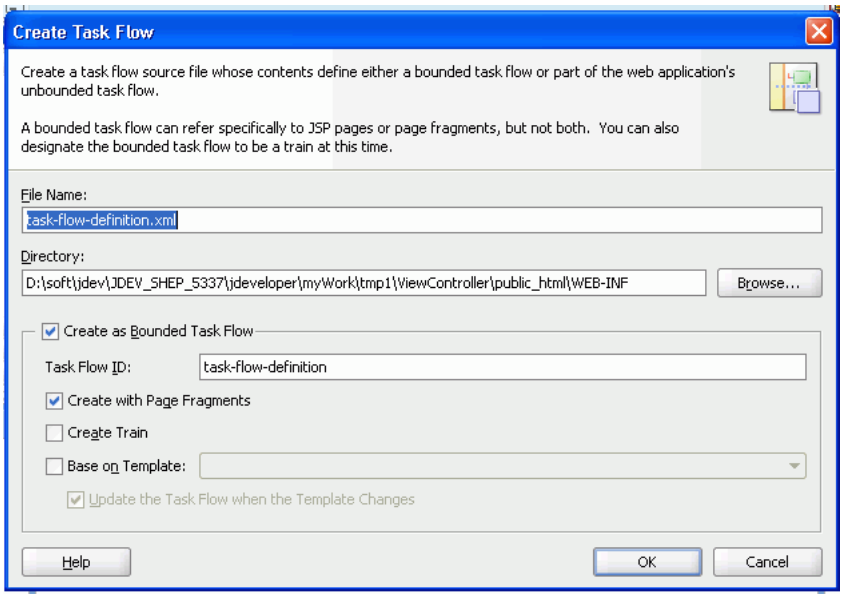
To create a task flow:

1. Create the task flow:
   a. In the Application Navigator, right-click the project where you want to create the task flow and choose **New**.

b. In the New Gallery, expand **Web Tier**, select **JSF/Facelets** and then select **ADF Task Flow**, and click **OK**.

The dialog shown in Figure 18–9 displays.

Figure 18–9 Create Task Flow Dialog



c. In the Create Task Flow dialog, the **Create as Bounded Task Flow** checkbox is selected by default. Deselect it to create a source file that will be incorporated into the application's unbounded task flow.

Deselecting the checkbox automatically changes the default value in the **File Name** field. This value will be used to name the XML source file for the task flow you are creating. The XML source file contains metadata describing the activities and control flow rules in the task flow.

Tip: The default name for the unbounded task flow is `adfc-config.xml`. The default name for the source file for a bounded task flow matches the value specified in the **Task Flow ID** field.

Because a single project can contain multiple task flows, a number may be added to the default value in the **File Name** field in order to give the source file a unique name, for example, `task-flow-definition3.xml`.

d. In the Create Task Flow dialog, the **Create with Page Fragments** checkbox is selected by default.

Clear this checkbox if you want the view activities that you add to the task flow to reference JSF pages that render in the main browser window as the root page. Leave the **Create with Page Fragments** checkbox selected if you want the view activities that you add to the task flow to reference page fragments files (`.jsff`) that the task flow displays in an ADF region at runtime.

e. Click **OK**.

A diagram representing the task flow displays in the editor.

Tip: You can view a thumbnail of the entire task flow diagram by clicking the diagram and then choosing **View** > **Thumbnail** from the main menu.

f. After you create the task flow, you can update it using the diagram, overview, and source editors

You can also use the Structure window to update the task flow.

Tip: There are other ways to create task flows, for example, by refactoring the contents of an existing task flow into a new task flow. For more information, see Section 18.6, "Refactoring to Create New Task Flows and Task Flow Templates".

2. In the ADF Task Flow page of the Component Palette, from the Component panel, drag and drop an activity onto the diagram.

Normally, you would start with a view activity. For more detailed procedures for adding any type of activity, see Section 18.3.1, "How to Add an Activity to a Task Flow."

- If you drag a view activity onto the diagram, you can double-click it to display the wizard for the JSF page or page fragment that the task flow is configured to invoke. Use the wizard to define characteristics for the page or page fragment. For more information, see Section 19.2, "Using View Activities".

Note: You can also add a view activity to a task flow by dragging a page from the Application Navigator and dropping it on the diagram for the task flow.

- If you drag a router activity onto the diagram, you can user the Property Inspector to create an expression whose evaluation will determine which control flow rule will be followed. For more information, see Section 19.4, "Using Router Activities."
- If you drag a method call activity onto the diagram, you can use the Property Inspector to configure the method to be called. For more information, see Section 19.5, "Using Method Call Activities."
- If you drag a task flow call activity onto the diagram, you can double-click it to display the Create Bounded Task Flow dialog where you can define settings for a new bounded task flow. For more information, see Section 19.6, "Using Task Flow Call Activities".
- If you are creating a bounded task flow, and you drag a task flow return activity onto the diagram, you can use the Property Inspector to configure the activity. For more information, see Section 19.7, "Using Task Flow Return Activities."

3. Create control flow cases between the activities (for more information and detailed procedures, see Section 18.4.1, "How to Add a Control Flow Rule to a Task Flow"):

a. In the ADF Task Flow page of the Component Palette, select **Control Flow Case**.

b. On the diagram, click a source activity, for example a view, and then click the destination activity. For example in Figure 18–15, two activities have been

linked with a control flow. Both source (`view1`) and the destination (`view2`) activities are linked.

c. Set the outcome value, using either the **From Action** attribute (if the outcome is to be determine by a method) or the **From Outcome** attribute (if the outcome can be set as a `String`).

4. If you are creating a bounded task flow, once that flow is complete, you may want to designate one of the activities as the default activity. Doing so makes sure that this specific activity executes first whenever the task flow runs. By default, the first activity you added to the task flow is the default. To change to a different activity, right-click the appropriate activity in the diagram and choose **Mark Activity > Default Activity**. For more information, see Section 18.2.3, "What You May Need to Know About the Default Activity in a Bounded Task Flow."

18.2.2 What Happens When You Create a Task Flow

A new XML source file is created every time you create a new unbounded or bounded task flow. By default, the XML source file for the unbounded task flow is called `adfc-config.xml`.

As shown in Example 18–1, `<adfc-config>` appears first as the top-level element in all ADF Controller XML source files. Bounded task flows, activities and control flow rules are defined inside the `<adfc-config>` element. Bounded task flows are identified within the source file by the `<task-flow-definition>` metadata element.

Example 18–1 Bounded Task Flow XML Source File

```
<?xml version="1.0" encoding="windows-1252" ?>
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2" id="__
1">
  <task-flow-definition id="task-flow-definition">
    <use-page-fragments/>
  </task-flow-definition>
</adfc-config>
```

A bounded task flow is identified by its task flow reference, which is comprised of a unique combination of identifier and document name. Example 18–2 shows a sample task flow reference within a task flow call activity.

Example 18–2 Task Flow Reference

```
<adfc-config xmlns="http://xmlns.oracle.com/adf/Controller" version="1.2">
  <task-flow-definition id="task-flow-definition">
   <use-page-fragments/>
.
.
    <task-flow-call id="taskFlowCall">
      <task-flow-reference>
        <document>/WEB-INF/target-task-flow-definition.xml</document>
        <id>my-task-flow</id>
      </task-flow-reference>
    </task-flow-call>
.
.
  </task-flow-definition>
</adfc-config>
```

Note: If you use JDeveloper to create the bounded task flow, specify only one ID (indicating one bounded task flow) per document.

You assign both identifier and document name when you create the bounded task flow. As shown in Example 18–2, the identifier is the value in the **Task Flow ID** field. The document name is the value in the **File Name** field.

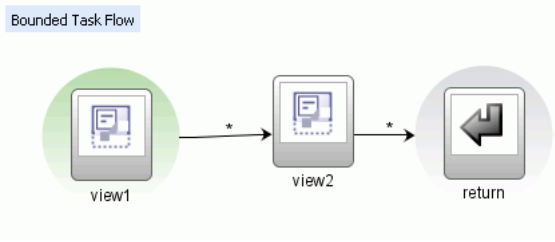
18.2.3 What You May Need to Know About the Default Activity in a Bounded Task Flow

The default activity is the first activity to execute in a bounded task flow. For example, the default activity always executes first when a task flow call activity passes control to the bounded task flow.

Unbounded task flows do not have default activities.

As shown in Figure 18–10, a green circle identifies the default activity in a task flow diagram.

Figure 18–10 Default Activity in a Bounded Task Flow



The first activity that you add to a new bounded task flow diagram is automatically identified as the default activity. You can also right-click any activity in the task flow diagram and choose **Mark Activity > Default Activity**. The default can be any activity type and it can be located anywhere in the control flow of the bounded task flow. To find the default activity, right-click anywhere on the task flow diagram and choose **Go to Default Activity**.

A bounded task flow can have only one default activity. If you mark a second activity as the default, the first is unmarked automatically. To unmark an activity manually, right-click the activity in the task flow diagram and choose **Unmark Activity > Default Activity**.

You should not specify a train stop in the middle of a train as a default activity (for more information, see Section 22.7, "Using Train Components in Bounded Task Flows").

Example 18–3 contains sample metadata for a default activity called `SurveyPrompt` in a bounded task flow:

Example 18–3 Default Activity Metadata in a Bounded Task Flow

```
<task-flow-definition id="survey">
    <default-activity>SurveryPrompt</default-activity>
    <view id="SurveryPrompt">
      <page>/SurveryPrompt.jsff</page>
    </view>
    <use-page-fragments/>
</task-flow-definition>
```

18.2.4 What You May Need to Know About Memory Scope for Task Flows

Each task flow in your Fusion web application defines a pageFlow scope to manage state. The pageFlow scope begins when the task flow begins and ends when the task flow ends. A pageFlow scope defines a unique storage area for each instance of a task flow within an application which is used to pass data values between the activities in the task flow. When one task flow calls another, the calling task flow cannot access the called task flow's pageFlow scope. This means, for example, that a UI component on a page referenced by a task flow's view activity cannot access the pageFlow scope of another task flow even if this task flow is an ADF region embedded in the same page as the UI component.

You can define multiple managed beans with task flows. Figure 18–11 shows examples from the Fusion Order Demo application where the default unbounded task flow (`adfc-config.xml`) and the customer registration task flow reference multiple managed beans with varying scope values. You can determine the scope assigned to a managed bean.

Figure 18–11 Managed Beans Registered with Task Flows

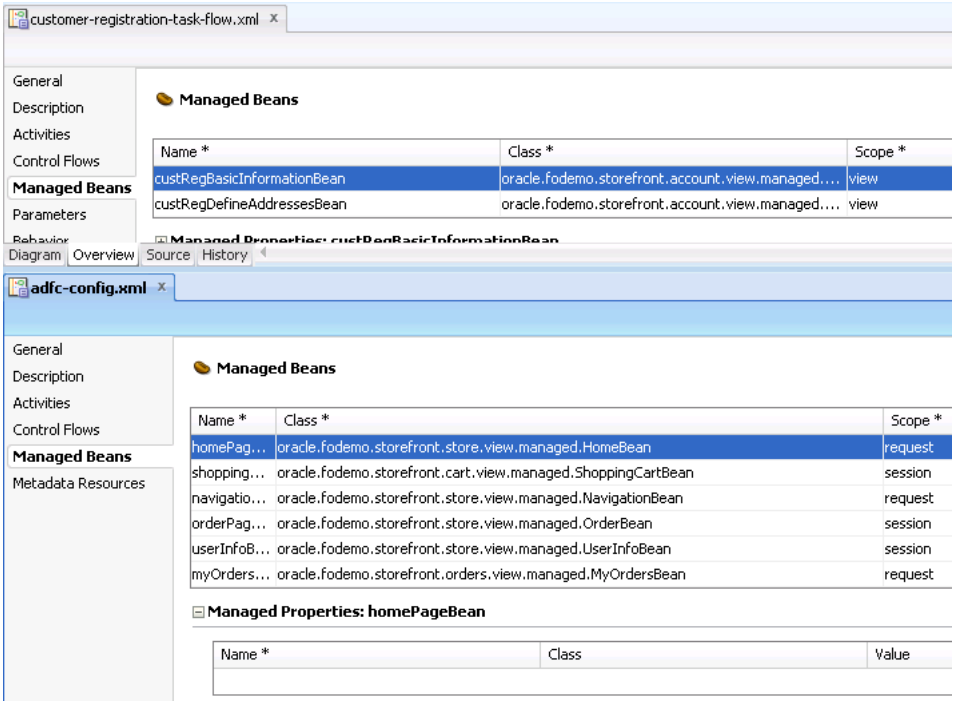


Table 18–3 lists available scopes for managed beans and describes when it is appropriate to use each scope in a managed bean that you define with a task flow. The table lists the scopes in order of their life span. For example, the application scope has a longer life span than the request scope. For more information the life span of object scopes, see Section 25.3, "About Object Scope Lifecycles."

Table 18–3 Memory Scope for ADF Managed Beans

Scope	Description
application	The application scope lasts until the application stops. Values that you store in a managed bean with this scope are available to every session and every request that uses the application. Avoid using this scope in a task flow because it persists beyond the life span of the task flow.

Table 18–3 (Cont.) Memory Scope for ADF Managed Beans

Scope	Description
session	The session scope begins when a user first accesses a page in the application and ends when the user's session times out due to inactivity, or when the application invalidates the session. Use this scope only for information that is relevant to the whole session, such as user or context information. Avoid using it to pass values from one task flow to another. Instead, use parameters to pass values between task flows. Using parameters gives your task flow a clear contract with other task flows that call it or are called by it. Another reason to avoid use of session scope is because it may persist beyond the life span of the task flow. Finally, if an end user opens multiple browser windows within the same session, session scoped values are shared across these browser windows. Conflicts can occur between these browser windows.
pageFlow	Choose this scope if you want the managed bean to be accessible across the activities within a task flow. A managed bean that has a pageFlow scope shares state with pages from the task flow that access it. A managed bean that has a pageFlow scope exists for the life span of the task flow.
view	Use this scope for managed bean objects that are needed only within the current view activity and not across view activities. The life span of this scope begins and ends when the current `viewId` of a view port changes. If you specify view, the application retains managed bean objects used on a page as long as the user continues to interact with the page. These objects are automatically released when the user leaves the page. Both JSF and Oracle ADF have an implementation of this scope. In a Fusion web application, when you use view scope in an expression, it resolves to the Oracle ADF implementation.
request	Use request scope when the managed bean does not need to persist longer than the current request.
backingBean	A backing bean is a convention to describe a managed bean that stores accessors for UI components and event handling code on a JSF page. It exists for the duration of a request and should not be used to maintain state. Use this scope if it is possible that your task flow appears in two ADF regions on the same JSF page and you want to isolate each instance of ADF region.

When you define a managed bean with a task flow, JDeveloper generates an entry similar to the following in the task flow's source file:

```
<managed-bean id="__15">
    <managed-bean-name id="__16">egBackingBean</managed-bean-name>
    <managed-bean-class id="__13">oracle....egBackingBean</managed-bean-class>
    <managed-bean-scope id="__14">backingBean</managed-bean-scope>
</managed-bean>
```

The `<managed-bean-scope>` element holds the value for the scope of the managed bean (`backingBean` in the example).

When you bind a UI component to a managed bean, JDeveloper appends `Scope` to the scope name in the EL expression that it generates to reference the managed bean. For example, the binding attribute of a table component that references the managed bean has the following EL expression:

```
<af:table id="cartTab"
  ...
  binding="#{backingBeanScope.egBackingBean.table}"
 ...
</af:table>
```

Restrict the scope of the managed bean that you reference through a UI component's binding attribute to backingBean or request scope. Instances of UI components cannot be serialized. Objects in scopes other than backingBean and request are expected to be serializable. For this reason, you should not bind UI components to managed beans that have a scope other than backingBean or request. Note that JDeveloper defaults the binding attribute for UI components and region fragments to use the backingBean scope.

Note: Write EL expressions that explicitly qualify the scope to access when writing EL expressions to access custom scopes unique to Oracle ADF (pageFlow, backingBean, and view scopes). For example, write an EL expression to access a pageFlow scope as follows:

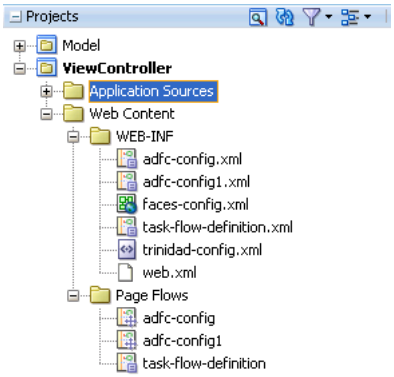
```
#{pageFlowScope.inpTxtBB.uiComponent}
```

18.2.5 What Happens at Runtime: Using Task Flows

A single application can have multiple unbounded task flow XML source files and multiple bounded task flow XML source files. The set of files that combine to produce the unbounded task flow is referred to as the application's *ADF Controller bootstrap configuration files*. The unbounded task flow is assembled at runtime by combining one or more ADF Controller bootstrap configuration files. All activities within the bootstrap configuration files that are not contained within a bounded task flow are considered to be within the unbounded task flow.

The names of the source files within a single application must be different. The example in Figure 18–12 contains two unbounded task flows (`adfc-config`, `adfc-config2`) and a bounded task flow (`task-flow-definition`).

Figure 18–12 Application with Two Unbounded Task Flow XML Source Files



18.3 Adding Activities to a Task Flow

After you create a task flow, you add activities to the task flow and configure the control flow between the activities. The Component Palette in JDeveloper displays available activities and control flows. You drag and drop the activities and control flows from the Component Palette to the diagram for the task flow. You can then configure control flow between the activities so that the task flow performs the task you want it to perform.

18.3.1 How to Add an Activity to a Task Flow

After you create a task flow, a task flow diagram and the Component Palette automatically display. The task flow diagram is a visual editor on which you can add the activities and control flows for the task flow. You can add them to the diagram by dragging them from the Component Palette.

Before you begin:

It may be helpful to have an understanding of what activities you can add to a task flow. For more information, see Section 18.3, "Adding Activities to a Task Flow."

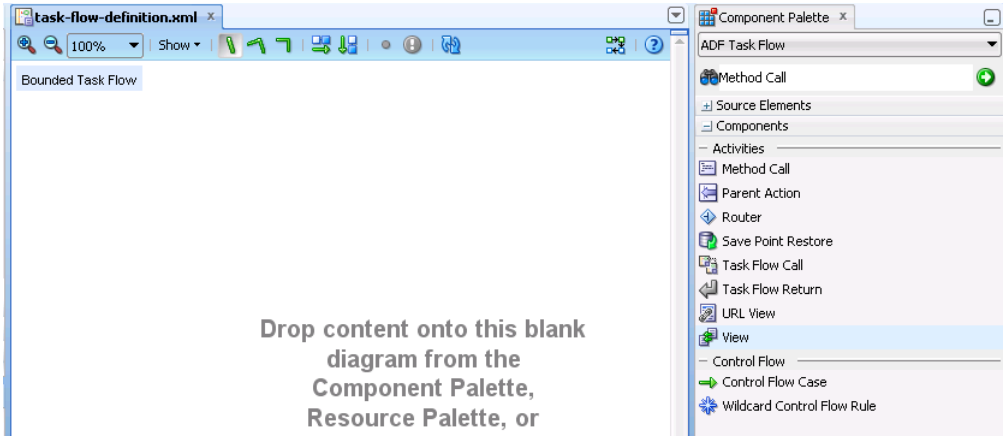
You may also find it helpful to understand functionality that can be added to task flows using other ADF features. For more information, see Section 18.1.5, "Additional Functionality for ADF Task Flows."

To add an activity to a task flow:

1. In the Application Navigator, double-click a task flow source file.

   For example, Figure 18–13 shows the editor and the Component Palette that automatically appears when you double-click a task flow source file (`task-flow-definition.xml`).

Figure 18–13 Task Flow Diagram



2. In the ADF Task Flow page of the Component Palette, from the Component panel, drag and drop an activity onto the diagram

   - If you drag a view activity onto the diagram, you can double-click it to display the wizard to create a JSF page or page fragment. For more information, see Section 19.2, "Using View Activities".
   - If you drag a task flow call activity onto the diagram, you can double-click it to display the Create Bounded Task Flow dialog where you can define settings for a new bounded task flow. For more information, see Section 19.6, "Using Task Flow Call Activities".

Tip: Each activity you drag to the task flow diagram can display optional status icons and a tooltip that provides additional information about the activity. For example, after you drag a view activity to the task flow diagram, it may display a warning icon until you associate it with a JSF page.

To turn on the icons, select **Show** at the top of the task flow diagram, and then select **Status** and one of the following:

- **Error**: Displays when there is a problem in the task flow metadata which prevents it from running. For example, a view activity in the metadata can contain a `<bookmark>` or `<redirect>` element, but not both.
- **Warning**: Displays when an activity is incomplete. For example, a view activity that doesn't have a physical page associated with it or a task flow call that doesn't have a task flow reference associated with it are both considered incomplete activities. The resulting task flow metadata may prevent it from running.

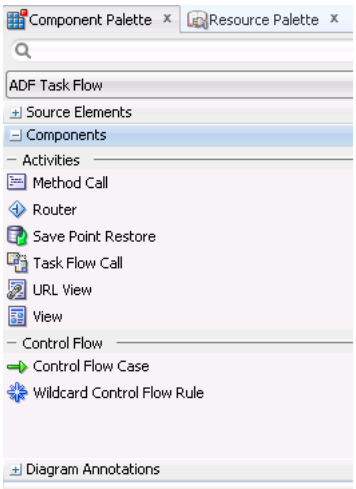
You can drag your mouse over a secondary icon to read a tooltip describing the icon's meaning.

18.3.2 What Happens When You Add an Activity to a Task Flow

As shown in Figure 18–14, the Component Palette contains separate sections for components and diagram annotations. The contents of the **Components** section differ slightly depending on whether you are creating a bounded or the unbounded task flow. For example, if you are creating a bounded task flow, the **Components** section contains an additional task flow return activity.

Figure 18–14 displays the activities you can add to the unbounded task flow.

Figure 18–14 Component Palette for an Unbounded Task Flow



18.4 Adding Control Flow Rules to Task Flows

An ADF Controller control flow rule defines how control passes from one activity to another in a task flow. A control flow rule can contain one or more control flow cases. A control flow case identifies the activity to which control flow passes. A control case also has options that allow you to configure conditional navigation (using the `<if>`

element) and/or limit navigation based on the value of the action from where the control flow originates (using the `from-action` element).

Control flow rules are based on JSF navigation rules, but capture additional information. JSF navigation is always between pages, whereas control flow rules describe transitions between activities. For example, a control flow rule can indicate a transition between a view activity and a subsequent method call activity. Or, it can indicate that control passes from the page (view activity) to another task flow.

The following task flow activities cannot be the source of a control flow rule:

- Save Point Restore
- Task Flow Return
- URL View

The basic structure of a control flow rule mimics a JSF navigation rule. Table 18–4 describes how metadata maps from JSF navigation rules to ADF Controller control flow rules.

Table 18–4 Mapping of JSF Navigation Rules to ADF Controller Control Flow Rules

JSF Navigation Rule	ADF Controller Control Flow Rule
Navigation Rule	Control Flow Rule
From View ID	From Activity ID
Navigation Case	Control Flow Case
From Action	From Action
From Outcome	From Outcome
If	If
To View ID	To Activity ID

When using ADF task flows, perform all application navigation using ADF Controller control flow rules instead of using JSF navigation rules in the `faces-config.xml` file. ADF Controller delegates navigation handling when it does not find a matching control flow case in its metadata. However, not all ADF Controller functionality is guaranteed to work correctly if navigation is performed by a non-ADF Controller navigation handler. For more information about how ADF Controller evaluates control flow rules, see Section 18.4.4, "What Happens at Runtime: Evaluating Control Flow Rules."

Use the task flow diagram as a starting point for creating basic control flows between activities. Later, you can edit control flow properties in the Structure window, Property Inspector or overview editor for the task flow diagram.

> **Tip:** You can drag and drop an activity onto an existing control flow. This splits the existing control flow into two, with the activity in the center.

18.4.1 How to Add a Control Flow Rule to a Task Flow

You create a control flow rule by dragging a Control Flow Case from the ADF Task Flow page of the Component Palette and dropping it on a source task flow activity and a target task flow activity.

Before you begin:

It may be helpful to have an understanding of what a control flow rule is. For more information, see Section 18.4, "Adding Control Flow Rules to Task Flows."

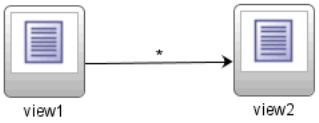
You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Section 18.1.5, "Additional Functionality for ADF Task Flows."

To add a control flow rule to a task flow:

1. In the Application Navigator, double-click the source file for the task flow where you want to add a control flow rule. For example, `adfc-config.xml`.
2. In the ADF Task Flow page of the Component Palette, select **Control Flow Case**.
3. On the diagram, click a source task flow activity, for example a view activity, and then click the destination task flow activity.

   Figure 18–15 shows two task flow activities (**view1**) and (**view2**) that have been linked with a control flow case.

Figure 18–15 Control Flow Case



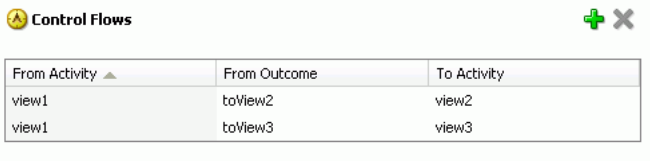
   JDeveloper adds the control flow case to the diagram. Each line that JDeveloper adds between task flow activities represents a control flow case. The arrow indicates the direction of the control flow case. The **From Outcome** element contains a value that can be matched against values specified in the `action` attribute of UI components.

4. To change the value of **From Outcome**, select the text next to the control flow in the diagram. By default, this text is the wildcard * character as shown in Figure 18–15. You can overwrite the text with a new value, for example, `toView2`.
5. To change the values of **From Activity ID** (identifies the source activity) or **To Activity ID** (identifies the target activity), drag either end of the arrow in the diagram to a new activity.

   Tip: After you select the control flow in the task flow diagram, you can also change its properties in the Property Inspector or the Structure window. The Structure window is helpful for displaying the relationship between control rules and cases.

   You can also click **Control Flows** on the overview editor for the task flow diagram to add cases, as shown in Figure 18–16. To add a case, make sure that the **From Activity** (source activity) and the **To Activity** (target activity) for the rule have already been added to the task flow.

Figure 18–16 Control Flows on Overview Editor for the Task Flow



6. Optionally, in the Property Inspector expand the **Behavior** section, and write an EL expression in the **If** field that must evaluate to `true` before control can pass to the activity identified by **To Activity ID**.

18.4.2 How to Add a Wildcard Control Flow Rule

You can add a wildcard control flow rule to an unbounded or bounded task flow. The steps for adding it are similar to those for adding any activity to a task flow diagram.

Before you begin:

It may be helpful to have an understanding of the control flow rule options available to you. For more information, see Section 18.4, "Adding Control Flow Rules to Task Flows."

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Section 18.1.5, "Additional Functionality for ADF Task Flows."

To add a wildcard control flow rule:

1. In the Application Navigator, double-click the source file for a task flow to display the task flow diagram. For example, double-click `adfc-config.xml`.
2. In the ADF Task Flow page of the Component Palette, from the Component panel, drag and drop a **Wildcard Control Flow Rule** onto the diagram.
3. Select **Control Flow Case** in the ADF Task Flow page list of the Component Palette.
4. In the task flow diagram, drag the control flow case from the wildcard control flow rule to the target activity.

   The target can be any activity type.
5. By default, the label below the wildcard control flow rule is *. This is the value for the **From Activity ID** element. To change this value, select the wildcard control flow rule in the diagram. In the Property Inspector for the wildcard control flow rule, enter a new value in the **From Activity ID** field. For example, enter `project*`. The wildcard must be a trailing character in the new label.

   Tip: You can also change the **From Activity ID** value in the overview editor for the task flow diagram.
6. Optionally, in the Property Inspector expand the **Behavior** section, and write an EL expression in the **If** field that must evaluate to `true` before control can pass to the activity identified by **To Activity ID**.

18.4.3 What Happens When You Create a Control Flow Rule

Understanding the elements that define the rules in the source file for the task flow helps when creating control flow rules directly in the task flow diagram, task flow

overview editor, or Structure window, or when adding them directly in the XML source file. Example 18–4 shows the general syntax of a control flow rule element in the task flow source file.

Example 18–4 Control Flow Rule Syntax in the Source File

```
<control-flow-rule>
    <from-activity-id>from-view-activity</from-activity-id>
  <control-flow-case>
    <from-action>actionmethod</from-action>
    <from-outcome>outcome</from-outcome>
    <to-activity-id>destinationActivity</to-activity-id>
    <if>#{myBean.someCondition}</if>
  </control-flow-case>
  <control-flow-case>
   ....
  </control_flow-case>
</control-flow-rule>
```

Control flow rules can consist of the following metadata:

- `control-flow-rule`: A mandatory wrapper element for control flow case elements.
- `from-activity-id`: The identifier of the activity where the control flow rule originates, for example, `source`.

  A trailing wildcard (*) character in `from-activity-id` is supported. The rule will apply to all activities that match the wildcard pattern. For example, `login*` matches any logical activity ID name beginning with the literal `login`. If you specify a single wildcard character in the metadata (not a trailing wildcard), the control flow automatically converts to a wildcard control flow rule activity in the diagram. For more information, see Section 18.4.2, "How to Add a Wildcard Control Flow Rule".
- `control-flow-case`: A mandatory wrapper element for each case in the control flow rule. Each case defines a different control flow for the same source activity. A control flow rule must have at least one control flow case.
- `from-action`: An optional element that limits the application of the rule to outcomes from the specified action method. The action method is specified as an EL binding expression, such as `#{backing_bean.cancelButton_action}`.

  In Example 18–4, control passes to `destinationActivity` only if `outcome` is returned from `actionmethod`.

  The value in `from-action` applies only to a control flow originating from a view activity, not from any other activity types. Wildcards are not supported in `from-action`.
- `from-outcome`: Identifies a control flow case that will be followed based on a specific originating activity outcome. All possible originating activity outcomes should be accommodated with control flow cases.

  If you leave both the `from-action` and the `from-outcome` elements empty, the case applies to all outcomes not identified in any other control flow cases defined for the activity, thus creating a default case for the activity. Wildcards are not supported in `from-outcome`.
- `to-activity-id`: A mandatory element that contains the complete identifier of the activity to which the navigation is routed if the control flow case is performed. Each control flow case can specify a different `to-activity-id`.

- `if`: An optional element that accepts an EL expression as a value. If the EL expression evaluates to `true` at runtime, control flow passes to the activity identified by the `to-activity-id` element.

18.4.4 What Happens at Runtime: Evaluating Control Flow Rules

At runtime, ADF Controller evaluates control flow rules from the most specific to the least specific match to determine the next transition between activities. Evaluation is based on the following priority, which is similar to that for JSF navigation rules:

1. `from-activity-id`, `from-action`, `from-outcome`
2. `from-activity-id`, `from-outcome`
3. `from-activity-id`

ADF Controller first searches for a match in all three elements: `from-activity-id`, `from-action`, and `from-outcome`. If there is no match, ADF Controller searches for a match in just the `from-activity-id` and `from-outcome` elements. Finally, ADF Controller searches for a match in the `from-activity-id` element alone.

If ADF Controller cannot find a control flow rule within its metadata to match a request, it allows the standard JSF navigation handler to find a match.

The unbounded task flow can have more than one ADF Controller XML source file. Because control flow rules can be defined in more than one ADF Controller XML source file, similar rules may be defined in different files. If there is a conflict in which two or more cases have the same `from-activity-id`, and the same `from-action` or `from-outcome` values, the last case (as listed in the `adfc-config.xml`, bootstrap, or bounded task flow source file) is used. If the conflict is among rules defined in different source files, the rule in the last source file to be loaded is used.

ADF Controller also implements the following interface with a number of restrictions:

`javax.faces.application.ConfigurableNavigationHandler`

The restrictions are:

- The `Map` object returned by `getNavigationCases()` is not modifiable. Any runtime changes to control flow rules must be made with the customization features provided with the MDS framework. For more information, see Chapter 39, "Customizing Applications with MDS."
- Do not invoke the `performNavigation()` method after the JSF Invoke Application phase. This is to make sure that the view ID does not change between the ADF Prepare Render phase and the JSF Render Response phase. For more information about how the JSF and ADF phases integrate in the lifecycle of a page request, see Chapter 25, "Understanding the Fusion Page Lifecycle."
- The metadata values for a task flow view activity's bookmark and redirect properties populate the corresponding information for the navigation case objects returned by the `getNavigationCases()`.

18.5 Testing Task Flows

The procedure for running and debugging task flows differs depending on whether the task flow is bounded or unbounded, whether it contains pages or page fragments, or whether it accepts input parameters.

JDeveloper and Oracle ADF provides a number of features that facilitate your testing of the task flows that you create. Among these is the ability to set ADF declarative

breakpoints on task flow activities. For more information, see Section 36.9.1, "How to Set and Use Task Flow Activity Breakpoints." In addition, you can also modify a task flow's metadata for an application that is currently executing in the Integrated WebLogic Server. The changes that you make to the task flow can be deployed to the application without stopping the application and redeploying it to the Integrated WebLogic Server (also known as *hot reloading*). To do this, you invoke the make command on the modified task flow (right-click the task flow and select **Make** from the context menu that appears). This deploys the modified task flow to the application executing in the Integrated WebLogic Server. For more information about hot reloading, see the Section 36.4, "Reloading Oracle ADF Metadata in Integrated WebLogic Server."

The Configure Default Domain dialog displays the first time you run your application and start a new domain in Integrated WebLogic Server. Use the dialog to define an administrator password for the new domain. Passwords you enter can be eight characters or more and must have a numeric character.

18.5.1 How to Run a Bounded Task Flow That Contains Pages

You can run or debug a bounded task flow that contains view activities that are pages.

For information on running a bounded task flow that contains view activities that are page fragments, see Section 18.5.2, "How to Run a Bounded Task Flow That Uses Page Fragments".

Note: You can select a view activity inside a task flow diagram or the Application Navigator and choose **Run** to run a bounded task flow.

In a bounded task flow, you must designate the view as a default activity and run the task flow from the Application Navigator. For more information, see Section 18.2.3, "What You May Need to Know About the Default Activity in a Bounded Task Flow".

If the first activity that runs in the task flow is an activity type other than view, you must use a bounded task flow.

Before you begin:

It may be helpful to have an understanding of the factors that affect how you test a task flow. For more information, see Section 18.5, "Testing Task Flows."

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Section 18.1.5, "Additional Functionality for ADF Task Flows."

To run or debug a bounded task flow that uses pages:

- In the task flow diagram, right-click the task flow and choose either **Run** or **Debug**.
- You can also run the task flow directly by entering its URL in the browser. For example:

  ```
  http://somecompany.com/internalApp/MyApp/faces/adf.task-flow?adf.tfId=displayHelp&adf.tfDoc=%2FWEB-INF%2Fdisplayhelp.xml&topic=createPurchaseOrder
  ```

  For more information, see Section 19.6.6, "What You May Need to Know About Calling a Bounded Task Flow Using a URL."

- You can right-click the bounded task flow in the Application Navigator and choose either **Run** or **Debug**.

18.5.2 How to Run a Bounded Task Flow That Uses Page Fragments

Bounded task flows that use page fragments are intended to run only within an ADF region. A *page fragment* is a JSF document that renders as content in another JSF page. For more information, see Section 21.1.1, "About Page Fragments and ADF Regions".

Before you begin:

It may help to understand what factors affect how you test a task flow. For more information, see Section 18.5, "Testing Task Flows."

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Section 18.1.5, "Additional Functionality for ADF Task Flows."

To run or debug a bounded task flow that uses page fragments:

1. If you have not done so already, create a JSF page containing a region that binds to the bounded task flow. JDeveloper automatically creates the region for you in a JSF page when you drop a bounded task flow containing page fragments on a page. For more information about creating regions, see Section 21.2, "Creating an ADF Region."
2. Create a view activity in the project's unbounded task flow that refers to the page. See Section 18.3.1, "How to Add an Activity to a Task Flow" for more information.
3. Right-click the view activity in the Application Navigator or in the task flow diagram and choose **Run**.

18.5.3 How to Run a Bounded Task Flow That Has Parameters

Before you run a bounded task flow with parameters, you must first run a bounded task flow containing pages. For more information about bounded task flow input parameters, see Chapter 20, "Using Parameters in Task Flows".

Before you begin:

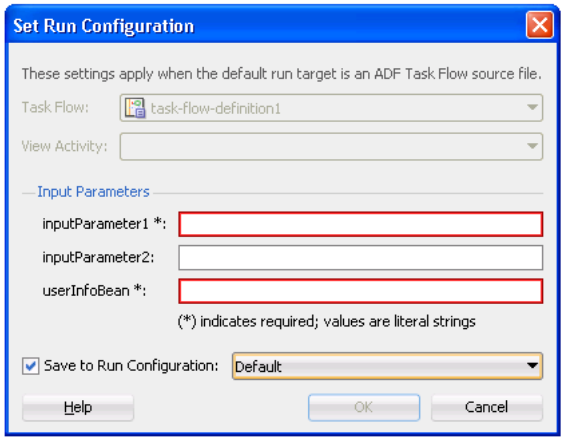
It may be helpful to have an understanding of the factors that affect how you test a task flow. For more information, see Section 18.5, "Testing Task Flows."

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Section 18.1.5, "Additional Functionality for ADF Task Flows."

To run a bounded task flow that has input parameter definitions:

1. If the bounded task flow has defined input parameters, the Set Run Configuration dialog displays after you select either **Run** or **Debug**, as shown in Figure 18–17.

Figure 18–17 Set Run Configuration dialog



2. In the **Input Parameters** list, enter values that you want to be passed as input parameters to the task flow. If you do not specify a value, the input parameter is not used when calling the bounded task flow.

   Each required input parameter in the list displays with an asterisk, as shown in Figure 18–17. You must specify the parameter value as a literal string. You cannot specify an EL expression.

3. Click **OK**.

18.5.4 How to Run a JSF Page When Testing a Task Flow

You can run a JSF page by right-clicking the page in the Application Navigator and choosing **Run**. However, if the page contains navigation UI components, such as a button or link, navigation is not guaranteed to work.

Before you begin:

It may be helpful to have an understanding of the factors that affect how you test a task flow. For more information, see Section 18.5, "Testing Task Flows."

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Section 18.1.5, "Additional Functionality for ADF Task Flows."

To run a JSF Page with fully functioning navigation:

1. Create a bounded or unbounded task flow. See Section 18.2.1, "How to Create a Task Flow" for more information.
2. Add a view activity to the task flow. See Section 18.3.1, "How to Add an Activity to a Task Flow" for more information.
3. In the Application Navigator, select the JSF page you want to run and drop it on top of the view activity in the task flow diagram.

   This associates the view activity with the JSF page.

4. In the diagram, right-click the view activity and choose **Run**.

18.5.5 How to Run the Unbounded Task Flow

To run or debug the unbounded task flow, you select a specific view activity with which to start the unbounded task flow.

Before you begin:

It may be helpful to have an understanding of the factors that affect how you test a task flow. For more information, see Section 18.5, "Testing Task Flows."

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Section 18.1.5, "Additional Functionality for ADF Task Flows."

To run a view activity in the unbounded task flow:

- In the task flow diagram, right-click the view activity and choose either **Run** or **Debug**.

  The unbounded task flow runs beginning with the selected view activity.

- If you have selected something other than a single view activity (or have nothing selected), you are prompted to select one in the Set Run Configuration dialog.

18.5.6 How to Set a Run Configuration for a Project

A *run configuration* contains settings that determine how projects run, such as specifying the first activity to run in a task flow. You can define one or more run configurations for a project. Within a run configuration, you can designate an ADF Controller source file as the default run target. When you run the project, the source file is the first to run.

Before you begin:

It may be helpful to have an understanding of the factors that affect how you test a task flow. For more information, see Section 18.5, "Testing Task Flows."

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Section 18.1.5, "Additional Functionality for ADF Task Flows."

To define a default task flow run target:

1. Select the project in the Application Navigator.
2. From the main menu, choose **Run** > **Choose Active Run Configuration** > **Manage Run Configurations**.
3. In the Manage Run Configurations dialog, choose **Run/Debug/Profile** and choose **New**.
4. In the New dialog, enter the name for the new run configuration.
5. If you want to base the new configuration on an existing one, choose a configuration in the **Copy Settings From** dropdown list.
6. Click **OK** to exit the dialog.
7. Click **Edit**.
8. In the **Default Run Target** field in the Edit Run Configuration dialog, specify a source file for the task flow that should run first when you run the project.

   Once you choose a task flow, you can set a view activity (for the unbounded task flow) or input parameters (for bounded task flows).
9. In the left panel of the Edit Configuration dialog, click **ADF Task Flow**.
10. In the **Task Flow** dropdown list, located on the right panel, select the task flow containing the run target.

11. If you are running the unbounded task flow, the Edit Run Configuration dialog displays the **Run Target Activity** list. Select the view activity that will run first in the application.
12. Click **Open**.

    The next time you run the project, the saved run configuration will be available in the **Run** > **Choose Active Run Configuration** menu.

    If you are running a bounded task flow that has been set up to accept input parameters, a dialog will display a section for specifying values for all input parameters defined for the bounded task flow. See Chapter 18.5.3, "How to Run a Bounded Task Flow That Has Parameters" for more information.

18.6 Refactoring to Create New Task Flows and Task Flow Templates

You can convert existing activities, JSF page flows, and JSF pages into new ADF Controller components such as bounded task flows and task flow templates.

18.6.1 How to Create a Bounded Task Flow from Selected Activities

You can create a new bounded task flow based on activities you select in an existing bounded or unbounded task flow.

Before you begin:

It may be helpful to have an understanding of the factors that affect how you refactor a task flow. For more information, see Section 18.6, "Refactoring to Create New Task Flows and Task Flow Templates."

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Section 18.1.5, "Additional Functionality for ADF Task Flows."

To create a new bounded task flow from selected activities:

1. In the Application Navigator, double-click the source file for the unbounded or bounded task flow containing the activities you want to extract to a new bounded task flow.
2. In the task flow diagram, select one or more activities.

   Tip: To select multiple activities in a diagram, click the left mouse button and drag the cursor over the activities.

   You can also press the Ctrl key while selecting each activity.

3. Right-click your selection and choose **Extract Task Flow** from the context menu that appears.

   The Extract Bounded Task Flow dialog displays. This dialog allows you to specify the file name and directory location for the new bounded task flow. For more information, see Section 18.2, "Creating a Task Flow".

The new bounded task flow displays in the task flow diagram. Table 18–5 describes the properties that JDeveloper automatically sets for the new bounded task flow.

Table 18–5 Properties Updated in the New Bounded Task Flow

Property	Value
Task flow definition ID	Value you entered in the **File Name** field in the Extract Bounded Task Flow dialog.
Default activity	Determined as the destination of all incoming control flow cases. If more than one destination exists, an error is flagged and the entire operation is rolled back.
Control flow rules	Control flow cases with selected source activities are included in the new bounded task. A *source activity* is an activity from which a control flow leads. The new bounded task flow includes the following types of control flow cases: ▪ Both the source and target activities in the control flow case were selected to create the new task flow. ▪ Only the source activity was selected to create the new task flow. Destinations are changed to the corresponding new task flow return activities added for each outcome.

The following changes automatically occur in the originating task flow (the task flow containing the activities you selected as the basis for the new task flow):

- A new task flow call activity is added to the originating task flow. The task flow call activity calls the new bounded task flow.
- The selected activities are removed from the originating task flow.
- Existing control flow cases associated with the activities you selected are removed from the originating task flow. They are replaced with new control flow cases:
  - An incoming control flow case to the old activity is redirected to the new task flow call activity.
  - An outgoing control flow case from the old activity is redirected from the new task flow call activity.

18.6.2 How to Create a Task Flow from JSF Pages

You can create a new bounded task flow based on selected pages in a JSF page flow. JDeveloper converts the JSF pages that are part of a flow (that is, those that are linked by JSF navigation cases) to view activities in the new task flow.

Before you begin:

It may be helpful to have an understanding of the factors that affect how you refactor a task flow. For more information, see Section 18.6, "Refactoring to Create New Task Flows and Task Flow Templates."

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Section 18.1.5, "Additional Functionality for ADF Task Flows."

To create a new task flow from selected JSF pages in a page flow.

1. In the Application Navigator, double-click the source file for the page flow containing the pages you want to use in the new bounded task flow.
2. In the task flow diagram, select one or more JSF pages.

Tip: To select multiple elements in a diagram, click the left mouse button and drag the cursor over the elements.

You can also press the Ctrl key while selecting each element.

3. Right-click your selection and choose **Generate ADF Task Flow**.

   The Create Task Flow dialog displays, which allows you to create a new unbounded or bounded task flow. For more information, see Section 18.2, "Creating a Task Flow".

18.6.3 How to Convert Bounded Task Flows

You can convert an existing bounded task flow to an unbounded task flow or change whether the views it contains are pages or page fragments. Table 18–6 describes the results of each conversion.

Table 18–6 Converting Bounded Task Flows

Conversion	Result
Bounded task flow to unbounded task flow	Loses all metadata not valid for unbounded task flows, such as parameter definitions and transactions.
Bounded task flow to use JSF pages	Converts page fragments associated with any view activities in the task flow to JSF pages. Old page fragments are saved if you select the **Keep Page Fragment** checkbox. New JSF page names default to the name of the old page fragment.
Bounded task flow to use page fragments	Converts all pages associated with view activities in the bounded task flow to page fragments. Old pages are saved if you select the **Keep Page** checkbox. New page fragment names default to the name of the old page

Before you begin:

It may be helpful to have an understanding of the factors that affect how you refactor a task flow. For more information, see Section 18.6, "Refactoring to Create New Task Flows and Task Flow Templates."

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Section 18.1.5, "Additional Functionality for ADF Task Flows."

To convert a bounded task flow:

1. In the Application Navigator, double-click the source file for the bounded task flow that you want to convert.
2. In the task flow diagram, right-click anywhere other than on an activity or control flow.
3. Choose the appropriate menu option:
   - **Convert to Unbounded Task Flow**
   - **Convert to Task Flow with Page Fragments**
   - **Convert to Task Flow with Pages**

18.7 What You Should Know About Task Flow Constraints

Table 18–7 summarizes assumptions about and constraints for using task flows, activities, and other associated ADF Controller features.

Table 18–7 ADF Controller Features Assumptions and Constraints

Feature Area	Assumption/ Constraint	Description
ADF Controller objects and diagram UI	JSF view layer	ADF Controller operates in a JSF environment. Oracle's web-based Fusion web application strategy focuses on JSF as the sole view layer technology.
	Dependent on Oracle ADF Faces	ADF Controller extensions are implemented on top of Oracle ADF Faces. They are dependent on the ADF Faces libraries, but ADF Controller can run against any JSF implementation, providing these libraries are present.
	Navigation and state management encapsulated	ADF Controller encapsulates both navigation and, to some extent, state management. JSF and the Servlet API are still available for the basic management of state at the application, session, and request levels.
	Model layer	ADF model layer is used to implement the application's model layer.
	Dependent on MDS	ADF Controller metadata is stored in MDS. However, MDS is currently not capable of loading `faces-config.xml`. If the customization features that MDS provides are required, you should use ADF task flows exclusively in order to define managed beans and control flow rules.
	No supported migration path from struts or model 1	There is no support for a migration from Struts or Model 1 to the Fusion ADF Controller. However, you can create a new ADF bounded task flow based on selected pages in a JSF page flow. For more information, see Section 18.6.2, "How to Create a Task Flow from JSF Pages".
Bounded task flow	Exposed as page flow-scoped state	ADF Controller manages implementation of a page flow scoped-state. Any auto-management functions provided by the framework, such as back button support and state cleanup function, assume page flow-scoped data. In order for an application to fully implement such functions for all of its pages, the entire application should be exposed as an ADF bounded task flow, using nested bounded task flows as needed. The application should store any state requiring versioning within the page flow scope.
	Transactional boundaries	The developer will use ADF bounded task flows to manage transaction boundaries.

Table 18–7 (Cont.) ADF Controller Features Assumptions and Constraints

Feature Area	Assumption/ Constraint	Description
Page flow scope	Access availability within ADF lifecycle	An application cannot attempt to access the page flow scope early in the ADF lifecycle before ADF Controller is ready to provide it. Page flow scope is not guaranteed to be available for access until after Before and After listeners have executed on the Restore View phase. ADF Controller uses before and after listeners on the Restore View phase to synchronize the server side state with the request. This is where things such as browser back-button detection and bookmark dereference are handled.
Navigation	Navigation	When using ADF Controller task flows, perform all application navigation should be performed using ADF Controller control flow rules instead of using navigation rules in `faces-config.xml`. Although the ADF Controller delegates navigation handling when no matching control flow cases are found in ADF Controller metadata, not all ADF Controller functionality is guaranteed to work correctly if navigation is performed by a non-ADF Controller `NavigationHandler`.

19 Working with Task Flow Activities

This chapter describes how to use activities in your task flows. The chapter contains detailed information about each task flow activity that displays in the Component Palette and its properties.

This chapter includes the following sections:

- Section 19.1, "About Task Flow Activities"
- Section 19.2, "Using View Activities"
- Section 19.3, "Using URL View Activities"
- Section 19.4, "Using Router Activities"
- Section 19.5, "Using Method Call Activities"
- Section 19.6, "Using Task Flow Call Activities"
- Section 19.7, "Using Task Flow Return Activities"
- Section 19.8, "Using Save Point Restore Activities"
- Section 19.9, "Using Parent Action Activities"
- Section 19.10, "Using Task Flow Activities with Page Definition Files"

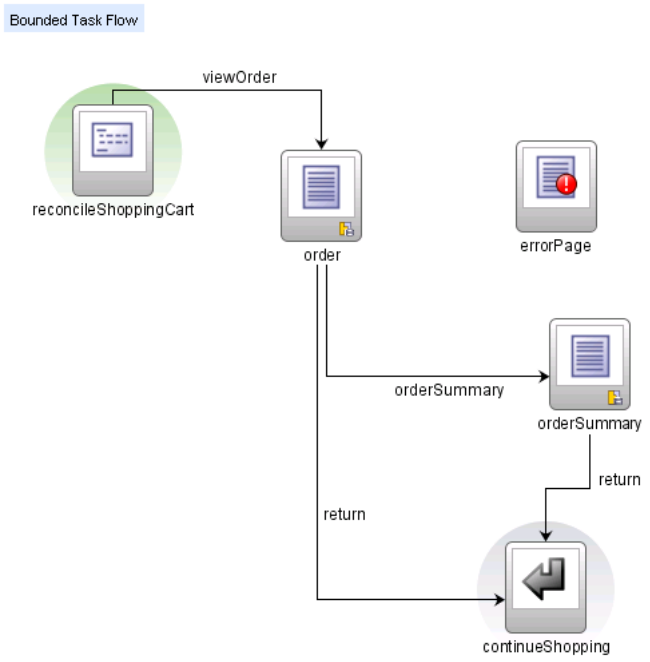
19.1 About Task Flow Activities

An *activity* represents a piece of work that is performed when the task flow runs. It displays in the task flow's diagram editor as a node. You can add most activities to both bounded and unbounded task flows, although some activity types can be added only to a bounded task flow.

The bounded task flow shown in Figure 19–1 contains activities that run in order to check out a user from the shopping application:

1. A call to a method synchronizes the items a user may have chosen before logging in with those selected after logging in
2. A page (view activity) that displays the items the user has currently selected and another page that summarizes the order
3. An activity that causes control to return back to the calling unbounded task flow shown in Figure 19–1

Figure 19–1 Checkout Bounded Task Flow in Fusion Order Demo Application



A task flow consists of activities and control flow cases that define the transitions between activities. Table 19–1 describes the types of activities and control flows you can add to a task flow by dragging and dropping from the Component Palette.

Table 19–1 Task Flow Activities and Control Flows

Icon	Component Name	Description
	Method Call	Invokes a method, typically a method on a managed bean. A method call activity can be placed anywhere within an application's control flow to invoke application logic based on control flow rules. See Section 19.5, "Using Method Call Activities" for more information.
	Parent Action	Allows a bounded task flow to generate outcomes that are passed to its parent view activity. See Section 19.9, "Using Parent Action Activities" for more information.
	Router	Evaluates an EL expression and returns an outcome based on the value of the expression. For example, a router in a credit check task flow might evaluate the return value from a previous method call and generate success, failure, or retry outcomes based on various cases. These outcomes can then be used to route control to other activities in the task flow. See Section 19.4, "Using Router Activities" for more information.
	Save Point Restore	Restores a previous persistent save point, including application state and data, in an application supporting save for later functionality. See Section 22.6, "Using Save Points in Task Flows" for more information.

Table 19–1 (Cont.) Task Flow Activities and Control Flows

Icon	Component Name	Description
	Task Flow Call	Calls a bounded task flow from the unbounded task flow or another bounded task flow. See Section 19.6, "Using Task Flow Call Activities" for more information.
	Task Flow Return	Identifies when a bounded task flow completes and sends control flow back to the caller. (Available for bounded task flows only). See Section 19.7, "Using Task Flow Return Activities" for more information.
	URL View	Redirects the root view port (for example, a browser page) to any URL-addressable resource, even from within the context of an ADF region. See Section 19.3, "Using URL View Activities" for more information.
	View	Displays a JSF page or page fragment. Multiple view activities can represent the same page or same page fragment. See Section 19.2, "Using View Activities" for more information. See Section 24.3, "Creating a Web Page" for more information about pages and page fragments.
	Control Flow Case	Identifies how control passes from one activity to the next in the application. See Section 18.1.3, "About Control Flows" for more information.
	Wildcard Control Flow Rule	Represents a control flow case that can originate from any activities whose IDs match a wildcard expression. For example, it can represent a control case `from-activity-id` containing a trailing wildcard such as `foo*`. See Section 18.4.2, "How to Add a Wildcard Control Flow Rule" for more information.

Table 19–2 describes the annotations (notes and attachments) you can add to a task flow.

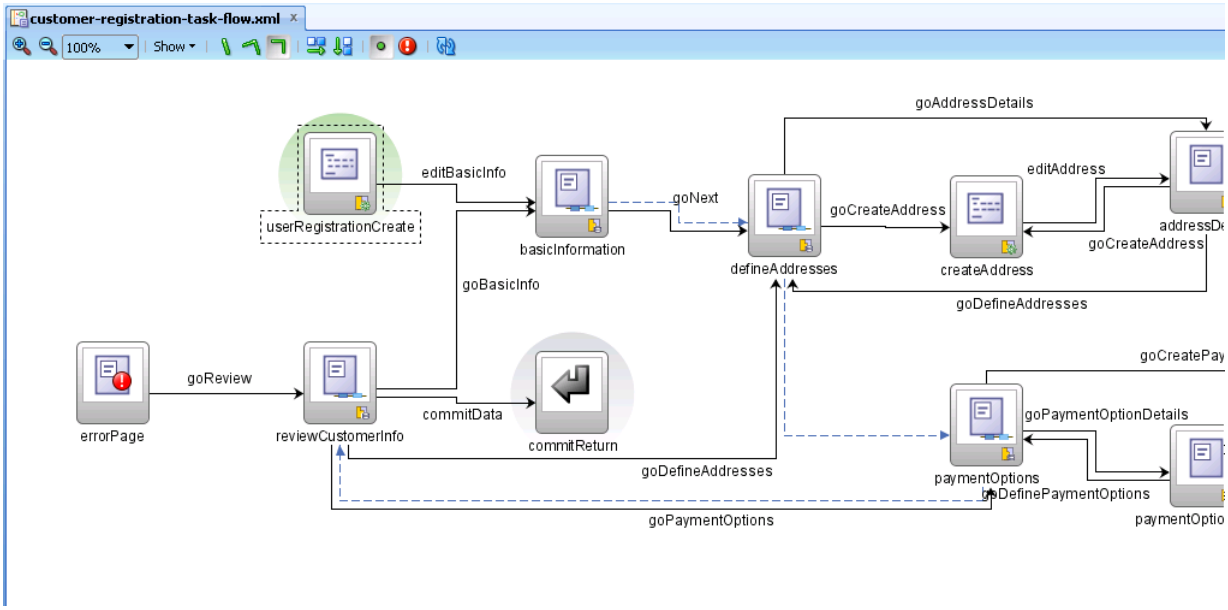
Table 19–2 Task Flow Diagram Annotations

Icon	Icon Name	Description
	Note	Adds a note to the task flow diagram. You can select the note in the diagram to add or edit text.
	Note Attachment	Attaches an existing note to an activity or a control flow case in the diagram.

19.1.1 Task Flow Activities Use Cases and Examples

Figure 19–2 shows the diagram for the customer registration task flow in the Fusion Order Demo application. This task flow uses many of the activities that a bounded task flow supports in order to register a customer. For more information about the Fusion Order Demo application, see Chapter 2, "Introduction to the ADF Sample Application."

Figure 19–2 Customer Registration Task Flow



19.1.2 Additional Functionality for Task Flow Activities

You may find it helpful to understand other ADF features before you configure or use task flow activities. Additionally, you may want to read about what you can do with your configured task flows. Following are links to other functionality that may be of interest.

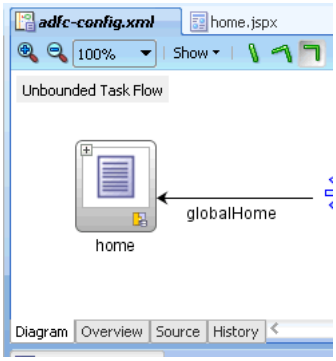
- Task flows can invoke managed beans. For more information about managed beans, see Section 24.4, "Using a Managed Bean in a Fusion Web Application."
- Bounded task flows can be secured by defining the privileges that are required for someone to use them. For more information, see Section 35, "Enabling ADF Security in a Fusion Web Application."
- Bounded task flows can be packaged in ADF Library JARs. For more information, see Section 38.3, "Packaging a Reusable ADF Component into an ADF Library."
- Task flow activities can be associated with page definition files. For more information about page definition files, see Section 13.7, "Working with Page Definition Files."

19.2 Using View Activities

One of the primary types of task flow activity is the view activity. A view activity displays a JSF page or page fragment. A *page fragment* is a JSF document that renders as content in another JSF page. Page fragments are typically used in bounded task flows that can be added to a JSF page as a region, as described in Section 21.2, "Creating an ADF Region."

Figure 19–3 shows the `home` view activity in the Fusion Order Demo application.

Figure 19–3 View Activity



> **Tip:** Click the + icon in the upper-left part of the view activity to see a thumbnail preview of the referenced page or page fragment.

XML metadata in the source file of the task flow associates a view activity with a physical JSF page or page fragment. An `id` attribute identifies the view activity. The `<page>` element identifies the page or page fragment's file name. Example 19–1 shows the metadata that corresponds to the view activity in Figure 19–3:

Example 19–1 Metadata for a View Activity

```
<view id="home">
   <page>/home.jspx</page>
 </view>
```

The view activity ID and page name do not have to be the same.

The steps for adding a view activity to a task flow are the same as those for adding any activity to a task flow diagram. For more information, see Section 18.3.1, "How to Add an Activity to a Task Flow." After you add the view activity, you can double-click it to display a wizard that allows you to create a new page or page fragment. The wizard allows you to choose between one of the following two document types for the JSF page or page fragments that you create:

- Facelets

  The file extension for pages in this document type is `.jsf` and `.jsff` for page fragments.

- JSP XML

  The file extension for pages in this document type is `.jspx` and `.jsff` for page fragments.

Use one document type only in your task flows. For example, do not use the Facelets document type for view activities in one task flow and the JSP XML document type for view activities in another task flow. For more information about the document types, see Section 1.3.7, "Implementing the User Interface with JSF."

You also use the wizard to choose the page layout for the page or page fragment that you create. For more information about page layouts, see the "Using Quick Start Layouts" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*. In addition, you can specify whether or not to automatically expose UI components on the page or page fragment in a managed bean. For more information, see Section 24.4, "Using a Managed Bean in a Fusion Web Application."

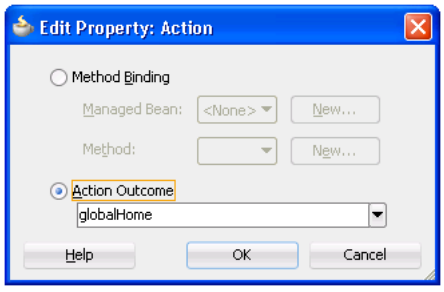
After you complete the wizard, JDeveloper automatically associates the completed page or page fragment with the view activity. As an alternative to using the wizard, you can associate a page or page fragment with a view activity by dragging an existing page or page fragment from the Application Navigator and dropping it on top of an existing view activity. If no view activity exists, drag a page or page fragment to any other location on the diagram. When you drop the page or page fragment, JDeveloper automatically creates a new view activity associated with the page or page fragment. During creation, a default `id` for the view activity is automatically generated (for example `Home`) based on the name of the page or page fragment.

19.2.1 Passing Control Between View Activities

You can configure view activities in your task flow to pass control to each other at runtime. For example, to pass control from one view activity (view activity A) to a second view activity (view activity B), you could configure a command component (a button or a link) on the page associated with view activity A. Set the command component's **Action** attribute to the control flow case `from-outcome` that corresponds to the task flow activity that you want to invoke (in our example, view activity B). At runtime, the end user initiates the control flow case by invoking the command component. You can navigate from a view activity to another activity using either a constant or dynamic value on the **Action** attribute of the UI component.

A constant value of the component's **Action** attribute is an action outcome, as shown in Figure 19–4, that always triggers the same control flow case. When an end user clicks the component, the activity specified in the control flow case is performed. There are no alternative control flows.

Figure 19–4 Edit Property dialog



A dynamic value of the component's **Action** attribute is bound to a managed bean or a method. The value returned by the method binding determines the next control flow case to invoke. For example, a method might verify user input on a page and return one value if the input is valid and another value if the input is invalid. Each of these different action values trigger different navigation cases, causing the application to navigate to one of two possible target pages.

You can also write an EL expression that must evaluate to `true` before control passes to the target view activity. You write the EL expression as a value for the `<if>` child element of the control flow case in the task flow. For more information, see Section 18.4.1, "How to Add a Control Flow Rule to a Task Flow."

19.2.1.1 How to Pass Control Between View Activities

You pass control to a view activity by specifying the value of the control flow case's `from-outcome` attribute as the value for the `action` attribute of the command component.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you configure the passing of control between view activities. For more information, see Section 19.2.1, "Passing Control Between View Activities."

You may also find it helpful to understand functionality that can be added using other ADF features. For more information, see Section 19.1.2, "Additional Functionality for Task Flow Activities."

The following procedure assumes that you have already created a:

- Target view activity to receive control after an end user invokes the command component at runtime
- Page that hosts a command component for the end user to invoke at runtime

To pass control to a view activity:

1. Add a UI component to the JSF page using one of the following techniques:
   - Open the JSF page. In the ADF Faces page of the Component Palette, from the General Controls panel, drag a navigation UI component, such as **Button**, and drop it on the page.
   - Open the JSF page. From the Data Controls panel, drag and drop an operation or a method onto the JSF page and choose **Rich Command Button** or **Rich Command Link** from the context menu.
2. Select the UI component and open the Property Inspector.
3. On the Common page, expand the **Button Action** section.
4. From the **Action** dropdown menu, choose **Edit**.
5. Select **Action Outcome**.
6. From the **Action Outcome** dropdown list select a value.

   The list contains control flow case `from-outcomes` already defined for the view activity associated with the page.

   Tips: The `action` attribute of the UI component can be bound either to a literal string to hardcode a navigation case, or it can be bound to a method binding expression that points to a method, which takes no arguments and returns a String. It can't be bound to any other type of EL expression.

7. Click **OK**.

19.2.1.2 What Happens When You Pass Control Between View Activities

Example 19–2 contains an example of a control flow case defined in the XML source file for a bounded or unbounded task flow.

Example 19–2 Control Flow Case Defined in XML Source File

```
<control-flow-rule>
    <from-activity-id>Start</from-activity-id>
       <control-flow-case>
          <from-outcome>toOffices</from-outcome>
          <to-activity-id>WesternOffices</to-activity-id>
       </control-flow-case>
</control-flow-rule>
```

As shown in Example 19–3, a button on a JSF page associated with the Start view activity specifies `toOffices` as the **action** attribute. When the user clicks the button, control flow passes to the `WesternOffices` activity specified as the `to-activity-id` in the control flow metadata.

Example 19–3 Static Navigation Button Defined in a View Activity

```
<af:commandButton text="Go" action="toOffices">
```

19.2.2 Bookmarking View Activities

Bookmarking is available only for view activities within unbounded task flows.

When an end user bookmarks a page associated with a view activity, the URL that displays in the browser's address field for the view is saved as the bookmark. In most cases, this URL cannot be used to redisplay the page associated with the view. For example, the URL may contain Microsoft OS Windows state information that cannot be used to redisplay the page.

The bookmark URL should contain information that enables dynamic content on the page to be reproduced. For example, if an end user bookmarks a page displaying a customer's contact information, the bookmark URL needs to contain not only the page but also some identifier for the customer. This will enable contact information for the same customer to display when he returns to the page using the bookmark.

To ensure that the URL for a page displayed in a browser can be used as a bookmark, identify the view activity associated with the page as bookmarkable.

At runtime, you can identify if a view activity within the unbounded task flow has been designated as bookmarkable using the `ViewBookmarkable()` method. The method is located off the view port context.

After you designate a view activity as bookmarkable, you can optionally specify one or more URL parameters. The value of `url-parameter` is an EL expression. The EL expression specifies where the parameters that will be included in the URL are retrieved when the bookmarkable URL is generated. The EL expression also stores a value from the URL when the bookmarkable URL is dereferenced. The `converter` option identifies a method that performs conversion and validation when parameters are passed via bookmarkable view activity URLs.

In addition, you can specify an optional method that is invoked after updating the application model with submitted URL parameter values and before rendering the view activity. You can use this method to retrieve additional information based on URL parameter key values.

Example 19–4 contains the URL syntax for a bookmarked view activity.

Example 19–4 Unbounded Task Flow View Activity URL Syntax

```
<server root>/<app_context>/faces/<view activity id>?<param name>=<param
value>&...
```

The syntax of the URL for the bookmarked view activity is:

- `<server root>`: Provided by customization at site or admin level, for example, `http://mycompany.com/internalApp`.

- `<app context>`: The web application context root, for example, `myapp`. The context root is the base path of a web application. For example, `<app_context>` maps to the physical location of the WEB-INF node on the server.
- `faces`: The faces servlet mapping. The value in `faces` points to the node containing the `faces-config.xml` configuration file.
- `<view activity id>`: The identifier for the bookmarked view activity, for example, `edit-customers`.
- `<param name>`: The name of the bookmarked view activity URL parameter, for example, `customer-id`.
- `<param value>`: The parameter value, derived from an EL expression, for example, `#{pageFlowScope.employee.id}`. The value of the EL expression must be capable of being represented as a string.

Example 19–5 contains a sample URL for a bookmarkable view activity in an unbounded task flow.

Example 19–5 Sample URL for Bookmarkable View Activity

```
http://mycompany.com/internalApp/MyApp/faces/edit-customers?customer-id=1234&...
```

19.2.2.1 How to Create a Bookmarkable View Activity

To create a bookmarkable view activity, designate a view activity as bookmarkable, specify a URL parameter in the bookmark, and specify a method that is executed after the bookmark is dereferenced.

Before you begin:

It may be helpful to have an understanding of the syntax required to create a bookmarkable view activity. For more information, see Section 19.2.2, "Bookmarking View Activities."

You may also find it helpful to understand functionality that uses other task flow features. Section 19.1.2, "Additional Functionality for Task Flow Activities."

To designate a view activity as bookmarkable:

1. In the unbounded task flow diagram, select the view activity.
2. In the Property Inspector, click **Bookmark**.
3. In the **bookmark** dropdown list, select **true**.
4. Expand the **URL Parameters** section to add optional URL parameters that will be included in the URL for the bookmarked view activity:
   - **name**: A name for the parameter.
   - **value**: A settable EL expression that, when evaluated, specifies the parameter value, for example, `#{pageFlowScope.employeeID}`. The value must be capable of being represented as a string.
   - **converter**: (optional): An EL expression to an object that implements `oracle.adf.controller.URLParameterConverter`.

   The **value** is where the parameters that will be included in the URL are retrieved from when the bookmarkable URL is generated. In addition, parameters are stored here when the bookmarkable URL is dereferenced.

If the EL expression entered in value returns `NULL`, the parameter is omitted from the bookmarked view activity URL.

The **name** and **value** are used to append a bookmark parameter to the view activity URL, as shown in Example 19–5.

5. In the **converter** field, you can enter an optional value binding to use for each bookmark URL parameter value, for example, `#{pageFlowScope.employee.idConverter}`.

   A URL parameter converter's `getAsObject()` method takes a single string value as its input parameter and returns an object of the appropriate type. ADF Controller invokes the converter method on the URL parameters before applying the parameter value to the application's model objects. Similarly, the converter's `getAsString()` method takes an object as its input parameter and returns a string representation that is used on the URL.

   In a JSF application, data values are converted and validated using the converters and validators specified with the UI components on the submitting page. In a Fusion web application using a bookmark URL, there is no submitting page to handle the conversion and validation. Therefore, you have the option of designating a converter to use for each URL parameter.

19.2.2.2 What Happens When You Designate a View as Bookmarkable

When you designate a view activity as bookmarkable, a bookmark element is added to the metadata for the view activity, as shown in Example 19–6. The bookmark element can optionally contain metadata specifying URL parameters and a method that is executed after the bookmark is dereferenced.

Example 19–6 Sample Metadata for a Bookmarkable View Activity

```
<view id="employee-view">
  <page>/folderA/folderB/display-employee-info.jspx</page>
     <bookmark>
        <url-parameter>
          <name>employee-id</name>
          <value>#{pageFlowScope.employee.id}</value>
          <converter>#{pageFlowScope.employee.validateId}</converter>
        </url-parameter>
       <method>#{pageFlowScope.employee.queryData}</method>
     </bookmark>
</view>
```

19.2.3 Specifying HTTP Redirect for a View Activity

You can configure a view activity so that it redirects at runtime to a URL in response to a client request. This causes ADF Controller to create a new browser URL for the view activity. The original URL for the view activity is no longer used. You cannot configure HTTP redirect for a view activity if you have already made the view activity bookmarkable, as described in Section 19.2.2, "Bookmarking View Activities."

19.2.3.1 How to Specify HTTP Redirect for a View Activity

You set a view activity's `redirect` property to `true`.

Before you begin:

It may be helpful to have an understanding of the affect of setting a view activity's `redirect` property to `true`. For more information, see Section 19.2.3, "Specifying HTTP Redirect for a View Activity."

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Section 19.1.2, "Additional Functionality for Task Flow Activities."

To specify HTTP redirect for a view activity:

1. In the task flow diagram, select the view activity.
2. In the Property Inspector, expand the **General** section.
3. In the **Redirect** dropdown list, select **true**.

19.2.3.2 What Happens When You Specify HTTP Redirect for a View Activity

The ADF Controller issues an HTTP redirect in response to a view activity request. The redirected request creates a new browser URL for the view activity. The original view URL is no longer used.

When specified, a HTTP redirect occurs in response to a client `GET` request. For a client `GET`, the `#{bindings}` EL scope is invalid until the ADF Controller and ADF Model layer set up a new bindings context for the page. Therefore, you cannot use EL expressions, such as the following, for view activities for which you specified HTTP redirect:

```
#{bindings.foo}
```

Note: If you want `http://www.mycompany.org/x.html` to instead display what is at `http://www.mycompany.org/y.html`, do not use refresh techniques such as:

```
<META HTTP-EQUIV=REFRESH CONTENT="1;
URL=http://www.example.org/bar">
```

This technique could adversely affect back button behavior. If an end user clicks a browser back button, the refresh occurs again, and navigation is forward, not backward as expected.

In this situation, use HTTP redirect instead.

19.3 Using URL View Activities

You can use a URL view activity to redirect the root view port (for example, a browser page) to any URL-addressable resource, even from within the context of an ADF region. URL addressable resources include:

- Bounded task flows
- View activities in the unbounded task flow
- Addresses external to the current web application (for example, `http://www.oracle.com`)

To display the resource, you specify an EL expression that evaluates at runtime to generate the URL to the resource. In addition, you can specify EL expressions that, when evaluated, are added as parameters and parameter values to the URL.

A URL view activity redirects the client regardless of the view port (root view port or an ADF region) from which it is executed. The `<redirect>` element of a view activity performs in a similar way, except that it can be used only if the view activity is within the root view port. The `<redirect>` element is ignored within the context of an ADF region. For more information, see Section 19.2.3, "Specifying HTTP Redirect for a View Activity".

Redirecting elsewhere within the same application using URL view activities (not the `<redirect>` element) is handled similarly to back button navigation since the task flow stack is cleaned up. Redirecting out of the web application is handled like dereferencing a URL to a site external to the application.

19.3.1 How to Add a URL View Activity to a Task Flow

You can add a URL view activity to a bounded or unbounded task flow.

Before you begin:

It may be helpful to have an understanding of the affect of adding a URL view to the functionality of a task flow. For more information, see Section 19.3, "Using URL View Activities."

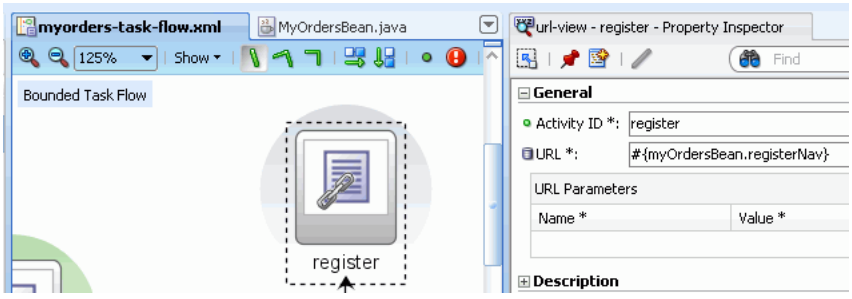
You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Section 19.1.2, "Additional Functionality for Task Flow Activities."

To add a URL view activity to a task flow:

1. In the ADF Task Flow page of the Component Palette, from the Component panel, drag and drop a URL view activity onto the diagram.
2. In the task flow diagram, select the URL view activity.
3. On the General page of the Property Inspector, in the **Activity ID** field, enter an ID that identifies the URL view activity.
4. Click the button next to the **URL** field to invoke the Expression Builder and write an EL expression that renders a URL at runtime.

   For example, Figure 19–5 shows a URL activity (**register**) in the Fusion Order Demo application's `myorders-task-flow.xml` bounded task flow with an EL expression (`#{myOrdersBean.registerNav}`) that retrieves a URL at runtime.

Figure 19–5 URL View Activity



5. Expand the **URL Parameters** section to add optional URL parameters that will be included in the URL:
   - **name**: A name for the parameter.

- **value**: An EL expression that, when evaluated, generates the parameter value.
- **converter**: A settable EL expression that, when evaluated, specifies a method to perform conversion and validation when parameters are passed via bookmarkable view activity URLs. For more information, see Section 35.3, "Enabling ADF Security".

19.3.2 What You May Need to Know About URL View Activities

Task flow URL view activities can be used within JSF portlets. Create the URL using one of the following options if you configure a task flow URL view activity for use within a JSF portlet:

- Invoke one of the following methods from the `ControllerContext` class in the `oracle.adf.controller` package:
  - `getLocalViewActivityURL()`
  - `getGlobalViewActivityURL()`

  Do not invoke the `encodeActionURL()` method from the `ADFPortletContainerExternalContext` class with the response from the `getLocalViewActivityURL()` or `getGlobalViewActivityURL()` methods before you invoke the `redirect()` method from the `ADFPortletContainerExternalContext` class. The `getLocalViewActivityURL()` and `getGlobalViewActivityURL()` methods both perform the necessary encoding of the URL.
- Supply a fully-qualified absolute URL.
- Supply a context path relative URL.
- Supply a URL relative to the current view.

A task flow URL view activity in a JSF portlet behaves as follows:

- If the redirect URL refers to a location in the JSF portlet application and does not contain a `queryString` parameter named `x_DirectLink` with a value of `true`, the portlet in the containing page navigates to the URL specified in the URL view activity.
- Otherwise, the client redirects to the URL specified in the URL view activity.

For more information about using a task flow's URL view activity, see Section 19.3, "Using URL View Activities."

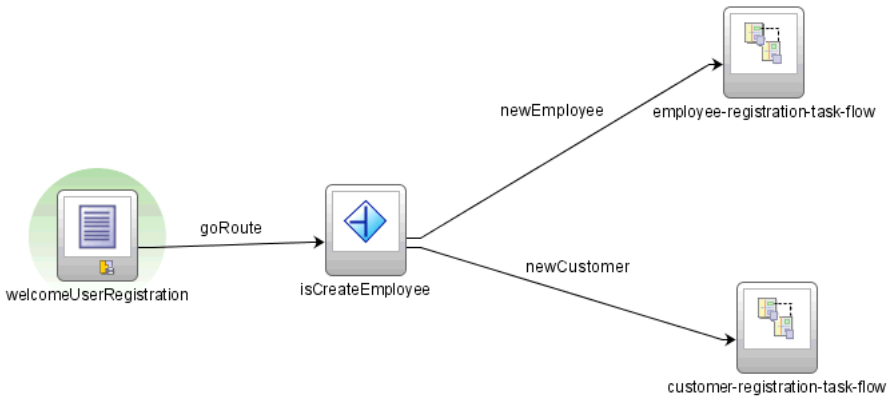
For information about the methods that you can use to get a URL, see the following reference documents:

- *Oracle Fusion Middleware Java API Reference for Oracle ADF Controller*

19.4 Using Router Activities

Use a router activity to route control to activities based on the runtime evaluation of EL expressions. Figure 19–6 shows a router activity (**isCreateEmployee**) can branch to different control flows leading from it to different activities.

Figure 19–6 Router for Alternate Control Flow Cases



Each control flow corresponds to a different router case. Each router case uses the following elements to choose the activity to which control is next routed:

- `expression`: an EL expression that evaluates to `true` or `false` at runtime.

  The router activity returns the outcome that corresponds to the EL expression that returns `true`.

- `outcome`: a value returned by the router activity if the EL expression evaluates to `true`, for example, `newCustomer`.

  If the router case `outcome` matches a `from-outcome` on a control flow case, control passes to the activity that the control flow case points to. If none of the cases for the router activity evaluate to `true`, or if no router activity cases are specified, the `outcome` specified in the router **Default Outcome** field (if any) is used.

For example, if you want to configure control flow on whether a user clicks the **Create a New Customer** or **Create a New Employee** button on the `welcomeUserRegistration` page fragment view activity in Figure 19–6. You add a router case where the EL expression evaluates if the user entered a new customer in the input text field on the `welcomeUserRegistration` page fragment and an outcome (for example, `newCustomer`) for the router activity to return if the EL expression returns `true` (the user entered a new customer)

In Figure 19–6, if the EL expression to determine if a user entered a new customer evaluates to `true`, control passes to the `customer-registration-task-flow` task flow call activity, based on the control flow case `from-outcome`.

Best Practice:

Use a router activity if your routing condition can be expressed in an EL expression.

Using a router activity allows you to do more when you are designing the task flow that contains it. The router activity allows you to show more information about the condition on the task flow. This makes it more readable and useful to someone who looks at the diagram for your task flow.

Using a router activity also makes it easier to modify your application later. For example, you may want to modify or add a routing condition later.

19.4.1 How to Configure Control Flow Using a Router Activity

You define a control flow by dragging a router activity from the Component Palette to the diagram for the task flow. You configure the properties of the router activity (Activity ID and Default Outcome) and add router cases to the router activity.

Before you begin:

It may be helpful to have an understanding of the properties of a router activity that can affect functionality. For more information, see Section 19.4, "Using Router Activities."

You may also find it helpful to understand other functionality that can be added using other task flow activity features. For more information, see Section 19.1.2, "Additional Functionality for Task Flow Activities."

To configure control flow using the router activity:

1. In the ADF Task Flow page of the Component Palette, drag and drop a router activity onto the diagram.
2. In the task flow diagram, select the router activity.
3. From JDeveloper's main menu, choose **View** > **Property Inspector**.
4. In the Property Inspector, expand the **General** category and set values for the following properties:
   - **Activity ID**: write a value that identifies the router activity in the task flow's source file.
   - **Default Outcome**: specify an activity that the router activity passes control to if no control flow cases evaluate to `true` or if no control flow case is specified.
5. Click the **Add** icon next to **Cases** and specify values for each router case that you add, as described in the following list:
   - **Expression**: An EL expression that evaluates to `true` or `false` at runtime.

     For example, to reference the value in an input text field of a view activity, write an EL expression similar to the following:

     ```
     #{pageFlowScope.value=='view2'}
     ```

     If this EL expression returns `true`, the router activity invokes the outcome that you specify in the **Outcome** field.

- **Outcome**: the outcome the router activity invokes if the EL expression specified by **Expression** returns `true`.

  Create a control flow case or a wildcard control flow rule for each outcome in the diagram of your task flow. For example, for each outcome in a control flow case, make sure that there is a corresponding `from-outcome`. In Figure 19–6, the value for both the case `outcome` and the control flow case element `from-outcome` is `newCustomer`. This makes sure that control flow passes to the `newCustomer` activity, the target of the control flow element.

19.4.2 What Happens When You Configure Control Flow Using a Router Activity

JDeveloper writes values to the source file of the task flow based on the values that you specify for the properties of the router activity. Example 19–7 shows the values that JDeveloper writes where values are specified for the router activity as follows:

- The value of Default Outcome is `toRegion3`.
- Two control flow cases are specified. If the input value is `1`, the outcome is `toRegion1` and if input value is `2`, the outcome is `toRegion2`.

At runtime, the router activity passes control to the outcome in the control flow case where the EL expression returned `true`.

Example 19–7 Router Activity Metadata in a Task Flow Source File

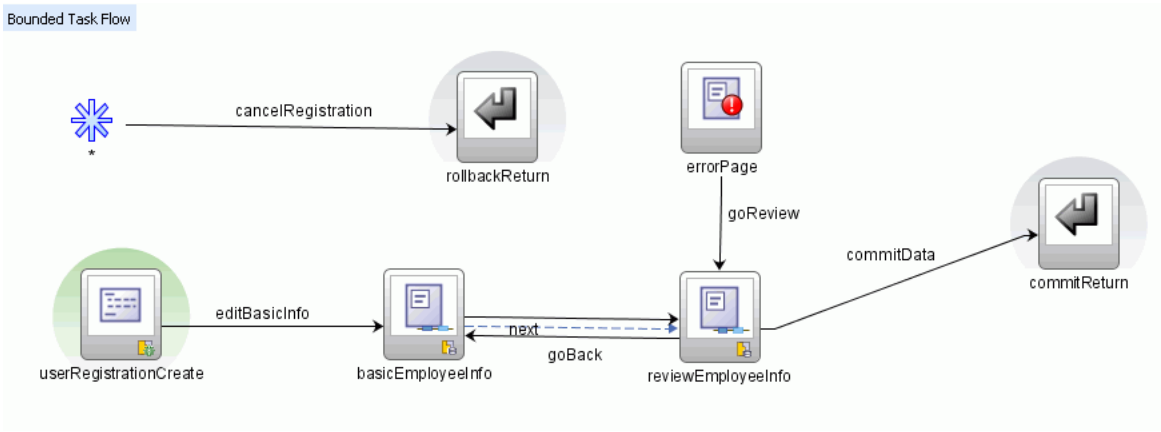
```
<default-activity id="__3">Router1</default-activity>
    <router id="Router1">
      <case>
        <expression>#{binding.Region.InputValue='1'}</expression>
        <outcome id="__4">toRegion1</outcome>
      </case>
      <case>
        <expression>#{binding.Region.InputValue='2'}</expression>
        <outcome id="__5">toRegion2</outcome>
      </case>
      <default-outcome>toRegion3</default-outcome>
    </router>
```

19.5 Using Method Call Activities

A *method call* activity allows you to call a custom or built-in method that invokes application logic from anywhere within an application's control flow. You can specify methods to perform tasks such as initialization before displaying a page, cleanup after exiting a page, exception handling, and so on.

Figure 19–7 shows where the Fusion Order Demo application uses a method call activity in the Employee Registration bounded task flow. The activity calls `userRegistrationCreate`, a method exposed on the `StoreServiceAM` data control.

Figure 19–7 Method Call Activity in employee-registration-task-flow



You can set an outcome for the method that specifies a control flow case to pass control to after the method finishes. You can specify the outcome as either:

- `fixed-outcome`: On successful completion, the method always returns this single outcome, for example, `success`. If the method does not complete successfully, an outcome is not returned. If the method type is void, you must specify a `fixed-outcome`, not a `to-string`.
- `to-string`: If specified as true, the outcome is based on calling the `toString()` method on the Java object returned by the method. For example, if `toString()` returns `editBasicInfo`, navigation goes to the `editBasicInfo` control flow case shown in Figure 19–7.

As shown in Example 19–8, the method outcome and the method result are two different values. The `<return-value>` element specifies where to put the result of the `calculateSalesTax` method. The `<outcome>` element indicates which control flow case to use after the method finishes.

Example 19–8 Method Call Activity Metadata with Return and Outcome Elements

```
<method-call id="calculateSalesTax">
      <method>#{pageFlowScope.taxService.calculateSalesTax}</method>
      <return-value>#{pageFlowScope.result}</return-value>
      <outcome>
         <fixed-outcome>gotoError</fixed-outcome>
       </outcome>
 </method-call
```

For more information about control flows, see Section 18.1.3, "About Control Flows."

Best Practice:

You can use a method call on a task flow to invoke a method before a page renders, or you can use an `invokeAction` on a page definition.

If you want your method to execute before the application renders the page, it is usually best to use a method call activity in the task flow rather than an `invokeAction` in the page definition file. By adding your method as a method activity on a task flow, it is easier to invoke logic between pages. This allows you to do more at the time you're designing the task flow. You can also show more information on the task flow, thus making it more readable and useful to someone else who looks at your diagram.

You might want to use an `invokeAction` instead of a method call for one of the following reasons:

- You want the method to be executed in more than one phase of the page's lifecycle.
- You plan to reuse the page and page definition file, and want the method to be tied to the page.
- You are not using ADF Controller.

19.5.1 How to Add a Method Call Activity

Drag a method call activity from the Component Palette to the task flow diagram. You can associate the method call activity with an existing method by dropping a data control operation from the Data Controls panel directly onto the method call activity in the task flow diagram.

In the Fusion Order Demo application, for example, you could drag a `setCurrentRowWithKey` or `setCurrentRowWithKeyValues` operation to the diagram from the Data Control Iterator to display or select the current row in a table.

Note: Parameters for data control method parameters are defined in the page definition for the corresponding page rather than within ADF Controller metadata.

You can also drag methods and operations directly to the task flow diagram. A new method call activity is created automatically after you drop it on the diagram. You can specify an EL expression and other options for the method.

Tip: To identify the method that a method call activity invokes, right-click the method call activity in the diagram of the task flow and choose **Go to Method**. JDeveloper navigates to the method that the method call activity invokes.

Before you begin:

Create a bounded or unbounded task flow. For more information, see Section 18.2, "Creating a Task Flow."

It may be helpful to have an understanding of how the properties of a method call activity affect the functionality of a task flow. For more information, see Section 19.5, "Using Method Call Activities."

You may also find it helpful to understand other functionality that can be added using other task flow activity features. For more information, see Section 19.1.2, "Additional Functionality for Task Flow Activities."

To add a method call activity to a task flow:

1. In the ADF Task Flow page of the Component Palette, from the Component panel, drag and drop a method call activity onto the diagram.

   The method call activity optionally displays a default ID, `methodCall`*n*, and a warning icon that indicates that a method EL expression has not yet been specified.

   For more information about turning on the warning icons, see Section 18.3.1, "How to Add an Activity to a Task Flow".

2. If you want to change the default ID, click the text that appears under the method call activity in the task flow diagram.

   You can enter a name for the method call, for example, `addItemToCart`.

3. In the task flow diagram, select the method call activity.

4. On the General page of the Property Inspector, enter an EL expression for the method in the **Method** field.

   For example, enter an EL binding expression similar to the following:

   ```
   #{bindings.addItemstoCart.execute}
   ```

   Note: The `bindings` variable in the EL expression references an ADF Model binding from the current binding container. In order to specify the bindings variable, you must specify a binding container definition or page definition. For more information, see Section 13.7, "Working with Page Definition Files".

   You can also use the Edit Property dialog box shown in Figure 19–8 to build the EL expression for the method:

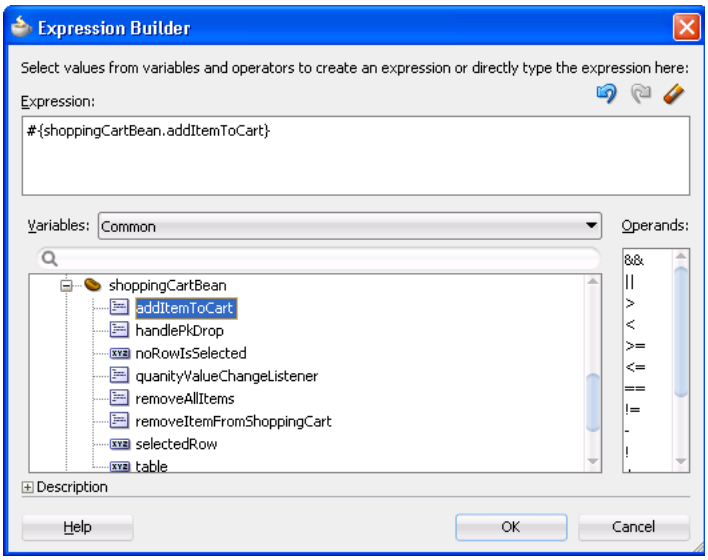
   a. In the General page of the Property Inspector, from the **Method** dropdown menu, choose **Expression Builder**.

   b. In the Expression Builder dialog, expand a node, for example, **ADF Bindings** and choose a method. Or, under the **ADF Managed Beans** node, navigate to the managed bean containing the method you want to call and select the method.

   c. Click **Insert Into Expression**.

      The Expression Builder dialog should look similar to Figure 19–8. In Figure 19–8, for example, the `addItemToCart` method and `shoppingCartBean` combine to form the EL expression shown at the top of the Expression Builder.

Figure 19–8 EL Expression for Method in Expression Builder Dialog



d. Click **OK**.

Tip: If the method call activity is going to invoke a managed bean method, double-click the method call activity in the diagram for the task flow. This invokes a dialog where you can specify the managed bean method you want to invoke.

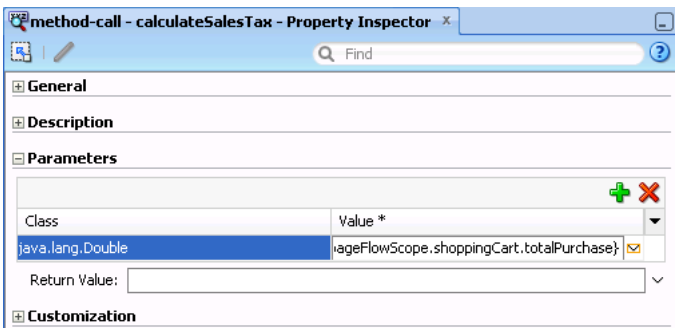
5. In the General page of the Property Inspector, specify one of the following in the Outcome section:

   - **Fixed Outcome**: On successful completion, the method always returns this single outcome, for example, `success`. If the method does not complete successfully, an outcome is not returned. If the method type is void, you must specify a value for **Fixed Outcome** rather than for **to-string**.
   - **tostring()**: If you select **true**, the outcome is based on calling the `toString()` method on the Java object returned by the method.

19.5.2 How to Specify Method Parameters and Return Values

You can specify parameters and return values for a method. Figure 19–9 shows a single parameter defined for a method called `calculateSalesTax`. The **Value** field contains an EL expression that evaluates to the parameter value at runtime.

Figure 19–9 Method Parameters in Property Inspector



If parameters have not already been created by associating the method call activity to an existing method, add the parameters yourself.

Before you begin:

It may be helpful to have an understanding of how the properties of a method call activity affect the functionality of a task flow. For more information, see Section 19.5, "Using Method Call Activities."

You may also find it helpful to understand other functionality that can be added using other task flow activity features. For more information, see Section 19.1.2, "Additional Functionality for Task Flow Activities."

To add method parameters:

1. Add a method call activity to the task flow diagram.

   For more information, see Section 19.5.1, "How to Add a Method Call Activity."

2. In the task flow diagram, select the method call activity.
3. In the Property Inspector, click **Parameters**.
4. On the Parameter page, expand the **Parameters** section.
5. Click the plus **Add** icon.
6. In the **class** field, enter the parameter class, for example, `java.lang.Double`.
7. In the **value** field, enter an EL expression indicating where the value for the parameter will be retrieved, for example, `#{pageFlowScope.shoppingCart.totalPurchasePrice}`.

   Tip: You can click the icon next to the **value** field and choose **Expression Builder** to search for the method parameters.

8. In the **return-value** field, enter an EL expression indicating where to store the method return value, for example, `#{pageFlowScope.Return}`.
9. Click **OK**.
10. Repeat the above steps to add additional parameters.

19.5.3 What Happens When You Add a Method Call Activity

After you specify parameters and return values for a method, the XML source file is updated. Example 19–9 shows how a method call to `userRegistrationCreate` appears in the XML source file for a bounded task flow.

Example 19–9 Call to userRegistrationCreate method

```
<method-call id="userRegistrationCreate">
     <method>#{bindings.userRegistrationCreate.execute}</method>
     <outcome>
        <fixed-outcome>editBasicInfo</fixed-outcome>
     </outcome>
<method-call>
```

19.6 Using Task Flow Call Activities

You can use a task flow call activity to call a bounded task flow from either the unbounded task flow or a bounded task flow. A task flow call activity allows you to call a bounded task flow located within the same or a different application.

The called bounded task flow executes its default activity first. There is no limit to the number of bounded task flows that can be called. For example, a called bounded task flow can call another bounded task flow, which can call another, and so on.

To pass parameters into a bounded task flow, you must specify input parameter values on the task flow call activity. These values must correspond to the input parameter definitions on the called bounded task flow. For more information, see Section 19.6.3, "How to Specify Input Parameters on a Task Flow Call Activity".

Also note the following:

- The value on the task flow call activity **Input Parameter** specifies where the value will be taken from within the calling task flow.
- The value on the **Input Parameter Definition** for the called task flow specifies where the value will be stored within the called bounded task flow once it is passed.

  Tip: When a bounded task flow is associated with a task flow call activity, input parameters are automatically inserted on the task flow call activity based on the input parameter definitions defined on the bounded task flow. Therefore, the application developer needs only to assign values to the task flow call activity input parameters.

By default, all objects are passed by reference. Primitive types (for example, int, long, or boolean) are always passed by value.

The technique for passing return values out of the bounded task flow to the caller is similar to the way that input parameters are passed. See Section 20.5, "Specifying a Return Value from a Bounded Task Flow" for more information.

19.6.1 How to Call a Bounded Task Flow Using a Task Flow Call Activity

Add a task flow call activity to the calling bounded or unbounded task flow to call a bounded task flow.

Before you begin:

It may be helpful to have an understanding of how a task flow call activity interacts with a task flow. For more information, see Section 19.6, "Using Task Flow Call Activities."

You may also find it helpful to understand other functionality that can be added using other task flow activity features. For more information, see Section 19.1.2, "Additional Functionality for Task Flow Activities."

To call a bounded task flow:

1. Open the calling task flow in the diagram editor.
2. In the ADF Task Flow page of the Component Palette, from the Component panel, drag and drop a **Task Flow Call** activity onto the diagram.
3. Identify the called task flow using one of the following techniques:
   - In the task flow diagram, double-click the task flow call activity.

The Create Bounded Task Flow dialog displays, where you specify options for creating a new bounded task flow.

- Drag an existing bounded task flow from the Application Navigator and drop it on the task flow call activity.

  Tips: You can drop a bounded task flow on a page or page fragment. If the bounded task flow consists of pages (not page fragments), you can choose to add a Go Link or Go Button UI component on the page where you drop the task flow. An end user can click the button or link to call the task flow. This may in turn automatically generate the task flow call activity if the page is already associated with an existing view activity in a task flow.

  You cannot drop a bounded task flow from one application to a task flow diagram contained in another application using the Application Navigator, even though both applications appear in the navigator. In addition, you cannot drop a bounded task flow contained in one project onto a task flow diagram contained in another project.

  Instead, you can package the bounded task flow in an ADF library, then reuse it in your current application or project. You can then drag the bounded task flow from the Resource Catalog or from the Component Palette page that is created when you import the library. For more information, see Section 38.2.2, "Using the Resource Palette".

- If you know the name of the bounded task flow that you want to invoke, carry out the following steps:

  a. In the task flow diagram, select the task flow call activity.

  b. On the General page of the Property Inspector, select **Static** from the Task Flow Reference dropdown list.

  c. In the **Document** field, enter the name of the source file for the bounded task flow to call. For example, `called-task-flow-definition.xml`.

  d. In the **ID** field, enter the bounded task flow ID contained in the XML source file for the called bounded task flow, for example, `targetTaskFlow`.

- If you do not know the name of the bounded task flow to invoke and it is dependent on an end user selection at runtime, carry out the following steps:

  a. In the task flow diagram, select the task flow call activity.

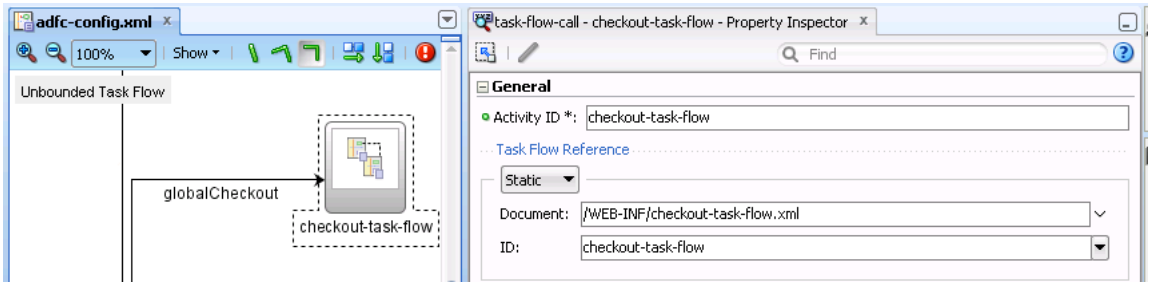
  b. On the General page of the Property Inspector, select **Dynamic** from the Task Flow Reference dropdown list.

  c. Select **Expression Builder** from the **Dynamic Task Flow Reference** dropdown menu.

  d. Write an EL expression that identifies the ID of the bounded task flow to invoke at runtime.

  Figure 19–10 shows the checkout-task-flow task flow call activity in an unbounded task flow of the Fusion Order Demo application. This task flow call activity invokes the checkout-task-flow located in the `checkout-task-flow.xml` file.

Figure 19–10 Task Flow Call Activity That Invokes a Bounded Task Flow



19.6.2 What Happens When You Call a Bounded Task Flow Using a Task Flow Call Activity

JDeveloper generates metadata entries in the source file for the task flow that calls the bounded task flow. Example 19–10 shows an example from the Fusion Order Demo application's `adfc-config.xml` that references the checkout bounded task flow (`checkout-task-flow.xml`). At runtime, the task flow call activity calls the checkout bounded task flow.

Example 19–10 Task Flow Call Activity in Fusion Order Demo Application's adfc-config.xml File

```
<task-flow-call id="checkout-task-flow">
    <task-flow-reference>
      <document>/WEB-INF/checkout-task-flow.xml</document>
      <id>checkout-task-flow</id>
    </task-flow-reference>
  </task-flow-call>
```

19.6.3 How to Specify Input Parameters on a Task Flow Call Activity

The suggested method for mapping parameters between a task flow call activity and its called bounded task flow is to first specify input parameter definitions for the called bounded task flow. Then you can drag the bounded task flow from the Application Navigator and drop it on the task flow call activity. The task flow call activity input parameters will be created automatically based on the bounded task flow's input parameter definition. For more information, see Section 20.3, "Passing Parameters to a Bounded Task Flow".

You can, of course, first specify input parameters on the task flow call activity. Even if you have defined them first, they will automatically be replaced based on the input parameter definitions of the called bounded task flow, once it is associated with the task flow call activity.

If you haven't yet created the called bounded task flow, you may still find it useful to specify input parameters on the task flow call activity. Doing so at this point allows you to identify any input parameters you expect the task flow call activity to eventually map when calling a bounded task flow.

Before you begin:

It may be helpful to have an understanding of how a task flow call activity interacts with a task flow. For more information, see Section 19.6, "Using Task Flow Call Activities."

You may also find it helpful to understand other functionality that can be added using other task flow activity features. For more information, see Section 19.1.2, "Additional Functionality for Task Flow Activities."

To specify input parameters on a task flow call activity:

1. Select the task flow call activity in the diagram for the task flow.
2. In the Property Inspector, expand the **Parameters** section and click the **Add** icon to enter a name for the parameter.

   Tip: Dropping a bounded task flow on a task flow call activity in a diagram automatically populates the `name` field.

3. Enter a parameter value, for example, `#{pageFlowScope.callingTaskflowParm}`.

   The value specifies where the parameter value will be taken from within the calling task flow.

   By default, all objects are passed by reference. Primitive types (for example, `int`, `long`, or `boolean`) are always passed by value.

4. After you have specified an input parameter, you can specify a corresponding input parameter definition for the called bounded task flow. For more information, see Section 20.3, "Passing Parameters to a Bounded Task Flow".

19.6.4 How to Call a Bounded Task Flow Using a URL

You can call a bounded task flow that does not use page fragments (`.jsff`) in another web application using a URL. You use a task flow call activity to call the bounded task flow that you want to invoke. You write an EL expression for the task flow call activity's `remote-app-url` property that, when evaluated, returns a URL.

In addition to writing a value for the `remote-app-url` property, you specify values for task flow reference properties that identify the bounded task flow to call. The task flow reference and the `remote-app-url` property are combined at runtime to generate a URL to the called bounded task flow in the remote web application.

You also need to set visibility properties for the bounded task flow in the remote web application that you want to call so that it invokes when it receives the URL generated from the task flow call activity in the calling task flow.

Be aware that JSF portlets provide all content from the same web application. As a result, do not configure your web application to invoke a remote task flow using a URL if plan to use your web application in a JSF portlet.

Note: If, in JDeveloper's Application Navigator, you right-click the bounded task flow that you want to call using a URL and select **Run** from the context menu, the bounded task flow executes as if it were called using a URL at runtime. For this reason, make sure to set the visibility properties for the bounded task flow even if you want to execute it as part of testing in JDeveloper.

Before you begin:

It may be helpful to have an understanding of how a task flow call activity interacts with a task flow. For more information, see Section 19.6, "Using Task Flow Call Activities."

You may also find it helpful to understand other functionality that can be added using other task flow activity features. For more information, see Section 19.1.2, "Additional Functionality for Task Flow Activities."

To call a bounded task flow using a URL:

1. Open the task flow in your web application that you want to configure to invoke a bounded task flow in a remote web application.
2. In the ADF Task Flow page of the Component Palette, from the Component panel, drag a **Task Flow Call** activity and drop it on the diagram for the task flow.
3. In the Property Inspector, expand the **General** section and write values for the properties to invoke a bounded task flow.

   For more information, see Section 19.6.1, "How to Call a Bounded Task Flow Using a Task Flow Call Activity".
4. For the **Remote Application URL** property, use the Expression Builder to write an EL expression that, when evaluated, returns a string with the parts required to construct a URL of the remote web application.

   For example, the following example EL expression invokes a managed bean method that returns a string with the parts required to construct a URL:

   ```
   #{myOrdersBean.createOrder}
   ```
5. Open the bounded task flow that you specified values for in step 3.

   Note: The bounded task flow you specify cannot use page fragments (`.jsff`).
6. In the Structure window, right-click the node for the bounded task flow (task flow definition) and choose **Go to Properties**.
7. In the Property Inspector, expand the **General** category and set values for the following properties:

   - **URL Invoke**: select `url-invoke-allowed` from the dropdown list if you want to allow a URL to invoke the bounded task flow. Select `url-invoke-disallowed` if you do not want to allow a URL to invoke the bounded task flow. Selecting this value returns a HTTP 403 status code if a URL attempts to invoke the bounded task flow. The default value (`calculated`) allows a URL to invoke the bounded task flow if the bounded task flow does not specify an initializer and it has a view activity as its default activity. If the bounded task flow does not meet these conditions, a HTTP 403 status code is returned. Selecting `url-invoke-allowed` or `url-invoke-disallowed` overrides the default behavior.
   - **Library Internal:** set to `true` if you want the bounded task flow to be internal when you package it in an ADF Library JAR. The default value is `false`.

     For more information about packaging a bounded task flow in an ADF Library JAR, see Section 38.3, "Packaging a Reusable ADF Component into an ADF Library".

8. Save and close the bounded task flow.

19.6.5 What Happens When You Configure a Bounded Task Flow to be Invoked by a URL

JDeveloper generates metadata entries in the source file of the task flow that invokes the task flow call activity to the bounded task flow in a remote web application. Example 19–11 shows an example entries for a task flow call activity.

Example 19–11 Metadata Entries for a Task Flow Call Activity to a Bounded Task Flow

```
<task-flow-call id="createOrder">
    <task-flow-reference>
      <document id="__6">myorders-task-flow.xml</document>
      <id id="__5">myorders-task-flow</id>
    </task-flow-reference>
    <remote-app-url id="__7">#{myOrdersBean.createOrder}</remote-app-url>
  </task-flow-call>
```

The `createOrder` method in Example 19–11 returns a string with the URL syntax required to invoke a bounded task flow. For more information about the URL syntax, including descriptions of the required parts in the returned string and an example URL, see Section 19.6.6, "What You May Need to Know About Calling a Bounded Task Flow Using a URL."

JDeveloper also generates entries in the source file for the bounded task flow to invoke when you configure it so it can be called by a URL. Example 19–12 shows sample metadata entries that allow a bounded task flow to be invoked by a URL.

Example 19–12 Metadata Entries to Allow a URL Invoke a Bounded Task Flow

```
<task-flow-definition id="task-flow-definition3">
    <visibility id="__2">
      <url-invoke-allowed/>
      <library-internal/>
    </visibility>
  </task-flow-definition>
```

19.6.6 What You May Need to Know About Calling a Bounded Task Flow Using a URL

Adding a context parameter in your local application´s deployment descriptor may ease the administration of interaction with a remote web application.

Context Parameter for Remote Web Application

Consider adding a context parameter to the deployment descriptor (`web.xml`) for your Fusion web application (local application) if you use a URL to invoke a bounded task flow in another Fusion web application (remote application). Set the value of the context parameter to the URL of the remote application. Use the context parameter name when writing EL expressions in the local application, as shown in the following example where `remoteAppUrl` is the name of the context parameter:

```
#{initParam.remoteAppUrl}
```

If the URL of the remote application changes, you can update the context parameter to reference the changed URL.

URL Syntax to Invoke a Bounded Task Flow

Typically, you write an EL expression that references a managed bean method which, in turn, retrieves the required parts of the URL or you could write an EL expression that renders the URL directly, as shown in Example 19–13.

Example 19–13 Example URL to Invoke a Bounded Task Flow

```
http://somecompany.com/internalApp/MyApp/faces/adf.task-flow?adf.tfId=displayHelp&
adf.tfDoc=%2FWEB-INF%2Fdisplayhelp.xml&topic=createPurchaseOrder
```

Example 19–14 describes the parts of the URL syntax to invoke a bounded task flow.

Example 19–14 URL Syntax for a Call to a Bounded Task Flow Using Named Parameters

```
<server root>/<app_context>/faces/adf.task-flow?adf.tfid=<task flow definition
ID>&adf.tfDoc=<document name>&<named parameter>=<named parameter value>
```

The following list describes each part of the URL syntax in Example 19–14:

- `<server root>`: Provided by customization at site or admin level. For example:

  `http://mycompany.com/internalApp`

  The `<server root>` value depends on the application server where you deploy the bound task flow. The bounded task flow URL is a resource within the JSF servlet's URL path.
- `<app context>`: The Web application context root, for example, `MyApp`. The context root is the base path of a Web application.
- `faces`: Faces servlet mapping.
- `adf.task-flow`: A reserved keyword that identifies the ADF Controller for the remote web application.
- `adf.tfId`: A URL parameter that supplies the task flow ID to call.
- `<task flow ID>`: The identifier of the bounded task flow to call, for example, `displayHelp`. This is the same task flow ID that is used when calling locally. Note that this identifier is not the same as the task flow call activity instance ID. The parameter value must be represented as a string.
- `adf.tfDoc`: A URL parameter that supplies the document name containing the bounded task flow ID to be called.
- `<document name>`: A document name containing the bounded task flow ID to be called, for example,`%2FWEB-INF%2FtoUppercase%2FdisplayHelp.xml`. If you are handcrafting the bounded task flow URL, you are responsible for the appropriate encoding.
- `<named parameter>`: (optional) The name of an input parameter definition for the called bounded task flow, for example, `topic`. You must supply all required input parameter definitions.
- `<named parameter value>`: (optional) The value of the input parameter.

> **Note:** URL parameter names that begin with an underscore ('_') are intended for internal use only and should not be used. Although you may see these names on URLs generated by ADF Controller, you should not attempt to use or depend on them.

Object Type for Parameter Converters

Parameter converters, if specified, can be used to convert task flow parameter values to and from the string representations used in a URL. A parameter converter is an EL expression that evaluates to an object of the following type:

`oracle.adf.controller.UrlParameterConverter`

If you do not specify a parameter converter, a default converter checks the parameters for cross-site-scripting (XSS) attacks. If you know that the parameter values used in your application contain special characters, you should create your own implementation of `UrlParameterConverter` and use it to perform conversion of the task flow parameter values.

19.6.7 How to Specify Before and After Listeners

Task flow call activity before and after listeners are used to identify the start and end of a bounded task flow. Specifying a listener in the task flow call activity means that the listener executes on that specific usage of the called bounded task flow.

You specify the listener as an EL expression for a method that will be called upon entry or exit of a bounded task flow, for example, `<before-listener>#{global.showState}</before-listener>`. The method cannot have parameters.

- Before listener: An EL expression for a Java method called before a bounded task flow is entered. It is used when the caller needs to know when a bounded task flow is being initiated.
- After listener: An EL expression for a Java method called after a bounded task flow returns. It is used when the caller needs to know when a bounded task flow exits and control flow returns to the caller.

If multiple before listeners or multiple after listeners are specified, they are called in the order in which they appear in the source document for the unbounded or bounded task flow. A task flow call activity can have only have before listener and one after listener.

In order for the task flow call after listeners to be called, control flow must return from the bounded task flow using a control flow rule. If an end user leaves a bounded task flow using the browser back button or other URL, task flow call after listeners will not be called. You must use a bounded task flow finalizer to release all acquired resources and perform cleanup of a bounded task flow that the end user left by clicking a browser back button.

Before you begin:

It may be helpful to have an understanding of how a task flow call activity interacts with a task flow. For more information, see Section 19.6, "Using Task Flow Call Activities."

You may also find it helpful to understand other functionality that can be added using other task flow activity features. For more information, see Section 19.1.2, "Additional Functionality for Task Flow Activities."

To specify a before or after listener on a task flow call activity:

1. In the diagram of the calling bounded task flow, select the task flow call activity.
2. In the Property Inspector, click **Listeners**.
3. Click the button next to either **before-listener** or **after-listener**.

4. In the Expression Builder dialog, drill down to the Java class containing the method for the listener.
5. Open the class node and select the listener method.

   When you are done, your EL expression might look like `#{pageFlowscope.managedBean.methodListener}`.
6. Click **OK**.

19.6.8 What Happens When You Add a Task Flow Call Activity

After you add a task flow call activity to a task flow diagram, you must specify a reference to the called bounded task flow using one of the methods described in Section 19.6.1, "How to Call a Bounded Task Flow Using a Task Flow Call Activity". For example, if you drop an existing bounded task flow on the task flow call activity, JDeveloper generates the task flow reference automatically. The task flow reference is used to invoke the called bounded task flow.

If the task flow reference is static, each task flow reference consists of:

- id: The bounded task flow id contained in the XML source file for the called bounded task flow. For example, a called task flow might have an ID called `targetFlow`. The same XML source file can contain multiple bounded task flows, each bounded task flow identified by a unique ID.

  Note: If you use JDeveloper to create the bounded task flow, there is only one bounded task flow per document.

- document: The name of the XML source file containing the ID of the called bounded task flow. If **document** is not specified, `adfc-config.xml` is assumed.

  The document is a physical XML file and must be accessible via MDS.

Example 19–15 contains an example static task flow reference within a task flow call activity. In order to invoke a bounded task flow, you need to know its id and name of the file containing the id.

Example 19–15 Static Task Flow Reference

```
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2" id="__
1">
...
  <task-flow-definition id="task-flow-definition">
    <default-activity>view1</default-activity>
    <task-flow-call id="taskFlowCall">
      <task-flow-reference>
        <document>/WEB-INF/called-task-flow-definition.xml</document>
        <id>called-task-flow-definition</id>
      </task-flow-reference>
    </task-flow-call>
  </task-flow-definition>
...
</adfc-config>
```

Example 19–16 shows the metadata that JDeveloper generates for a dynamic task flow reference within a task flow call activity.

Example 19–16 Dynamic Task Flow Reference

```
<?xml version="1.0" encoding="windows-1252" ?>
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2"
             id="__1">
  <task-flow-definition id="bounded_tf">
    <default-activity id="__2">taskFlowCall1</default-activity>
    <task-flow-call id="taskFlowCall1">
      <dynamic-task-flow-reference id="__3">#{EL_Expression_Retrieve_
                           TaskflowID}</dynamic-task-flow-reference>
    </task-flow-call>
  </task-flow-definition>
</adfc-config>
```

19.6.9 What Happens at Runtime When a Task Flow Call Activity Invokes a Task Flow

The ADF Controller performs the following steps when a bounded task flow is called using a task flow call activity:

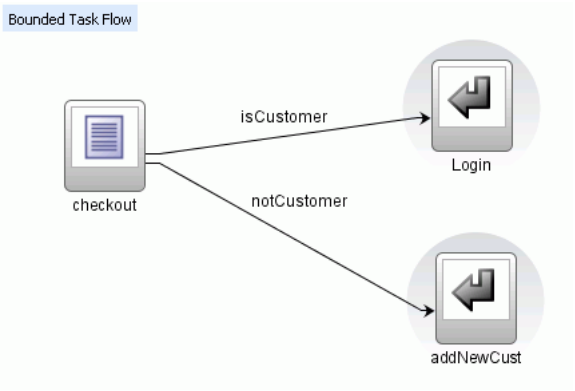
1. Verifies that the user is authorized to call the bounded task flow.
2. Invokes task flow call activity before listener or listeners, if specified (see Section 19.6.7, "How to Specify Before and After Listeners").
3. Evaluates the input parameter values on the bounded task flow.
4. Pushes the called bounded task flow onto the stack and initializes its page flow scope.
5. Sets input parameter values in the called bounded task flow's context.
6. Invokes a bounded task flow initializer method, if one is specified.
7. Executes the bounded task flow's default activity.

19.7 Using Task Flow Return Activities

A task flow return activity identifies the point in an application's control flow where a bounded task flow completes and sends control flow back to the caller. You can use a task flow return activity only within a bounded task flow.

A gray circle around a task flow return activity icon indicates that the activity is an exit point for a bounded task flow. A bounded task flow can have zero to many task flow return activities. Figure 19–11 shows a bounded task flow with task flow return activities named **Login** and **addNewCust**.

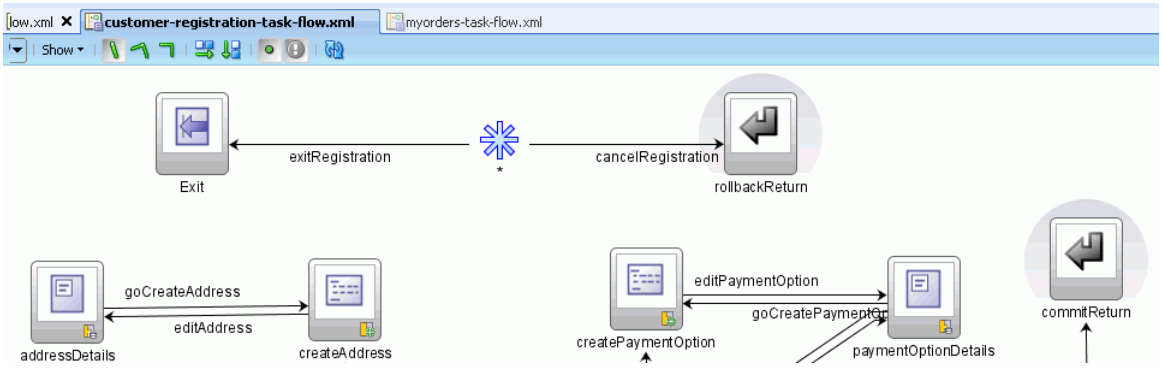
Figure 19–11 Multiple Task Flow Return Activities



Each task flow return activity specifies an `outcome` that is returned to the calling task flow. For example, the `outcome` for the `rollbackReturn` task flow return activity in the Fusion Order Demo application customer registration bounded task flow is `cancelCreateCust`.

Figure 19–12 shows an extract of the customer registration bounded task flow (`customer-registration-task-flow.xml`). This task flow contains two task flow return activities (`rollbackReturn` and `commitReturn`). The `rollbackReturn` task flow return activity's `outcome` is `cancelCreateCust`. In contrast, the `commitReturn` task flow return activity's `outcome` returns `globalhome`.

Figure 19–12 Task Flow Return Activities Specifying Different Outcomes



The `outcome` returned to the task flow that invokes the customer registration task flow depends on the end user action. You can configure control flow cases in the invoking task flow to determine the next action by the invoking task flow. Set the `from-outcome` element of a control flow case to the value returned by the task flow return activity's `outcome` to invoke an action based on that outcome. This determines control flow upon return from the customer registration task flow.

Set a value for the **Restore Save Point** property to specify if model changes made in a bounded task flow are saved or discarded when the bounded task flow exits by using a task flow return activity. Set to `true` to roll back transactions to the ADF Model save point that was created when the Fusion web application first entered the bounded task flow. The default value is `false`. You can specify a value for this property only if the bounded task flow on which the task flow return activity is located was entered without starting a new transaction. For more information, see Section 22.2.1, "How to Enable Transactions in a Bounded Task Flow".

Before you begin:

It may be helpful to have an understanding of how a task flow return activity interacts with a task flow. For more information, see Section 19.7, "Using Task Flow Return Activities."

You may also find it helpful to understand other functionality that can be added using other task flow activity features. For more information, see Section 19.1.2, "Additional Functionality for Task Flow Activities."

To add a task flow return activity to a bounded task flow:

1. In the ADF Task Flow page of the Component Palette, from the Component panel, drag and drop a task flow return activity onto the diagram for the bounded task flow.
2. In the task flow diagram, select the task flow return activity.
3. On the Common page of the Property Inspector, expand the **Outcome** section.
4. In the **Name** field, enter an outcome, for example, `preferredCustomer`.

   Specifying this returns an outcome to the caller when the bounded task flow exits. You can specify only one outcome per task flow return activity. The calling task flow should define control flow rules to handle control flow upon return. See Section 18.4.1, "How to Add a Control Flow Rule to a Task Flow" for more information.
5. In the Property Inspector, expand the **Behavior** section.
6. In the **Reentry** dropdown list, choose one of the following options:
   - **reentry-allowed**: Reentry is allowed on any view activity within the bounded task flow.
   - **reentry-not-allowed**: Reentry of the bounded task flow is not allowed.

     If you specify **reentry-not-allowed** on a bounded task flow, an end user can still click the browser back button and return to a page within the bounded task flow. However, if the user does anything on the page such as clicking a button, an exception (for example, `InvalidTaskFlowReentry`) is thrown indicating the bounded task flow was reentered improperly. The actual reentry condition is identified upon the submit of the reentered page.

   Your selection defines the default behavior when the bounded task flow is reentered by an end user clicking a browser's Back button. This selection applies only if **reentry-outcome-dependent** has been set on the bounded task flow where the task flow return activity is located. For more information, see Section 22.3, "Reentering Bounded Task Flows".
7. In the **End Transaction** dropdown list, choose one of the following options:
   - **commit**: Select to commit the existing transaction to the database.
   - **rollback**: Select to roll back the transaction to what it was on entry of the called task flow. This has the same effect as canceling the transaction, since it rolls back a new transaction to its initial state when it was started on entry of the bounded task flow.

     If you do not specify commit or rollback, the transaction is left open to be closed by calling bounded task flow.
8. In the **Restore Save Point** dropdown list, select **true** when either of the following conditions apply:

- If **Always Begin New Transaction** (`new-transaction`) is not selected on the bounded task flow on which the task flow return activity is located
- ADF model changes made within a bounded task flow should be discarded when exiting using the task flow call activity. The transaction is rolled back to the save point created on entry of the bounded task flow.

For more information, see Section 22.2.1, "How to Enable Transactions in a Bounded Task Flow".

19.8 Using Save Point Restore Activities

The save point restore activity allows you to restore a previous persistent save point in an application supporting save for later functionality. A save point captures a snapshot of the Fusion web application at a specific instance. Save point restore enables the application to restore whatever was captured when the save point was originally created.

When a save point is restored, ADF Controller terminates the saved application and restarts the application that was executing when the end user performed a save. The end user's original location in the application is displayed. Once the `save-point-id` is restored, it is deleted from its method of persistence (database or Java object cache).

A save point restore activity is not required within every individual application supporting save for later capabilities. It is only required within the applications responsible for restoring the previously persistent `save-point-ids`. For example, a save point restore activity would not be required within a Create Expense Report application, but would be within the application used to select previously saved expense reports for further updates.

Section 22.6, "Using Save Points in Task Flows" contains detailed information about enabling save for later capabilities in a task flow and provides an example of how to use the save point restore activity to retrieve the saved application state and data.

19.9 Using Parent Action Activities

A bounded task flow running in an ADF region may need to trigger navigation of its parent view activity. The parent action activity allows a bounded task flow running in an ADF region to generate outcomes that it passes to the parent view activity. The outcomes are used to navigate the task flow containing the parent view activity rather than navigating the task flow of the ADF region.

For more information, see Section 21.7.1, "How to Trigger Navigation Outside of an ADF Region's Task Flow."

19.10 Using Task Flow Activities with Page Definition Files

Page definition files define the binding objects that populate data at runtime. They are typically used in a Fusion web application to bind page UI components to data controls. A number of task flow activities can also use page definition files to bind to data controls. These are:

- Method call

  You can drag and drop a data control operation from the Data Control panel onto a task flow to create a method call activity or onto an existing method call activity. In both cases, the method call activity binds to the data control operation.

- Router

Associating a page definition file with a router activity creates a binding container. At runtime, this binding container makes sure that the router activity references the correct binding values when it evaluates the router activity cases' EL expressions. Each router activity case specifies an `outcome` to return if its EL expression evaluates to `true`. For this reason, only add data control operations to the page definition file that evaluate to `true` or `false`.

- Task flow call

  Associating a page definition file with a task flow call activity creates a binding container. At runtime, the binding container is in context when the task flow call activity passes input parameters. The binding container makes sure that the task flow call activity references the correct values if it references binding values when passing input parameters from a calling task flow to a called task flow.

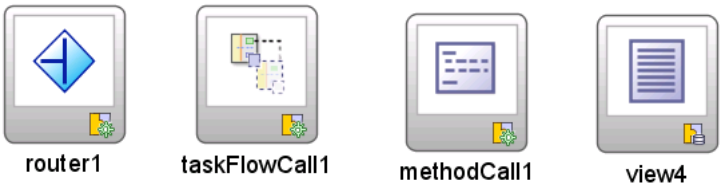
- View

  You cannot directly associate a view activity with a page definition file. Instead, you associate the page that the view activity references.

If you right-click any of the above task flow activities (except view activity) in the diagrammer for a task flow, JDeveloper displays an option on the context menu that enables you to create a page definition file (**Create Page Definition**) if one does not yet exist. If a page definition file does exist, JDeveloper displays a context menu option for all task flow activities to go to the page definition file (**Go to Page Definition**). JDeveloper also displays a context menu option (**Edit Binding**) when you right-click a method call activity that is associated with a page definition file.

A task flow activity that is associated with a page definition file displays an icon in the lower-right section of the task flow activity icon. Figure 19–13 shows an example for each of the task flow activities.

Figure 19–13 Task Flow Activities Associated with Page Definition Files



19.10.1 How to Associate a Page Definition File with a Task Flow Activity

JDeveloper provides a context menu option that you can access from the task flow activity. You use this context menu option to associate the task flow activity with a page definition file.

Before you begin:

It may be helpful to have an understanding of how task flow activities use page definition files. For more information, see Section 19.10, "Using Task Flow Activities with Page Definition Files."

You may also find it helpful to understand other functionality that can be added using other task flow activity features. For more information, see Section 19.1.2, "Additional Functionality for Task Flow Activities."

To associate a page definition file with a task flow activity

1. In the diagrammer for the task flow, right-click the task flow activity for which you want to create a page definition file.
2. Choose **Create Page Definition** from the context menu that appears.
3. In the resulting page definition file, add the bindings that you want your task flow activity to reference at runtime.

   For more information about page definition files, see Section 13.7, "Working with Page Definition Files".

19.10.2 What Happens When You Associate a Page Definition File with a Task Flow Activity

At design time, JDeveloper generates a page definition file for the task flow activity. The filename of the page definition file comprises the originating task flow and either the name of the task flow activity or the data control operation to invoke. For example, `taskflowName_taskflowName_methodCall1PageDef.xml` or `taskflowName_taskflowName_CreateInsertPageDef.xml`.

JDeveloper also generates an EL expression from the task flow activity to the binding in the created page definition file. Example 19–17 shows a method call activity that references a `CreateInsert` action binding.

Example 19–17 Task Flow Activity Referencing an Action Binding

```
<method-call id="CreateInsert">
      <method>#{bindings.CreateInsert.execute}</method>
      <outcome>
        <fixed-outcome>CreateInsert</fixed-outcome>
      </outcome>
    </method-call>
```

At runtime, a binding container makes sure that a task flow activities' EL expressions reference the correct value.

20

Using Parameters in Task Flows

This chapter describes how to specify parameters in view activities and in bounded task flows.

This chapter includes the following sections:

- Section 20.1, "About Using Parameters in Task Flows"
- Section 20.2, "Passing Parameters to a View Activity"
- Section 20.3, "Passing Parameters to a Bounded Task Flow"
- Section 20.4, "Sharing Data Controls Between Task Flows"
- Section 20.5, "Specifying a Return Value from a Bounded Task Flow"

20.1 About Using Parameters in Task Flows

A task flow´s ability to accept input parameters and return parameter values allow you to manipulate data in task flows and share data between task flows. Using these abilities, you can optimize the reuse of task flows in your Fusion web application.

You can use view activity input page parameters as aliases. The alias allows you to map bounded task flow input parameters to page parameters. The view activity input page parameters map managed beans and any information available to the calling task flow to a managed bean on the page itself. To pass values out of view activities, store values in page flow scope or managed beans. For information about using view activities in a task flow, see Section 19.2, "Using View Activities".

For example, a page might specify `#{pageFlowScope.empNo}` as a page parameter and a bounded task flow might specify `#{pageFlowScope.employeeID}` as the value of an input parameter definition.

The `from-value` on the view activity input page parameter would be `#{pageFlowScope.employeeID}` and the `to-value` would be `#{pageFlowScope.empNo}`. This enables reuse of both the page definition and bounded task flow because you do not have to redefine parameters for every context in which each is used.

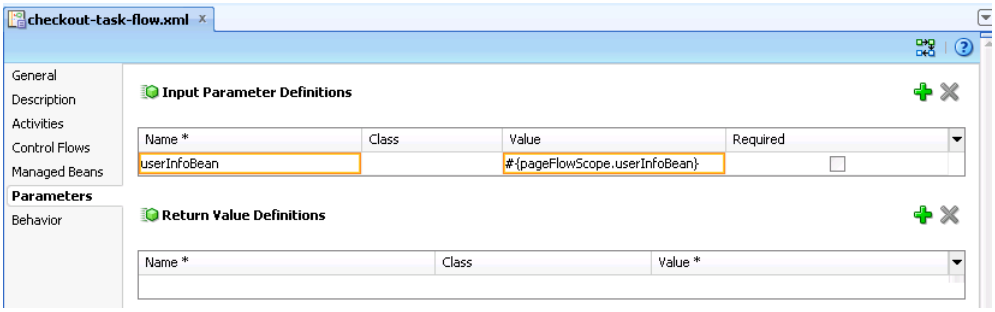
Other values contained within the task flow can be mapped to input page parameters, not just bounded task flow input parameter definition values.

20.1.1 Task Flow Parameters Use Cases and Examples

Figure 20–1 shows how the check out bounded task flow in the Fusion Order Demo application defines input parameters to receive data about a customer who wants to purchase a product. For more information about the Fusion Order Demo, including

how to access the source files, see Chapter 2, "Introduction to the ADF Sample Application."

Figure 20–1 Check Out Task Flow



20.1.2 Additional Functionality for Task Flows Using Parameters

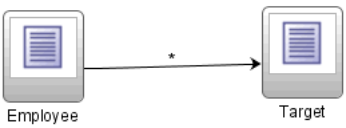
You may find it helpful to understand other ADF features before you configure or use a task flow with parameters. Additionally, you may want to read about what you can do with your configured task flows. Following are links to other functionality that may be of interest.

- Data controls can be shared between task flows. For more information about data controls, see Chapter 13.3, "Exposing Application Modules with Oracle ADF Data Controls."
- Task flows can invoke managed beans. For more information about managed beans, see Section 24.4, "Using a Managed Bean in a Fusion Web Application."
- Bounded task flows can be secured by defining the privileges that are required for someone to use them. For more information, see Chapter 35, "Enabling ADF Security in a Fusion Web Application."

20.2 Passing Parameters to a View Activity

Figure 20–2 illustrates how to specify an input page parameter mapping. You can pass a parameter to the Employee activity as a `pageFlowScope` value or a value on a managed bean. The Employee activity can pass a value to the Target activity by placing it within `pageFlowScope` or a managed bean to the Target activity, based on the EL expression you specified in the `to-value`.

Figure 20–2 Task Flow with Two Activities



20.2.1 How to Pass Parameters to a View Activity

You add one or more input page parameters to the view activity that you want to pass parameters to.

Before you begin:

It may be helpful to have an understanding of the configuration options available to you before you define an input page parameter for a view activity. For more information, see Section 20.2, "Passing Parameters to a View Activity."

You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Section 20.1.2, "Additional Functionality for Task Flows Using Parameters."

To define an input page parameter for a view activity:

1. In the Application Navigator, double-click the source file for the task flow that contains the view activity to which you want to pass a parameter.
2. Select the view activity in the diagram for the task flow and, in the Property Inspector, expand the **Page Parameters** category.
3. In the Property Inspector, click **Page Parameters**.
4. In the **Input Page Parameter** section, click the **Add** icon and enter values as follows:
   - **From Value**: write an EL expression that, when evaluated, specifies where the view activity retrieves the value for the input page parameter. For example, write an EL expression similar to the following:

     ```
     #{pageFlowScope.EmployeeID}
     ```

     You can configure the view activity to retrieve a value that is input to the task flow if you write an EL expression for **From Value** that corresponds to the value for the input parameter that you defined for the task flow, as described in Section 20.3.1, "How to Pass an Input Parameter to a Bounded Task Flow."
   - **To Value**: write an EL expression, that when evaluated, specifies where the page associated with the view activity can retrieve the value of the input page parameter. For example, write an EL expression similar to the following:

     ```
     #{pageFlowScope.EmployeeNo}
     ```

20.2.2 What Happens When You Pass Parameters to a View Activity

JDeveloper writes entries to the source file of the task flow at design time, as illustrated in Example 20–1.

Example 20–1 Metadata to Define an Input Page Parameter on a View Activity

```
<view id="reducedAccess">
 <page>/secured/Information.jspx</page>
      <input-page-parameter>
        <from-value>#{res['infoUsage.reducedAccess.messageHeader']}</from-value>
        <to-value>#{pageFlowScope.infoPageHeaderText}</to-value>
      </input-page-parameter>
      <input-page-parameter>
        <from-value>#{res['infoUsage.reducedAccess.messageHeader']}</from-value>
        <to-value>#{pageFlowScope.infoPageMsg}</to-value>
      </input-page-parameter>
      <input-page-parameter>
        <from-value>info</from-value>
        <to-value>#{pageFlowScope.infoPageType}</to-value>
      </input-page-parameter>
    </view>
```

At runtime, the view activity retrieves the value of the input parameter from the location you specified in the `<from-value>` element. The view activity makes the value of the parameter available to its associated page in the location you specified in the `<to-value>` element.

20.2.3 What You May Need to Know About Specifying Parameter Values

You can specify parameter values using standard EL expression if you call a bounded task flow using a task flow call activity or you render the bounded task flow in an ADF region or an ADF dynamic region. For example, you can specify parameters using the following syntax for EL expressions:

- `#{bindings.bindingId.inputValue}`
- `#{bindings.bindingId}`
- `#{inputValue}`

Example 20–2 shows the metadata for a task flow binding that renders in an ADF region.

Example 20–2 ADF Region taskFlow Binding

```
<taskFlow id="Department1" taskFlowId="/WEB-INF/Department.xml#Department"
              xmlns="http://xmlns.oracle.com/adf/Controller/binding"
              Refresh="ifNeeded">
    <parameters>
        <parameter id="DepartmentId" value="#{bindings.DepartmentId.inputValue}"
                   xmlns="http://xmlns.oracle.com/adfm/uimodel"/>
     </parameters>
</taskFlow>
```

Appending `inputValue` to the EL expression makes sure that you assign the parameter the value of the binding rather than the actual binding object.

20.3 Passing Parameters to a Bounded Task Flow

A called bounded task flow can accept input parameters from the task flow that calls it. To pass an input parameter to a bounded task flow, you specify one or more:

- Input parameters on the task flow call activity in the calling task flow

  Input parameters specify where the calling task flow stores parameter values.

- Input parameter definitions on the called bounded task flow

  Input parameter definitions specify where the called bounded task flow can retrieve parameter values at runtime.

Specify the same name for the input parameter that you define on the task flow call activity in the calling task flow and the input parameter definition on the called bounded task flow. Do this so you can map input parameter values to the called bounded task flow.

If you do not specify an EL expression to reference the value of the input parameter, the EL expression for `value` defaults to the following at runtime:

`#{pageFlowScope.`*`parmName`*`}`

where *`parmName`* is the value you entered for the input parameter name.

In an input parameter definition for a called bounded task flow, you can specify an input parameter as required. If the input parameter does not receive a value at runtime

or design time, the task flow raises an error. An input parameter that you do not specify as required can be ignored during task flow call activity creation.

Task flow call activity input parameters can be passed by reference or passed by value when calling a task flow using a task flow call activity unless you are calling a task flow in an ADF region. If the task flow renders in an ADF region, the task flow call activity passes the input parameters by reference. By default, primitive types (for example, `int`, `long`, or `boolean`) are passed by value (`pass-by-value`).

The Pass By Value check box applies only to objects, not primitives and is used to override the default setting of passing by reference. Mixing the two, however, can lead to unexpected behavior in cases where parameters reference each other. If input parameter A on the task flow call activity is passed by value and if input parameter B on the task flow call activity is passed by reference, and B has a reference to A, the result can be two different instances of A and B.

Section 20.3.1, "How to Pass an Input Parameter to a Bounded Task Flow," describes how to pass an input parameter from a calling task flow to a called bounded task flow using a task flow call activity. Although you can pass parameter values from any activity on the calling task flow, the passed parameter in Section 20.3.1, "How to Pass an Input Parameter to a Bounded Task Flow" contain the value of an input text field on a page in the calling task flow.

If you call a bounded task flow using a URL rather than a task flow call activity, you pass parameters and values on the URL itself. For more information, see Section 19.6.4, "How to Call a Bounded Task Flow Using a URL".

Instead of explicitly passing data controls as parameters between task flows, you can simply share them by specifying the `data-control-scope` option on the called bounded task flow. For more information, see Section 20.4, "Sharing Data Controls Between Task Flows".

A called task flow can also return values to the task flow that called it when it exits. For more information about returning values from a bounded task flow, see Section 20.5, "Specifying a Return Value from a Bounded Task Flow."

20.3.1 How to Pass an Input Parameter to a Bounded Task Flow

You define values on the calling task flow and the called task flow.

Before you begin:

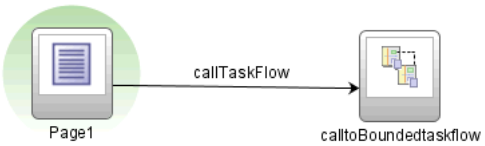
- Create a calling and called task flow

  The calling task flow can be bounded or unbounded. The called task flow must be bounded. For more information about creating task flows, see Section 18.2, "Creating a Task Flow.".

- Add a task flow call activity to the calling task flow

  Figure 20–3 shows an example where the view activity passes control to the task flow call activity.

Figure 20–3 Calling Task Flow



- It may be helpful to have an understanding of the configuration options available to you before you configure a bounded task flow to receive an input parameter. For more information, Section 20.3, "Passing Parameters to a Bounded Task Flow."
- You may also find it helpful to understand functionality that can be added using other task flow features. For more information, see Section 20.1.2, "Additional Functionality for Task Flows Using Parameters."

To pass an input parameter to a bounded task flow:

1. In the Application Navigator for your project, double-click the JSF page that contains an input component where an end user enters a value that gets passed to a bounded task flow as a parameter at runtime.

   The JSF page that you open should be referenced by a view activity in the calling task flow.

2. Select an input text component on the JSF page where an end user enters a value at runtime.

3. In the Property Inspector, enter a value for the input text component.

   You can specify the value as an EL expression, for example `#{pageFlowScope.inputValue}`.

4. In the Application Navigator, double-click the name of the called task flow to open its diagram.

5. Click the **Overview** tab for the called task flow.

6. In the overview editor, click the **Parameters** navigation tab and click the **Add** icon to define a new entry in the **Input Parameter Definition** section.

7. In the **Name** field, enter a name for the parameter, for example, `inputParm1`.

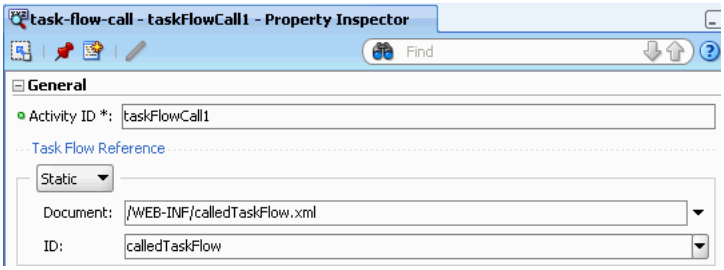
8. In the **Value** field, enter an EL expression where the parameter value is stored and referenced, for example, `#{pageFlowScope.inputValue}`.

9. In the editor, open the diagram for the calling task flow.

10. In the Application Navigator, drag the called bounded task flow and drop it on top of the task flow call activity that is located on the calling task flow.

    Dropping a bounded task flow on top of a task flow call activity in a diagram automatically creates a task flow reference to the bounded task flow. As shown in Figure 20–4, the task flow reference contains the bounded task flow ID and a document name. The bounded task flow ID (`id`) is an attribute of the bounded task flow's `<task-flow-definition>` element. The document name points to the source file for the task flow that contains the ID.

Figure 20–4 Task Flow Reference in Property Inspector



11. In the Property Inspector for the task flow call activity, click **Parameters** and expand the **Input Parameters** section.
12. Enter a name that identifies the input parameter.

    Because you dropped the bounded task flow on a task flow call activity having defined input parameters, the name should be already be specified. You must keep the same input parameter name.
13. Enter a parameter value, for example, `#{pageFlowScope.parm1}`.

    The value on the task flow call activity input parameter specifies where the calling task flow stores parameter values.

    The value on the input parameter definition for the called task flow specifies where the value will be retrieved from for use within the called bounded task flow once it is passed.
14. At runtime, the called task flow is able to use the input parameter. Since you specified `pageFlowScope` as the `value` in the input parameter definition for the called task flow, you can use the parameter value anywhere in the called bounded task flow. For example, you can pass it to a view activity on the called bounded task flow. For more information, see Section 19.2.1.2, "What Happens When You Pass Control Between View Activities".

20.3.2 What Happens When You Pass an Input Parameter to a Bounded Task Flow

JDeveloper writes entries to the source files for the calling task flow and called task flow based on the values that you select. Example 20–3 shows an input parameter definition specified on a a bounded task flow.

Example 20–3 Input Parameter Definition

```
<task-flow-definition id="sourceTaskflow">
...
   <input-parameter-definition>
      <name>inputParameter1</name>
      <value>#{pageFlowScope.parmValue1}</value>
      <class>java.lang.String</class>
   </input-parameter-definition>
...
</task-flow-definition>
```

Example 20–4 shows the input parameter metadata for the task flow call activity that calls the bounded task flow shown in Example 20–3. At runtime, the task flow call activity calls the bounded task flow and passes it the value specified by its `value` element.

Example 20–4 Input Parameter on Task Flow Call Activity

```
<task-flow-call id="taskFlowCall1">
...
    <input-parameter>
      <name>inputParameter1</name>
      <value>#{pageFlowScope.newCustomer}</value>
      <pass-by-value/>
    </input-parameter>
...
</task-flow-call>
```

20.4 Sharing Data Controls Between Task Flows

You can share data controls between task flows. A called bounded task flow can reference and modify the value of the data control owned by its calling task flow. This allows the called task flow to share the same data control instance as its parent. Both task flows look in the same place, the data control frame, to get the data control instance.

In the Fusion Order Demo application, for example, you may have a value that is used to display a row in a product table on a page. Clicking a button on the page updates a shopping cart form in an ADF region. The form updates with the product name, based on the selected value in the table.

A data control frame is the container associated with a task flow that contains data controls. A bounded task flow's transactions operate on data controls in the data control frame. A new `DataControlFrame` is created for each task flow that is entered. This results in each task flow having its own unique instance of any data control it uses.

To specify whether data controls are shared between the calling and called task flows, you must set a `data-control-scope` value of either `shared` or `isolated` on the called bounded task flow. The default value is `shared`.

If `data-control-scope` is set to `shared` on both the calling and called bounded task flow, the data control frame used will be the one for the calling task flow. In Example 20–5, the data control frame for Bounded Task Flow A will also be used by both B and C.

If `data-control-scope` is set to `isolated` on both the calling and called bounded task flow, the task flows both use their own data control frames.

Example 20–5 Sharing Data Controls

```
Bounded Task Flow A - isolated
  Bounded Task Flow B - shared
    Bounded Task Flow C - shared
```

Note: A task flow that is configured to share data controls cannot share data controls with the calling task flow if the calling task flow uses a URL to invoke the called task flow. ADF Controller throws an exception if this scenario occurs.

A caller of a bounded task flow can share its caller's data controls to any depth. For example, Task Flow A can share data controls with called bounded Task Flow B. Called Task Flow C can share the same data controls.

A bounded task flow that specifies `data-control-scope` as `shared` and is used within an ADF region shares the data controls of the task flow in the parent view port. For more information about view ports, see Section 21.1.2, "About View Ports and ADF Regions".

A new data control frame is created for the unbounded task flow in the call stack of each `RootViewPort`. Each browser window is isolated from all other browser windows within the same HTTP session. The browser windows do not share data controls. However, if an end user opens a new browser window, the two browser windows have the same server-side state and share data controls.

For performance reasons, data controls are not instantiated until needed. For more information about data controls, see Section 13.3, "Exposing Application Modules with Oracle ADF Data Controls".

20.4.1 How to Share a Data Control Between Task Flows

You share data controls between task flows by specifying a value for the `data-control-scope` element of the called task flow.

Before you begin:

It may be helpful to have an understanding of the properties that determine how you share data controls between task flows. For more information, see Section 20.4, "Sharing Data Controls Between Task Flows."

You may also find it helpful to understand functionality that can be added using other task flow features and parameters. For more information, see Section 20.1.2, "Additional Functionality for Task Flows Using Parameters."

To share a data control between task flows:

1. In the Property Inspector for the called task flow, expand the **Behavior** category.
2. Select the **Share data controls with calling task flow** check box.

> **Note:** After you select the **Share data controls with calling task flow** check box, you may need to configure transaction options for the task flow. For more information, see, Section 20.4.3, "What You May Need to Know About Managing Transactions".

20.4.2 What Happens When You Share a Data Control Between Task Flows

JDeveloper writes an entry in the source file for the called task flow when you select the **Share data controls with calling task flow** check box, as illustrated in Example 20–6.

At runtime, the called task flow shares data controls with the calling task flow.

Example 20–6 Metadata to Share Data Controls Between Task Flows

```
<task-flow-definition id="task-flow-definition">
    ...
    <data-control-scope id="__1">
      <shared/>
    </data-control-scope>
```

```
  ...
</task-flow-definition>
```

20.4.3 What You May Need to Know About Managing Transactions

Data controls cannot be shared across more than one transaction at the same time. If your task flow is involved in managing transactions, the value you select for the `data-control-scope` option may affect the `transaction` option settings for a bounded task flow. Table 20–1 describes how these options interact.

Table 20–1 Interaction of transaction and data-control-scope option settings

transaction Option	data-control-scope Isolated Option	data-control-scope shared Option
`requires-existing-transaction`	Invalid.	If a transaction is not already open, an exception is thrown. The existing transaction is never committed.
`requires-transaction`	Always begins a new transaction.	Begins a new transaction if one is not already open.
`new-transaction`	Always begins a new transaction.	Begins a new transaction if one is not already open. If one is already open, an exception is thrown.
`<default> (none)`	A new data control frame is created without an open transaction.	The task flow uses the calling task flow's transaction for shared data controls and for any new data controls it creates. The task flow does not create or require a transaction.

20.5 Specifying a Return Value from a Bounded Task Flow

You can configure a bounded task flow to return a parameter value to the task flow that calls it. The value that the bounded task flow returns is in addition to the outcome that it returns to the caller when the bounded task flow invokes a task flow return activity, as described in Section 19.7, "Using Task Flow Return Activities." To return a value, you must specify:

- Return value definitions on the called bounded task flow

  The return value definition specifies where you store the value that you want to return when the called bounded task flow exits.

- Return values on the task flow call activity in the calling task flow to identify where the calling task flow can find the returned value

You can configure the calling task flow to ignore return value definition from the called task flow by not identifying any return values on the task flow call activity in the calling task flow.

The task flow call activity returns values by reference. For this reason, you do not need to make a copy of the values that you want to return to the calling task flow.

20.5.1 How to Specify a Return Value from a Bounded Task Flow

You specify a return value definition on the called task flow and add a parameter to the task flow call activity in the calling task flow that retrieves the return value at runtime.

Before you begin:

Create a bounded or unbounded task flow (calling task flow) and a bounded task flow (called task flow). For more information, see Section 18.2, "Creating a Task Flow."

It may be helpful to have an understanding of the interaction between the calling task flow and the called task flow. For more information, see Section 20.5, "Specifying a Return Value from a Bounded Task Flow."

You may also find it helpful to understand functionality that can be added using other task flow features and parameters. For more information, see Section 20.1.2, "Additional Functionality for Task Flows Using Parameters."

To specify a return value from a called bounded task flow:

1. In the Application Navigator, double-click the source file for the called task flow.
2. In the overview editor, click the **Parameters** navigation tab.
3. Click the **Add** icon next to the **Return Value Definitions** category and add values as follows to define a return value:
   - **Name**: Enter a name to identify the return value. For example, `returnValue1`.
   - **Class**: Enter a Java class that defines the data type of the return value. The default value is `java.lang.String`.
   - **Value**: Enter an EL expression that specifies where to store the return value when the called task flow exits. For example, enter an EL expression similar to the following:

     ```
     #{pageFlowScope.ReturnValueDefinition}
     ```
4. In the Application Navigator, double-click the source file for the calling task flow.
5. In the ADF Task Flow page of the Component Palette, from the Component panel, drag and drop a task call activity onto the diagram.
6. In the Property Inspector for the task flow activity, expand the **Parameters** section, click the **Add** icon next to the **Return Values** entry and add values as follows to define a return value:
   - **Name**: Enter a name to identify the return value. For example, `returnValue1`.

     The value you enter must match the value you entered for the **Name** field when you defined the return value definition in step 3.
   - **Value**: Enter an EL expression that specifies where to store the return value when the called task flow exits. For example, enter an EL expression similar to the following:

     ```
     #{pageFlowScope.ReturnValueDefinition}
     ```

     The value you enter must match the value you entered for the **Value** field when you defined the return value definition in step 3.

20.5.2 What Happens When You Specify a Return Value from a Bounded Task Flow

At design time, JDeveloper writes entries to the source files for the task flows that you configured. Example 20–7 shows an example entry that JDeveloper writes to the source file for the calling task flow.

Example 20–7 Metadata in the Calling Task Flow to Specify a Return Value

```
<task-flow-call id="taskFlowCall1">
      <return-value id="__3">
        <name id="__4">returnValue1</name>
         <value id="__2">#{pageFlowScope.ReturnValueDefinition}</value>
      </return-value>
    </task-flow-call>
```

Example 20–8 shows an example entry that JDeveloper writes to the source file for the called task flow.

Example 20–8 Metadata in the Called Task Flow to Specify a Return Value

```
<return-value-definition id="__2">
      <name id="__3">returnValue1</name>
      <value>#{pageFlowScope.ReturnValueDefinition}/</value>
      <class>java.lang.String</class>
 </return-value-definition>
```

At runtime, the called task flow returns a value. If configured to do so, the task flow call activity in the calling task flow retrieves this value.

21

Using Task Flows as Regions

This chapter describes how to render task flows in pages or page fragments using ADF regions.

This chapter includes the following sections:

- Section 21.1, "About Using Task Flows in ADF Regions"
- Section 21.2, "Creating an ADF Region"
- Section 21.3, "Specifying Parameters for an ADF Region"
- Section 21.4, "Specifying Parameters for ADF Regions Using Parameter Maps"
- Section 21.5, "Refreshing an ADF Region"
- Section 21.6, "Configuring Activation of an ADF Region"
- Section 21.7, "Navigating Outside an ADF Region's Task Flow"
- Section 21.8, "Creating ADF Dynamic Regions"
- Section 21.9, "Adding Additional Task Flows to an ADF Dynamic Region"
- Section 21.10, "Configuring a Page To Render an Unknown Number of Regions"

21.1 About Using Task Flows in ADF Regions

You can execute a bounded task flow in a JSF page or page fragment (`.jsff`) by using an ADF region. A primary reason for executing a bounded task flow as an ADF region is reuse. You can isolate specific pieces of application functionality in a bounded task flow and an ADF region in order to reuse it throughout the application. You can extract, configure, and package application functionality within a bounded task flow so that it can be added to other pages using an ADF region. ADF regions can be reused wherever needed, which means they are not dependent on a parent page. This also means that you can isolate the presentation of the parent pages from the ADF region; menus, buttons, and navigation areas are not affected by what is displayed in the ADF region. If you modify a bounded task flow, the changes apply to any ADF region that uses the task flow.

An ADF region comprises the following:

- An `af:region` tag that appears in the page or page fragment where you render the region
- An instance object that implements `RegionModel` from the following package:

  `oracle.adf.view.rich.model`

For more information about `RegionModel`, see the *Oracle Fusion Middleware Java API Reference for Oracle ADF Faces*

- One or other of the following:
  - A task flow binding (`taskFlow`) in the page definition that identifies the bounded task flow to use in the ADF region
  - A multi task flow binding (`multiTaskFlow`) in the page definition that identifies a list of bounded task flows to use in the ADF region

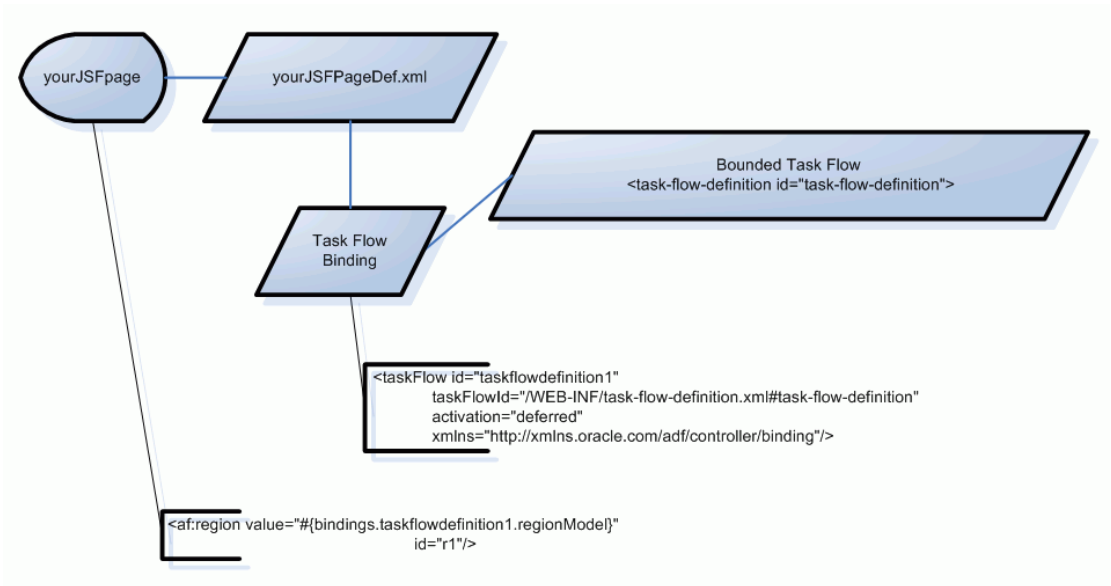
When first rendered, the ADF region's content is that of the first view activity in the bounded task flow. The view activities used in the bounded task flow must be associated with page fragments, not pages.

You can pass values to the ADF region using task flow binding input parameters or contextual events. In addition, you can configure the task flow binding's `parametersMap` property to determine what input parameters the task flow binding passes from the bounded task flow to the ADF region.

ADF regions can be configured so that you determine when the region activates or refreshes. You can also configure an ADF region and a bounded task flow so that navigation control stops in the bounded task flow and passes to the page that contains the ADF region. Finally, you can create dynamic regions (ADF dynamic regions) where the task flow binding determines at runtime what bounded task flow renders in the region and configure a dynamic region link so that end users can change the bounded task flow that renders in the ADF dynamic region at runtime.

Figure 21–1 shows how an ADF region references a bounded task flow using a task flow binding in the page definition file of the page that hosts the ADF region.

Figure 21–1 ADF Region Referencing a Bounded Task Flow



21.1.1 About Page Fragments and ADF Regions

A page fragment is a JSF document (file extension is `.jsff`) that renders as content in another JSF page. A page fragment should not have more than one root component. Wrapping multiple root components in a single root component is recommended so that you optimize the display performance of the page fragment. In addition, if a page

fragment has only one visual root component and a `popup` component (which is invisible to end users until invoked), it is also recommended to wrap these components in a single root component. For example, place the `popup` component in a `panelStretchLayout` component's bottom facet with the `bottomHeight` attribute set to `0` pixels.

If a page fragment has more than one root component, the Fusion web application logs a message at runtime, as shown in Example 21–1, where `r1` identifies the ADF region that renders the page fragment.

Example 21–1 Log Message for a Page Fragment with Multiple Root Components

```
<RegionRenderer> <encodeAll> The region component with id: r1 has detected a page
 fragment with multiple root components. Fragments with more than one root
 component may not display correctly in a region and may have a negative impact
 on performance. It is recommended that you restructure the page fragment to have
 a single root component.
```

Apart from having only one root component element, a page fragment must not contain any of the following tags:

- `<af:document>`
- `<f:view>`
- `<f:form>`
- `<html>`
- `<head>`
- `<body>`

These tags can only appear once in a document and do not support nesting in a JSF page. For example, a page fragment embedded in a page cannot have an `<html>` tag because the JSF page already has one.

Example 21–2 contains an example of a simple page fragment. Unlike a JSF page, it contains no `<f:view>` or `<f:form>` tags.

Example 21–2 Page Fragment Source Code

```
<?xml version='1.0' encoding='UTF-8'?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.1"
          xmlns:af="http://xmlns.oracle.com/adf/faces/rich">
  <af:commandButton text="commandButton 1" id="cb1"/>
</jsp:root>
```

You must nest a page fragment that you include in another JSF page within a region (`af:region` tag). A bounded task flow that you add to a JSF page as a region cannot call pages; it must call page fragments.

21.1.2 About View Ports and ADF Regions

A view port is a display area capable of navigating independently of other view ports. A browser window and an ADF region are both examples of view ports. The root view port displays the main page in a browser window. The root view port may have child view ports, for example, regions on the page, but does not have a parent view port.

Java classes that implement the `ViewPortContext` interface in the `oracle.adf.controller` package get you more information about view ports. For

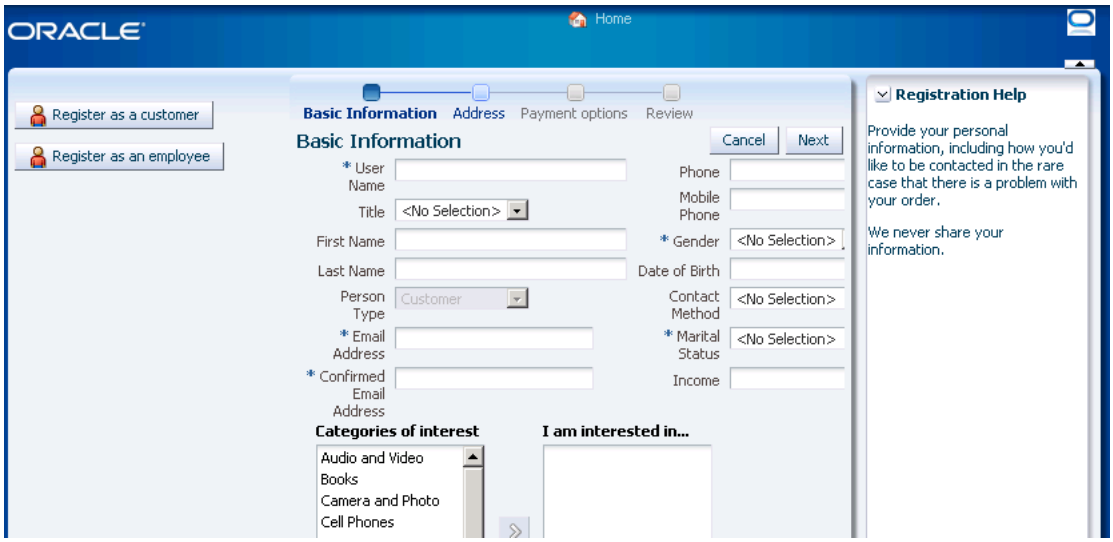
more information, see the *Oracle Fusion Middleware Java API Reference for Oracle ADF Controller*.

21.1.3 Task Flows and ADF Region Use Cases and Examples

Figure 21–2 shows the Registration page (`register.jspx`) in the Fusion Order Demo application which renders two ADF regions. One of these regions (Registration Help) is static; it appears regardless of actions that an end user invokes elsewhere on the Registration page. It displays help information for the end user by rendering the view activities defined in the Fusion Order Demo application's `help-task-flow.xml` task flow. For more information about creating this type of region, see Section 21.2, "Creating an ADF Region."

The second region that `register.jspx` renders is a dynamic region (ADF dynamic region). The task flow that it renders depends on the end user's action. If the end user clicks **Register as a customer** on the registration page, the ADF dynamic region renders the customer registration task flow (`customer-registration-task-flow.xml`). Alternatively, if the end user clicks **Register as an employee**, the ADF dynamic region renders the employee registration task flow (`employee-registration-task-flow.xml`). For more information about creating this type of region, see Section 21.8, "Creating ADF Dynamic Regions."

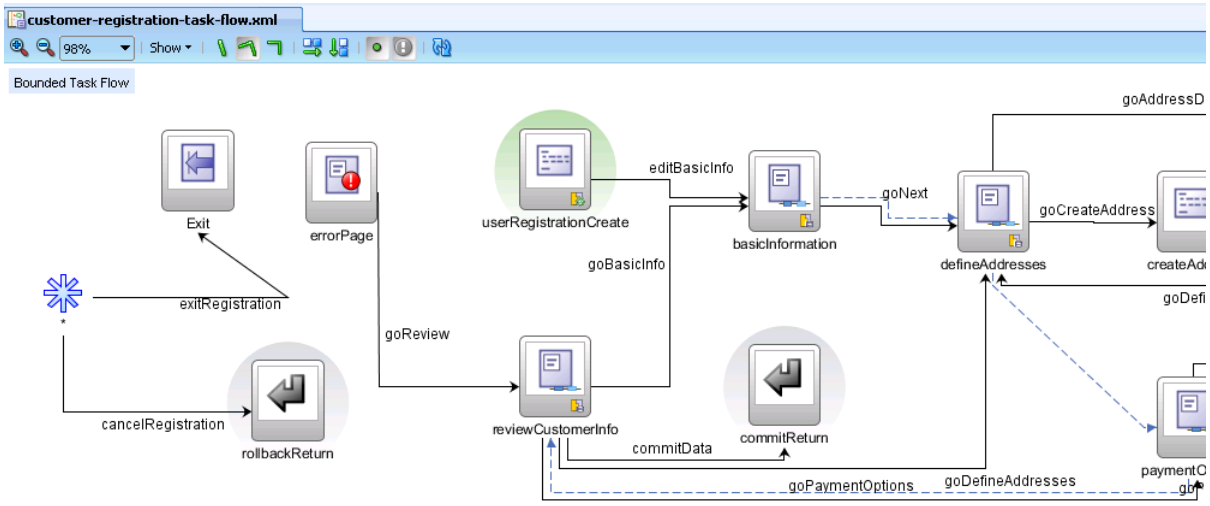
Figure 21–2 ADF Dynamic Region and ADF Region in the Fusion Order Demo Application



The task flows that you render in an ADF region can be simple (for example, the help task flow has one view activity that renders help information) or can involve a number of steps where you guide end users through a process to complete a task, as in the case of the customer registration task flow.

Figure 21–3 shows parts of the customer registration task flow. It contains view activities that allow end users to review information they enter (for example, `reviewCustomerInfo`), an exception handler activity to display a message when an error occurs (`errorPage`), and task flow return activities to cancel or commit the changes that end users make. All these task flow activities can be rendered within an ADF region.

Figure 21–3 Customer Registration Bounded Task Flow in Fusion Order Demo Application



One special case is the task flow return activity. In many cases, you cannot return control when the bounded task flow finishes execution because there is no caller (except the page or page fragment that hosts the bounded task flow) to which you can return control. For this reason, design control flow appropriately in a bounded task flow that you intend to render in an ADF region.

21.1.4 Additional Functionality for Task Flows that Render in ADF Regions

You may find it helpful to understand how a task flow that renders in an ADF region interacts with other task flow and ADF functionality. Following are links to other functionality that may be of interest.

- You can set security on a bounded task flow that displays in an ADF region and associated page definitions. If an end user displays a page that contains an ADF region he or she is not authorized to view, the contents of the ADF region do not display. No authentication mechanism is triggered. For more information, see Chapter 35, "Enabling ADF Security in a Fusion Web Application".
- Task flows can invoke managed beans. For more information about managed beans, see Section 24.4, "Using a Managed Bean in a Fusion Web Application."
- You can use contextual events to exchange information with a bounded task flow. For more information, see Section 34, "Using Contextual Events."

21.2 Creating an ADF Region

You create an ADF region by dragging and dropping a bounded task flow that contains at least one view activity or one task flow call activity to the page where you want to render the ADF region. This makes sure that the ADF region you create has content to display at runtime.

The bound task flow's view activities must be associated with page fragments (`.jsff`). If you attempt to drag and drop a bounded task flow that is associated with pages rather than page fragments, JDeveloper does not display the context menu that allows you to create an ADF region. You can convert a bound task flow that uses pages to use page fragments. For more information, see Section 18.6.3, "How to Convert Bounded Task Flows."

The context menu that JDeveloper displays to create an ADF region presents options to create a non-dynamic and a dynamic region. A dynamic region (ADF dynamic region) determines the bounded task flow that it renders at runtime. For more information about creating an ADF dynamic region, see Section 21.8, "Creating ADF Dynamic Regions." You determine, at design time, the bounded task flow that a non-dynamic region (ADF region) displays.

Before you create an ADF region, you need to do the following:

- Create a bounded task flow with one or more view activities associated with page fragments or one task flow call activity to a task flow with view activities

  For more information, see Section 18.2, "Creating a Task Flow.".

- Create a page to host the ADF region

21.2.1 How to Create an ADF Region

Drag a bounded task flow from the Application Navigator to the page where you want to render an ADF region.

Before you begin:

It may be helpful to have an understanding of the requirements for a bounded task flow that you use in an ADF region. For more information, see Section 21.2, "Creating an ADF Region."

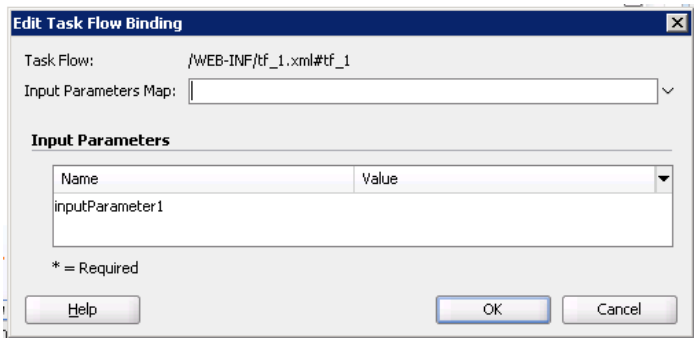
You may also find it helpful to understand functionality that can be added using other task flow features and ADF region features. For more information, see Section 21.1.4, "Additional Functionality for Task Flows that Render in ADF Regions."

To create an ADF region:

1. In the Application Navigator, drag the bounded task flow onto the JSF page and drop it where you want to locate the ADF region.
2. From the context menu that appears, choose **Create** > **Region**.

   The Edit Task Flow Binding dialog appears if the bounded task flow that you drop on the JSF page has an input parameter defined, as described in Section 20.3, "Passing Parameters to a Bounded Task Flow." For more information about specifying parameters for an ADF region, see Section 21.3, "Specifying Parameters for an ADF Region."

Figure 21–4 Edit Task Flow Binding Dialog for an ADF Region



3. In the Structure window, right-click the node for the ADF region that you added (**af:region**) and choose **Go to Properties**.

4. Review or modify (as appropriate) the following properties which JDeveloper automatically populates with default values in the Property Inspector for the ADF region:
   - **Id**: An ID that the JSF page uses to reference the ADF region, for example, `r1`.
   - **Value**: An EL reference to the ADF region model, for example, `#{bindings.region1.regionModel}`. This is the region model that describes the behavior of the region.
   - **Rendered**: If `true` (the default value), the ADF region renders when the JSF page renders.
5. For information about how to map parameters between the view activity associated with the JSF page and the ADF region, see Section 21.3, "Specifying Parameters for an ADF Region".

21.2.2 What Happens When You Create an ADF Region

When you drop a bounded task flow onto a JSF page to create an ADF region, JDeveloper adds an `af:region` tag to the page. The `af:region` tag references an object that implements `RegionModel`. Example 21–3 shows a sample of the metadata that JDeveloper adds to the JSF page.

Example 21–3 Metadata Added to a JSF Page to Create an ADF Region

```
<af:region value="#{bindings.tf_register_employee1.regionModel}"
                   id="r1"/>
```

JDeveloper also adds a task flow binding to the page definition file of the page that hosts the ADF region. Example 21–4 shows a sample of the metadata that JDeveloper adds. The task flow binding provides a bridge between the ADF region and the bounded task flow. It binds a specific instance of an ADF region to its associated bounded task flow and maintains all information specific to the bounded task flow. The `taskFlowId` attribute specifies the directory path and the name of the source file for the bounded task flow.

Example 21–4 Metadata Added to Page Definition to Create a Task Flow Binding

```
<taskFlow id="tf_register_employee1"
              taskFlowId="/WEB-INF/tf_register_employee.xml#tf_register_employee"
              activation="deferred"
              xmlns="http://xmlns.oracle.com/adf/controller/binding"/>
```

The bounded task flow preserves all of the data bindings when you drop it on the JSF page. At runtime, a request for a JSF page containing an ADF region initially handles like any other request for a JSF page. As the JSF page definition executes, data loads in to the JSF page. When the component tree for the parent JSF page encounters the `<af:region>` tag, it executes it in order to determine the first page fragment of content to display. Once it determine the first page fragment of content, it adds the appropriate UI components from the page fragment to the component tree for the parent JSF page.

The task flow binding creates an object for its task flow that implements the following interface in order to get the current view activity:

`oracle.adf.controller.ViewPortContext`

The task flow binding's `taskFlowId` attribute can also reference an EL expression that evaluates to one of the following:

- `java.lang.String`
- `oracle.adf.controller.TaskFlowId`

You use this capability if you create an ADF dynamic region. For more information, see Section 21.8, "Creating ADF Dynamic Regions."

21.3 Specifying Parameters for an ADF Region

You can make input parameters that you defined for a bounded task flow available to an ADF region by adding them to the task flow binding that the ADF region references. Use EL expressions to reference input parameters available in memory scopes, managed beans, or the ADF binding layer.

Specifying input parameters is one method of providing information to an ADF region. An alternative method is to use contextual events. The nature of the information that you want to provide to the ADF region determines the method you choose to provide the information. For example, choose:

- Input parameters if the required information is at the beginning of the task flow and a change in the value of this information requires a restart of the task flow.

  For example, you have a page that contains a table of employees. An ADF region on that page contains a task flow to enroll a selected employee in a benefits program. A change in the selected employee requires that you restart the benefits enrollment task flow from the beginning for the newly selected employee. Using input parameters to the task flow is the right decision for this use case.

  You can pass input parameters by reference or by value. If you pass by reference, an update on the main page for the selected employee's information, such as last name, is automatically reflected in the task flow running in the ADF region without restarting the task flow.

- Contextual events if you can only determine the information to exchange after the start of a task flow and a change in the information does not require a restart of the task flow. For example, the Fusion Order Demo application uses contextual events to display the appropriate help topic when the customer registration task flow renders in the `register.jspx` page. For more information about contextual events, see Chapter 34, "Using Contextual Events."

For information about creating an ADF region and adding task flow bindings, see Section 21.2, "Creating an ADF Region." For information about how to define an input parameter for a bounded task flow, see Section 20.3, "Passing Parameters to a Bounded Task Flow".

21.3.1 How to Specify Parameters for an ADF Region

Use EL expressions to specify parameters available in memory scopes, managed beans, or the ADF binding layer as input for the ADF region.

Before you begin:

It may be helpful to have an understanding of the requirements for specifying parameters for an ADF region. For more information, see Section 21.3, "Specifying Parameters for an ADF Region."

You may also find it helpful to understand functionality that can be added using other task flow features and ADF region features. For more information, see Section 21.1.4, "Additional Functionality for Task Flows that Render in ADF Regions."

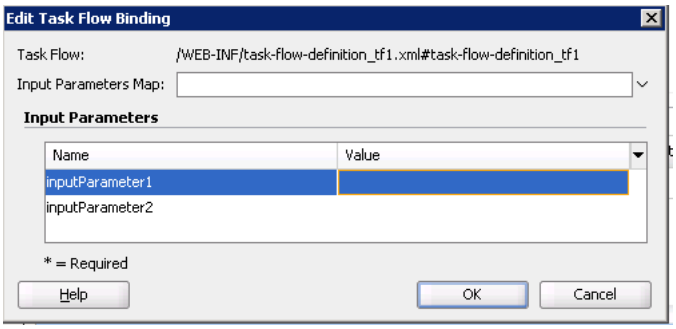
To specify input parameters for an ADF region:

1. In the Application Navigator, right-click the JSF page that holds the ADF region and choose **Go to Page Definition**.
2. In the overview editor for the page definition file, select the task flow binding for which you want to specify parameters.
3. Click the Edit icon to display the Edit Task Flow Binding dialog, as shown in Figure 21–5.

   If you defined input parameters for the bounded task flow, as described in Section 20.3, "Passing Parameters to a Bounded Task Flow", the Edit Task Flow Binding dialog lists these parameters in the Input Parameters section.

   Note: You can write an EL expression that references a list of input parameters specified in a managed bean using the Input Parameters Map field of the Edit Task Flow Binding dialog. For more information about implementing this functionality, see Section 21.4, "Specifying Parameters for ADF Regions Using Parameter Maps."

Figure 21–5 Edit Task Flow Binding Dialog



4. Write an EL expression that retrieves the value of each input parameter you want to specify for the ADF region. Note that you must write an EL expression for parameters that you defined as required. For example, write an EL expression similar to the following:

   ```
   #{pageFlowScope.inputParameter1}
   ```

5. Click **OK**.

21.3.2 What Happens When You Specify Parameters for an ADF Region

JDeveloper writes entries that are child elements of the `taskFlow` element in the page definition of the JSF page, as illustrated in example Example 21–5.

Example 21–5 Metadata Entries to Specify Input Parameters for an ADF Region

```
<taskFlow id="tflow_tf11"
   taskFlowId="/WEB-INF/tflow_tf1.xml#tflow_tf1"
   activation="deferred"
   xmlns="http://xmlns.oracle.com/adf/controller/binding">
  <parameters>
    <parameter id="inputParameter1"
      value="#{pageFlowScope.inputParameter1}"/>
    <parameter id="inputParameter2"
      value="#{pageFlowScope.inputParameter2}"/>
```

```
  </parameters>
</taskFlow>
```

At runtime, the values specified by the EL expression in the value `attribute` are passed to the ADF region.

21.4 Specifying Parameters for ADF Regions Using Parameter Maps

In addition (or as an alternative) to listing all the input parameters on the task flow binding, as described in Section 21.3, "Specifying Parameters for an ADF Region," you can use the `parametersMap` property of the task flow binding to specify a parameter map object on a managed bean. The parameter map object that you reference must be of a type that implements the following interface:

`java.util.Map`

The parameter map object that you reference specifies keys that map to the values you want to input to the ADF region. Using this approach reduces the number of `parameter` child elements that appear under the task flow binding (`taskFlow`) element or the multi task flow binding (`multiTaskFlow`) in the page definition for the page. This approach also allows you more flexibility in determining what input parameters pass to the ADF region. In particular, it provides a way to pass parameters to ADF dynamic regions as the number of input parameters can differ between the different task flow bindings in an ADF dynamic region. For more information, see Section 21.8, "Creating ADF Dynamic Regions."

You can configure an ADF region or an ADF dynamic region's task flow binding to reference a parameter map. You can also configure a multi task flow binding to reference a parameter map. Make sure that the name of an input parameter you define for a bounded task flow matches the name of a key that you define in the parameter map object.

If you specify a parameter with the same `id` attribute in a `<parameter>` element and in a `<parametersMap>` element, the `<parametersMap>` element always takes precedence. This is because the `<parametersMap>` element may need to override the static value of the parameter if you create an ADF dynamic region, as described in Section 21.8, "Creating ADF Dynamic Regions."

21.4.1 How to Create a Parameter Map to Specify Input Parameters for an ADF Region

You configure the task flow binding's `parametersMap` property to reference the parameter map object that defines the key-value pairs you want to pass to the ADF region.

Before you begin:

It may be helpful to have an understanding of the configuration options available to you when passing input parameters to an ADF region. For more information, see Section 21.4, "Specifying Parameters for ADF Regions Using Parameter Maps."

You may also find it helpful to understand functionality that can be added using other task flow and ADF region features. For more information, Section 21.1.4, "Additional Functionality for Task Flows that Render in ADF Regions."

To create a parameter map to specify input parameters for an ADF region:

1. Create a managed bean or edit an existing managed bean so that it returns an object that implements the `java.util.Map` interface.

Configure the managed bean so the object returns key-value pairs with the values that you want to pass to the ADF region. For more information about managed beans, see Section 24.4, "Using a Managed Bean in a Fusion Web Application."

2. In the Application Navigator, right-click the JSF page that holds the ADF region and choose **Go to Page Definition**.
3. In the overview editor for the page definition file, select the task flow binding or multi task flow binding for which you want to specify a parameter map.
4. Click the Edit icon to display the appropriate dialog (Edit Task Flow Binding dialog or the Edit Multi Task Flow Binding).
5. Select **Expression Builder** from the Input Parameters Map dropdown menu.
6. Write or build an EL expression that references a parameter map. For example, write an EL expression similar to the following:

   `#{pageFlowScope.userInfoBean.parameterMap}`
7. Click **OK**.

21.4.2 What Happens When You Create a Parameter Map to Specify Input Parameters

At runtime, the task flow binding or multi task flow binding evaluates the EL expression specified for its `parametersMap` property. It returns values to the ADF region from the managed bean for keys that match the name of the input parameters defined for the bounded task flow that render in the ADF region.

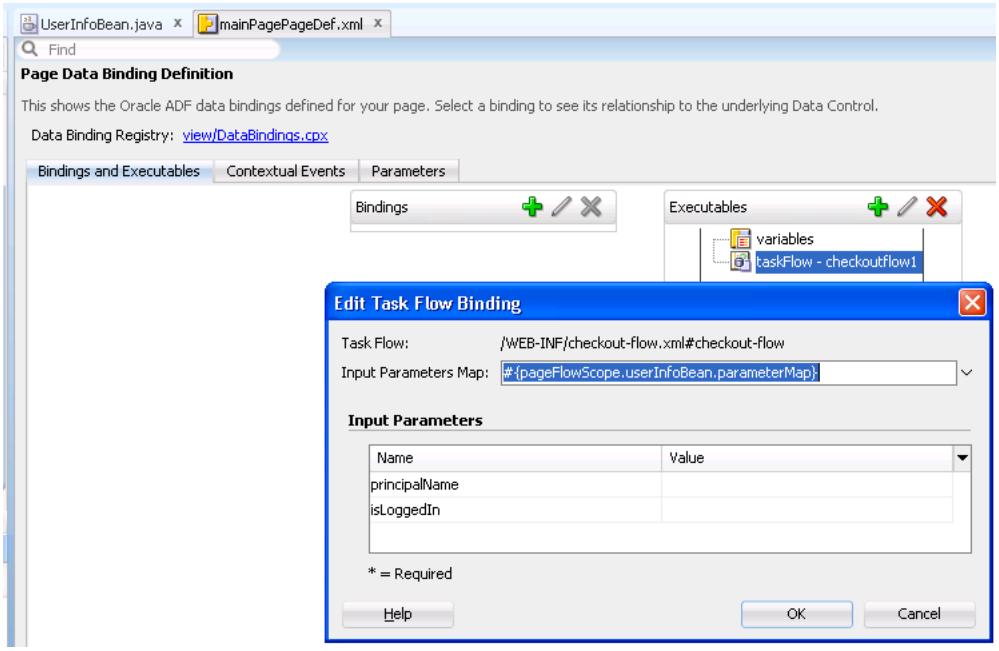
Example 21–6 shows code snippets from a managed bean that puts two values (`isLoggedIn` and `principalName`) into a parameter map named `parameterMap`.

Example 21–6 Managed Bean Defining a Parameter Map

```
import java.util.HashMap;
import java.util.Map;

public class UserInfoBean {
    private Map<String, Object> parameterMap = new HashMap<String, Object>();

    public Map getParameterMap() {

        parameterMap.put("isLoggedIn", getSecurity().isAuthenticated());
        parameterMap.put("principalName", getSecurity().getPrincipalName());
        return parameterMap;
    }
}
```

Figure 21–6 shows the Edit Task Flow Binding dialog after you close the Expression Builder dialog, having specified the parameter map object (`parameterMap`) shown in Example 21–6. The **Input Parameters** field in the Edit Task Flow Binding dialog lists the input parameters defined for the bounded task flow associated with this ADF region (`checkout-flow`). The task flow binding retrieves the values for these parameters from the managed bean shown in Example 21–6.

Figure 21–6 EL Expression Referencing Parameter Map on Task Flow Binding



Example 21–7 shows the metadata that appears for the task flow binding in the page definition file of the page that renders the ADF region. The metadata for the task flow binding references both the bounded task flow (`taskFlowId` attribute) and the managed bean (`parametersMap`).

Example 21–7 Task Flow Binding Specifying a Parameter Map

```
<taskFlow id="checkoutflow1"
              taskFlowId="/WEB-INF/checkout-flow.xml#checkout-flow"
              activation="deferred"
              xmlns="http://xmlns.oracle.com/adf/controller/binding"
              parametersMap="#{pageFlowScope.userInfoBean.parameterMap}"/>
```

You specify the `<parameterMap>` element in the page definition.

Note: If you specify `Refresh="ifNeeded"`, parameters are not supported in the `<parameterMap>` element. The only condition that determines whether the region need to be refreshed is the boolean value returned by the evaluation of `RefreshCondition`. For more information, see Section 21.5.2, "What You May Need to Know About Refreshing an ADF Region".

21.5 Refreshing an ADF Region

You can configure when an ADF region refreshes and whether it invokes a task flow. An ADF region can only invoke a task flow when the ADF region is in an active state. An ADF region in an inactive state cannot invoke a task flow and returns a null value for the ID of the referenced task flow to the parent page.

21.5.1 How to Configure the Refresh of an ADF Region

You set values for the task flow binding of the task flow associated with the ADF region to determine when an ADF region switches from an inactive to an active state and to determine when an ADF region refreshes.

Before you begin:

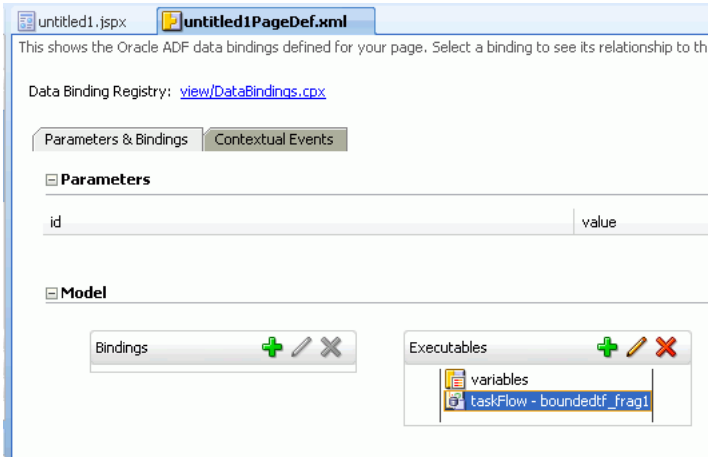
It may be helpful to have an understanding of the requirements for a bounded task flow that you use in an ADF region. For more information, see Section 21.5, "Refreshing an ADF Region."

You may also find it helpful to understand functionality that can be added using other task flow features and ADF region features. For more information, see Section 21.1.4, "Additional Functionality for Task Flows that Render in ADF Regions."

To configure the refreshing of an ADF Region:

1. In the Application Navigator, select the page that contains the ADF region, right-click and choose **Go to Page Definition**.
2. In the page definition file, select the task flow binding or multi task flowing binding in the **Executables** section, as illustrated in Figure 21–7.

Figure 21–7 Task Flow Binding



3. In the Property Inspector, select a value from the Refresh dropdown menu, as described in the following list:
   - **default**: refresh the ADF region once when the page that hosts the ADF region appears or when the EL expression you set as a value for the RefreshCondition property in step 4 returns `true`.
   - **ifNeeded**: refresh the ADF region if the value of a task flow binding parameter changes. Do not set a value for the RefreshCondition property in step 4 if you select this value.
4. If you selected **default** as the value for the Refresh property, select **Edit** from the RefreshCondition dropdown menu to invoke the Expression Builder and write an EL expression that returns a boolean value at runtime. If the EL expression returns `true`, the ADF region refreshes.
5. In the Property Inspector, select a value from the activation dropdown menu, as described in the following list:

- **conditional**: activates the ADF region if the EL expression set as a value for the task flow binding `active` property returns `true`.
- **deferred**: select this option if your application uses Facelets XHTML pages in the view layer and you want to activate the ADF region when a Facelets XHTML page first requests a `viewID`. If your application uses JSP technology in the view layer, selecting this option has the same effect as selecting **immediate** (the ADF region activates immediately) provided that the parent component for the ADF region is not a `popup` or `panelTabbed` component that has its `childCreation` attribute set to `deferred`. If this latter scenario occurs, the parent component (`popup` or `panelTabbed`) determines behavior.

  This option is recommended if your application uses Facelets XHTML pages. For more information about Facelets with ADF Faces, see the "Getting Started with ADF Faces and JDeveloper" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
- **immediate**: activates the ADF region immediately. This is the default value.

The value that you select in the dropdown menu determines when an ADF region switches from an inactive to an active state. An ADF region must have an active state before it can invoke a task flow.

6. If you selected **conditional** as the value for the activation property, select **Edit** from the active dropdown menu to invoke the Expression Builder and write an EL expression that returns a boolean value at runtime. If the EL expression returns `true`, the ADF region invokes the task flow.

21.5.2 What You May Need to Know About Refreshing an ADF Region

An ADF region initially refreshes when the parent JSF page on which the region is located first displays. During the initial refresh, any ADF region task flow binding parameter values are passed in from the parent page. The parameter values are used to display the initial page fragment within the ADF region. If the bounded task flow relies on input parameter values that are not available when the page is first created, make sure that your task flow behaves correctly if the input parameter values are null by, for example, using a `NullPointerException` object. Alternatively (or additionally), make sure that the task flow does not activate until the input parameters become available by configuring the task flow binding's `active` property.

An ADF region task flow binding can be refreshed again based on one of the following task flow binding attributes:

- Neither `Refresh` or `RefreshCondition` attributes are specified (default)

  If neither the `Refresh` nor `RefreshCondition` task flow binding attribute is specified, the ADF region is refreshed only once at the time the parent page is first displayed unless you configure a value for the task flow binding's `active` property.
- `RefreshCondition="#{EL.expression}"`

  The ADF region is refreshed a single time when its `RefreshCondition` evaluates true. The `RefreshCondition` must evaluate to a boolean value.

  At the time the `RefreshCondition` is evaluated, if the variable `bindings` is used within the EL Expression, the context refers to the binding container of the parent page, not the page fragment displayed within the ADF region.

`RefreshCondition` is independent of the change of value of binding parameters. If the task flow binding parameters do not change, nothing within the ADF region will change.

- `Refresh="ifNeeded"`

  Any change to a task flow binding parameter value causes the ADF region to refresh.

  If the ADF region task flow binding does not have parameters, `Refresh="ifNeeded"` is equivalent to not specifying the `Refresh` attribute.

  If you set `Refresh` to `ifNeeded`, the `RefreshCondition` attribute should not be specified.

  `Refresh="ifNeeded"` is not supported if you pass parameters to the task flow binding using a dynamic parameter map. You must instead use the `RefreshCondition="#{EL.Expression}"`.

  For more information about specifying parameter values using a parameter map, see Section 21.4, "Specifying Parameters for ADF Regions Using Parameter Maps."

The `RefreshCondition` and `Refresh` properties are mutually exclusive. The `Refresh="ifNeeded"` property takes precedence over `RefreshCondition`. If the bindings variable is used within the EL expression at the time `RefreshCondition` is evaluated, the context refers to the binding container of the parent page, not the page fragment displayed within the ADF region. The expression is evaluated during the `PrepareRender` phase of the ADF page lifecycle. For more information, see Chapter 25, "Understanding the Fusion Page Lifecycle".

Example 21–8 contains a sample task flow binding located within the page definition of a page on which an ADF region has been added.

Example 21–8 Refresh Option Specified in ADF Region Binding

```
<taskFlow id="Department1" taskFlowId="/WEB-INF/Department#Department"
    Refresh="ifNeeded"
    xmlns="http://xmlns.oracle.com/adf/controller/binding">
  <parameters>
    <parameter id="DepartmentId" value="#{bindings.DepartmentId.inputValue}"
        xmlns="http://xmlns.oracle.com/adfm/uimodel"/>
  </parameters>
</taskFlow>
```

You do not need to refresh an ADF region to refresh the data controls inside the ADF region. During the ADF lifecycle, the refresh events telling the iterators to update will be propagated to the binding container of the current page of the ADF region.

21.6 Configuring Activation of an ADF Region

You can configure when an ADF region activates. This has the effect of determining when the task flows contained in ADF regions activate. Configuring the activation of ADF regions can optimize the performance of a Fusion web application's page. For example, a page that contains 5 ADF regions may reference 5 task flows. You may not want these 5 task flows to execute simultaneously when the Fusion web application page loads so you configure an `activation` property to determine when a task flow executes.

21.6.1 How to Configure Activation of an ADF Region

You configure the `activation` property for the task flow binding associated with the ADF region to determine when to activate the ADF region.

Before you begin:

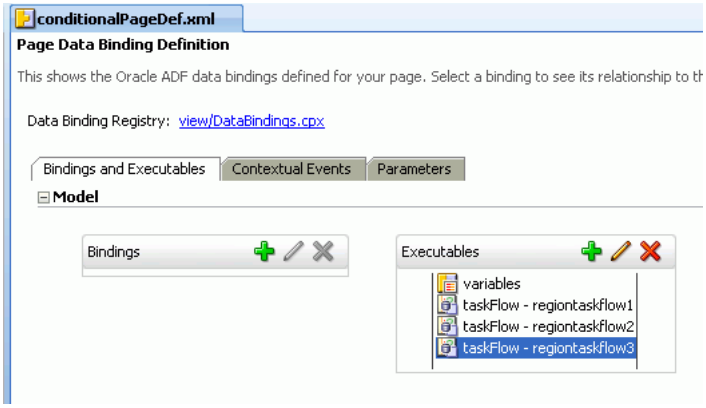
It may be helpful to have an understanding of the configuration options for activating an ADF region. For more information, see Section 21.6, "Configuring Activation of an ADF Region."

You may also find it helpful to understand functionality that can be added using other task flow features and ADF region features. For more information, see Section 21.1.4, "Additional Functionality for Task Flows that Render in ADF Regions."

To configure the activation of an ADF region:

1. In the Application Navigator, select the page that contains the ADF region(s), right-click and choose **Go to Page Definition**.
2. In the page definition file, select the task flow binding or multi task flow binding in the **Executables** section, as illustrated in Figure 21–8.

Figure 21–8 Task Flow Bindings for ADF Region Configured for Activation



3. In the Property Inspector, select a value from the activation dropdown menu, as described in the following list:
   - **conditional**: activates the ADF region if the EL expression set as a value for the task flow binding `active` property returns `true`.
   - **deferred**: select this option if your application uses Facelets XHTML pages in the view layer and you want to activate the ADF region when a Facelets XHTML page first requests a `viewID`. If your application uses JSP technology in the view layer, selecting this option has the same effect as selecting **immediate** (the ADF region activates immediately).

     This option is recommended if your application uses Facelets XHTML pages. For more information about Facelets with ADF Faces, see the "Getting Started with ADF Faces and JDeveloper" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
   - **immediate**: activates the ADF region immediately. This is the default value.

   The value that you select in the dropdown menu determines when an ADF region switches from an inactive to an active state. An ADF region must have an active state before it can invoke a task flow.

4. If you selected **conditional** as the value for the `activation` property, select **Edit** from the active dropdown menu to invoke the Expression Builder and write an EL expression that returns a boolean value at runtime. If the EL expression returns `true`, the ADF region invokes the task flow. If the EL expression returns `false`, the ADF region deactivates a task flow that was active in the ADF region.

21.6.2 What Happens When You Configure Activation of an ADF Region

The behavior of the ADF region at runtime depends on the options that you set for the `activation` property.

Figure 21–9 shows the default behavior where all task flows in ADF regions execute when the page loads the ADF regions (Tab # 1, Tab # 2, Tab # 3).

Figure 21–9 Default Activation of ADF Regions

Figure 21–10 shows an example where the activation property was set to `deferred` and the application uses Facelets. The regions in Tab # 1 and Tab # 2 have an active state because the end user navigated to these regions. The region in Tab # 3 is inactive.

Figure 21–10 Deferred Activation of ADF Regions

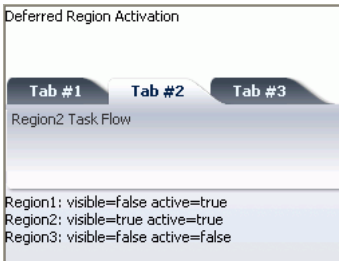


Figure 21–11 shows an example where the `activation` property was set to `conditional` and the `active` property to an EL expression that returns a `boolean` value at runtime. The region in Tab # 1 is in an active state because the EL expression specified for the `active` property returns `true`. The regions in Tab # 2 and Tab # 3 are inactive because the EL expression specified for the `active` property returns `false`.

Note that the following events occur if a region deactivates a task flow (the value returned by the `active` property changes from `true` to `false`):

- A task flow with an active transaction rolls back the transaction.

  For more information about transaction options in task flows, see Section 22.2, "Managing Transactions in Task Flows."

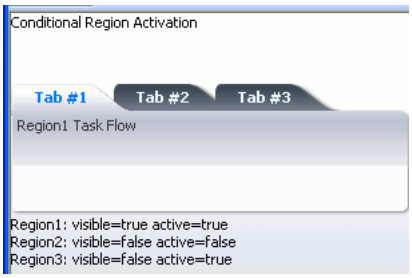
- If a task flow has a data control frame with a `data-control-scope` value of `isolated`, the task flow releases the data control frame and any data controls in the data control frame.

For more information about data controls, see Section 20.4, "Sharing Data Controls Between Task Flows" and Section 22.2.3, "What You May Need to Know About Data Control Scoping with Task Flows."

- ADF Controller releases the view port data structures for the region (including pageFlow and view scopes).

  For more information about view ports, see Section 21.1.2, "About View Ports and ADF Regions." For more information about memory scopes, see Section 18.2.4, "What You May Need to Know About Memory Scope for Task Flows."

Figure 21–11 Conditional Activation of ADF Regions



21.7 Navigating Outside an ADF Region's Task Flow

A bounded task flow running in an ADF region may need to trigger navigation of its parent view activity or to navigate to the root page of its application. The parent action activity exposes properties (`parent-outcome` and `root-outcome`) that you configure if you want to implement either of these use cases.

For example, you might specify a value for `parent-outcome` if you have a page that displays employee information and an ADF region that contains an enroll button for an employee. After the enroll page completes and the ADF region returns, the employee information page refreshes with the next employee.

Note: An ADF region does not maintain state when you navigate away from the region.

21.7.1 How to Trigger Navigation Outside of an ADF Region's Task Flow

You add a parent action activity to the bounded task that runs in the ADF region and configure it so that it navigates to the parent's view activity.

Before you begin:

It may be helpful to have an understanding of the configuration options for activating an ADF region. For more information, see Section 21.6, "Configuring Activation of an ADF Region."

You may also find it helpful to understand functionality that can be added using other task flow features and ADF region features. For more information, see Section 21.1.4, "Additional Functionality for Task Flows that Render in ADF Regions."

To trigger navigation outside of an ADF region's task flow:

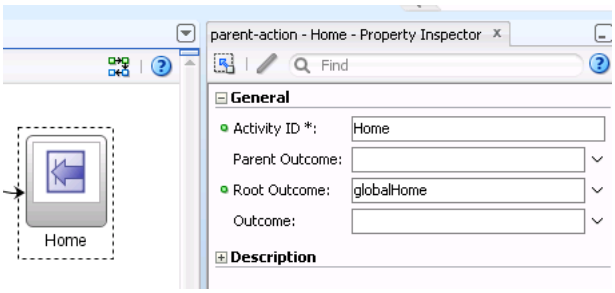
1. In the Application Navigator, double-click the XML file for the bounded task flow that runs in the ADF region.

2. In the ADF Task Flow page of the Component Palette, from the Component panel, drag and drop a **Parent Action** onto the diagram.
3. Select the parent action activity in the diagram for the bounded task flow.
4. In the Property Inspector, choose the appropriate option:
   - **Parent Outcome**: enter a literal value or write an EL expression that returns an outcome so that the parent action activity navigates to the parent view activity. The outcome is then used to navigate the parent view activity's task flow rather than navigating the ADF region's bounded task flow.

     The parent view activity's task flow should contain a control flow case or wildcard control flow rule that accepts the value you specify for `parent-outcome`.
   - **Root Outcome**: enter a literal value or write an EL expression that returns an outcome so that the parent action activity navigates to the root page of the application.

     For example, to navigate from an ADF region to the home page in the Fusion Order Demo application, you specify `globalHome` as the value for **Root Outcome**, as illustrated in Figure 21–12.

Figure 21–12 Navigating to Home Page



Note: **Parent Outcome** and **Root Outcome** are mutually exclusive.

5. (Optional) In the **Outcome** field, enter a literal value that specifies an outcome for control flow within the ADF region after the parent action activity adds `parent-outcome` or `root-outcome` to the appropriate queue.

   Specifying an `outcome` element is useful in cases where the parent view activity or root page does not navigate as a result of the `parent-outcome` or `root-outcome` sent by the parent action activity. In addition, the ADF region should not continue to display the same view. If you do not specify a value for `outcome`, the ADF region's `viewId` remains unchanged.

21.7.2 What Happens When You Configure Navigation Outside a Task Flow

At design time, JDeveloper writes entries to the source file for the bounded task flow based on the property values you set for the parent action activity. Example 21–9 shows sample entries that JDeveloper generates when you write a literal value for the Parent Outcome and Outcome properties of the parent action activity.

Example 21–9 Metadata for a Parent Action Activity with a Parent Outcome

```
<parent-action id="parentAction1">
```

```
    <parent-outcome>parent_outcome</parent-outcome>
    <outcome id="__2">outcome</outcome>
</parent-action>
```

Example 21–10 shows sample entries that JDeveloper generates when you write a literal value for the Root Outcome and Outcome properties of the parent action activity.

Example 21–10 Metadata for a Parent Action Activity with a Root Outcome

```
<parent-action id="parentAction1">
      <root-outcome>root_outcome</root-outcome>
      <outcome id="__2">outcome</outcome>
</parent-action>
```

At runtime, the bounded task flow navigates to its parent view activity or the root page of the Fusion web application depending on the property that you configured for the parent action activity.

21.7.3 What You May Need to Know About How a Page Determines the Capabilities of an ADF Region

In some cases, a page may need to determine the capabilities currently available within one of the ADF regions that it contains. For example, the page may need to initiate control flow within the ADF region based on its current state using the `queueActionEventInRegion()` method. *Region capabilities* are used by a parent page to identify the current outcomes of one of its regions based on the region's current state. They identify to the parent page whether or not an outcome is possible within the region.

The following scenario describes how region capabilities might typically be used in an application:

1. An ADF region within a page displays its associated page fragment.
2. A user selects a button or performs some other action in the ADF region.
3. The parent page identifies the outcomes (region capabilities) for the ADF region.
4. The parent page updates its buttons based on the ADF region capabilities.

The syntax of an EL expression to determine an ADF region's capabilities is as follows:

```
#{bindings.[regionId].regionModel.capabilities['outcome']}
```

where `regionId` is the ID of the ADF region component on the page and `outcome` is the possible outcome being verified within the ADF region.

Region capabilities require the availability of the specified ADF region's `regionModel` through an EL expression. The EL expression should never access bindings in any other binding container other than the current binding container. Region capabilities cannot be used in some cases. For example, a nested ADF region where a `regionModel` cannot be reached within the current binding container. This is because a nested region's nested binding container might not yet exist or might have already been released.

21.8 Creating ADF Dynamic Regions

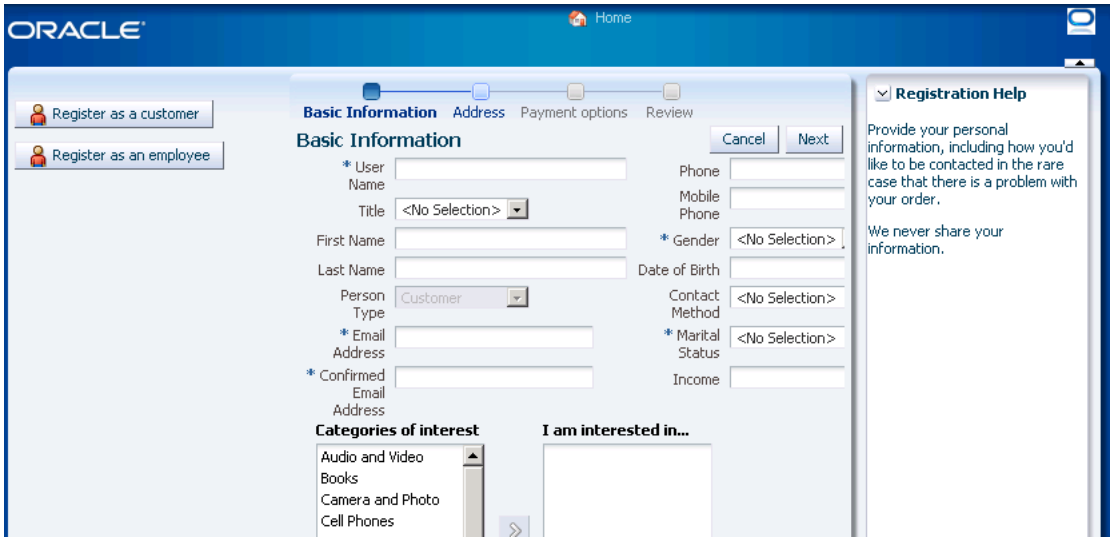
An ADF dynamic region is an ADF region where the task flow binding dynamically determines the value of its `taskFlowId` attribute at runtime. This allows the Fusion

web application to determine which bounded task flow to execute within the ADF dynamic region based on the result of evaluation of the task flow binding's `taskFlowId` attribute.

Figure 21–13 shows a runtime example from the Fusion Order Demo application where the `register.jspx` page renders a different registration task flow in its center facet in response to the command component button that the end user clicks. For example, if the end user clicks **Register as an employee**, the Fusion Order Demo application renders the `employee-registration-task-flow` in the ADF dynamic region.

At runtime, the ADF dynamic region swaps the task flows that it renders during the Prepare Render lifecycle phase. To give components from a previous task flow the opportunity to participate smoothly in the lifecycle phases, do not navigate off the `regionModel` until the JSF Invoke Application lifecycle phase. It is good practice, therefore, not to navigate a task flow in a dynamic region in response to contextual events. For more information about lifecycle phases, see Chapter 25, "Understanding the Fusion Page Lifecycle," and for more information about contextual events, see Chapter 34, "Using Contextual Events."

Figure 21–13 Customer Registration Task Flow in an ADF Dynamic Region



You create an ADF dynamic region by dragging and dropping a bounded task flow to the page where you want to render the ADF dynamic region. The view activities in the bounded task flow must be associated with page fragments (`.jsff`).

If you attempt to drag and drop a bounded task flow that is associated with pages rather than page fragments, JDeveloper does not display the context menu that allows you to create an ADF dynamic region. You can convert a bound task flow that uses pages to use page fragments. For more information, see Section 18.6.3, "How to Convert Bounded Task Flows."

After you create an ADF dynamic region, you can add additional bounded task flows to the dynamic region by creating ADF dynamic region links, as described in Section 21.9, "Adding Additional Task Flows to an ADF Dynamic Region."

21.8.1 How to Create an ADF Dynamic Region

Drag and drop bounded task flows to the page where you want to render the ADF dynamic region.

Before you begin:

It may be helpful to have an understanding of the configuration options available to you when creating an ADF dynamic region. For more information, see Section 21.8, "Creating ADF Dynamic Regions."

You may also find it helpful to understand other functionality that can be added using other task flow and ADF region features. For more information, see Section 21.1.4, "Additional Functionality for Task Flows that Render in ADF Regions."

To create an ADF dynamic region:

1. In the Application Navigator, select and open the JSF page where you want to create the ADF dynamic region.
2. Drag and drop the first bounded task flow onto the JSF page.
3. From the context menu that JDeveloper displays, choose **Create** > **Dynamic Region**.
4. Choose the appropriate option in the Choose Managed Bean for Dynamic Region dialog that JDeveloper displays:
   - If you want an existing managed bean to store the bounded task flow's ID, select an existing managed bean from the **Managed Bean** dropdown list.

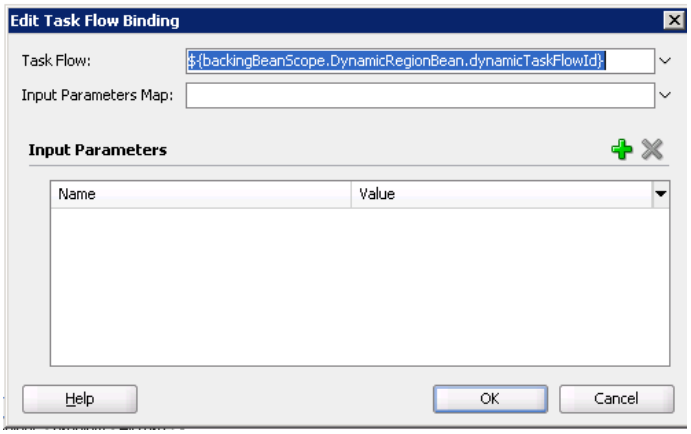
     The managed bean passes the value of the bounded task flow's ID into the task flow binding of the ADF dynamic region. Make sure that the managed bean you select is not used by another ADF dynamic region.
   - If no managed bean exists for the page, click the **Add** icon next to the **Managed Bean** dropdown list to create a new one. Enter values in the dialog that appears as follows:
     a. **Bean Name**: Enter a name for the new managed bean. For example, enter `DynamicRegionBean`.
     b. **Class Name**: Enter the name of the new managed bean class.
     c. **Package**: Enter the name of the package that is to contain the managed bean or browse to locate it.
     d. **Extends**: Enter the name of the Java class that the managed bean extends. The default value is `java.lang.Object`.
     e. **Scope**: This field is read-only and its value is set to `backingBean`. For more information about the memory scope for managed beans, see Section 18.2.4, "What You May Need to Know About Memory Scope for Task Flows."
     f. Click **OK** to close the dialogs where you configure the managed bean.
5. Choose the appropriate option in the Edit Task Flow Binding dialog that JDeveloper displays:
   - Click **OK** if you do not want specify input parameters or an input parameter map for the ADF dynamic region.
   - Specify input parameters for the ADF dynamic region.

     For more information, see Section 21.3, "Specifying Parameters for an ADF Region."

   Figure 21–14 shows the Edit Task Flow Binding dialog that JDeveloper displays after you configure a managed bean.

Note: You can write an EL expression that references a list of input parameters specified in a managed bean using the Input Parameters Map field of the Edit Task Flow Binding dialog. For more information about implementing this functionality, see Section 21.4, "Specifying Parameters for ADF Regions Using Parameter Maps."

Figure 21–14 Edit Task Flow Binding Dialog for an ADF Dynamic Region



6. Click **OK**.

21.8.2 What Happens When You Create an ADF Dynamic Region

When you drop a bounded task flow onto a JSF page to create an ADF dynamic region, JDeveloper adds an `af:region` tag to the page. The `af:region` tag contains a reference to a task flow binding. Example 21–11 shows a sample of the metadata that JDeveloper adds to the JSF page.

Example 21–11 Metadata Added to a JSF Page to Create an ADF Dynamic Region

```
<af:region value="#{bindings.dynamicRegion1.regionModel}" id="r1"/>
```

JDeveloper also adds a task flow binding to the page definition file of the page that hosts the ADF dynamic region. Example 21–12 shows a sample of the metadata that JDeveloper adds. The task flow binding provides a bridge between the ADF dynamic region and the bounded task flow. It binds the ADF dynamic region to the bounded task flow and maintains all information specific to the bounded task flow.

Example 21–12 Metadata Added to Page Definition to Create a Task Flow Binding

```
<taskFlow id="dynamicRegion1"
      taskFlowId="${backingBeanScope.ManagedBeanName.dynamicTaskFlowId}"
              activation="deferred"
              xmlns="http://xmlns.oracle.com/adf/controller/binding"/>
```

The `taskFlowId` attribute in the task flow binding metadata specifies the managed bean that determines at runtime what bounded task flow to associate with the ADF dynamic region. JDeveloper creates this managed bean or modifies an existing managed bean to store this data. Example 21–13 shows extracts of the code that JDeveloper writes to the managed bean.

Example 21–13 Managed Bean Entries to Retrieve Bounded Task Flow for an ADF Dynamic Region

```
import oracle.adf.controller.TaskFlowId;
 ...
    private String taskFlowId = "/directoryPath/toTaskFlow";
 ...
    public TaskFlowId getDynamicTaskFlowId() {
        return TaskFlowId.parse(taskFlowId);
    }
...
```

At runtime, the managed bean stores the value of the bounded task flow's ID (`taskFlowId`) that displays inside the ADF dynamic region. The managed bean swaps the different bounded task flows into the task flow binding of the ADF dynamic region.

When an ADF dynamic region reinitializes, the Fusion web application must reinitialize the task flow binding associated with the ADF dynamic region. This includes evaluating if there are new input parameters and input parameter values to pass to the ADF dynamic region.

21.9 Adding Additional Task Flows to an ADF Dynamic Region

An ADF dynamic region link swaps a bounded task flow for another bounded task flow within an ADF dynamic region. An end user clicks a command component (for example, a button or a link) to update the ADF dynamic region with the new bounded task flow.

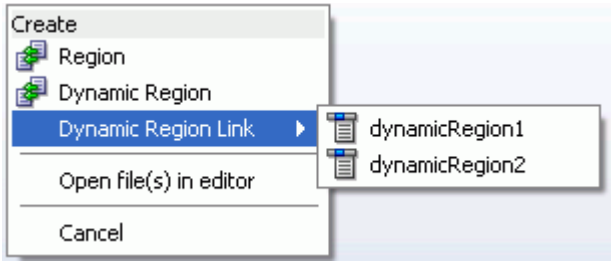
For example, a bounded task flow in the ADF dynamic region displays general information about an employee such as an ID and photo. When the end user clicks a link command component labeled **Details**, the ADF dynamic region updates with a table containing more information about the employee from another bounded task flow. The end user's action (clicking the link) invokes a method on the ADF dynamic region's managed bean. The value of the new bounded task flow is passed to the method, and the ADF dynamic region refreshes with the new bounded task flow. The new bounded task flow now displays within the ADF dynamic region.

By default, a ADF dynamic region link swaps another bounded task flow in for the original, but cannot swap back to the original. To toggle back to the original bounded task flow, you could add a second ADF dynamic region link on the page that, when clicked, swaps the current task flow back to the original one.

You can add an ADF dynamic region link if you already have at least one ADF dynamic region on a page and are adding a new bounded task flow as an ADF dynamic region to the same page. After you drop the bounded task flow on the page and choose to create an ADF dynamic region link, a menu displays all of the dynamic regions currently on the page, as shown in Figure 21–15.

JDeveloper displays a list of the current dynamic regions in a document when you choose **Dynamic Region Link** from the menu that appears after you drag and drop a bounded task that you want to add as an option to an existing dynamic region in the document, as shown in Figure 21–15.

Figure 21–15 ADF Dynamic Region Link Menu



Use this menu to select the ADF dynamic region within which you want to display the contents of the bounded task flow.

> **Tip:** You can use the values in the dynamic region link in other UI components. For example, you could create a selection list in which each of the items in the list links to a different bounded task flow. All of the linked bounded task flows would display in the same dynamic region. The links perform a method in the class behind the managed bean created to swap the bounded task flow it displays.

21.9.1 How to Create an ADF Dynamic Region Link

You drag and drop a bounded task flow to a page that already contains an ADF dynamic region and you select **Dynamic Region Link** on the context menu that JDeveloper displays to view a list of ADF dynamic regions to which you can create a link.

Before you begin:

It may be helpful to have an understanding of the configuration required before you attempt to create an ADF dynamic region link. For more information, see Section 21.9, "Adding Additional Task Flows to an ADF Dynamic Region."

You may also find it helpful to understand functionality that can be added using other task flow and ADF region features. For more information, see Section 21.1.4, "Additional Functionality for Task Flows that Render in ADF Regions."

To create an ADF dynamic region link:

1. In the Application Navigator, select and open the JSF page where you want to create the ADF dynamic region link.

   This procedure assumes that you have already added at least one ADF dynamic region to the JSF page that you open. For information about adding an ADF dynamic region to a JSF page, see Section 21.8, "Creating ADF Dynamic Regions."

2. Drag a bounded task flow and drop it anywhere on the page.

   The view activities of the bounded task flow must be associated with page fragments You can convert a bounded task flow that uses pages to use page fragments. For more information, Section 18.6.3, "How to Convert Bounded Task Flows.".

3. From the context menu that JDeveloper displays, choose **Dynamic Region Link**.

   A menu displays a list of all ADF dynamic regions that have already been added to the page.

4. Select the name of the ADF dynamic region in which you want to display the contents of the bounded task flow.

5. Click **OK**.

21.9.2 What Happens When You Create an ADF Dynamic Region

JDeveloper adds a command link to the page, as shown in Example 21–14. JDeveloper also updates the managed bean for the ADF dynamic region and updates the corresponding task flow binding with any new parameters.

Example 21–14 Dynamic Region Link

```
<af:commandLink text="region2" action="#{RegionBean.region2}"
  id="dynamicRegionLink1"/>
```

21.10 Configuring a Page To Render an Unknown Number of Regions

You can configure a page at design time when you do not know the number of regions that render in the page at runtime. For example, you want to add or remove regions to tabs using the `panelTabbed` component. Each tab that you add or remove renders a region that references a bounded task flow.

To implement this functionality, you configure the JSF page definition file's `multiTaskFlow` element to reference a list of task flow bindings contained in a managed bean of the following type:

`oracle.adf.controller.binding.TaskFlowBindingAttributes`

Example 21–15 shows the code for the class of a managed bean named `MultiBean` that returns a list containing two task flows.

Example 21–15 Managed Bean Returning a List of Task Flow Bindings

```
package view;

import java.util.ArrayList;
import java.util.List;

import oracle.adf.controller.TaskFlowId;
import oracle.adf.controller.binding.TaskFlowBindingAttributes;

public class MultiBean {
    private List<TaskFlowBindingAttributes> mTaskFlowBindingAttrs = new
ArrayList<TaskFlowBindingAttributes>(5);

    public MultiBean() {
        TaskFlowBindingAttributes tfAttr = new TaskFlowBindingAttributes();
        tfAttr.setId("region1");
        tfAttr.setTaskFlowId(new TaskFlowId("/WEB-INF/r1.xml", "r1"));
        mTaskFlowBindingAttrs.add(tfAttr);

        tfAttr = new TaskFlowBindingAttributes();
        tfAttr.setId("region2");
        tfAttr.setTaskFlowId(new TaskFlowId("/WEB-INF/r2.xml", "r2"));
        mTaskFlowBindingAttrs.add(tfAttr);

    }

    public List<TaskFlowBindingAttributes> getTaskFlowList() {
        return mTaskFlowBindingAttrs;
    }
```

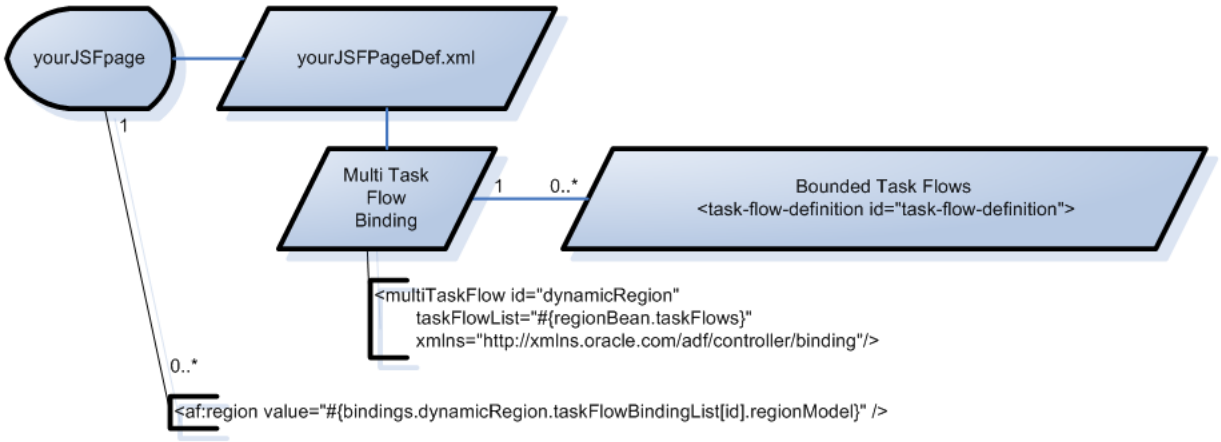
```
}
```

For more information about the `TaskFlowBindingAttributes` class, see the *Oracle Fusion Middleware Java API Reference for Oracle ADF Controller*.

At runtime, the `multiTaskFlow` binding references a list of one or more task flows that an end user can add or remove to the page using command components that you also expose. Figure 21–16 shows the relationships between the different parts that make up this use case.

Note: The page fragments that the bounded task flows render must use the Facelets document type.

Figure 21–16 ADF Regions Derived from a Multi Task Flow Binding



21.10.1 How to Configure a Page to Render an Unknown Number of Regions

Configure the JSF page definition file's `multiTaskFlow` element to reference a list of bounded task flows. This list can be modified at runtime by adding or removing elements of type `TaskFlowBindingAttributes`.

Before you begin:

It may be helpful to have an understanding of the configuration options available to you when configuring a page to render an unknown number of ADF regions. For more information, see Section 21.10, "Configuring a Page To Render an Unknown Number of Regions."

You may also find it helpful to understand other functionality that can be added using other task flow and ADF region features. For more information, see Section 21.1.3, "Task Flows and ADF Region Use Cases and Examples."

To configure a page to render an unknown number of regions:

1. Create a managed bean with a pageFlow scope. The managed bean that you create returns a list of type `TaskFlowBindingAttributes`. Example 21–15 shows the code for the class of a managed bean that returns such a list.

   For more information about how to create a managed bean, see Section 24.4.1, "How to Use a Managed Bean to Store Information."

2. In the Application Navigator, right-click the JSF page where you want to add or remove ADF regions at runtime and choose **Go to Page Definition** from the context menu.

   Click **Yes** if a confirmation dialog appears.

3. In the overview editor for the page definition file, click the **Add** icon in the Executables section to display the Insert Item dialog.

4. Select **ADF Task Flow Bindings** from the dropdown list, then select **multiTaskFlow** as the item to create and click **OK**.

5. In the Insert multiTaskFlow dialog, enter the following values:

   - **id***: Enter a unique ID for the multi task flow binding.
   - **taskFlowList ***: Enter an EL expression that returns the list of `TaskFlowBindingAttributes` to return at runtime. For example, to return the list in Example 21–15, enter the following EL expression:

     ```
     #{pageFlowScope.multiBean.taskFlowList}
     ```

6. In the Application Navigator, select and open the JSF page where you want end users to add or remove ADF regions at runtime.

7. In the ADF Faces page of the Component Palette, from the Operations panel, drag a **For Each** component and drop it on the component in the Structure window that you want to enclose the For Each component.

8. In the Property Inspector for the For Each component, enter values in the following fields:

   - **Items**: Enter an EL expression that references the method that return the list of task flow bindings from the multi task flow binding runtime object.

     For example, enter an EL expression similar to the following:

     ```
     #{bindings.multiRegion1.taskFlowBindingList}
     ```

   - **Var**: Enter a value to identify the list of task flow bindings.

9. In the JSF page that contains the For Each component, insert a Region component inside the For Each component that references the Region component's region model using an EL expression, as shown in Example 21–16.

Example 21–16 ADF Region Referencing a Region Model

```
<af:forEach var="tf" items="#{bindings.multiRegion1.taskFlowBindingList}">
  <af:panelBox text="#{tf.name}" id="pb1">
    <f:facet name="toolbar"/>
    <af:region value="#{tf.regionModel}" id="r1"/>
  </af:panelBox>
</af:forEach>
```

21.10.2 What Happens When You Configure a Page to Render an Unknown Number of Regions

Metadata similar to that shown in Example 21–17 appears in the JSF page definition file for the multi task flow binding.

Example 21–17 Metadata Added to Page Definition to Create a Multi Task Flow Binding

```
<multiTaskFlow id="multiRegion1"
```

```
taskFlowList="${pageFlowScope.multiBean.taskFlowList}" activation="deferred"
xmlns="http://xmlns.oracle.com/adf/controller/binding"/>
```

The `taskFlowList` attribute in the multi task flow binding metadata specifies the managed bean. The managed bean stores the list of objects describing the task flows that can be added to the page at runtime.

21.10.3 What You May Need to Know About Configuring a Page to Render an Unknown Number of Regions

- We recommend that you limit the number of ADF regions that you create in a page to 10.
- Each task flow binding inherits attributes defined for the multi task flow binding unless you override this behavior using the methods exposed by `TaskFlowBindingAttributes`.
- Each task flow binding inherits parameters defined in the multi task flow binding in addition to those defined using the `parametersMap` property, as described in Section 21.4, "Specifying Parameters for ADF Regions Using Parameter Maps."

22

Creating Complex Task Flows

This chapter describes how to use advanced features of task flows.

This chapter includes the following sections:

- Section 22.1, "About Creating Complex Task Flows"
- Section 22.2, "Managing Transactions in Task Flows"
- Section 22.3, "Reentering Bounded Task Flows"
- Section 22.4, "Handling Exceptions in Task Flows"
- Section 22.5, "Configuring Your Application to Use Save Points"
- Section 22.6, "Using Save Points in Task Flows"
- Section 22.7, "Using Train Components in Bounded Task Flows"
- Section 22.8, "Creating Task Flow Templates"
- Section 22.9, "Creating a Page Hierarchy Using Task Flows"

22.1 About Creating Complex Task Flows

After creating a task flow, adding activities to it, and configuring control flow between the activities, you can extend the task flow´s functionality by adding some of the features described in the following list:

- Transaction management can be configured for bounded task flows so that the Fusion web application commits or rolls back the task flow's activities as a group.
- Reentry options can be configured for bounded task flows to determine the response if a user clicks the browser's back button.
- Task flows contain a number of features that allow you to proactively handle exceptions. In addition, you can write custom code to handle exceptions thrown by a task flow
- Task flows can also be configured to capture the state of a Fusion web application at a particular instance using save points. These allow you to save the application's state if, for example, a user leaves a page without finalizing a task.
- Train components are ADF Faces components that render navigation items to guide a user through a multistep process. You can configure a bounded task flow to render these navigation items by default for the view activities in a task flow.
- Task flow templates can serve as a starting point for the creation of task flows that share common elements.

- Create a page hierarchy from the view activities that the default unbounded task flow in your application (and additional task flows in the application) reference.
- Invoke initializers and finalizers when a bounded task flow is entered and exited. An *initializer* is custom code that is invoked when a bounded task flow is entered. A *finalizer* is custom code that is invoked when a bounded task flow is exited via a task flow return activity or because an exception occurred. The finalizer is a method on a managed bean. Common finalizer tasks include releasing all resources acquired by the bounded task flow and performing cleanup before exiting the task flow.

  You specify both the initializer and the finalizer as an EL expression for a method on a managed bean, for example:

  ```
  #{pageFlowScope.modelBean.releaseResources}
  ```

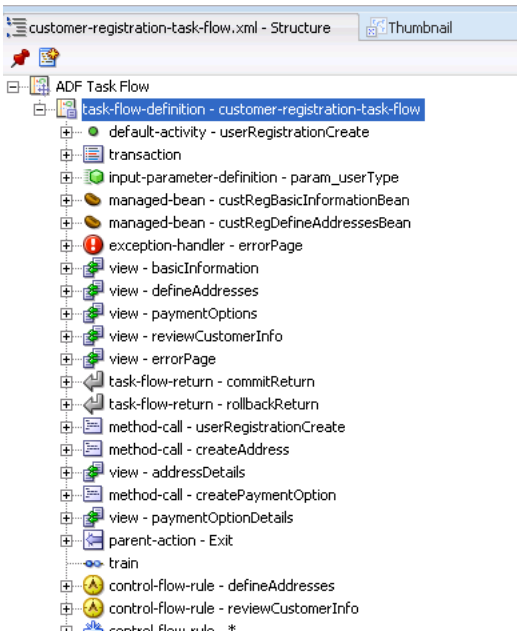
  There are two techniques for running initializer code at the beginning of the bounded task flow, depending on whether or not the task flow may be reentered via a browser back button:

  - No reentry via back button expected: Designate a method call activity as the default activity (first to execute) in the bounded task flow. The method call activity calls a custom method containing the intializer code. For more information, see Section 19.5, "Using Method Call Activities" and Section 26.4.4, "What You May Need to Know About the Browser Back Button and Navigating Through Records".
  - Back button reentry possible: Specify an intializer method using an option on the bounded task flow metadata. For more information, see Section 18.2, "Creating a Task Flow". Use this technique if you expect that a user may reenter the task flow using a browser back button. In such a case, the designated default activity for the bounded task flow may never be called, but a method designated using the Initializer method will.

22.1.1 Complex Task Flows Use Cases and Examples

Figure 22–1 shows the Structure window for the customer registration task flow in the Fusion Order Demo application. This task flow implements many of the use cases discussed in Section 22.1, "About Creating Complex Task Flows." For example, it specifies an error page to handle errors, makes use of the ADF Faces `train` component to render navigation items, and specifies two task flow return activities that either commit or roll back changes that the task flow makes as part of transaction management. For more information about the Fusion Order Demo application that contains this task flow, see Chapter 2, "Introduction to the ADF Sample Application."

Figure 22–1 Customer Registration Task Flow in the Structure Window



22.1.2 Additional Functionality for Complex Task Flows

You may find it helpful to understand other ADF features before you configure or use task flows. Additionally, you may want to read about what you can do with your task flows. Following are links to other functionality that may be of interest.

- You can specify exception handlers to manage errors that occur during execution of task flows. For more information about error handling, see Section 16.3, "Customizing Error Handling."
- Task flow save points can save application state. For more information about application state management, see Section 43, "Application State Management."
- Bounded task flows can be secured by defining the privileges that are required for someone to use them. For more information, see Section 35, "Enabling ADF Security in a Fusion Web Application."
- You can use Business Process Execution Language (BPEL) with task flows to:
  - Invoke a BPEL process from an unbounded or bounded task flow to perform a function or use services
  - Call a bounded task flow from a BPEL process in order to model user interactions with a web interface

  You can use a task flow method call activity to invoke a managed bean method or a web service that interacts with a BPEL process. For more information about the task flow method call activity, see Section 19.5, "Using Method Call Activities."
- You can write custom code that extends your task flows. For example, you can write custom code to invoke when a task flow throws an exception, as described in Section 22.4.3, "How to Designate Custom Code as an Exception Handler." For information about the APIs that you can use to write custom code, see the following reference documents:
  - *Oracle Fusion Middleware Java API Reference for Oracle ADF Controller*
  - *Oracle Fusion Middleware Java API Reference for Oracle ADF Faces*

22.2 Managing Transactions in Task Flows

A *transaction* is a persisted collection of work that can be committed or rolled back together as a group. You can use a bounded task flow to represent a transaction and to declaratively manage transaction boundaries. In the Fusion Order Demo application, the customer registration and employee registration task flows are both implemented with the use of task flow return activities. The **Cancel** button implements rollback in the task flows. The **Register** button on the `reviewCustomerInfo.jsff` and `reviewEmployeeInfo.jsff` page fragment files implement commit functionality.

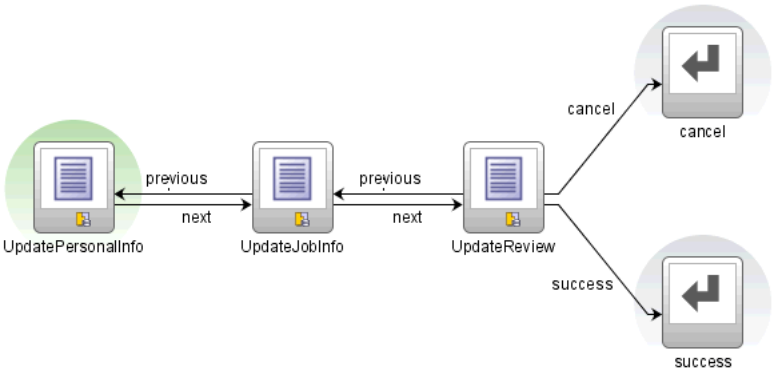
An end user can navigate from the shopping cart to initiate a backorder request for an out-of-stock item. The backorder request application is implemented as a bounded task flow that initiates a new transaction upon entry.

Transaction options on the called bounded task flow specify whether a called bounded task flow should join an existing transaction, create a new one, or create a new one only if there is no existing transaction.

If the called bounded task flow is able to start a new transaction (based on the `transaction` option that you selected), you can specify whether the transaction will be committed or rolled back when the task flow returns to its caller. The commit and rollback options are set on the task flow return activity that returns control back to the calling task flow. The same task flow that starts a transaction must also resolve the transaction.

In a called bounded task flow, you can specify two different return task flow activities that result in either committing or rolling back a transaction in the called bounded task flow. Each of the task flow return activities passes control back to the same calling task flow. The difference is that one task flow return activity specifies the commit option, while the other specifies the rollback option. As shown in Figure 22–2, if transaction processing successfully completes, control flow passes to the success task flow return activity, which specifies options to commit the transaction. If the transaction is cancelled before completion, the cancel task flow activity specifies options to roll back the transaction.

Figure 22–2 Task Flow Return Activities in Called Bounded Task Flow



If no transaction option is specified, a transaction is not started on entry of the called bounded task flow. A runtime exception is thrown if the bounded task flow attempts to access transactional services.

Use the `restore-save-point` option on the task flow return activity if you want to discard the changes an end user makes within a called bounded task flow when the called bounded task flow exits. ADF Controller rolls back to the previous ADF Model save point that was created when the bounded task flow was entered. The

`restore-save-point` option applies only to cases when a bounded task flow is entered by joining an existing transaction (either the `requires-existing-transaction` or `requires-transaction` option is also specified) and a save point is created upon entry.

> **Tip:** You can use a return activity to call the `commit` action or you can use a button bound to the `commit` action. If possible, use a task flow return activity. Using a task flow return activity will commit all data controls used by the ADF task flow. It also makes it easier to see where your application is doing commits and rollbacks, and is therefore easier to maintain.
>
> You must use `invokeAction` if:
>
> - You want to commit before the end of a task flow.
> - You are not using ADF Controller.
> - The task flow uses multiple data controls and you do not want to commit all of them.

22.2.1 How to Enable Transactions in a Bounded Task Flow

Define transaction options on a bounded task flow that is called by another task flow. Add a task flow return activity on the called bounded task flow that returns control to the task flow that calls the bounded task flow.

Before you begin:

It may be helpful to have an understanding of what a transaction is and how you can configure it. For more information, see Section 22.2, "Managing Transactions in Task Flows."

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Section 22.1.2, "Additional Functionality for Complex Task Flows."

To enable a bounded task flow to run as a transaction:

1. In the Application Navigator, double-click the source file for the called bounded task flow.
2. In the overview editor, click the **Behavior** navigation tab and expand the **Transaction** section.
3. Choose one of the following from the dropdown list:
   - **<No Controller Transaction>**: The called bounded task flow does not participate in any transaction management.

     It is recommended that you do not select this option for a bounded task flow that is parent to multiple child bounded task flows that also have the **<No Controller Transaction>** setting. Instead, select **Use Existing Transaction if Possible** for the parent task flow to avoid the creation of multiple database connections by each child bounded task flow.
   - **Always Use Existing Transaction**: When called, the bounded task flow participates in an existing transaction already in progress.
   - **Use Existing Transaction If Possible**: When called, the bounded task flow either participates in an existing transaction if one exists, or starts a new transaction upon entry of the bounded task flow if one doesn't exist.

- **Always Begin New Transaction**: A new transaction starts when the bounded task flow is entered, regardless of whether or not a transaction is in progress. The new transaction completes when the bounded task flow exits.

Note: After choosing a transaction option, you may also need to select the **Share data controls with calling task flow** option for the bounded task flow to determine whether there are any interactions between the options. For more information, see Section 20.4.3, "What You May Need to Know About Managing Transactions."

---

4. Optionally, deselect the **Share data controls with calling task flow** checkbox so that data controls are not shared with the calling task flow if you chose one of the following options in step 3:
   - **Use Existing Transaction If Possible**
   - **Always Begin New Transaction**

   The default behavior is to share data controls. For more information, see Section 22.2.3, "What You May Need to Know About Data Control Scoping with Task Flows."
5. Optionally, select the **No save point on task flow entry** checkbox to prevent the creation of an ADF Model save point on task flow entry if you chose one of the following options in step 3:
   - **Always Use Existing Transaction**
   - **Use Existing Transaction If Possible**

   An ADF Model save point is a saved snapshot of the ADF Model state. Selecting the **No save point on task flow entry** checkbox means that overhead associated with a save point is not created for the transaction.
6. Select the task flow return activity in the called bounded task flow.
7. In the Property Inspector, expand the **Behavior** section.
8. If the called bounded task flow supports creation of a new transaction (bounded task flow specifies **Use Existing Transaction If Possible** or **Always Begin New Transaction** options), select one of the following in the **End Transaction** dropdown list:
   - **commit**: Select to commit the existing transaction to the database.
   - **rollback**: Select to roll back a new transaction to its initial state on task flow entry. This has the same effect as cancelling the transaction.
9. In the **Restore Save Point** dropdown list, select **true** if you want changes the user makes within the called bounded task flow to be discarded when the task flow exits. The save point that was created upon task flow entry will be restored.

22.2.2 What Happens When You Specify Transaction Options

Example 22–1 shows the metadata for transaction options on a called bounded task flow. The `<new-transaction>` element indicates that a new transaction always starts when the called bounded task flow is invoked.

Example 22–1 Called Bounded Task Flow Metadata

```
<task-flow-definition id="trans-taskflow-definition">
```

```
    <default-activity>taskFlowReturn1</default-activity>
    <transaction>
      <new-transaction/>
    </transaction>
    <task-flow-return id="taskFlowReturn1">
      <outcome>
        <name>success</name>
        <commit/>
      </outcome>
    </task-flow-return>
  </task-flow-definition>
```

Example 22–1 also shows the metadata for transaction options on the task flow return activity on the called task flow. The `<commit/>` element commits the existing transaction to the database. The `<outcome>` element specifies a literal outcome, for example, `success`, that is returned to the caller when the bounded task flow exits. The calling ADF task flow can define control flow rules based on this outcome to For more information about defining control flow upon return, see Section 19.7, "Using Task Flow Return Activities."

22.2.3 What You May Need to Know About Data Control Scoping with Task Flows

The ADF Model layer exposes the `DataControlFrame` interface to manage a transaction in which the data controls within the frame participate. The `DataControlFrame` interface exposes methods such as:

- `beginTransaction()`
- `commit()`
- `rollback()`

Similarly, ADF Controller allows a task flow to demarcate a transaction boundary, to begin a transaction at task flow entry, and to either commit or roll back the transaction on task flow exit. It does this by invoking methods exposed by the ADF Model layer's `DataControlFrame` interface.

ADF Controller supports the transaction options listed in Table 22–1. The behavior of these transaction options depends on whether you select or deselect the **Share data controls with calling task flow** checkbox (XML element: `<data-control-scope>`) in the overview editor for a task flow.

Table 22–1 Transaction Settings Behavior

Transaction Setting	Share Data Control Scope	Isolate Data Control Scope
<No Controller Transaction>	The `DataControlFrame` is shared without an open transaction.	A new `DataControlFrame` is created without an open transaction.
Always Begin New Transaction XML element: `<new-transaction/>`	Begins a new transaction if one is not already open and throws an exception if one is already open.	Always begins a new transaction.
Always Use Existing Transaction XML element: `<requires-existing-transaction/>`	Throws an exception if the transaction is not already open.	Invalid. The checkbox cannot be deselected.
Use Existing Transaction if Possible XML element: `<requires-transaction/>`	Begins a new transaction if one is not already open.	Always begins a new transaction.

22.3 Reentering Bounded Task Flows

To deal with cases in which the end user clicks the back button to navigate back into a bounded task flow that was already exited, you can specify `task-flow-reentry` options for the bounded task flow. These options specify whether a page in the bounded task flow can be reentered.

Upon reentry, bounded task flow input parameters are evaluated using the current state of the application, not the application state existing at the time of the original bounded task flow entry.

Note: Different browsers handle the back button differently. In order to ensure that back button navigation is properly detected across all browsers, the view activities within the task flow need to be properly configured. When a task flows uses the `reentry-not-allowed` or `reentry-outcome-dependent` option, the `redirect` attribute on each view activity within the task flow should be set to `true`. See Section 19.3, "Using URL View Activities" for more information on how to configure view activities.

22.3.1 How to Set Reentry Behavior

You can set reentry behavior on a bounded task flow by specifying `task-flow-reentry` options.

Before you begin:

It may be helpful to have an understanding of what reentry options you can configure for a bounded task flow. For more information, see Section 22.3, "Reentering Bounded Task Flows."

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Section 22.1.2, "Additional Functionality for Complex Task Flows."

To set reentry behavior:

1. In the Application Navigator, double-click the source file for the bounded task flow.
2. In the overview editor, click the **Behavior** navigation tab.
3. In the **Task Flow Reentry** dropdown list, choose one of the following:
   - **reentry-allowed**: Reentry is allowed on any view activity within the bounded task flow.
   - **reentry-not-allowed**: Reentry of the bounded task flow is not allowed. If you specify `reentry-not-allowed` on a bounded task flow, an end user can still click the browser back button and return to a page within the bounded task flow. However, if the user does anything on the page such as clicking a button, an exception (for example, `InvalidTaskFlowReentry`) is thrown indicating the bounded task flow was reentered improperly. The actual reentry condition is identified upon the submit of the reentered page.

     You can set up an exception handler to display the exception and route control flow in order to navigate to the default activity of the called bounded task flow. If the bounded task flow was not called from another bounded task flow, a normal web error is posted and handled as specified in the `web.xml` file.

- **reentry-outcome-dependent**: Reentry of a bounded task flow using the browser back button is dependent on the outcome that was received when the same bounded task flow was previously exited via task flow return activities. If specified, any task flow return activities on the called bounded task flow must also specify either `reentry-allowed` or `reentry-not-allowed` to define outcome-dependent reentry behavior.

  If you choose this option, the user can navigate to a task flow using a back button based solely on how the user originally exited the task flow. For example, a task flow representing a shopping cart can be reentered if the user exited by canceling an order, but not if the user exited by completing the order.

22.3.2 How to Set Outcome-Dependent Options

You can set outcome-dependent options on bounded task flows that have specified the `reentry-outcome-dependent` option, as described in Section 22.3.1, "How to Set Reentry Behavior."

Before you begin:

It may be helpful to have an understanding of what reentry options you can configure for a bounded task flow. For more information, see Section 22.3, "Reentering Bounded Task Flows."

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Section 22.1.2, "Additional Functionality for Complex Task Flows."

To set outcome-dependent options:

1. In the task flow diagram for the bounded task flow, select the task flow return activity.

   For information about adding a task flow return activity, see Section 19.7, "Using Task Flow Return Activities."

2. In the Property Inspector, expand the **General** section.
3. In the **Name** field, enter the name of literal outcome, for example, `success` or `failure`.
4. Expand the **Behavior** section.
5. In the **Reentry** dropdown list, choose one of the following options:

   - **reentry-allowed**: Reentry is allowed on any view activity within the bounded task flow.
   - **reentry-not-allowed**: Reentry of the bounded task flow is not allowed. If you specify `reentry-not-allowed` on a bounded task flow, an end user can still click the browser back button and return to a page within the bounded task flow. However, if the user does anything on the page such as clicking a button, an exception (for example, `InvalidTaskFlowReentry`) is thrown indicating the bounded task flow was reentered improperly. The actual reentry condition is identified upon the submit of the reentered page.

     You can set up an exception handler to display the exception and route control flow in order to navigate to the default activity of the called bounded task flow. If the bounded task flow was not called from another bounded task flow, a normal web error is posted and handled as specified in the `web.xml` file.

22.3.3 What You Should Know About Managed Bean Values Upon Task Flow Reentry

When an end user reenters a bounded task flow using a browser's back button, and reentry is allowed, the value of a managed bean is reset to the value of the managed bean before the end user exited the bounded task flow. The managed bean value is reset before a view activity in the reentered bounded task flow renders. Any changes that occur before the reentry of the bounded task flow are lost. To change this behavior, specify the `<redirect>` element on the view activity in the reentered bounded task flow. When the end user reenters the bounded task flow using the back button, the managed bean has the new value from the parent task flow, not the original value from the child task flow that is reentered.

22.4 Handling Exceptions in Task Flows

During execution of a task flow, exceptions can occur that may require some kind of exception handling, for example:

- A method call activity throws an exception.
- A custom method you have written as a task flow intializer or finalizer throws an exception.
- A user is not authorized to execute the activity.

To handle exceptions thrown from an activity or caused by some other type of ADF Controller error, you can designate one activity in a bounded or unbounded task flow as an exception handler.

When a task flow throws an exception, control flow passes to the designated exception handling activity. For example, the exception handling activity might be a view activity that displays an error message. Alternatively, the activity might be a router activity that passes control flow to a method based on an EL expression that evaluates the type of exception. For example:

```
#{controllerContext.currentViewPort.exceptionData.class.name ==
'oracle.adf.controller.ControllerException'}
```

After control flow passes to the exception handling activity, flow from the exception handling activity uses standard control flow rules. For example, you designate a router activity as the exception handling activity. At runtime, the task flow passes control to the exception handling activity (in this example, a router activity) in response to an exception. In addition to designating the router activity as an exception handler, you can define task flow control cases that the router invokes based on the type of exception that it has to handle. This allows you to manage your end user's application session gracefully when an exception occurs. For more information, see Section 18.1.3, "About Control Flows."

You can optionally specify an exception handler for both bounded and unbounded task flows. Each task flow can have only a single exception handler. However, a task flow called from another task flow can have a different exception handler from that of the caller. In addition, a region on a page can have a different exception handler from that of the task flow containing the page. The exception handler activity can be any supported activity type, for example, a view or router activity.

If a bounded task flow does not have a designated exception handler activity, control passes to an exception handler activity in a calling bounded task flow, if there is a calling task flow and if it contains an exception handler activity. The exception is propagated up the task flow stack until either an exception handler activity or the top-level unbounded task flow is reached. If no exception handler is found, the exception is propagated to the web container.

If a bounded task flow does have a designated exception handler activity, make sure the exception handler activity leaves the application in a valid state after it handles an exception. One way to do this is to redirect to a view activity in the same task flow after the exception handler activity.

Other modules, such as the ADF Model, also provide exception handling capabilities. In certain scenarios this can determine the way that your application handles exceptions. For example, a databound method activity is a method activity that has a page definition and an EL expression with the following format:

`#{bindings.somebindingname.execute}`

where *somebindingname* is a method binding defined in the page definition.

An exception thrown by any type of binding is caught by the ADF Model which calls the `reportException()` method and stores the exception in the binding container. Later, when the page renders, the error displays in the page.

When a method activity invokes a method binding, there is no page to display an error that the exception raises because the exception occurs during navigation between two pages. To make the application aware of an error, Oracle ADF rethrows the exception so that it is caught by the ADF Controller exception handler mechanism that navigates to an exception handler activity if one exists.

Keep this in mind, particularly if you decide to override methods, such as the `reportException()` method, described in Section 16.3, "Customizing Error Handling," because in that scenario both the ADF Controller and ADF Model exception handlers will be called.

22.4.1 How to Designate an Activity as an Exception Handler

You can designate an exception handler activity for a bounded task flow running as an ADF Region. If an exception occurs in the bounded task flow and it is not handled by the task flow's exception handler, the exception is not propagated up the task flow stack of the parent page. Instead, it becomes an unhandled exception.

Before you begin:

It may be helpful to have an understanding of what exception handling options you can configure for a task flow. For more information, see Section 22.4, "Handling Exceptions in Task Flows."

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Section 22.1.2, "Additional Functionality for Complex Task Flows."

To designate an activity as an exception handler for a task flow:

1. Right-click the activity in the task flow diagram, and choose **Mark Activity** > **Exception Handler**.

   A red exclamation point is superimposed on the activity in the task flow to indicate that it is an exception handler. Figure 22–3 shows an example.

Figure 22–3 Example of an Activity Designated as an Exception Handler

2. To unmark the activity, right-click the activity in the task flow diagram, and choose **Unmark Activity** > **Exception Handler**.

   If you mark an activity as an exception handler in a task flow that already has a designated exception handler, the old handler is unmarked.

22.4.2 What Happens When You Designate an Activity as an Exception Handler

After you designate an activity to be the exception handling activity for a task flow, JDeveloper updates the task flow metadata with an `<exception-handler>` element that specifies the ID of the activity, as shown in Example 22–2.

Example 22–2 <exception-handler> element

```
<exception-handler id="__8>activityID</exception-handler>
```

22.4.3 How to Designate Custom Code as an Exception Handler

Rather than designate a task flow activity as the activity to invoke, you can write custom code to invoke when a task flow throws an exception. This requires you to:

- Write a Java class that extends the class `ExceptionHandler` from the following package:

  `oracle.adf.view.rich.context.ExceptionHandler`

- Register the Java class that you write as a service in the `.adf\META-INF` directory of your Fusion web application

Example 22–3 shows custom code that checks if an exception thrown by a task flow corruptions to a particular type of error message (`ADF_FACES-30108`). If it is, the custom code redirects the task flow to the `faces/SessionExpired.jspx` page.

Example 22–3 Custom Code for an Exception Handler

```
package oracle.fodemo.frmwkext;

import javax.faces.context.ExternalContext;
import javax.faces.context.FacesContext;
import javax.faces.event.PhaseId;
import oracle.adf.view.rich.context.ExceptionHandler;

public class CustomExceptionHandler extends ExceptionHandler {

    public CustomExceptionHandler() {
        super();
    }

    public void handleException(FacesContext facesContext, Throwable throwable,
                                PhaseId phaseId) throws Throwable {
```

```
            String error_message;
            error_message = throwable.getMessage();

            if (error_message != null &&
                error_message.indexOf("ADF_FACES-30108") > -1) {
                ExternalContext ectx = facesContext.getExternalContext();
                ectx.redirect("faces/SessionExpired.jspx");
            }

            else {
                //Code to execute if the if condition is not met
                throw Throwable
            }
        }
}
```

For information about the APIs that you can use to write custom code, see the following reference documents:

- *Oracle Fusion Middleware Java API Reference for Oracle ADF Controller*
- *Oracle Fusion Middleware Java API Reference for Oracle ADF Faces*

Before you begin:

It may be helpful to have an understanding of what exception handling options you can configure for a task flow. For more information, see Section 22.4, "Handling Exceptions in Task Flows."

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Section 22.1.2, "Additional Functionality for Complex Task Flows."

To designate custom code as an exception handler:

1. In the `.adf\META-INF` directory of your application, create a directory named **services** so that you have the following directory path:

   `application_root\.adf\META-INF\services`

   where `application_root` refers to the root directory of application.

2. Create a text file named **oracle.adf.view.rich.context.ExceptionHandler** in the **services** folder.

3. Write the package name and class name of the custom code that you wrote to handle exceptions in the text file named **oracle.adf.view.rich.context.ExceptionHandler**.

   For example, if you want to register the custom code in Example 22–3, write the following:

   `oracle.fodemo.frmwkext.CustomExceptionHandler`

4. Save and close the text file.

22.4.4 What Happens When You Designate Custom Code as an Exception Handler

At runtime, the task flow passes control to the custom code that you specified if the task flow throws an exception.

22.4.5 What You May Need to Know About Handling Exceptions During Transactions

Designate an exception handling activity for a bounded task flow that is enabled to run as a transaction. A Fusion web application attempts to commit a transaction if you set `commit` as the value for a task flow return activity's End Transaction property on a bounded task flow that runs as a transaction. If an exception occurs when the Fusion web application attempts to commit a transaction, the exception handling activity receives control and provides the end user with an opportunity to correct the exception. You can use the exception handling activity (for example, a view activity) to display a warning message to an end user with information about how to correct the exception and how to recommit the transaction. For information about enabling a bounded task flow as a transaction and setting commit as a value for the End Transaction property, see Section 22.2.1, "How to Enable Transactions in a Bounded Task Flow."

22.4.6 What You May Need to Know About Handling Validation Errors

For validation errors on a JSF page, you can rely on standard JSF to attach validator error messages to specific components on a page or to the whole page. A component-level validator typically attaches an error message inline with the specific UI component. There is no need to pass control to an exception handler activity.

In addition, your application should define validation logic on data controls that are executed during the Validate Model Updates phase of the JSF lifecycle. In this way, data errors are found as they are submitted to the server without waiting until attempting the final commit.

Validations done during the Validate Model Updates phase typically do not have direct access to the UI components because the intention is to validate the model after the model has been updated. These validations are often things like checking to see whether dependent fields are in sync. In these cases, the error message is usually attached to the whole page, which this logic can access.

You should attach errors detected during the Validate Model Updates phase to the JSF page, and call `FacesContext.renderResponse()`. This signals that following this phase, the current (submitting) page should be rendered showing the attached error messages. There is no need to pass control to an exception handler activity.

For more information, see Chapter 8, "Implementing Validation and Business Rules Programmatically".

22.5 Configuring Your Application to Use Save Points

Before you can add save points to a task flow, as described in Section 22.6, "Using Save Points in Task Flows" and configure related functionality, you need to make sure that the Fusion web application allows save points. To do this, you define a value for the `<savepoint-datasource>` element in the `adf-config.xml` file to specify the JNDI name for the data source that contains the save points' database table. You may also need to run a SQL script (`adfc_create_save_point_table.sql`), as described in Section 22.5.3, "What You May Need to Know About the Database Table for Save Points," to create the database table that stores save points. Once your Fusion web application starts using save points, you can use another SQL script (`adfc_cleanup_save_point_table.sql`) to delete expired save points.

22.5.1 How to Configure Your Fusion Web Application to Use Save Points

You define a value for the `<savepoint-datasource>` element in your application's `adf-config.xml` file to specify the JNDI name for the data source that contains the save points' database table. Optionally, you can also specify an expiration time for save points.

Before you begin:

It may be helpful to have an understanding of what save point options you can configure for a task flow. For more information, see Section 22.5, "Configuring Your Application to Use Save Points."

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Section 22.1.2, "Additional Functionality for Complex Task Flows."

To configure your Fusion web application to allow save points:

1. With the Fusion web application open in JDeveloper, open the Application Resources pane in the Application Navigator.
2. Expand **Descriptors**, then expand **ADF META-INF**.
3. Right-click `adf-config.xml` and choose **Open** from the context menu.
4. On the Controller page of the overview editor, write a value for the Data Source property to specify the JNDI name for the data source that contains the save points' database table.

   For example, write the following:

   `java:comp/env/jdbc/`*`Connection1`*`DS`

   where `Connection1` is the JDeveloper connection name.
5. Optionally, write a value in seconds for the Expiration property to specify the time between when a task flow creates a save point and when the save point manager removes it. The default value is 86400 seconds.

   For more information, see Section 22.6.9, "What You May Need to Know About the Time-to-Live Period for a Save Point."
6. Save the `adf-config.xml` file.

22.5.2 What Happens When You Configure a Fusion Web Application to Use Save Points

JDeveloper generates an entry, similar to that illustrated in Example 22–4, in the `adf-config.xml` file to specify the JNDI name for the data source that contains the save points' database table.

Example 22–4 Save Point Data Source Definition in adf-config.xml

```
<adf-controller-config xmlns="http://xmlns.oracle.com/adf/controller/config">
    ...
    <savepoint-datasource>
      java:comp/env/jdbc/Connection1DS
    </savepoint-datasource>
    <savepoint-expiration>
      86399
    </savepoint-expiration>
  </adf-controller-config>
```

For more information about the `adf-config.xml` file, see Section A.11, "adf-config.xml".

22.5.3 What You May Need to Know About the Database Table for Save Points

A database table named `ORADFCSAVPT` stores save points. If this database table does not exist, it is created the first time that a save point is created if your Fusion web application has the necessary permissions to create a database table. If your Fusion web application does not have the necessary permissions, you or an administrator with the necessary permissions can use SQL scripts to create and maintain the `ORADFCSAVPT` database table. These SQL scripts are:

- `adfc_cleanup_save_point_table.sql`

  Each save point in the `ORADFCSAVPT` database table has an expiration date. Use this script to delete save points that have passed their expiration date.

- `adfc_create_save_point_table.sql`

  Use this script to create the `ORADFCSAVPT` database table that stores save points.

You can find these SQL scripts in the following directory of your JDeveloper installation:

`jdev_install``\oracle_common\common\sql`

22.6 Using Save Points in Task Flows

You can configure a task flow to capture the state of a Fusion web application at a particular instance creating what is called a *save point*. This allows you to save application state if, for example, a user leaves a page without finalizing it. The application state can be restored at a later point.

Table 22–2 describes what information a save point captures.

Table 22–2 Saved Application State Information

Saved State Information	Description
User Interface State	UI state of the current page, including selected tabs, selected checkboxes, selected table rows, and table column sort order. This state assumes the end user cannot select the browser back button on save point restore.

Table 22–2 (Cont.) Saved Application State Information

Saved State Information	Description
Managed Beans	State information saved in several possible memory scopes, including session and page flow scope. The managed beans must be serializable in order to be saved. If you have page flow scope beans that are not serializable and you attempt to create a save point, a runtime exception occurs. Request scope is not supported since its lifespan is a single HTTP request and its lifespan can't be used to store cross request application state. Save points will not save and restore application-scoped managed beans since they're not passivated in failover scenarios. Therefore, the application is always responsible for ensuring that all required application-scoped state is available. Potential naming conflicts for managed beans already existing within the session scope at restore time will not occur because multiple managed beans using the same name should not be implemented within an application.
Navigation State	Task flow call stack, which ADF Controller maintains as one task flow calls another at runtime. The task flow call stack tracks where the end user is in the application and the navigation path for getting there. The task flow stack also identifies the starting point of any persisted data transactions originated for the end user.
ADF Model State	Fusion web applications use ADF Model to represent the persisted data model and business logic service providers. The ADF Model holds any data model updates made from when the current bounded task flow begins. The model layer determines any limits on the saved state lifetime. For more information, see Chapter 43, "Application State Management".

You add a method call activity to a bounded task flow that invokes a `createSavePoint` method to create save points. Later, you use a save point restore activity to restore application state and data associated with the created save points.

The same save point can be used if a user repeatedly performs a save for later on a task flow instance that executes in one session within the same browser window. The new save point overwrites the existing save point when a user performs a save for later following navigation from page to page in a task flow. For more information about restoring a save point, see Section 22.6.3, "How to Restore a Save Point".

You can specify the `createSavePoint` method exposed by the `currentViewPort` node of the ADF Controller Objects in the Expression Builder. Alternatively, you can write a custom method that updates the save point with the values of attributes you specify in your custom method, as illustrated in Example 22–5.

Example 22–5 Example Custom Method for Creating a Save Point

```
package viewController;

import java.io.Serializable;

import oracle.adf.controller.ControllerContext;
import oracle.adf.controller.savepoint.SavePointManager;

public class SaveForLater implements Serializable {
    public SaveForLater() {
        super();
    }

    public String saveTaskFlow() {
        ControllerContext cc = ControllerContext.getInstance();
        if (cc != null) {
            SavePointManager mgr = cc.getSavePointManager();
            if (mgr != null) {
                String id = mgr.createSavePoint();
                System.out.println("Save point is being set " + id);
            ...
```

The `SavePointListener` interface exposes methods that notify clients when save point events occur. The following package contains the `SavePointListener` interface:

`oracle.adf.controller.savepoint`

> **Note:** All save points created inside a bounded task flow are deleted when the bounded task flow exits.

22.6.1 How to Add a Save Point to a Task Flow

You drag and drop a method call activity to the task flow and configure it to invoke the `createSavePoint` method or to invoke a custom method if you created one.

Before you begin:

It may be helpful to have an understanding of what save point options you can configure for a task flow. For more information, see Section 22.6, "Using Save Points in Task Flows."

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Section 22.1.2, "Additional Functionality for Complex Task Flows."

Using a save point in a Fusion web application requires that you leave the value of the `jbo.locking.mode` property set to the default value `optimistic`. The value `pessimistic` causes an old session to lock until the session has timed out. In pessimistic mode, if you run an application and change data without committing changes to the database, you may get an error when you create a save point and try to restore it at a later point. For information about the `jbo.locking.mode` property, see Section 43.11.1, "How to Confirm That Applications Use Optimistic Locking."

To add a save point to a task flow:

1. Open the bounded task flow that you want to configure and navigate to the diagram editor.
2. In the ADF Task Flow page of the Component Palette, from the Component panel, drag and drop a **Method Call** activity onto the diagram.
3. In the Property Inspector, expand the **General** node and write an EL expression for the Method property to specify the save point method that the method call activity invokes.

   If you use the Expression Builder to specify the `createSavePoint` method exposed by the `currentViewPort` node of the ADF Controller Objects, the resulting EL expression is similar to the following:

   ```
   #{controllerContext.currentViewPort.createSavePoint}
   ```
4. Use a control flow to connect the method call activity with other activities in the bounded task flow.

   For more information, see Section 18.4.1, "How to Add a Control Flow Rule to a Task Flow".
5. Optionally, configure save point options in the Fusion web application's `adf-config.xml` file to determine, for example, if implicit save points can be created for the application.

For more information, see Section 22.6.7, "How to Enable Implicit Save Points".

22.6.2 What Happens When You Add Save Points to a Task Flow

JDeveloper generates entries similar to those shown in Example 22–6 in the task flow's source file when you configure a method call activity to invoke the `createSavePoint` method.

Example 22–6 Method Call Metadata to Invoke the createSavePoint Method

```
<method-call id="methodCall1">
      <method id="__3">#{controllerContext.currentViewPort.createSavePoint}</method>
    </method-call>
```

22.6.3 How to Restore a Save Point

Use the save point restore activity to restore a previously persisted save point for an application. The save point restore activity uses the save point that was originally created by invoking the `createSavePoint` method to identify the save point to restore.

You can obtain a list of the current persisted save points with `createSavePoint`. However, ADF Controller does not determine which save points to restore. A user must select the save point from a list or the application developer must select it programmatically. The savepoint ID is then passed to a save point restore activity to perform the restore.

Before you begin:

It may be helpful to have an understanding of what save point options you can configure for a task flow. For more information, see Section 22.6, "Using Save Points in Task Flows."

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Section 22.1.2, "Additional Functionality for Complex Task Flows."

To add a save point restore activity to a bounded or unbounded task flow:

1. Open the bounded or unbounded task flow where you want to add the save point restore activity and navigate to the diagram editor.
2. In the ADF Task Flow page of the Component Palette, from the Components panel, drag a **Save Point Restore** activity and drop it on the diagram for the task flow.
3. In the Property Inspector, expand the **General** node and write an EL expression for the **Save Point ID** property that, when evaluated, retrieves the save point that was originally created when the `createSavePoint` method was invoked.

   If you use the Expression Builder to specify the `getSavePoint` method of the ADF Controller Objects, the resulting EL expression is similar to the following:

   ```
   #{SessionScope.myBean.savepointID}
   ```

22.6.4 What Happens When You Restore a Save Point

JDeveloper generates entries similar to Example 22–7 in the task flow's source file when you add a save point restore activity that gets a save point ID.

Example 22–7 Metadata for a Save Point Restore Activity in a Task Flow

```
<save-point-restore id="savePointRestore1">
      <save-point-id id="__4">#{sessionScope.myBean.savepointID}</save-point-id>
    </save-point-restore>
```

22.6.5 How to Use the Save Point Restore Finalizer

When using the save point restore activity, you may need to invoke application-specific logic as part of restoring the application state. You can write an EL expression for the Save Point Restore Finalizer property of a bounded task flow that specifies a finalizer method. The bounded task flow invokes the specified method after the task flow's state has been restored. It performs any necessary logic to make sure that the application's state is correct before proceeding with the restore.

Before you begin:

It may be helpful to have an understanding of what save point options you can configure for a task flow. For more information, see Section 22.6, "Using Save Points in Task Flows."

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Section 22.1.2, "Additional Functionality for Complex Task Flows."

To use the save point restore finalizer:

1. In the Structure window, right-click the node for the bounded task flow (`task-flow-definition`) and select **Go to Properties**.
2. In the Property Inspector, expand the **General** tab, and select **Expression Builder** from the **Save Point Restore Finalizer** dropdown menu.
3. Write an EL expression that specifies the finalizer method to invoke.

22.6.6 What Happens When a Task Flow Invokes a Save Point Restore Finalizer

JDeveloper generates entries similar to Example 22–8 in the task flow's source file when you write an EL expression for the Save Point Restore Finalizer property.

Example 22–8 Metadata to Invoke a Save Point Restore Finalizer

```
<task-flow-definition id="task-flow-definition1">
    <save-point-restore-finalizer id="__2">#{sessionScope.MyBean.invokeFinalizer}
      </save-point-restore-finalizer>
  </task-flow-definition>
```

22.6.7 How to Enable Implicit Save Points

A save point in a task flow can be categorized as *implicit* or *explicit*. An explicit save point requires an end user action before a bounded or unbounded task flow creates a save point. For example, an end user clicks a button that invokes a method call activity that, in turn, creates a save point.

An implicit save point can only originate from a bounded task flow. It includes everything from when the originating task flow creates a save point. It occurs when data is saved automatically because:

- A session times out due to end user inactivity

- An end user logs out without saving the data
- An end user closes the only browser window, thus logging out of the application
- An end user navigates away from the current application using control flow rules (for example, uses a `goLink` component to go to an external URL) and having unsaved data.

Enabling implicit save points requires you to add an element to your Fusion web application's `adf-config.xml` file and to make the bounded task flow critical.

You configure the `adf-config.xml` file in your application and the bounded task flow(s) for which you want to create implicit save points. Enabling implicit save points involves a performance cost because your Fusion web application has to do extra work that it would otherwise not do. This is why you have to explicitly enable implicit save points in your application and specify the task flows to which it applies.

Before you begin:

It may be helpful to have an understanding of what save point options you can configure for a task flow. For more information, see Section 22.6, "Using Save Points in Task Flows."

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Section 22.1.2, "Additional Functionality for Complex Task Flows."

To enable implicit save points:

1. In the Application Resources panel of the Application Navigator, expand **Descriptors**, then expand **ADF META-INF**.
2. Right-click `adf-config.xml` and choose **Open** from the context menu.
3. On the Controller page of the overview editor, select the **Enable Implicit Savepoints** checkbox.

   JDeveloper generates the following entry in the `adf-config.xml` file:

   ```
   <adf-controller-config xmlns="http://xmlns.oracle.com/adf/controller/config">
       ...
       <enable-implicit-savepoints>true</enable-implicit-savepoints>
   </adf-controller-config>
   ```

   For more information about the `adf-config.xml` file, see Section A.9, "adfc-config.xml".
4. In the Application Navigator, double-click the source file for the bounded task flow.
5. In the overview editor, click the **Behavior** navigation tab and select the **Critical** checkbox.

22.6.8 What You May Need to Know About Enabling Implicit Save Points

If multiple windows are open when the implicit save point is created, a different save point is created for each browser window. This includes everything from the root view port of the browser window on down. You can write an EL expression for the Method property of a method call activity to retrieve the list of implicit save points using the `savePointManager` node under ADF Controller Objects. The resulting EL expression is similar to the following:

```
ControllerContext.savePointManager.listSavePointIds
```

Implicit save points are generated only if a critical task flow is present in any of the page flow stacks for any view port under the current root view port. An implicit save point is not generated if the request is for an ADF Controller resource, such as:

- Task flow call activity
- Task flow return activity
- Save point restore activity
- A dialog

Implicit save points are deleted when the task flow at the bottom of the stack completes or a new implicit save point is generated, whichever comes earlier.

22.6.9 What You May Need to Know About the Time-to-Live Period for a Save Point

An application-level property (`savepoint-expiration`) that is defined in the `adf-config.xml` file determines the period between when a task flow creates a save point and when the save point manager removes it (time-to-live period). The default value is 86400 seconds (24 hours).

You can change the time-to-live period for individual save points by calling the `setSavePointTimeToLive` method on an instance of `SavePointManger` from the following package:

`oracle.adf.controller.savepoint`

An instance of `SavePointManager` can be obtained as follows:

```
SavePointManager mgr = ControllerContext.getInstance().getSavePointManager();
```

Example 22–9 shows the syntax for the `setSavePointTimeToLive` method.

Example 22–9 Syntax for the setSavePointTimeToLive Method

```
public void setSavePointTimeToLive(long timeInSeconds) {
   }
```

If you supply a value for the `setSavePointTimeToLive` method argument (`timeInSeconds` in Example 22–9) equal to or less than zero, the default value is used (86400).

The `SavePointManger` defines methods that help you manage save points. For example, it defines `getSavePoint` and `removeSavePoint` methods that can retrieve and remove save points. Note that the `removeSavePoint` method does not get called automatically when a save point expires. You must explicitly call the `removeSavePoint` method to remove save points (including expired save points) from the `ORADFCSAVPT` database table. Alternatively, Oracle ADF provides a SQL script (`adfc_cleanup_save_point_table.sql`) that removes expired save points. For more information, see Section 22.5.3, "What You May Need to Know About the Database Table for Save Points."

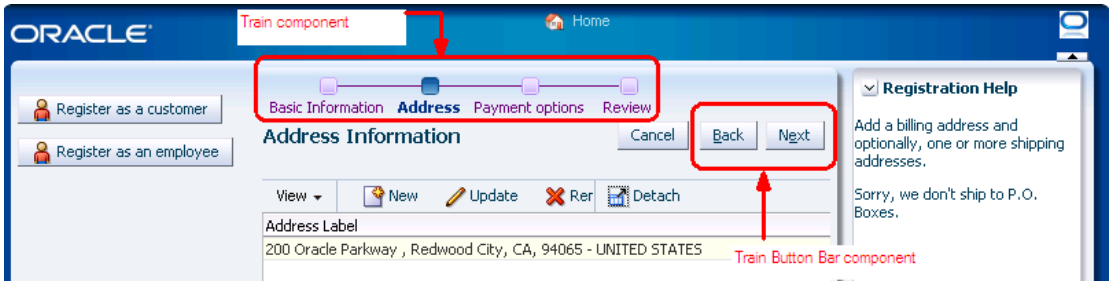
Consider calling the `setSavePointTimeToLive` method at the same time that you call the method to create save points, as illustrated in Example 22–5. For more information about the `SavePointManger`, see the *Oracle Fusion Middleware Java API Reference for Oracle ADF Controller*.

22.7 Using Train Components in Bounded Task Flows

You can create a task flow as a train. This allows you to create a user interface where you navigate end users through a multistep process. For example, the Fusion Order Demo application's customer registration task flow renders a train that navigates end users through a number of train stops in order to complete the customer registration process, as illustrated in Figure 22–4.

ADF Faces provides the user interface components that render the train functionality in the task flow view activities that appears to the end user. These components are the `train` and `trainButtonBar` components. Figure 22–4 shows the `defineAddresses.jsff` page (a task flow view activity) using the `train` and `trainButtonBar` components to navigate the end user through the address creation step in the customer registration task flow.

Figure 22–4 Train and Train Button Bar Components in the Customer Registration Task Flow



The `train` component in Figure 22–4 renders four train stops. Each stop corresponds to a page fragment in the customer registration task flow where the end user can enter and review information to complete the registration process. Figure 22–4 shows the Address train stop where the end user enters an address for postage. The other stops are:

- Basic Information
- Payment Options
- Review

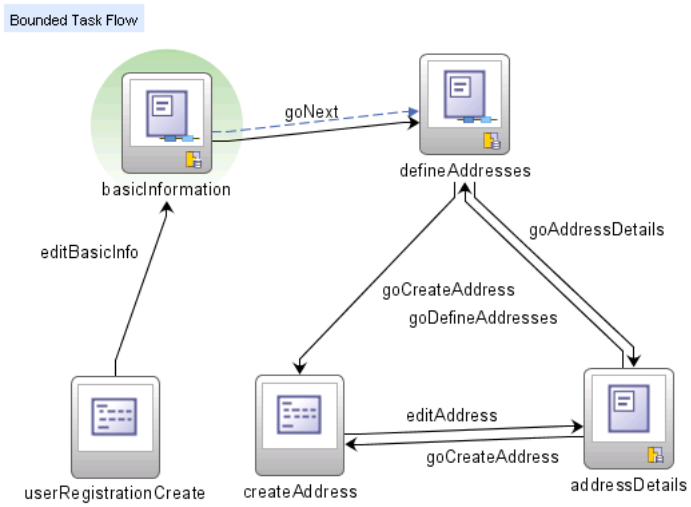
The Train Button Bar component, also shown in Figure 22–4, is optional and provides additional controls to navigate between the train stops. This component can be used in conjunction with the train component to provide multiple ways to navigate through train stops.

Bounded task flows and task flow templates can make use of these train components if you select the **Create Train** checkbox on the dialogs provided by JDeveloper to create a bounded task flow or a task flow template. A bounded task flow or a task flow template can render one train only. If you want to use multiple trains, create a separate bounded task flow or task flow template for each train.

Figure 22–5 displays an extract from the design time view of the customer registration task flow where you can see the `basicInformation` and `defineAddresses` view activities that render at runtime as the **Basic Information** and **Address** train stops in Figure 22–4.

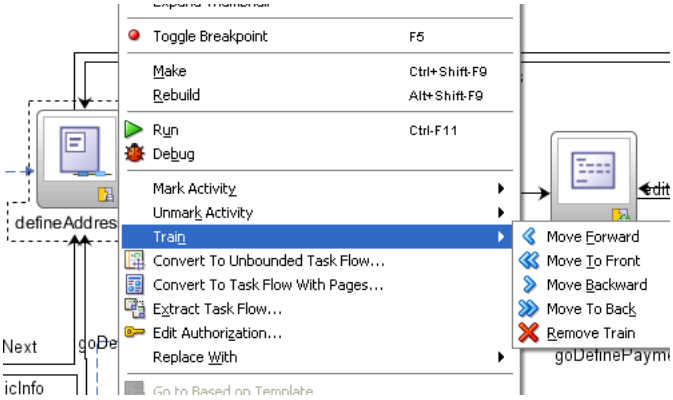
The dotted line labeled **goNext** connecting the `basicInformation` view activity to the `defineAddresses` view activity indicates the sequence in which the end user navigates between the view activities when the view activities render as train stops.

Figure 22–5 Detail of Customer Registration Task Flow



JDeveloper displays a context menu with options to change the position of the activity when your right-click a view activity or task flow call activity that is configured as a train stop, as illustrated in Figure 22–6.

Figure 22–6 Context Menu to Edit a Train Stop



Configure a task flow call activity as a train stop when you want to group a number of activities as a train stop or call a child train stop. For example, there are cases when other task flow activities, such as router and method call activities, should be considered part of a train stop along with the corresponding view activity. A method call activity might need to precede a view activity for initialization purposes. When grouped this way, the activities can be performed as a set each time the train stop is visited. For more information, see Section 22.7.3, "Grouping Task Flow Activities to Execute Between Train Stops".

Branching using router activities and control flow cases is supported on the task flow diagram containing the train, as well as in child bounded task flows called from the train.

22.7.1 Creating a Task Flow as a Train

You need to configure a bounded task flow to use train components before you can implement the type of functionality discussed in Section 22.7, "Using Train

Components in Bounded Task Flows." Use one of the methods outlined in the following list to configure a bounded task flow to use train components:

- Select the **Create Train** checkbox in the dialog that JDeveloper displays when you create a new bounded task flow or task flow template.

  For more information, see Section 18.2, "Creating a Task Flow" or Section 22.8, "Creating Task Flow Templates."

- Right-click an existing bounded task flow in the diagram and choose **Train** > **Create Train** from the context menu that JDeveloper displays.
- Open an existing bounded task flow in the overview editor, click the **Behavior** navigation tab and select the **Train** checkbox.

Once you have configured the bounded task flow to use train components, you drag and drop the task flow activities that you want to render in the train to the diagram in the bounded task flow.

22.7.1.1 How to Create a Train in a Bounded Task Flow

You drag and drop the task flow activities that you want to render in the train to the diagram for the bounded task flow.

Before you begin:

Make sure that you configure the bounded task flow to use train components, as described in Section 22.7, "Using Train Components in Bounded Task Flows."

It may be helpful to have an understanding of the configuration options that you can configure for a train. For more information, see Section 22.7.1, "Creating a Task Flow as a Train."

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Section 22.1.2, "Additional Functionality for Complex Task Flows."

To create a train in a bounded task flow:

1. Open the bounded task flow or task flow template where you want to render train stops.
2. Drag each view activity or JSF page you want to include in the train to the diagram for the bounded task flow.
   - If you drag a JSF page, JDeveloper automatically adds a view activity to the diagram.
   - Double-click any view activity that you drag to the diagram to invoke the dialog to create a new JSF page.

   You can reorder the train stop sequence at a later point. For more information, see Section 22.7.4, "Disabling the Sequential Behavior of Train Stops in a Train."
3. In the diagram for the bounded task flow, double-click each view activity that you want to define as a train stop.

   JDeveloper displays a dialog to create a new JSF page if the view activity is not yet associated with a JSF page. Use the dialog to create a new JSF page.

   If the view activity is already associated with a JSF page, JDeveloper opens the page.

4. In the ADF Faces page of the Component Palette, from the General Controls panel, in the Location group, drag a **Train**, and, optionally, a **Train Button Bar** component and drop it on the JSF page.

   You must manually add each Train and Train Button Bar component that you want to appear in a task flow view activity. Consider using a page template. For more information, see Section 24.2, "Using Page Templates."

22.7.1.2 What Happens When You Create a Task Flow as a Train

JDeveloper writes the `<train/>` element to the source file for the bounded task flow or task flow template that you enable as a train, using one of the methods outlined in Section 22.7.1, "Creating a Task Flow as a Train."

JDeveloper writes entries to the JSF page where you added `train` and `trainButtonBar` components. Example 22–10 shows code snippets from the `defineAddresses.jsff` page fragment in the Fusion Order Demo application. Both types of train component bind to instances of the train model object (`trainModel`).

Example 22–10 Metadata for Train Components in the Fusion Order Demo Application

```
<af:train value="#{controllerContext.currentViewPort.taskFlowContext.trainModel}"
        id="t1"/>
 ...
 <af:trainButtonBar value="#{controllerContext.currentViewPort.taskFlowContext.trainModel}"
       id="tbb1"/>
```

Dotted lines appear between each view activity that you add to the bounded task flow and define as a train stop, as illustrated in Figure 22–5. The dotted lines indicate the order in which the bounded task flow navigates through the train stops. JDeveloper defines the sequence in the order that you add the view activities to the bounded task flow. You can change the sequence. For more information, see Section 22.7.4, "Disabling the Sequential Behavior of Train Stops in a Train." You can also skip a train stop. For more information, see Section 22.7.6, "Configuring a Train to Skip a Train Stop."

22.7.2 Invoking a Child Bounded Task Flow from a Train Stop

An alternative approach to grouping task flow activities in one train stop, as described in Section 22.7.3, "Grouping Task Flow Activities to Execute Between Train Stops," is to create another bounded task flow as a train that you invoke from your current bounded task flow. We refer to this bounded task flow as a *child bounded task flow*. You configure the parent bounded task flow to invoke it using a task flow call activity. To implement this functionality, you:

- Create a child bounded task flow as a train
- Add the task flow activities that you want end users to access from the train stop on the parent bounded task flow to the child bounded task flow
- Designate a task flow call activity as a train stop in the parent bounded task flow to invoke the child bounded task flow
- Add a task flow return activity to the child bounded task flow that returns control to the parent bounded task flow once the child bounded task flow finishes execution

Task flow activities in the child bounded task flow execute together regardless of whether the end user visits the train stop for the first time or returns at a later point. Nonview task flow activities in the child bounded task flow typically precede the view

activity in the child bounded task flow's flow control. You can invoke a child bounded task flow from a bounded task flow that itself is a child to another bounded task flow.

22.7.2.1 How to Invoke a Child Bounded Task Flow From a Train Stop

You create a bounded task flow as a train, add task flow activities to it, and configure a task flow call activity on the parent bounded task flow to invoke it.

Before you begin:

It may be helpful to have an understanding of the configuration options that you can configure for a child bounded task flow that you invoke from a train. For more information, see Section 22.7.2, "Invoking a Child Bounded Task Flow from a Train Stop."

You may also find it helpful to read about the additional functionality that you can add using other task flow features. For more information, see Section 22.1.2, "Additional Functionality for Complex Task Flows."

To invoke a child bounded task flow from a train stop:

1. Create a child bounded task flow as a train. For more information, see Section 22.7.1, "Creating a Task Flow as a Train."
2. Add the task flow activities that you want the child bounded task flow to invoke to the child bounded task flow. In particular, make sure that you add a task flow return activity to the child bounded task flow to return control to the parent task flow when the child bounded task flow finishes execution.

   For more information, see Section 19.7, "Using Task Flow Return Activities."
3. Add a task flow call activity to the parent task flow to render as a train stop that invokes the child bounded task at runtime.

   For more information, see Section 19.6, "Using Task Flow Call Activities."

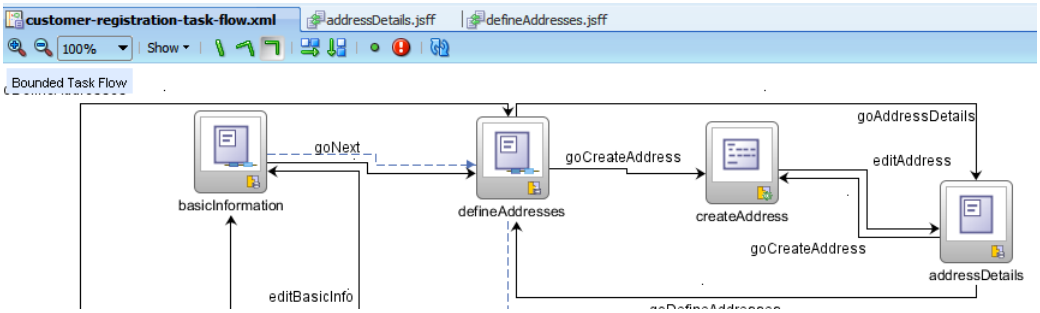
22.7.3 Grouping Task Flow Activities to Execute Between Train Stops

You can group task flow activities together to form a single train stop. For example, the train stop in the Fusion Order Demo application that renders as **Address** at runtime groups together the following task flow activities:

- `defineAddress` view activity
- `createAddress` method call activity
- `addressDetails` view activity

Each time a user visits the runtime **Address** train stop, the customer registration task flow provides access to the task flow activities listed previously. The `defineAddresses` view activity renders the train stop and the associated `defineAddresses.jsff` page fragment. The `defineAddresses.jsff` page fragment, in turn, exposes a command button to invoke the `createAddress` method call activity. This method call activity validates user input in the Address page and defines an outcome that (optionally) passes control to the `addressDetails` view activity. The `addressDetails` view activity renders the Address Information page at runtime where an end user may choose to enter additional addresses.

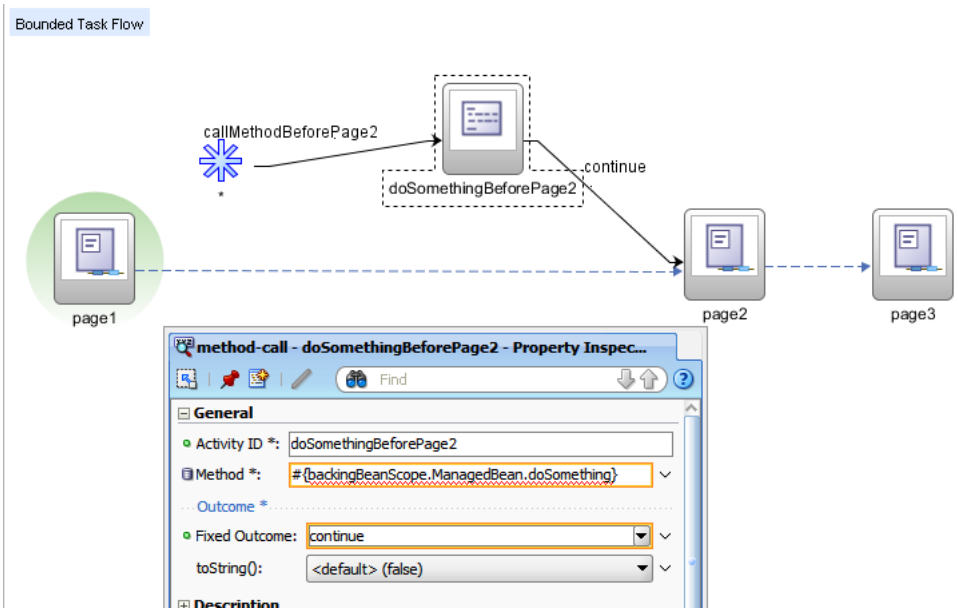
Figure 22–7 Task Flow Activities Grouped into One Train Stop



The train considers all task flow activities that lead from the first nonview activity through the next view activity to be part of the train. Make sure that all task flow activities, except for view and task flow call activities, for the train stop follow the view or task flow call activity that you define as a train stop.

You can also group task flow activities to execute between train stops. Figure 22–8 shows a possible configuration where a method call activity (`doSomethingBeforePage2`) invokes before control flow passes to `page2`. The `callMethodBeforePage2` wildcard control flow rule passes control to the `doSomethingBeforePage2` method call activity. This method call activity defines a fixed outcome (continue) that passes control to the next train stop (`page2`) in the train.

Figure 22–8 Method Call Activity Between Train Stops



22.7.4 Disabling the Sequential Behavior of Train Stops in a Train

By default, train stops are sequential. This means end users can select a train stop only after visiting the previous train stop in the train. Figure 22–9 shows the train component that the customer registration task flow renders. End users cannot visit the **Payment** options and **Review** train stops until they first visit the **Address** train stop. All train stops in this train are sequential.

Figure 22–9 Sequential Train Stops in the Customer Registration Task Flow

You can change this default behavior (make a train stop nonsequential) by configuring the view activity that renders the train stop to make it nonsequential.

22.7.4.1 How to Disable the Sequential Behavior of a Train

You write a value for the task flow view activity's `sequential` property that evaluates to `false` at runtime.

Before you begin:

It may be helpful to have an understanding of the configuration options that you can configure for a train. For more information, see Section 22.7, "Using Train Components in Bounded Task Flows."

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Section 22.1.2, "Additional Functionality for Complex Task Flows."

To disable the sequential behavior of a train stop:

1. In the Application Navigator, double-click the source file for the bounded task flow containing the view activity that renders the train stop.
2. In the diagram for the bounded task flow, select the view activity.
3. In the Property Inspector, expand the **Train Stop** and write a value in the **Sequential** field to determine if the train stop is sequential or nonsequential.

   Write `false` or write an EL expression that resolves to `false` at runtime to make the train stop nonsequential. For example, write an EL expression similar to the following:

   ```
   #{myTrainModel.isSequential}
   ```

22.7.4.2 What Happens When You Disable the Sequential Behavior a Train Stop

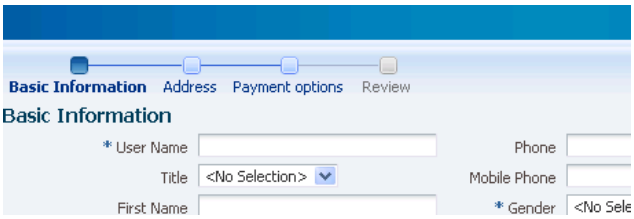
JDeveloper writes an entry to the source file for the bounded task flow, as illustrated in Example 22–11.

Example 22–11 Metadata to Disable the Sequential Behavior of a Train Stop

```
<view id="paymentOptionDetails">
      <page>/account/paymentOptionDetails.jsff</page>
      <train-stop id="__1">
        <sequential>#{myTrainModel.isSequential}</sequential>
      </train-stop>
    </view>
```

Figure 22–10 shows the corresponding runtime view when the value specified for `sequential` evaluates to `false`. End users can now navigate directly to payment options by clicking the **Payment options** train stop.

Figure 22–10 Train with a Nonsequential Train Stop



22.7.5 Changing the Label of a Train Stop

You can configure a train stop to display a label that you define. The label that you define can be a static value or a value in a resource bundle that you reference using an EL expression, for example. For more information about using localized strings, see the "Internationalizing and Localizing Pages" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

22.7.5.1 How to Change the Label of a Train Stop

You change the label of a train stop by configuring a value for the view activity's Display Name property.

Before you begin:

It may be helpful to have an understanding of the configuration options that you can configure for a train. For more information, see Section 22.7, "Using Train Components in Bounded Task Flows."

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Section 22.1.2, "Additional Functionality for Complex Task Flows."

To change the label of a train stop:

1. In the Application Navigator, double-click the source file for the bounded task flow containing the view activity that renders the train stop.
2. In the diagram for the bounded task flow, select the view activity.
3. In the Property Inspector, expand the **Description** section and write a value in the **Display Name** to specify the label that the train stop renders at runtime.

   Write a literal value or write an EL expression that references a value in a resource bundle to render at runtime.

22.7.5.2 What Happens When You Change the Label of a Train Stop

JDeveloper writes an entry to the source file for the bounded task flow, as illustrated in Example 22–12.

Example 22–12 Metadata to Change the Label of a Train Stop

```
<view id="basicInformation">
      <display-name>Basic Information</display-name>
      <page>/account/basicInformation.jsff</page>
      <train-stop>
        <display-name>Basic Information</display-name>
      </train-stop>
    </view>
```

At runtime, the Fusion web application displays the value you specified for the `display-name` property as the label for the train stop.

22.7.6 Configuring a Train to Skip a Train Stop

You can configure a train so that it skips an individual train stop. At runtime, the Fusion web application disables the train stop that you configure to skip. The end user can only navigate to next train stop in the train.

Implement this functionality if you want the train to execute at a train stop other than the first train stop. For example, configure train stops 1 and 2 to skip if you want to execute train stop 3 first. Use this approach rather than defining the view activity associated with the train stop as the default activity, as discussed in Section 18.2.3, "What You May Need to Know About the Default Activity in a Bounded Task Flow."

22.7.6.1 How to Configure a Train to Skip a Train Stop

You write a value for the view activity's `skip` property that evaluates to `true` at runtime.

Before you begin:

It may be helpful to have an understanding of the configuration options that you can configure for a train. For more information, see Section 22.7, "Using Train Components in Bounded Task Flows."

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Section 22.1.2, "Additional Functionality for Complex Task Flows."

To configure a train to skip a train stop:

1. In the Application Navigator, double-click the source file for the bounded task flow containing the view activity that renders the train stop.
2. In the diagram for the bounded task flow, select the view activity.
3. In the Property Inspector, expand the **Train Stop** and write a value in the **Skip** field to determine if the train navigates to the train stop or skips it.

   Write `true` or write an EL expression that resolves to `true` at runtime to make the train skip the train stop. For example, write an EL expression similar to the following:

   ```
   #{myTrainModel.shouldSkip}
   ```

22.7.6.2 What Happens When You Configure a Train to Skip a Train Stop

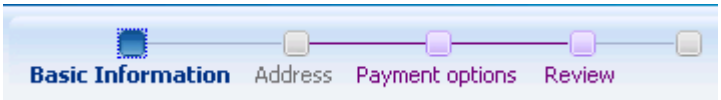
JDeveloper writes an entry to the source file for the bounded task flow, as illustrated in Example 22–13.

Example 22–13 Metadata to Make a Train Skip a Train Stop

```
<view id="defineAddresses">
     <display-name>Address</display-name>
     <page>/account/defineAddresses.jsff</page>
     <train-stop>
       <display-name>Address</display-name>
        <skip>#{myTrainModel.shouldSkip}</skip>
     </train-stop>
   </view>
```

Example 22–11 shows the corresponding runtime view when the value specified for `skip` evaluates to `true`. End users must now navigate to the next train stop in the train (Payment options) if the train is sequential.

Figure 22–11 Train Configured to Skip a Train Stop



22.8 Creating Task Flow Templates

You can create task flow templates for yourself or other application developers to use as a starting point when creating new bounded task flows. Unbounded task flows cannot be created using task flow templates. A bounded task flow created from a task flow template will have definitions for the same set of task flow activities, control flows, input parameters, and managed beans as the task flow template.

An example use case for a task flow template is where you define a task flow activity as an exception handler on a bounded task flow. The task flow activity could be a view activity associated with a page that you display to end users when an exception occurs. Rather than define this view activity in each bounded task flow, you define it in a task flow template and select the task flow template when you create new bounded task flows. For more information about exception handling, see Section 22.4, "Handling Exceptions in Task Flows."

You can base a new task flow template on an existing task flow template. You can also refactor an existing bounded task flow to create a new task flow template. For more information, see Section 18.6.3, "How to Convert Bounded Task Flows".

When you create a bounded task flow or another task flow template based on a task flow template, you can specify that subsequent changes you make to a task flow template be automatically propagated to bounded task flows and templates. To do this, you select the **Update the Task Flow When the Template Changes** checkbox in the dialog that you use to create a bounded task flow or a template. Subsequent changes that you make to the task flow template (add new view activities, for example) get propagated to the bounded task flows. You can change, update, or disassociate the parent task flow template of a child bounded task flow or task flow template at any point during development of the child.

At runtime, the contents of a child bounded task flow or a child task flow template combine with the contents of the parent task flow template if you selected the **Update the Task Flow When the Template Changes** checkbox when creating the child. The child task flow and task flow template override the parent task flow template when conflict occurs, even if you selected the **Update the Task Flow When the Template Changes** checkbox when creating the child task flow or task flow template. For example, you create a parent task flow template to use as a train, as described in Section 22.7, "Using Train Components in Bounded Task Flows," and then create a bounded task flow based on this task flow template. Later, you disable the train component on the parent task flow template. The child task flow overrides the parent task flow template and continues to execute as a train.

Table 22–3 describes in more detail how ADF Controller resolves conflicts between parent task flow templates and child task flows and child task flow templates.

Table 22–3 Conflict Resolution between Parent Templates and Child Task Flows

Bounded Task Flow Metadata	Combination Algorithm
Default activity	Child bounded task flow or child task flow template overrides parent task flow template.
Transaction	Child bounded task flow or child task flow template overrides parent task flow template as an entire block of metadata, including all subordinate elements.
Task flow reentry	Child bounded task flow or child task flow template overrides parent task flow template as an entire block of metadata, including all subordinate elements.
Control flow rules	Combination algorithm occurs at the control flow case level, not the control flow rule level. Control flow cases fall into the following categories: ■ Both from action and from outcome specified ■ Only from action specified ■ Only from outcome specified ■ Neither from action nor from outcome specified Each of these categories is merged additively. The child bounded task flow or template overrides parent task flow template for identical matches within each of the four categories.
Input parameter definitions	Child bounded task flow or child task flow template overrides parent task flow template for identical input parameter definition names.
Return value definitions	Child bounded task flow or child task flow template overrides parent task flow template for identical return value definition names.
Activities	Child bounded task flow or child task flow template overrides parent task flow template for identical activity IDs.
Managed beans	Child bounded task flow or child task flow template overrides parent task flow template for identical managed bean names.
Initializer	Child bounded task flow or child task flow template overrides parent task flow template.
Finalizer	Child bounded task flow or child task flow template overrides parent task flow template.
Critical	Child bounded task flow or child task flow template overrides parent task flow template.
Use page fragments	Child bounded task flow or child task flow template overrides parent task flow template.
Exception handler	Child bounded task flow or child task flow template overrides parent task flow template.
Security - permission	Child bounded task flow or child task flow template overrides parent task flow template. Privilege maps are additive. Child bounded task flow or child task flow template overrides parent task flow template for identical privilege map operations.
Security - transport guarantee	Child bounded task flow or child task flow template overrides parent task flow template.
Train	Child bounded task flow or child task flow template overrides parent task flow template.

Validations at both design time and runtime verify that the resulting parent-child extension hierarchy does not involve cycles of the same task flow template.

22.8.1 How to Create a Task Flow Template

You create a task flow template by selecting ADF Task Flow Template from JDeveloper's New Gallery.

Before you begin:

It may be helpful to have an understanding of what task flow template options you can configure. For more information, see Section 22.8, "Creating Task Flow Templates."

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Section 22.1.2, "Additional Functionality for Complex Task Flows."

To create a task flow template:

1. In the Application Navigation, right-click the project where you want to create the task flow and choose **New**.
2. In the New Gallery, expand **Web Tier**, select **JSF/Facelets** and then **ADF Task Flow Template**, and click **OK**.
3. In the Create ADF Task Flow Template dialog, choose the appropriate options:
   - **File Name**: Accept the default value proposed by JDeveloper or enter a new file name. The value is used to name the XML source file for the task flow template you create. The source file includes the activities and control flow rules that are in the task flow template. The default name for the XML source file is `task-flow-template.xml`.
   - **Create with Page Fragments**: Leave this checkbox selected (the default) if you expect that a bounded task flow based on the template will be used as an ADF region. Clear the checkbox if you want to add JSF pages instead of JSF page fragments to a task flow based on the task flow template.
4. Click **OK**.

22.8.2 What Happens When You Create a Task Flow Template

As shown in Example 22–14, an XML file is created each time you create a new task flow template using JDeveloper. You can find the XML file in the Application Navigator in the location that you specified in the **Directory** field of the Create ADF Task Flow Template dialog, for example, `. . . /WEB-INF`.

The contents of the XML source file for the task flow template can be similar to those of a bounded task flow. One difference is the inclusion of the `<task-flow-template>` tag.

Example 22–14 Task flow template source file

```
<?xml version="1.0" encoding="windows-1252" ?>
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2" id="__
1">
  <task-flow-template id="task-flow-template">
     <default-activity>view1</default-activity>
     <view id="view1">view1.jsff</view>
  </task-flow-template>
</adfc-config>
```

22.8.3 What You May Need to Know About Task Flow Templates

If you use a task flow template that contains bindings, you must change the component IDs of task flows based on the task flow template. Doing this makes sure that the IDs are unique. Task flows generated from the template inherit the same ID as the template. This may cause an exception at runtime. For more information, see Section 24.2.1, "How to Use ADF Data Binding in ADF Page Templates."

If your Fusion web application has configured ADF security, you have to make grants on both the task flow template and task flows that use the template. Usually, the grant on the task flow template is to the `anonymous-role` application role so that the grant that really matters is on the task flow, but not always. For more information about ADF security, see Section 35, "Enabling ADF Security in a Fusion Web Application."

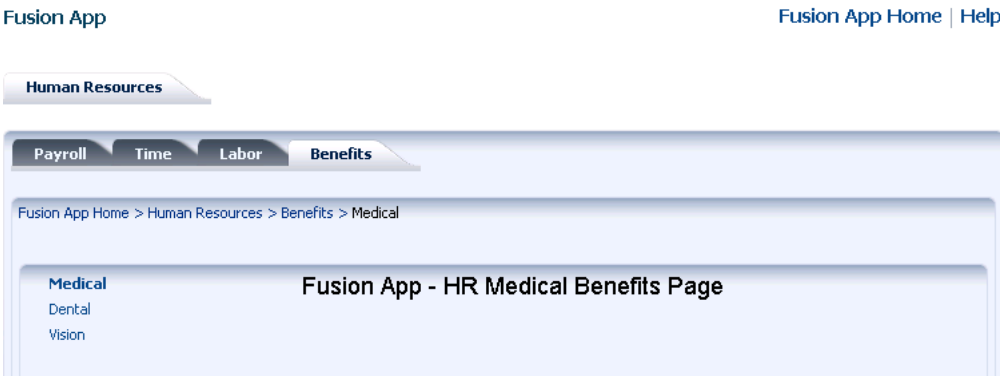
22.9 Creating a Page Hierarchy Using Task Flows

Creating a page hierarchy is a useful way of organizing the JSF pages in your Fusion web application so that end users can more easily navigate the application. End users access information on the pages by navigating a path of links. Figure 22–12 shows a sample page hierarchy.

Figure 22–12 Page Hierarchy

To navigate this hierarchy, an end user clicks links on each page to drill down or up to another level of the hierarchy. For example, clicking **Human Resources** on the Fusion App home page displays the Human Resources page hierarchy shown in Figure 22–12. Clicking the link on the **Benefits** tab displays the page hierarchy shown in Figure 22–13.

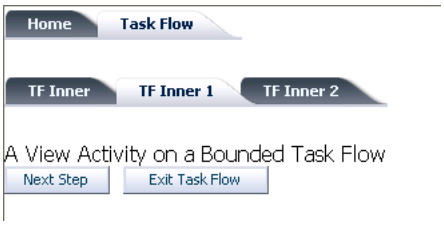
Figure 22–13 Benefits Page



The user can click links on the Benefits page to display other pages, for example, the Medical, Dental or Vision pages. The breadcrumbs on each page indicate where the current page fits in the hierarchy. The user can click each node in a breadcrumb to navigate to other pages in the hierarchy. The bold tab labels match the path to the current page shown the breadcrumbs.

Pages referenced by view activities in a bounded task flow can also be included in any page hierarchy that you generate. Figure 22–14 shows the runtime view of a page hierarchy that renders view activities referenced by a bounded task flow.

Figure 22–14 Runtime Menu Hierarchy Including a Bounded Task Flow



You can use ADF Controller with an `XMLMenuModel` implementation to create the previously-discussed page hierarchies. If you do, JDeveloper generates the following for you:

- Control flow metadata that determines the view or page to display when an end user selects a menu item
- An `XMLMenuModel` metadata file
- Default navigation widgets such as **Previous** and **Next** button
- Breadcrumbs
- Managed bean configuration

If you decide not to use ADF Controller, you can create the page hierarchy using an `XMLMenuModel` implementation. For more information about this method of building a page hierarchy, see the "Using a Menu Model to Create a Page Hierarchy" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

22.9.1 How to Create a Page Hierarchy

Create an unbounded task flow or open an existing one. Add view activities or bounded task flows to the unbounded task flow. Each view activity or bounded task

flow that you add to the unbounded task flow contains references to pages to appear in the proposed page hierarchy. Use JDeveloper's Create ADF Menu Model dialog to generate an `XMLMenuModel` metadata file. Organize the item nodes in the generated `XMLMenuModel` metadata file to create the page hierarchy you want. Connect submenus to parent menus to finalize the hierarchy.

Figure 22–15 shows an example page hierarchy that consists of view activities:

- The top-level menu (**Home Page**) is the root parent page. It contains a single tab that links to the **Human Resources** submenu.

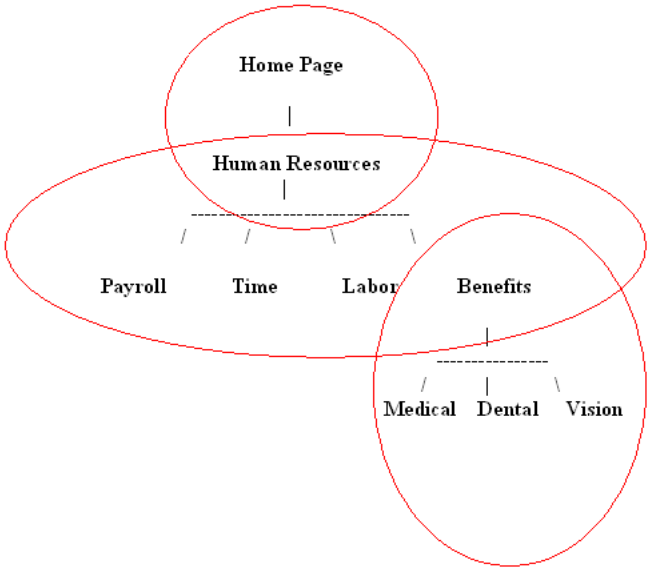
  In JDeveloper, **Home Page** page is represented as an item node and **Human Resources** page as a shared node.

- **Human Resources** has four tab links to **Payroll**, **Time**, **Labor**, and **Benefits** pages.

  In this menu, **Human Resources** is a group node that references child item nodes (**Payroll**, **Time**, and **Labor**) and a shared node (**Benefits**) that references the Benefits submenu.

- **Benefits** is a group node that references child item nodes (**Medical**, **Dental**, and **Vision**) pages.

Figure 22–15 Menu Hierarchy

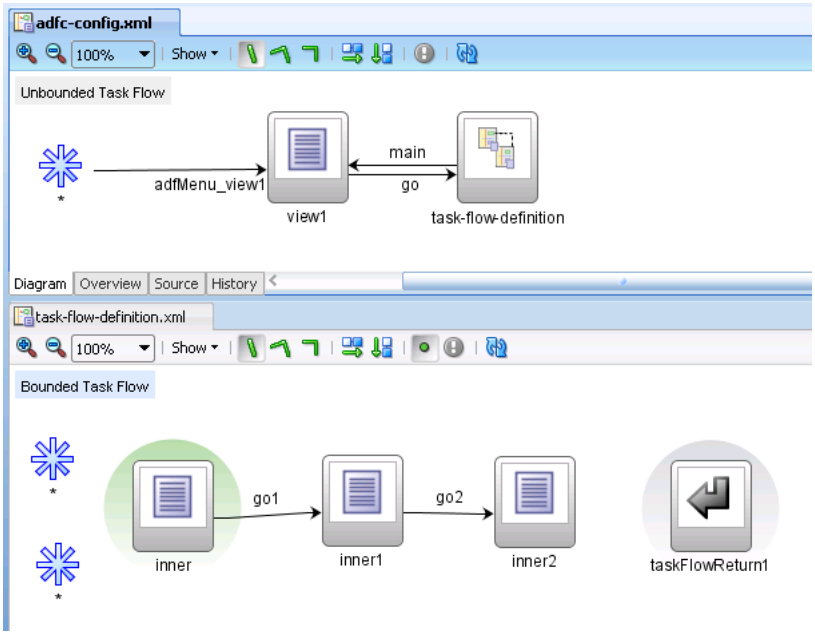


Note: It is possible to create the entire menu hierarchy in one menu model. However, breaking a menu hierarchy into submenus makes maintenance easier. In addition, breaking the menu hierarchy into smaller submenu models enables each separate development organization to develop its own menu. These separate menus can later be combined using shared nodes to create the complete menu hierarchy.

Figure 22–16 shows the corresponding design-time view in JDeveloper of the unbounded and bounded task flows that render the page hierarchy shown in Figure 22–14. The unbounded task flow (`adfc-config.xml`) contains a view activity

(**view1**) and a task flow call activity (**task-flow-definition**) that invokes the bounded task flow (`task-flow-definition.xml`) shown in the lower part of Figure 22–16.

Figure 22–16 Design Time Menu Hierarchy Including a Bounded Task Flow



22.9.1.1 How to Create an XMLMenuModel Metadata File

You use JDeveloper's Create ADF Menu Model dialog to generate an `XMLMenuModel` metadata file once you have defined what menus (unbounded task flows) and nodes (pages) you want to appear in the final page hierarchy.

Before you begin:

It may be helpful to have an understanding of what page hierarchy options you can configure. For more information, see Section 22.9, "Creating a Page Hierarchy Using Task Flows."

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Section 22.1.2, "Additional Functionality for Complex Task Flows."

To create the XMLMenuModel metadata file:

1. Create an unbounded task flow for each menu in the final page hierarchy.

   For example, to achieve the page hierarchy illustrated in Figure 22–15, you create two unbounded task flows (Human Resources menu and Benefits menu).

   For more information about creating an unbounded task flow, see Section 18.2.1, "How to Create a Task Flow".

   > **Tip:** Prefix the name of the file for unbounded task flows that you create with `adfc-` to help you to identify the file as the source of an unbounded task flow, as opposed to a bounded task flow.

2. Add view activities that reference pages to each unbounded task flow. The pages referenced by the view activities correspond to the menu nodes in the menu.

For example, the Benefits menu contains one group node (benefits) and three item nodes (medical, dental and vision) so you add four view activities to the unbounded task flow for the Benefits menu, as illustrated in Figure 22–17.

Figure 22–17 View Activities on a Task Flow

Do not add view activities for menus that include other menus using shared nodes. For example, the Human Resources menu in Figure 22–15 has a tab called **Benefits** that references the Benefits menu using a shared node. The bounded task flow for the Benefits menu already includes a view activity for Benefits so there is no need to add a view activity to the bounded task flow for the Human Resources menu.

For more information about view activities, see Section 19.2, "Using View Activities.".

Note: If the page hierarchy includes pages referenced by a bounded task flow, add a task flow call activity to the unbounded task flow that calls the bounded task flow.

---

3. In the Application Navigator, right-click the file(s) for each of the unbounded task flows you created in step 1 and choose **Create ADF Menu Model**.
4. In the Create ADF Menu Model dialog, enter a file name for the `XMLMenuModel` metadata file and a directory to store it.
5. Click **OK**.

22.9.1.2 How to Create a Submenu with a Hierarchy of Group and Child Nodes

You open the `XMLMenuModel` metadata file you created and convert the item nodes that you want to make group nodes to group nodes. You then create a hierarchy where a group node is a parent to one or more item nodes.

Before you begin:

It may be helpful to have an understanding of what page hierarchy options you can configure. For more information, see Section 22.9, "Creating a Page Hierarchy Using Task Flows."

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Section 22.1.2, "Additional Functionality for Complex Task Flows."

To create a submenu with a hierarchy of group and item nodes:

1. In the Application Navigator, select and open the `XMLMenuModel` metadata file.

   An item node appears in the Structure window for each view activity in the unbounded task flow. By default, no hierarchy is defined.

2. Drag and drop the item nodes to become child nodes of the item node that you are going to convert to a group node.

   Each item node that you convert to a group node must have at least one child item node. For example, to create the menu hierarchy in Figure 22–15, you convert the item node for Benefits to a group node, you drag and drop the item nodes for Medical, Dental, and Vision so that they become child nodes of the Benefit item node.

3. In the Structure window, right-click the parent item node and choose **Convert To groupNode**.

4. Enter a new identifier or accept the default value in the **id** field of the groupNode Properties dialog that appears

   The identifier must be unique among all of the nodes in all of the `XMLMenuModel` metadata files. It is good practice to specify a value that identifies the node. For example, if you change the Benefits node to a group node, you can update its default ID, `itemNode_benefits`, to `groupNode_benefits`.

5. In the **idref** field of the groupNode Properties dialog, enter the ID of one of the other nodes in the menu, for example, `itemNode_Medical`.

   The value you enter can be an ID of a child item node that is a group node or an item node.

6. Enter or change the existing default value in the **label** field to match what you want to appear at runtime.

   For example, you might change `label_benefits` to `Benefits`.

7. Accept the rest of the default values in the fields and click **Finish**

   A Confirm Convert dialog asks if you want to delete the `action` and `focusViewID` attributes on the `groupNode` element. Group nodes do not use these attributes, always click **OK**

8. Click **OK**

22.9.1.3 How to Attach a Menu Hierarchy to Another Menu Hierarchy

You use a shared node element to link two menus together. For example, the Human Resources menu illustrated in the menu hierarchy in Figure 22–15 contains four submenus (Payroll, Time, Labor, and Benefits). The Benefits submenu is itself a menu with submenu entries. In the `XMLMenuModel` metadata file for the Human Resources menu, you convert the item node for the Benefits submenu to a shared node. You write an EL expression for an attribute (`ref`) of the newly-created shared node that references the `XMLMenuModel` metadata file for the Benefits menu.

Before you begin:

It may be helpful to have an understanding of what page hierarchy options you can configure. For more information, see Section 22.9, "Creating a Page Hierarchy Using Task Flows."

You may also find it helpful to read about additional functionality that you can add using other task flow features. For more information, see Section 22.1.2, "Additional Functionality for Complex Task Flows."

To attach a menu hierarchy to another hierarchy using a shared node:

1. In the Application Navigator, select and open the `XMLMenuModel` metadata file for the menu that is going to reference the other menu.

2. In the Structure window, select a node, right-click and select the appropriate menu options to insert a `sharedNode` element.
3. In the **ref** field of the Insert sharedNode dialog that appears, enter an EL expression to reference the `XMLMenuModel` metadata file for the other menu.
4. Click **OK**.

> **Note:** If your page hierarchy has more than one unbounded task flow, ensure that the file name for each additional unbounded task flow appears as a value for the `<metadata-resources>` element in the `adfc_config.xml` file. For more information, see Section 22.9.2, "What Happens When You Create a Page Hierarchy".

22.9.2 What Happens When You Create a Page Hierarchy

Changes occur in a number of different files when you create a page hierarchy.

Changes to the adfc-config.xml File

When you create a new unbounded task flow, JDeveloper automatically adds a reference in the `adfc-config.xml` file to the source file for the newly-created unbounded task flow. In Example 22–15, `adfc-unbounded_tflow.xml` is the name of the source file for a newly-created unbounded task flow.

Example 22–15 Unbounded task flow referenced by adfc-config.xml

```
<?xml version="1.0" encoding="windows-1252" ?>
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2"
             id="__1">
  <metadata-resource id="__2">/WEB-INF/adfc-unbounded_
tflow.xml</metadata-resource>
</adfc-config>
```

For more information about `adfc-config.xml`, see Section A.9, "adfc-config.xml".

At runtime, the Fusion web application loads the `adfc-config.xml` file when it first starts. The `adfc-config.xml` file can contain:

- ADF navigation metadata for an unbounded task flow
- ADF activity metadata for an unbounded task flow
- Managed bean definitions used by ADF activities

XMLMenuModel Metadata File

JDeveloper generates an `XMLMenuModel` metadata file with nodes for each of the view activities that you added to the unbounded task flow, as illustrated in Example 22–2.

Example 22–16 Example XMLMenuModel Metadata File

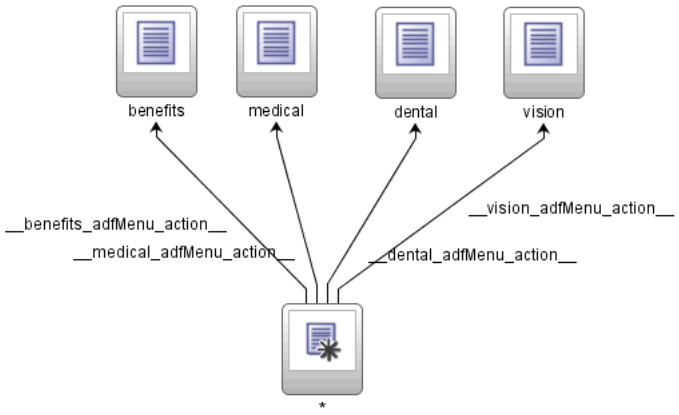
```
<?xml version="1.0" encoding="windows-1252" ?>
<menu xmlns="http://myfaces.apache.org/trinidad/menu">
  <groupNode id="groupNode_benenfits" label="Benefits" idref="itemNode_medical">
    <itemNode id="itemNode_medical" label="label_medical"
              action="adfMenu_medical" focusViewId="/medical"/>
    <itemNode id="itemNode_dental" label="label_dental" action="adfMenu_dental"
              focusViewId="/dental"/>
    <itemNode id="itemNode_vision" label="label_vision" action="adfMenu_vision"
              focusViewId="/vision"/>
```

```
  </groupNode>
</menu>
```

Diagram for an Unbounded Task Flow

JDeveloper updates the file for the unbounded task flow with the control flow rules and managed beans used to navigate the page hierarchy. Figure 22–18 shows the updated unbounded task flow in the diagrammer that corresponds unbounded task flow in Figure 22–17.

Figure 22–18 Updated Unbounded Task Flow



23

Using Dialogs in Your Application

This chapter describes how to create dialogs in your application. It describes how you can use ADF Controller and task flows to create dialogs or, alternatively, use the ADF Faces dialog framework.

This chapter includes the following sections:

- Section 23.1, "About Using Dialogs in Your Application"
- Section 23.2, "Running a Bounded Task Flow in a Modal Dialog"
- Section 23.3, "Using the ADF Faces Dialog Framework"

23.1 About Using Dialogs in Your Application

Use dialogs if you want to show information to end users in a secondary browser window external to the browser window that displays the end user's current page. For example, you want to display help information to end users to assist them with a task in the primary browser window or you want end users to select a value from a list of values. The help information example is a use case where a modeless dialog is appropriate. A *modeless dialog* allows end users work in both the primary window and the dialog at the same time. For the use case where you want an end user to select a value, a modal dialog is more appropriate. A *modal dialog* prevents an end user accessing the page that invoked the dialog until they execute an action requested by the dialog (for example, select a value).

Use the ADF Faces dialog framework if you want to configure modeless dialogs for your end users. If you plan to configure modal dialogs for your end users, configure an ADF Controller bounded task flow to invoke one or more dialogs.

23.1.1 Using Dialogs in Your Application Use Cases and Examples

Section 23.2, "Running a Bounded Task Flow in a Modal Dialog" describes how you can run a bounded task flow in a modal dialog to retrieve input from an end user, and return to a view activity that called the bounded task with the retrieved input. If your application does not use ADF Controller task flows or you want to use modeless dialogs, Section 23.3, "Using the ADF Faces Dialog Framework" describes how you can use the ADF Faces dialog framework to render one or more pages in a dialog.

23.1.2 Additional Functionality for Using Dialogs in Your Application

You may find it helpful to understand other ADF features before you configure or use dialogs in your application. Additionally, you may want to read about what you can do with the dialogs you configure. Following are links to other functionality that may be of interest.

- If your application uses ADF Controller task flows, you may find it helpful to understand more about the features that task flows offer. For more information, see Chapter 18, "Getting Started with ADF Task Flows."
- For more information about ADF Faces components, see the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
- You can also write custom code for your dialogs using the APIs provided by Oracle ADF. Make sure when you write custom code that you do not import packages that are marked internal, as in the following example:

  ```
  import oracle.adfinternal.controller.*;
  ```

  For information about the APIs that you can use to write custom code, see the following reference documents:

  - *Oracle Fusion Middleware Java API Reference for Oracle ADF Controller*
  - *Oracle Fusion Middleware Java API Reference for Oracle ADF Faces*

23.2 Running a Bounded Task Flow in a Modal Dialog

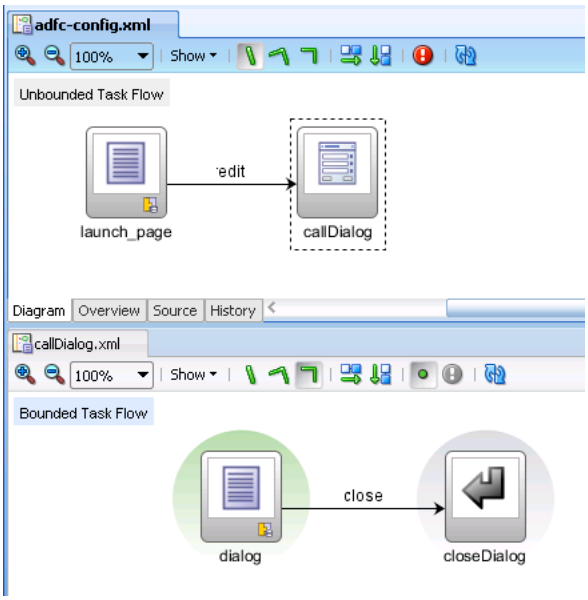
You can configure a bounded task flow to run in a modal dialog, retrieve input from an end user, and return to the view activity that called the bounded task flow with the retrieved input. The bounded task flow that you configure must reference pages, not page fragments.

Figure 23–1 shows an example of the configuration required. The task flow in `adfc-config.xml` contains a view activity (**launch_page**) from where the end user invokes the `callDialog.xml` task flow that renders a modal dialog. For more information, see Section 23.2.1, "How to Run a Bounded Task Flow in a Modal Dialog."

When the end user closes the dialog, control and any modified values return to the launch page in the calling task flow in `adfc-config.xml`. For more information, see Section 23.2.2, "How to Return a Value From a Modal Dialog."

In addition, you can configure the application to refresh part of the launch page after the modal dialog returns control. For more information, see Section 23.2.3, "How to Refresh a Page After a Modal Dialog Returns."

Figure 23–1 Task Flow Activities to Invoke a Modal Dialog



23.2.1 How to Run a Bounded Task Flow in a Modal Dialog

You add a view activity and a task flow call activity to an existing task flow. The view activity invokes a page where an end user can invoke an action that, in turn, invokes the bounded task flow to appear in a modal dialog.

Before you begin:

It may be helpful to have an understanding of how the attributes you configure affect the functionality of a bounded task flow in a modal dialog. For more information, see Section 23.2, "Running a Bounded Task Flow in a Modal Dialog."

You may also find it helpful to understand functionality that can be added using other ADF features. For more information, see Section 23.1.2, "Additional Functionality for Using Dialogs in Your Application."

To run a bounded task flow in a modal dialog box:

1. In the diagram editor for the existing task flow, double-click the view activity to open the associated page.
2. Select the command component that the end user clicks at runtime to invoke the bounded task flow as a modal dialog box (for example, a `commandButton` component).
3. In the Property Inspector, expand the **Common** section and set the `action` attribute to the control flow case to invoke the bounded task flow.

   For example, `edit` in Figure 23–1.
4. Set the `useWindow` attribute to `true` to invoke the bounded task flow in a popup dialog.
5. Return to the diagram editor for the existing task flow and select the task flow call activity.
6. In the Property Inspector, expand the **Behavior** section and set the `run-as-dialog` attribute to `true` to run the bounded task flow as a dialog.

The bounded task flow that the task flow activity calls can now behave as a dialog; it can be invoked by the command component you configured in steps 2 to 4 and can return a value to the view activity that renders the command component, as described in Section 23.2.2, "How to Return a Value From a Modal Dialog." Note that you need to make these other changes to run a bounded task flow in a modal dialog. Setting the `run-as-dialog` attribute to `true` is not sufficient.

7. For the `display-type` attribute, select `external-window` (the default value) if you want to render the dialog in an external browser window or `inline-popup` if you want to render the dialog in the same browser window.

23.2.2 How to Return a Value From a Modal Dialog

You can configure a bounded task flow that renders in a modal dialog to return a value to the view activity that invoked the bounded task flow when the end user dismisses the modal dialog. The returned value can, for example, be displayed in an input component on the page associated with the view activity.

You must configure the bounded task flow that is called by the task flow call activity to declare input parameters and return values. For more information, see Section 20.3, "Passing Parameters to a Bounded Task Flow".

You specify a method binding for a method with one argument (a return event) as the value for the `returnListener` attribute of the command component (for example, a `commandButton` component). The `returnListener` attribute sets this value in the input component on the page associated with the view activity. Specify a backing bean for the input component and set the input component's `partialTrigger` attribute to the ID of the command component.

You also need to specify:

- A return value definition on the called bounded task flow to indicate where to take the return value from upon exit of the called bounded task flow.
- Return values on the task flow call activity in the existing task flow to indicate where the existing task flow can find return values.

  For more information, see Section 20.5, "Specifying a Return Value from a Bounded Task Flow".

For more information about creating backing beans, input components, and command components, see the "Using Input Components and Defining Forms" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Before you begin:

Configure a bounded task flow to run in a modal dialog. For more information, see Section 23.2.1, "How to Run a Bounded Task Flow in a Modal Dialog".

It may be helpful to have an understanding of how the attributes you configure affect the functionality of bounded task flows in modal dialogs. For more information, see Section 23.2, "Running a Bounded Task Flow in a Modal Dialog."

You may also find it helpful to understand functionality that can be added using other ADF features. For more information, see Section 23.1.2, "Additional Functionality for Using Dialogs in Your Application."

To specify a return value:

1. In the diagram editor for the existing task flow, select the task flow call activity.

2. In the Property Inspector, expand the **Behavior** section and set the `run-as-dialog` attribute to `true`.
3. Set the `dialog-return-value` to the name of the return value definition you specified for the target bounded task flow.

   For information about how to specify the return value definition on a bounded task flow, see Section 20.5, "Specifying a Return Value from a Bounded Task Flow".
4. In the Application Navigator, double-click the page that launches the modal dialog.
5. Select the input component on the page and, in the Property Inspector, expand the **Behavior** section to specify an EL expression for the `partialTriggers` attribute.

   The EL expression you specify identifies the command component, accepts the return value of the command component, and specifies a backing bean. For example, enter an EL expression with syntax similar to the following:

   ```
   #{pageFlowScope.backingBean.gotoModalDialog}
   ```

   Where `gotoModalDialog` identifies the command component.
6. Select the command component and, in the Property Inspector, expand the **Behavior** section.
7. In the **Secondary Window**, enter an EL expression that references a return listener method in the page's backing bean as a value for the `returnListener` attribute.

   The return listener method you specify processes the return event that is generated when an end user dismisses the modal dialog. Enter an EL expression with syntax similar to the following:

   ```
   #{pageBean.listenerMethod}
   ```

23.2.3 How to Refresh a Page After a Modal Dialog Returns

You can configure the page from where end users invoke a modal dialog to refresh after end users close the modal dialog. You may want to configure this behavior if the modal dialog returns a value or end users edit existing data in the page through controls in the modal dialog.

You implement this functionality by configuring the command component that invokes the modal dialog to listen for a return event. When it receives the return event, it invokes a backing bean method that executes a partial page render event on your page. Example 23–1 shows a `commandButton` component that invokes a modal dialog and listens for a return event.

Example 23–1 ReturnListener Attribute Referencing a Backing Bean Method

```
<af:commandButton text="Edit"
   binding="#{backingBeanScope.backing_launch_page.cb1}"
   id="cb1" action="edit" useWindow="true"
   returnListener="#{backingBeanScope.backing_launch_page.backFromPopup}"/>
```

The `commandButton` component's `returnListener` attributes listens for the return event from invoking the modal dialog. When it receives the return event, it invokes a backing bean method similar to the following example:

Example 23–2 Backing Bean Method to Invoke a Partial Page Event

```
public void backFromPopup(ReturnEvent returnEvent) {
        AdfFacesContext adfFacesContext;
```

```
        adfFacesContext = AdfFacesContext.getCurrentInstance();
        adfFacesContext.addPartialTarget(this.getF1());
}
```

The backing bean method (`backFromPopup`) takes the return event as an argument and, in our example, invokes a partial page render event on a form in the page.

Before you begin:

It may be helpful to have an understanding of how you configure other options for modal dialogs that you invoke from a page associated with a view activity in a task flow. For more information, see Section 23.2, "Running a Bounded Task Flow in a Modal Dialog."

You may also find it helpful to understand other functionality that can be added using other task flow and dialog framework features. For more information, see Section 23.1.1, "Using Dialogs in Your Application Use Cases and Examples."

To refresh a page after a modal dialog returns:

1. In the diagram editor for the existing task flow, double-click the view activity that references the page where your end user invokes a command component to launch the modal dialog.
2. Select the command component and, in the **Property Inspector**, expand the **Behavior** section to enter, in the **ReturnListener** field under the **Secondary Window** label, the name of the backing bean method that you want JDeveloper to generate.
3. In the Application Navigator, expand **Application Sources** and then the package that contains the backing bean class where JDeveloper generated a method with the value that you entered in step 2.
4. Write a method body similar to Example 23–2 to invoke a partial page render event on your page.

23.2.4 What You May Need to Know About Dialogs in an Application that Uses Task Flows

If your Fusion web application renders modeless dialogs (non modal dialogs without task flows) and also uses ADF Controller features, such as task flows, you specify the `dialog:syntax` in the control flow rules of the application's `adfc-config.xml` file rather than the `faces-config.xml` file. Specify `dialog:syntax` in navigation rules within the `faces-config.xml` file if your Fusion web application does not use ADF Controller features.

Example 23–3 shows an example of what you can specify in the `adfc-config.xml` file.

Example 23–3 adfc-config.xml file with dialog:syntax

```
<?xml version="1.0" encoding="windows-1252" ?>
  <adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2" id="__
1">
    <view id="view1"
        <page>/view1.jspx</page>
    </view>
    <view id="dialog">
       <page>/dialog/untitled1.jspx</page>
   </view>
```

```
   <control-flow-rule>
        <from-activity-id>test</from-activity-id>
        <control-flow-case>
          <from-outcome>dialog:test</from-outcome
          <to-activity-id>dialog</to-activity-id>
         </control-flow-case>
</adfc-config>
```

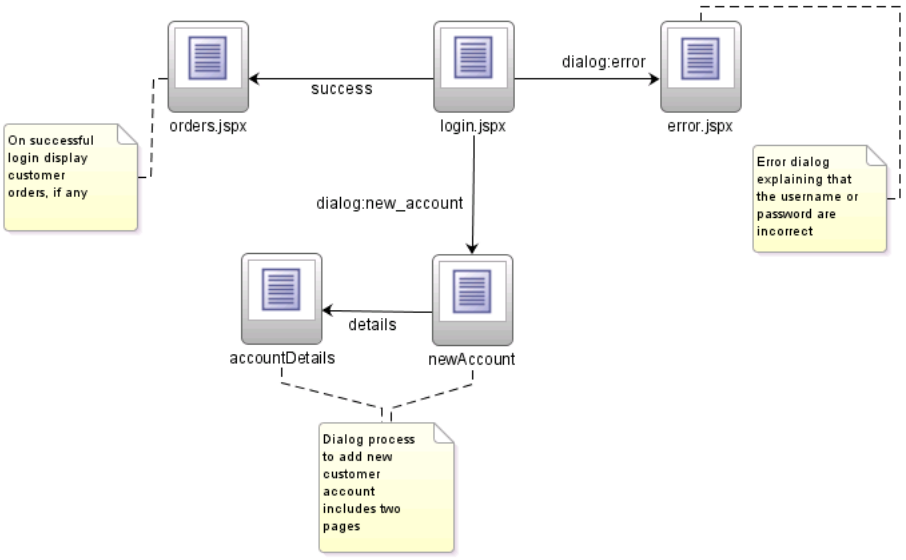
23.3 Using the ADF Faces Dialog Framework

You can use the ADF Faces dialog framework to create modal and modeless dialogs in an application that does not use the ADF Controller and task flows. The dialog framework enables you to display a page or series of pages in a new browser window instead of displaying it in the same window (using the same view ID) as the current page. There may also be cases where you want to use a series of inline dialogs, that is, dialogs that are part of the parent page, but that have a flow of their own, but that do not use a separate view ID. This is important for applications that do not support popups such as, for example, applications that run on client devices or that use the Active Data Service described in Chapter 45, "Using the Active Data Service." Ordinarily, you would need to use JavaScript to open the dialog and manage the process. With the dialog framework, ADF Faces has made it easy to open a new browser window as well as manage dialogs and processes without using JavaScript.

> **Note:** If your application uses the Fusion technology stack with the ADF Controller, then you should use task flows to create dialogs launched in a separate window, or multiple dialog processes. For more information, see Section 23.2, "Running a Bounded Task Flow in a Modal Dialog".

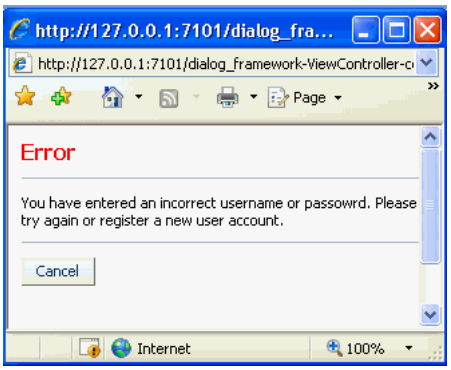
Consider a simple application that requires users to log in to see their orders. Figure 23–2 shows the page flow for the application, which consists of five pages - `login.jspx`, `orders.jspx`, `new_account.jspx`, `account_details.jspx`, and `error.jspx`.

Figure 23–2 Page Flow of an External Dialog Sample Application



When an existing user logs in successfully, the application displays the Orders page, which shows the user's orders, if any. When a user does not log in successfully, the Error page displays in a separate popup dialog window, as shown in Figure 23–3.

Figure 23–3 Error Page Popup

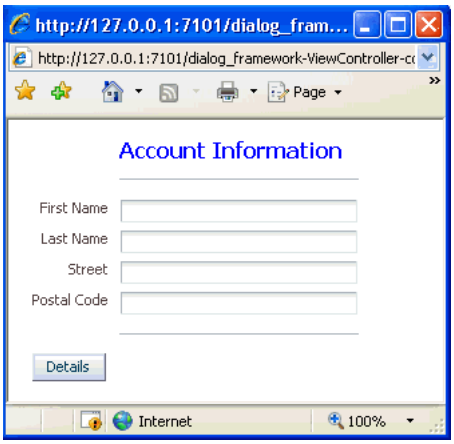


On the Error page there is a **Cancel** button. When the user clicks **Cancel**, the popup dialog closes and the application returns to the Login page and the original flow, as shown in Figure 23–4.

Figure 23–4 Login Page

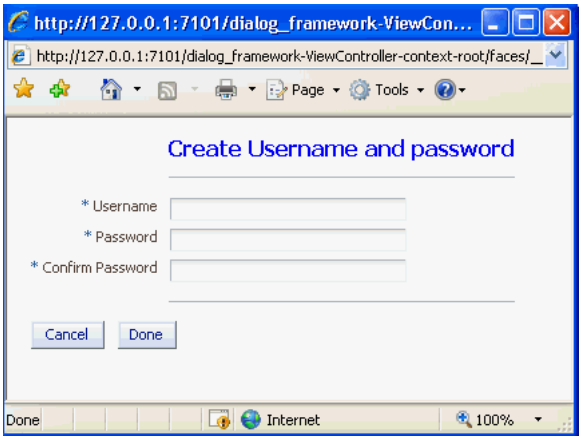
When a new user clicks the **New User** link on the Login page, the New Account page displays in a popup dialog in a new window, as shown in Figure 23–5.

Figure 23–5 New Account Page in a Separate Window



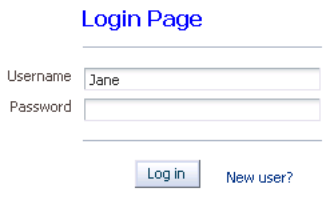
After entering information such as first name and last name, the user then clicks the **Details** button to display the Account Details page in the same popup dialog, as shown in Figure 23–6. In the Account Details page, the user enters other information and confirms a password for the new login account. There are two buttons on the Account Details page - **Cancel** and **Done**.

Figure 23–6 Account Details Page in a Popup Dialog



If the new user decides not to proceed with creating a new login account and clicks **Cancel**, the popup dialog closes and the application returns to the Login page. If the new user clicks **Done**, the popup dialog closes and the application returns to the Login page where the **Username** field is now populated with the user's first name, as shown in Figure 23–7. The new user can then proceed to enter the new password and log in successfully.

Figure 23–7 LogIn Page with Username Field Populated



> **Note:** The dialog framework should not be used to have more than one dialog open at a time, or to launch dialogs that have a life span outside of the life span of the base page.

To make it easy to support dialog page flows in your applications, ADF Faces has built in the dialog functionality to action components (such as `commandMenuItem` and `commandButton`). For ADF Faces to know whether or not to open a page in a new flow from an action component, the following conditions must exist:

- There must be a JSF navigation rule with an outcome that begins with `dialog:`.
- The command component's action outcome must begin with `dialog:`.
- The `useWindow` attribute on the command component must be `true`.

> **Note:** If the `useWindow` attribute is `false`, or if you configure the popup to be a separate window (and not inline) and the client device does not support popups, ADF Faces automatically shows the page in the current window instead of using a popup window; code changes are not required to facilitate this action.

The page that is displayed in a dialog is an ordinary JSF page, but for purposes of explaining how to implement external dialogs in this chapter, a page that is displayed in a popup dialog is called the *dialog* page, and a page from which the dialog is opened is called the *originating* page. A *dialog process* starts when the originating page opens a dialog (which can contain one dialog page or a series of dialog pages), and ends when the user dismisses the dialog and returns to the originating page.

The tasks for supporting a dialog page flow in an application are:

1. Define a JSF navigation rule for opening a dialog.
2. Create the JSF page from which a dialog is opened.
3. Create the dialog page and return a dialog value.
4. Optional: Pass a value into a dialog.
5. Handle the return value.

The tasks can be performed in any order.

23.3.1 How to Define a JSF Navigation Rule for Opening a Dialog

You manage the navigation into a dialog flow by defining a standard JSF navigation rule with a special `dialog:` outcome.

To define a navigation rule to open a dialog:

1. In the `adfc-config.xml` file, create a page flow for your originating page and dialog pages. For detailed procedures, see Section 18.4.1, "How to Add a Control Flow Rule to a Task Flow".
2. When creating navigation rules to the dialog pages, the outcome must begin with `dialog:`. For example, in the login sample application shown in Figure 23–2, the outcome from the Login page to the New Account dialog page is `dialog:newAccount`.

At runtime, the dialog navigation rules on their own simply show the specified pages in the originating page. But when used with command components with `dialog:` action outcomes and with `useWindow` attributes set to `true`, ADF Faces knows to open the pages in dialogs.

23.3.2 How to Create the JSF Page That Opens a Dialog

In the originating page, you need to use a command component to launch the dialog. The command component's action value needs to be the outcome to the dialog that is to be launched.

Before you begin:

It may be helpful to have an understanding of how the attributes you configure affect the functionality of the ADF Faces dialog framework. For more information, see Section 23.1, "About Using Dialogs in Your Application."

You may also find it helpful to understand functionality that can be added using other ADF features. For more information, see Section 23.1.2, "Additional Functionality for Using Dialogs in Your Application."

To create the JSF Page that opens a dialog

1. Create a JSF page.

   For more information, see the "Creating a View Page" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

2. Add a command component to the page.

   For more information about adding a command component to a page, see the "Using Buttons and Links for Navigation" section in *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework.*

   Note the following when setting the attributes on the command component:

   - **Action**: Set the action attribute to the outcome that navigates to the dialog, as created in Section 23.3.1, "How to Define a JSF Navigation Rule for Opening a Dialog."

     Tip: The action value can be either a static string or the return of a method on a managed bean.

     For example, the action attribute on the command component of the Login page is bound to a method that determines whether to navigate to the Orders page or to the Error dialog page, based on the returned outcome. If the method returns `dialog:error`, the error dialog opens. If the method returns success, the user navigates to the orders page.

   - **ActionListener**: As an alternative to setting the action attribute, configure the `actionListener` attribute to invoke the `launchDialog` method from an instance of the following class:

     `oracle.adf.view.rich.context.AdfFacesContext`

     For more information about the `launchDialog` method and the `AdfFacesContext` class, see the *Oracle Fusion Middleware Java API Reference for Oracle ADF Faces*.

   - **UseWindow**: Set to `true` to have the dialog open.

Tip: When set to false, ADF Faces shows the dialog page in the current window after preserving all of the state of the current page - you do not have to write any code to facilitate this.

- **WindowHeight** and **WindowWidth**: Set the desired size of the dialog window. These values will set the `contentWidth` and `contentHeight` attributes on the popup component for the dialog.

  Tip: While the user can change the values of these attributes at runtime, the values will not be retained once the user leaves the page unless you configure your application to use change persistence. For information about enabling and using change persistence, see the "Allowing User Customization on JSF Pages" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

- **PartialSubmit**: Set to true. This prevents the originating page from reloading (and hence being visible only momentarily) when the popup dialog is displayed.
- **WindowEmbedStyle**: Set to `inlineDocument` if you want the ensuing dialog to open in a popup that belongs to the originating page. Set to `window` if you want the ensuing dialog to open in a separate browser.
- **WindowModalityType**: Set to `applicationModal` if you want the dialog to be modal. Modal dialogs do not allow the user to return to the originating page until the dialog has been dismissed. Set to `modeless` if you want the user to be able to go back and forth between the originating page and the dialog.

When a command component is about to open a dialog, it delivers a `LaunchEvent` event. The `LaunchEvent` event stores information about the component that is responsible for opening a popup dialog, and the root of the component tree to display when the dialog process starts. A `LaunchEvent` can also pass a map of parameters into the dialog. For more information, see Section 23.3.5, "How to Pass a Value into a Dialog."

23.3.3 How to Create the Dialog Page and Return a Dialog Value

A dialog page is just like any other JSF page, with one exception. In a dialog page, you must provide a way to tell ADF Faces when the dialog process finishes, that is, when the user dismisses the dialog or series of dialogs.

For example, the New Account page and Account Details page belong in the same dialog process. A dialog process can have as many pages as you desire, you only need to notify the framework that the dialog process has ended once.

You do this declaratively using the `returnActionListener` tag as a child to the command component used to close the dialog. However, if you need to provide a return value or other action event processing, you can bind the `actionListener` attribute on the command component to a method that calls the `AdfFacesContext.returnFromDialog()` method. This method lets you send back a return value in the form of a `java.lang.Object` or a `java.util.Map` of parameters. You do not have to know where you are returning the value to - ADF Faces automatically takes care of it.

At runtime, the `AdfFacesContext.returnFromDialog()` method tells ADF Faces when the user dismisses the dialog. This method can be called whether the dialog

page is shown in a popup dialog or in the main window. If a popup dialog is used, ADF Faces automatically closes it.

Before you begin:

It may be helpful to have an understanding of how the attributes you configure affect the functionality of the ADF Faces dialog framework. For more information, see Section 23.1, "About Using Dialogs in Your Application."

You may also find it helpful to understand functionality that can be added using other ADF features. For more information, see Section 23.1.2, "Additional Functionality for Using Dialogs in Your Application."

To close a dialog window and optionally return a value:

1. To the dialog page, add a command component. If that component will be used to close the window, set the immediate attribute to `true`.

   If the button will be used to navigate to another page in the dialog process, configure the button as though it were standard navigation, and set the `useWindow` attribute to `false`, which will cause the next page to display in the same dialog window, preserving the state of the previous page.

2. If you need to end the dialog process and close the dialog, but do not need to return a value, in the Component Palette, from the Operations panel, in the Listeners group, drag a **Return Action Listener** and drop it as a child to the command component.

   The `returnActionListener` tag calls the `returnFromDialog` method on the `AdfFacesContext` object - no backing bean code is needed.

   No attributes are used with the `af:returnActionListener` tag. The `immediate` attribute on the `af:commandButton` component is set to `true`: if the user clicks **Cancel** without entering values in the required **Password** and **Confirm Password** fields, the default JSF `ActionListener` can execute during the Apply Request Values phase instead of the Invoke Application phase, thus bypassing input validation. For more information, see the "Using the JSF Lifecycle with ADF Faces" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

3. If you need to end the dialog process and do need to return a value, create a method on a managed bean that handles the `action` event and returns the needed values using the `returnFromDialog` method on the current instance of `AdfFacesContext`.

   Note: The `AdfFacesContext.returnFromDialog()` method returns `null`. This is all that is needed in the backing bean to handle the Cancel action event.

   For example, when the user clicks **Done** on the Account Details page, the process ends and returns the user input values. Example 23–4 shows the code for the event handler method to which **Done** button is bound. The method gets the customer information, then either creates a Faces message for an incorrect password, or sets the values on the new customer object and return that object.

Example 23–4 Action Listener Method for the Done Button in a Managed Bean

```
public void done(ActionEvent e)
{
```

```
  AdfFacesContext afContext = AdfFacesContext.getCurrentInstance();
  String firstname = afContext.getPageFlowScope().get("firstname").toString();
  String lastname = afContext.getPageFlowScope().get("lastname").toString();
  String street = afContext.getPageFlowScope().get("street").toString();
  String zipCode = afContext.getPageFlowScope().get("zipCode").toString();
  String country = afContext.getPageFlowScope().get("country").toString();
  String password = afContext.getPageFlowScope().get("password").toString();
  String confirmPassword =
   afContext.getPageFlowScope().get("confirmPassword").toString();
  if (!password.equals(confirmPassword))
  {
    FacesMessage fm = new FacesMessage();
    fm.setSummary("Confirm Password");
    fm.setDetail("You've entered an incorrect password. Please verify that you've
     entered a correct password!");
    FacesContext.getCurrentInstance().addMessage(null, fm);
  }
  else
  {
    //Get the return value
    Customer cst = new Customer();
    cst.setFirstName(firstname);
    cst.setLastName(lastname);
    cst.setStreet(street);
    cst.setPostalCode(zipCode);
    cst.setCountry(country);
    cst.setPassword(password);
    // And return it
    afContext.getCurrentInstance().returnFromDialog(cst, null);
  }
}
```

23.3.4 What Happens at Runtime: Raising the Return Event from the Dialog

When the dialog is dismissed, ADF Faces generates a return event (`ReturnEvent`). The `AdfFacesContext.returnFromDialog()` method sends a return value as a property of the return event. The return event is delivered to the return listener (`ReturnListener`) that is registered on the command component that opened the dialog (for example, the **New User** `commandLink` on the Login page). How you would handle the return value is described in Section 23.3.7, "How to Handle the Return Value."

23.3.5 How to Pass a Value into a Dialog

To pass a value into a dialog, you use a `LaunchListener` listener bound to a handler method for the `LaunchEvent`. You can use the `getDialogParameters()` method to add a parameter to a `Map` using a key-value pair.

Before you begin:

It may be helpful to have an understanding of how the attributes you configure affect the functionality of the ADF Faces dialog framework. For more information, see Section 23.1, "About Using Dialogs in Your Application."

You may also find it helpful to understand functionality that can be added using other ADF features. For more information, see Section 23.1.2, "Additional Functionality for Using Dialogs in Your Application."

To Pass a Value into a Dialog:

1. Create a handler method for the `LaunchEvent` that uses the `getDialogParameters` method to get the parameters from a dialog.

   For example, in the sample application, a new user can enter a name in the **Username** field on the Login page, and then click the **New User?** link. When the New Account dialog page displays in a popup dialog, the **First Name** input field is automatically populated with the name that was entered in the Login page. To accomplish this, you create a handler that uses the `getDialogParameters` method to put the value of the username field into the dialog, as shown in Example 23–5.

Example 23–5 LaunchEvent Listener Method for the New User Command Link in a Backing Bean

```
public void handleLaunch(LaunchEvent event)
{
  //Pass the current value of the field into the dialog
  Object usr = username;
  event.getDialogParameters().put("firstname", getUsername());
}
// Use by inputText value binding
private String username;
public String getUsername()
{
  return username;
}
public void setUsername(String username)
{
  this.username = username;
}
```

2. Bind the launchListener attribute of the command component used to navigate to the next page, to the handler method created in Step 1.

   Example 23–6 shows the code for the `commandLink` component, whose `launchListener` attribute is bound to the handler method.

Example 23–6 Input Field and New User Command Link on the Login Page

```
<af:inputText label="Username" value="#{backing_login.username}"/>
<af:commandLink id="cmdLink" text="New User?"
                action="dialog:newAccount"
                useWindow="true" partialSubmit="true"
                launchListener="#{backing_login.handleLaunch}"
                returnListener="#{backing_login.handleReturn}"
                windowHeight="200" windowWidth="500" />
```

3. On the resulting page, use the `pageFlowScope` object to retrieve the key and value via a special EL expression in the format `#{pageFlowScope.someKey}`, as shown in Example 23–7

Example 23–7 Input Field on the New Account Page

```
<af:inputText label="First name" value="#{pageFlowScope.firstname}"/>
```

23.3.6 What Happens at Runtime: Handling the LaunchEvent

In ADF Faces, a process always gets a copy of all the values that are in the `pageFlowScope` of the page from which a dialog is launched. When the `getDialogParameters()` method has added parameters to a `Map`, those parameters also become available in `pageFlowScope`, and any page in the dialog process can get the values out of `pageFlowScope` by referring to the `pageFlowScope` objects via EL expressions.

Note: Unlike `sessionScope`, `pageFlowScope` values are visible only in the current page flow or process. If the user opens a new window and starts navigating, that series of windows has its own process; values stored in each window remain independent. Clicking on the browser's Back button automatically resets `pageFlowScope` to its original state. When you return from a process the `pageFlowScope` is back to the way it was before the process started. To pass values out of a process you would use `AdfFacesContext.returnFromDialog()`, `sessionScope` or `applicationScope`.

23.3.7 How to Handle the Return Value

To handle a return value once the dialog is dismissed, you define a return listener on the command component that launched the dialog. For example, in the sample application, once a new user enters information, that information needs to be handled once the dialog process is complete.

Before you begin:

It may be helpful to have an understanding of how the attributes you configure affect the functionality of the ADF Faces dialog framework. For more information, see Section 23.1, "About Using Dialogs in Your Application."

You may also find it helpful to understand functionality that can be added using other ADF features. For more information, see Section 23.1.2, "Additional Functionality for Using Dialogs in Your Application."

To handle the return value:

1. Create a handler method for the `returnEvent`. You use the `getReturnValue()` method to retrieve the return value, because the return value is automatically added as a property of the `ReturnEvent`.

   Example 23–8 shows the code for the return listener method that handles the return value.

Example 23–8 Return Listener Method for the New User Link in a Backing Bean

```
public void handleReturn(ReturnEvent event)
{
  if (event.getReturnValue() != null)
  {
    Customer cst;
    String name;
    String psw;
    cst = (Customer)event.getReturnValue();
    name = cst.getFirstName();
    psw = cst.getPassword();
    CustomerList.getCustomers().add(cst);
```

```
    inputText1.setSubmittedValue(null);
    inputText1.setValue(name);
    inputText2.setSubmittedValue(null);
    inputText2.setValue(psw);
  }
}
```

2. Bind the `returnListener` attribute on the command component that launched the dialog to the `returnEvent` handler method created in Step 1.

23.3.8 What Happens at Runtime: Handling the ReturnEvent on the Launching Component

At runtime in the sample application, when ADF Faces delivers a `ReturnEvent` to the `ReturnListener` registered on the `commandLink` component, the `handleReturn()` method is called and the return value is processed accordingly. The new user is added to a customer list, and as a convenience to the user any previously submitted values in the Login page are cleared and the input fields are populated with the new information.

Part V

Creating a Databound Web User Interface

Part IV contains the following chapters:

- Chapter 24, "Getting Started with Your Web Interface"
- Chapter 25, "Understanding the Fusion Page Lifecycle"
- Chapter 26, "Creating a Basic Databound Page"
- Chapter 27, "Creating ADF Databound Tables"
- Chapter 28, "Command Components to Invoke Functionality in the View Layer"
- Chapter 29, "Displaying Master-Detail Data"
- Chapter 30, "Creating Databound Selection Lists and Shuttles"
- Chapter 31, "Creating ADF Databound Search Forms"
- Chapter 32, "Using More Complex Databound ADF Faces Components"
- Chapter 33, "Creating Databound ADF Data Visualization Components"
- Chapter 34, "Using Contextual Events"

24

Getting Started with Your Web Interface

This chapter describes the steps you may need to take before using the Data Controls panel to create databound UI components on JSF pages.

This chapter includes the following sections:

- Section 24.1, "About Developing a Web Application with ADF Faces"
- Section 24.2, "Using Page Templates"
- Section 24.3, "Creating a Web Page"
- Section 24.4, "Using a Managed Bean in a Fusion Web Application"

24.1 About Developing a Web Application with ADF Faces

Most of what you need to know to get started with your web interface is covered in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*. However, using the ADF Model layer for data binding instead of JSF managed beans provides additional functionality, such as the ability to declaratively bind components to your business services. For more information on what ADF Model can provide, see Section 1.2.2, "ADF Model Layer." This chapter provides a high-level overview of the web interface development process as detailed in the Faces guide, and also provides information about the additional functionality available when you use ADF Model data binding.

Following the development process outlined in Chapter 1, "Introduction to Building Fusion Web Applications with Oracle ADF", developing a web application with ADF Faces and using ADF Model for data binding involves the following steps:

- Creating ADF Faces templates for your pages (optional)
- Creating the individual pages and page fragments for regions to be used within a page
- Creating any needed managed beans

Additionally, the lifecycle of a Fusion web application is different from that of a standard JSF or ADF Faces application. For more information about how the lifecycle works, see Chapter 25, "Understanding the Fusion Page Lifecycle."

Managed beans are Java classes that give you the flexibility to add UI level code to your pages and task flows. You can use them for functions such as front end data manipulation and event handling.

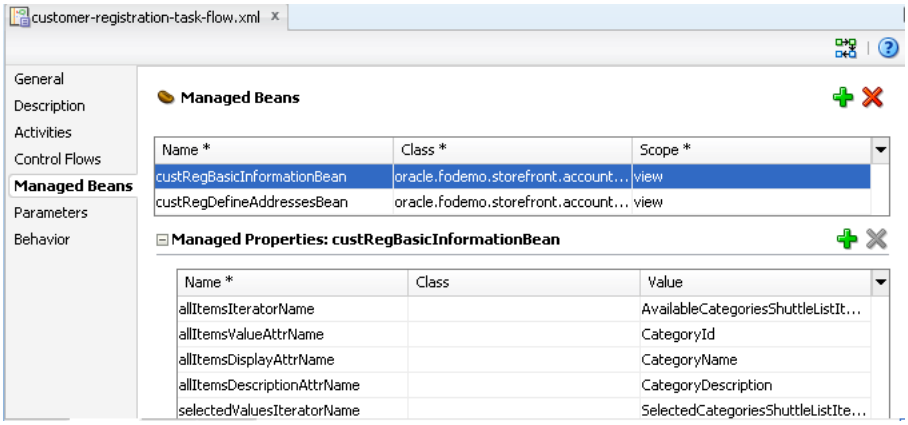
ADF Faces page templates provides structure, consistency, and reusability for building web pages. You can create a page template and apply it to several pages for a

consistent look and feel. It saves you time and effort because you do not need to layout the same elements each time you create a page.

24.1.1 Page Template and Managed Beans Use Cases and Examples

In the `customer-registration-task-flow`, the `CustRegBasicInformationBean` is used for code to support a shuttle component to shuttle the customers Categories of Interest from the available list to the I am Interested in list. The managed bean contains the code necessary to populate the available list and the selected list. Figure 24–1 shows the managed beans used in the `customer-registration-task-flow`.

Figure 24–1 Managed Beans in the customer-registration-task-flow



24.1.2 Additional Functionality for Page Templates and Managed Beans

You may find it helpful to understand other ADF features before you configure or use the ADF Model layer. Additionally, you may want to read about what you can do with your model layer configurations. Following are links to other functionality that may be of interest.

- For more information about using page templates, see Chapter 13, "Using ADF Model in a Fusion Web Application," and the "How to Create a Page Template" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
- For more information about managed beans in a standard JSF application, see the Java EE 5 tutorial on Sun's web site (`http://java.sun.com`). For more information about managed beans and scope values, see Section 25.3, "About Object Scope Lifecycles."

24.2 Using Page Templates

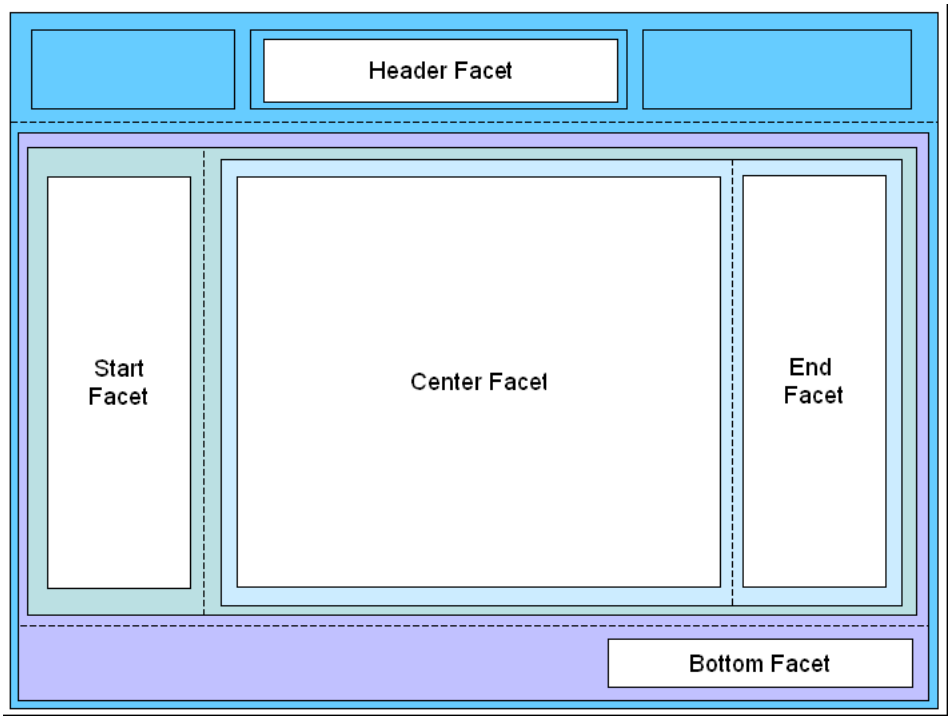
As you design the flow of your application, you can begin to think about the design of your pages. To ensure consistency throughout your application, you use ADF page templates. These page templates provide structure and consistency for other developers building web pages. Page templates typically contain static areas which cannot be changed when they are used, and dynamic areas, where developers can place content specific to the page they are building.

For example, the StoreFront module of the Fusion Order Demo application contains a page template that provides a top area for branding and navigation, a bottom area for copyright information, and a center area for the main content of the page. Page developers do not need to do anything to the branding and copyright information when they use the template. They need only to develop the main content area.

In addition to using ADF Faces components to build a page template, you can add attributes to the template definition. These attributes provide placeholders for specific information that the page developer needs to provide when using the page template. For example, the page template in the StoreFront module application contains an attribute for a welcome message. When page developers use this page template, they can add a different welcome message for each page.

You can also add facet references to the page template. These references act as placeholders for content on the page. Figure 24–2 shows a rendition of how the `StoreFrontTemplate` template used in the StoreFront module application uses facet references.

Figure 24–2 Facets in the StoreFrontTemplate



In this page template, facet references are used inside four different `panelSplitter` components. When the home page was created using this page template, the navigational links were placed in the Header facet and the accordion panels that hold the navigation trees and search panels were placed in the Start facet. The cart summary was placed in the End facet, and the main portion of the page was placed in the Center facet. The copyright information was placed in the Bottom facet.

When you choose to add databound components to a page template, an associated page definition file and the other metadata files that support ADF Model layer data binding are created. Each component is bound in the same fashion as for standard JSF pages, as described in Chapter 13, "Using ADF Model in a Fusion Web Application." You can also create model parameters for the page template. The values for these parameter can then be set at runtime by the calling page.

For example, if you wanted a list of products to appear on each page that uses the page template, you could drag and drop the `ProductName` attribute of the `Products` collection as a list. Or, if you wanted the pages to display the currently selected product ID, you could create a model parameter for the page template that would evaluate to that product's ID.

> **Note:** Page templates are primarily a project artifact. While they can be reused between projects and applications, they are not fully self-contained and will always have some dependencies to external resources, for example, any ADF data binding, `Strings` from a message bundle, images, and managed beans.

If a page template does not contain databound components, it can be referenced dynamically by the calling page using an EL expression. That is, the page template to be used can be determined at runtime. For instance, a page may use templateA or templateB based on user selection. When you add a page template to a page, an `af:pageTemplate` tag is added to the page. The `af:pageTemplate` tag includes a `viewId` attribute that specifies the page template the page will use. You can set `viewId` with an EL expression to a managed bean method that returns the page template `Id`, as shown in Example 24–1.

Example 24–1 Page with Dynamic Page Template (not Databound Page Template Only)

```
<af:pageTemplate
    id="pt1"
    viewId="#{myBean.templateViewId}"
```

If the page template has databound components, setting the `viewId` with an EL expression is not enough. Because databound components require access to the binding container, you must specify the page template as well as its associated binding container.

For databound page templates, you use the `pageTemplateModel` to manage both the page template `Id` and the associated binding container. In the JSF page, instead of using the `viewId` attribute, you set the `value` attribute to the `pageTemplateModel`. You must also modify the page executable section of the calling page's page definition file and create a managed bean with methods to process the page template `Id`s. For detailed instructions, see Section 24.2.3, "How to Add a Databound Page Template to a Page Dynamically."

24.2.1 How to Use ADF Data Binding in ADF Page Templates

Creating a page template for use in an application that uses ADF Business Components and ADF Model layer data binding is the same as creating a standard ADF Faces page template, as documented in the "Using Page Templates" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*. Once you create the template, you can drag and drop items from the Data Controls panel. JDeveloper automatically adds the page definition file when you drag and drop items from this panel.

The Create JSF Page Template wizard also allows you to create model parameters for use by the template.

Before you begin:

It may help to understand the options that are available to you when you create a basic page template. For more information, see Section 24.2, "Using Page Templates."

You may also find it useful to understand additional functionality that can be used with page templates. For more information, see Section 24.1.2, "Additional Functionality for Page Templates and Managed Beans."

You will need to complete this task:

Create a page template that uses ADF Business Components and ADF Model layer databinding.

To add model parameters to a template:

1. Create a page template following the instructions in the "How to Create a Page Template" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*. However, do not complete the dialog.
2. In the Create JSF Page Template dialog, select **Create Associated ADFm Page Definition**.

   Note: You only need to select this checkbox if you wish to add parameters. JDeveloper automatically adds the page definition file when you drag and drop items from the Data Controls panel.

3. Click the **Model Parameters** tab.
4. Click the **Add** icon.
5. Enter the following for the parameter:
   - **Id**: Enter a name for the parameter.
   - **Value**: Enter a value or an EL expression that can resolve to a value. If needed, click the Invoke Expression Builder (...) button to open the Expression Builder. You can use this to build the expression. For more information about EL expressions and the EL expression builder, see Chapter 13.8, "Creating ADF Data Binding EL Expressions."
   - **Option**: Select the option that determines how the parameter value will be specified.
     - **optional**: The binding definition's value is used only if the parameter is not specifically set by the caller. This is the default.
     - **final**: The binding definition has the expression to access the value that should be used for this parameter.
     - **mandatory**: The parameter value has to be set by the caller.
   - **Read Only**: Select if the parameter's value is to be read-only and should not be overwritten
6. Create more parameters as needed. Use the order buttons to arrange the parameters into the order in which you want them to be evaluated.

You can now use the Data Controls panel to add databound UI components to the page, as you would for a standard JSF page, as described in the remaining chapters in this part of the book.

> **Note:** If your template contains any method actions bound to a method iterator, you cannot change the value of the `refresh` attribute on the iterator to anything other than `Default`. If set to anything other than `Default`, the method will not execute.

24.2.2 What Happens When You Use ADF Model Layer Bindings on a Page Template

When you add ADF databound components to a template or create model parameters for a template, a page definition file is created for the template, and any model parameters are added to that file.

> **Note:** This section describes what happens for a statically assigned page template. For information about dynamic templates, see Section 24.2.3, "How to Add a Databound Page Template to a Page Dynamically."

Example 24–2 shows what the page definition file for a template for which you created a `productId` model parameter might look like.

Example 24–2 Model Parameters in a Page Definition for a Template

```
<parameters>
  <parameter id="productID" readonly="true"
             value="#{bindings.productId.inputValue}"/>
</parameters>
<executables/>
<bindings/>
```

Parameter binding objects declare the parameters that the page evaluates at the beginning of a request. For more information about binding objects and the ADF lifecycle, see Chapter 13, "Using ADF Model in a Fusion Web Application." However, since a template itself is never executed, the page that uses the template (the *calling* page) must access the binding objects created for the template (including parameters or any other type of binding objects created by dragging and dropping objects from the Data Controls panel onto the template).

In order to access the template's binding objects, the page definition file for the calling page must instantiate the binding container represented by the template's page definition file. As a result, a reference to the template's page definition is inserted as an executable into the calling page's page definition file, as shown in Example 24–3.

Example 24–3 Reference to Template's Page Definition as an Executable

```
<executables>
  <page path="oracle.foddemo.storefront.pageDefs.templates_MyTemplatePageDef"
        id="pageTemplateBinding"/>
</executables>
```

In this example, the calling page was built using the `MyTemplate` template. Because the page definition file for the `MyTemplate` template appears as an executable for the calling page, when the calling page's binding container is instantiated, it will in turn instantiate the `MyTemplatePageDef`'s binding container, thus allowing access to the parameter values or any other databound values.

Because there is an ID for this reference (in this case, `pageTemplateBinding`), the calling page can have components that are able to access values from the template.

When you create a JSF page from a template, instead of you having to repeat the code contained within the template, you can use the `af:pageTemplate` tag on the calling page. This tag contains the path to the template JSF page.

Additionally, when the template contains any ADF data binding, the value of that tag is the ID set for the calling page's reference to the template's page definition, as shown in Example 24–4. This binding allows the component access to the binding values from the template.

Example 24–4 Page Template Page Definition Reference

```
<af:pageTemplate viewId="/MyTemplate.jspx"
                 value="#{bindings.pageTemplateBinding}".../>
```

24.2.3 How to Add a Databound Page Template to a Page Dynamically

You can dynamically add a page template without databound components by using an EL expression to select the page template. For more information on how to do this, see Section 24.2, "Using Page Templates."

You can also statically add a page template with databound components. For more information on how to do this, see Section 24.2.1, "How to Use ADF Data Binding in ADF Page Templates."

This section describes how to dynamically add a page template with databound components to a page. For more information, see Section 24.2, "Using Page Templates."

You use the `pageTemplateModel` to dynamically manage a page template and its binding container. You use an EL expression in the page definition file to set the page template `Id`. You create managed bean methods to return the page template `Id`.

Before you begin:

It may help to understand the options that are available to you when you create a basic page template. For more information, see Section 24.2, "Using Page Templates."

You may also find it useful to understand additional functionality that can be used with page templates. For more information, see Section 24.1.2, "Additional Functionality for Page Templates and Managed Beans."

You will need to complete this task:

Create a page template that uses ADF Business Components and ADF Model layer databinding.

To add a databound page template to a page dynamically:

1. Add the page template to the page as described in Section 24.2.1, "How to Use ADF Data Binding in ADF Page Templates."
2. In the JSF page source editor, remove the `viewId` attribute and change the `value` attribute to the `pageTemplateModel`. You do not need to create a `pageTemplateModel` explicitly. You can use the `pageTemplateModel` from the corresponding page definition file's page executable binding. For example, for a page executable binding called `pageTemplate1`, you would add the following line under the `af:pageTemplate` tag:

   ```
   value="#{bindings.pageTemplate1.templateModel}"/>
   ```

3. In the page definition file `<executable>` section, make the following changes to the `<page>` section:

- Remove the `path` attribute. It is no longer needed. The `pageTemplateModel` manages databound components' access to the binding container.
- Change the `id` attribute to the page executable binding. In this example, it is `pageTemplate1`.
- Add a `Refresh` attribute and set it to `ifNeeded`.
- Add a `viewId` attribute and set it to an EL expression with a managed bean method that returns the current page template `Id`.

For example, for a page executable binding of `pageTemplate1`, the `id` attribute would also be `pageTemplate1`:

```
<executables>
  <page id="pageTemplate1"
        viewId="#{myBean.templateViewId}"
        Refresh="ifNeeded"/>
  ...
</executables>
```

4. Create a `pageFlowScope` managed bean with a method that returns the current page template `Id`.

   The managed bean code should be similar to that of Example 24–5. In this example, `gettemplateViewId()` obtains the user's page template selection and returns the page template `Id`. `setMDTemplateViewId()` sets the page template to be `MDPageTemplate` and `setPopupTemplateViewId()` sets the page template to be `PopupPageTemplate`.

Example 24–5 Managed Bean Code to Process Page Templates Dynamically

```
public class myClass {
    final private String MDPageTemplate = "/MDPageTemplate.jspx";
    final private String PopupPageTemplate = "/PopupPageTemplate.jspx";
    private String templateViewId;

    public myClass() {
        super();
        templateViewId = MDPageTemplate;
    }
    public String gettemplateViewId() {
        return templateViewId;
    }
    public void setMDTemplateViewId(ActionEvent ae) {
        templateViewId = MDPageTemplate;
    }
    public void setPopupTemplateViewId(ActionEvent ae) {
        templateViewId = PopupPageTemplate;
    }
}
```

24.2.4 What Happens at Runtime: How Pages Use Templates

When a page is created using a template that contains ADF data binding, the following happens:

1. The calling page follows the standard JSF/ADF lifecycle, as documented in Chapter 25, "Understanding the Fusion Page Lifecycle." As the page enters the Restore View phase, the URL for the calling page is sent to the binding context, which finds the corresponding page definition file.

2. During the Initialize Context phase, the binding container for the calling page is created based on the page definition file.
3. During the Prepare Model phase, the page template executable is refreshed. At this point, the binding container for the template is created based on the template's page definition file, and added to the binding context.
4. The lifecycle continues, with UI components and bindings from both the page and the template being processed.

24.3 Creating a Web Page

Creating a web page for an application that uses ADF Model layer data binding is no different than described in the "Creating a JSF Page" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*. You can create pages either by double-clicking a view activity in a task flow or by using the New Gallery. When creating the page (or dropping a view activity onto a task flow), you can choose to create the page as a JSF JSP or as a JSF JSP fragment. JSF fragments provide a simple way to create reusable page content in a project, and are what you use when you wish to use task flows as regions on a page. When you modify a JSF page fragment, the JSF pages that consume the page fragment are automatically updated.

Note: Although JDeveloper supports XHTML files to be used in applications that use Facelets, the `faces-config.xml` and `adfc-config.xml` diagrammers do not support XHTML. In order to add navigation to these files, you have to manually edit the code by clicking the **Source** tab.

When you begin adding content to your page, you typically use the Component Palette and Data Controls panel of JDeveloper. The Component Palette contains all the ADF Faces components needed to declaratively design your page. Once you have the basic layout components placed on the page, you can then drag and drop items from the Data Controls panel to create ADF Model databound UI components. The remaining chapters in this part of the book explain in detail the different types of databound components and pages you can create using ADF Model data binding.

24.4 Using a Managed Bean in a Fusion Web Application

Managed beans are Java classes that you register with the application using various configuration files. When the JSF application starts up, it parses these configuration files, and the beans listed within them are made available. The managed beans can be referenced in an EL expression, allowing access to the beans' properties and methods. Whenever a managed bean is referenced for the first time and it does not already exist, the Managed Bean Creation Facility instantiates the bean by calling the default constructor method on it. If any properties are also declared, they are populated with the declared default values.

Often, managed beans handle events or some manipulation of data that is best handled at the front end. For a more complete description of how managed beans are used in a standard JSF application, see the Java EE 5 tutorial on Sun's web site (`http://java.sun.com`).

Best Practice: Use managed beans to store logic that is related to the UI rendering only. All application data and processing should be handled by logic in the business layer of the application. Similar to how you store data-related logic in the database using PL/SQL rather than a Java class, the rule of thumb in a Fusion web application is to store business-related logic in the middle tier. This way, you can expose this logic as business service methods, which can then become accessible to the ADF Model layer and be available for data binding.

In an application that uses ADF data binding and ADF task flows, managed beans are registered in different configuration files from those used for a standard JSF application. In a standard JSF application, managed beans are registered in the `faces-config.xml` configuration file. In a Fusion web application, managed beans can be registered in the `faces-config.xml` file, the `adfc-config.xml` file, or a task flow definition file. Which configuration file you use to register a managed bean depends on what will need to access that bean, whether or not it needs to be customized at runtime, what the bean's scope is, and in what order all the beans in the application need to be instantiated. Table 24–1 describes how registering a bean in each type of configuration file affects the bean.

Note: Registering managed beans within the `faces-config.xml` file is not recommended in a Fusion web application.

Managed beans accessed within the task flow definition must be registered in that task flow's definition file.

Table 24–1 Effects of Managed Bean Configuration Placement

Managed Bean Placement	Effect
`adfc-config.xml`	▪ Managed beans can be of any scope. However, any backing beans for page fragments or declarative components should use `BackingBean` scope. For more information regarding scope, see Section 25.3, "About Object Scope Lifecycles." ▪ When executing within an unbounded task flow, `faces-config.xml` will be checked for managed bean definitions before the `adfc-config.xml` file. ▪ Lookup precedence is enforced per scope. Request-scoped managed beans take precedence over session-scoped managed beans. Therefore, a request-scoped managed bean named `foo` in the `adfc-config.xml` file will take precedence over a session-scoped managed bean named `foo` in the current task flow definition file. ▪ Already instantiated beans take precedence over new instances being instantiated. Therefore, an existing session-scoped managed bean named `foo` will always take precedence over a request-scoped bean named `foo` defined in the current task flow definition file.

Table 24–1 (Cont.) Effects of Managed Bean Configuration Placement

Managed Bean Placement	Effect
Task flow definition file	■ Managed bean can be of any scope. However, managed beans of `pageFlow` scope or `view` scope that are to be accessed within the task flow definition must be defined within the task flow definition file. Any backing beans for page fragments in a task flow should use `BackingBean` scope. ■ Managed bean definitions within task flow definition files will be visible only to activities executing within the same task flow. ■ When executing within a bounded task flow, `faces-config.xml` will be checked for managed bean definitions before the currently executing task flow definition. If no match is found in either location, `adfc-config.xml` and other bootstrap configuration files will be consulted. However, this lookup in other `adfc-config.xml` and bootstrap configuration files will only occur for session- or application-scoped managed beans. ■ Lookup precedence is enforced per scope. Request-scoped managed beans take precedence over session-scoped managed beans. Therefore, a request-scoped managed bean named `foo` in the `adfc-config.xml` file will take precedence over a session-scoped managed bean named `foo` in the current task flow definition file. ■ Already instantiated beans take precedence over new instances being instantiated. Therefore, an existing session-scoped managed bean named `foo` will always take precedence over a request-scoped bean named `foo` registered in the current task flow definition file. ■ Customizations are allowed.
`faces-config.xml`	■ Managed beans can be of any scope other than `pageFlow` scope or `view` scope. ■ When searching for any managed bean, the `faces-config.xml` file is always consulted first. Other configuration files will be searched only if a match is not found. Therefore, beans registered in the `faces-config.xml` file will always win any naming conflict resolution. ■ No customizations can be made.

As a general rule for Fusion web applications, a bean that may be used in more than one page or task flow, or one that is used by pages within the main unbounded task flow (`adfc-config`), should be registered in the `adfc-config.xml` configuration file. A managed bean that will be used only by a specific task flow should be registered in that task flow's definition file. There should be no beans registered in the `faces-config.xml` file.

> **Note:** If you create managed beans from dialogs within JDeveloper, the bean is registered in the `adfc-config.xml` file, if it exists.

For example, in the StoreFront module, the `myOrdersBean` managed bean is used by the `myOrders.jspx` page to handle the case where a user decides to cancel editing an order, and the edits have already been committed to the model but have not yet been

persisted to the database. Because this bean is used by a page within the `adfc-config` unbounded task flow, it is registered in the `adfc-config.xml` file. The `custRegBasicInformationBean` is a managed bean used by the `basicInformation` JSF fragment to handle the selections in the shuttle component on that page. Because it is used solely within the `customer-registration` task flow, it is registered in the `customer-registration-task-flow` definition file.

This section describes how to create a managed bean for use within a task flow (either the default `adfc-config` flow or any bounded task flow). For more information regarding managed beans and how they are used as backing beans for JSF pages, see the "Creating and Using Managed Beans" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

24.4.1 How to Use a Managed Bean to Store Information

Within the editors for a task flow definition, you can create a managed bean and register it with the JSF application at the same time.

Before you begin:

It may help to understand the options that are available to you when you create a managed bean. For more information, see Section 24.4, "Using a Managed Bean in a Fusion Web Application."

You may also find it useful to understand additional functionality that can be used with managed beans. For more information, see Section 24.1.2, "Additional Functionality for Page Templates and Managed Beans."

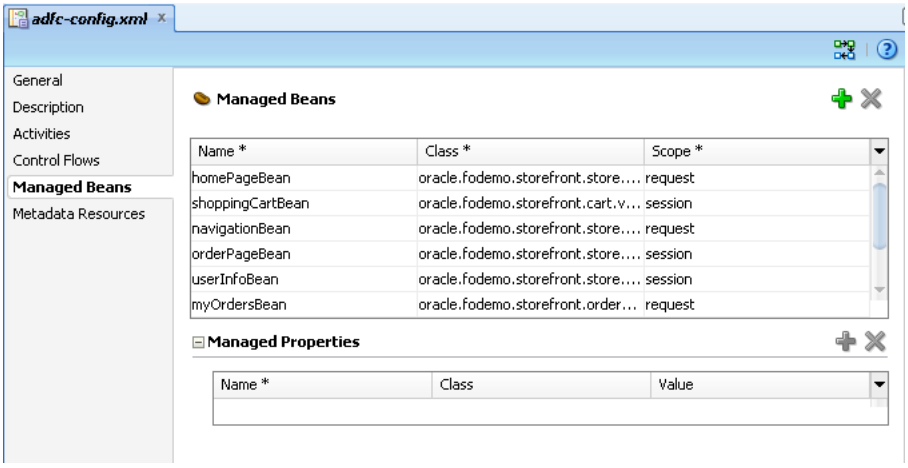
You will need to complete this task:

Create the configuration file (if it doesn't already exist) that you want the managed bean to be associated with. This can be `faces-config.xml`, `adfc-config.xml`, or a bounded task flow definition file.

To create a managed bean for a task flow:

1. In the Application Navigator, double-click either the `adfc-config.xml` file or any other task flow definition file.
2. At the bottom of the window, click the **Overview** tab.
3. In the overview editor, click the **Managed Beans** navigation tab. Figure 24–3 shows the editor for the `adfc-config.xml` file.

Figure 24–3 Managed Beans in the adfc-config.xml File



4. Click the **Add** icon to add a row to the **Managed Beans** table.
5. In the fields, enter the following:
   - **managed-bean-name**: A name for the bean.
   - **managed-bean-class**: If the corresponding class has already been created for the bean, use the browse (...) button for the **managed-bean-class** field to search for and select the class. If a class does not exist, enter the name you'd like to use. Be sure to include any package names as well. You can then use the drop-down menu to choose **Generate Class**, and the Java file will be created for you.
   - **managed-bean-scope**: The bean's scope. For more information about the different object scopes, see Section 25.3, "About Object Scope Lifecycles."

Note: When determining what scope to register a managed bean with or to store a value in, keep the following in mind:

- Always try to use the narrowest scope possible.
- If your managed bean takes part in component binding by accepting and returning component instances (that is, if UI components on the page use the binding attribute to bind to component properties on the bean), then the managed bean must be stored in `BackingBean` scope. If it can't be stored in one of those scopes (for example, if it needs to be stored in `sessionScope` for high availability reasons), then instead of using component binding, you need to use the ComponentReference API. For more information, see the "What You May Need to Know About Component Bindings and Managed Beans" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*
- Use the `sessionScope` scope only for information that is relevant to the whole session, such as user or context information. Avoid using the `sessionScope` scope to pass values from one page to another.
- You can also set the scope to `none`. While not technically a scope, `none` means that the bean will not live within any particular scope, but will instead be instantiated each time it is referenced. You should set a bean's scope to `none` when it is referenced by another bean.

6. You can optionally add needed properties for the bean. With the bean selected in the **Managed Beans** table, click the **Add** icon for the **Managed Properties** table. Enter a property name (other fields are optional).

Note: While you can declare managed properties using this editor, the corresponding code is not generated on the Java class. You will need to add that code by creating private member fields of the appropriate type and then using the **Generate Accessors** menu item on the context menu of the source editor to generate the corresponding getter and setter methods for these bean properties.

24.4.2 What Happens When You Create a Managed Bean

When you use the configuration editor to create a managed bean and elect to generate the Java file, JDeveloper creates a stub class with the given name and a default constructor. Example 24–6 shows the code added to the `MyBean` class stored in the view package.

Example 24–6 Generated Code for a Managed Bean

```
package view;

public class MyBean {
    public MyBean() {
```

```
    }
}
```

You now need to add the logic required by your task flow or page. You can then refer to that logic using an EL expression that refers to the `managed-bean-name` value given to the managed bean. For example, to access the `myInfo` property on the bean, the EL expression would be:

```
#{my_bean.myInfo}
```

JDeveloper also adds a `managed-bean` element to the appropriate task definition file. Example 24–7 shows the `managed-bean` element created for the `MyBean` class.

Example 24–7 Managed Bean Configuration on the adfc-config.xml File

```
<managed-bean>
  <managed-bean-name>my_bean</managed-bean-name>
  <managed-bean-class>view.MyBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

24.4.3 How to Set Managed Bean Memory Scopes in a Server-Cluster Environment

Typically, in an application that runs in a clustered environment, a portion of the application's state is serialized and copied to another server or a data store at the end of each request so that the state is available to other servers in the cluster.

> **Note:** If the managed bean will be calling `set` and `get` methods on ADF Faces components, you cannot serialize the managed beans because ADF Faces components are not serializable. You will need to access the ADF Faces components in another way.

When you are designing an application to run in a clustered environment, you must:

- Ensure that all managed beans with a lifespan longer than one request are serializable (that is, they implement the `java.io.Serializable` interface). Specifically, beans stored in session scope, page flow scope, and view scope need to be serializable.

  > **Tip:** To identify failures with objects stored in page flow scope and view scope, use `writeObject()`. This method provides additional information in an exception about the object and scope that failed to serialize. Additional information might be a region's page flow scope and the key of the object. **:**

- Make sure that the framework is aware of changes to managed beans stored in ADF scopes (view scope and page flow scope).

When a value within a managed bean in either view scope or page flow scope is modified, the application needs to notify the framework so that it can ensure that the bean's new value is replicated.

In Example 24–8, an attribute of an object in view scope is modified.

Example 24–8 Code That Modifies an Object in viewScope

```
Map<String, Object> viewScope =
```

```
    AdfFacesContext.getCurrentInstance().getViewScope();
MyObject obj = (MyObject)viewScope.get("myObjectName");
Obj.setFoo("newValue");
```

Without additional code, the framework will be unaware of this change and it will not know that a new value needs to be replicated within the cluster. To inform the framework that an object in an ADF scope has been modified and that replication is needed, use the `markScopeDirty()` method, as shown in Example 24–9. The `markScopeDirty()` method accepts only `viewScope` and `pageFlowScope` as parameters.

Example 24–9 Additional Code to Notify Oracle ADF of Changes to an Object

```
ControllerContext ctx = ControllerContext.getInstance();
ctx.markScopeDirty(viewScope);
```

This code is needed for any request that modifies an existing object in one of the ADF scopes. If the scope itself is modified by the scope's `put()`, `remove()`, or `clear()` methods, it is not necessary to notify the framework.

If an application is not deployed to a clustered environment, the tracking of changes to ADF memory scopes is not needed, and by default, this functionality is disabled. To enable ADF Controller to track changes to ADF memory scopes and replicate the page flow scope and view scope within the server cluster, set the `<adf-scope-ha-support>` parameter in the `adf-config.xml` file to `true`. Because scope replication has a small performance overhead, it should be enabled only for applications running in a server-cluster environment.

Example 24–10 shows `adf-scope-ha-support` set to `true` in the `adf-config.xml` file.

Example 24–10 adf-scope-ha-support Parameter in the adf-config.xml File

```
<?xml version="1.0" encoding="US-ASCII" ?>
<adf-config xmlns="http://xmlns.oracle.com/adf/config"
      xmlns:adfc="http://xmlns.oracle.com/adf/controller/config">
   <adfc:adf-controller-config>
...
    <adfc:adf-scope-ha-support>true</adfc:adf-scope-ha-support>
...
  </adfc:adf-controller-config>
...
</adf-config>
```

25

Understanding the Fusion Page Lifecycle

This chapter describes the ADF page lifecycle, its phases, and how to best use the lifecycle within a Fusion web application.

This chapter includes the following sections:

- Section 25.1, "About the Fusion Page Lifecycle"
- Section 25.2, "About the JSF and ADF Page Lifecycles"
- Section 25.3, "About Object Scope Lifecycles"
- Section 25.4, "Customizing the ADF Page Lifecycle"

25.1 About the Fusion Page Lifecycle

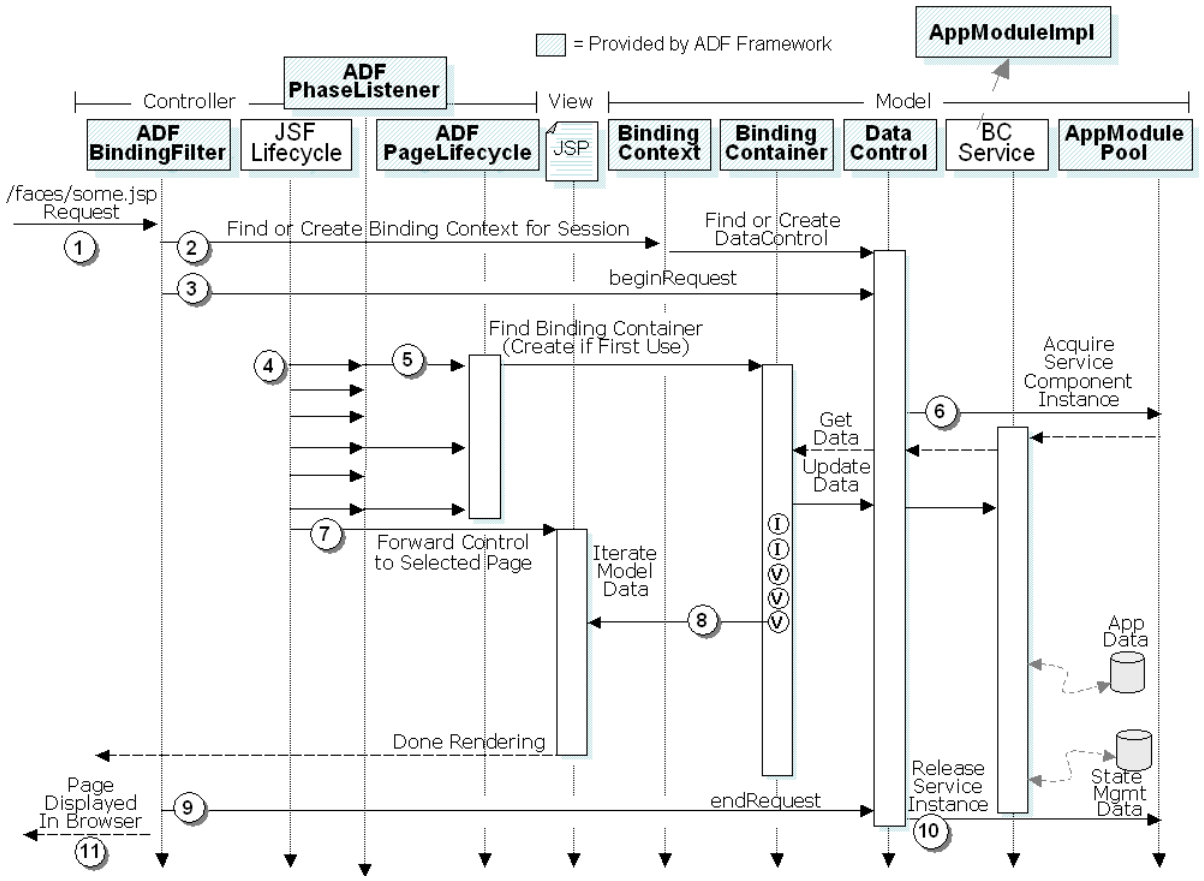
When a page is submitted and a new page requested, the application invokes both the ADF Faces page lifecycle, which extends the standard JSF request lifecycle, and the ADF page lifecycle. The ADF Faces page lifecycle handles submitting the values on the page, validating component values, navigating pages, displaying components on the resulting page, and saving and restoring state. The JSF lifecycle phases use a UI component tree to manage the display of the faces components. This tree is a runtime representation of a JSF page: each UI component tag in a page corresponds to a UI component instance in the tree. The `FacesServlet` servlet manages the request processing lifecycle in JSF applications. `FacesServlet` creates an object called `FacesContext`, which contains the information necessary for request processing, and invokes an object that executes the lifecycle.

For more details about the extended JSF lifecycle, see the "Understanding the JSF and ADF Faces Lifecycles" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

The ADF page lifecycle handles preparing and updating the data model, validating the data at the model layer, and executing methods on the business layer. The ADF page lifecycle uses the binding container to make data available for easy referencing by the page during the current page request.

The combined JSF and ADF page lifecycle is only one sequence within a larger sequence of events that begins when an HTTP request arrives at the application server and continues until the page is returned to the client. This overall sequence of events can be called the *web page lifecycle*. It follows processing through the model, view, and controller layers as defined by the MVC architecture. The page lifecycle is not a rigidly defined set of events, but is rather a set of events for a typical use case. Figure 25–1 shows a sequence diagram of the lifecycle of a web page request using JSF and Oracle ADF in tandem.

Figure 25–1 Lifecycle of a Web Page Request Using JSF and Oracle ADF



The basic flow of processing a web page request using JSF and Oracle ADF happens as follows:

1. A web request for `http://yourserver/yourapp/faces/some.jsp` arrives from the client to the application server.
2. The `ADFBindingFilter` object looks for the ADF binding context in the HTTP session, and if it is not yet present, initializes it for the first time. Some of the functions of the `ADFBindingFilter` include finding the name of the binding context metadata file, and finding and constructing an instance of each data control.
3. The `ADFBindingFilter` object invokes the `beginRequest()` method on each data control participating in the request. This method gives the data control a notification at the start of every request so that it can perform any necessary setup.
4. The JSF `Lifecycle` object, which is responsible for orchestrating the standard processing phases of each request, notifies the `ADFPhaseListener` class during each phase of the lifecycle, so that it can perform custom processing to coordinate the JSF lifecycle with the ADF Model data binding layer. For more information about the details of the JSF and ADF page lifecycle phases, see Section 25.2, "About the JSF and ADF Page Lifecycles."

Note: The `FacesServlet` class (in `javax.faces.webapp`), configured in the `web.xml` file of a JSF application, is responsible for initially creating the JSF `Lifecycle` class (in `javax.faces.lifecycle`) to handle each request. However, since it is the `Lifecycle` class that does all the interesting work, the `FacesServlet` class is not shown in the diagram.

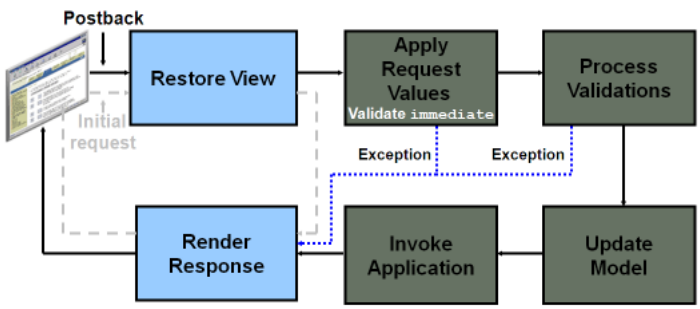
5. The `ADFPhaseListener` object creates an ADF `PageLifecycle` object to handle each request and delegates the appropriate before and after phase methods to corresponding methods in the ADF `PageLifecycle` class. If the binding container for the page has never been used before during the user's session, it is created.
6. The first time an application module data control is referenced during the request, it acquires an instance of the application module from the application module pool.
7. The JSF `Lifecycle` object forwards control to the page to be rendered.
8. The UI components on the page access value bindings and iterator bindings in the page's binding container and render the formatted output to appear in the browser.
9. The `ADFBindingFilter` object invokes the `endRequest()` method on each data control participating in the request. This method gives a data control notification at the end of every request, so that they can perform any necessary resource cleanup.
10. An application module data control uses the `endRequest` notification to release the instance of the application module back to the application module pool.
11. The user sees the resulting page in the browser.

The ADF page lifecycle also contains phases that are defined simply to notify ADF page lifecycle listeners before and after the corresponding JSF phase is executed (that is, there is no implementation for these phases). These phases allow you to create custom listeners and register them with any phase of both the JSF and ADF page lifecycles, so that you can customize the ADF page lifecycle if needed, both globally or at the page level.

25.2 About the JSF and ADF Page Lifecycles

Figure 25–2 shows a high-level view of the combined JSF and ADF page lifecycles.

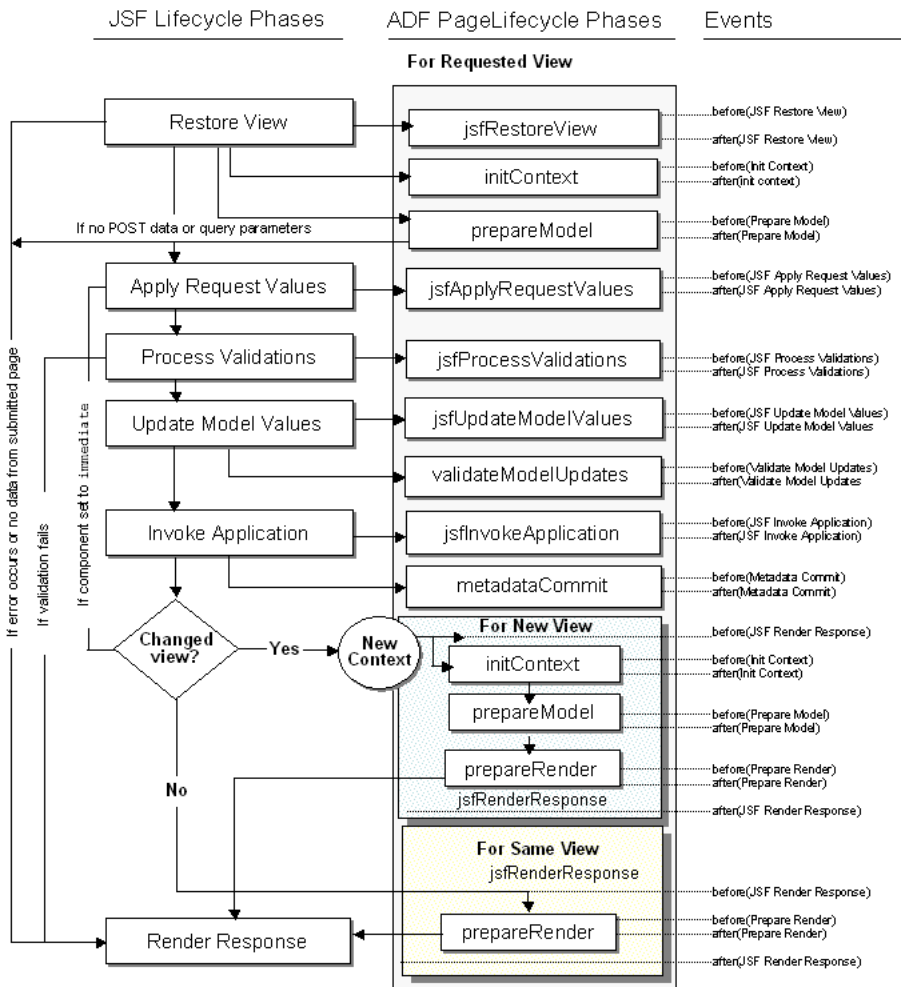
Figure 25–2 JSF and ADF Page Lifecycles



Say for example, you have a page with some text displayed in an input text component and a command button. When that page is first rendered, the component tree is built during the Restore View phase and then the lifecycle goes straight to the Render Response phase, as the components are rendered. When the user clicks the button, the full lifecycle is invoked. The component tree is rebuilt, the input text component extracts any new value during the Apply Request Values phase and Process Validations phase, and if there is an error, for example due to validation, then the lifecycle jumps to the Render Response phase. Otherwise, the model is then updated with the new value during the Update Model phase, and then any application processing associated with the command button (such as navigation), is executed during the Invoke Application phase.

Figure 25–3 shows the details of how the JSF and ADF Faces and ADF model phases integrate in the lifecycle of a page request.

Figure 25–3 Lifecycle of a Page Request in a Fusion Web Application



In a JSF application that uses the ADF Model layer, the phases in the page lifecycle are as follows:

- Restore View: The URL for the requested page is passed to the `bindingContext` object, which finds the page definition file that matches the URL. The component tree of the requested page is either newly built or restored. All the component tags, event handlers, converters, and validators on the submitted page have access

to the `FacesContext` instance. If the component tree is empty, (that is, there is no data from the submitted page), the page lifecycle proceeds directly to the Render Response phase.

If any discrepancies between the request state and the server-side state are detected, an error will is thrown and the page lifecycle jumps to the Render Response phase.

- JSF Restore View: Provides before and after phase events for the Restore View phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener were registered with the Restore View phase. The Initialize Context phase of the ADF Model page lifecycle listens for the `after(JSF Restore View)` event and then executes. The ADF Controller uses listeners for the before and after events of this phase to synchronize the server-side state with the request. For example, it is in this phase that browser back button detection and bookmark reference are handled. After the before and after listeners are executed, the page flow scope is available.
- Initialize Context: The page definition file is used to create the `bindingContainer` object, which is the runtime representation of the page definition file for the requested page. The `LifecycleContext` class used to persist information throughout the ADF page lifecycle phases is instantiated and initialized with values for the associated request, binding container, and lifecycle.
- Prepare Model: The ADF page lifecycle enters the Prepare Model phase by calling the `BindingContainer.refresh(PREPARE_MODEL)` method. During the Prepare Model phase, `BindingContainer` page parameters are prepared and then evaluated. If parameters for a task flow exist, they are passed into the flow.

  Next, any executables that have their refresh property set to `prepareModel` are refreshed based on the order of entry in the page definition file's `<executables>` section and on the evaluation of their `RefreshCondition` properties (if present). When an executable leads to an iterator binding refresh, the corresponding data control will be executed, and that leads to execution of one or more collections in the service objects. If an iterator binding fails to refresh, a JBO exception will be thrown and the data will not be available to display. For more information, see Section 25.2.2, "What You May Need to Know About Using the Refresh Property Correctly."

  If the incoming request contains no `POST` data or query parameters, then the lifecycle forwards to the Render Response phase.

  If the page was created using a template, and that template contains bindings using the ADF Model layer, the template's page definition file is used to create the binding container for the template. The container is then added to the binding context.

  If any `taskFlow` executable bindings exist (for example, if the page contains a region), the `taskFlow` binding creates an ADF Controller `ViewPortContext` object for the task flow, and any nested binding containers for pages in the flow are then executed.
- Apply Request Values: Each component in the tree extracts new values from the request parameters (using its decode method) and stores those values locally. Most associated events are queued for later processing. If you have set a component's `immediate` attribute to `true`, then the validation, conversion, and events associated with the component are processed during this phase and the lifecycle skips the Process Validations, Update Model Values, and Invoke Application phases. Additionally, any associated iterators are invoked. For more information about ADF Faces validation and conversion, see the "Validating and

Converting Input" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

- JSF Apply Request Values: Provides before and after phase events for the Apply Request Values phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener were registered with the Apply Request Values phase.
- Process Validations: Local values of components are converted and validated on the client. If there are errors, the lifecycle jumps to the Render Response phase. At the end of this phase, new component values are set, any validation or conversion error messages and events are queued on `FacesContext`, and any value change events are delivered. Exceptions are also caught by the binding container and cached.
- JSF Process Validations: Provides before and after phase events for the Process Validations phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener were registered with the Process Validations phase.
- Update Model Values: The component's validated local values are moved to the model and the local copies are discarded. For any updateable components (such as an `inputText` component), corresponding iterators are refreshed, if the refresh condition is set to the default (`deferred`) and the refresh condition (if any) evaluates to `true`.

  If you are using a backing bean for a JSF page to manage your UI components, any UI attributes bound to a backing bean property will also be refreshed in this phase.
- JSF Update Model Values: Provides before and after phase events for the Update Model Values phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener were registered with the Update Model Values phase.
- Validate Model Updates: The updated model is now validated against any validation routines set on the model. Exceptions are caught by the binding container and cached.
- Invoke Application: Any action bindings for command components or events are invoked.
- JSF Invoke Application: Provides before and after phase events for the Invoke Application phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener were registered with the Invoke Application phase.
- Metadata Commit: Changes to runtime metadata are committed. This phase stores any runtime changes made to the application using the Metadata Service (MDS). For more information about using MDS to persist runtime changes, see Chapter 39, "Customizing Applications with MDS."
- Initialize Context (only if navigation occurred in the Invoke Application lifecycle): The Initialize Context phase listens for the `beforeJSFRenderResponse` event to execute. The page definition file for the next page is initialized.
- Prepare Model (only if navigation occurred in the Invoke Application lifecycle): Any page parameters contained in the next page's definition are set.
- Prepare Render: The binding container is refreshed to allow for any changes that may have occurred in the Apply Request Values or Validation phases. Any iterators that correspond to read-only components (such as an `outputText` component) are refreshed. Any dynamic regions are switched, if needed. The

`prepareRender` event is sent to all registered listeners, as is the `afterJSFRenderResponse` event.

> **Note:** Instead of displaying `prepareRender` as a valid phase for a selection, JDeveloper displays `renderModel`, which represents the `refresh(RENDER_MODEL)` method called on the binding container.

- Render Response: The components in the tree are rendered as the Java EE web container traverses the tags in the page. State information is saved for subsequent requests and the Restore View phase.

  In order to lessen the wait time required to display both a page and any associated data, certain ADF Faces components such as the `table` component, use data streaming for their initial request. When a page contains one or more of these components, the page goes through the normal lifecycle. However, instead of fetching the data during that request, a special separate request is run. Because the page has just rendered, only the Render Response phase executes for the components that use data streaming, and the corresponding data is fetched and displayed. If the user's action (for example scrolling in a table), causes a subsequent data fetch another request is executed. Tables, trees, tree tables, and data visualization components all use data streaming.
- JSF Render Response: Provides before and after phase events for the Render Response phase. You can create a listener and register it with the before or after event of this phase, and the application will behave as if the listener were registered with the Render Response phase.

25.2.1 What You May Need to Know About Partial Page Rendering and Iterator Bindings

ADF Faces provides an optimized lifecycle that you can use when you want the page request lifecycle (including conversion and validation) to be run only for certain components on a page, usually for components whose values have changed.

One way to use the optimized lifecycle is to manually set up dependencies so that the events from one component act as triggers for another component, known as the *target*. When any event occurs on the trigger component, the lifecycle is run on any target components, as well as on any child components of both the trigger and the target, causing only those components to be rerendered. This is considered a partial page rendering (PPR)

Example 25–1 shows radio buttons as triggers and a `panelGroupLayout` component that contains the output text to be the target (the `panelGroupLayout` component must be the target because the `outputText` component may not always be rendered).

Example 25–1 Example of Partial Page Rendering

```
<af:form>
  <af:inputText label="Required Field" required="true"/>
 <af:selectBooleanRadio id="show" autoSubmit="true" text="Show"
                        value="#{validate.show}"/>
  <af:selectBooleanRadio id="hide" autoSubmit="true" text="Hide"
                         value="#{validate.hide}"/>
  <af:panelGroupLayout partialTriggers="show hide" id="panel">
    <af:outputText value="You can see me!" rendered="#{validate.show}"/>
  </af:panelGroupLayout>
</af:form>
```

Because the `autoSubmit` attribute is set to `true` on the radio buttons, when they are selected, a `SelectionEvent` is fired. Because the `panelGroupLayout` component is set to be a target for both radio components, when that event is fired, only the `selectOneRadio` (the trigger), the `panelGroupLayout` component (the trigger's target), and its child component (the `outputText` component) are processed through the lifecycle. Because the `outputText` component is configured to render only when the **Show** radio button is selected, the user is able to select that radio button and see the output text, without having to enter text into the required input field above the radio buttons.

Instead of having to set partial triggers and targets within code manually, the ADF Faces framework provides automatic PPR. Automatic PPR happens when an event has a corresponding event root component, or when the component itself is an event root. An event root component determines the boundaries of PPR. For example, the `attributeChangeEvent` considers the `inputText` component that invokes the event, its event root. Therefore, in the previous example, when the value changes for the `inputText` component, because it is the initial boundary for the PPR, the lifecycle would be run only on that `inputText` component and any child components.

Along with event roots, by default for Fusion web applications, automatic PPR also occurs for all UI components associated with the same iterator binding. All associated components refresh whenever any one of them has a value change event. This functionality is controlled by setting the `changeEventPolicy` attribute for iterators to `ppr`. By default, this is set globally.

For example, say you create a number of `inputText` components using the Data Controls panel (for example, when you create a form). When the `attributeChangeEvent` occurs on one of those `inputText` components, because the other `inputText` components are associated with the same iterator, all those `inputText` components will also be refreshed.

Another example might be a page that contains a `panelSplitter` component, where the left part of the splitter contains a form to create a new customer (created by dropping a form using the `CustomerInfoVO1` collection), and the right part of the splitter contains instructions for using the form. Because the bindings for the `inputText` components that make up the form are all associated with the `CustomerInfoVO1Iterator` iterator, anytime the form is submitted and a value for one of the `inputText` components has changed, only the `inputText` components associated with the `CustomerInfoVO1Iterator` will be processed through the lifecycle. The rest of the page, including the instructions in the right side of the splitter, will not be refreshed.

Note: In order for automatic PPR to happen, the event and event root component must be listed in Table 5-1 of the "Events and Partial Page Rendering" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*. If it is not listed, you will need to set up PPR manually, as described in the Enabling Partial Page Rendering Declaratively section of the same guide.

For example, say you create a form and a table from the same data control collection. The form shows the details for a row of data, while the table shows all rows in a collection, with the row currently displayed in the form shown as selected. When you click the **Next** button on the form, you want the table to refresh to show the new row as selected. Because the selection event is not supported by automatic PPR, the table will not refresh, even though the table and form both use the same iterator. You will need to set the **Next** and **Previous** buttons to be triggers of PPR for the target table.

Note: Only UI components that have a control binding defined in the page definition file associated with the configured iterator will be refreshed.

If you directly access the iterator from a managed bean and expose iterator values to UI components through that bean, then automatic PPR will not happen for those UI components. In these cases, you need to configure PPR manually. For more information, see the "Enabling Partial Page Rendering Declaratively" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Note: The ADF Faces query components do not support automatic PPR. If you set `changeEventPolicy` to be `ppr` globally, any iterators associated with bindings for those components will have the `changeEventPolicy` attribute set to `none`.

By default, all new applications use global PPR. You do not have to set it. You can however, override the setting in the page definition file for a page.

Before you begin:

It may be helpful to have an understanding of partial page rendering. For more information, see Section 25.2.1, "What You May Need to Know About Partial Page Rendering and Iterator Bindings."

To set iterator bindings to use PPR:

1. To set a specific iterator binding to use or not use PPR, open the associated page definition file, select the iterator, and in the Property Inspector, set **ChangeEventPolicy**.
2. To set all iterator bindings to use PPR:

1. In the Application Navigator, expand the Application Resources panel, expand the **Descriptors** and **ADF META-INF** nodes, and double click **adf-config.xml**.
2. In the overview editor, select the **Model** tab, and then select **Default Change Event Policy is Partial Page Rendering (PPR)**.

For more information about how the ADF Faces framework uses PPR, see the "Rendering Partial Page Content" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

25.2.2 What You May Need to Know About Using the Refresh Property Correctly

For scalability reasons, at runtime the iterator bindings in the binding container release any reference they have to a row set iterator at the end of each request. During the next request, each iterator binding rebinds itself to a "live" row set iterator that is tracking the current row of some data collection. The process of rebinding an ADF iterator binding during the ADF page lifecycle is known as *refreshing* the iterator. By default, this happens only once, on demand, when the iterator is first accessed by the client layer during the lifecycle. This initial access to the iterator typically occurs during page rendering, while EL expressions in the page, which reference the iterator or a control binding related to that iterator, are being evaluated. Alternatively, you can write code that programmatically causes the first access to the iterator to be before the Prepare Render phase. To force the iterator binding to refresh its row set iterator earlier in the lifecyle, you can set the `refresh` attribute on the iterator binding to a value other than the default.

> **Tip:** Refreshing an iterator binding does not forcibly reexecute its query each time. The first time the view object instance's row set iterator is accessed during a particular user's session, it will implicitly execute the view object's query if it was not already executed, as long as a search binding in the page definition related to that iterator has not already cleared the row set in preparation for allowing the user to enter search criteria before executing the query.
>
> Subsequent refreshing of the iterator binding related to that view object instance on page requests that are part of the same logical unit of work will only access the row set iterator again, not forcibly reexecute the query. When you want reexecuting the query to refresh its data, use the `Execute` or `ExecuteWithParams` built-in operation, or programmatically call the `executeQuery()` method on the iterator binding.

The `refresh` and `refreshCondition` attributes are used to determine when and whether to invoke an executable, such as an iterator binding, or an `invokeAction`. The value of the `refresh` attribute determines the lifecycle phase in which to invoke the executable, while the value of the `refreshCondition` attribute provides an optional boolean condition whose outcome determines whether the refresh will occur at the indicated lifecycle phase or not. By default, when JDeveloper adds an executable to a page definition (for example, when you drop an operation as a command component), the `refresh` attribute for the executable binding is set to `deferred`, which enforces execution whenever the binding is accessed the first time. If no `refreshCondition` value exists, the executable is invoked. If a value for `refreshCondition` exists, then that value is evaluated as an EL expression, and if the return value of the evaluation is `true`, then the executable is invoked. If the value evaluates to `false`, the executable is not invoked. For details about the `refresh` attribute, see Section A.8.1, "PageDef.xml Syntax.".

For most cases in a Fusion web application, you should not need to change the `refresh` or `refreshCondition` values on an iterator binding. These values are set to ensure that the correct data is displayed.

While the `invokeAction` executable continues to be supported for upward compatibility from previous releases, in Oracle ADF 11*g* you use a method activity in a task flow to call an action binding (or any backing bean method) to perform some application behavior before the page is rendered. For example, for a page used to create an object, you might have a task flow that begins with a method activity that calls the `CreateInsert` operation. The task flow then proceeds to the view activity for the page where the user inputs data. Modeling this behavior as discreet method call activities provides a much cleaner separation of application logic and data bindings, making applications both self-documenting and easier to maintain. For more information, see Section 26.6, "Creating an Input Form."

However, for completeness' sake, in case you encounter situations where you decide to change the `refresh` attribute for iterator bindings or for `invokeAction` (for example, if you have programmatic code that runs after the Prepare Model phase and needs to access the iterator programmatically), you should do so, informed of the following information regarding the valid values:

- `deferred` (the default): On demand

  Tip: Any `invokeAction` executable in a page definition file must have a value other than the default (`deferred`) for its `refresh` attribute, or it will not be refreshed and invoked.

- `prepareModel`: During the Prepare Model phase.
- `renderModel`: During the Prepare Render phase.

  Tip: Notice in Figure 25–3 that the key distinction between the Prepare Model phase and the Prepare Render phase is that one comes before JSF's Invoke Application phase, and one after. Since JSF's Invoke Application phase is when action listeners fire, if you need your iterator refreshed or the method or operation associated with the `invokeAction` to execute after these action listeners have performed their processing, you'll want to set the `refresh` attribute to `renderModel`.

- `ifNeeded`: During the Prepare Model and Prepare Render phases, only if needed. For iterators, the refresh is considered needed if the binding has not yet been refreshed. To determine if the execution is needed for an `invokeAction`, the framework compares the current set of evaluated parameter values with the set that was used to invoke the method action binding previously. If the parameter values for the current invocation are exactly the same as those used previously, the `invokeAction` does not invoke its bound method action binding. Use this setting if the `invokeAction` executable binds to a method action binding that accepts parameters.

  Tip: For `invokeAction` executables bound to methods that do not take parameters, the `invokeAction` executable will be called twice. To use the `invokeAction` executable with parameterless methods, you should use ensure that the condition associated with the `refreshCondition` attribute evaluates to invoke the method only if the value has changed. This will prevent multiple invocations.

- `prepareModelIfNeeded` and `renderModelIfNeeded`: Same as `ifNeeded`, except that it is executed during the named phase.
- `never`: Not valid for `invokeAction` executables. For iterators, the iterator will never be refreshed. Use when your own code calls `getRowSetIterator()` on the iterator binding.
- `always`: Not valid for `invokeAction` executables. For iterators, the iterator will always be refreshed (potentially multiple times) during both the Prepare Model and Prepare Render phases, as well as during the Update Model phase.
- `refreshAfter`: Use to handle dependencies between executables. For example, you can set the condition so that this executable refreshes after another executable.

  Tip: You can determine the order of executable invocation using the `refreshAfter` attribute. For example, suppose you have two `invokeAction` elements—one with an ID of `myAction` and another with an ID of `anotherAction`—and you want `myAction` to fire after `anotherAction`. You would set the `refreshAfter` condition on `myAction` to `anotherAction`.

25.2.3 What You May Need to Know About Task Flows and the Lifecycle

Task flows are initially refreshed when the parent binding container (the one associated with the page) is refreshed. This happens in the Prepare Model phase. On a subsequent request, the task flow will be refreshed during the Prepare Render phase, depending on its `refresh` and `refreshCondition` attributes and its parameter value.

Note: Any child page fragment's page definition still handles the refresh of the bindings of the child page fragments.

Tip: If you have a region on a page that is not initially disclosed (for example, a popup dialog), the parameters still need to be available when the parent page is rendered, even though the region might not be displayed. If a region requires parameters, but those parameter values will not be available when the parent page is rendered, then you should use dynamic regions. If the parameters are null, an empty task flow can be used until the parameters for the region are ready and that region can display. To swap in an empty task flow, you set the dynamic region's `taskFlowId` attribute to an empty string.

If you set an EL expression as the value of the `refreshCondition` attribute, it will be evaluated during the Prepare Render phase of the lifecycle. When the expression evaluates to `true`, the task flow will be refreshed again. When `refreshCondition` evaluates to `false`, the behavior is the same as if the `refreshCondition` had not been specified.

Note: If the variable `bindings` is used within the EL expression, the context refers to the binding container of the parent page, not the page fragment displayed within the region.

The valid values for the `refresh` property of a task flow executable are as follows:

- `default`: The region will be refreshed only once, when the parent page is first displayed.
- `ifNeeded`: Refreshes the region only if there has been a change to `taskFlow` binding parameter values. If the `taskFlow` binding does not have parameters, then `ifNeeded` is equivalent to the default. When `ifNeeded` is used, the `refreshCondition` attribute is not considered.

> **Note:** Setting the `refresh` attribute to `ifNeeded` takes precedence over any value for the `refreshCondition` attribute. Also note that `ifNeeded` is not supported when you pass parameters to the `taskFlow` binding using a dynamic parameter `Map`. Instead, use `refreshCondition="#{EL.Expression}"`.

Because the only job of the `taskFlow` binding is to refresh its parameters, setting `Refresh` to `always` does not make sense. If the `taskFlow` binding's parameters don't change, there is no reason to refresh the ADF region.

Note that the child page fragment's page definition still handles the refresh of the bindings of the child page fragments.

25.3 About Object Scope Lifecycles

At runtime, ADF objects such as the binding container and managed beans are instantiated. Each of these objects has a defined lifespan set by its scope attribute. You can access a scope as a `java.util.Map` from the `RequestContext` API. For example, to access an object named `foo` in the request scope, you would use the expression `#{requestScope.foo}`.

There are six types of scopes in a Fusion web application:

- Application scope: The object is available for the duration of the application.
- Session scope: The object is available for the duration of the session.

> **Note:** There's no window uniqueness for session scope, all windows in the session share the same session scope instance. If you are concerned about multiple windows being able to access the same object (for example to ensure that managed beans do not conflict across windows), you should use a scope that is window-specific, such as page flow or view scope.

- Page flow scope: The object is available for the duration of a bounded task flow.

> **Note:** Because this is not a standard JSF scope, EL expressions must explicitly include the scope to reference bean. For example, to reference the `MyBean` managed bean from the `pageFlowScope` scope, your expression would be `#{pageFlowScope.MyBean}`.

- Request scope: The object is available from the time an HTTP request is made until a response is sent back to the client.
- Backing bean scope: Used for managed beans for page fragments and declarative components only, the object is available from the time an HTTP request is made

until a response is sent back to the client. This scope is needed for fragments and declarative components because there may be more than one page fragment or declarative component on a page, and to prevent collisions, any values must be kept in separate scope instances. Therefore, any managed bean for a page fragment or declarative component must use backing bean scope.

Note: Because this is not a standard JSF scope, EL expressions must explicitly include the scope to reference bean. For example, to reference the `MyBean` managed bean from the backing bean scope, your expression would be `#{backingBeanScope.MyBean}`.

- View scope: The object is available until the view ID for the current view activity changes. This scope can be used to hold values for a given page. However, unlike request scope, which can be used to store a value needed from one page to the next, anything stored in view scope will be lost once the view ID changes.

Note: Both JSF and ADF Faces have an implementation of view scope. Only the ADF Faces view scope survives page redirects and refreshes. In a Fusion web application, when you use `viewScope` in an expression, it resolves to the ADF Faces view scope.

Because this view scope is not a standard JSF scope, EL expressions must explicitly include the scope to reference bean. For example, to reference the `MyBean` managed bean from the view scope, your expression would be `#{viewScope.MyBean}`.

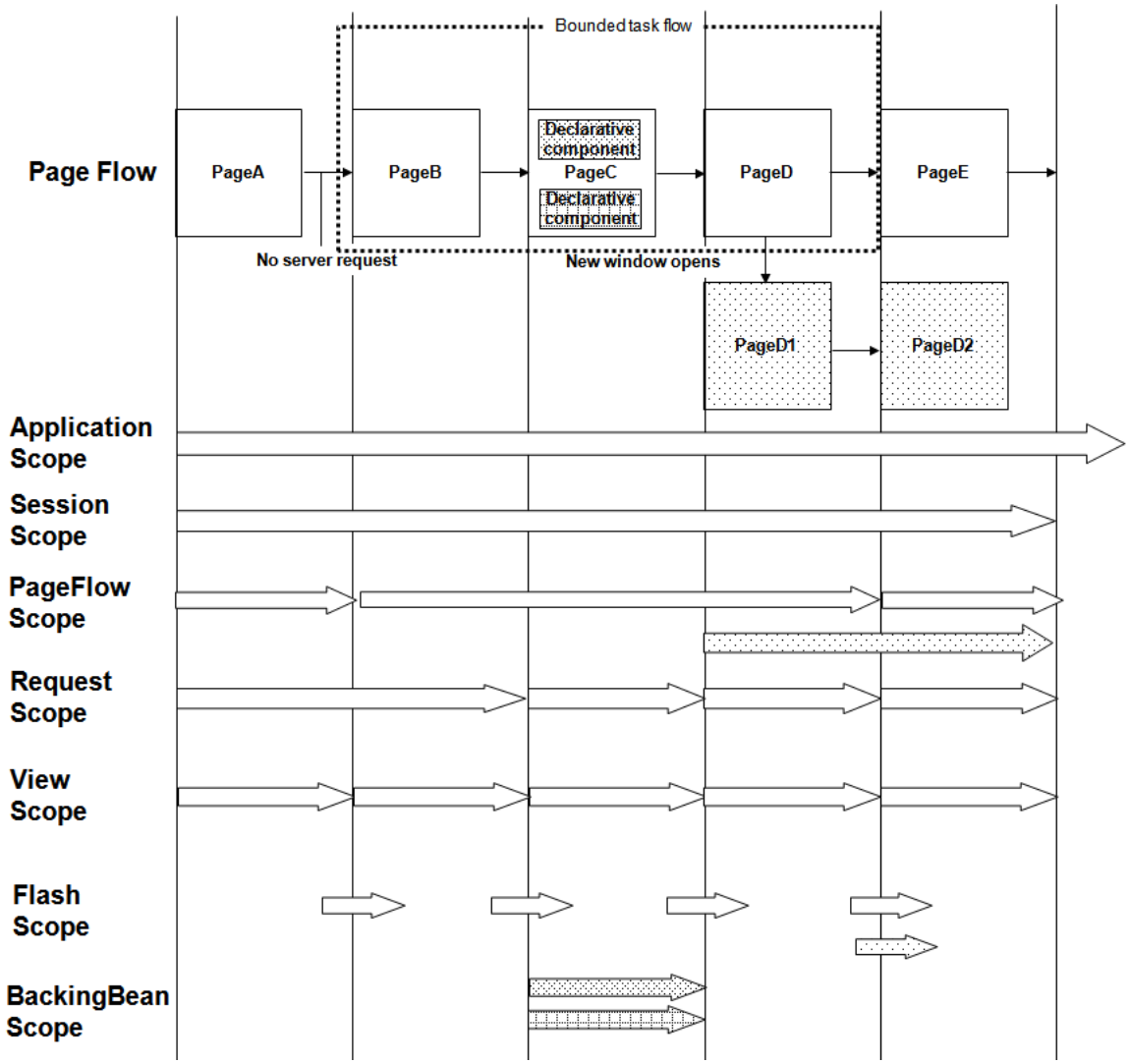
Note: When you create objects (such as a managed bean) that require you to define a scope, you can set the scope to `none`, meaning that it will not live within any particular scope, but will instead be instantiated each time it is referenced.

Object scopes are analogous to global and local variable scopes in programming languages. The wider the scope, the higher availability of an object. During their life, these objects may expose certain interfaces, hold information, or pass variables and parameters to other objects. For example, a managed bean defined in session scope will be available for use during multiple page requests. However, a managed bean defined in request scope will only be available for the duration of one page request.

By default, the binding container and the binding objects it contains are defined in session scope. However, the *values* referenced by value bindings and iterator bindings are undefined between requests and for scalability reasons do not remain in session scope. Therefore, the values that binding objects refer to are valid only during a request in which that binding container has been prepared by the ADF lifecycle. What stays in session scope are only the binding container and binding objects themselves.

Figure 25–4 shows the time period during which each type of scope is valid.

Figure 25–4 Relationship Between Scopes and Page Flow



When determining what scope to register a managed bean with, always try to use the narrowest scope possible. Only use the session scope for information that is relevant to the whole session, such as user or context information. Avoid using session scope to pass values from one task flow to another. When creating a managed bean for a page fragment or a declarative component, you must use backing bean scope.

Managed beans can be registered in either the `adfc-config.xml` or the configuration file for a specific task flow. For more information about using managed beans in a Fusion application, see Section 24.4, "Using a Managed Bean in a Fusion Web Application."

Note: Registering managed beans within the `faces-config.xml` file is not recommended in a Fusion web application.

25.3.1 What You May Need to Know About Object Scopes and Task Flows

When determining what scope to use for variables within a task flow, you should use any of the scope options other than application or session scope. These two scopes will persist objects in memory beyond the life of the task flow and therefore compromise

the encapsulation and reusable aspects of a task flow. In addition, application and session scopes may keep objects in memory longer than needed, causing unneeded overhead.

When you need to pass data values between activities within a task flow, you should use page flow scope. View scope is recommended for variables that are needed only within the current view activity, not across view activities. Request scope should be used when the scope does not need to persist longer than the current request. It is the only scope that should be used to store UI component information. Lastly, backing bean scope must be used for backing beans in your task flow if there is a possibility that your task flow will appear in two region components or declarative components on the same page and you would like to achieve region instance isolations.

25.4 Customizing the ADF Page Lifecycle

The ADF lifecycle contains clearly defined phases that notify ADF lifecycle listeners before and after the corresponding JSF phase is executed. You can customize this lifecycle by creating a custom phase listener that invokes your needed code, and then registering it with the lifecycle to execute during one of these phases.

For example, if you do not want to use the default "Managed State" release level for application modules, you can set the release level from the after-prepareRender phase of the ADF lifecycle using a custom ADF page phase-listener class (for more information, see Section 43.4.5, "How to Set Release Level in an ADF PagePhaseListener").

Note: An application cannot have multiple phase listener instances. An initial `ADFPhaseListener` instance is, by default, registered in the `META-INF/faces-config.xml` configuration file. Registering, for example, a customized subclass of the `ADFPhaseListener` creates a second instance. In this scenario, only the instance that was most recently registered is used.

The following warning message indicates when an instance has been replaced by a newer one: "ADFc: Replacing the ADF Page Lifecycle implementation with *class name of the new listener*."

25.4.1 How to Create a Custom Phase Listener

To create a custom phase listener, you must create a listener class that implements the `PagePhaseListener` interface. You then add methods that execute code either before or after the phase that the code needs to execute.

Example 25–2 contains a template that you can modify to create a custom phase listener. See Section 4.13.1, "How to Generate Custom Classes," for more information about creating a class in JDeveloper.

Example 25–2 Example Custom Phase Listener

```
public class MyPagePhaseListener implements PagePhaseListener
{
   public void afterPhase(PagePhaseEvent event)
   {
      System.out.println("In afterPhase " + event.getPhaseId());
   }
```

```
   public void beforePhase(PagePhaseEvent event)
   {
      System.out.println("In beforePhase " + event.getPhaseId());
   }
}
```

Once you create the custom listener class, you need to register it with the phase in which the class needs to be invoked. You can either register it globally (so that the whole application can use it), or you can register it only for a single page.

25.4.2 How to Register a Listener Globally

To customize the ADF lifecycle globally, register your custom phase listener by editing the `adf-settings.xml` configuration file. The `adf-settings.xml` file is shared by several ADF components, including ADF Controller, to store configuration information.

To register the listener in adf-settings.xml:

1. In the Application Navigator, beneath the **META-INF** node, double-click the **adf-settings.xml** file.
2. In the editor window, click the **Source** tab
3. In the source editor, scroll down to `<adfc-controller-config xmlns= "http://xmlns.oracle.com/adf/controller/config">`

   If this entry does not exist, add it to the file.
4. Enter the remaining elements shown in italics:

```
<?xml version="1.0" encoding="US-ASCII" ?>
<adf-config xmlns="http://xmlns.oracle.com/adf/config">
  .
  .
  .
    <adfc-controller-config
xmlns="http://xmlns.oracle.com/adf/controller/config">
       <lifecycle>
         <phase-listener>
            <listener-id>MyPagePhaseListener</listener-id>
            <class>mypackage.MyPagePhaseListener</class>
         </phase-listener>
       </lifecycle>
   </adfc-controller-config>
  .
  .
  .
</adf-config>
```

5. Add values for the following elements:
   - `<listener-id>` A unique identifier for the listener (you can use the fully qualified class name)
   - `<class>` The class name of the listener

25.4.3 What You May Need to Know About Listener Order

You can specify multiple phase listeners in the `adf-settings.xml` and, optionally, the relative order in which they are called. When registering a new listener in the file, you determine the position in the list of listeners using two parameters:

- `beforeIdSet`: The listener is called before any of the listeners specified in `beforeIdSet`
- `afterIdSet`: The listener is called after any of the listeners specified in `afterIdSet`

Example 25–3 contains an example configuration file in which multiple listeners have been registered for an application.

Example 25–3 adf-settings.xml Configuration File with Multiple Listener Registration

```
<lifecycle>
   <phase-listener>
      <listener-id>MyPhaseListener</listener-id>
      <class>view.myPhaseListener</class>
      <after-id-set>
         <listener-id>ListenerA</listener-id>
         <listener-id>ListenerC</listener-id>
      </after-id-set>
      <before-id-set>
         <listener-id>ListenerB</listener-id>
         <listener-id>ListenerM</listener-id>
         <listener-id>ListenerY</listener-id>
      </before-id-set>
   </phase-listener>
</lifecycle>
```

In the example, `MyPhaseListener` is a registered listener that executes after listeners A and C but before listeners B, M, and Y. To execute `MyPhaseListener` after listener B, move the `<listener-id>` element for listener B under the `<after-id-set>` element.

25.4.4 How to Register a Lifecycle Listener for a Single Page

To customize the lifecycle of a single page, you set the `ControllerClass` attribute on the page definition file. This listener will be valid only for the lifecycle of the particular page described by the page definition. For more information about the page definition file and its role in a Fusion web application, see Section 13.7, "Working with Page Definition Files."

You specify a different controller class depending on whether it is for a standard JSF page or a page fragment:

To customize the ADF Lifecycle for a single page or page fragment:

1. In the Application Navigator, right-click the page or page fragment and choose **Go To Page Definition**.
2. In the Structure window, select the page definition node.
3. In the Property Inspector, click the dropdown menu next to **ControllerClass** and choose **Edit**.
4. Click the **Hierarchy** tab and navigate to the appropriate controller class for the page or page fragment. Following are the controller classes to use for different types of pages:

- Standard JSF page - specify
  `oracle.adf.controller.v2.lifecycle.PageController`

  If you need to receive afterPhase/beforePhase events, specify
  `oracle.adf.controller.v2.lifecycle.PagePhaseListener`

- Page fragment - specify
  `oracle.adf.model.RegionController`

  Tip: You can specify the value of the page definition's `ControllerClass` attribute as a fully qualified class name or you can enter an EL expression that resolves to a class directly in the **ControllerClass** field.

  When using an EL expression for the value of the `ControllerClass` attribute, the Structure window may show a warning indicating that e "`#{YourExpression}`" is not a valid class. You can safely ignore this warning.

25.4.5 What You May Need to Know About Extending RegionController for Page Fragments

Bindings inside page fragments used as regions are refreshed through the `refreshRegion` and `validateRegion` events of the `RegionController` interface. These events are available if you specify `oracle.adf.model.RegionController` in the **ControllerClass** field as described in Section 25.4.4, "How to Register a Lifecycle Listener for a Single Page."

As shown in Example 25–4, you can use the `refreshRegion` event to add custom code that executes before the region is refreshed. For example, you may want to refresh the bindings used by the page fragment in the region so that the refreshed binding values are propagated to the inner binding container.

To do this, create a new class that implements the `RegionController` interface. Then, write the following `refreshRegion` method, including your custom code that you want to execute before the Prepare Model phase.

Example 25–4 regionRefresh Method

```
public boolean refreshRegion(RegionContext regionCtx)
   {
      int refreshFlag = regionCtx.getRefreshFlag();
      if (refreshFlag == RegionBinding.PREPARE_MODEL)
      {
         // Execute some code before
      }
      // Propagate the refresh to the inner binding container
      regionCtx.getRegionBinding().refresh(refreshFlag);

      return false;
   }

   public boolean validateRegion(RegionContext regionCtx)
   {
      // Propagate the validate to the inner binding container
      regionCtx.getRegionBinding().validate();

      return false;
   }
```

As shown in Example 25–4, the refresh flag value can be:

- `RegionBinding.PREPARE_MODEL` - corresponds to the event occurring during the ADF Lifecycle `prepareModel` phase
- `RegionBinding.RENDER_MODEL` - corresponds to the event occurring during the ADF Lifecycle `prepareRender` phase

26

Creating a Basic Databound Page

This chapter describes how to use the Data Controls panel to create databound forms using ADF Faces components.

This chapter includes the following sections:

- Section 26.1, "About Creating a Basic Databound Page"
- Section 26.2, "Using Attributes to Create Text Fields"
- Section 26.3, "Creating a Basic Form"
- Section 26.4, "Incorporating Range Navigation into Forms"
- Section 26.5, "Creating a Form to Edit an Existing Record"
- Section 26.6, "Creating an Input Form"
- Section 26.7, "Using a Dynamic Form to Determine Data to Display at Runtime"
- Section 26.8, "Modifying the UI Components and Bindings on a Form"

26.1 About Creating a Basic Databound Page

You can create UI pages that allow you to display and collect information using data controls created for your business services. For example, using the Data Controls panel, you can drag an attribute for an item, and then choose to display the value either as read-only text or as an input text field with a label. JDeveloper creates all the necessary JSF tag and binding code needed to display and update the associated data. For more information about the Data Controls panel and the declarative binding experience, see Chapter 13, "Using ADF Model in a Fusion Web Application."

Instead of having to drop individual attributes, JDeveloper allows you to drop all attributes for an object at once as a form. The actual UI components that make up the form depend on the type of form dropped. You can create forms that display values, forms that allow users to edit values, and forms that collect values (input forms).

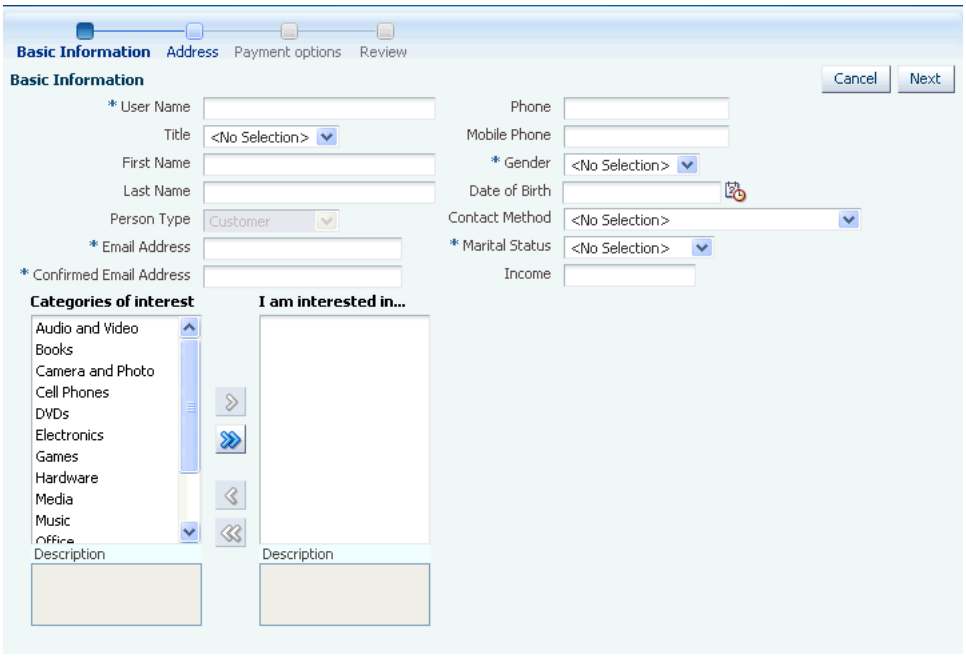
Once you drop the UI components, you can then drop built-in operations as command UI components that allow you to navigate through the records in a collection or that allow users to operate on the data, such as committing, deleting, or creating a record. For example, you can create a button that allows users to delete data objects displayed in the form. You can also modify the default components to suit your needs.

26.1.1 ADF Databound Forms Use Cases and Examples

You use forms when you need to collect or display a row of data or related data. For example, the StoreFront module contains a page that allows users to register information about themselves, as shown in Figure 26–1. This form was created by

dragging and dropping the `CustomerRegistration` collection from the Data Controls panel.

Figure 26–1 Register Customer Form in the StoreFront Module

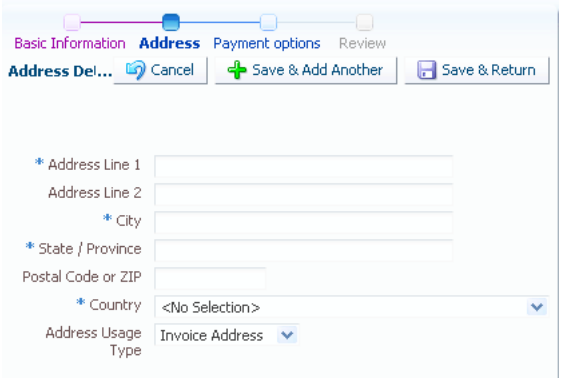


When you create a form, you can also choose to add navigation, so that the user can navigate quickly between records, as shown in Figure 26–2.

Figure 26–2 Navigate Between Records in a Form

You can also add command buttons that invoke processing on the row displayed in the form. For example, you can change values for a record and save those changes in an edit form, as shown in Figure 26–1, or you can create an input form that allows users to create a new record. Figure 26–3 shows an input form where a user can create a new address.

Figure 26–3 Input Form



When the attributes to display in a form are only available at runtime, you can create a dynamic form.

26.1.2 Additional Functionality for Databound Forms

You may find it helpful to understand other ADF Faces features before you implement your forms. Additionally, once you have added a form to you page, you may find that you need to add functionality such as security. Following are links to other functionality that you may find useful.

- **ADF view objects**: Much of how the components display and function in the form is controlled by the corresponding view objects. For more information, see Chapter 5, "Defining SQL Queries Using View Objects."
- **ADF Model and data binding**: When you create forms in an ADF Web application, you use ADF Model and data binding. For more information, see Appendix 13, "Using ADF Model in a Fusion Web Application."
- **ADF Faces**: You also use ADF Faces UI components. For detailed information about developing with ADF Faces, see the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
- **Advanced UI controls**: Basic forms use input and output components. You can also use more advanced components, such as lists, tables trees, and search forms. For more information, see the remaining chapters in Part V, "Creating a Databound Web User Interface".
- **Task flows**: If your form takes part in a transaction, for example an input form, then you may need to use an ADF task flow to invoke certain operations before or after the form is rendered. For more information, see Part IV, "Creating ADF Task Flows".
- **Page lifecycle:** For information about how forms works with the page lifecycle, see Chapter 25, "Understanding the Fusion Page Lifecycle."
- **Validation**: You may want certain fields to be validated before they are submitted to the data store. For more information, see Chapter 7, "Defining Validation and Business Rules Declaratively" and Chapter 16, "Using Validation in the ADF Model Layer."
- **Active data**: If your application uses active data, then you can have the data in your UI components update automatically, whenever the data in the data source changes. For more information, see Chapter 45, "Using the Active Data Service."

26.2 Using Attributes to Create Text Fields

JDeveloper allows you to create text fields declaratively in a WYSIWYG development environment for your JSF pages, meaning you can design most aspects of your pages without needing to look at the code. When you drag and drop items from the Data Controls panel, JDeveloper declaratively binds ADF Faces text UI components to attributes on a data control using an attribute binding.

26.2.1 How to Create a Text Field

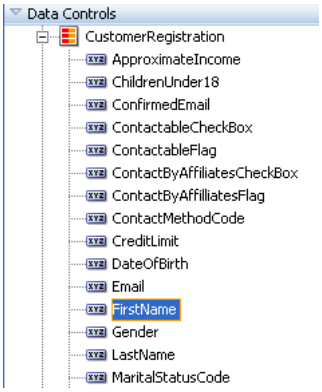
To create a text field that can display or update an attribute, you drag and drop an attribute of a collection from the Data Controls panel.

To create a bound text field:

1. From the Data Controls panel, select an attribute for a collection. For a description of the icons that represent attributes and other objects in the Data Controls panel, see Section 13.4, "Using the Data Controls Panel."

   For example, Figure 26–4 shows the `FirstName` attribute under the `CustomerRegistration` collection of the `StoreServiceAMDataControl` data control in the StoreFront module. This is the attribute to drop to display or enter the customer's first name.

Figure 26–4 Attributes Associated with a Collection in the Data Controls Panel



2. Drag the attribute onto the page, and from the context menu choose the type of widget to display or collect the attribute value. For an attribute, you are given the following choices:

   - **Text**:
     - **ADF Input Text w/ Label**: Creates an ADF Faces `inputText` component with a nested `validator` component. The `label` attribute is populated.

       Tip: For more information about validators and other attributes of the `inputText` component, see the "Using Input Components and Defining Forms" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

     - **ADF Input Text**: Creates an ADF Faces `inputText` component with a nested `validator` component. The `label` attribute is not populated.

  - **ADF Output Text w/ Label**: Creates a `panelLabelAndMessage` component that holds an ADF Faces `outputText` component. The `label` attribute on the `panelLabelAndMessage` component is populated.
  - **ADF Output Text:** Creates an ADF Faces `outputText` component. No label is created.
  - **ADF Output Formatted w/Label**: Same as ADF Output Text w/Label, but uses an `outputFormatted` component instead of an `outputText` component. The `outputFormatted` component allows you to add a limited amount of HTML formatting. For more information, see the "Displaying Output Text and Formatted Output Text" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*
  - **ADF Output Formatted**: Same as ADF Output Formatted w/Label, but without the label.
  - **ADF Label**: An ADF Faces `outputLabel` component.

- **List of Values:** Creates ADF LOV lists. For more information about how these lists work, see Section 5.12, "Working with List of Values (LOV) in View Object Attributes." For more information about using the lists on a JSF page, see Section 30.3, "Creating a Selection List."
- **Single selections**: Creates single selection lists. For more information about creating lists on a JSF page, see Section 30.3, "Creating a Selection List."

Note: These selections are your choices by default. However, the list of components available to use for an attribute can be configured as a control hint on the associated entity or view object. For more information, see Section 4.6, "Defining Attribute Control Hints for Entity Objects" and Section 5.13, "Defining UI Hints for View Objects."

For the purposes of this chapter, only the text components (and not the lists) will be discussed.

26.2.2 What Happens When You Create a Text Field

When you drag an attribute onto a JSF page and drop it as a UI component, among other things, a page definition file is created for the page (if one does not already exist). For a complete account of what happens when you drag an attribute onto a page, see Section 13.4.2, "What Happens When You Use the Data Controls Panel." Bindings for the iterator and attributes are created and added to the page definition file. Additionally, the necessary JSPX page code for the UI component is added to the JSF page.

26.2.2.1 Creating and Using Iterator Bindings

Whenever you create UI components on a page by dropping an item that is part of a collection from the Data Controls panel (or you drop the whole collection as a form or table), JDeveloper creates an iterator binding if it does not already exist. An iterator binding references an iterator for the data collection, which facilitates iterating over its data objects. It also manages currency and state for the data objects in the collection. An iterator binding does not actually access the data. Instead, it simply exposes the object that can access the data and it specifies the current data object in the collection. Other bindings then refer to the iterator binding in order to return data for the current

object or to perform an action on the object's data. Note that the iterator binding is not an iterator. It is a binding to an iterator. In the case of ADF Business Components, the actual iterator is the default row set iterator for the default row set of the view object instance in the application module's data model.

For example, if you drop the `FirstName` attribute under the `CustomerRegistration` collection, JDeveloper creates an iterator binding for the `CustomerRegistration` collection.

> **Tip:** There is one iterator binding created for each collection. This means that when you drop two attributes from the same collection (or drop the collection twice), they use the same binding. This is fine, unless you need the binding to behave differently for the different components. In that case, you will need to manually create separate iterator bindings.

The iterator binding's `rangeSize` attribute determines how many rows of data are fetched from a data control each time the iterator binding is accessed. This attribute gives you a relative set of 1-*n* rows positioned at some absolute starting location in the overall row set. By default, it the attribute set to `25`. For more information about using this attribute, see Section 26.4.2.2, "Iterator RangeSize Attribute." Example 26–1 shows the iterator binding created when you drop an attribute from the `CustomerRegistration` collection.

Example 26–1 Page Definition Code for an Iterator Binding for an Attribute Dropped from a Collection

```
<executables>
  <iterator Binds="CustomerRegistration" RangeSize="25"
            DataControl="StoreServiceAMDataControl"
            id="CustomerRegistrationIterator"/>
</executables>
```

For information regarding the iterator binding element attributes, see Appendix B, "Oracle ADF Binding Properties."

This metadata allows the ADF binding container to access the attribute values. Because the iterator binding is an executable, by default, it is invoked when the page is loaded, thereby allowing the iterator to access and iterate over the `CustomerRegistration` collection. This means that the iterator will manage all the `CustomerRegistration` objects in the collection, including determining the current `CustomerRegistration` or range of `CustomerRegistration` objects.

26.2.2.2 Creating and Using Value Bindings

When you drop an attribute from the Data Controls panel, JDeveloper creates an attribute binding that is used to bind the UI component to the attribute's value. This type of binding presents the value of an attribute for a single object in the current row in the collection. Value bindings can be used both to display and to collect attribute values.

For example, if you drop the `PrincipalName` attribute under the `CustomerRegistration` collection as an ADF Output Text w/Label widget onto a page, JDeveloper creates an attribute binding for the `PrincipalName` attribute. This allows the binding to access the attribute value of the current record. Example 26–2 shows the attribute binding for `PrincipalName` created when you drop the attribute from the `CustomerRegistration` collection. Note that the attribute value references the iterator named `CustomerRegistrationIterator`.

Example 26–2 Page Definition Code for an Attribute Binding

```
<bindings>
    ...
  <attributeValues IterBinding="CustomerRegistrationIterator"
                   id="PrincipalName">
    <AttrNames>
      <Item Value="PrincipalName"/>
    </AttrNames>
  </attributeValues>
</bindings>
```

For information regarding the attribute binding element properties, see Appendix B, "Oracle ADF Binding Properties."

26.2.2.3 Using EL Expressions to Bind UI Components

When you create a text field by dropping an attribute from the Data Controls panel, JDeveloper creates the UI component associated with the widget dropped by writing the corresponding tag to the JSF page.

For example, when you drop the `PrincipalName` attribute as an Output Text w/Label widget, JDeveloper inserts the tags for a `panelLabelAndMessage` component and an `outputText` component. It creates an EL expression that binds the `label` attribute of the `panelLabelAndMessage` component to the `label` property of hints created for the `PrincipalName`'s binding. This expression evaluates to the label hint set on the view object (for more information about hints, see Section 5.13, "Defining UI Hints for View Objects"). It creates another expression that binds the `outputText` component's `value` attribute to the `inputValue` property of the `PrincipalName` binding, which evaluates to the value of the `PrincipalName` attribute for the current row. An ID is also automatically generated for both components.

> **Tip:** JDeveloper automatically generates IDs for all ADF Faces components. You can override these values as needed.

Example 26–3 shows the code generated on the JSF page when you drop the `PrincipalName` attribute as an Output Text w/Label widget.

Example 26–3 JSF Page Code for an Attribute Dropped as an Output Text w/Label

```
<af:panelLabelAndMessage label="#{bindings.PrincipalName.hints.label}" id="plam1">
  <af:outputText value="#{bindings.PrincipalName.inputValue}" id="ot1"/>
</af:panelLabelAndMessage>
```

If instead you drop the `PrincipalName` attribute as an Input Text w/Label widget, JDeveloper creates an `inputText` component. As Example 26–4 shows similar to the output text component, the value is bound to the `inputValue` property of the `PrincipalName` binding. Additionally, the following properties are also set:

- `label`: Bound to the `label` property of the control hint set on the object.
- `required`: Bound to the `mandatory` property of the control hint.
- `columns`: Bound to the `displayWidth` property of the control hint, which determines how wide the text box will be.
- `maximumLength`: Bound to the precision property of the control hint. This control hint property determines the maximum number of characters per line that can be entered into the field.

In addition, JDeveloper adds a validator component.

Example 26–4 JSF Page Code for an Attribute Dropped as an Input Text w/Label

```
<af:inputText value="#{bindings.PrincipalName.inputValue}"
              label="#{bindings.PrincipalName.hints.label}"
              required="#{bindings.PrincipalName.hints.mandatory}"
              columns="#{bindings.PrincipalName.hints.displayWidth}"
              maximumLength="#{bindings.PrincipalName.hints.precision}">
              shortDesc="#{bindings.PrincipalName.hints.tooltip}" id="it1">
  <f:validator binding="#{bindings.PrincipalName.validator}"/>
</af:inputText>
```

You can change any of these values to suit your needs. For example, the `mandatory` control hint on the view object is set to `false` by default, which means that the required attribute on the component will evaluate to `false` as well. You can override this value by setting the `required` attribute on the component to `true`. If you decide that all instances of the attribute should be mandatory, then you can change the control hint on the view object, and all instances will then be required. For more information about these properties, see Appendix B, "Oracle ADF Binding Properties."

26.3 Creating a Basic Form

Instead of dropping each of the individual attributes of a collection to create a form, you can a complete form that displays or collects data for all the attributes on an object.

For example, you could create a page that displays basic information about registered users in the StoreFront module by dragging and dropping the `CustomerInfoVO1` collection.

You can also create forms that provide more functionality than simply displaying data from a collection. For information about creating a form that allows a user to update data, see Section 26.5, "Creating a Form to Edit an Existing Record." For information about creating forms that allow users to create a new object for the collection, see Section 26.6, "Creating an Input Form". You can also create search forms. For more information, see Chapter 31, "Creating ADF Databound Search Forms."

26.3.1 How to Create a Form

To create a form using a data control, you bind the UI components to the attributes on the corresponding object in the data control. JDeveloper allows you to do this declaratively by dragging and dropping a collection or a structured attribute from the Data Controls panel.

Before you begin:

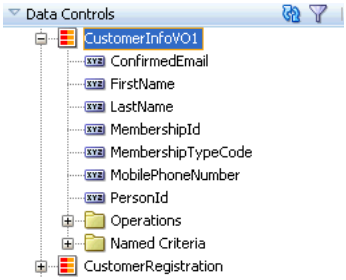
It may be helpful to have an understanding of basic databound forms. For more information, see Section 26.3, "Creating a Basic Form."

You may also find it helpful to understand other ADF functionality and features. For more information, see Section 26.1.2, "Additional Functionality for Databound Forms."

To create a basic form:

1. From the Data Controls panel, select the collection that represents the data you wish to display. Figure 26–5 shows the `CustomerInfoVO1` collection for the `StoreServiceAMDataControl` data control.

Figure 26–5 CustomerInfo View Object in the Data Controls Panel



2. Drag the collection onto the page, and from the context menu choose the type of form that will be used to display or collect data for the object. For a form, you are given the following choices:

   – **ADF Form**: Launches the Edit Form Fields dialog that allows you to select individual attributes instead of having JDeveloper create a field for every attribute by default. It also allows you to select the label and UI component used for each attribute. By default, ADF `inputText` components are used for most attributes. Each `inputText` component has the `label` attribute populated.

     Attributes that are dates use the `InputDate` component. Additionally, if a control type control hint has been created for an attribute, or if the attribute has been configured to be a list, then the component set by the hint is used instead. `InputText` components contain a validator tag that allows you to set up validation for the attribute, and if the attribute is a number or a date, a converter is also included.

     Tip: For more information about validators, converters, and other attributes of the `inputText` component, see the "Using Input Components and Defining Forms" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework.*

   – **ADF Read-Only Form**: Same as the ADF Form, but only read-only `outputText` components are used. Since the form is meant to display data, no `validator` tags are added (converters are included). Attributes of type `Date` use the `outputText` component when in a read-only form. All components are placed inside `panelLabelAndMessage` components, which have the `label` attribute populated, and are placed inside a `panelFormLayout` component.

   – **ADF Search Form**: Creates a form that can be used to execute a Query-by-Example (QBE) search. For more information, see Chapter 31, "Creating ADF Databound Search Forms."

3. In the Edit Form Fields dialog, configure your form.

   You can elect to include navigational controls that allow users to navigate through all the data objects in the collection. For more information, see Section 26.4, "Incorporating Range Navigation into Forms." You can also include a **Submit** button used to submit the form. This button submits the HTML form and applies the data in the form to the bindings as part of the JSF/ADF page lifecycle. For additional help in using the dialog, click **Help**. All UI components are placed inside a `panelFormLayout` component.

4. If you are building a form that allows users to update data, you now need to drag and drop an operation that will perform the update. For more information, see Section 26.5, "Creating a Form to Edit an Existing Record."

26.3.2 What Happens When You Create a Form

Dropping an object as a form from the Data Controls panel has the same effect as dropping a single attribute, except that multiple attribute bindings and associated UI components are created. The attributes on the UI components (such as `value`) are bound to properties on that attribute's binding object (such as `inputValue`) or to the values of control hints set on the corresponding business object. Example 26–5 shows some of the code generated on the JSF page when you drop the `CustomerInfoVO1` collection as a default ADF Form.

Note: If an attribute is marked as hidden on the associated view or entity object, then no corresponding UI is created for it.

Example 26–5 Code on a JSF Page for an Input Form

```
<af:panelFormLayout id="pfl1">
  <af:inputText value="#{bindings.PersonId.inputValue}"
                label="#{bindings.PersonId.hints.label}"
                required="#{bindings.PersonId.hints.mandatory}"
                columns="#{bindings.PersonId.hints.displayWidth}"
                maximumLength="#{bindings.PersonId.hints.precision}"
                shortDesc="#{bindings.PersonId.hints.tooltip}" id="it2">
    <f:validator binding="#{bindings.PersonId.validator}"/>
  </af:inputText>
  <af:inputText value="#{bindings.FirstName.inputValue}"
                label="#{bindings.FirstName.hints.label}"
                required="#{bindings.FirstName.hints.mandatory}"
                columns="#{bindings.FirstName.hints.displayWidth}"
                maximumLength="#{bindings.FirstName.hints.precision}"
                shortDesc="#{bindings.FirstName.hints.tooltip}"
                id="it1">
    <f:validator binding="#{bindings.FirstName.validator}"/>
  </af:inputText>
  <af:inputText value="#{bindings.LastName.inputValue}"
                label="#{bindings.LastName.hints.label}"
                required="#{bindings.LastName.hints.mandatory}"
                columns="#{bindings.LastName.hints.displayWidth}"
                maximumLength="#{bindings.LastName.hints.precision}"
                shortDesc="#{bindings.LastName.hints.tooltip}" id="it7">
    <f:validator binding="#{bindings.LastName.validator}"/>
  </af:inputText>
. . .
</af:panelFormLayout>
```

Note: For information regarding the validator and converter tags, see the "Validating and Converting Input" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework.*

When you choose to create an input form using an object that contains a defined list of values (LOV), then a `selectOneChoice` component is created instead of an `inputText` component. For example, the `PersonsVO` view object contains a defined

LOVs for the `Title`, `Gender`, `MaritalStatusCode`, and `PersonTypeCode` attributes. When you drop the `Persons` data control object as an ADF Form, instead of as an empty input text field, a dropdown list showing all values is created. For more information about how these lists work, see Section 5.12, "Working with List of Values (LOV) in View Object Attributes." For more information about using the lists on a JSF page, see Section 30.3, "Creating a Selection List."

Note: If the object contains a structured attribute (an attribute that is neither a Java primitive type nor a collection), that attribute will not appear in the dialog, and it will not have a corresponding component in the form. You will need to create those fields manually.

26.4 Incorporating Range Navigation into Forms

When you create an ADF Form, if you choose to include navigational controls, JDeveloper includes ADF Faces command components bound to existing navigational logic on the data control. This built-in logic allows the user to navigate through all the data objects in the collection. For example,Figure 26–6 shows a form that would be created if you drag the `CustomerInfoVO1` collection and drop it as an ADF Form that uses navigation.

Figure 26–6 Navigation in a Form

26.4.1 How to Insert Navigation Controls into a Form

By default, when you choose to include navigation when creating a form using the Data Controls panel, JDeveloper creates **First**, **Last**, **Previous**, and **Next** buttons that allow the user to navigate within the collection.

You can also add navigation buttons to an existing form manually.

Before you begin:

It may be helpful to have an understanding of navigation controls. For more information, see Section 26.4, "Incorporating Range Navigation into Forms."

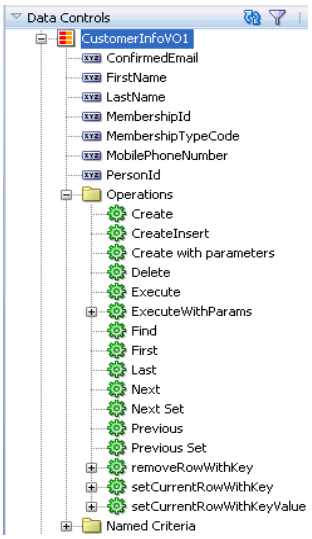
You may also find it helpful to understand other ADF functionality and features. For more information, see Section 26.1.2, "Additional Functionality for Databound Forms."

To manually add navigation buttons:

1. From the Data Controls panel, select the operation associated with the collection of objects on which you wish the operation to execute, and drag it onto the JSF page.

   For example, if you want to navigate through a collection of persons, you would drag the **Next** operation associated with the `Persons` collection. Figure 26–7 shows the operations associated with the `CustomerInfoVO1` collection.

Figure 26–7 Operations Associated with a Collection



2. From the ensuing context menu, choose either **ADF Button** or **ADF Link**.

   Tip: You can also drop the **First**, **Previous**, **Next**, and **Last** buttons at once. To do so, drag the corresponding collection, and from the context menu, choose **Navigation > ADF Navigation Buttons**.

26.4.2 What Happens When You Create Command Buttons

When you drop any operation as a command component, JDeveloper:

- Defines an action binding in the page definition file for the associated operations
- Configures the iterator binding to use partial page rendering for the collection
- Inserts code in the JSF page for the command components

26.4.2.1 Action Bindings for Built-in Navigation Operations

Action bindings execute business logic. For example, they can invoke built-in methods on the action binding object. These built-in methods operate on the iterator or on the data control itself, and are represented as operations in the Data Controls panel. JDeveloper provides navigation operations that allow users to navigate forward, backwards, to the last object in the collection, and to the first object.

Like value bindings, action bindings for operations contain a reference to the iterator binding when the action binding is bound to one of the iterator-level actions, such as `Next` or `Previous`. These types of actions are performed by the iterator, which determines the current object and can therefore determine the correct object to display when a navigation buttons is clicked. Action bindings to other than iterator-level actions, for example for a custom method on an application module, or for the commit or rollback operations, will not contain this reference.

Action bindings use the `RequiresUpdateModel` property, which determines whether or not the model needs to be updated before the action is executed. In the case of navigation operations, by default this property is set to `true`, which means that any changes made at the view layer must be moved to the model before navigation can occur. Example 26–6 shows the action bindings for the navigation operations.

Example 26–6 Page Definition Code for an Operation Action Binding

```
<action IterBinding="CustomerInfoVO1Iterator" id="First"
         RequiresUpdateModel="true" Action="first"/>
<action IterBinding="CustomerInfoVO1Iterator" id="Previous"
        RequiresUpdateModel="true" Action="previous"/>
<action IterBinding="CustomerInfoVO1Iterator" id="Next"
        RequiresUpdateModel="true" Action="next"/>
<action IterBinding="CustomerInfoVO1Iterator" id="Last"
        RequiresUpdateModel="true" Action="last"/>
```

26.4.2.2 Iterator RangeSize Attribute

Iterator bindings have a `rangeSize` attribute that the binding uses to determine the number of data objects to make available for the page for each iteration. This attribute helps in situations when the number of objects in the data source is quite large. Instead of returning all objects, the iterator binding returns only a set number, which then become accessible to the other bindings. Once the iterator reaches the end of the range, it accesses the next set. Example 26–7 shows the default range size for the `CustomerInfoVO` iterator.

Example 26–7 RangeSize Attribute for an Iterator

```
<iterator Binds="CustomerInfoVO" RangeSize="25"
          DataControl="StoreServiceAMDataControl"
          id="CustomerInfoVO1Iterator"
          ChangeEventPolicy="ppr"/>
```

Note: This `rangeSize` attribute is not the same as the `rows` attribute on a table component. For more information, see Table 27–1, " ADF Faces Table Attributes and Populated Values".

By default, the `rangeSize` attribute is set to `25`. This means that a user can view 25 objects, navigating back and forth between them, without needing to access the data source. The iterator keeps track of the current object. Once a user clicks a button that requires a new range (for example, clicking the **Next** button on object number 25), the binding object executes its associated method against the iterator, and the iterator retrieves another set of 25 records. The bindings then work with that set. You can change this setting as needed. You can set it to `-1` to have the full record set returned.

Note: When you create a navigateable form using the Data Controls panel, the `CacheResults` property on the associated iterator is set to `true`. This ensures that the iterator's state, including currency information, is cached between requests, allowing it to determine the current object. If this property is set to `false`, navigation will not work.

Table 26–1 shows the built-in navigation operations provided on data controls and the result of invoking the operation or executing an event bound to the operation. For more information about action events, see Section 26.4.3, "What Happens at Runtime: How Action Events and Action Listeners Work."

Table 26–1 Built-in Navigation Operations

Operation	When invoked, the associated iterator binding will...
First	Move its current pointer to the beginning of the result set.
Last	Move its current pointer to the end of the result set.
Previous	Move its current pointer to the preceding object in the result set. If this object is outside the current range, the range is scrolled backward a number of objects equal to the range size.
Next	Move its current pointer to the next object in the result set. If this object is outside the current range, the range is scrolled forward a number of objects equal to the range size.
Previous Set	Move the range backward a number of objects equal to the range size attribute.
Next Set	Move the range forward a number of objects equal to the range size attribute.

26.4.2.3 EL Expressions Used to Bind to Navigation Operations

When you create command components using navigation operations, the command components are placed in a `panelGroupLayout` component. JDeveloper creates an EL expression that binds a navigational command button's `actionListener` attribute to the `execute` property of the action binding for the given operation.

At runtime an action binding will be an instance of the `FacesCtrlActionBinding` class, which extends the core `JUCtrlActionBinding` implementation class. The `FacesCtrlActionBinding` class adds the following methods:

- `public void execute(ActionEvent event)`: This is the method that is referenced in the `actionListener` property, for example `#{bindings.First.execute}`.

  This expression causes the binding's operation to be invoked on the iterator when a user clicks the button. For example, the **First** command button's `actionListener` attribute is bound to the `execute` method on the `First` action binding.

- `public String outcome()`: This can be referenced in an `Action` property, for example `#{bindings.Next.outcome}`.

  This can be used for the result of a method action binding (once converted to a `String`) as a JSF navigation outcome to determine the next page to navigate to.

  Note: Using the outcome method on the action binding implies tying the view-controller layer too tightly to the model, so it should be rarely used.

Every action binding for an operation has an `enabled` boolean property that Oracle ADF sets to `false` when the operation should not be invoked. By default, JDeveloper binds the UI component's `disabled` attribute to this value to determine whether or not the component should be enabled. For example, the UI component for the **First** button has the following as the value for its `disabled` attribute:

```
#{!bindings.First.enabled}
```

This expression evaluates to `true` whenever the binding is not enabled, that is, when operation should not be invoked, thereby disabling the button. In this example,

because the framework will set the `enabled` property on the binding to `false` whenever the first record is being shown, the **First** button will automatically be disabled because its `disabled` attribute is set to be `true` whenever `enabled` is `False`. For more information about the `enabled` property, see Appendix B, "Oracle ADF Binding Properties."

Example 26–8 shows the code generated on the JSF page for navigation operation buttons. For more information about the `partialSubmit` attribute on the button, see the "Enabling Partial Page Rendering Declaratively" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Example 26–8 JSF Code for Navigation Buttons Bound to ADF Operations

```
<f:facet name="footer">
   <af:panelGroupLayout>
     <af:commandButton actionListener="#{bindings.First.execute}"
                       text="First"
                       disabled="#{!bindings.First.enabled}"
                       partialSubmit="true" id="cb1"/>
     <af:commandButton actionListener="#{bindings.Previous.execute}"
                       text="Previous"
                       disabled="#{!bindings.Previous.enabled}"
                       partialSubmit="true" id="cb2"/>
     <af:commandButton actionListener="#{bindings.Next.execute}"
                       text="Next"
                       disabled="#{!bindings.Next.enabled}"
                       partialSubmit="true" id="cb3"/>
     <af:commandButton actionListener="#{bindings.Last.execute}"
                       text="Last"
                       disabled="#{!bindings.Last.enabled}"
                       partialSubmit="true" id="cb4"/>
   </af:panelGroupLayoutr>
 </f:facet>
```

26.4.3 What Happens at Runtime: How Action Events and Action Listeners Work

An action event occurs when a command component is activated. For example, when a user clicks a **Submit** button, the form is submitted and, subsequently, an action event is fired. Action events might affect only the user interface (for example, a link to change the locale, causing different field prompts to display), or they might involve some logic processing in the back end (for example, a button to navigate to the next record).

An action listener is a class that wants to be notified when a command component fires an action event. An action listener contains an action listener method that processes the action event object passed to it by the command component.

In the case of the navigation operations, when a user clicks, for example, the **Next** button, an action event is fired. This event object stores currency information about the current data object, taken from the iterator. Because the component's `actionListener` attribute is bound to the `execute` method of the `Next` action binding, the `Next` operation is invoked when the event fires. This method takes the currency information passed in the event object to determine what the next data object should be.

In addition, when a user clicks a navigation button, only those components associated with the same iterator as the button's action binding are processed through the

lifecycle. For more information, see Section 25.2.1, "What You May Need to Know About Partial Page Rendering and Iterator Bindings."

26.4.4 What You May Need to Know About the Browser Back Button and Navigating Through Records

You must use the navigation buttons to navigate through the records displayed in a form; you cannot use the browser's back or forward buttons. Because navigation forms automatically use PPR, only part of the page goes through the lifecycle, meaning that when you click a navigation button, the components displaying the data are refreshed and display new data, and you actually remain on the same page. Therefore, when you click the browser's back button, you will be returned to the page that was rendered before the page with the form, instead of to the previous record displayed in the form.

For example, say you are on a page that contains a link to view all current orders. When you click the link, you navigate to a page with a form and the first order, Order #101, is displayed. You then click **Next** and Order #102 is displayed. You click **Next** again, and Order #103 is displayed. If you click the browser's back button, you will not be shown Order #102. Instead, you will be returned to the page that contained the link to view all current orders.

26.5 Creating a Form to Edit an Existing Record

You can create a form that allows a user to edit the current data, and then commit those changes to the data source. To do this, you use operations that can modify data records associated with the collection or the data control itself to create command buttons. For example, you can use the `Delete` operation to create a button that allows a user to delete a record from the current range. Or you can use the built-in **Submit** button to submit changes.

> **Tip:** While you can use the `Create` operation on a form to create a new object, using the ADF Creation Form instead provides additional built-in functionality. See Section 26.6, "Creating an Input Form" for more information.

It is important to note that these operations are executed only against objects in the ADF cache. You need to use the `Commit` operation on the root data control to actually commit any changes to the data source. You use the data control's `Rollback` operation to roll back any changes made to the cached object. If the page is part of a transaction within a bounded task flow, you would most likely use these operations to resolve the transaction in a task flow return activity. For more information, see Section 22.2, "Managing Transactions in Task Flows."

26.5.1 How to Create Edit Forms

To use the operations on a form, you follow the same procedures as with the navigation operations.

Before you begin:

It may be helpful to have an understanding of creating edit forms. For more information, see Section 26.5, "Creating a Form to Edit an Existing Record."

You may also find it helpful to understand other ADF functionality and features. For more information, see Section 26.1.2, "Additional Functionality for Databound Forms."

To create an edit form:

1. From the Data Controls panel, drag the collection for which you wish to create the form, and select **ADF Form** from the context menu.

   This creates a form using `inputText` components, which will allow the user to edit the data in the fields.

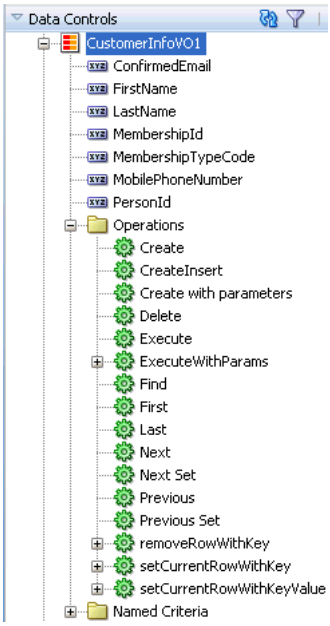
2. In the Edit Form Fields dialog, select **Include Submit Button** and click **OK**.

3. You can either create new buttons for the operations you want to include, or you can rebind the **Submit** button so that it invokes another operation. To keep the **Submit** button as is and create new buttons for the operations, continue with this step. To rebind the **Submit** button, see Step 5.

   From the Data Controls panel, select the operation associated with the collection of objects on which you wish the operation to execute, and drag it onto the JSF page. If you simply want to be able to edit the data, then the **Submit** button is all that is required.

   For example, if you want to be able to delete a customer record, you would drag the `Delete` operation associated with the `CustomerInfoVO1` collection. Figure 26–8 shows the operations associated with a collection.

Figure 26–8 Operations Associated with a Collection



4. From the ensuing context menu, choose either **ADF Button** or **ADF Link**.

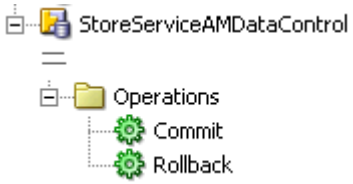
5. To rebind the Submit button, right-click the button in the Structure window, and choose **Bind to ADF Control**. In the Bind to ADF Control dialog, select the operation to which you want the button bound. Ensure that the operation you select is associated with the collection on which the form is based.

   For example, if you want to be able to delete a customer record, you would select the `Delete` operation associated with the `CustomerInfoVO1` collection. Figure 26–8 shows the operations associated with a collection.

6. If the page is not part of a transaction within a bounded task flow, then you need to create buttons that allow the user either to commit or roll back the changes. From the Data Controls panel, drag the **Commit** and **Rollback** operations

associated with the root-level data control, and drop them as either a command button or a command link. Figure 26–9 shows the commit and rollback operations for the `StoreServiceAMDataControl` data control (note that for viewing simplicity, the figure omits details in the tree that appear for each view object).

Figure 26–9 Commit and Rollback Operations for a Data Control



If the page is part of a transaction within a bounded task flow, then you can simply enter `Commit` and `Rollback` as the values for the transaction resolution when creating the task flow return activity. For more information, see Section 22.2, "Managing Transactions in Task Flows."

26.5.2 What Happens When You Use Built-in Operations to Change Data

Dropping any data control operation as a command button causes the same events as does dropping navigation operations. For more information, see Section 26.4.2, "What Happens When You Create Command Buttons."

The only difference is that the action bindings for the `Commit` and `Rollback` operations do not require a reference to the iterator, because they execute a method on the application module (the data control itself), as opposed to the iterator. Note that the `Rollback` action has the `RequiresUpdateModel` property set to `false`. This is because the model should not be updated before the operation is executed, since all changes need to be discarded. Example 26–9 shows the action bindings generated in the page definition file for these operations.

Example 26–9 Action Bindings for Commit and Rollback Operations

```
<action id="Commit" RequiresUpdateModel="true" Action="commitTransaction"
        DataControl="StoreServiceAMDataControl"/>
<action id="Rollback" RequiresUpdateModel="false"
        Action="rollbackTransaction"
        DataControl="StoreServiceAMDataControl"/>
```

Table 26–2 shows the built-in non-navigation operations provided on data controls and data control objects, along with the result of invoking the operation or executing an event bound to the operation. For more information about action events, see Section 26.4.3, "What Happens at Runtime: How Action Events and Action Listeners Work."

Table 26–2 More Built-in Operations

Operation	When invoked, the associated iterator binding will...
`CreateInsert`	Creates a row directly before the current row, inserts the new record into the row set, then moves the current row pointer to the new row. Note that the range does not move, meaning that the last row in the range may now be excluded from the range. For more information about using the `CreateInsert` operation to create objects, see Section 26.6, "Creating an Input Form."

Table 26–2 (Cont.) More Built-in Operations

Operation	When invoked, the associated iterator binding will...
`Create`	Creates a row directly before the current row, then moves the current row pointer to the new row. Note that the range does not move, meaning that the last row in the range may now be excluded from the range. Also note that the record will not be inserted into the row set, preventing a blank row should the user navigate away without actually creating data. The new row will be created when the user submits the data. For more information, see Section 27.4.5, "What You May Need to Know About Create and CreateInsert."
`CreateWith Parameters`	Same as the `CreateInsert` operation (the new record is inserted into the row set), however uses named parameters to create the object.
`Delete`	Deletes the current row from the cache and moves the current row pointer to the next row in the result set. Note that the range does not move, meaning that a row may be added to the end of the range. If the last row is deleted, the current row pointer moves to the preceding row. If there are no more rows in the collection, the enabled attribute is set to `disabled`.
`RemoveRowWithKey`	Uses the row key as a `String` converted from the value specified by the input field to remove the data object in the bound data collection.
`SetCurrentRowWith Key`	Sets the row key as a `String` converted from the value specified by the input field. The row key is used to set the currency of the data object in the bound data collection. For an example of when this is used, see Section 27.2.3, "What You May Need to Know About Setting the Current Row in a Table."
`SetCurrentRowWith KeyValue`	Sets the current object on the iterator, given a key's value. For more information, see Section 27.2.3, "What You May Need to Know About Setting the Current Row in a Table."
`ExecuteWithParams`	Refreshes the data collection by first assigning new values to the named bind variables passed as parameters, then (re)executing the view object's query. You would use this operation in the same manner as you would use the `CreateInsert` operation to create an input form. For more information, see Section 26.6, "Creating an Input Form." This operation appears only for view objects that have defined one or more named bind variables at design time. For more information, see Section 5.10, "Working with Bind Variables.".
`Commit`	Causes all items currently in the cache to be committed to the database.
`Rollback`	Clears the cache and returns the transaction and iterator to the initial state. Resets the `ActionListener` method.
`Execute` and `Find`	These operations are used only in search forms. See Chapter 31, "Creating ADF Databound Search Forms" for more information.

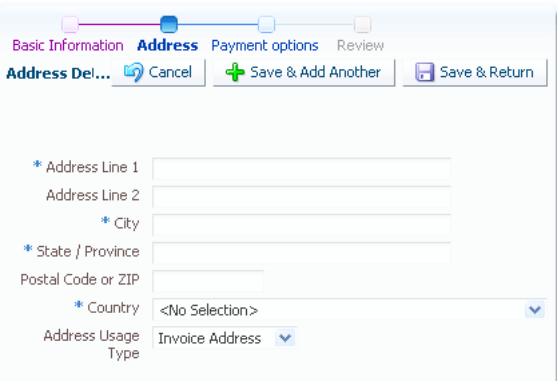
26.6 Creating an Input Form

You can create a form that allows a user to enter information for a new record and then commit that record to the data source. You need to use a task flow that contains a method activity that will call the `CreateInsert` operation before the page with the input form is displayed. This method activity causes a blank row to be inserted into the row set which the user can then populate using a form.

For example, in the StoreFront module, the `customer-registration-task-flow` task flow contains the `createAddress` method activity, which calls the `CreateInsert` operation on the `CustomerAddress` view object. Control is then

passed to the `addressDetails` view activity, which displays a form where the user can enter a new address, as shown in Figure 26–10.

Figure 26–10 Create an Address



Note: If your application does not use task flows, then the calling page should invoke the `createInsert` operation similar to the way in which a task flow's method activity would. For example, you could provide application logic within an event handler associated with a command button on the calling page.

26.6.1 How to Create an Input Form Using a Task Flow

Instead of using command components to create new objects, you use a method activity in a task flow.

Before you begin:

Before you create the input form, you need to create a bounded task flow that will contain both the form and the method activity that will execute the `CreateInsert` operation. For more information, see Section 18.2, "Creating a Task Flow."

It may be helpful to have an understanding of input forms. For more information, see Section 26.6, "Creating an Input Form."

You may also find it helpful to understand other ADF functionality and features. For more information, see Section 26.1.2, "Additional Functionality for Databound Forms."

To create an input form:

1. To the bounded task flow, add a method call activity. Have this activity execute the `CreateInsert` operation associated with the collection for which you are creating the form. For procedures on using method activities, see Section 19.5, "Using Method Call Activities."
2. In the Property Inspector, enter a string for the **fixed-outcome** property. For example, the `createAddress` method activity in the `customer-registration-task-flow` task flow has `editAddress` as the `fixed-outcome` value.
3. Add a view activity that represents the page for the input form. For information on adding view activities, see Section 18.3, "Adding Activities to a Task Flow."

4. Add a control flow case from the method activity to the view activity. In the Property Inspector, enter the value of the **fixed-outcome** property of the method activity set in Step 2 as the value of the `from-outcome` of the control flow case.
5. Open the page for the view activity in the design editor, and from the Data Controls panel, drag the collection for which the form will be used to create a new record, and choose **ADF Form** from the context menu.

   Tip: If you want the user to be able to create multiple entries before committing to the database, do the following:

   1. In the task flow, add another control flow case from the view activity back to the method activity, and enter a value for the `from-outcome` method. For example, you might enter `createAnother`.
   2. Drag and drop a command component from the Component Palette onto the page, and set the `action` attribute to the `from-outcome` just created. This will cause the task flow to return to the method activity and reinvoke the `CreateInsert` operation.

6. Because you need to commit the new data, the application needs to execute the `commit` operation of the data control. To do this, you can add a button that navigates to a return activity that calls the `commit` operation. For procedures for using a return activity, see Section 19.7, "Using Task Flow Return Activities."

   Best Practice: Use the return activity unless the task flow contains data managed by more than one data control, or you need to commit the data before the end of the flow. In those cases, you can add a button to the page bound to the commit button.

   If you need to add a commit button to the page, do the following:

   1. In the Data Controls panel, drag the commit operation associated with the data control that contains the collection associated with the input form, and drop it as a command button.
   2. In the Structure window, select the command button for the commit operation.
   3. In the Property Inspector, set the action to the outcome String that will navigate back to the method activity. You then need to add a control flow case from the page back to the activity, using the same outcome value.
   4. Set the command button's `disabled` property to `false`.

      By default, JDeveloper binds the `disabled` attribute of the button to the `enabled` property of the binding, causing the button to be disabled when the `enabled` property is set to `false`. For this binding, the `enabled` property is `false` until an update has been posted. For the purposes of an input form, the button should always be enabled, since there will be no changes posted before the user needs to create the new object.

26.6.2 What Happens When You Create an Input Form Using a Task Flow

When you use an ADF Form to create an input form, JDeveloper:

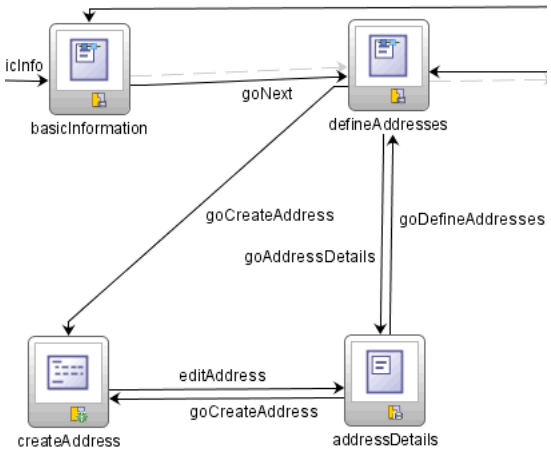
- Creates an iterator binding for the collection and an action binding for the `CreateInsert` operation in the page definition for the method activity. The `CreateInsert` operation is responsible for creating a row in the row set and populating the data source with the entered data. In the page definition for the page, JDeveloper creates an iterator binding for the collection and attribute bindings for each of the attributes of the object in the collection, as for any other

form. If you created command buttons or links using the `Commit` and `Rollback` operations, JDeveloper also creates an action bindings for those operations.

- Inserts code in the JSF page for the form using ADF Faces `inputText` components, and in the case of the operations, `commandButton` components.

For example, the StoreFront module contains a page that displays all the addresses for a customer in a table. The table includes an **Add** button that navigates to a form where you can input data for a new address. Once the address is created, you return to the page with the table and the new address is displayed. Figure 26–11 shows the `customer-registration-task-flow` task flow with the `createAddress` method activity.

Figure 26–11 Task Flow for an Input Form



Example 26–10 shows the page definition file for the method activity.

Example 26–10 Page Definition Code for a Creation Method Activity

```
<executables>
  <iterator id="CustomerAddressIterator" RangeSize="25"
            Binds="CustomerAddress" DataControl="StoreServiceAMDataControl"/>
</executables>
<bindings>
  <action id="CreateInsert" IterBinding="CustomerAddressIterator"
          InstanceName="StoreServiceAMDataControl.CustomerAddress"
          DataControl="StoreServiceAMDataControl" RequiresUpdateModel="true"
          Action="createInsertRow"/>
</bindings>
```

26.6.3 What Happens at Runtime: CreateInsert Action from the Method Activity

When the `createMethodCall` activity is accessed, the `CreateInsert` action binding is invoked, which executes the `CreateInsertRow` operation, and a new blank instance for the collection is created. Note that during routing from the method activity to the view activity, the method activity's binding container skips validation for required attributes, allowing the blank instance to be displayed in the form on the page.

26.6.4 What You May Need to Know About Displaying Sequence Numbers

Because the `Create` action is executed before the page is displayed, if you are populating the primary key using sequences, the next number in the sequence will appear in the input text field, unlike the rest of the fields, which are blank. The sequence number is displayed because the associated entity class contains a method that uses an eager fetch to generate a sequence of numbers for the primary key attribute. The eager fetch populates the value as the row is created. Therefore, using sequences works as expected with input forms.

However, if instead you've configured the attribute's type to `DBSequence` (which uses a database trigger to generate the sequence), the number would not be populated until the object is committed to the database. In this case, the user would see a negative number as a placeholder. To avoid this, you can use the following EL expression for the `Rendered` attribute of the input text field:

```
#{bindings.EmployeeId.inputValue.value > 0}
```

This expression will display the component only when the value is greater than zero, which will not be the case before it is committed. Similarly, you can simply set the `Rendered` attribute to `false`. However, then the page will never display the input text field component for the primary key.

26.7 Using a Dynamic Form to Determine Data to Display at Runtime

ADF Faces offers a library of dynamic components that includes dynamic form and dynamic table widgets that you can drop from the Data Controls panel. Dynamic components differ from standard components in that all the binding metadata is created at runtime. This dynamic building of the bindings allows you set display information using control hints on a view object instead of configuring the information in the Edit Form Fields dialog as you drop the control onto the page. Then if you want to change how the data displays, you need only change it on the view object, and all dynamic components bound to that view object will change their display accordingly. With standard components, if you want to change any display attributes (such as the order or grouping of the attributes) you would need to change each page on which the data is displayed.

For example, in the StoreFront module, you could set the Category and Field Order attribute hints on the `CustomerInfoVO` view object that groups the `FirstName` and `LastName` attributes together and at the top of a form (or at the leftmost columns of a table), the `ConfirmedEmail` and `MobilePhoneNumber` together and at the bottom of a form (or the rightmost columns of a table), and the `MembershipID` and `MembershipType` together and at the middle of a form or table. You could make it so that the `PersonId` does not display at all. For more information about control hints on view objects, see Section 5.13, "Defining UI Hints for View Objects."

Figure 26–12 shows a dynamic form at runtime created by dragging and dropping the `CustomerInfoVO` view object with control hints set as described previously, as a dynamic form. Note that the input fields are grayed out because the view object is nonupdateable.

Figure 26–12 Dynamic Form Displays Based on Hints Set on the View Object

26.7.1 How to Use Dynamic Forms

To use dynamic forms you first need to set control hints (especially the order and grouping hints) on any corresponding view objects. Next you import the libraries for the dynamic components. You can then drop the dynamic form or table widgets onto your page.

Before you begin:

It may be helpful to have an understanding of dynamic forms. For more information, see Section 26.7, "Using a Dynamic Form to Determine Data to Display at Runtime."

You may also find it helpful to understand other ADF functionality and features. For more information, see Section 26.1.2, "Additional Functionality for Databound Forms."

To use dynamic components:

1. Set UI hints on the corresponding view objects. For the Category hint, enter a string that can be used to group attributes together. For example, in the `CustomerInfoVO` hints, the `FirstName` and `LastName` attributes both have `name` as the value for the Category UI hint. The Field Order hint determines the order the attributes are displayed within a category. For example, in the `CustomerInfoVO` hints, the `FirstName` attribute has a Field Order value of `1` and the `LastName` attribute has a Field Order value of `2`.

   For procedures on creating UI hints, see Section 5.13, "Defining UI Hints for View Objects."

2. If not already included, import the dynamic component library.

   1. In the Application Navigator, right-click the view project in which the dynamic components will be used, and from the context menu, choose **Project Properties**.
   2. In the tree, select **JSP Tag Libraries**.
   3. On the JSP Tag Libraries page, click **Add**.
   4. In the Choose Tag Libraries dialog, select **ADF Dynamic Components**, and click **OK**.
   5. On the JSP Tag Libraries page, click **OK**.

3. From the Data Controls panel, select the collection that represents the view object.
4. Drag the collection onto the page, and from the context menu, choose **Forms > ADF Dynamic Form**.

   Tip: If dynamic components are not listed, then the library was not imported into the project. Repeat Step 2.

5. In the Property Inspector, enter the following: for the **Category** field:

- **Category**: Enter the string used as the value for the Category UI hint for the first group you'd like to display in your form. For example, in Figure 26–12, the Category value would be `name`.
- **Editable**: Enter `true` if you want the data to be editable (the default). Enter `false` if the data should be read-only.

6. Repeat Steps 4 and 5 for each group that you want to display on the form. For example, the form in Figure 26–12 is actually made up of three different forms: one for the category `name`, one for the category `membership`, and one for the category `contact`.

26.7.2 What Happens When You Use Dynamic Components

When you drop a dynamic form, only a binding to the iterator is created. Example 26–11 shows the page definition for a page that contains one dynamic form component created by dropping the `CustomerInfoVO` collection. Note that no attribute bindings are created.

Example 26–11 Page Definition Code for a Dynamic Form

```
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="11.1.1.53.2" id="DynamicFormPageDef"
                Package="package.pageDefs">
  <parameters/>
  <executables>
    <iterator Binds="CustomerInfoVO1" RangeSize="25"
              DataControl="StoreServiceAMDataControl"
              id="CustomerInfoVO1Iterator"/>
  </executables>
  <bindings/>
</pageDefinition>
```

JDeveloper inserts a a `form` tag which contains a dynamic form tag for each of the forms dropped. The form tag's value is bound to the iterator binding, as shown in Example 26–12. This binding means the entire form is bound to the data returned by the iterator. You cannot set display properties for each attribute individuality, nor can you rearrange attributes directly on the JSF page.

Example 26–12 JSF Page Code for a Dynamic Form

```
<af:document>
  <af:messages/>
  <af:form>
    <dynamic:form value="#{bindings.CustomerInfoVO1Iterator}"
                  category="name" id="f1"/>
    <dynamic:form value="#{bindings.CustomerInfoVO1Iterator}"
                  category="member" id="f2"/>
    <dynamic:form value="#{bindings.CustomerInfoVO1Iterator}"
                  category="contact" id="f3"/>
  </af:form>
</af:document>
```

Tip: You can set certain properties that affect the functionality of the form. For example, you can make a form available for upload, set the rendered property, or set a partial trigger. To do this, select the `af:form` tag in the Structure window, and set the needed properties using the Property Inspector.

26.7.3 What Happens at Runtime: How Attribute Values Are Dynamically Determined

When a page with dynamic components is rendered, the bindings are created just as they are when items are dropped from the Data Controls panel at design time, except that they are created at runtime. For more information, see Section 26.3.2, "What Happens When You Create a Form."

> **Tip:** While there is a slight performance hit because the bindings need to be created at runtime, there is also a performance gain because the JSF pages do not need to be regenerated and recompiled when the structure of the view object changes.

26.8 Modifying the UI Components and Bindings on a Form

Once you use the Data Controls panel to create any type of form (except a dynamic form), you can then delete attributes, change the order in which they are displayed, change the component used to display data, and change the attribute to which the components are bound.

> **Note:** You cannot change how a dynamic form displays using the following procedures. You must change display information on the view object or entity object instead.

26.8.1 How to Modify the UI Components and Bindings

You can modify certain aspects of the default components dropped from the Data Controls panel. You can use the Structure window to change the order in which components are displayed, to add new components or change existing components, or to delete components. You can use the Property Inspector to change or delete bindings, or to change the label displayed for a component.

To modify default components and bindings:

1. Use the Structure window to do the following:
   - Change the order of the UI components by dragging them up or down the tree. A black line with an arrowhead denotes where the UI component will be placed.
   - Add a UI component. Right-click an existing UI component in the Structure window and choose to place the new component before, after, or inside the selected component. You then choose from a list of UI components.
   - Bind a UI component. Right-click an existing UI component in the Structure window and choose **Bind to ADF Control**. You can then select the object to which you want your component bound.
   - Rebind a UI component. Right-click an existing UI component in the Structure window and choose **Rebind to another ADF Control**. You can then select the new control object to which you want your component bound.
   - Delete a UI component. Right-click the component and choose **Delete**. If you wish to keep the component, but delete the binding, you need to use the Property Inspector. See the second bullet point in Step 2.
2. With the UI component selected in the Structure window, you can then do the following in the Property Inspector:

- Add a non-ADF binding for the UI component. Enter an EL expression in the **Value** field, or use the dropdown menu and choose **Edit**.
- Delete a binding for the UI component by deleting the EL expression.
- Change the label for the UI component. By default, the label is bound to the binding's `label` property of its hint. This property allows your page to use the UI control hints for labels that you have defined for your entity object attributes or view object attributes. The UI hints allow you to change the value once and have it appear the same on all pages that display the label.

  You can change the label just for the current page. To do so, select the `label` attribute. You can enter text or an EL expression to bind the label value to something else, for example, a key in a properties or resource file.

  For example, the `inputText` component used to display the name of a product might have the following for its `Label` attribute:

  ```
  #{bindings.ProductName.hints.label}
  ```

  However, you could change the expression to instead bind to a key in a properties file, for example:

  ```
  #{properties['productName']}
  ```

  In this example, `properties` is a variable defined in the JSF page used to load a properties file.

26.8.2 What Happens When You Modify Attributes and Bindings

When you modify how an attribute is displayed by moving or changing the UI component, JDeveloper changes the corresponding code on the JSF page. When you use the binding editors to add or change a binding, JDeveloper adds the code to the JSF page, and also adds the appropriate elements to the page definition file.

27

Creating ADF Databound Tables

This chapter describes how to use the Data Controls panel to create data bound tables using ADF Faces components.

This chapter includes the following sections:

- Section 27.1, "About Creating ADF Databound Tables"
- Section 27.2, "Creating a Basic Table"
- Section 27.3, "Creating an Editable Table"
- Section 27.4, "Creating an Input Table"
- Section 27.5, "Modifying the Attributes Displayed in the Table"

27.1 About Creating ADF Databound Tables

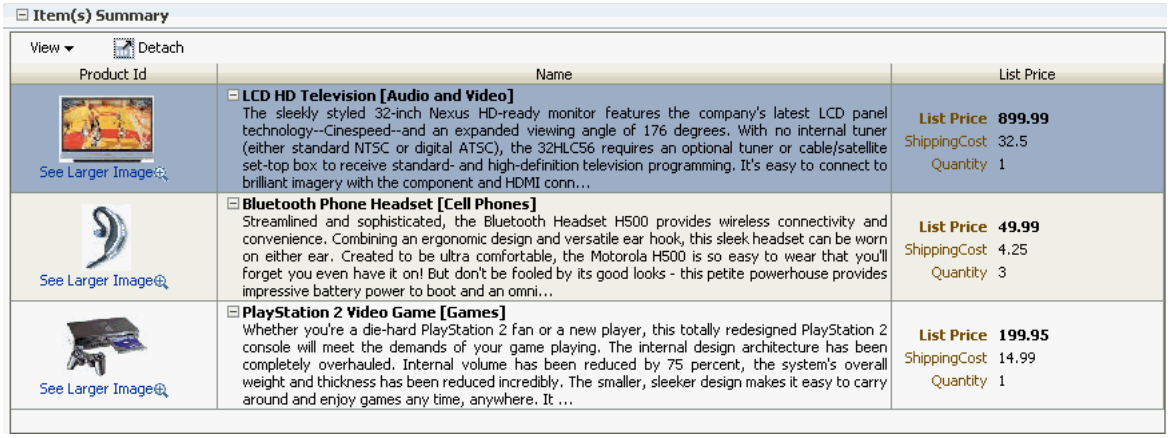
Unlike forms, tables allow you to display more than one data object from a collection at a time.

You can create tables that simply display data, or you can create tables that allow you to edit or create data. Once you drop a collection as a table, you can add command buttons bound to actions that execute some logic on a selected row. You can also modify the default components to suit your needs.

27.1.1 ADF Databound Tables Use Cases and Examples

Figure 27–1 shows the Items Ordered tab of the My Orders page in the StoreFront module application, which uses a browse table to display the items for a given order.

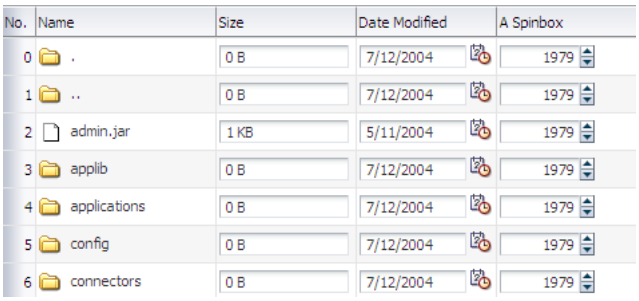
Figure 27–1 The Orders Table



Unlike a form that may display information for one row, a table displays information for all (or a set) of rows.

You can create a table that simply displays information, as shown in Figure 27–1, or you can allow the user to edit the information, as shown in Figure 27–2.

Figure 27–2 Edit Table



You can also use a table with a button to create new records, which are then inserted into the data collection.

27.1.2 Additional Functionality for Databound Tables

You may find it helpful to understand other ADF Faces features before you implement your table and tree components. Additionally, once you have added a tree or table component to your page, you may find that you need to add functionality such as validation and accessibility. Following are links to other functionality that table and tree components can use.

- **ADF view objects**: Much of how the components display and function in the table is controlled by the corresponding view objects. For more information, see Chapter 5, "Defining SQL Queries Using View Objects."
- **ADF Model and data binding**: When you create tables in an ADF Web application, you use ADF Model and data binding. For more information, see Appendix 13, "Using ADF Model in a Fusion Web Application."
- **ADF Faces**: For detailed information about developing with ADF Faces, see the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

- **Command buttons**: You may want to add command button or links that invoke some functionality against the data set. For more advanced functionality than what is covered in this chapter, see Chapter 28, "Command Components to Invoke Functionality in the View Layer."
- **Filtered tables**: You can create tables that provide filtering, in effect executing a query-by-example search against the data in the table. For more information, see Section 31.5, "Creating Standalone Filtered Search Tables from Named View Criteria."
- **Task flows**: If your table takes part in a transaction, for example an input table, then you may need to use an ADF task flow to invoke certain operations before or after the table is rendered. For more information, see Part IV, "Creating ADF Task Flows".
- **Validation**: You may want certain fields to be validated before they are submitted to the data store. For more information, see Appendix 7, "Defining Validation and Business Rules Declaratively" and Appendix 16, "Using Validation in the ADF Model Layer."
- **Active data**: If your application uses active data, then you can have the data in your tables and trees update automatically, whenever the data in the data source changes. For more information, see Chapter 45, "Using the Active Data Service."

Note: If you wish to use active data, and your application uses ADF Business Components, then your tables must conform to the following:

- The binding represents homogeneous data (that is, only one rule), although an accessor can still access a like accessor.
- The binding rule contains a single attribute.
- The table does not use filtering.
- The tree component's `nodeStamp` facet contains a single `outputText` tag and contains no other tags.

27.2 Creating a Basic Table

Unlike with forms where you bind the individual UI components that make up a form to the individual attributes on the collection, with a table you bind the ADF Faces `table` component to the complete collection or to a range of *n* data objects at a time from the collection. The individual components used to display the data in the columns are then bound to the attributes. The iterator binding handles displaying the correct data for each object, while the `table` component handles displaying each object in a row. JDeveloper allows you to do this declaratively, so that you don't need to write any code.

27.2.1 How to Create a Basic Table

To create a table using a data control, you bind the `table` component to a collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel.

Tip: You can also create a table by dragging a table component from the Component Palette.

Before you begin:

It may be helpful to have an understanding of ADF Faces databound tables. For more information, see Section 27.2, "Creating a Basic Table."

You may also find it helpful to understand other ADF functionality and features. For more information, see Section 27.1.2, "Additional Functionality for Databound Tables."

To create a databound table:

1. From the Data Controls panel, select a collection.

   For example, to create a simple table in the StoreFront module that displays products in the system, you would select the `Products` collection.

2. Drag the collection onto a JSF page, and from the context menu, choose the appropriate table.

   When you drag the collection, you can choose from the following types of tables:

   - **ADF Table**: Allows you to select the specific attributes you wish your editable table columns to display, and what UI components to use to display the data. By default, ADF `inputText` components are used for most attributes, thus enabling the table to be editable. Attributes that are dates use the `inputDate` component. Additionally, if a control type control hint has been created for an attribute, or if the attribute has been configured to be a list, then the component set by the hint is used instead.
   - **ADF Read-Only Table:** Same as the **ADF Table**; however, each attribute is displayed in an `outputText` component.
   - **ADF Dynamic and ADF Read-Only Dynamic Table**: Allows you to create a table when the attributes returned and displayed are determined dynamically at runtime. This component is helpful when the attributes for the corresponding object are not known until runtime, or you do not wish to hardcode the column names in the JSF page. For more information about working with dynamic components, see Section 26.7, "Using a Dynamic Form to Determine Data to Display at Runtime."

3. The Edit Table Columns dialog shows each attribute in the collection, and allows you to determine how these attributes will behave and appear as columns in your table.

   Note: If the collection contains a structured attribute (an attribute that is neither a Java primitive type nor a collection), the attributes of the structured attributes will also appear in the dialog.

   Using this dialog, you can do the following:

   - Allow the ADF Model layer to handle selection by selecting the **Row Selection** checkbox. Selecting this option means that the iterator binding will access the iterator to determine the selected row or rows. Select this option unless you do not want the table to allow selection.
   - Allow the ADF Model layer to handle column sorting by selecting the **Enabling Sorting** checkbox. Selecting this option means that the iterator binding will access the iterator, which will perform an order-by query to determine the order. Select this option unless you do not want to allow column sorting.

- Allow the columns in the table to be filtered using entered criteria by selecting the **Enabling Filtering** checkbox. Selecting this option allows the user to enter criteria in text fields above each column. That criteria is then used to build a Query-by-Example (QBE) search on the collection, so that the table will display only the results returned by the query. For more information, see Section 31.5, "Creating Standalone Filtered Search Tables from Named View Criteria."
- Group columns for selected attributes together under a parent column, by selecting the desired attributes (shown as rows in the dialog), and clicking the **Group** button. Figure 27–3 shows how three grouped columns appear in the visual editor after the table is created.

Figure 27–3 Grouped Columns in an ADF Faces Table

Group			Name
Product Id	Supplier Id	Category Id	
#{...ProductId}	#{...SupplierId}	#{...CategoryId}	#{...ProductName}
#{...ProductId}	#{...SupplierId}	#{...CategoryId}	#{...ProductName}
#{...ProductId}	#{...SupplierId}	#{...CategoryId}	#{...ProductName}

- Change the display label for a column. By default, the label is bound to the `labels` property for any control hint defined for the attribute on the table binding. This binding allows you to change the value of a label text once on the view object, and have the change appear the same on all pages that display the label.

  Instead of using this default, you can enter text or an EL expression to bind the label value to something else, for example, a key in a resource file.
- Change the value binding for a column. You can change the column to be bound to a different attribute. If you simply want to rearrange the columns, you should use the order buttons. If you do change the attribute binding for a column, the label for the column also changes.
- Change the UI component used to display an attribute. The UI components are set based on the table you selected when you dropped the collection onto the page, on the type of the corresponding attribute (for example, `inputDate` components are used for attributes that are dates), and on whether or not default components were set as control hints on the corresponding view object. You can change to another component using the dropdown menu.

  Tip: If one of the attributes for your table is also a primary key, you may want to choose a UI component that will not allow a user to change the value.

  Tip: If you want to use a component that is not listed in the dropdown menu, use this dialog to select the `outputText` component, and then manually add the other tag to the page.
- Change the order of the columns using the order buttons.
- Add a column using the **Add** icon. There's no limit to the number of columns you can add. When you first click the icon, JDeveloper adds a new column line at the bottom of the dialog and populates it with the values from the first attribute in the bound collection; subsequent new columns are populated with values from the next attribute in the sequence, and so on.

- Delete a column using the **Delete** icon.

4. Once the table is dropped on the page, you can use the Property Inspector to set other display properties of the table. For example, you may want to set the width of the table to a certain percentage or size.For more information about display properties, see the "Using Tables and Trees" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

   Tip: When you set the table width to 100%, the table will not include borders, so the actual width of the table will be larger. To have the table set to 100% of the container width, expand the **Style** section of the Property Inspector, select the **Box** tab, and set the `Border Width` attribute to 0 pixels.

5. If you want the user to be able to edit information in the table and save any changes, you need to provide a way to submit and persist those changes. For more information, see Section 27.3, "Creating an Editable Table." For procedures on creating tables that allow users to input data, see Section 27.4, "Creating an Input Table."

27.2.2 What Happens When You Create a Table

Dropping a table from the Data Controls panel has the same effect as dropping a text field or form. Briefly, JDeveloper does the following:

- Creates the bindings for the table and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

For more information, see Section 26.2.2, "What Happens When You Create a Text Field."

27.2.2.1 Iterator and Value Bindings for Tables

When you drop a table from the Data Controls panel, a tree value binding is created. A tree consists of a hierarchy of nodes, where each subnode is a branch off a higher level node. In the case of a table, it is a flattened hierarchy, where each attribute (column) is a subnode off the table. Like an attribute binding used in forms, the tree value binding references the iterator binding, while the iterator binding references an iterator for the data collection, which facilitates iterating over the data objects in the collection. Instead of creating a separate binding for each attribute, only the tree binding to the table node is created. In the tree binding, the `AttrNames` element within the `nodeDefinition` element contains a child element for each attribute that you want to be available for display or reference in each row of the table.

The tree value binding is an instance of the `FacesCtrlHierBinding` class that extends the core `JUCtrlHierBinding` class to add two JSF specific properties: collectionModel.

- `collectionModel`: Returns the data wrapped by an object that extends the `javax.faces.model.DataModel` object that JSF and ADF Faces use for collection-valued components like tables.
- `treeModel`: Extends `collectionModel` to return data that is hierarchical in nature. For more information, see Chapter 29, "Displaying Master-Detail Data."

Example 27–1 shows the value binding for the table created when you drop the `Products` collection.

Example 27–1 Value Binding Entries for a Table in the Page Definition File

```
<bindings>
  <tree IterBinding="ProductsIterator" id="Products">
    <nodeDefinition DefName="oracle.fodemo.storefront.store.queries.ProductsVO">
      <AttrNames>
        <Item Value="ProductId"/>
        <Item Value="SupplierId"/>
        <Item Value="CategoryId"/>
        <Item Value="ProductName"/>
        <Item Value="CostPrice"/>
        <Item Value="ListPrice"/>
        .
        .
        .
     </AttrNames>
   </nodeDefinition>
  </tree>
</bindings>
```

Only the table component needs to be bound to the model (as opposed to the columns or the text components within the individual cells), because only the table needs access to the data. The tree binding for the table drills down to the individual structure attributes in the table, and the table columns can then derive their information from the table component.

27.2.2.2 Code on the JSF Page for an ADF Faces Table

When you use the Data Controls panel to drop a table onto a JSF page, JDeveloper inserts an ADF Faces `table` component, which contains an ADF Faces `column` component for each attribute named in the table binding. Each column then contains another component (such as an `inputText` or `outputText` component) bound to the attribute's value. Each column's heading is bound to the `labels` property for the control hint of the attribute.

> **Tip:** If an attribute is marked as hidden on the associated view or entity object, no corresponding UI is created for it.

Example 27–2 shows a simplified code excerpt from a table created by dropping the `Products` collection as a read only table.

Example 27–2 Simplified JSF Code for an ADF Faces Table

```
<af:table value="#{bindings.Products.collectionModel}" var="row"
          rows="#{bindings.Products.rangeSize}"
          emptyText="#{bindings.Products.viewable ? 'No data to display.':
                                                    'Access Denied.'}"
          fetchSize="#{bindings.Products.rangeSize}"
          selectedRowKeys="#{bindings.Products.collectionModel.selectedRow}"
          selectionListener="#{bindings.Products.collectionModel.makeCurrent}"
          rowSelection="single" id="t1">
  <af:column sortProperty="#{bindings.Products.hints.ProductId.name}"
             sortable="true"
             headerText="#{bindings.Products.hints.ProductId.label}" id="c1">
    <af:outputText value="#{row.ProductId}" id="ot1"/>
      <af:convertNumber groupingUsed="false"Cos
                        pattern="#{bindings.Products.hints.ProductId.format}"/>
  </af:column>
  <af:column sortProperty="#{bindings.Products.hints.SupplierId.name}"
             sortable="true"
```

```
            headerText="#{bindings.Products.hints.SupplierId.label}" id="c2">
    <af:outputText value="#{row.SupplierId}" id="ot2">
      <af:convertNumber groupingUsed="false"
                        pattern="#{bindings.Products.hints.SupplierId.format}"/>
    </af:outputText>
  </af:column>
  <af:column sortProperty="#{bindings.Products.hints.CostPrice.name}"
            sortable="true"
            headerText="#{bindings.Products.hints.CostPrice.label}" id="c3">
    <af:outputText value="#{row.CostPrice}" id="ot3">
      <af:convertNumber groupingUsed="false"
                        pattern="#{bindings.Products.hints.CostPrice.format}"/>
    </af:outputText>
  </af:column>
.
.
.
</af:table>
```

The tree binding iterates over the data exposed by the iterator binding. Note that the table's value is bound to the `collectionModel` property, which accesses the `collectionModel` object. The table wraps the result set from the iterator binding in a `collectionModel` object. The `collectionModel` allows each item in the collection to be available within the table component using the `var` attribute.

In the example, the table iterates over the rows in the current range of the `Products` iterator binding. The iterator binding binds to a row set iterator that keeps track of the current row. When you set the `var` attribute on the table to `row`, each column then accesses the current data object for the current row presented to the table tag using the `row` variable, as shown for the value of the `af:outputText` tag:

```
<af:outputText value="#{row.ProductId}"/>
```

When you drop an ADF Table (as opposed to an ADF Read Only Table), instead of being bound to the row variable, the value of the input component is implicitly bound to a specific row in the binding container through the `bindings` property, as shown in Example 27–3. Additionally, JDeveloper adds validator and converter components for each input component. By using the bindings property, any raised exception can be linked to the corresponding binding object or objects. The controller iterates through all exceptions in the binding container and retrieves the binding object to get the client ID when creating `FacesMessage` objects. This retrieval allows the table to display errors for specific cells. This strategy is used for all input components, including selection components such as lists.

Example 27–3 Using Input Components Adds Validators and Converters

```
<af:table value="#{bindings.Products.collectionModel}" var="row"
          rows="#{bindings.Products.rangeSize}"
          emptyText="#{bindings.Products.viewable ? 'No data to display.':
                                                    'Access Denied.'}"
          fetchSize="#{bindings.Products.rangeSize}"
          selectedRowKeys="#{bindings.Products.collectionModel.selectedRow}"
          selectionListener="#{bindings.Products.collectionModel.makeCurrent}"
          rowSelection="single" id="t1">
  <af:column sortProperty="bindings.Products.hints.ProductId.name" sortable="true"
            headerText="#{bindings.Products.hints.ProductId.label}" id="c1">
    <af:inputText value="#{row.bindings.ProductId.inputValue}"
                  label="#{bindings.Products.hints.ProductId.label}"
                  required="#{bindings.Products.hints.ProductId.mandatory}"
                  columns="#{bindings.Products.hints.ProductId.displayWidth}"
```

```
                maximumLength="#{bindings.Products.hints.ProductId.precision}"
                shortDesc="#{bindings.Products.hints.ProductId.tooltip}"
                id="it1">
      <f:validator binding="#{row.bindings.ProductId.validator}"/>
      <af:convertNumber groupingUsed="false"
                        pattern="#{bindings.Products.hints.ProductId.format}"/>
    </af:inputText>
  </af:column>
```

For more information about using ADF Faces validators and converters, see the "Validating and Converting Input" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Table 27–1 shows the other attributes defined by default for ADF Faces tables created using the Data Controls panel.

Table 27–1 ADF Faces Table Attributes and Populated Values

Attribute	Description	Default Value
`rows`	Determines how many rows to display at one time.	An EL expression that, by default, evaluates to the `rangeSize` property of the associated iterator binding, which determines how many rows of data are fetched from a data control at one time. Note that the value of the `rows` attribute must be equal to or less than the corresponding iterator's `rangeSize` value, as the table cannot display more rows than are returned.
`first`	Index of the first row in a range (based on 0).	An EL expression that evaluates to the `rangeStart` property of the associated iterator binding.
`emptyText`	Text to display when there are no rows to return.	An EL expression that evaluates to the viewable property on the iterator. If the table is viewable, the attribute displays **No data to display** when no objects are returned. If the table is not viewable (for example, if there are authorization restrictions set against the table), it displays **Access Denied**.
`fetchSize`	Number of rows of data fetched from the data source.	An EL expression that, by default, evaluates to the `rangeSize` property of the associated iterator binding. For more information about the `rangeSize` property, see Section 26.4.2.2, "Iterator RangeSize Attribute." This attribute can be set to a larger number than the `rows` attribute. Note that to improve scrolling behavior in a table, when the table's iterator binding is expected to manage a data set consisting of over 200 items, and the view object is configured to use range paging, the iterator actually returns a set of ranges instead of just one range. For more information about using range paging, see Section 42.1.5, "Efficiently Scrolling Through Large Result Sets Using Range Paging."
`selectedRowKeys`	The selection state for the table.	An EL expression that by default, evaluates to the selected row on the collection model.
`selectionListener`	Reference to a method that listens for a selection event.	An EL expression that by default, evaluates to the `makeCurrent` method on the collection model.

Table 27–1 (Cont.) ADF Faces Table Attributes and Populated Values

Attribute	Description	Default Value
`rowSelection`	Determines whether rows are selectable.	Set to `single` to allow one row to be selected at a time. Set to `multiple` to allow more than one row to be selected.
Column Attributes		
`sortProperty`	Determines the property on which to sort the column.	Set to the columns corresponding attribute binding value.
`sortable`	Determines whether a column can be sorted.	Set to `false`. When set to `true`, the iterator binding will access the iterator to determine the order.
`headerText`	Determines the text displayed at the top of the column.	An EL expression that, by default, evaluates to the label control hint set on the corresponding attribute.

27.2.3 What You May Need to Know About Setting the Current Row in a Table

When you use tables in an application and you allow the ADF Model layer to manage row selection, the current row is determined by the iterator. When a user selects a row in an ADF Faces table, the row in the table is shaded, and the component notifies the iterator of the selected row. To do this, the `selectedRowKeys` attribute of the table is bound to the collection model's selected row, as shown in Example 27–4.

Example 27–4 Selection Attributes on a Table

```
<af:table value="#{bindings.Products1.collectionModel}" var="row"
.
.
.
          selectedRowKeys="#{bindings.Products.collectionModel.selectedRow}"
          selectionListener="#{bindings.Products.collectionModel.
                                                              makeCurrent}"
          rowSelection="single">
```

This binding binds the selected keys in the table to the selected row of the collection model. The `selectionListener` attribute is then bound to the collection model's `makeCurrent` property. This binding makes the selected row of the collection the current row of the iterator.

> **Note:** If you create a custom selection listener, you must create a method binding to the `makeCurrent` property on the collection model (for example `#{binding.Products.collectionModel.makeCurrent}`) and invoke this method binding in the custom selection listener before any custom logic.

Although a table can handle selection automatically, there may be cases where you need to programmatically set the current row for an object on an iterator.

You can call the `getKey()` method on any view row to get a `Key` object that encapsulates the one or more key attributes that identify the row. You can also use a `Key` object to find a view row in a row set using the `findByKey()`. At runtime, when either the `setCurrentRowWithKey` or the `setCurrentRowWithKeyValue` built-in

operation is invoked by name by the data binding layer, the `findByKey()` method is used to find the row based on the value passed in as a parameter before the found row is set as the current row.

The `setCurrentRowWithKey` and `setCurrentRowWithKeyValue` operations both expect a parameter named `rowKey`, but they differ precisely by what each expects that `rowKey` parameter value to be at runtime:

setCurrentRowWithKey Operation

`setCurrentRowWithKey` expects the `rowKey` parameter value to be the *serialized string representation* of a view row key. This is a hexadecimal-encoded string that looks like this:

```
000200000002C20200000002C1020000000100000010A5AB7DAD9
```

The serialized string representation of a key encodes all of the key attributes that might comprise a view row's key in a way that can be conveniently passed as a single value in a browser URL string or form parameter. At runtime, if you inadvertently pass a parameter value that is not a legal serialized string key, you may receive exceptions like `oracle.jbo.InvalidParamException` or `java.io.EOFException` as a result. In your web page, you can access the value of the serialized string key of a row by referencing the `rowKeyStr` property of an ADF control binding (for example. `#{bindings.SomeAttrName.rowKeyStr}`) or the row variable of an ADF Faces table (e.g. `#{row.rowKeyStr}`).

setCurrentRowWithKeyValue Operation

The `setCurrentRowWithKeyValue` operation expects the `rowKey` parameter value to be the literal value representing the key of the view row. For example, its value would be simply "`201`" to find product number `201`.

Note: If you write custom code in an application module class and need to find a row based on a serialized string key passed from the client, you can use the `getRowFromKey()` method in the `JboUtil` class in the `oracle.jbo.client` package:

```
static public Row getRowFromKey(RowSetIterator rsi, String sKey)
```

The first parameter is the view object instance in which you'd like to find the row. The second parameter is the serialized string format of the key.

27.3 Creating an Editable Table

You can create a table that allows the user to edit information within the table, and then commit those changes to the data source. To do this, you use operations that can modify data records associated with the collection (or the data control itself) to create command buttons, and place those buttons in a toolbar in the table. For example, you might use the `Delete` operation to create a button that allows a user to delete a record from the current range. Or you can use the built-in **Submit** button to submit changes.

> **Tip:** To create a table that allows you to insert a new record into the data store, see Section 27.4, "Creating an Input Table."

It is important to note that these operations are executed only against objects in the ADF cache. You need to use the `Commit` operation on the root data control to actually commit any changes to the data source. You use the data control's `Rollback`

operation to roll back any changes made to the cached object. If the page is part of a transaction within a bounded task flow, you would most likely use these operations to resolve the transaction in a task flow return activity. For more information, see Section 22.2, "Managing Transactions in Task Flows."

When you decide to use editable components to display your data, you have the option of the table displaying all rows as editable at once, or displaying all rows as read-only until the user double-clicks within the row. Figure 27–4 shows a table whose rows all have editable fields. The page renders using the components that were added to the page (for example, `inputText`, `inputDate`, and `inputNumberSpinbox` components).

Figure 27–4 Table with Editable Fields

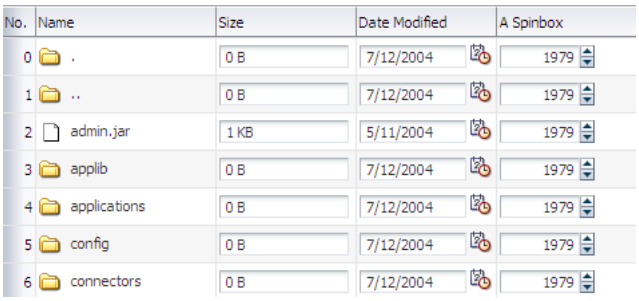
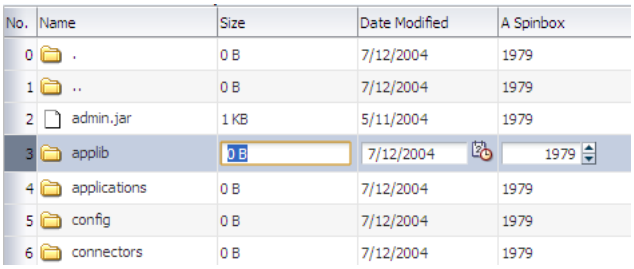


Figure 27–5 shows the same table, but configured so that the user must double-click (or single-click if the row is already selected) a row in order to edit or enter data. Note that `outputText` components are used to display the data in the non-selected rows, even though the same input components as in Figure 27–4 were used to build the page. The only row that actually renders those components is the row selected for editing.

Figure 27–5 Click to Edit a Row



For more information about how ADF Faces table components handle editing, see the "Editing Data in Tables, Trees, and Tree Tables" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

27.3.1 How to Create an Editable Table

To create an editable table, you follow similar procedures to creating a basic table, then you add command buttons bound to operations. However, in order for the table to contain a toolbar, you need to add an ADF Faces component that associates the toolbar with the items in the collection used to build the table.

Before you begin:

It may be helpful to have an understanding of editable databound tables. For more information, see Section 27.3, "Creating an Editable Table."

You may also find it helpful to understand other ADF functionality and features. For more information, see Section 27.1.2, "Additional Functionality for Databound Tables."

To create an editable table:

1. From the Data Controls panel, select a collection.

   For example, to create a simple table in the StoreFront module that will allow you to edit products in the system, you would select the `Products` collection.

2. Drag the collection onto a JSF page, and from the context menu, choose **ADF Table**.

   This creates an editable table using input components.

3. Use the ensuing Edit Table Columns dialog to determine how the attributes should behave and appear as columns in your table. Be sure to select the **Row Selection** checkbox, which will allow the user to select the row to edit.

   For more information about using this dialog to configure the table, see Section 27.2.1, "How to Create a Basic Table."

4. With the table selected in the Structure window, expand the **Behavior** section of the Property Inspector and set the **EditingMode** attribute. If you want all the rows to be editable select **editAll**. If you want the user to click into a row to make it editable, select **clickToEdit**.

5. From the Structure window, right-click the table component and select **Surround With** from the context menu.

6. In the Surround With dialog, ensure that **ADF Faces** is selected in the dropdown list, select the **Panel Collection** component, and click **OK**.

   The `panelCollection` component's toolbar facet will hold the toolbar which, in turn, will hold the command components used to update the data.

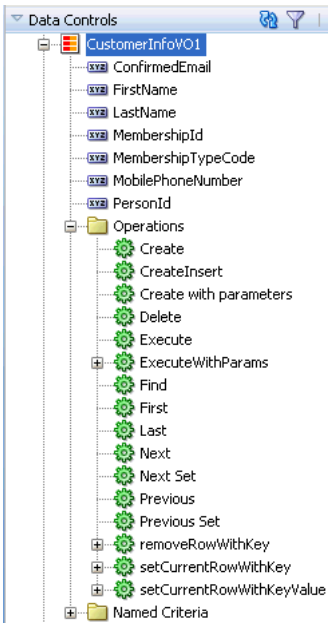
7. In the Structure window, right-click the `panelCollection`'s **toolbar** facet folder and from the context menu, choose **Insert inside toolbar** > **Toolbar**.

   This creates a toolbar that already contains a default menu that allows users to change how the table is displayed and a **Detach** link that detaches the entire table and displays it such that it occupies the majority of the space in the browser window. For more information about the `panelCollection` component, see the "Displaying Table Menus, Toolbars, and Status Bars" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

8. From the Data Controls panel, select the operation associated with the collection of objects on which you wish the operation to execute, and drag it onto the `toolbar` component in the Structure window. This will place the databound command component inside the toolbar.

   For example, if you want to be able to delete a product record, you would drag the `Delete` operation associated with the `Products` collection. Figure 27–6 shows the operations associated with a collection.

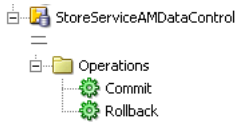
Figure 27–6 Operations Associated with a Collection



9. Choose **Operations > ADF Toolbar Button** from the context menu.
10. To create a **Submit** button that submits changes to the cache, right-click the **toolbar** component in the Structure window and choose **Insert inside af:toolbar** > **Toolbar Button**.
11. If the page is not part of a transaction within a bounded task flow, then you need to create buttons that allow the user to either commit or rollback the changes. From the Data Controls panel, drag the `Commit` and `Rollback` operations associated with the root-level data control, and drop them as either a command button or command link into the toolbar.

    Figure 27–7 shows the commit and roll back operations for the `StoreServiceAMDataControl` data control.

Figure 27–7 Commit and Rollback Operations for a Data Control



If the page is part of a transaction within a bounded task flow, then you can simply enter `Commit` and `Rollback` as the values for the transaction resolution when creating the task flow return activity. For more information, see Section 22.2, "Managing Transactions in Task Flows."

27.3.2 What Happens When You Create an Editable Table

Creating an editable table is similar to creating a form used to edit records. Action bindings are created for the operations dropped from the Data Controls panel. For details on what happens when you create an editable table, see Section 26.4.2, "What Happens When You Create Command Buttons."

27.4 Creating an Input Table

You can create a table that allows users to insert a new blank row into a table and then add values for each column (any default values set on the corresponding entity or view object will be automatically populated).

27.4.1 How to Create an Input Table

When you create an input table, you want the user to see the new blank row in the context of the other rows within the current row set. To allow this insertion, you need to use the `CreateInsert` operation instead of the `Create` operation (as you would use with forms). The `CreateInsert` operation actually creates the new row within the row set instead of only in the cache.

ADF Faces components can be set so that one component refreshes based on an interaction with another component, without the whole page needing to be refreshed. This is known as *partial page rendering*. When the user clicks a button to create the new row, you want the table to refresh to display that new row. To have that happen, you need to configure the table to respond to that user action.

Before you begin:

It may be helpful to have an understanding of ADF Faces input tables. For more information, see Section 27.4, "Creating an Input Table."

You may also find it helpful to understand other ADF functionality and features. For more information, see Section 27.1.2, "Additional Functionality for Databound Tables."

You must do the following in JDeveloper before you create an input table.

1. Create an editable table, as described in Section 27.3, "Creating an Editable Table."
2. If your table is not part of a bounded task flow, be sure to include buttons bound to the `Commit` and `Rollback` operations.

To create an input table:

1. From the Data Controls panel, drag the **CreateInsert** operation associated with the dropped collection and drop it as a toolbar button into the toolbar. You may want to change the ID to something more recognizable, such as CreateInsert. This will make it easier to identify when you need to select it as the partial trigger.

   Note: If the collection is a child to another collection (for example, the `OrderItems` collection is a child to the `Orders` collection), then instead of using the `CreateInsert` operation, you need to use the `Create with Parameters` operation. This is because the record to create for the child collection must first be associated with

2. In the Structure window, select the table component. In the Property Inspector, expand the **Behavior** section.
3. In the Property Inspector, click the dropdown menu for the **PartialTriggers** attribute, and select **Edit**.
4. In the Edit Property dialog, expand the toolbar facet for the `panelCollection` component and then expand the toolbar that contains the **CreateInsert** command component. Select that component and shuttle it to the **Selected** panel. Click **OK**. This sets that component to be the trigger that will cause the table to refresh.

27.4.2 What Happens When You Create an Input Table

When you use the `CreateInsert` operation to create an input table, JDeveloper:

- Creates an iterator binding for the collection, an action binding for the `CreateInsert` operation, and attribute bindings for the table. The `CreateInsert` operation is responsible for creating the new row in the row set. If you created command buttons or links using the `Commit` and `Rollback` operations, JDeveloper also creates an action bindings for those operations.
- Inserts code in the JSF page for the table using ADF Faces `table`, `column`, and `inputText` components, and in the case of the operations, `commandButton` components.

Example 27–5 shows the page definition file for an input table created from the `Products` collection (some attributes were deleted in the Edit Columns dialog when the collection was dropped).

Example 27–5 Page Definition Code for an Input Table

```
<executables>
  <iterator Binds="Products" RangeSize="25"
            DataControl="StoreServiceAMDataControl" id="ProductsIterator"/>
</executables>
<bindings>
  <tree IterBinding="ProductsIterator" id="Products">
    <nodeDefinition DefName="oracle.fodemo.storefront.store.queries.ProductsVO">
      <AttrNames>
        <Item Value="ProductId"/>
        <Item Value="ProductName"/>
        <Item Value="CostPrice"/>
        <Item Value="ListPrice"/>
        <Item Value="Description"/>
        <Item Value="CategoryName"/>
        <Item Value="CategoryDescription"/>
        <Item Value="ProductImageId"/>
      </AttrNames>
    </nodeDefinition>
  </tree>
  <action IterBinding="ProductsIterator" id="CreateInsert"
          RequiresUpdateModel="true" Action="createInsertRow"/>
  <action id="Commit" RequiresUpdateModel="true" Action="commitTransaction"
          DataControl="StoreServiceAMDataControl"/>
  <action id="Rollback" RequiresUpdateModel="false"
          Action="rollbackTransaction"
          DataControl="StoreServiceAMDataControl"/>
</bindings>
```

Example 27–6 shows the code added to the JSF page that provides partial page rendering, using the `CreateInsert` command toolbar button as the trigger to refresh the table.

Example 27–6 Partial Page Trigger Set on a Command Button for a Table

```
<af:form>
  <af:panelCollection id="pc1">
    <f:facet name="menus"/>
    <f:facet name="toolbar">
      <af:toolbar id="tb1">
        <af:commandToolbarButton actionListener="#{bindings.CreateInsert.execute}"
                                 text="CreateInsert"
```

```
                                disabled="#{!bindings.CreateInsert.enabled}"
                                id="CreateInsert"/>
            <af:commandToolbarButton actionListener="#{bindings.Commit.execute}"
                                text="Commit"
                                disabled="false" id="ctb2"/>
            <af:commandToolbarButton actionListener="#{bindings.Rollback.execute}"
                                text="Rollback"
                                disabled="#{!bindings.Rollback.enabled}"
                                immediate="true" id="ctb3">
              <af:resetActionListener/>
            </af:commandToolbarButton>
          </af:toolbar>
        </f:facet>
        <f:facet name="statusbar"/>
        <af:table value="#{bindings.Products.collectionModel}" var="row"
                  rows="#{bindings.Products.rangeSize}"
                  emptyText="#{bindings.Products.viewable ? \'No data to display.\' :
                                                          \'Access Denied.\'}"
                  fetchSize="#{bindings.Products.rangeSize}"
                  rowSelection="single" partialTriggers="CreateInsert" id="t1">
          <af:column sortProperty="ProductId" sortable="false"
                     headerText="#{bindings.Products.hints.ProductId.label}" id="c1">
            <af:inputText value="#{row.ProductId}" simple="true"
                          required="#{bindings.Products.hints.ProductId.mandatory}"
                          columns="#{bindings.Products.hints.ProductId.displayWidth}"
                          maximumLength="#{bindings.Products.hints.
                                                productId.precision}" id="it1"/>
          </af:column>
.
.
.
        </af:table>
      </af:panelCollection>
    </af:form>
```

27.4.3 What Happens at Runtime: How CreateInsert and Partial Page Refresh Work

When the button bound to the `CreateInsert` operation is invoked, the action executes, and a new instance for the collection is created and inserted as the page is rerendered. Because the button was configured to be a trigger that causes the table to refresh, the table redraws with the new empty row shown at the top. When the user clicks the button bound to the `Commit` action, the newly created rows in the row set are inserted into the database. For more information about partial page refresh, see the "Rendering Partial Page Content" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

27.4.4 What You May Need to Know About Creating a Row and Sorting Columns

If your table columns allow sorting, and the user has sorted on a column before inserting a new row, then that new row will not be sorted. To have the column sort with the new row, the user must first sort the column opposite to the desired sort, and then resort. This is because the table assumes the column is already sorted, so clicking on the desired sort order first will have no effect on the column.

For example, say a user had sorted a column in ascending order, and then added a new row. Initially, that row appears at the top. If the user first clicks to sort the column again in ascending order, the table will not resort, as it assumes the column is already in ascending order. The user must first sort on descending order and then ascending order.

If you want the data to automatically sort on a specific column in a specific order after inserting a row, then programmatically queue a `SortEvent` after the commit, and implement a handler to execute the sort.

27.4.5 What You May Need to Know About Create and CreateInsert

When you use the `Create` or `CreateInsert` operation to declaratively create a new row, it performs the following lines of code:

```
// create a new row for the view object
Row newRow = yourViewObject.createRow();
// mark the row as being "initialized", but not yet new
newRow.setNewRowState(Row.STATUS_INITIALIZED);
```

However, if you are using the `CreateInsert` operation, it performs the additional line of code to insert the row into the row set:

```
// insert the new row into the view object's default rowset
yourViewObject.insertRow(newRow);
```

When you create a row in an entity-based view object, the `Transaction` object associated with the current application module immediately takes note of the fact. The new entity row that gets created behind the view row is already part of the `Transaction`'s list of pending changes. When a newly created row is marked as having the *initialized* state, it is removed from the `Transaction`'s pending changes list and is considered a blank row in which the end user has not yet entered any data values. The term *initialized* is appropriate since the end user will see the new row initialized with any default values that the underlying entity object has defined. If the user never enters any data into any attribute of that initialized row, then it is as if the row never existed. At transaction commit time, since that row is not part of the `Transaction`'s pending changes list, no `INSERT` statement will be attempted for it.

As soon as at least one attribute in an initialized row is set, it automatically transitions from the initialized status to the new status (`Row.STATUS_NEW`). At that time, the underlying entity row is enrolled in the `Transaction`'s list of pending changes, and the new row will be permanently saved the next time you commit the transaction.

Note: If the end user performs steps that result in creating many initialized rows but never populating them, it might seem like a recipe for a slow memory leak. However, the memory used by an initialized row that never transitions to the new state will eventually be reclaimed by the Java virtual machine's garbage collector.

27.5 Modifying the Attributes Displayed in the Table

Once you use the Data Controls panel to create a table, you can then delete attributes, change the order in which they are displayed, change the component used to display them, and change the attribute binding for the component. You can also add new attributes, or rebind the table to a new data control.

27.5.1 How to Modify the Displayed Attributes

You can modify the following aspects of a table that was created using the Data Controls panel:

- Change the binding for the label of a column

- Change the UI component bound to an attribute
- Add or delete columns that represent the attributes
- Reorder the columns in the table
- Enable selection and sorting

To change the attributes for a table:

1. In the Structure window, select the table component.
2. In the Property Inspector, expand the Columns section to change, add, or delete the attributes that are displayed as columns. You can also change the UI component used within the column to display the data.
3. Expand other sections of the Property Inspector to change the display and behavior of the table, such as filtering and sorting.

27.5.2 How to Change the Binding for a Table

Instead of modifying a binding, you can completely change the object to which the table is bound.

To rebind a table:

1. Right-click the table in the Structure window and choose **Rebind to Another ADF Control**.
2. In the Bind to ADF Control dialog, select the new collection to which you want to bind the table. Note that changing the binding for the table will also change the binding for all the columns. You can then use the procedures in Section 27.5.1, "How to Modify the Displayed Attributes" to modify those bindings.

> **Tip:** You can also rebind a table by dragging a different view object on top of the existing table.

27.5.3 What Happens When You Modify Bindings or Displayed Attributes

When you simply modify how an attribute is displayed by moving the UI component or changing the UI component, JDeveloper changes the corresponding code on the JSF page. When you use the binding editors to add or change a binding, JDeveloper adds the code to the JSF page, and also adds the appropriate elements to the page definition file.

28

Command Components to Invoke Functionality in the View Layer

This chapter describes how to add more complex bindings to your pages. It describes how to use methods that take parameters for creating forms and command components.

This chapter includes the following sections:

- Section 28.1, "About Command Components"
- Section 28.2, "Creating Command Components to Execute Methods"
- Section 28.3, "Setting Parameter Values Using a Command Component"
- Section 28.4, "Overriding Declarative Methods"

> **Note:** Some of the implementation methods in this chapter are intended for page-level designs. If you are using task flows, you may be able to perform many of the same functions. For more information, see Chapter 18, "Getting Started with ADF Task Flows."

28.1 About Command Components

Once you create a basic page and add navigation capabilities, you may want to add more complex features, such as passing parameters between pages or providing the ability to override declarative actions. Oracle ADF provides many features that allow you to add this complex functionality using very little actual code.

Some of the functions described in this chapter may be performed using other methodology. For example, if you are using task flows instead of individual page flows, you should use the task flow parameter passing mechanism. Or, if you are using ADF Business Components, you should use the validation rules on entity objects in the data model project, rather than using ADF Model validation rules. For more information about validation rules for Business Components, see Chapter 7, "Defining Validation and Business Rules Declaratively."

28.1.1 Command Component Use Cases and Examples

You can use command components to perform an action, such as passing parameters, overriding declarative actions, or perform some logic. You can bind a command component to a custom method in the Data Control panel. For example, if you have a custom method called `updateItemInCart(Integer, Integer, Boolean)`, you can create a button bound to this method so when the user presses this button, the shopping cart will be updated.

You can also use the `setPropertyListener` tag within a command component to assign values when an event fires. For example, you have a button that initiates a search and displays the results on another page depending on a parameter value stored in a managed bean. You use the `setPropertyListener` property to pass this parameter value to the method that checks this value.

You can add code to a declarative method that is bound to a command component. For example, after you have created a declarative `Commit` button, you can add code to a managed bean and then override the `Commit` operation for additional processing

28.1.2 Additional Functionality for Command Components

You may find it helpful to understand other ADF features before you work with command components. Following are links to other functionality that may be of interest.

- For information about creating custom methods that can be added to the page as a command component, see Section 9.7, "Customizing an Application Module with Service Methods," and Section 9.9, "Publishing Custom Service Methods to UI Clients."
- For information about creating command components from operations, see Section 26.4.2, "What Happens When You Create Command Buttons."
- If you want to use EL expressions in your methods, see Section 26.4.2.3, "EL Expressions Used to Bind to Navigation Operations."
- For more information about the command button's `actionListener` attribute, see Section 26.4.3, "What Happens at Runtime: How Action Events and Action Listeners Work."
- For more information about ADF bindings, see Section 13, "Using ADF Model in a Fusion Web Application."
- If you are using task flows, you can use the task flow parameter passing mechanism. For more information, see Chapter 20, "Using Parameters in Task Flows."
- If you considering using task flow call methods, see Section 19.5, "Using Method Call Activities."
- If you are using managed beans, see Section 24.4, "Using a Managed Bean in a Fusion Web Application."

28.2 Creating Command Components to Execute Methods

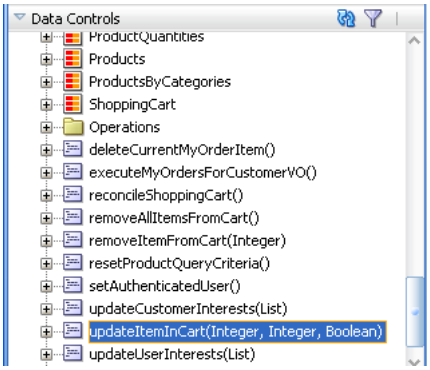
When your application contains custom methods, these methods appear in the Data Controls panel. You can then drag these methods and drop them as command buttons. When a user clicks the button, the method is executed.

For more information about creating custom methods, see Section 9.7, "Customizing an Application Module with Service Methods," and Section 9.9, "Publishing Custom Service Methods to UI Clients."

Note: If you are using task flows, you can call methods directly from the task flow definition. For more information, see Section 19.5, "Using Method Call Activities."

For example, the application module in the StoreFront module of the Fusion Order Demo application contains the `updateItemInCart(Integer, Integer, Boolean)` method. This method updates the items in the shopping cart. To allow the user to execute this method, you drag the `updateItemInCart(Integer, Integer Boolean)` method from the Data Controls panel, as shown in Figure 28–1.

Figure 28–1 Methods in the Data Controls Panel



28.2.1 How to Create a Command Component Bound to a Custom Method

In order to perform the required business logic, many methods require a value for their parameter or parameters. This means that when you create a button bound to the method, you need to specify where the value for the parameter(s) is to be retrieved from. For example, if you use the `updateItemInCart(Integer, Integer, Boolean)` method, you need to specify the items to be updated.

Before you begin:

It may be helpful to have an understanding of custom methods. For more information, see Section 28.2, "Creating Command Components to Execute Methods."

You may also find it helpful to understand functionality that can be added using other command components. For more information, see Section 28.1.2, "Additional Functionality for Command Components."

To create a command component bound to a custom method:

1. From the Data Controls panel, drag the method onto the page.

   Tip: If you are dropping a button for a method that needs to work with data in a table or form, that button must be dropped inside the table or form.

2. From the context menu, choose **Create > Methods** > **ADF Button**.

   If the method takes parameters, the Edit Action Binding dialog opens. In the Edit Action Binding dialog, enter values for each parameter or click the **Show EL Expression Builder** menu selection in the **Value** column of **Parameters** to launch the EL Expression Builder.

28.2.2 What Happens When You Create Command Components Using a Method

When you drop a method as a command button, JDeveloper:

- Defines a method action binding for the method.

- If the method takes any parameters, JDeveloper creates `NamedData` elements that hold the parameter values.
- Inserts code in the JSF page for the ADF Faces command component.
- Binds the button to the method using `actionListener`.
- Uses the return value from the method call.

28.2.2.1 Defining Method Action Binding

JDeveloper adds an action binding for the method. Action bindings use the `RequiresUpdateModel` property, which determines whether or not the model needs to be updated before the action is executed. For command operations, this property is set to `true` by default, which means that any changes made at the view layer must be moved to the model before the operation is executed.

28.2.2.2 Using Parameters in a Method

When you drop a method that takes parameters onto a JSF page, JDeveloper creates a method action binding. This binding is what causes the method to be executed when a user clicks the command component. When the method requires parameters to run, JDeveloper also creates `NamedData` elements for each parameter. These elements represent the parameters of the method.

For example, the `updateItemInCart(Integer, Integer, Boolean)` method action binding contains `NamedData` elements for the parameter. This element is bound to the value specified when you created the action binding. Example 28–1 shows the method action binding created when you drop the `updateItemInCart(Integer, Integer, Boolean)` method, and bind the `Integer` parameter (named `productId`) and the other `Integer` parameter (named `quantity`) and the `Boolean` parameter (named `isSet`) to the appropriate variables.

Example 28–1 Method Action Binding for a Parameter Method

```
<methodAction id="updateItemInCart"
                  InstanceName="StoreServiceAMDataControl.dataProvider"
                  DataControl="StoreServiceAMDataControl"
                  RequiresUpdateModel="true" Action="invokeMethod"
                  MethodName="updateItemInCart" IsViewObjectMethod="false">
      <NamedData NDName="productId" NDType="java.lang.Integer"/>
      <NamedData NDName="quantity" NDType="java.lang.Integer"/>
      <NamedData NDName="isSet" NDType="java.lang.Boolean"/>
</methodAction>
```

28.2.2.3 Adding ADF Faces Component Code to JSF Page

JDeveloper adds code for the ADF Faces component to the JSF page. This code is the same as code for any other command button, as described in Section 26.4.2.3, "EL Expressions Used to Bind to Navigation Operations." However, instead of being bound to the `execute` method of the action binding for a built-in operation, the button is bound to the `execute` method of the method action binding for the method that was dropped.

28.2.2.4 Using EL Expressions to Bind to Methods

Like creating command buttons using operations, when you create a command button using a method, JDeveloper binds the button to the method using the

actionListener attribute. The button is bound to the execute property of the action binding for the given method using an EL expression. This EL expression causes the binding's method to be invoked on the application module. For more information about the command button's actionListener attribute, see Section 26.4.3, "What Happens at Runtime: How Action Events and Action Listeners Work."

> **Tip:** Instead of binding a button to the execute method on the action binding, you can bind the button to the method in a backing bean that overrides the execute method. Doing so allows you to add logic before or after the original method runs. For more information, see Section 28.4, "Overriding Declarative Methods."

Like navigation operations, the disabled property on the button uses an EL expression to determine whether or not to display the button. Example 28–2 shows the EL expression used to bind the command button to the updateItemInCart(Integer, Integer, Boolean) method.

Example 28–2 JSF Code to Bind a Command Button to a Method

```
<af:commandButton actionListener="#{bindings.updateItemInCart.execute}"
                          text="updateItemInCart"
                          disabled="#{!bindings.updateItemInCart.enabled}"/>
```

> **Tip:** When you drop a command button component onto the page, JDeveloper automatically gives it an ID based on the number of the same type of component that was previously dropped. For example, commandButton1, commandButton2. If you change the ID to something more descriptive, you must manually update any references to it in any EL expressions in the page.

28.2.2.5 Using the Return Value from a Method Call

You can also use the return value from a method call. Example 28–3 shows a custom method that returns a string value.

Example 28–3 Custom Method That Returns a Value

```
/**
 * Custom method.
*/
    public String getHelloString() {
        return ("Hello World");
    }
```

Example 28–4 shows the code in the JSF page for the command button and an outputText component.

Example 28–4 Command Button to Call the Custom Method

```
<af:commandButton actionListener="#{bindings.getHelloString.execute}"
        text="getHelloString"
        disabled="#{!bindings.getHelloString.enabled}"
        id="helloButtonId"/>
<af:outputText value="#{bindings.return.inputValue}"
        id="helloOutputId"/>
```

When the user clicks the command button, it calls the custom method. The method returns the string "Hello World" to be shown as the value of the `outputText` component.

28.2.3 What Happens at Runtime: Command Button Method Bindings

When the user clicks the button, the method binding causes the associated method to be invoked, passing in the value bound to the `NamedData` element as the parameter. For example, if a user clicks a button bound to the `updateItemInCartItem(Integer, Integer, Boolean)` method, the method takes the values of the product `Id` and quantity and updates the shopping cart.

28.3 Setting Parameter Values Using a Command Component

There may be cases where an action on one page needs to set parameters that will be used to determine application functionality. For example, you can create a search command button on one page that will navigate to a results table on another page. But the results table will display only if a parameter value is `false`.

You can use a managed bean to pass this parameter between the pages, and to contain the method that is used to check the value of this parameter. The managed bean is instantiated as the search page is rendered, and a method on the bean checks that parameter. If it is `null` (which it will be the first time the page is rendered), the bean sets the value to `true`.

For more information about creating custom methods, see Section 9.7, "Customizing an Application Module with Service Methods," and Section 9.9, "Publishing Custom Service Methods to UI Clients."

Note: If you are using task flows, you can use the task flow parameter passing mechanism. For more information, see Chapter 20, "Using Parameters in Task Flows."

A `setPropertyListener` component with `type` property set to `action`, which is nested in the command button that executed this search, is then used to set this flag to `false`, thus causing the results table to display once the search is executed. For information about using managed beans, see Section 24.4, "Using a Managed Bean in a Fusion Web Application."

28.3.1 How to Set Parameters Using setPropertyListener Within a Command Component

You can use the `setPropertyListener` component to set values on other objects. This component must be a child of a command component.

Before you begin:

It may be helpful to have an understanding of how `setPropertyListener` and managed bean can be used to set a value. For more information, see Section 28.3, "Setting Parameter Values Using a Command Component."

You may also find it helpful to understand functionality that can be added using other command components. For more information, see Section 28.1.2, "Additional Functionality for Command Components."

You will need to complete this task:

Create a command component on the page.

To use the setPropertyListener component:

1. In the Component Palette, from the Operations panel, drag a **setPropertyListener** component and drop it as a child to the command component.

   Or right-click the component and select **Insert inside Button > ADF Faces > setPropertyListener**.

2. In the Insert Set Property Listener dialog, enter the parameter value in the **From** field.

3. Enter the parameter target in the **To** field.

   Tip: Consider storing the parameter value on a managed bean or in scope instead of setting it directly on the resulting page's page definition file. By setting it directly on the next page, you lose the ability to easily change navigation in the future. For more information, see Section 24.4, "Using a Managed Bean in a Fusion Web Application." Additionally, the data in a binding container is valid only during the request in which the container was prepared. The data may change between the time you set it and the time the next page is rendered.

4. From the **Type** dropdown menu, select **Action**.

5. Click **OK**.

28.3.2 What Happens When You Set Parameters

The `setPropertyListener` component lets the command component set a value before it navigates to the next page. When you set the `from` attribute either to the source of the value you need to pass or to the actual value, the component will be able to access that value. When you set the `to` attribute to a target, the command component is able to set the value on the target. Example 28–5 shows the code on the JSF page for a command component that takes the value `false` and sets it as the value of the `initialSearch` flag on the `searchResults` managed bean.

Example 28–5 JSF Page Code for a Command Button Using a setPropertyListener Component

```
<af:commandButton actionListener="#{bindings.Execute.execute}"
                  text=Search>
      <af:setPropertyListener from="#{false}"
                              to="#{searchResults.initialSearch}"/>
                              type="action"/>
</af:commandButton>
```

28.3.3 What Happens at Runtime: setPropertyListener for a Command Component

When a user clicks the command component, before navigation occurs, the `setPropertyListener` component sets the parameter value. In Example 28–4, the `setPropertyListener` takes the value `false` and sets it as the value for the `initialSearch` attribute on the `searchResults` managed bean. Now, any component that needs to know this value in determining whether or not to render can access it using the EL expression `#{searchResults.initialSearch}`.

28.4 Overriding Declarative Methods

When you drop an operation or method as a command button, JDeveloper binds the button to the `execute` method for the operation or method. However, there may be occasions when you need to add logic before or after the existing logic.

> **Note:** If you are using task flows, you can call custom methods from the task flow. For more information, see Section 18, "Getting Started with ADF Task Flows."

JDeveloper allows you to add logic to a declarative operation by creating a new method and property on a managed bean that provides access to the binding container. By default, this generated code executes the operation or method. You can then add logic before or after this code. JDeveloper automatically binds the command component to this new method, instead of to the `execute` property on the original operation or method. Now when the user clicks the button, the new method is executed.

For example, in the Fusion Order Demo application Orders page, the `Commit` operation requires additional processing. The **Commit** button is renamed to **Submit Orders** and logic is added to the `submitOrders` method in the `orderPageBean` managed bean.

In order to override a declarative method, you must have a managed bean to hold the new method to which the command component will be bound. If your page has a backing bean associated with it, JDeveloper adds the code needed to access the binding object to this backing bean. If your page does not have a backing bean, JDeveloper asks you to create one.

28.4.1 How to Override a Declarative Method

You can add a command component and override its declarative methods using a managed bean.

Before you begin:

It may be helpful to have an understanding of how to override declarative methods. For more information, see Section 28.4, "Overriding Declarative Methods."

You may also find it helpful to understand functionality that can be added using other command components. For more information, see Section 28.1.2, "Additional Functionality for Command Components."

You will need to complete this task:

Create the method that will override the declarative method in a managed bean. Operations are available by default.

> **Note:** You cannot override the declarative method if the command component currently has an EL expression as its value for the `Action` attribute, because JDeveloper will not overwrite an EL expression. You must remove this value before continuing.

To override a declarative method:

1. Drag the operation or method to be overridden onto the JSF page and drop it as a UI command component.

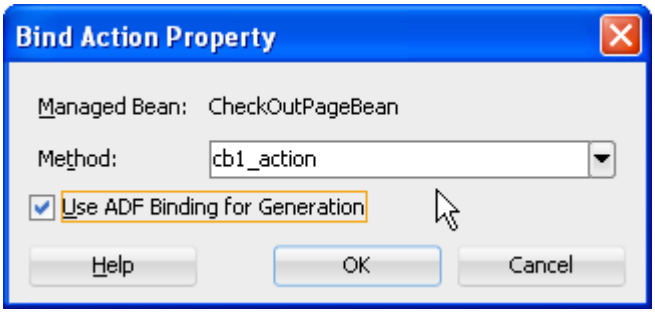
The component is created and bound to the associated binding object in the ADF Model layer with the `ActionListener` attribute.

For more information about creating command components using methods on the Data Controls panel, see Section 28.2, "Creating Command Components to Execute Methods."

For more information about creating command components from operations, see Section 26.4.2, "What Happens When You Create Command Buttons."

2. On the JSF page, double-click the component.
3. In the Bind Action Property dialog, identify the backing bean and the method to which you want to bind the component, using one of the following techniques:
   - If auto-binding has been enabled on the page, the backing bean is already selected for you, as shown in Figure 28–2.

Figure 28–2 Bind Action Property Dialog for a Page with Auto-Binding Enabled



     – To create a new method, enter a name for the method in the **Method** field, which initially displays a default name.

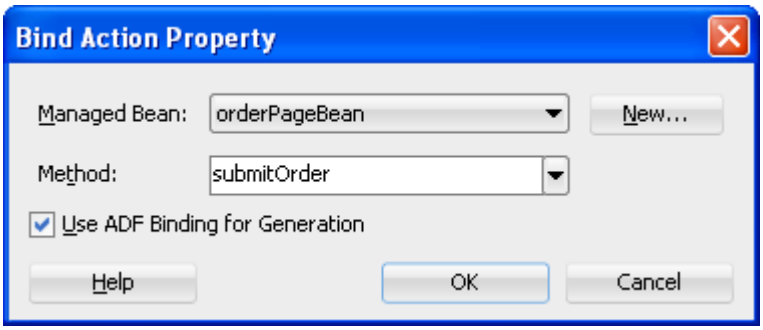
       or

     – To use an existing method, select a method from the dropdown list in the **Method** field.

     – Select **Use ADF Binding for Generation**.

   - If the page is not using auto-binding, you can select from an existing backing bean or create a new one, as shown in Figure 28–3.

Figure 28–3 Bind Action Property Dialog for a Page with Auto-Binding Disabled



     – Click **New** to create a new backing bean. In the Create Managed Bean dialog, name the bean and the class, and set the bean's scope.

       or

– Select an existing backing bean and method from the dropdown lists.

Note: Whenever there is a value for the `ActionListener` attribute on the command component, JDeveloper understands that the button is bound to the `execute` property of a binding. If you have removed that binding, you will not be given the choice to generate the ADF binding code. You will need to insert the code manually, or to set a dummy value for the `ActionListener` before double-clicking the command component.

---

4. After identifying the backing bean and method, click **OK** in the Bind Action Property dialog

   JDeveloper opens the managed bean in the source editor. Example 28–6 shows the code inserted into the bean. In this example, a command button is bound to the `Commit` operation.

Example 28–6 Generated Code in a Backing Bean to Access the Binding Object

```
public String submitOrder() {
        BindingContainer bindings = getBindings();
        OperationBinding operationBinding =
              bindings.getOperationBinding("Commit");
        Object result = operationBinding.execute();
        if (!operationBinding.getErrors().isEmpty()) {
            return null;
        }
}
```

5. You can now add logic either before or after the binding object is accessed, as shown in Example 28–7.

Example 28–7 Code Added to the Overridden Method

```
public String submitOrder() {
        DCBindingContainer bindings =
            (DCBindingContainer)JSFUtils.resolveExpression("#{bindings}");
        OperationBinding operationBinding =
            bindings.getOperationBinding("Commit");
        JUCtrlAttrsBinding statusCode =
            (JUCtrlAttrsBinding)bindings.findNamedObject("OrderStatusCode");
        statusCode.setAttribute("OrderStatusCode", "PENDING");
        JUCtrlAttrsBinding orderDate =
            (JUCtrlAttrsBinding)bindings.findNamedObject("OrderDate");
        orderDate.setAttribute("OrderDate", new Date());
        JUCtrlAttrsBinding orderId =
            (JUCtrlAttrsBinding)bindings.findNamedObject("OrderId");
        JSFUtils.storeOnSession("orderId", orderId.getAttribute("OrderId"));
        JUCtrlAttrsBinding invoiceTotal =
            (JUCtrlAttrsBinding)bindings.findNamedObject("InvoiceTotal");
        JUCtrlAttrsBinding orderTotal =
            (JUCtrlAttrsBinding)bindings.findNamedObject("OrderTotal");
        orderTotal.setAttribute("OrderTotal",
             invoiceTotal.getAttribute("InvoiceTotal"));
        Object result = operationBinding.execute();
        ShoppingCartBean shoppingCartBean =
             (ShoppingCartBean)JSFUtils.resolveExpression("#{shoppingCartBean}");
        shoppingCartBean.removeAllItems();
```

```
        return "orderSummary";
}
```

The code in Example 28–7 uses the FOD utility method `JSFUtils.resolveExpression` to resolve EL expressions. The code for such a method would be similar to the code in Example 28–8.

Example 28–8 Utility Method to Solve EL Expressions

```
public static Object resolveExpression(String expression) {
        FacesContext facesContext = getFacesContext();
        Application app = facesContext.getApplication();
        ExpressionFactory elFactory = app.getExpressionFactory();
        ELContext elContext = facesContext.getELContext();
        ValueExpression valueExp =
            elFactory.createValueExpression(elContext, expression, Object.class);
        return valueExp.getValue(elContext);
    }
```

In addition to any processing logic, you may also want to write conditional logic to return one of multiple outcomes. For example, you might want to return `null` if there is an error in the processing, or another outcome value if the processing was successful. A return value of `null` causes the navigation handler to forgo evaluating navigation cases and to immediately redisplay the current page.

> **Tip:** To trigger a specific navigation case, the outcome value returned by the method must exactly match the outcome value in the navigation rule, including case.

The command button is now bound to this new method using the `Action` attribute instead of the `ActionListener` attribute. If a value had previously existed for the `Action` attribute (such as an outcome string), that value is added as the return for the new method. If there was no value, the return is kept as `null`.

28.4.2 What Happens When You Override a Declarative Method

When you override a declarative method, JDeveloper adds a managed property to your backing bean with the managed property value of `#{bindings}` (the reference to the binding container), and it adds a strongly typed bean property to your class of the `BindingContainer` type, which the JSF runtime will then set with the value of the managed property expression `#{bindings}`. JDeveloper also adds logic to the UI command action method. This logic includes the strongly typed `getBindings()` method used to access the current binding container.

The code does the following:

- Accesses the binding container.
- Finds the binding for the associated method, and executes it.
- Adds a return for the method that can be used for navigation. By default, the return is `null`. If an outcome string had previously existed for the button's `Action` attribute, that attribute is used as the return value. You can change this code as needed.

JDeveloper automatically rebinds the UI command component to the new method using the `Action` attribute, instead of the `ActionListener` attribute. Example 28–9 shows the code when a `Commit` operation is declaratively added to a page.

Example 28–9 JSF Page Code for a Command Button Bound to a Declarative Method

```
<af:commandButton actionListener="#{bindings.Commit.execute}"
                  text="Commit"
                  disabled="#{!bindings.Commit.enabled}"/>
```

Example 28–10 shows the code after the method on the page's backing bean is overridden. Note that the `action` attribute is now bound to the backing bean's method.

Example 28–10 JSF Page Code for a Command Button Bound to an Overridden Method

```
<af:commandButton text="#{res['order.cart.submit']}"
           action="#{orderPageBean.submitOrder}"/>
```

Tip: If when you click the button that uses the overridden method you receive this error:

```
SEVERE: Managed bean main_bean could not be created
The scope of the referenced object: '#{bindings}' is
shorter than the referring object
```

it is because the managed bean that contains the overriding method has a scope that is greater than `request` (that is, either `session` or `application`). Because the data in the binding container referenced in the method has a scope of `request`, the scope of this managed bean must be set to the same or a lesser scope.

29

Displaying Master-Detail Data

This chapter describes how to create various types of pages that display master-detail related data.

This chapter includes the following sections:

- Section 29.1, "About Displaying Master-Detail Data"
- Section 29.2, "Common Functionality for Master-Detail Tables, Forms, and Trees"
- Section 29.3, "Using Tables and Forms to Display Master-Detail Objects"
- Section 29.4, "Using Trees to Display Master-Detail Objects"
- Section 29.5, "Using Tree Tables to Display Master-Detail Objects"
- Section 29.6, "Using Selection Events with Trees and Tables"

For information about using a selection list to populate a collection with a key value from a related master or detail collection, see Chapter 30, "Creating Databound Selection Lists and Shuttles".

29.1 About Displaying Master-Detail Data

In ADF Business Components, a master-detail relationship refers to two view object instances that are related by a view link. As described in Section 5.1, "About View Objects", a view link represents the relationship between two view objects, which is usually, but not necessarily, based on a foreign-key relationship between the underlying data tables. The view link associates a row of one view object instance (the master object) with one or more rows of another view object instance (the detail object).

To display master-detail data on a page using ADF data binding, you exclusively use data model view link instances, which support master-detail coordination. When using ADF Business Components in combination with the ADF Model layer and ADF Faces UI components, the data model automatically updates to reflect any changes to the row sets of these business objects.

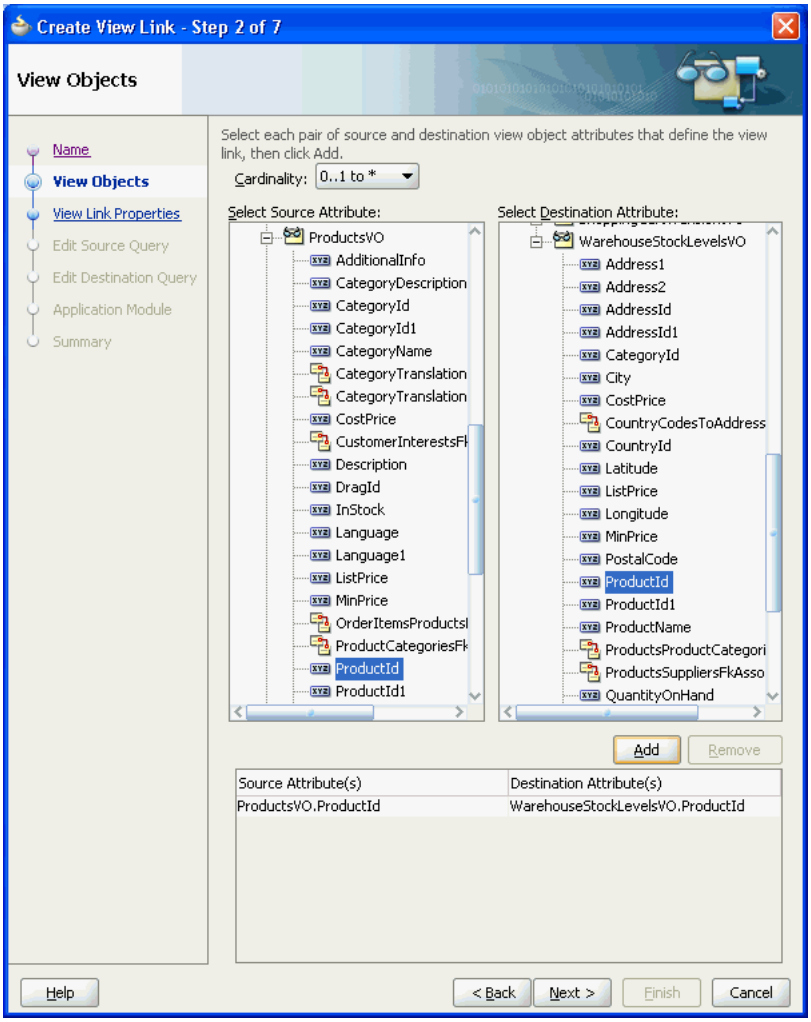
There are many instances where data needs to be presented in a hierarchical manner. In a master-detail relationship, when the user changes the selected item in the master object, the data set displayed in the detail object changes with it. For example, selecting the television categories in the master object would display all the models of televisions in the detail object.

Master-detail tables, forms, trees, and tree tables provide an excellent way to display hierarchical data.

29.1.1 Master-Detail Tables, Forms, and Trees Use Cases and Examples

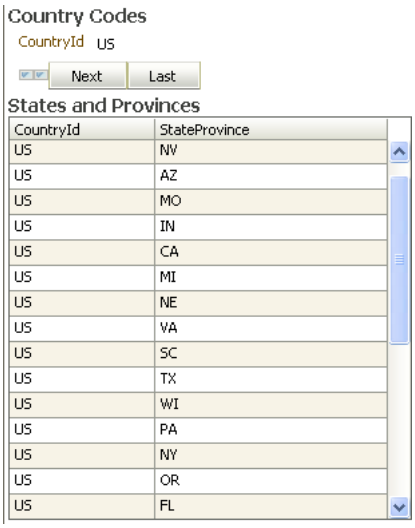
To enable master-detail coordination, you must add both the master view object and the detail view object instances to the application module data model. For example, in the Fusion Order Demo application, there is a view link from the `ProductsVO` view object to the `WarehouseStockLevelsVO` view object based on the `ProductId` attribute, both contained in the application module data model, as shown in Figure 29–1. A change in the current row of the master view object instance causes the row set of the detail view object instance to refresh to include the details for the current master.

Figure 29–1 View Link between Products and WarehouseStockLevels View Objects



When objects have a master-detail relationship, you can declaratively create pages that display the data from both objects simultaneously. For example, the page shown in Figure 29–2 displays a country code in a form at the top of the page and its related states and provinces in a table at the bottom of the page. This is possible because the objects have a master-detail relationship. In this example, the `Country Code` is the master object and `States` is the detail object. ADF iterators automatically manage the synchronization of the detail data objects displayed for a selected master data object. Iterator bindings simplify building user interfaces that allow scrolling and paging through collections of data and drilling-down from summary to detail information.

Figure 29–2 Detail Table



You display master and detail objects in forms and tables. The master-detail form can display these objects on separate pages. For example, you can display the master object in a table on one page and detail objects in a read-only form on another page.

Note: There are some cases when the master-detail UI components that JDeveloper provides cannot provide the functionality you require. For example, you may need to bind components programatically instead of using the master-detail UI components.

A master object can have many detail objects, and each detail object can in turn have its own detail objects, down to many levels of depth. If one of the detail objects in this hierarchy is dropped from the Application Navigator as a master-detail form on a page, only its immediate parent master object displays on the page. The hierarchy will not display all the way up to the topmost parent object.

If you display the detail object as a tree or tree table object, it is possible to display the entire hierarchy with multiple levels of depth, starting with the topmost master object, and traversing detail children objects at each node.

Figure 29–3 shows an example of a master form with detail table. When the user navigates through the departments using the command buttons, the table lists the employees in that department.

Figure 29–3 Master Form Detail Table

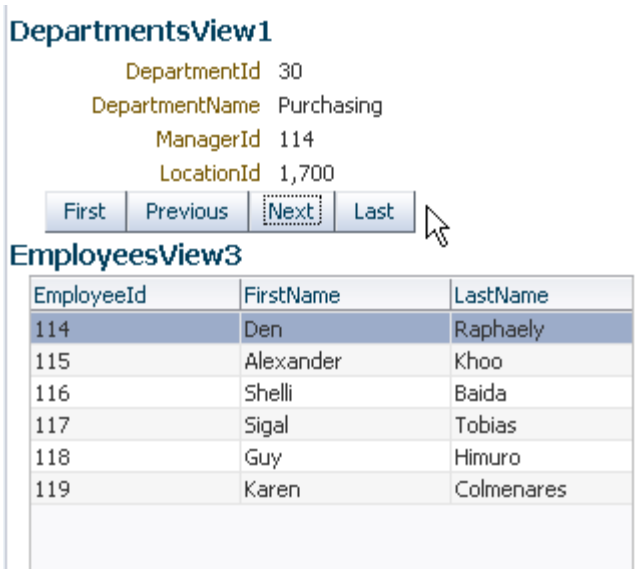


Figure 29–4 shows a tree that has multiple levels of nodes. You can use trees to display hierarchical information. In this example, a tree is used to display the folders and items in file system. The user can expand nodes to traverse down a branch of the tree to access the leaf items.

Figure 29–4 Multi-level Tree

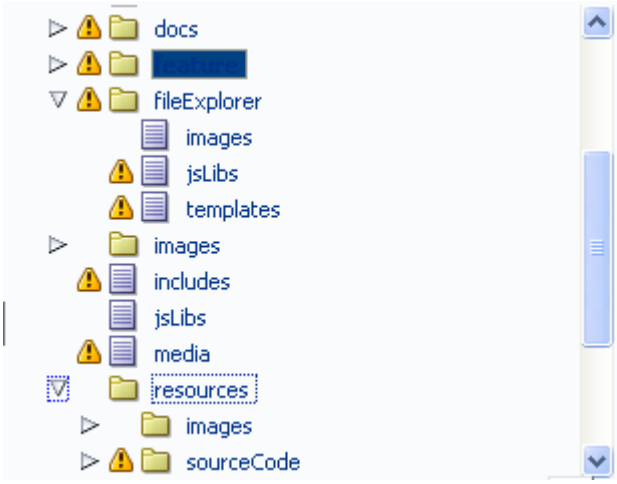
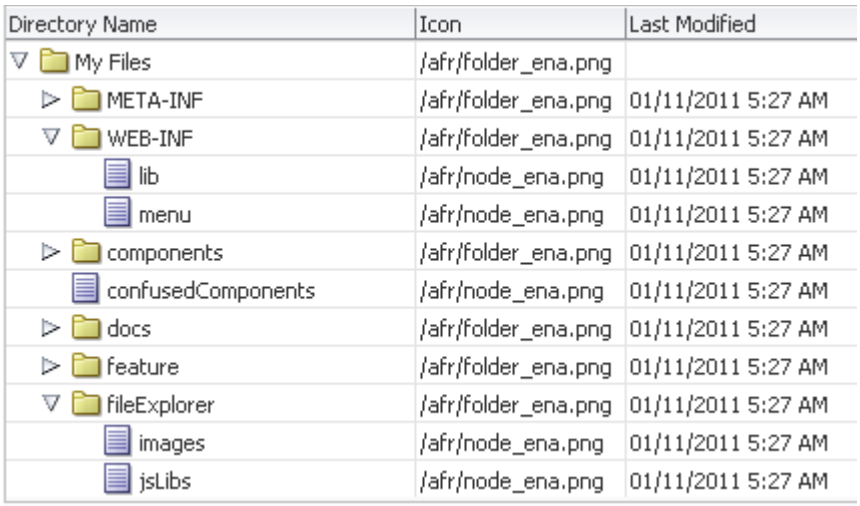


Figure 29–5 show the same file structure displayed using a tree table instead of a tree. In addition to the hierarchical node levels of a tree, a tree table can also display column information for each node. You can also focus on subtrees as well as collapsing and expanding the elements.

Figure 29–5 Tree Table



29.1.2 Additional Functionality for Master-Detail Tables, Forms, and Trees

You may find it helpful to understand other ADF features before you configure or use the ADF Model layer. Additionally, you may want to read about what you can do with your model layer configurations. Following are links to other functionality that may be of interest.

- For information about adding master-detail objects to the data model, see Section 5.6.4, "How to Enable Active Master-Detail Coordination in the Data Model."
- For information about the icons displayed on the Data Controls panel, see "Section 13.4.1, "How to Use the Data Controls Panel."
- If you want to modify the default forms or tables, see Section 26.3, "Creating a Basic Form" or Section 27.2, "Creating a Basic Table."
- For more information about creating view links, see Section 5.6.6, "How to Access the Detail Collection Using the View Link Accessor."
- For more information about creating and using managed beans, see Section 24.4, "Using a Managed Bean in a Fusion Web Application."
- For more information about creating listener methods, see the "Using ADF Faces Server Events" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
- For more information about finding the event source component, see the "How to Return the Original Source of the Event" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

29.2 Common Functionality for Master-Detail Tables, Forms, and Trees

Master-Detail tables, forms, trees, and tree tables requires a master-detail relationship to be established in the model. When a master-detail relationship is present, it is reflected in the Data Controls panel.

29.2.1 Identifying Master-Detail Objects on the Data Controls Panel

You can declaratively create pages that display master-detail data using the Data Controls panel. The Data Controls panel displays master-detail related objects in a hierarchy that mirrors the one you defined in the application module data model, where the detail objects are children of the master objects. For information about adding master-detail objects to the data model, see Section 5.6.4, "How to Enable Active Master-Detail Coordination in the Data Model."

To display master-detail objects as form or table objects, drag the detail object from the Data Controls panel and drop it on the page. Its master object is automatically created on the page.

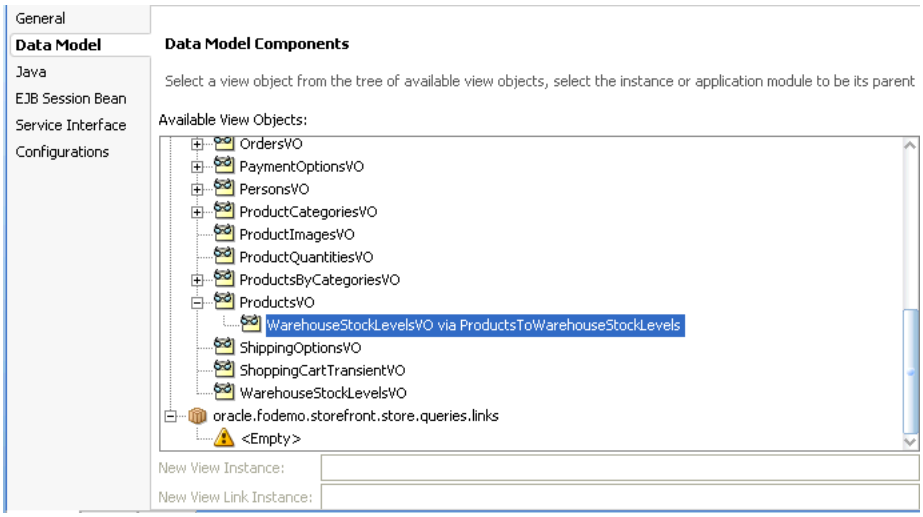
Figure 29–6 shows two master-detail related collections in the Data Controls panel of the Fusion Order Demo application. The `Products` collection is an instance of the `ProductsVO` view object, and the `WarehouseStockLevels` collection, which appears as a child of the `Products` collection, is an instance of the `WarehouseStockLevelsVO` view object.

> **Note:** The master-detail hierarchy displayed in the Data Controls panel does not reflect the cardinality of the relationship (that is, one-to-many, one-to-one, many-to-many). The hierarchy simply shows which collection (the master) is being use to retrieve one or more objects from another collection (the detail).

Figure 29–6 Master-Detail Objects in the Data Controls Panel

The master-detail hierarchy on the Data Controls panel reflects the hierarchy defined in the application module data model, as shown in Figure 29–7. The hierarchy was established by creating a view link from the `ProductsVO` view object to the `WarehouseStockLevelsVO` view object. Next, an instance of the resulting detail view object, `WarehouseStockLevelsVO via ProductsToWarehouseStockLevels`, was added to the application module data model shown in Figure 29–7.

Figure 29–7 Master-Detail Hierarchy Defined in the Application Module Data Model



In the Fusion Order Demo application, the view link between the `ProductsVO` view object and `WarehouseStockLevelsVO` view object is a one-way relationship. If the view link were bidirectional and both sets of master and detail view objects were added to the application module data model, then the Data Controls panel would also display the `WarehouseStockLevelsVO` collection at the same node level as the `Products` collection, and the detail instance of the `Products` collection as a child of the `WarehouseStockLevelsVO` collection.

> **Tip:** By default, when you define a view link using the Create View Link wizard, the source view object is the master and the destination view object is the detail. However, if you choose to generate accessors in both the source and the destination view objects, then the master-detail relationship is bidirectional. If both sets of master-detail view objects resulting from a bidirectional view link are added to the application module data model, then instances of both sets of view objects will appear independently on the Data Controls panel.

For more information about the icons displayed on the Data Controls panel, see "Section 13.4.1, "How to Use the Data Controls Panel."

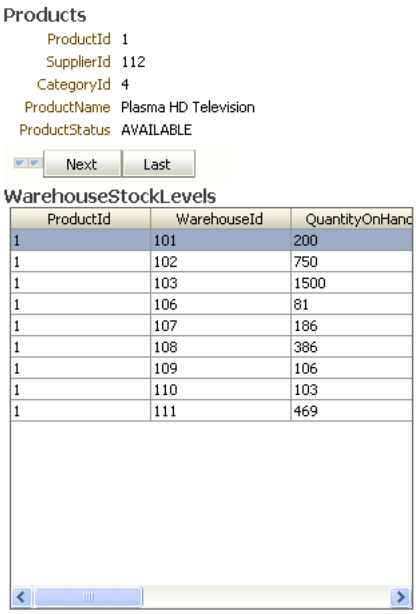
29.3 Using Tables and Forms to Display Master-Detail Objects

You can create a master-detail browse page in a single declarative action using the Data Controls panel. All you have to do is drop the detail collection on the page and choose the type of widget you want to use.

The prebuilt master-detail widgets available from the Data Controls panel include range navigation that enables the end user to scroll through the data objects in collections. You can delete unwanted attributes by removing the text field or column from the page.

Figure 29–8 shows an example of prebuilt master-detail widget, which displays products information in a form at the top of the page and stock levels in a table at the bottom of the page. When the user clicks the **Next** button to scroll through the records in the master data at the top of the page, the page automatically displays the related detail data.

Figure 29–8 Prebuilt Data Controls Panel Master-Detail Widget



29.3.1 How to Display Master-Detail Objects in Tables and Forms

If you do not want to use the prebuilt master-detail widgets, you can drag and drop the master and detail objects individually from the Data Controls panel as tables and forms on a single page or on separate pages.

The Data Controls panel enables you to create both the master and detail widgets on one page with a single declarative action using prebuilt master-detail forms and tables. For information about displaying master and detail data on separate pages, see Section 29.3.4, "What You May Need to Know About Displaying Master-Detail Widgets on Separate Pages."

Before you begin:

It may help to understand the options that are available to you when you create a master-detail table and form. For more information, see Section 29.3, "Using Tables and Forms to Display Master-Detail Objects."

You may also find it useful to understand additional functionality that can be used with master-detail tables and trees. For more information, see Section 29.1.2, "Additional Functionality for Master-Detail Tables, Forms, and Trees."

To create a master-detail page using the prebuilt ADF master-detail forms and tables:

1. From the Data Controls panel, locate the detail object, as described in Section 29.2.1, "Identifying Master-Detail Objects on the Data Controls Panel."
2. Drag and drop the detail object onto the JSF page.

   Note: If you want to create an editable master-detail form, drop the master object and the detail object separately on the page.

3. In the context menu, choose one of the following **master-details**UI components:

- **ADF Master Table, Detail Form**: Displays the master objects in a table and the detail objects in a read-only form under the table.

  When a specific data object is selected in the master table, the first related detail data object is displayed in the form below it. The user must use the form navigation to scroll through each subsequent detail data object.

- **ADF Master Form, Detail Table**: Displays the master objects in a read-only form and the detail objects in a read-only table under the form.

  When a specific master data object is displayed in the form, the related detail data objects are displayed in a table below it.

- **ADF Master Form, Detail Form**: Displays the master and detail objects in separate forms.

  When a specific master data object is displayed in the top form, the first related detail data object is displayed in the form below it. The user must use the form navigation to scroll through each subsequent detail data object.

- **ADF Master Table, Detail Table**: Displays the master and detail objects in separate tables.

  When a specific master data object is selected in the top table, the first set of related detail data objects is displayed in the table below it.

If you want to modify the default forms or tables, see Section 26.3, "Creating a Basic Form" or Section 27.2, "Creating a Basic Table."

29.3.2 What Happens When You Create Master-Detail Tables and Forms

When you drag and drop a collection from the Data Controls panel, JDeveloper does many things for you, including adding code to the JSF page and the corresponding entries in the page definition file. For a full description of what happens and what is created when you use the Data Controls panel, see Section 13.4.1, "How to Use the Data Controls Panel."

29.3.2.1 Code Generated in the JSF Page

The JSF code generated for a prebuilt master-detail widget is similar to the JSF code generated when you use the Data Controls panel to create a read-only form or table. If you are building your own master-detail widgets, you might want to consider including similar components that are automatically included in the prebuilt master-detail tables and forms.

The tables and forms in the prebuilt master-detail widgets include a `panelHeader` tag that contains the fully qualified name of the data object populating the form or table. You can change this label as needed using a string or an EL expression that binds to a resource bundle.

If there is more than one data object in a collection, a form in a prebuilt master-detail widget includes four `commandButton` tags for range navigation: `First`, `Previous`, `Next`, and `Last`. These range navigation buttons enable the user to scroll through the data objects in the collection. The `actionListener` attribute of each button is bound to a data control operation, which performs the navigation. The `execute` property used in the `actionListener` binding, invokes the operation when the button is clicked. (If the form displays a single data object, JDeveloper automatically omits the range navigation components.) For more information about range navigation, see Section 26.4, "Incorporating Range Navigation into Forms."

Tip: If you drop an **ADF Master Table, Detail Form** or **ADF Master Table, Detail Table** widget on the page, the parent tag of the detail component (for example, `panelHeader` tag or `table` tag) automatically has the `partialTriggers` attribute set to the `id` of the master component (see Section 27.4.1, "How to Create an Input Table" for more information about partial triggers). At runtime, the `partialTriggers` attribute causes only the detail component to be rerendered when the user makes a selection in the master component, which is called *partial rendering*. When the master component is a table, ADF uses partial rendering, because the table does not need to be rerendered when the user simply makes a selection in the facet. Only the detail component needs to be rerendered to display the new data.

29.3.2.2 Binding Objects Defined in the Page Definition File

Example 29–1 shows the page definition file created for a master-detail page that was created by dropping the `WarehouseStockLevels` collection, which is a detail object under the `Products` object, on the page as an **ADF Master Form, Detail Table**.

The `executables` element defines two iterators: one for the product (the master object) and one for the WarehouseStockLevels (the detail object). The underlying view link from the master view object to the detail view object establishes the relationship between the two iterators. At runtime, the ADF data model and the row set iterator for the detail view object instance keep the row set of the detail view object refreshed to the correct set of rows for the current master row as that current row changes (for more information, see Section 29.3.3, "What Happens at Runtime: ADF Iterator for Master-Detail Tables and Forms").

The `bindings` element defines the value bindings for the form and the table. The attribute bindings that populate the text fields in the form are defined in the `attributeValues` elements. The `id` attribute of the `attributeValues` element contains the name of each data attribute, and the `IterBinding` attribute references an iterator binding to display data from the master object in the text fields.

The attribute bindings that populate the text fields in the form are defined in the `attributeValues` elements. The `id` attribute of the `attributeValues` element contains the name of each data attribute, and the `IterBinding` attribute references an iterator binding to display data from the master object in the text fields.

The range navigation buttons in the form are bound to the action bindings defined in the `action` elements. As in the attribute bindings, the `IterBinding` attribute of the action binding references the iterator binding for the master object.

The table, which displays the detail data, is bound to the table binding object defined in the `table` element. The `IterBinding` attribute references the iterator binding for the detail object.

For more information about the elements and attributes of the page definition file, see Section A.8, "pageNamePageDef.xml."

Example 29–1 Binding Objects Defined in the Page Definition for a Master-Detail Page

```
<executables>
    <iterator Binds="Products" RangeSize="25"
              DataControl="StoreServiceAMDataControl" id="ProductsIterator"/>
    <iterator Binds="WarehouseStockLevels" RangeSize="25"
              DataControl="StoreServiceAMDataControl"
              id="WarehouseStockLevelsIterator"/>
</executables>
```

```
<bindings>
    <methodAction id="findHelpTextById" RequiresUpdateModel="true"
                  Action="invokeMethod" MethodName="findHelpTextById"
                  IsViewObjectMethod="false"
                  DataControl="LookupServiceAMDataControl"
                  InstanceName="LookupServiceAMDataControl.dataProvider"
                  ReturnName="LookupServiceAMDataControl.methodResults.
                      findHelpTextById_LookupServiceAMDataControl_
                      dataProvider_findHelpTextById_result">
                  <NamedData NDName="helpId" NDType="java.lang.Long"
                       NDOption="2"/>
    </methodAction>
    <tree IterBinding="ProductsIterator" id="Products">
      <nodeDefinition DefName="oracle.fodemo.storefront.store.queries.ProductsVO">
        <AttrNames>
          <Item Value="ProductId"/>
          <Item Value="SupplierId"/>
          <Item Value="CategoryId"/>
          <Item Value="ProductName"/>
          ...
          <Item Value="CostPrice"/>
          <Item Value="DragId"/>
        </AttrNames>
      </nodeDefinition>
    </tree>
    <tree IterBinding="WarehouseStockLevelsIterator" id="WarehouseStockLevels">
      <nodeDefinition DefName="oracle.fodemo.storefront.store.
           queries.WarehouseStockLevelsVO">
        <AttrNames>
          <Item Value="ProductId"/>
          <Item Value="WarehouseId"/>
          ...
          <Item Value="ShippingClassCode"/>
        </AttrNames>
      </nodeDefinition>
    </tree>
</bindings>
```

29.3.3 What Happens at Runtime: ADF Iterator for Master-Detail Tables and Forms

At runtime, an ADF iterator determines which row from the master table object to display in the master-detail form. When the form first displays, the first master table object row appears highlighted in the master section of the form. Detail table rows that are associated with the master row display in the detail section of the form.

As described in Section 29.3.2.2, "Binding Objects Defined in the Page Definition File," ADF iterators are associated with underlying `row setIterator` objects. These iterators manage which data objects, or rows, currently display on a page. At runtime, the row set iterators manage the data displayed in the master and detail components.

Both the master and detail row set iterators listen to row set navigation events, such as the user clicking the range navigation buttons, and display the appropriate row in the UI. In the case of the default master-detail components, the row set navigation events are the command buttons on a form (**First**, **Previous**, **Next**, **Last**).

The row set iterator for the detail collection manages the synchronization of the detail data with the master data. Because of the underlying view link from the master view

object to the detail view object, the detail row set iterator listens for row navigation events in both the master and detail collections. If a row set navigation event occurs in the master collection, the detail row set iterator automatically executes and returns the detail rows related to the current master row.

29.3.4 What You May Need to Know About Displaying Master-Detail Widgets on Separate Pages

The default master-detail components display the master-detail data on a single page. However, using the master and detail objects on the Data Controls panel, you can also display the collections on separate pages, and still have the binding iterators manage the synchronization of the master and detail objects.

For example, in the Fusion Order Demo application, a product table and product details are displayed on the home page. However, the page could display the product table only. It could provide a button called **Details**. If the user clicked the **Details** button, the application would navigate to a new page that displays all the related details in a list. A button on the list's page would enable the user to return to the service request page.

To display master-detail objects on separate pages, create two pages, one for the master object and one for the detail object, using the individual tables or forms available from the Data Controls panel. Remember that the detail object iterator manages the synchronization of the master and detail data. Be sure to drag the appropriate detail object from the Data Controls panel when you create the page to display the detail data. For more information, see Section 29.2.1, "Identifying Master-Detail Objects on the Data Controls Panel."

To handle the page navigation, create an ADF task flow, and then add two view activities to it, one for the master page and one for the detail page. (See Section 18.2, "Creating a Task Flow" for information about associating a View activity with an existing JSF page). Add command buttons or links to each page, or use the default **Submit** button available when you create a form or table using the Data Controls panel. Each button must specify a navigation rule outcome value in the `action` attribute. In the `task-flow-defintion.xml` file, add a navigation rule from the master data page to the detail data page, and another rule to return from the detail data page to the master data page. The `from-outcome` value in the navigation rules must match the outcome value specified in the action attribute of the buttons. For information about adding navigation between pages, see Section 18.2, "Creating a Task Flow."

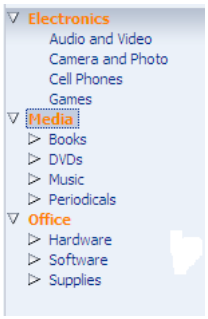
29.4 Using Trees to Display Master-Detail Objects

In addition to tables and forms, you can also display master-detail data in hierarchical trees. The ADF Faces `tree` component is used to display hierarchical data. It can display multiple root nodes that are populated by a binding on a master object. Each root node in the tree may have any number of branches, which are populated by bindings on detail objects. A tree can have multiple levels of nodes, each representing a detail object of the parent node. Each node in the tree is indented to show its level in the hierarchy.

The `tree` component includes mechanisms for expanding and collapsing the tree nodes; however, it does not have focusing capability. If you need to use focusing, consider using the ADF Faces `treeTable` component (for more information, see Section 29.5, "Using Tree Tables to Display Master-Detail Objects"). By default, the icon for each node in the tree is a folder; however, you can use your own icons for each level of nodes in the hierarchy.

Figure 29–9 shows an example of a tree located on the `home.jspx` page of the Fusion Order Demo application. The tree displays two levels of nodes: root and branch. The root node displays parent product categories such as Media, Office, and Electronics. The branch nodes display and subcategories under each parent category, such as Hardware, Supplies, and Software under the Office parent category.

Figure 29–9 Databound ADF Faces Tree



29.4.1 How to Display Master-Detail Objects in Trees

A *tree* consists of a hierarchy of nodes, where each subnode is a branch off a higher level node. Each node level in a databound ADF Faces `tree` component is populated by a different data collection. In JDeveloper, you define a databound tree using the Edit Tree Binding dialog, which enables you to define the rules for populating each node level in the tree. There must be one rule for each node level in the hierarchy. Each rule defines the following node-level properties:

- The data collection that populates that node level
- The attributes from the data collection that are displayed at that node level
- A view link accessor attribute that returns a detail object to be displayed as a branch of the current node level (for information about view link accessors, see Section 5.6.6.2, "Programmatically Accessing a Detail Collection Using the View Link Accessor")

To create the Browse tree on the Fusion Order Demo home page shown in Figure 29–9, a view object, `ParentProductCategories` was created to return a list of parent categories. Another view object, `ProductCategories` was created to return subcategories of products. A view link was created from the `ParentProductCategories` view object to the `ProductCategories` view object, thus establishing a master-detail relationship.

To add a third-level node, for example, a list of products under each subcategory, a view link would need to exist from the `ProductCategories` object to the `Products` view object. For more information about creating view links, see Section 5.6.6, "How to Access the Detail Collection Using the View Link Accessor."

In the case where a branch of the tree is recursive, a single view object accompanied by a self-referential view link must be defined in the data model project. For example, in a collection defined by `EmployeesView`, the root node of each branch is specified by the `ManagerId` attribute and the child nodes of the same branch are the employees (also known as "direct reports") who are related to the `ManagerId`. The source and destination view object named by the self-referential view link are both defined as `EmployeesView` with the destination renamed to `DirectReports` for clarify. For more information about creating self-referential view links, see Section 5.7.1, "How to Create a Recursive Master-Detail Hierarchy for an Entity-Based View Object."

Before you begin:

It may help to understand the options that are available to you when you create a master-detail table and form. For more information, see Section 29.4, "Using Trees to Display Master-Detail Objects."

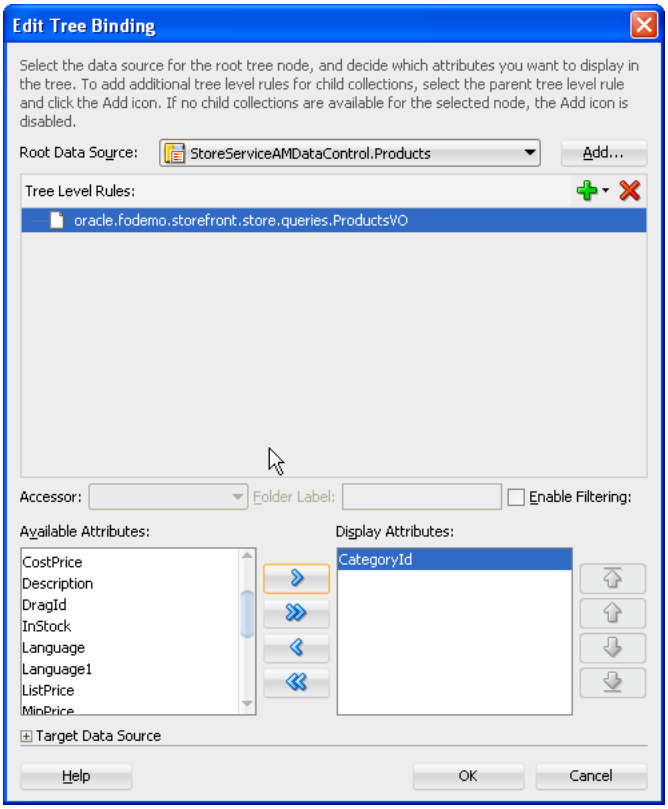
You may also find it useful to understand additional functionality that can be used with master-detail tables and trees. For more information, see Section 29.1.2, "Additional Functionality for Master-Detail Tables, Forms, and Trees."

To display master-detail objects in a tree:

1. Drag the master object from the Data Controls panel, and drop it onto the page. This should be the master data that will represent the root level of the tree.
2. In the context menu, choose **Trees** > **ADF Tree**.

   JDeveloper displays the Edit Tree Binding dialog, as shown in Figure 29–10. You use the binding editor to define a rule for each level that you want to appear in the tree.

Figure 29–10 Edit Tree Binding Dialog



3. In the **Root Data Source** dropdown list, select the data collection that will populate the root node level. This will be the master data collection. By default, this is the same collection that you dragged from the Data Controls panel to create the tree, which was a master collection.

   Tip: If you don't see the data collection you want in the **Root Data Source** list, click the **Add** button. In the Add Data Source dialog, select a data control and an iterator name to create a new data source.

4. Click the **+** icon to add the root data source you selected to the **Tree Level Rules** list.
5. In the **Tree Level Rules** list, select the data source you just added.
6. In the **Accessor** dropdown list, select a view link accessor attribute.

   The list displays only the accessor attributes that return the detail collections for the master collection you selected. For example, if you are defining the `ParentProductCategories` node level and you want to add a detail level that displays all subcategories under each parent category, you would select the accessor attribute that returns the `ProductCategories` collection.

   If you select **<none>**, the node will not expand to display any detail collections, thus ending the branch.

   View link accessor attributes, which return data collections, are generated when you create a view link. The **Accessor** field displays all accessor attributes that return detail collections for the master collection selected in the **Tree Level Rules** list. For more information about view objects, view links, and view link accessors, see Section 5.6.6, "How to Access the Detail Collection Using the View Link Accessor."
7. Select an attribute in the **Available Attributes** list and move it to the **Display Attributes** list.

   The attribute will be used to display nodes at the master level. For example, for the Parent Product Categories level, you might select the `Categories` attribute.

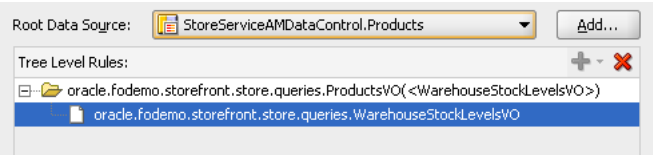
   When you are finished, the Tree Binding Editor should contain values similar to those in Figure 29–10.

   After defining a rule for the master level, you must next define a second rule for the detail level that will appear under the master level in the tree.

   For example, in the sample tree shown in Figure 29–9, the first rule added to the tree binding editor populates the parent category nodes (Media, Office, and Electronics). The detail level rule populates the product category nodes (for example, Hardware, Supplies, and Software under the Media parent category). ‘
8. To add a second rule, click the **Add** icon above the **Tree Level Rules** list.

   A detail data source should appear automatically under the master data source, as shown in Figure 29–11.

Figure 29–11 Master-Detail Tree Level Rules

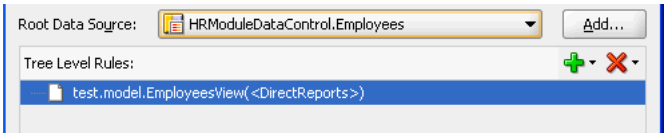


For example, if you specified `Products` as the master Root Data Source, `WarehouseStockLevels` will automatically appear underneath in the **Tree Level Rules** list, because the two data sources share a master-detail relationship.

If you are creating a tree with a recursive master-detail hierarchy, then you only need to define a rule that specifies a data source with a self accessor. A recursive tree displays root nodes based on a single collection and displays the child nodes from the attributes of a self accessor that recursively fetches data from that collection. The recursive tree differs from a typical master-detail tree because it

requires only a single rule to define the branches of the tree. A recursive data source should display the data source followed by the name of the self accessor in brackets, as shown in Figure 29–12.

Figure 29–12 Recursive Tree Level Rule



For example, in a collection defined by `EmployeesView`, the root node of each branch could be specified by the `ManagerId` for the employee and the child nodes of the same branch are the employees who are related to the `ManagerId`, as specified by the self accessor `DirectReports`.

9. Click **OK**.
10. You can add data sources to the **Tree Level Rules** list to increase the number of nodes that display in the tree. The order of the remaining data sources should follow the hierarchy of the nodes you want to display in the tree.

29.4.2 What Happens When You Create an ADF Databound Tree

When you drag and drop from the Data Controls panel, JDeveloper does many things for you. For a full description of what happens and what is created when you use the Data Controls panel, see Section 13.4.1, "How to Use the Data Controls Panel."

When you create a databound tree using the Data Controls panel, JDeveloper adds binding objects to the page definition file, and it also adds the tree tag to the JSF Page. The resulting UI component is fully functional and does not require any further modification.

29.4.2.1 Code Generated in the JSF Page

Example 29–2 shows the code generated in a JSF page when you use the Data Controls panel to create a tree. This sample tree displays two levels of nodes: parent product categories and product categories. The `ParentProductCategories` collection was used to populate the root node.

Example 29–2 Code Generated in the JSF Page for a Databound Tree

```
<af:tree id="productCategoriesTree" contentDelivery="immediate"
            selectionListener="#{homePageBean.
                  productCategoriesTreeSelectionListener}"
            rowSelection="single"
            value="#{bindings.ParentProductCategories.treeModel}"
            var="node" initiallyExpanded="true">
      <f:facet name="nodeStamp">
            <af:panelGroupLayout>
                  <af:outputText rendered="#{node.ParentCategoryId eq null}"
                        value="#{node.CategoryName}"
                           inlineStyle="color:#FF8000;font-weight:bold;"/>
                  <af:outputText rendered="#{node.ParentCategoryId ne null}"
                        value="#{node.CategoryName}"/>
            </af:panelGroupLayout>
      </f:facet>
</af:tree>
```

By default, the af:tree tag is created inside a form. The value attribute of the tree tag contains an EL expression that binds the tree component to the ParentProductCategories tree binding object in the page definition file. The treeModel property in the binding expression refers to an ADF class that defines how the tree hierarchy is displayed, based on the underlying data model. The var attribute provides access to the current node.

In the f:facet tag, the nodeStamp facet is used to display the data for each node. Instead of having a component for each node, the tree repeatedly renders the nodeStamp facet, similar to the way rows are rendered for the ADF Faces table component.

The ADF Faces tree component uses an instance of the oracle.adf.view.faces.model.PathSet class to display expanded nodes. This instance is stored as the treeState attribute on the component. You may use this instance to programmatically control the expanded or collapsed state of an element in the hierarchy. Any element contained by the PathSet instance is deemed expanded. All other elements are collapsed.

29.4.2.2 Binding Objects Defined in the Page Definition File

Example 29–3 shows the binding objects defined in the page definition file for the ADF databound tree.

Example 29–3 Binding Objects Defined in the Page Definition File for a Databound Tree

```
<executables>
    <iterator Binds="ParentProductCategories" RangeSize="25"
              DataControl="StoreServiceAMDataControl"
              id="ParentProductCategoriesIterator"/>
</executables>
<bindings>
    <tree IterBinding="ParentProductCategoriesIterator"
          id="ParentProductCategories">
      <nodeDefinition
          DefName="oracle.fodemo.storefront.store.queries.ProductCategoriesVO">
        <AttrNames>
          <Item Value="CategoryId"/>
        </AttrNames>
        <Accessors>
          <Item Value="ProductCategoriesVO"/>
          <Item Value="ParentCategoryIdProductCategoriesVO"/>
        </Accessors>
      </nodeDefinition>
    </tree>
</bindings>
```

The page definition file contains the rule information defined in the Tree Binding Editor. In the executables element, notice that although the tree displays two levels of nodes, only one iterator binding object is needed. This iterator iterates over the master collection, which populates the root nodes of the tree. The accessor you specified in the node rules returns the detail data for each branch node.

The tree element is the value binding for all the attributes displayed in the tree. The iterBinding attribute of the tree element references the iterator binding that populates the data in the tree. The AttrNames element within the tree element

defines binding objects for *all* the attributes in the master collection. However, the attributes that you select to appear in the tree are defined in the `AttrNames` elements within the `nodeDefinition` elements.

The `nodeDefinition` elements define the rules for populating the nodes of the tree. There is one `nodeDefinition` element for each node, and each one contains the following attributes and subelements:

- `DefName`: An attribute that contains the fully qualified name of the data collection that will be used to populate the node.
- `id`: An attribute that defines the name of the node.
- `AttrNames`: A subelement that defines the attributes that will be displayed in the node at runtime.
- `Accessors`: A subelement that defines the accessor attribute that returns the next branch of the tree.

The order of the `nodeDefintion` elements within the page definition file defines the order or level of the nodes in the tree, were the first `nodeDefinition` element defines the root node. Each subsequent `nodeDefinition` element defines a sub-node of the one before it.

For more information about the elements and attributes of the page definition file, see Appendix A.8, "pageNamePageDef.xml."

29.4.3 What Happens at Runtime: Displaying an ADF Databound Tree

`Tree` components use `org.apache.myfaces.trinidad.model.TreeModel` to access data. This class extends `CollectionModel`, which is used by the ADF Faces `table` component to access data. For more information about the `TreeModel` class, refer to the ADF Faces Javadoc.

When a page with a `tree` is displayed, the iterator binding on the tree populates the root nodes. When a user collapses or expands a node to display or hide its branches, a `DisclosureEvent` event is sent. The `isExpanded` method on this event determines whether the user is expanding or collapsing the node. The `DisclosureEvent` event has an associated listener.

The `DisclosureListener` attribute on the `tree` is bound to the accessor attribute specified in the node rule defined in the page definition file. This accessor attribute is invoked in response to the `DisclosureEvent` event; in other words, whenever a user expands the node the accessor attribute populates the branch nodes.

29.5 Using Tree Tables to Display Master-Detail Objects

Use the ADF Faces `treeTable` component to display a hierarchy of master-detail collections in a table. The advantage of using a `treeTable` component rather than a `tree` component is that the `treeTable` component provides a mechanism that enables users to focus the view on a particular node in the tree.

For example, you can create a tree table that displays three levels of nodes: countries, states or provinces, and cities. Each root node represents an individual country. The branches off the root nodes display the state or provinces in the country. Each state or province node branches to display the cities contained in it.

As with trees, to create a tree table with multiple nodes, it is necessary to create view links between the view objects. The view links establish the master-detail relationships For example, to create a tree table with three levels of country, state, and city, it was

necessary to create view links from the `CountryCodes` object to the `StatesandProvinces` view object, and another view link from the `StatesandProvinces` view object to the `Cities` view object. For more information about creating view links, see Section 5.6, "Working with Multiple Tables in a Master-Detail Hierarchy."

A databound ADF Faces `treeTable` displays one root node at a time, but provides navigation for scrolling through the different root nodes. Each root node can display any number of branch nodes. Every node is displayed in a separate row of the table, and each row provides a focusing mechanism in the leftmost column.

You can edit the following `treeTable` component properties in the Property Inspector:

- Range navigation: The user can click the **Previous** and **Next** navigation buttons to scroll through the root nodes.
- List navigation: The list navigation, which is located between the **Previous** and **Next** buttons, enables the user to navigate to a specific root node in the data collection using a selection list.
- Node expanding and collapsing mechanism: The user can open or close each node individually or use the **Expand All** or **Collapse All** command links. By default, the icon for opening and closing the individual nodes is an arrowhead with a plus or minus sign. You can also use a custom icon of your choosing.
- Focusing mechanism: When the user clicks on the focusing icon (which is displayed in the leftmost column) next to a node, the page is redisplayed showing only that node and its branches. A navigation link is provided to enable the user to return to the parent node.

29.5.1 How to Display Master-Detail Objects in Tree Tables

The steps for creating an ADF Faces databound tree table are exactly the same as those for creating an ADF Faces databound tree, except that you drop the data collection as an **ADF Tree Table** instead of an **ADF Tree**.

29.5.2 What Happens When You Create a Databound Tree Table

When you drag and drop from the Data Controls panel, JDeveloper does many things for you. For a full description of what happens and what is created when you use the Data Controls panel, see Section 13.4.1, "How to Use the Data Controls Panel."

When you create a databound tree table using the Data Controls panel, JDeveloper adds binding objects to the page definition file, and it also adds the `treeTable` tag to the JSF Page. The resulting UI component is fully functional and does not require any further modification.

29.5.2.1 Code Generated in the JSF Page

Example 29–4 shows the code generated in a JSF page when you use the Data Controls panel to create a tree table. This sample tree table displays two levels of nodes: products and stock levels.

By default, the `treeTable` tag is created inside a form. The `value` attribute of the `tree` table tag contains an EL expression that binds the `tree` component to the binding object that will populate it with data. The `treeModel` property refers to an ADF class that defines how the tree hierarchy is displayed, based on the underlying data model. The `var` attribute provides access to the current node.

Example 29–4 Code Generated in the JSF Page for a Databound ADF Faces Tree Table

```
<af:form>
      <af:treeTable value="#{bindings.Products.treeModel}" var="node"
                    selectionListener="#{bindings.Products.treeModel.makeCurrent}"
                    rowSelection="single">
        <f:facet name="nodeStamp">
          <af:column customizationId="column1">
            <af:outputText value="#{node}"/>
          </af:column>
        </f:facet>
        <f:facet name="pathStamp">
          <af:outputText value="#{node}"/>
        </f:facet>
      </af:treeTable>
</af:form>
```

In the `facet` tag, the `nodeStamp` facet is used to display the data for each node. Instead of having a component for each node, the tree repeatedly renders the `nodeStamp` facet, similar to the way rows are rendered for the ADF Faces `table` component. The `pathStamp` facet renders the column and the path links above the table that enable the user to return to the parent node after focusing on a detail node.

29.5.2.2 Binding Objects Defined in the Page Definition File

The binding objects created in the page definition file for a tree table are exactly the same as those created for a tree.

29.5.3 What Happens at Runtime: Events

`Tree` components use `oracle.adf.view.faces.model.TreeModel` to access data. This class extends `CollectionModel`, which is used by the ADF Faces `table` component to access data. For more information about the `TreeModel` class, refer to the ADF Faces Javadoc.

When a page with a tree table is displayed, the iterator binding on the `treeTable` component populates the root node and listens for a row navigation event (such as the user clicking the **Next** or **Previous** buttons or selecting a row from the range navigator). When the user initiates a row navigation event, the iterator displays the appropriate row.

If the user changes the view focus (by clicking on the component's focus icon), the `treeTable` component generates a focus event (`FocusEvent`). The node to which the user wants to change focus is made the current node before the event is delivered. The `treeTable` component then modifies the `focusPath` property accordingly. You can bind the `FocusListener` attribute on the tree to a method on a managed bean. This method will then be invoked in response to the focus event.

When a user collapses or expands a node, a disclosure event (`DisclosureEvent`) is sent. The `isExpanded` method on the disclosure event determines whether the user is expanding or collapsing the node. The disclosure event has an associated listener, `DisclosureListener`. The `DisclosureListener` attribute on the tree table is bound to the accessor attribute specified in the node rule defined in the page definition file. This accessor attribute is invoked in response to a disclosure event (for example, the user expands a node) and returns the collection that populates that node.

The treeTable component includes **Expand All** and **Collapse All** links. When a user clicks one of these links, the treeTable sends a DisclosureAllEvent event. The isExpandAll method on this event determines whether the user is expanding or collapsing all the nodes. The table then expands or collapses the nodes that are children of the root node currently in focus. In large trees, the expand all command will not expand nodes beyond the immediate children. The ADF Faces treeTable component uses an instance of the oracle.adf.view.faces.model.PathSet class to determine expanded nodes. This instance is stored as the treeState attribute on the component. You can use this instance to programmatically control the expanded or collapsed state of a node in the hierarchy. Any node contained by the PathSet instance is deemed expanded. All other nodes are collapsed. This class also supports operations like addAll() and removeAll().

Like the ADF Faces table component, a treeTable component provides for range navigation. However, instead of using the rows attribute, the treeTable component uses a rowsByDepth attribute whose value is a space-separated list of non-negative numbers. Each number defines the range size for a node level on the tree. The first number is the root node of the tree, and the last number is for the branch nodes. If there are more branches in the tree than numbers in the rowsByDepth attribute, the tree uses the last number in the list for the remaining branches. Each number defines the limit on the number items displayed at one time in each branch. If you want to display all items in a branch, specify 0 in that position of the list.

For example, if the rowsByDepth attribute is set to 0 0 3, all root nodes will be displayed, all direct children of the root nodes will be displayed, but only three nodes will display per branch after that. The treeTable component includes links to navigate to additional nodes, enabling the user to display the additional nodes.

For more information about the ADF Faces treeTable component, refer to the oracle.adf.view.faces.component.core.data.CoreTreeTable class in the ADF Faces Javadoc.

29.5.4 How to Use the TargetIterator Property

You can expand a node binding in the page definition editor to view the page's node Definition elements. These are the same tree binding rules that you can configure in the tree binding dialog.

For each node definition (rule), you can specify an optional TargetIterator property. Its value is an EL expression that is evaluated at runtime when the user selects a row in the tree. The EL expression evaluates an iterator binding in the current binding container. The iterator binding's view row key attributes match (in order, number, and data type) the view row key of the iterator from which the nodeDefinition type's rows are retrieved for the tree.

At runtime, when the tree control receives a selectionChanged event, it passes in the list of keys for each level of the tree. These keys uniquely identify the selected node.

The tree binding starts at the top of the tree. For each tree level whose key is present in the Currently Selected Tree Node Keys list, if there is a TargetIterator property configured for that nodeDefinition, the tree binding performs a setCurrentRowWithKey() operation on the selected target iterator. It uses the key from the appropriate level of the Currently Selected Tree Node Keys list.

For example, you may have created DeptEO and EmpEO entity objects, and created view links based on these entity objects. The view link accessor in the DeptVO object that returns the linked collection of employees in that department is named EmployeesInDepartment. The application module will have a DepartmentsTree

view object instance of type `DeptVO`. It will also have an `EditDepartment` view object instance of type `DeptVO`, and a view link for the `EditEmployees` view object instance of type `EmpVO`.

Before you begin:

It may help to understand the options that are available to you when you create a tree table. For more information, see Section 29.5, "Using Tree Tables to Display Master-Detail Objects."

You may also find it useful to understand additional functionality that can be used with master-detail tables and trees. For more information, see Section 29.1.2, "Additional Functionality for Master-Detail Tables, Forms, and Trees."

To use the property:

1. Drag the **DepartmentsTree** data collection from the Data Controls panel onto the page and choose **Create** > **Trees** > **ADF Tree**.
2. Configure tree binding rules to navigate the `EmployeesInDepartment` view link accessor attribute.

   This will access the children employee rows of the current department row.
3. Drag the **EditEmployees** detail view object instance to the page, and choose **Create** > **Master-Detail** > **ADF Master Form, Detail Form**.

   This creates a form to edit a department and a form to edit an employee in that department.
4. For the `DeptVO` node definition of the tree binding, configure the `TargetIterator` property to be `#{bindings.EditDepartmentIterator}`.
5. For the `EmpVO` node definition of the tree binding, configure the `TargetIterator` property to be `#{bindings.EditEmployeesIterator}`.

   When you run the master-detail form, clicking on an employee in any department in the tree will first set the current department row in the target iterator for the department of that selected employee. Then it will set the current employee row in the target iterator for the selected employee.

29.6 Using Selection Events with Trees and Tables

There may be cases when you need to determine which node in a tree or tree table has been selected in order to handle some processing in your application. For example, on the home page of the StoreFront module, when a user selects a category node in the **Browse** tree, a selection event is fired. The listener associated with this event needs to determine the product category of the node selected, and then to return all products whose category attribute matches that value.

Selection changes in a tree or table will also trigger contextual events for communicating between regions within a page. For more information, see Chapter 34, "Using Contextual Events."

29.6.1 How to Use Selection Events with Trees and Tables

To programmatically use selection events, you need to create a listener in a managed bean that will handle the selection event and perform the needed logic. You then need to bind the `selectionListener` attribute of the tree or table to that listener.

Before you begin:

It may help to understand the options that are available to you when you create a master-detail table and form. For more information, see Section 29.6, "Using Selection Events with Trees and Tables."

You may also find it useful to understand additional functionality that can be used with master-detail tables and trees. For more information, see Section 29.1.2, "Additional Functionality for Master-Detail Tables, Forms, and Trees."

To use selection events with trees and tables:

1. If one does not already exist, create a managed bean to contain the needed listener. For more information about creating and using managed beans, see Section 24.4, "Using a Managed Bean in a Fusion Web Application."
2. Create a listener method on the managed bean. For more information about creating listener methods, see the "Using ADF Faces Server Events" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*. Your listener should do the following:
   a. Access the component using the event source. Example 29–5 shows how the `productCategoriesTreeSelectionListener` method on the `HomeBean` managed bean accesses the tree that launched the selection event.

Example 29–5 Getting the Source of an Event

```
public void productCategoriesTreeSelectionListener(SelectionEvent evt) {
    RichTree tree = (RichTree)evt.getSource();
```

For more information about finding the event source component, see the "How to Return the Original Source of the Event" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

   b. Access the tree model to get the value of the model, use the `RowKeySet` object to get the currently selected node, and then set that as the current row on the model, as shown in Example 29–6. For more information about `RowKeySet` objects, see Section 29.6.2, "What Happens at Runtime: RowKeySet Objects and SelectionEvent Events."

Example 29–6 Setting the Current Row on a Tree Model

```
TreeModel model = (TreeModel)tree.getValue();
RowKeySet rowKeySet = evt.getAddedSet();
Object key = rowKeySet.iterator().next();
model.setRowKey(key);
```

   c. You can now add logic to execute against the currently selected row. For example, the `productCategoriesTreeSelectionListener` method uses the value binding of the selected row to determine the category ID, and then uses that value as the parameter for another method that when executed, returns all products with that category ID, as shown in Example 29–7.

Example 29–7 Returning Objects that Match a Given Attribute Value

```
JUCtrlValueBinding nodeBinding =
    (JUCtrlValueBinding)model.getRowData();
Number catId = (Number)nodeBinding.getAttribute("CategoryId");
_selectedCategory = (String)nodeBinding.getAttribute("CategoryName");
```

```
OperationBinding ob =
     ADFUtils.findOperation("ProductsByCategoriesExecuteWithParams");
ob.getParamsMap().put("category", catId);
ob.execute();
```

3. On the associated JSF page, select the tree or table component. In the Property Inspector, expand the **Behavior** section and set the value of the `SelectionListener` attribute to the listener method just created. You can use the **Edit** option from the dropdown method to declaratively select the bean and the method.

29.6.2 What Happens at Runtime: RowKeySet Objects and SelectionEvent Events

Whenever a user selects a node in a tree (or a row in a table), the component triggers selection events. A `selectionEvent` event reports which rows were just deselected and which rows were just selected. The current selection, that is, the selected row or rows, is managed by the `RowKeySet` object, which keeps track of all currently selected nodes by adding and deleting the associated key for the row into or out of the key set. When a user selects a new node, and the tree or table is configured for single selection, then the previously selected key is discarded and the newly selected key is added. If the tree or table is configured for multiple selection, then the newly selected keys are added to the set, and the previously selected keys may or may not be discarded, based on how the nodes were selected. For example, if the user pressed the `CTRL` key, then the newly selected nodes would be added to the current set.

30

Creating Databound Selection Lists and Shuttles

This chapter describes how to add selection lists and shuttle components to pages.

This chapter includes the following sections:

- Section 30.1, "About Selection Lists and Shuttles"
- Section 30.2, "Creating List of Values (LOV) Components"
- Section 30.3, "Creating a Selection List"
- Section 30.4, "Creating a List with Navigation List Binding"
- Section 30.5, "Creating a Databound Shuttle"

30.1 About Selection Lists and Shuttles

Selection list components present the user with a list of choices as input values. The selection list may be model-driven, obtained from a fixed list, or dynamically created at runtime. Selection list components can be rendered as lists, radio buttons, checkboxes, and list boxes. Some selection types can also be singular (select one) or multiple (select many).

Selection lists and shuttles work the same way as do standard JSF list components. ADF Faces list components, however, provide extra functionality such as support for label and message display, automatic form submission, and partial page rendering. When you present users with a list, they can readily see the available choices and make a selection. Picking from a list also eliminates typing errors that may occur.

List of values (LOV) components are UI components that allow the user to enter values by picking from a list that is generated by a query. The LOV displays inside a modal popup dialog that typically includes search capabilities. The `af:inputListOfValues` and `af:inputComboboxListOfValues` components, for example, offer additional features that are not available in selection lists, such as search fields inside the LOV modal dialog and queries based on multiple table columns. For more information, see Section G.2.6, "How to Create a Popup List of Values" and the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

List of values components offer more complex search and input capabilities by using the query component to create a popup search panel with a results table. The query component also has features such as auto-suggestion, smart list filtering, and custom facets for additional functionality.

When the user selects an item from a navigation list, a corresponding component bound to the list also changes its value in response to the selection. For example, when the user selects a product from a shopping list, the table that is bound to the products list updates to display the details of the selected product.

A shuttle allows the user to easily see the available items on an available list and the selected items in the selected list and to quickly move those items back and forth between the lists.

Shuttles provide a visual way to select items from an available list and at the same time see those selected items. ADF Faces provides the `selectManyShuttle` component and the `selectOrderShuttle` component, which allow reordering of the selected list. Both components can be implemented by adding code in a managed bean to populate the lists.

30.1.1 Selection Lists and Shuttles Use Cases and Examples

You can use a selection list component such as `selectOneChoice` for a single value input selection situation in which the list is relatively small. For example, you can use a `selectOneChoice` to select from a list of product colors or the type of credit card being used.

For larger lists and with more complex filtering, use an `af:inputListOfValues` or `af:inputComboboxListOfValues` component. These components use the query component to perform a transactional search to populate the list. For instance, you can use the `af:inputComboboxListOfValues` to select from a list of countries in an address input page.

Use the selectShuttle components when you want the user to be able to assemble a list of items and be able to select and unselect them. You should use the shuttle components when the number of items is large but not overwhelming to display. The user can review the selection and be able to iteratively select and unselect the items until the user is satisfied with the final selection. For instance, the available list could be the options available for a particular automobile and the user can shuttle the options he like to the selected list.

30.1.2 Additional Functionality for Selection Lists and Shuttles

You may find it helpful to understand other ADF features before you configure or use the ADF Model layer. Additionally, you may want to read about what you can do with your model layer configurations. Following are links to other functionality that may be of interest.

- For more information about using LOV components with search forms and the `af:query` component, see Section 31.1.2, "List of Values (LOV) Input Fields."
- You can use the LOV in a task flow that uses an isolated data control. For more information about shared and isolated data controls, see Section 20.4, "Sharing Data Controls Between Task Flows."
- LOVs are associated with a named view criteria. For more information about setting view criteria options, see Section 5.11.1, "How to Create Named View Criteria Declaratively."
- LOV components uses the `af:query` component to populate the selection list. For more information about the `af:query` component, see Chapter 31, "Creating ADF Databound Search Forms."

30.2 Creating List of Values (LOV) Components

List of values (LOV) components are input components that allow the user to enter values by picking from a list that is generated by a query. ADF Faces provides the `af:inputListOfValues` and `af:inputComboboxListOfValues` components. If you are creating a fully-featured LOV component, you must define the LOV in the view object as described in Section 5.12, "Working with List of Values (LOV) in View Object Attributes."

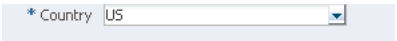
If you are using dependent LOVs as part of your search form, you must use them with the `af:query` component. For more information about using LOV components with search forms, see Section 31.1.2, "List of Values (LOV) Input Fields."

The `af:inputListOfValues` component has a search icon next to the search criteria field, as shown in Figure 30–1.

Figure 30–1 Search Criteria Input Field Defined as an inputListOfValues

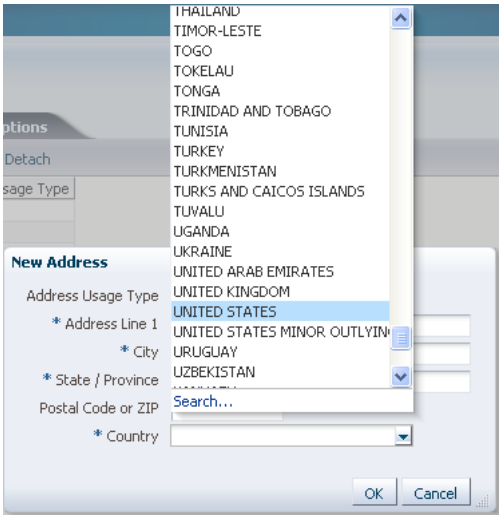
The `af:inputComboboxListOfValues` component has a dropdown icon next to the field, as shown in Figure 30–2 for the `PaymentOptionId` attribute.

Figure 30–2 Search Criteria Input Field Defined as an inputComboboxListOfValues



For `af:inputComboboxListOfValues`, clicking the dropdown icon displays the LOV dropdown list and either a **More** or a **Search** command link, as shown in Figure 30–3. A **Search** link appears when the LOV Search and Select popup dialog contains a search panel to refine the search. A **More** link appears when the popup dialog contains only the list-of-values table.

Figure 30–3 LOV Dropdown List with Search Link for inputComboboxListOfValues



The user can select any of the items in the dropdown list to populate the input field. The dropdown list includes a filtered list of values and an optional list of most recently used items. The width of each attribute in the dropdown list can be configured using the **DisplayWidth** UI hint for that attribute in the view object definition. The width of

the dropdown list will be the sum of the widths of all the attributes in the dropdown list.

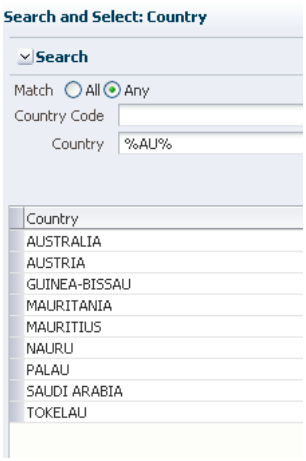
You can create custom content to be rendered in the Search and Select dialog using the `searchContent` facet. You can add components such as input text, `tables`, and `trees` to display the custom content. You will need to define the `returnPopupDataValue` attribute and implement a `returnPopupListener`. When you want to add the auto-complete feature with your custom popup, you can disable the popup by adding `launchPopupEvent.setLaunchPopup(false)` to your `LaunchPopupListener()` code. However, clicking on the **Search** link will still launch the Search and Select dialog. For more information about adding custom content, see the "Using List-of-Values Components" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

If the component's `readOnly` attribute is set to true, then the input field will not be rendered and the value cannot be changed. By default, the `readOnly` attribute is set to `false`, which also enables the `editMode` attribute to determine whether the user is permitted only to select a value from the list (`editMode` set to `select`) or whether the user can also enter a value into the input field (`editMode` set to `input`).

You can also set up the LOV component to display a list of selectable suggested items when the user types in a partial value. For example, when the user types in `Ca`, then a suggested list which partially matches `Ca` is displayed as a suggested items list. The user can select an item from the list to be entered into the input field. If there are no matches, a "No results found" message will be displayed. You add the `af:autoSuggestBehavior` tag from the Component Palette into the LOV and set the `selectedItem` attribute to the `suggestedItems` method implemented in ADF Model.

When the user clicks the **Search** or **More** link (or for `af:inputListOfValues`, the search icon), the LOV Search and Select dialog displays with the full list of values in a table format. The LOV Search and Select dialog launched from a **More** link is shown in Figure 30–4.

Figure 30–4 LOV Search and Select Dialog



The Search and Select popup dialog also presents a create function that allows the user to add a new row. Be aware that if you are creating a new record using the LOV Search and Select dialog, the new row will appear as an empty row in the table if the LOV has the **Query Automatically** control hint set to `false`. The content of the row will appear

when you perform a query using the **Search** button. Example 30–1 shows a code sample that uses a `commandLink` to invoke the create function.

Example 30–1 Create Function Sample Code

```
<f:facet name="customActions">
      <af:commandLink id="createLink" text="Create..."
         partialSubmit="true">
                <af:showPopupBehavior popupId="createSLPopup"
                alignId="createLink"/>
      </af:commandLink>
</f:facet>
```

If the LOV is part of a task flow that is using an isolated data control, and you use the create function to add a new record, the newly added record will not be displayed in the parent page. This is because the search region is also using an isolated data control scope, so the underlying view object updates are not displayed. For more information about shared and isolated data controls, see Section 20.4, "Sharing Data Controls Between Task Flows."

To programmatically refresh the LOV and display the view object updates, add a `returnListener` to the **Create** link with code similar to that shown in Example 30–2.

Example 30–2 Code to Refresh the LOV

```
public void refreshLOV() {
     BindingContainer bindings = this.getBindings();
     oracle.jbo.uicli.binding.JUCtrlListBinding lovBinding =
         (oracle.jbo.uicli.binding.JUCtrlListBinding)
          bindings.get("Description1");
     JUIteratorBinding lovIter = lovBinding.getIteratorBinding();
     RowSet rs = lovIter.getRowSetIterator().getRowSet();
     rs.executeQuery();

    //Add LOV as the partialTrigger

    AdfFacesContext.getCurrentInstance().addPartialTarget(this.getPlatformDesc());
```

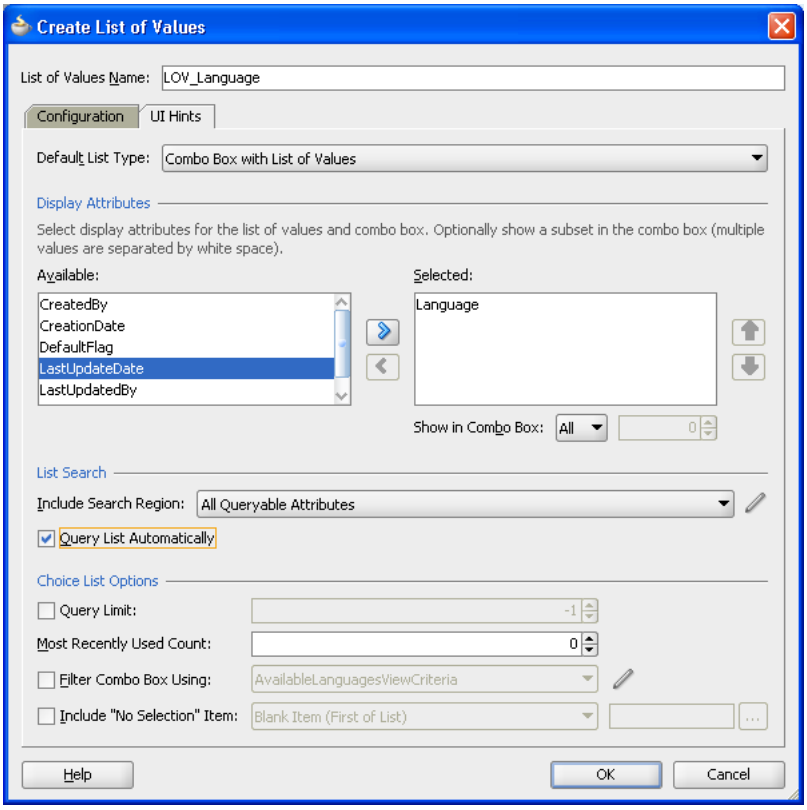
An LOV is associated with a data source via the view accessor. You can apply one or more view criteria to the view accessor associated with the LOV. The view accessor provides permanent filtering to the LOV data source. In addition to this permanent filtering, you may be able to apply other filters.

The LOV dialog may include a query panel to allow the user to enter criteria to search for the list of values, or it may contain only a table of results. When you define the LOV in the view object, you can use the UI hints to specify whether a search region should be displayed, which view criteria should be used as the search to populate the list of values, and which LOV component to use. Figure 30–5 shows the Create List of Values dialog and some of its options. In this example, the search region will be made available with the `AvailableLanguages1ViewCriteria` view criteria used for the query, and `af:inputComboboxListOfValues` as the component. Another useful option for the `af:inputComboboxListOfValues` is the **Show in Combo Box** option, which allows you to select the number of attributes to display in the dropdown list and in the Search and Select dialog. For more information on LOV UI hints, see Section 5.12.5, "How to Set User Interface Hints on a View Object LOV-Enabled Attribute."

For both `af:inputListOfValues` and `af:inputComboboxListOfValues`, if the user enters a partial match in the input search field and presses the Tab or Enter key, the LOV automatically launches the LOV Search and Select dialog and executes the query after applying an auto-generated view criteria with a single item representing the partial match value entered by the user. If there are matches, the Search and Select dialog displays all the entries matching the partially entered criteria. If there are no entries that match the entered partial match, then the dialog displays all the entries.

By default, in the `af:inputComboboxListOfValues` component *auto-complete* feature, the search is case-sensitive. If you want the search to be case-insensitive, create a view criteria to be associated with that LOV attribute. In the Create View Criteria dialog, deselect the **Ignore Case** checkbox for that view criteria item before you apply that view criteria to the LOV definition. For more information about setting view criteria options, see Section 5.11.1, "How to Create Named View Criteria Declaratively."

Figure 30–5 List of Values Dialog UI Hints Tab



You can also set up the LOV component to display a list of selectable suggested items when the user types in a partial value. For example, when the user types in `Ca`, then a suggested list which partially matches `Ca` is displayed as a suggested items list. The user can select an item from the list to be entered into the input field. You add this *auto-suggest behavior* feature by including an `af:autoSuggestBehavior` tag from the Component Palette in the LOV and set the `selectedItem` attribute to the `suggestedItems` method implemented in ADF Model.

If the LOV is an `af:inputComboboxListOfValues`, you can apply an additional view criteria to further filter the values in the dropdown list to create a *smart list*. If the **Filter Combo Box Using** UI hint is not applied, the dropdown list is filtered by the view criteria applied to the view accessor. Note that this smart list filter applies only to

the dropdown list. The full list is still used for validation and for searching from the LOV Search and Select popup dialog.

If both the auto-suggest behavior and smart list filter are enabled for an LOV, auto-suggest will search from the smart list first. If the user waits for two seconds without clicking, then auto-suggest will also search from the full list and append the results. You can also specify the number of suggested items returned by setting the `maxSuggestedItems` attribute (-1 indicates a complete list). If `maxSuggestedItems > 0`, a **More** link is rendered for the user to click on to launch the LOV's Search and Select dialog. Example 30–3 shows the code from an LOV with both auto-suggest behavior and a smart list.

Example 30–3 Auto-Suggest Behavior and Smart List

```
af:autoSuggestBehavior
    suggestItems="#{bindings.CountryName.suggestItems}"
    smartList="#{bindings.CountryName.smartList}"/>
    maxSuggestedItems="5"
```

You can define an attribute in the view object to be one or more LOVs. If multiple LOVs are defined on an attribute, each LOV has its own name, view accessor, data mappings, validator, and UI hints (except for the **Default List Type** hint, which is defined once for the attribute). To switch between LOVs to be used at runtime, an LOV switcher is used. The LOV switcher can be based on an existing attribute of type `String`, or created as a new `String` attribute solely for switching between LOVs. The LOV switcher returns the name of the LOV to use at runtime. For instance, you can create three LOVs for the price attribute, and designate the `CountryCode` attribute as the LOV switcher. At runtime, the value of `CountryCode` will switch the price attribute to use the LOV that reflects the country's price and currency.

If an attribute is defined as an LOV, you can set the **Support Multiple Value Selection** control hint in its view criteria to enable users to make multiple selections in the search criteria field. If multiple selection is enabled on an LOV attribute, and the `Equal to` or `Not equal to` operator is chosen, a `selectManyChoice` component will render in the query panel. The user can select multiple items as the search criteria.

If the value of an LOV depends on the value of another LOV, then the two LOVs are called *cascading* LOVs. For instance, the list of values for the City LOV depends on the value of the Country LOV that was selected. If the LOV is defined as a named bind variable, or if validation is enabled, the LOV query may behave differently depending on such conditions as whether null values for the named bind variable are acceptable. If the bind variable is null, you may want the LOV query to return an empty list. The view criteria's **Ignore Null Values** and **Validation** options can be used to define LOV query behavior.

If the view object has an LOV with a bind variable, you should set that view object bind variable's control hint to **Hide**. Otherwise, the bind variable will be displayed in the LOV Search and Select popup dialog. In the view object overview editor's **Query** tab, double-click the bind variable, click the **Control Hints** tab and select **Hide** from the **Display Hint** dropdown list.

Best Practice: An attribute that has been enabled as an LOV should not be used as search criteria in a manually created search form using the "Find Mode" iterator. Creating a search form in this way could result in SQL exception: `java.sql.SQLException: Attempt to set a parameter name that does not occur in the SQL`. It is best practice to use the query and quick query components for creating search forms. For more information, see Chapter 31, "Creating ADF Databound Search Forms."

Displaying an LOV list for a name attribute when the view object's key attribute is a numerical value requires mapping supplemental attributes on the LOV-enabled attribute. For example, to display a list of credit cards by name instead of by their payment option ID value, you can create an LOV on the name reference attribute and define the primary key attribute as a supplemental attribute to the LOV. For details about creating an LOV-enabled reference attribute, see Section 5.12.4, "How to Define an LOV to Display a Reference Attribute."

For more information about different usage scenarios and other view criteria options, see Section 5.11.1, "How to Create Named View Criteria Declaratively," and Section 5.11.3, "What You May Need to Know About Bind Variable Options."

30.2.1 How to Create an LOV

You can create the LOV using either the `af:inputListOfValues` or the `af:inputComboboxListOfValues` component. You can add auto-suggest behavior to the component to display a list of possible matches from user input.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create an LOV. For more information, see Section 30.2, "Creating List of Values (LOV) Components."

You may also find it helpful to understand functionality that can be used with LOVs. For more information, see Section 30.1.2, "Additional Functionality for Selection Lists and Shuttles."

You will need to complete this task:

Define the attribute to be an LOV in the view object by following the procedure described in Section 5.12, "Working with List of Values (LOV) in View Object Attributes."

To create an LOV:

1. From the Data Controls panel, drag and drop the attribute onto the JSF page and choose **Create > List of Values > ADF LOV Input** or **Create > List of Values > ADF LOV Choice List**.
2. In the Operations page of the Component Palette, from the ADF Faces panel, drag and drop **Auto Suggest Behavior** as a child to the LOV component.
3. In the Property Inspector, for each of the auto-suggest attributes, enter the:
   - EL expression that resolves to the `suggestItems` method.
   - EL expression that resolves to the `smartList` method.
   - Number of items to be displayed in the auto-suggest list. Enter -1 to display the complete list.

Example 30–4 shows an `inputListOfValues` component with the auto-suggest behavior and smart list features.

Example 30–4 LOV Component with Auto-Suggest Behavior

```
<af:inputListOfValues id="ProductsId"
          popupTitle="Search and Select: #{bindings.Products.hints.label}"
          value="#{bindings.Products.inputValue}"
          label="#{bindings.Products.hints.label}"
          model="#{bindings.Products.listOfValuesModel}"
          required="#{bindings.Products.hints.mandatory}"
          columns="#{bindings.Products.hints.displayWidth}"
          shortDesc="#{bindings.Products.hints.tooltip}">
    <f:validator binding="#{bindings.Products.validator}"/>
    <af:autoSuggestBehavior suggestItems="#{bindings.Products.suggestedItems}"
          smartList="#{bindings.Products.smartList}"
          maxSuggestedItems="5"/>
</af:inputListOfValues>
```

4. If you added the auto-suggest behavior, you must set the component's `autoSubmit` property to `true`.

30.2.2 What Happens When You Create an LOV

When you drag and drop an attribute from the Data Controls panel, JDeveloper does many things for you. For a full description of what happens and what is created when you use the Data Controls panel, see Section 26.2.2, "What Happens When You Create a Text Field."

When you drag and drop an attribute defined as an LOV from the Data Controls panel onto a JSF page as an `inputListOfValues` or `inputComboboxListOfValues` component, JDeveloper adds code to the page similar to that shown in Example 30–5. The `language` attribute in the `CountryCodes` view object has been defined as an LOV and its default component is set to be an `inputComboboxListOfValues`. The component gets its properties from the control hints defined declaratively in the view object. If you want to include the auto-suggest behavior, you must manually add that tag from the Component Palette.

Example 30–5 inputComboboxListOfValues Component Code in a JSF Page

```
<af:inputComboboxListOfValues id="languageId"
            popupTitle="Search and Select: #{bindings.Language.hints.label}"
            value="#{bindings.Language.inputValue}"
            label="#{bindings.Language.hints.label}"
            model="#{bindings.Language.listOfValuesModel}"
            required="#{bindings.Language.hints.mandatory}"
            columns="#{bindings.Language.hints.displayWidth}"
            shortDesc="#{bindings.Language.hints.tooltip}"
            readOnly="false">
       <f:validator binding="#{bindings.Language.validator}"/>
</af:inputComboboxListOfValues>
```

In the page definition file, JDeveloper adds code as shown in Example 30–6. The bindings section of the page definition specifies that the LOV is for the `language` attribute and the name of the LOV is `LOV_language`. JDeveloper adds the definitions for the iterator binding objects into the `executables` element, and the list of values binding object into the `bindings` element.

Example 30–6 inputComboboxListOfValues Component Code in Page Definition

```
<executables>
    <variableIterator id="variables"/>
    <iterator Binds="CountryCodesView1" RangeSize="25"
              DataControl="AppModuleDataControl"
              id="CountryCodesView1Iterator"/>
</executables>
<bindings>
    <listOfValues StaticList="false" IterBinding="CountryCodesView1Iterator"
                  Uses="LOV_Language" id="Language"/>
</bindings>
```

For more information about the page definition file and ADF data binding expressions, see Section 13.7, "Working with Page Definition Files,"and Section 13.8, "Creating ADF Data Binding EL Expressions."

30.2.3 What You May Need to Know About List Validators and LOV

If you are defining an attribute in the view object to be a list of values attribute, you can define an equivalent list validation rule in the underlying entity object as long as the list validation rule is based on the same list. You should not define other types of validation rules in the entity object for an attribute already defined as a list of values in the view object. Doing so may result in unexpected behavior and validation errors. For more information about defining validation rules, see Section 7.3, "Adding Validation Rules to Entity Objects and Attributes," and Section 7.4, "Using the Built-in Declarative Validation Rules."

30.3 Creating a Selection List

ADF Faces Core includes components for selecting a single value and multiple values from a list. For example, `selectOneChoice` allows the user to select an item from a dropdown list, and `selectManyChoice` allow the user to select several items from a list of checkboxes. Selection lists are described in Table 30–1.

Table 30–1 ADF Faces Single and Multiple List Components

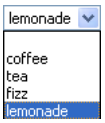
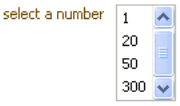
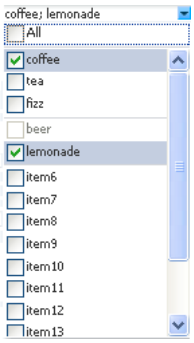
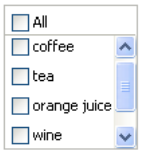
ADF Faces component	Description	Example
`SelectOneChoice`	Select a single value from a list of items.	
`SelectOneRadio`	Select a single value from a set of radio buttons.	
`SelectOneListbox`	Select a single value from a scrollable list of items.	

Table 30–1 (Cont.) ADF Faces Single and Multiple List Components

ADF Faces component	Description	Example
`SelectManyChoice`	Select multiple values from a scrollable list of checkboxes. Each selection displays at the top of the list.	
`SelectManyCheckbox`	Select multiple values from a group of checkboxes.	
`SelectManyListbox`	Select multiple values from a scrollable list of checkboxes,	

You can create selection lists using the `SelectOneChoice` ADF Faces component. The steps are similar for creating other single-value selection lists, such as `SelectOneRadio` and `SelectOneListbox`.

A databound selection list displays values from a data control collection or a static list and updates an attribute in another collection or a method parameter based on the user's selection. When adding a binding to a list, you use an attribute from the data control that will be populated by the selected value in the list.

> **Note:** By default, ADF Model list binding passes the index to the component. If you set `valuePassThru=true`, then you should set the list binding entry in the corresponding page definition file `Mode` attribute to `Object`. When `Mode=Object`, the list binding will pass an object instead of an index to the component. For more information, see Section 30.3.5, "What Happens When You Create a Fixed Selection List."

To create a selection list, you choose a base data source and a list data source in the Edit List Binding dialog:

- Base data source: Select the data collection that you want to bind to your control and that contains the attributes to be updated from user selections.
- List data source: Select the data collection that contains the attributes to display.

The data collection is based on the view object. For more information about creating a view object, see Section 5.2.1, "How to Create an Entity-Based View Object."

You can create three types of selection lists in the Edit List Binding dialog:

- Model-driven list: List selections are based on a list of values bound to a data collection. This type of selection list offers significant advantages over the other two, as described in Section 30.3.1, "How to Create a Model-Driven List."
- Static list: List selections are based on a fixed list that you create a manually by entering values one at a time into the editor. For more information, see Section 30.3.2, "How to Create a Selection List Containing Fixed Values."
- Dynamic list: List selections are generated dynamically based on one or more databound attribute values. For more information, see Section 30.3.3, "How to Create a Selection List Containing Dynamically Generated Values."

30.3.1 How to Create a Model-Driven List

A model-driven list is based on a list of values that is bound to a view data object. Lists of Values are typically used in forms to enable an end user to select an attribute value from a dropdown list instead of having to enter it manually. When the user submits the form with the selected values, ADF data bindings in the ADF Model layer update the value on the view object attributes corresponding to the databound fields.

> **Note:** One way to create a model-driven list is to drag a collection from the Data Controls panel onto a JSF page, choose one of the ADF Forms in the popup menu, and accept the defaults. The advantage is that if there are LOVs defined on the underlying view object attributes, all the LOVs on the entire form will be configured automatically. For more information, see Section 5.12.1, "How to Define a Single LOV-Enabled View Object Attribute."

You can also use the list of values as the basis for creating a selection list. The advantages of creating a model-driven list based on a list of values are:

- Reuse: The list of values is bound to a view data collection. Any selection list that you create based on the data collection can use the same list of values. Because you define the LOV on the individual attributes of view objects in a data model project with ADF Business Components, you can customize the LOV usage for an attribute once and expect to see the changes anywhere that the business component is used in the user interface.
- Translation: Values in the list of values can be included in resource bundles used for translation.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create an selection list component. For more information, see Section 30.3, "Creating a Selection List."

You may also find it helpful to understand functionality that can be used with selection lists. For more information, see Section 30.1.2, "Additional Functionality for Selection Lists and Shuttles."

You will need to complete these tasks:

1. Create a view object.
2. Create a view accessor on the object.

3. Create a list of values that is bound to an attribute on the base data source for the selection list. For example, you can create a list of values bound to the `CountryId` attribute of the `Addresses` view data object.

   For more information, see Section 5.12.1, "How to Define a Single LOV-Enabled View Object Attribute."

To create a model-driven selection list:

1. From the Data Controls panel, drag and drop the attribute onto the JSF page and choose **Create > Single Selections > ADF Select One Choice**.

   The Edit List Binding dialog displays. The view object collection containing the attribute you dropped on the JSF page is selected by default in the **Base Data Source** list.

   To select a different view data collection, click the **Add** icon next to the list.

2. Click the **Model Driven List** radio button.

3. In the **Base Data Source Attribute** list, select an attribute on which you based a list of values, for example, `CountryId`.

   The list contains all the attributes for the view data collection selected in the **Base Data Source** list.

4. If a list of values was created for the attribute you selected, it will be listed in the **Server List Binding Name** list.

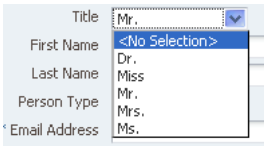
   For example, you could select `LOV_CountryId` in the **Server List Binding Name** list because a list of values was created for the `CountryId` attribute.

5. Click **OK**.

30.3.2 How to Create a Selection List Containing Fixed Values

You can create a selection list containing selections that you code yourself, rather than retrieving the values from another data source. See Section 30.3.3, "How to Create a Selection List Containing Dynamically Generated Values," for information about populating selection lists with values that are dynamically generated from another data source.

Figure 30–6 Selection List Bound to a Fixed List of Values



Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a selection list component. For more information, see Section 30.3, "Creating a Selection List."

You may also find it helpful to understand functionality that can be used with selection lists. For more information, see Section 30.1.2, "Additional Functionality for Selection Lists and Shuttles."

You will need to complete this task:

Prepare a list of values that you will enter into the component as a fixed list.

To create a list bound to a fixed list of values:

1. From the Data Controls panel, drag and drop the attribute onto the JSF page and choose **Create > Single Selections > ADF Select One Choice**.

   The Edit List Binding dialog displays. The view object collection containing the attribute you dropped on the JSF page is selected by default in the **Base Data Source** list.

   To select a different view data collection, click the **Add** icon next to the list.

2. Click the **Fixed List** radio button.

   The **Fixed List** option lets end users choose a value from a static list that you define.

3. In the **Base Data Source Attribute** list, choose an attribute.

   The **Base Data Source Attribute** list contains all of the attributes in the view data collection you selected in the **Base Data Source** list. For example, if you selected `CountryCodes` as the base data source, you can choose `CountryName` in the list.

4. In the **Set of Values** box, enter each value you want to appear in the list. Press the Enter key to set a value before typing the next value. For example, you could add the country codes `India`, `Japan`, and `Russia`.

   The order in which you enter the values is the order in which the list items are displayed in the `SelectOneRadio` control at runtime.

   The `SelectOneRadio` component supports a `null` value. If the user has not selected an item, the label of the item is shown as blank, and the value of the component defaults to an empty string. Instead of using blank or an empty string, you can specify a string to represent the `null` value. By default, the new string appears at the top of the list.

5. Click **OK**.

30.3.3 How to Create a Selection List Containing Dynamically Generated Values

You can populate a selection list component with values dynamically at runtime.

> **Tip:** Another option is to create a static view object or a database view object within a shared application module. Then use a model-driven LOV to create the list. This provides caching and translatability.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a selection list component. For more information, see Section 30.3, "Creating a Selection List."

You may also find it helpful to understand functionality that can be used with selection lists. For more information, see Section 30.1.2, "Additional Functionality for Selection Lists and Shuttles."

You will need to complete these tasks:

1. Define a data source for the list data source that provides the dynamic list of values.
2. Define a data source for the base data source that is to be updated based on the user's selection.

To create a selection list bound containing dynamically generated values:

1. From the Data Controls panel, drag and drop the attribute onto the JSF page and choose **Create > Single Selections > ADF Select One Choice**.

   The Edit List Binding dialog displays. The view object collection containing the attribute you dropped on the JSF page is selected by default in the **Base Data Source** list.

   To select a different view data collection, click the **Add** icon next to the list.

2. Click the **Dynamic List** radio button.

   The **Dynamic List** option lets you specify one or more base data source attributes that will be updated from another set of bound values.

3. Click the **Add** button next to **List Data Source**.

4. In the Add Data Source dialog, select the view data collection that will populate the values in the selection list.

   In the Fusion Order Demo application, for example, you could select `ProductQuantities`.

   Note: The list and base collections do not have to form a master-detail relationship, but the attribute in the list collection must have the same type as the base collection attributes.

5. Accept the default iterator name and click **OK**.

   The **Data Mapping** section of the Edit List Binding dialog updates with a default data value and list attribute. The **Data Value** control contains the attribute on the data collection that is updated when the user selects an item in the selection list. The **List Attribute** control contains the attribute that populates the values in the selection list.

6. You can accept the default mapping or select different attributes items from the **Data Value** and **List Attribute** lists to update the mapping.

   To add a second mapping, click **Add**.

7. Click **OK**.

30.3.4 What Happens When You Create a Model-Driven Selection List

When you drag and drop an attribute from the Data Controls panel, JDeveloper does many things for you. For a full description of what happens and what is created when you use the Data Controls panel, see Section 26.2.2, "What Happens When You Create a Text Field."

Example 30–7 shows the page source code after you add a model-driven `SelectOneChoice` component to it.

Example 30–7 Model-Driven SelectOneChoice List in JSF Page Source Code

```
<af:selectOneChoice value="#{bindings.CountryId1.inputValue}"
                            label="#{bindings.CountryId1.label}">
          <f:selectItems value="#{bindings.CountryId1.items}"/>
</af:selectOneChoice>
```

The `f:selectItems` tag, which provides the list of items for selection, is bound to the `items` property on the `CountryId1` list binding object in the binding container.

For more information about ADF data binding expressions, see Section 13.6, "Configuring the ADF Binding Filter."

In the page definition file, JDeveloper adds the list binding object definitions in the bindings element, as shown in Example 30–8.

Example 30–8 List Binding Object for the Model-Driven List in the Page Definition File

```
<bindings>
   <list IterBinding="AddressesView1Iterator" id="CountryId"
         Uses="LOV_AddressId" StaticList="false" ListOperMode="0">
     <AttrNames>
       <Item Value="AddressId"/>
     </AttrNames>
   </list>
   <list IterBinding="AddressesView1Iterator" id="CountryId1"
         Uses="LOV_CountryId" StaticList="false" ListOperMode="0">
     <AttrNames>
       <Item Value="CountryId"/>
     </AttrNames>
   </list>
 </bindings>
```

In the `list` element, the `id` attribute specifies the name of the list binding object. The `IterBinding` attribute references the variable iterator, whose current row is a row of attributes representing each variable in the binding container. The variable iterator exposes the variable values to the bindings in the same way as do other collections of data. The `AttrNames` element specifies the attribute value returned by the iterator.

For more information about the page definition file and ADF data binding expressions, see Section 13.7, "Working with Page Definition Files,"and Section 13.8, "Creating ADF Data Binding EL Expressions."

30.3.5 What Happens When You Create a Fixed Selection List

When you add a fixed selection list, JDeveloper adds source code to the JSF page and list and iterator binding objects to the page definition file.

Example 30–9 shows the page source code after you add a fixed `SelectOneChoice` component to it.

Example 30–9 Fixed SelectOneChoice List in JSF Page Source Code

```
<af:selectOneChoice value="#{bindings.CountryIdStatic.inputValue}"
                            label="#{bindings.CountryIdStatic.label}">
          <f:selectItems value="#{bindings.CountryIdStatic.items}"/>
        </af:selectOneChoice>
```

The `f:selectItems` tag, which provides the list of items for selection, is bound to the `items` property on the `CountryId` list binding object in the binding container. For more information about ADF data binding expressions, see Section 13.8, "Creating ADF Data Binding EL Expressions."

In the page definition file, JDeveloper adds the definitions for the iterator binding objects into the `executables` element, and the list binding object into the `bindings` element, as shown in Example 30–10.

Example 30–10 List Binding Object for the Fixed Selection List in the Page Definition File

```
<executables>
    <iterator Binds="Addresses1" RangeSize="10"
              DataControl="StoreFrontModuleDataControl"
              id="Addresses1Iterator"/>
  </executables>
  <bindings>
    <list IterBinding="Addresses1Iterator" id="CountryIdStatic" ListOperMode="0"
          StaticList="true">
      <AttrNames>
        <Item Value="CountryIdStatic"/>
      </AttrNames>
      <ValueList>
        <Item Value="India"/>
        <Item Value="Japan"/>
        <Item Value="Russia"/>
      </ValueList>
    </list>
  </bindings>
```

For complete information about page definition files, see Section 13.7, "Working with Page Definition Files."

30.3.6 What You May Need to Know About Values in a Selection List

Once you have created a list binding, you may want to access a value in the list. You can set the component's `valuePassThru` attribute to `true` either in the visual editor or in the Property Inspector. Example 30–11 shows the code for a `selectOneChoice` component.

Example 30–11 valuePassThru Attribute Set to true in selectOneChoice

```
 <af:selectOneChoice value="#{bindings.Language.inputValue}"
                     label="#{bindings.Language.label}"
                     required="#{bindings.Language.hints.mandatory}"
                     shortDesc="#{bindings.Language.hints.tooltip}"
                     valueChangeListener="#{myBean.valueChanged}"
                     id="soc1" valuePassThru="true">
      <f:selectItems value="#{bindings.Language.items}" id="si1"/>
</af:selectOneChoice>
```

Only when the `valuePassThru` attribute is set to `true` and the `Mode` attribute in the corresponding page definition file is set to `Object` can the value passed to the `valueChangeListener` be of type `Object` (which is the actual value). By default, `Mode` is set to `Index`. Example 30–12 shows the `Mode` attribute in the page definition file.

Example 30–12 Mode Attribute Set to Object in the Page Definition File

```
 <list IterBinding="AvailableLanguagesView1Iterator" id="Language"
          DTSupportsMRU="true" StaticList="false"
          ListIter="CountryCodesView1Iterator"
          Mode="Object"/>
```

30.3.7 What Happens When You Create a Dynamic Selection List

When you add a dynamic selection list to a page, JDeveloper adds source code to the JSF page and list and iterator binding objects to the page definition file.

Example 30–13 shows the page source code after you add a dynamic `SelectOneChoice` component to it.

Example 30–13 Dynamic SelectOneChoice List in JSF Page Source Code

```
<af:selectOneChoice value="#{bindings.Quantity.inputValue}"
                          label="#{bindings.Quantity.label}">
         <f:selectItems value="#{bindings.Quantity.items}"/>
       </af:selectOneChoice>
```

The `f:selectItems` tag, which provides the list of items for selection, is bound to the `items` property on the `Quantity` list binding object in the binding container. For more information about ADF data binding expressions, see Section 13.8, "Creating ADF Data Binding EL Expressions."

In the page definition file, JDeveloper adds the definitions for the iterator binding objects into the `executables` element, and the list binding object into the `bindings` element, as shown in Figure 30–14.

Example 30–14 List Binding Object for the Dynamic Selection List in the Page Definition File

```
<executables>
    <iterator Binds="OrderItems" RangeSize="-1"
              DataControl="StoreFrontModuleDataControl1"
              id="OrderItemsIterator"/>
    <iterator Binds="ProductQuantities" RangeSize="10"
              DataControl="StoreFrontModuleDataControl1"
              id="ProductQuantitiesIterator"/>
  </executables>
  <bindings>
    <list IterBinding="AddressesView1Iterator" id="CountryId"
          Uses="LOV_AddressId" StaticList="false" ListOperMode="0">
      <AttrNames>
        <Item Value="AddressId"/>
      </AttrNames>
    </list>
    <list IterBinding="ProductQuantities1Iterator" id="Quantity"
          StaticList="false" ListOperMode="0" ListIter="OrderItems1Iterator">
      <AttrNames>
        <Item Value="Quantity"/>
      </AttrNames>
      <ListAttrNames>
        <Item Value="Quantity"/>
      </ListAttrNames>
      <ListDisplayAttrNames>
        <Item Value="OrderId"/>
      </ListDisplayAttrNames>
    </list>
  </bindings>
```

By default, JDeveloper sets the `RangeSize` attribute on the `iterator` element for the `OrderItems` iterator binding to a value of `-1` thus allowing the iterator to furnish the full list of valid products for selection. In the `list` element, the `id` attribute specifies

the name of the list binding object. The `IterBinding` attribute references the iterator that iterates over the `ProductQuantities` collection. The `ListIter` attribute references the iterator that iterates over the `ProductList` collection. The `AttrNames` element specifies the base data source attributes returned by the base iterator. The `ListAttrNames` element defines the list data source attributes that are mapped to the base data source attributes. The `ListDisplayAttrNames` element specifies the list data source attribute that populates the values users see in the list at runtime.

For complete information about page definition files, see Section 13.7, "Working with Page Definition Files."

30.4 Creating a List with Navigation List Binding

Navigation list binding lets users navigate through the objects in a collection. As the user changes the current object selection using the navigation list component, any other component that is also bound to the same collection through its attributes will display from the newly selected object. In addition, if the collection whose current row you change is the master view object instance in a data model master-detail relationship, the row set in the detail view object instance is automatically updated to show the appropriate data for the new current master row.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a navigation list. For more information, see Section 30.4, "Creating a List with Navigation List Binding."

You may also find it helpful to understand functionality that can be used with navigation lists. For more information, see Section 30.1.2, "Additional Functionality for Selection Lists and Shuttles."

To create a list that uses navigation list binding:

1. From the Data Controls panel, drag and drop a collection to the page and choose **Create > Navigation > ADF Navigation Lists**.
2. In the Edit List Binding dialog, from the **Base Data Source** dropdown list, select the collection whose members will be used to create the list.

   This should be the collection you dragged from the Data Controls panel. If the collection does not appear in the dropdown menu, click the **Add** button to select the collection you want.
3. From the **Display Attribute** dropdown list, select the attribute that will display in the list. You can choose the selection that includes all the attributes, and you can choose **Select Multiple** to launch a selection dialog.

   In the Select Multiple Display Attributes dialog, shuttle the attributes you want to display from the **Available Attributes** list to the **Attributes to Display** list. Click **OK** to close the dialog.
4. Click **OK**.

30.5 Creating a Databound Shuttle

The `selectManyShuttle` and `selectOrderShuttle` components render two list boxes and buttons that allow the user to select multiple items from the leading (or available) list box and to move or shuttle those items over to the trailing (or selected) list box, and vice versa. You can also double-click an item to move it to the other list

box. Figure 30–10 shows an example of a rendered `selectManyShuttle` component. You can specify any text you want for the headers that display above the list boxes.

Figure 30–7 SelectManyShuttle Component

The only difference between `selectManyShuttle` and `selectOrderShuttle` is that in the `selectOrderShuttle` component, the user can reorder the items in the trailing list box by using the up and down arrow buttons on the side.

The Fusion Order Demo application uses a `selectManyShuttle` component to select customer interest categories from a **Categories of Interest** list box to an **I am interested in** list box. The leading list box on the left displays all the categories. The trailing list box on the right displays the categories the customer has selected.

Like other ADF Faces selection list components, the `selectManyShuttle` component can use the `f:selectItems` tag to provide the list of items available for display and selection in the leading list.

Before you can bind the `f:selectItems` tag, create a generic class that can be used by any page that requires a shuttle. In the class, declare and include getter and setter methods for the properties that describe the view object instance names that should be used for the list of all available choices (leading list or available product categories) and the list of selected choices (trailing list or assigned product categories). Example 30–15 shows the `CustRegBasicInformationBean` class that is created to manage the population and selection state of the shuttle component on the `basicInformation.jsff` page.

Example 30–15 CustRegBasicInformationBean Class

```
package oracle.fodemo.storefront.account.view.managed;
import java.io.Serializable;
import java.util.List;
import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;
import oracle.binding.OperationBinding;
import oracle.binding.BindingContainer;
import oracle.fodemo.storefront.adf.util.ADFUtils;
import javax.faces.event.ValueChangeEvent;
import oracle.fodemo.storefront.jsf.util.JSFUtils;
public class CustRegBasicInformationBean implements Serializable {
    String allItemsIteratorName;
    String allItemsValueAttrName;
    String allItemsDisplayAttrName;
    String allItemsDescriptionAttrName;
    String selectedValuesIteratorName;
    String selectedValuesValueAttrName;
```

```
    List selectedValues;
    List allItems;
    private boolean refreshSelectedList = false;
    public CustRegBasicInformationBean() {
    }

    public void setAllItemsIteratorName(String allItemsIteratorName) {
        this.allItemsIteratorName = allItemsIteratorName;
    }
    public String getAllItemsIteratorName() {
        return allItemsIteratorName;
    }

// other getter and setter methods are omitted
    public void setSelectedValues(List selectedValues) {
        this.selectedValues = selectedValues;
    }
    public void refreshSelectedList(ValueChangeEvent event) {
        refreshSelectedList = true;
    }
    public List getSelectedValues() {
        if (selectedValues == null || refreshSelectedList) {
            selectedValues =
                    ADFUtils.attributeListForIterator(selectedValuesIteratorName,
                                                      selectedValuesValueAttrName);
        }
        return selectedValues;
    }
    public void setAllItems(List allItems) {
        this.allItems = allItems;
    }

    public List getAllItems() {
        if (allItems == null) {
            allItems =
                    ADFUtils.selectItemsForIterator(allItemsIteratorName,
                    allItemsValueAttrName,
                    allItemsDisplayAttrName,
                    allItemsDescriptionAttrName);
        }
        return allItems;
    }
}
```

The `getAllItems()` method populates the shuttle's leading list. The `getSelectedValues()` method also returns a `List`, but the list defines the items in the shuttle's trailing list. The values returned by `getSelectedValues()` are a subset of the values returned by `getAllItems()`. Values that are in `getAllItems()` but not in `getSelectedValues()` will be ignored (and will generate an error in the server log). Note that the CustRegBasicInformationBean class calls several utility methods in the `ADFUtils` class. Also note that this class uses values for several properties of the base bean. Example 30–16 shows the managed bean and managed properties configured in `customer-registration-task-flow.xml` for working with the shuttle component.

Example 30–16 Managed Bean for the Shuttle Component in the customer-registration-task-flow File

```
<managed-bean>
      <managed-bean-name>
```

```
        custRegBasicInformationBean</managed-bean-name>
      <managed-bean-class>
        oracle.fodemo.storefront.account.view.managed.CustRegBasicInformationBean
      </managed-bean-class>
      <managed-bean-scope>view</managed-bean-scope>
      <managed-property>
        <property-name>allItemsIteratorName</property-name>
        <value>AvailableCategoriesShuttleListIterator</value>
      </managed-property>
      <managed-property>
        <property-name>allItemsValueAttrName</property-name>
        <value>CategoryId</value>
      </managed-property>
      <managed-property>
        <property-name>allItemsDisplayAttrName</property-name>
        <value>CategoryName</value>
      </managed-property>
      <managed-property>
        <property-name>allItemsDescriptionAttrName</property-name>
        <value>CategoryDescription</value>
      </managed-property>
      <managed-property>
        <property-name>selectedValuesIteratorName</property-name>
        <value>SelectedCategoriesShuttleListIterator</value>
      </managed-property>
      <managed-property>
        <property-name>selectedValuesValueAttrName</property-name>
        <value>CategoryId</value>
      </managed-property>
</managed-bean>
```

The `basicInformation.jsff` page uses the following iterator objects:

- `CustomerRegistrationIterator`: Iterates over the `CustomerRegistration` collection, which provides the context for the form above the shuttle.
- `SelectedCategoriesShuttleListIterator`: Iterates over the `SelectedCategoriesShuttleList` collection, which provides the list of categories assigned to the current customer.
- `AvailableCategoriesShuttleListIterator`: Iterates over the `AvailableCategoriesShuttleList` collection, which provides the list of product categories.

All the bindings of the `basicInformation.jsff` page are defined in the file `account_basicInformationPageDef.xml`. Example 30–17 shows part of the page definition file for the `basicInformation.jsff` page.

Example 30–17 Page Definition File for the basicInformation Page

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="11.1.1.48.68" id="account_basicInformationPageDef"
                Package="oracle.fodemo.storefront.pageDefs">
  <parameters/>
  <executables>
    <page path="oracle.fodemo.storefront.pageDefs.
          templates_StoreFrontTemplatePageDef"
          id="pageTemplateBinding"/>
```

```
    <iterator Binds="CustomerRegistration" RangeSize="25"
              DataControl="StoreServiceAMDataControl"
              id="CustomerRegistrationIterator" Refresh="ifNeeded"/>
    <iterator id="SelectedCategoriesShuttleListIterator"
              Binds="SelectedCategoriesShuttleList" RangeSize="-1"
              DataControl="StoreServiceAMDataControl"/>
    <iterator id="AvailableCategoriesShuttleListIterator"
              Binds="AvailableCategoriesShuttleList" RangeSize="-1"
              DataControl="StoreServiceAMDataControl"/>
    <invokeAction Binds="setHelpId" id="invokeSetHelpId" Refresh="ifNeeded"/>
  </executables>
```

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a shuttle. For more information, see Section 30.5, "Creating a Databound Shuttle."

You may also find it helpful to understand functionality that can be used with selection lists. For more information, see Section 30.1.2, "Additional Functionality for Selection Lists and Shuttles."

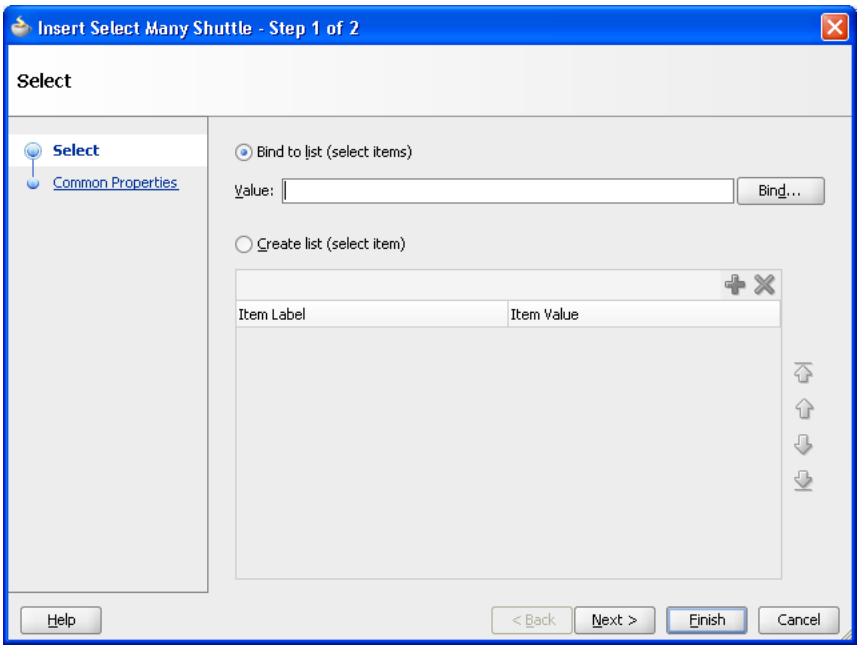
You will need to complete these tasks:

1. Create the relevant iterator bindings in the page definition file. Use Example 30–17 as a guide.
2. Create a class similar to the CustRegBasicInformationBean class. Use Example 30–15 as a guide.
3. Configure the required managed bean and managed properties in the task flow definition or, in the case of an unbounded task flow, the `adfc-config.xml` file. Use Example 30–16 as a guide.

To create a shuttle component:

1. In the Common Components page of the Component Palette, from the Common panel, drag and drop **SelectManyShuttle** onto the page. JDeveloper displays the Insert SelectManyShuttle wizard, as illustrated in Figure 30–8.

Figure 30–8 Insert SelectManyShuttle Wizard



2. Select **Bind to list (select items)** and click **Bind** to open the Expression Builder.
3. In the Expression Builder, expand **ADF Managed Beans > pageFlowScope > CustRegBasicInformationBean > allItems** to build the expression `#{pageFlowScope.custRegBasicInformationBean.allItems}` and click **Next**.

   This binds the `f:selectItems` tag to the `getAllItems()` method that populates the shuttle's leading list.
4. In the Common Properties page, click **Bind** next to the **Value** field to open the Expression Builder again.
5. In the Expression Builder, expand **ADF Managed Beans > pageFlowScope > CustRegBasicInformationBean > selectedValues** to build the expression `#{pageFlowScope.custRegBasicInformationBean.selectedValues}` and click **Finish**.

   This binds the `value` attribute of the `selectManyShuttle` component to the `getSelectedValues()` method that populates the shuttle's trailing list.

Example 30–18 shows the code for the `selectManyShuttle` component after you complete the Insert SelectManyShuttle dialog.

Example 30–18 SelectManyShuttle Component in basicInformation.jsff File

```
<af:selectManyShuttle
       value="#{pageFlowScope.custRegBasicInformationBean.selectedValues}"
              leadingHeader="#{res['basicinfo.leadingHeader']}"
              trailingHeader="#{res['basicinfo.trailingHeader']}"
              leadingDescShown="true" size="8"
              trailingDescShown="true"
              inlineStyle="background-color:transparent;"
              id="sms1">
       <f:selectItems
              value="#{pageFlowScope.custRegBasicInformationBean.allItems}"/>
</af:selectManyShuttle>
```

31

Creating ADF Databound Search Forms

This chapter describes how to create search forms to perform complex searches on multiple attributes and search forms to search on a single attribute. It also includes information on using named bind variables and using filtered table searches.

This chapter includes the following sections:

- Section 31.1, "About Creating Search Forms"
- Section 31.2, "Creating Query Search Forms"
- Section 31.3, "Setting Up Search Form Properties"
- Section 31.4, "Creating Quick Query Search Forms"
- Section 31.5, "Creating Standalone Filtered Search Tables from Named View Criteria"

31.1 About Creating Search Forms

You can create search forms that allow users to enter search criteria into input fields for known attributes of an object. The search criteria can be entered via input text fields or selected from a list of values in a popup list picker or dropdown list box. The entered criteria is constructed into a query to be executed. *Named bind variables* can be used to supply attribute values during runtime for the query. The results of the query can be displayed as a table, a form, or another UI component.

Oracle ADF supports two types of search forms: query and quick query. The *query search form* is a full-featured search form. The *quick query search form* is a simplified form with only one search criteria. Each of these search forms can be combined with a filtered table to display the results, thereby enabling additional search capabilities. You can also create a standalone filtered table to perform searches without the query or quick query search panel.

Search forms are based either on view criteria defined in view objects or on implicit view criteria defined by JDeveloper. Search forms are region-based components that are reusable and personalizable. They encapsulate and automate many of the actions and iterator management operations required to perform a query. You can create several search forms on the same page without any need to change or create new iterators.

A *filtered table* is a table that has additional Query-by-Example (QBE) search criteria fields above each searchable column. When the filtering option of a table is enabled, you can enter QBE-style search criteria for each column to filter the query results. For more information about tables, see Chapter 27, "Creating ADF Databound Tables."

For more information about individual query and table components, see the "Using Query Components" and the "Using Tables and Trees" chapters of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

31.1.1 Implicit and Named View Criteria

When you create data controls, all data collections will automatically include a Named Criteria node with an All Queriable Attributes criteria. This is the default view criteria that includes all the searchable attributes or columns of the data collection. You cannot edit or modify this view criteria. These implicit view criteria can be used in the same way as declaratively created (or named) view criteria during the creation of query and quick query search forms. For more information about creating named view criteria, see Section 5.11, "Working with Named View Criteria."

When you add additional named view criteria for that view object or collection, the new view criteria will be added to the Named Criteria node.

Note: Query search forms have certain restrictions for working with expressions that have nested view criteria and may not work with all types of nested expressions. For more information about the nested expressions supported by search forms, see Section 5.11.4, "What You May Need to Know About Nested Expressions."

In the Data Controls panel, a data collection's Named Criteria node will always include the implicit view criteria, regardless of whether any named view criteria were defined. The implicit view criteria is always available for every data collection.

31.1.2 List of Values (LOV) Input Fields

List of values (LOV) components are input components that allow the user to enter values by picking from a list that is generated by a query. ADF Faces provides the `af:inputListOfValues` and `af:inputComboboxListOfValues` components. If you are using dependent LOVs as part of your search form, you must use them with the `af:query` component. For more information about LOV components, see Section 30.2, "Creating List of Values (LOV) Components."

If an attribute is defined as an LOV, you can set the **Support Multiple Value Selection** control hint in its view criteria to enable users to make multiple selections in the search criteria field. If multiple selection is enabled on an LOV attribute, and the `Equal to` or `Not equal to` operator is chosen, a `selectManyChoice` component will render in the query panel. The user can select multiple items as the search criteria.

When the LOV is in a query component, if the **Support Multiple Value Selection** hint is not set and the `Equal to` or `Not equal to` operator is chosen, the query component will render a search criteria component according to the **Default List Type** control hint for the corresponding attribute in the view object.

The quick query component does not support multiple selection. It will always render the component specified by the **Default List Type** control hint. Table 31–1 shows the control hint selection and the default list component.

Table 31–1 Query and Quick Query Search Criteria Field Input Components

Default List Type Control Hint	Component
Input Text with List of Values	`af:inputListOfValues`

Table 31–1 (Cont.) Query and Quick Query Search Criteria Field Input Components

Default List Type Control Hint	Component
Combo Box with List of Values	`af:inputComboboxListOfValues`
Choice List, Combo Box, List Box, Radio Group	`af:selectOneChoice`

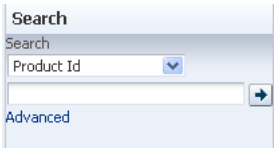
For more information about view criteria options, see Section 5.11.1, "How to Create Named View Criteria Declaratively," and Section 5.11.3, "What You May Need to Know About Bind Variable Options."

31.1.3 Search Form Use Cases and Examples

The search forms are based on the model-driven `af:query` and `af:quickQuery` components. Because these underlying components are model-driven, the search form will change automatically to reflect changes in the model. The view layer does not need to be changed. For example, if you define a list of values (LOV) on an attribute in the view object or entity object, the LOV will automatically show up in the search form as an LOV component. Or, if you modify a view criteria to include a new attribute for the `WHERE` clause, the search panel using this view criteria will automatically reflect that change by adding a search field for that attribute.

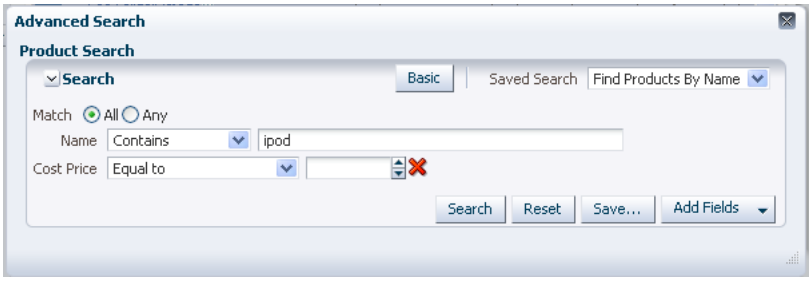
In the StoreFront module of the Fusion Order Demo application, a quick query is used to allow the user to perform a simple search among several attributes that can be used to find a product. For instance, user can select to search on product id and then entering a code, or search on product name and entering a name, as shown in Figure 31–1.

Figure 31–1 Quick Query



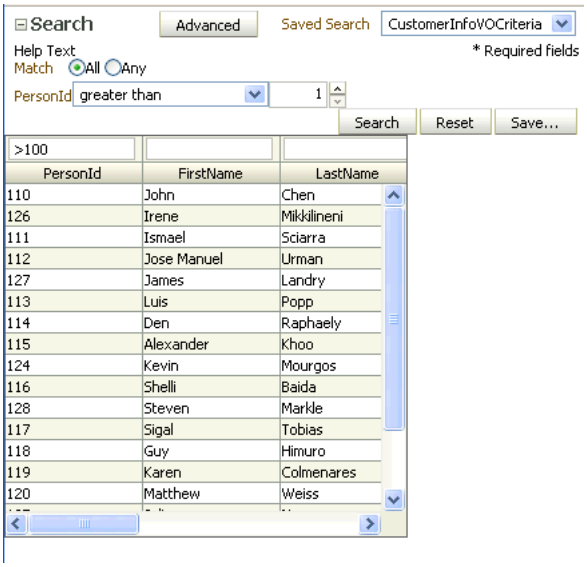
For more complex searches, such as including criteria for product name, price ranges, and availability, a query component is used. You can create the query component in several different modes to give the user different capabilities, as shown in Figure 31–2. The user can even add their own search criteria using the **Add Fields** button. The user can save the searches that were created to be used later.

Figure 31–2 Query Component



For simple searches, you can use the filtered table to provide Query-by-Example (QBE) searches, as shown in Figure 31–3.

Figure 31–3 Filtered Table Searches



31.1.4 Additional Functionality for Search Forms

You may find it helpful to understand other ADF features before you configure or use the ADF Model layer. Additionally, you may want to read about what you can do with your model layer configurations. Following are links to other functionality that may be of interest.

- Both the query and quick query components are based on the view criteria defined in a view object. You use the view criteria to set up the initial search fields and search criteria. You use control hints to further define the query component. For more information about creating named view criteria, see Section 5.11, "Working with Named View Criteria."
- Since filtered tables also provide search functions, you should consider them for simple searches. For more information about tables, see Chapter 27, "Creating ADF Databound Tables."
- List of value components use the query component in their search panel for creating the list of values selection list. For more information about LOV components, see Section 30.2, "Creating List of Values (LOV) Components."
- If your application is set up with a Metadata Storage (MDS), you can persist saved searches across transactions and personalize your searches. For more information about using MDS, see Chapter 39, "Customizing Applications with MDS."

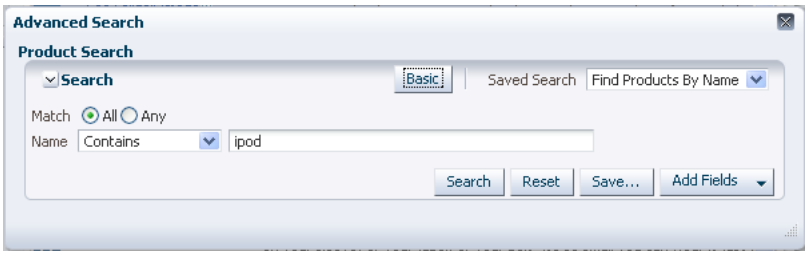
31.2 Creating Query Search Forms

The query search form is the standard form for complex transactional searches. You can build complex search forms with multiple search criteria fields each with a dropdown list of built-in operators. You can also add custom operators and customize the list. The query search form supports lists of values, `AND` and `OR` conjunctions, and saving searches for future use.

A query search form has a basic mode and an advanced mode. The user can toggle between the two modes using the basic/advanced button. At design time, you can declaratively specify form properties (such as setting the default state) to be either

basic or advanced. Figure 31–4 shows an advanced mode query search form with three search criteria.

Figure 31–4 Advanced Mode Query Search Form

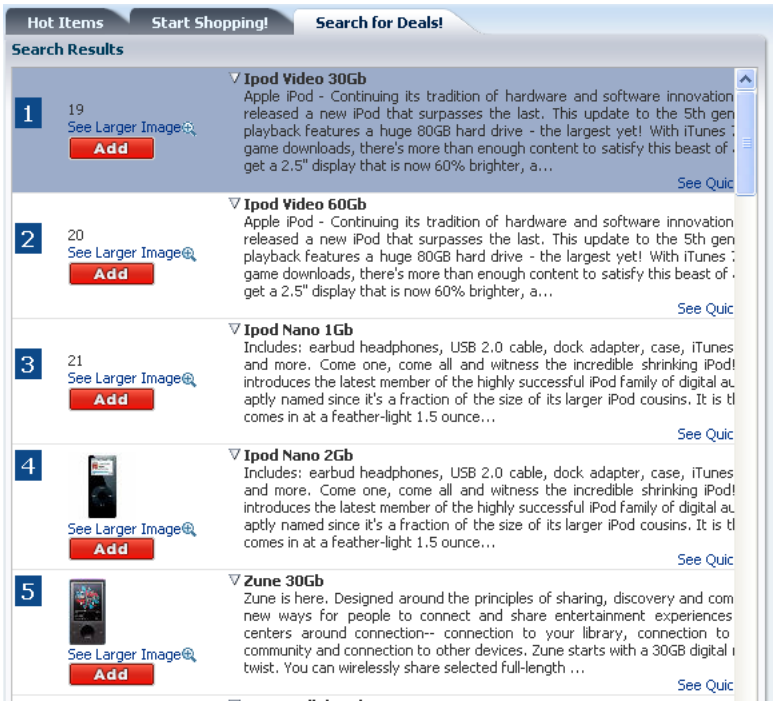


The advanced mode query form features are:

- Selecting search criteria operators from a dropdown list
- Adding custom operators and deleting standard operators
- Selecting `WHERE` clause conjunctions of either `AND` or `OR` (match all or match any)
- Dynamically adding and removing search criteria fields at runtime
- Saving searches for future use
- Personalizing saved searches

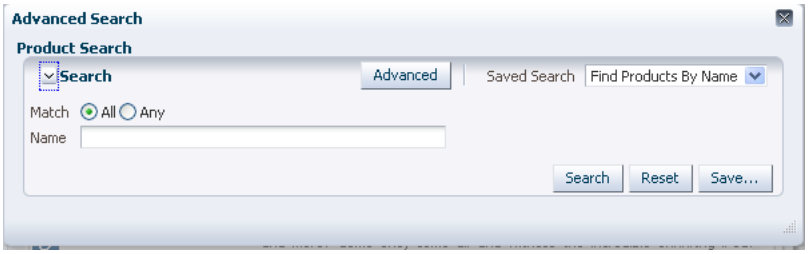
Typically, the query search form in either mode is used with an associated results table or tree table. For example, the query results for the search form in Figure 31–4 may be displayed in a table, as shown in Figure 31–5.

Figure 31–5 Results Table for a Query Search



The basic mode has all the features of the advanced mode except that it does not allow the user to dynamically add search criteria fields. Figure 31–6 shows a basic mode query search form with one search criteria field. Notice the lack of a dropdown list next to the **Save** button used to add search criteria fields in the advanced mode.

Figure 31–6 Basic Mode Query Form with One Search Criteria Field



In either mode, each search criteria field can be modified by selecting operators such as `Greater Than` and `Equal To` from a dropdown list, and the entire search panel can be modified by the **Match All/Any** radio buttons. Partial page rendering is also supported by the search forms in almost all situations. For example, if a `Between` operator is chosen, another input field will be displayed to allow the user to select the upper range.

A `Match All` selection implicitly uses `AND` conjunctions between the search criteria in the `WHERE` clause of the query. A `Match Any` selection implicitly uses `OR` conjunctions in the `WHERE` clause. Example 31–3 shows how a simplified `WHERE` clause may appear (the real `WHERE` in the view criteria is different) when `Match All` is selected for the search criteria shown in Figure 31–4.

Example 31–1 Simplified WHERE Clause Fragment When "Match All" Is Selected

```
WHERE (ProductId=4) AND (InStock > 2) AND (ProductName="Ipod")
```

Example 31–1 shows a simplified `WHERE` clause if **Match Any** is selected for the search criteria shown in Example 31–2.

Example 31–2 Simplified WHERE Clause Fragment When "Match Any" Is selected

```
WHERE (ProductId=4) OR (InStock > 2) OR (ProductName="Ipod")
```

If the view criteria for the query has mixed `AND` and `OR` conjunctions between the criteria items, then neither **Match All** nor **Match Any** will be selected when the component first renders. However, the user can select **Match All** or **Match Any** to override the conjunctions defined as the initial state in the view criteria.

Advanced mode query forms allow users to dynamically add search criteria fields to the query panel to perform more complicated queries. These user-created search criteria fields can be deleted, but the user cannot delete existing fields. Figure 31–7 shows how the **Add Fields** dropdown list is used to add the **CategoryId** criteria field to the search form.

Figure 31–7 Dynamically Adding Search Criteria Fields at Runtime

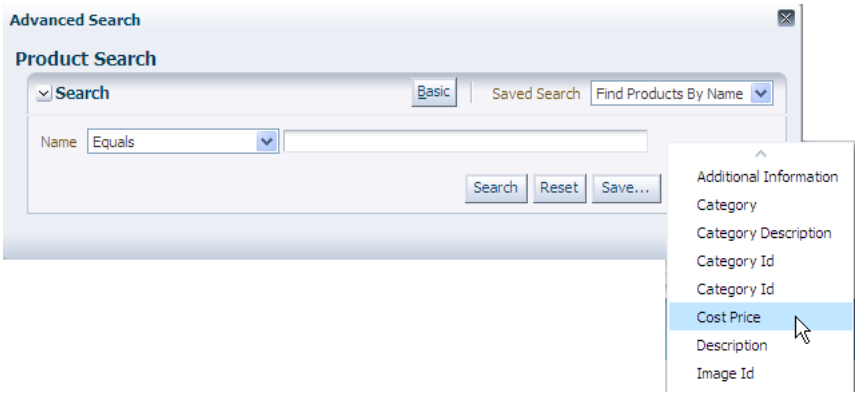
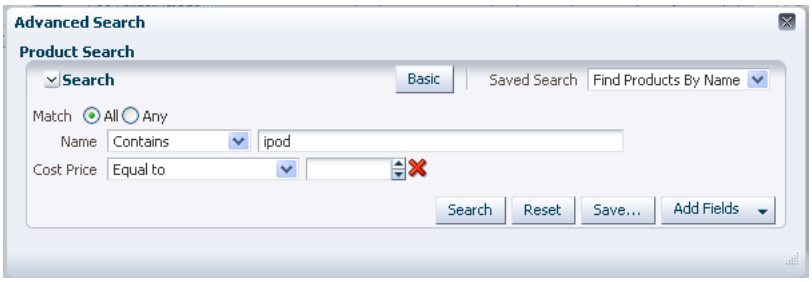


Figure 31–8 shows a user-added search criteria with the delete icon to its right. Users can click the delete icon to remove the criteria.

Figure 31–8 User-Added Search Criteria with Delete Icon



If either **Match All** or **Match Any** is selected and then the user dynamically adds the second instance of a search criterion, then both **Match All** and **Match Any** will be deselected. The user must reselect either **Match All** or **Match Any** before clicking the **Search** button.

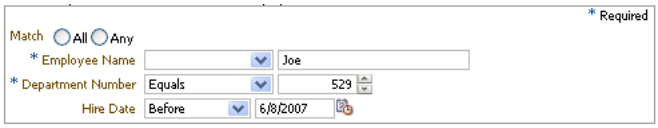
If you intend for a query search form to have both a basic and an advanced mode, you can define each search criteria field to appear only for basic, only for advanced, or for both. When the user switches from one mode to the other, only the search criteria fields defined for that mode will appear. For example, suppose three search fields for basic mode (A, B, C) and three search fields for advanced mode (A, B, D) are defined for a query. When the query search form is in basic mode, search criteria fields A, B, and C will appear. When it is in advanced mode, then fields A, B, and D will appear. Any search data that was entered into the search fields will also be preserved when the form returns to that mode. If the user entered `35` into search field C in basic mode, switched to advanced mode, and then switched back to basic, field C would reappear with value 35.

Along with using the basic or advanced mode, you can also determine how much of the search form will display. The default setting displays the whole form. You can also configure the query component to display in compact mode or simple mode. The compact mode has no header or border, and the **Saved Search** dropdown lists moves next to the expand/collapse icon. Figure 31–9 shows a query component set to compact mode.

Figure 31–9 Query Component in Compact Mode

The simple mode displays the component without the header and footer, and without the buttons normally displayed in those areas. Figure 31–10 shows the same query component set to simple mode.

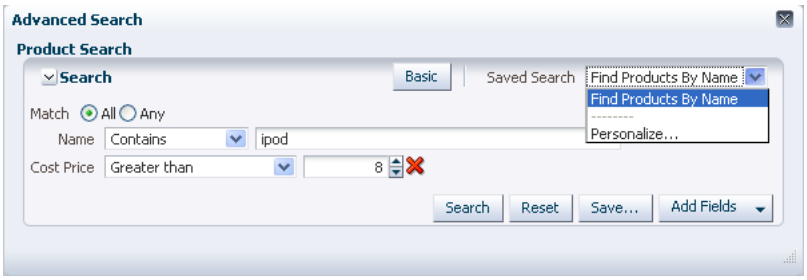
Figure 31–10 Query Component in Simple Mode



A query is associated with the view object that it uses for its query operation. In particular, a query component is the visual representation of the view criteria defined for that view object. If there are multiple view criteria defined, each can be selected from the **Saved Search** dropdown list. These saved searches are created at design time and are called *system searches*. For example, in the StoreFront module of the Fusion Order Demo application, there are two view criteria defined on the `ProductsVO` view object. When the query associated with that view criteria is run, both view criteria are available for selection, as shown in Figure 31–11.

Note: Be aware that opening and closing the query panel by clicking the panel expansion/collapse icon does not reset the applied view criteria. You can click the **Reset** button to reset the applied view criteria.

Figure 31–11 Query Form Saved Search Dropdown List



If there are no explicitly defined view criteria for a view object, you can use the default implicit view criteria.

Users can also create saved searches at runtime to save the state of a search for future use. The entered search criteria values, the basic/advanced mode state, and the layout of the results table/component can be saved by clicking the **Save** button to open a Save Search dialog, as shown in Figure 31–12. User-created saved searches persist for the session. If they are intended to be available beyond the session, you must configure a persistent data store to store them. For Oracle ADF, you can use an access-controlled

data source such as MDS. For more information about using MDS, see Chapter 39, "Customizing Applications with MDS."

Figure 31–12 Runtime Saved Search Dialog Window

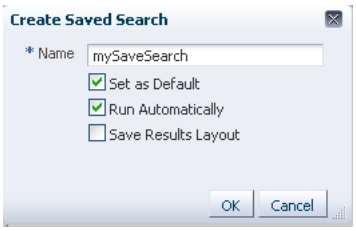


Table 31–1 lists the possible scenarios for creators of saved searches, the method of their creation, and their availability.

Table 31–2 Design Time and Runtime Saved Searches

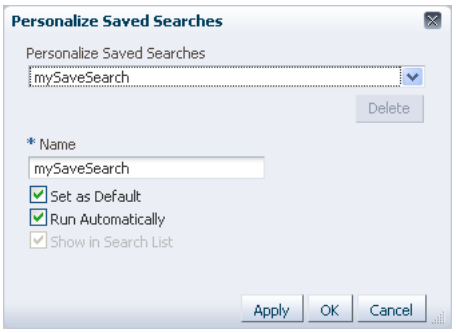
Creator	Created at Design time as View Criteria	Created at Runtime with the Save Button
Developer	Developer-created saved searches (system searches) are created during application development and typically are a part of the software release. They are created at design time as view criteria. They are usually available to all users of the application and appear in the lower part of the **Saved Search** dropdown list.	
Administrator		Administrator-created saved searches are created during predeployment by site administrators. They are created before the site is made available to the general end users. Administrators can create saved searches (or view criteria) using the JDeveloper design time when they are logged in with the appropriate role. These saved searches (or view criteria) appear in the lower part of the **Saved Search** dropdown list.
End User		End-user saved searches are created at runtime using the query form **Save** button. They are available only to the user who created them. End-user saved searches appear in the top part of the **Saved Search** dropdown list.

End users can manage their saved searches by using the Personalize function in the **Saved Search** dropdown list to bring up the Personalize Saved Searches dialog, as shown in Figure 31–13.

End users can use the Personalize function to:

- Update a saved search. Users can add and/or remove search criterion, update operators, update values, modify conjunctions and save it using the existing saved search name. In other words, overriding an existing saved search.
- Delete a saved search (including the currently active Saved Search)
- Set a saved search as the default
- Set a saved search to run automatically
- Set the saved search to show or hide from the **Saved Search** dropdown list

Figure 31–13 Personalize Saved Searches Dialog



Note: If in you are changing the value of a view criteria item programmatically, you must invoke the `ViewCriteria.saveState()` method to prevent the searchRegion binding from resetting the value of the view criteria item to the value that was specified at design time.

You create a query search form by dropping a named view criteria item from the Data Controls panel onto a page. You have a choice of dropping only a search panel, dropping a search panel with a results table, or dropping a search panel with a tree table.

If you choose to drop the search panel with a table, you can select the filtering option in the dialog to turn the table into a filtered table.

Typically, you would drop a query search panel with the results table or tree table. JDeveloper will automatically create and associate a results table or tree table with the query panel.

If you drop a query panel by itself and want a results component or if you already have an existing component for displaying the results, you will need to match the query panel's `ResultsComponentId` with the results component's `Id`.

Note: When you drop a named view criteria onto a page, that view criteria will be the basis for the initial search form. All other view criteria defined against that data collection will also appear in the **Saved Search** dropdown list. Users can then select any of the view criteria search forms, and also any end-user created saved searches.

31.2.1 Named Bind Variables in Query Search Forms

Instead of specifying a literal operand in a view criteria to be used in a search form, you have the option of specifying a named bind variable. The named bind variable performs like a parameter whose value can change at runtime without the need to change the SQL statement. It must be defined in the view object before it can be used in a view criteria.

If you specify a literal operand in the view criteria and leave the value blank, it will not appear in the SQL preview. When the view criteria is applied as a search form at runtime, that attribute is rendered as a blank input search field. If a value is specified for the literal operand, then the SQL preview will generate a SQL clause for it. When the view criteria is applied as a search form at runtime, the SQL statement is not automatically applied even though the value specified in the view criteria appears in the input search field. The SQL statement won't get applied until the user clicks **Search** (or when auto-execute is set to true).

If a named bind variable is used in the query defined in the view criteria, the SQL preview will display the `WHERE` clause with the bind variable. When the view criteria is applied as a search form at runtime, the bind variable will be rendered with a prompt and an input search field based on the name of the attribute (not on the name of the bind variable). The named bind variable input field may be `NULL`, it may contain the default value, or it may contain a value that has loaded from previous processing, such as from another page. The user can enter values for the named bind variable as in any other search criteria. When the search is executed, the value of the named bind variable will be evaluated with the other criteria, as defined by the SQL query statement.

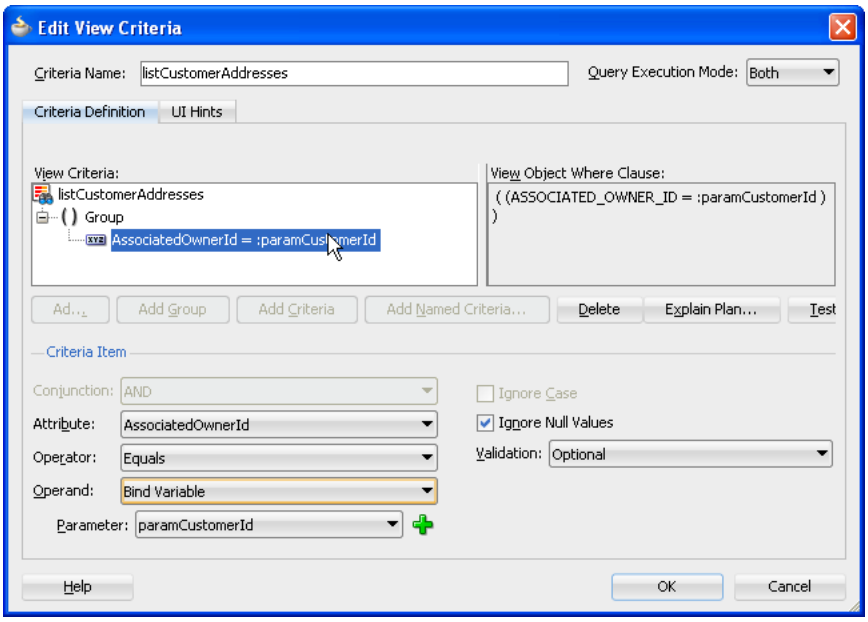
If the bind variable is used more than once in the same view criteria, each occurrence of the bind variable will be rendered as an individual input field. Because there is only one bind variable backing all the input fields, the value of all the fields will be synchronized. For instance, if you specify a view criteria that uses the same bind variable three times, then three input fields will be rendered. When the user enters a value into one input field, the other two input fields will have the same value. Using bind variables in this way eliminates the need for the user to enter the same value multiple times.

Another use of the bind variable is to pass a value from the base row into a search for an LOV search form.

Bind variables can also be used to pass parameter values from one page to another, such as when a customer ID is passed to another page for more detail processing. And, depending on the construct of the SQL statement, using the named bind variable may speed up the query because it may lessen the need to prepare a new statement, which means that the database does not need to reparse. For more information about creating and using named bind variables, see Section 5.10, "Working with Bind Variables."

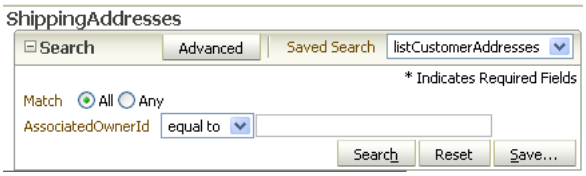
For example, in the StoreFront module of the Fusion Order Demo application, in the `listCustomerAddresses` view criteria, the `WHERE` clause checks to see whether the `AssociatedOwnerId` is the same as the value of the `paramCustomerId` named bind variable. This view criteria is in the `AddressesLookupVO` view object. The `listCustomerAddresses` view criteria as defined in the Edit View Criteria dialog is shown in Figure 31–14.

Figure 31–14 View Criteria with Named Bind Variable



A query search form for a view criteria with a named bind variable will render with a search field for the variable using the `inputText` component. Figure 31–15 shows the **AssociatedOwnerId** search field displayed as an `inputText` component.

Figure 31–15 Query Search Form with Named Bind Variable



31.2.2 How to Create a Query Search Form with a Results Table or Tree Table

You create a search form by dragging and dropping a view criteria from the Data Controls panel onto the page. You have the option of having a results table or only the query panel.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a query search form. For more information, see Section 31.2, "Creating Query Search Forms."

You may also find it useful to understand functionality that can be used with query search forms. For more information, see Section 31.1.4, "Additional Functionality for Search Forms."

You will need to complete these tasks:

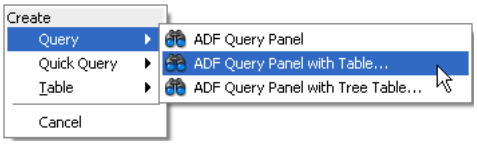
1. Create a view object.

2. Create view criteria if you intend to create search forms based on view criteria. You can also use the default implicit view criteria, which would include all queriable attributes in the collection.
3. Set the search form default properties. For more information about setting the default state of the search form, see Section 31.3.1, "How to Set Search Form Properties on the View Criteria." For information on how to create view criteria, see Section 5.11, "Working with Named View Criteria."

To create a query search form with a results table or tree table:

1. From the Data Controls panel, select the data collection and expand the **Named Criteria** node to display a list of named view criteria.
2. Drag the named view criteria item and drop it onto the page or onto the Structure window.
3. From the context menu, choose **Create > Query > ADF Query Panel with Table** or **Create > Query > ADF Query Panel with Tree Table**, as shown in Figure 31–16.

Figure 31–16 Data Controls Panel with Query Context Menu



4. In the Edit Table Columns dialog, you can rearrange any column and select table options. If you choose the filtering option, the table will be a filtered table.

After you have created the form, you may want to set some of its properties or add custom functions. For more information on how to do this, see Section 31.3, "Setting Up Search Form Properties."

31.2.3 How to Create a Query Search Form and Add a Results Component Later

You create a search form by dragging and dropping a view criteria from the Data Controls panel onto the page. You have the option of having a results table or only the query panel.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a query search form. For more information, see Section 31.2, "Creating Query Search Forms."

You may also find it useful to understand functionality that can be used with query search forms. For more information, see Section 31.1.4, "Additional Functionality for Search Forms."

You will need to complete these tasks:

1. Create a view object.
2. Create view criteria if you intend to create search forms based on view criteria. You can also use the default implicit view criteria, which would include all queriable attributes in the collection.
3. Set the search form default properties. For more information about setting the default state of the search form, see Section 31.3.1, "How to Set Search Form

Properties on the View Criteria." For information on how to create view criteria, see Section 5.11, "Working with Named View Criteria."

To create a query search form and add a results component in a separate step:

1. From the Data Controls panel, select the data collection and expand the **Named Criteria** node to display a list of named view criteria.
2. Drag the named view criteria item and drop it onto the page or onto the Structure window.
3. Choose **Create > Query > ADF Query Panel** from the context menu, as shown in Figure 31–16.
4. If you do not already have a results component, then drop the data collection associated with the view criteria as a component.
5. In the Property Inspector for the table, copy the value of the **Id** field.
6. In the Property Inspector for the query panel, paste the value of the table's ID into the query's **ResultsComponentId** field.

After you have created the search form, you may want to set some of its properties or add custom functions. See Section 31.3, "Setting Up Search Form Properties," for more information.

31.2.4 How to Persist Saved Searches into MDS

If you want saved searches to be persisted to MDS, you need to define the `/persdef` namespace in the `adf-config.xml` file. In addition, you need to perform the regular MDS configuration, such as specifying `metadatapath`. Example 31–3 shows an `adf-config.xml` file with the `/persdef` namespace defined.

Example 31–3 Sample adf-config.xml with /persdef Namespace

```
<persistence-config>
   <metadata-namespaces>
        <namespace path="/persdef" metadata-store-usage="mdsstore"/>
   </metadata-namespaces>
   <metadata-store-usages>
        <metadata-store-usage id="mdsstore" deploy-target="true"
                 default-cust-store="true">
        </metadata-store-usage>
   </metadata-store-usages>
</persistence-config>
```

In order for the added saved searches to be available the next time the user logs in, `cust-config` needs to be defined as part of the MDS configuration. For more information about setting `cust-config` and MDS, see Section 39.2.1, "How to Create Customization Classes."

If you are also saving the layout of the results component, the application must have the ADF PageFlow Runtime and ADF Controller Runtime libraries installed. Set the project's technology scope to include **ADF Page Flow** or automatically include these libraries by using the Fusion Web Application (ADF) application template.

31.2.5 How to Set Default Search Binding Behavior

A *search binding* provides the model-driven behavior for a search form at runtime. The search binding is related to a particular iterator and the view criteria on that iterator, whose runtime defaults are specified by the Binds and Criteria properties of the search

binding, respectively. The behavior of the search binding depends on the **Query Automatically** control hint. For more information about creating named view criteria, see Section 5.11, "Working with Named View Criteria."

The first time a search form is rendered in a task flow, the search binding's runtime behavior, also known as the initial AutoQuery-or-ClearRowSet behavior, is as follows:

- If **Query Automatically** is set to `true`: The search binding automatically executes the iterator when the search binding is initialized in a task flow. The user will be presented with the results of the search based on the view criteria.
- If **Query Automatically** is set to `false`: The search binding clears the row set related to its iterator. The user will be presented with an empty search results component, which allows the user to enter search criteria and click the **Search** button to execute the search.

The **Query Automatically** control hint is available declaratively only for named view criteria. For the All Queriable Attributes view criteria, you can use the `setProperty()` method on the view criteria API interface to configure the hint at runtime. You can also use this method to set the hint on named view criteria. In the method, set the `ViewCriteriaHints.CRITERIA_AUTO_EXECUTE` property to `true`.

If the user changes the search form by selecting a different view criteria from the dropdown **Saved Search** list, the runtime behavior is as follows:

- If **Query Automatically** is set to `true`: The search binding automatically executes the iterator after applying the new view criteria. The user will be presented with the results of the newly selected saved search.
- If **Query Automatically** is set to `false`: The search binding applies the new view criteria only, leaving any existing search results intact. The user is presented with the results of the previous search.

The search binding's `queryPerformed` property evaluates to `true` if the search binding has performed the query, either automatically or by the user clicking the **Search** button. Clicking the **Reset** button in the search form resets the view criteria and sets the `queryPerformed` property to `false`.

The search binding offers two properties that allow you to customize its default behavior: `TrackQueryPerformed` and `InitialQueryOverridden`. You can set the `TrackQueryPerformed` and the `InitialQueryOverridden` properties in the Property Inspector.

A search binding's `TrackQueryPerformed` property controls whether it manages its `queryPerformed` property during runtime at the page flow level or at the individual page or page fragment level. The valid values for `TrackQueryPerformed` are `pageFlow` and `page`, and the default is `pageFlow`. The search binding's behavior is as follows:

- If `TrackQueryPerformed` is set to `pageFlow`, then `queryPerformed` is initialized once per page flow and tracked at the page flow level. The user can navigate away from the search form page, return to the page within the life of the page flow, and the value of the `queryPerformed` flag will remain the same.
- If `TrackQueryPerformed` is set to `page`, then `queryPerformed` is initialized each time the user navigates to the page or page fragment. This happens when the page is navigated to for the first time and when the page is returned from another page within the page flow.

When `TrackQueryPerformed` of a search binding is set to `pageFlow`, its initial AutoQuery-or-ClearRowSet behavior is performed once during the page flow. In

contrast, when `TrackQueryPerformed` is set to `page`, the initial AutoQuery-or-ClearRowSet behavior is performed each time the user visits the page or page fragment.

A search binding's `InitialQueryOverridden` property controls whether it should suppress its initial AutoQuery-or-ClearRowSet behavior the first time the search binding is used in a page flow. If `InitialQueryOverridden` is `true`, then all valid values, including `true`, `false`, or a boolean-valued EL expression, are suppressed. The default value is `false`.

When you set the `InitialQueryOverridden` property to `true`, you are responsible for writing custom application logic to execute the query. Typically, the query should execute after the code applies some view criteria, sets some bind variables, or performs some other programmatic query setup. If your custom code fails to execute the query as expected with `InitialQueryOverridden` set to `true`, unless **Query Automatically** is set to `false`, the framework will still implicitly execute your query the first time during a user session in which there is an iterator binding reference. This occurs because when **Query Automatically** is not set to `false`, the iterator binding's default `executeQueryIfNeeded` behavior takes effect and executes the query.

When `InitialQueryOverridden` evaluates to `true` or boolean `true`, then the initial AutoQuery-or-ClearRowSet behavior is suppressed the first time the search binding is used in a page flow. If `TrackQueryPerformed` is set to `pageFlow`, then only the initial AutoQuery-or-ClearRowsSet behavior (that would have occurred for this search binding) is suppressed.

In contrast, if a search binding's `TrackQueryPerformed` property is set to `page`, then only the initial AutoQuery-or-ClearRowSet behavior is suppressed. Subsequent initial AutoQuery-or-ClearRowSet behaviors that occur due to the user's navigating back to the same page (or page fragment) are not affected by the `InitialQueryOverridden` property.

If you want to avoid the search binding's performing any initial AutoQuery-or-ClearRowSet behavior, then leave the `TrackQueryPerformed` set to `pageFlow` and set `InitialQueryOverridden` to `true`.

You should not use the `RefreshCondition` property of an iterator to reference the `queryPerformed` property of a search binding. Doing so will inadvertently prevent new rows from being created in that iterator's row set until after the search binding's query has been performed.

31.2.6 What You May Need to Know About Dependent Criterion

There are situations when one search criteria is dependent on the value of another criteria. For example, a bug database has a search form with a `Component` and a `Subomponent` search criteria, both defined as LOVs. When the user selects a value from the Component search criteria, that value must be submitted to the model so that `Subomponent` can be filtered and populated with the appropriate search list for the user to choose from.

If you have a search criteria that has dependents, you must set the root and dependent criteria's underlying view attribute's **Auto Submit** control hint to `true`. You can create a custom listener to trigger a partial submit to update the model and refresh the search panel when the query component detects a value change for the root criteria. You need to create a custom listener because the standard `QueryOperationListener` does not handle this event type. You can create a custom `QueryOperationListener` class using the `QueryOperationListener` interface. You then register this class by implementing it in a managed bean or by directly setting it in the JSF page.

31.2.7 What Happens When You Create a Query Form

When you drop a query search form onto a page, JDeveloper creates an `af:query` tag on the page. If you drop a query with table or tree table, then an `af:table` tag or `af:treeTable` tag will follow the `af:query` tag.

Under the `af:query` tag are several attributes that define the query properties. They include:

- The `id` attribute, which uniquely identifies the query.
- The `resultsComponentId` attribute, which identifies the component that will display the results of the query. Typically, this will be the table or tree table that was dropped onto the page together with the query. You can change this value to be the `id` of a different results component. For more information, see Section 31.2.3, "How to Create a Query Search Form and Add a Results Component Later."

In the page definition file, JDeveloper creates an iterator and a `searchRegion` entry in the `executables` section. Example 31–4 shows the sample code for a page definition file.

In the page definition file `executable` section:

- The iterator `binds` property is set to the name of the data collection. In the example, the value is set to `Products`.
- The iterator `id` property is set to a data collection iterator. In the example, the value is set to `SearchProductsIterator`
- If there is more than one view criteria defined for that data collection, the `searchRegion Criteria` property is set to the name of the view criteria that was dropped. In the example, the value is set to `FindByProductNameCriteria`. If there is only one view criteria defined, then there will not be a `Criteria` property.
- The `searchRegion Binds` property is set to the same value as the `iterator id` property. In the example, the value is set to `SearchProductsIterator`
- The `searchRegion id` property is set to the name of the view criteria concatenated with `Query`. In the example, the value is set to `FindByProductNameCriteriaQuery`.

If the query was dropped onto the page with a table or tree, then in the page definition file `bindings` section, a tree element is added with the `Iterbinding` property set to the search iterator. In this example, the value is set to `SearchProductsIterator`. This should be the same iterator defined in the `executable` section.

Example 31–4 Search Form Code in the Page Definition File

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="11.1.1.52.34" id="homePageDef"
                Package="oracle.fodemo.storefront.pageDefs"
                EnableTokenValidation="false">
            <parameters/>
<executables>
     ...
    <iterator Binds="Products" RangeSize="25"
              DataControl="StoreServiceAMDataControl"
              id="SearchProductsIterator"/>
    ...
    <searchRegion Criteria="FindByProductNameCriteria"
```

```
                    Customizer="oracle.jbo.uicli.binding.JUSearchBindingCustomizer"
                    Binds="SearchProductsIterator"
                    id="FindByProductNameCriteriaQuery"/>
    ...
</executables>
<bindings>
    <tree IterBinding="SearchProductsIterator"
          id="SearchProducts">
      <nodeDefinition DefName="oracle.fodemo.storefront.store.queries.ProductsVO"
           Name="SearchProducts">
        <AttrNames>
          <Item Value="ProductId"/>
          <Item Value="ProductName"/>
          <Item Value="ListPrice"/>
          <Item Value="MinPrice"/>
          ,,,
        </AttrNames>
      </nodeDefinition>
    </tree>
    ...
</bindings>
```

31.2.8 What Happens at Runtime: Search Forms

At runtime, the search form displays as a search panel on the page. The search panel will display in either basic mode or advanced mode, depending on the mode control hint when its corresponding view criteria was created. The **Saved Search** dropdown list will contain all the view criteria that are enabled (**Show in List** control hint enabled). The **Match All/Any** conjunction radio button may be enabled.

A search criteria field will be rendered for each search criteria defined in the view criteria. If the **Default List Type** control hint in the view object has been declared as an LOV or a selection list component, the search criteria field component is as shown in Table 31–1.

After the user enters the search criteria and clicks **Search**, a query against the view criteria is executed and the results are displayed in the associated table, tree table, or component.

31.3 Setting Up Search Form Properties

A query search form is based on a view criteria defined in a view object. When you create the view criteria, you also specify some of the search form properties. Later on, when you drop the named criteria onto the page to create a query component, you can specify other search form properties.

Search form properties that can be set on the view object include grouping related and dependent fields together in the query panel. For more information on setting hints in the view object, see Section 5.13.5, "How to Define UI Category Hints."

Search form properties that can be set when the view criteria is being created include:

- Default mode in basic or advanced mode
- Automatic query execution when the page loads
- Rendering of the search criteria field
- Enabling multiple selections for attributes defined as an LOV

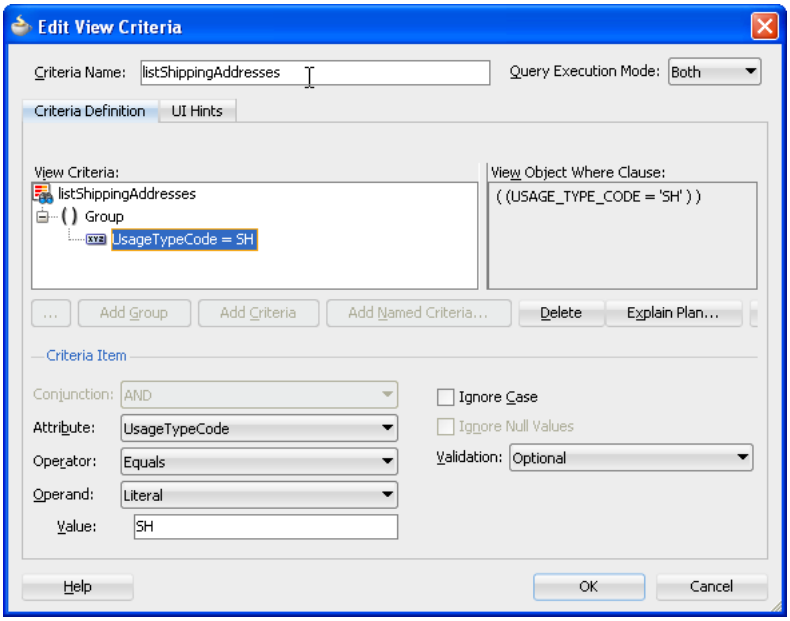
Search form properties that can be set after the query component has been added to the JSF page include:

- `id` of the results table or results component
- Show or hide of the basic/advanced button
- Position of the mode button
- Default, simple, or compact mode for display

31.3.1 How to Set Search Form Properties on the View Criteria

When you are creating a view criteria, you can declaratively set the initial state of several properties. Figure 31–17 shows the Edit View Criteria dialog for setting default options. For more information about view criteria, see Section 5.11, "Working with Named View Criteria."

Figure 31–17 Edit View Criteria Dialog



You must select the default mode of the query search form as either basic or advanced. The default is basic.

You also must declare whether each individual search criteria field will be available only in basic mode, only in advanced mode, available in both modes, or never displayed. If a search criteria field is declared only for basic mode, it will not appear when the user switches to advanced mode, and the reverse is true. If the field is declared for all, then it will appear in all modes. The default for search criteria field rendering is all modes.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a query search form. For more information, see Section 31.3, "Setting Up Search Form Properties."

You may also find it useful to understand functionality that can be used with query search forms. For more information, see Section 31.1.4, "Additional Functionality for Search Forms."

To set the default mode and search criteria field display option:

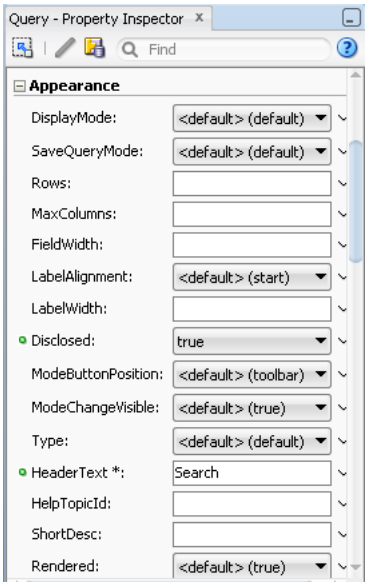
1. While creating a view criteria, in the Edit View Criteria dialog, click **UI Hints**.
2. From the **Search Region Mode** dropdown list, select either **Basic** or **Advanced**.
3. In the **Criteria Item UI Hints** section, select the criteria item you want to set.
4. In the **Rendered Mode** dropdown list, select **All**, **Basic**, **Advanced**, or **Never**.

31.3.2 How to Set Search Form Properties on the Query Component

After you have dropped the query search form onto a page, you can edit other form properties in the Property Inspector, as shown in Figure 31–18. Some of the common properties you may set are:

- Enabling or disabling the basic/advanced mode button
- Setting the ID of the query search form
- Setting the ID of the results component (for example, a results table)
- Selecting the default, simple, or compact mode for display
- Setting the criterionFeatures to `matchCaseDisplayed` to require all string-based search criterion to be case-sensitive or to `requiredDisplayed` to require all the criterion to be displayed.

Figure 31–18 Property Inspector for a Query Component



One common option is to show or hide the basic/advanced button. For more information on some of the other properties, see the "How to Add the Query Component" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*l

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a query search form. For more information, see Section 31.3, "Setting Up Search Form Properties."

You may also find it useful to understand functionality that can be used with query search forms. For more information, see Section 31.1.4, "Additional Functionality for Search Forms."

To enable or hide the basic/advanced button in the query form:

1. In the Structure window, double-click **af:query**.
2. In the Property Inspector, click the **Appearance** tab.
3. To enable the basic/advanced mode button, select **true** from the **ModeChangeVisible** field. To hide the basic/advance mode button, select **false** from the **ModeChangeVisible** field.

31.3.3 How to Create Custom Operators or Remove Standard Operators

You can create custom operators for each view criteria item by adding code to the view object XML file. For example, you can create a new operator called `more than a year`, which operates on a date attribute (greater than 365 days from the current date).

You can also remove standard operators from a view criteria item. For example, you can remove the standard operator `before` from the list.

For a list of standard operators, see the "Using Query Components" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a query search form. For more information, see Section 31.3, "Setting Up Search Form Properties."

You may also find it useful to understand functionality that can be used with query search forms. For more information, see Section 31.1.4, "Additional Functionality for Search Forms."

To add a custom operator:

1. In the Application Navigator, select the view object for the view criteria in which you want to add a custom operator.
2. In the editor window, click the **Source** tab.
3. In the XML editor, locate the code for the view criteria attribute, and add the `CompOper` code statements after the last item within the `ViewCriteriaItem` group.

   In Example 31–5, the `CompOper` code statements appear in bold.

Example 31–5 Adding the Custom Operator Code to the View Object XML

```
<ViewCriteriaRow
  Name="vcrow50"
  UpperColumns="1">
   <ViewCriteriaItem
     Name="LastUpdateDate"
     ViewAttribute="LastUpdateDate"
     Operator="="
     Conjunction="AND"
     Required="Optional">
        <CompOper
            Name="LastUpdateDate"
```

```
            ToDo="1"
            OperDescStrCode="LastUpdateDate_custOp_grt_year"
            Oper=">Y"
            MinCardinality="0"
            MaxCardinality="0" >
            <TransientExpression><![CDATA[ return " < SYSDATE - 365"
                ]]></TransientExpression>
        </CompOper>
    </ViewCriteriaItem>
```

The `CompOper` properties are:

- `Name`: Specify an `id` for the operation.
- `ToDo`: Set to `1` to add this custom operator. Set to `-1` to remove an operator.
- `OperDescStrCode`: Specify the `id` used in the message bundle to map to the description string, as described in Step 4.
- `Oper`: Set to a value that will be used programmatically to denote this operation in the SQL statement. In Example 31–5, `Oper` was set to `>Y` to denote greater than 1 year.
- `MinCardinality`: If there is an input range, set this property to the minimum value for the range. For example, if the range is months in a year, this value should be set to `1`. If there is no range, set it to `0`.
- `MaxCardinality`: If there is an input range, set this property to the maximum value for the range. For example, if the range is months in a year, this value should be set to `12`. If there is no range, set it to `0`.
- `TransientExpression`: Set the expression to perform the custom operator function. In Example 31–5, the expression is `![CDATA[return " > SYSDATE -365"]]`, which returns the string `" > SYSDATE -365"`.

4. Open the message bundle file for the view object and add an entry for the custom operator, using the `OperDescStrCode` identifier defined in the view object XML in Step 3.

   Example 31–6 shows the message bundle code for the `LastUpdateDate` custom operator described in Example 31–5.

Example 31–6 Adding the Custom Operator Entry for LastUpdateDate to the Message Bundle

```
public class AvailLangImplMsgBundle extends JboResourceBundle {
     static final Object[][] sMessageStrings =
  {
       { "LastUpdateDate_custOp_grt_year", "more than a year old"
   },
```

To remove a standard operator:

1. In the Application Navigator, select the view object for the view criteria in which you want to remove a standard operator.

Note: Before you attempt to remove the standard operator, make sure you do not remove the default operator for that view criteria item.

2. In the editor window, click the **Source** tab.
3. In the XML editor, locate the code for the view criteria attribute, and add the `CompOper` code statements after the last item within the `ViewCriteriaItem` group.

   In Example 31–7, the `CompOper` code statements appear in bold. The code in this example removes the `BEFORE` operator from the list of operators for the `LastUpdateDate` attribute.

Example 31–7 Removing a Standard Operator Code in the View Object XML

```
<ViewCriteriaRow
  Name="vcrow50"
  UpperColumns="1">
   <ViewCriteriaItem
     Name="LastUpdateDate"
     ViewAttribute="LastUpdateDate"
     Operator="="
     Conjunction="AND"
     Required="Optional">
        <CompOper
            Name="LastUpdateDate"
            ToDo="-1"
            Oper="BEFORE"
        </CompOper>
   </ViewCriteriaItem>
```

The `CompOper` properties are:

- `Name`: Specify an `id` for the operation.
- `ToDo`: Set to `-1` to remove an operator. Set to `1` to add an operator.

  Do not set `ToDo` to `-2`, which would remove all the operators.
- `Oper`: Set to the standard operator you want to remove from the list.

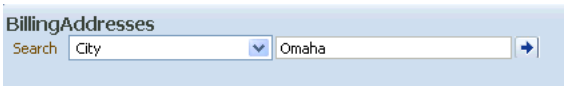
31.4 Creating Quick Query Search Forms

A quick query search form is intended to be used in situations where a single search will suffice or as a starting point to evolve into a full query search. Both the query and quick query search forms are ADF Faces components. A quick query search form has one search criteria field with a dropdown list of the available searchable attributes from the associated data collection. Typically, the searchable attributes are all the attributes in the associated view object. You can exclude attributes by setting the attribute's **Display Hint** property in the Control Hints page of the Edit Attribute dialog to **Hide**. The user can search against the selected attribute or search against all the displayed attributes. The search criteria field type will automatically match the type of its corresponding attribute. An **Advanced** link built into the form offers you the option to create a managed bean to control switching from quick query to advanced mode query search form. For more information, see the "Using Query

Components" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

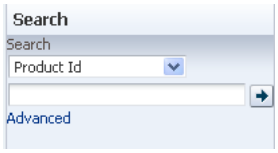
You can configure the form to have a horizontal layout, as shown in Figure 31–19.

Figure 31–19 Quick Query Search Form in Horizontal Layout



You can also choose a vertical layout, as shown in Figure 31–20.

Figure 31–20 Quick Query Search Form in Vertical Layout



You can use quick query search forms to let users search on a single attribute of a collection. Quick query search form layout can be either horizontal or vertical. Because they occupy only a small area, quick query search forms can be placed in different areas of a page. You can create a managed bean to enable users to switch from a quick query to a full query search. For more information about switching from quick query to query using a managed bean, see the "Using Query Components" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

If you drop a quick query panel with a results table or tree, JDeveloper will automatically create the results table, as described in Section 31.4.1, "How to Create a Quick Query Search Form with a Results Table or Tree Table." If you drop a quick query panel by itself and subsequently want a results table or component or if you already have one, you will need to match the quick query `Id` with the results component's `partialTrigger` value, as described in Section 31.4.2, "How to Create a Quick Query Search Form and Add a Results Component Later."

Note: A quick query search creates a dropdown list of all searchable attributes defined in the underlying view object. If you want to show only a subset of those attributes, you can set the attribute's **Display** control hint to **Hide** for those attributes you want to exclude. For more information about setting control hints on view objects, see Chapter 5, "Defining SQL Queries Using View Objects."

31.4.1 How to Create a Quick Query Search Form with a Results Table or Tree Table

You can create quick query searches using the full set of searchable attributes and simultaneously add a table or tree table as the results component.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a query search form. For more information, see Section 31.4, "Creating Quick Query Search Forms."

You may also find it useful to understand functionality that can be used with query search forms. For more information, see Section 31.1.4, "Additional Functionality for Search Forms."

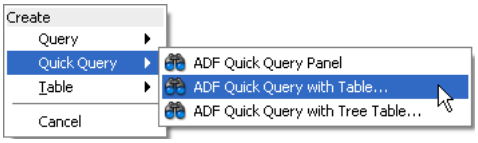
You will need to complete this task:

Create a view object to be the basis of the search form.

To create a quick query search form with a results table:

1. From the Data Controls panel, select the data collection and expand the **Named Criteria** node to display a list of named view criteria.
2. Drag the **All Queriable Attributes** item and drop it onto the page or onto the Structure window.
3. From the context menu, choose **Create > Quick Query > ADF Quick Query Panel with Table** or **Create > Quick Query > ADF Quick Query Panel with Tree Table**, as shown in Figure 31–21.
4. In the Edit Table Columns dialog, you can rearrange any column and select table options. If you choose the filtering option, the table will be a filtered table.

Figure 31–21 Data Control Panel with Quick Query Context Menu



31.4.2 How to Create a Quick Query Search Form and Add a Results Component Later

You can create quick query searches using the full set of searchable attributes and add a table or tree table as the results component later.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a query search form. For more information, see Section 31.4, "Creating Quick Query Search Forms."

You may also find it useful to understand functionality that can be used with query search forms. For more information, see Section 31.1.4, "Additional Functionality for Search Forms."

You will need to complete this task:

Create a view object to be the basis of the search form.

To create a quick query search form and add a results component in a separate step:

1. From the Data Controls panel, select the data collection and expand the **Named Criteria** node to display a list of named view criteria.
2. Drag the **All Queriable Attributes** item and drop it onto the page or onto the Structure window.
3. From the context menu, choose **Create > Quick Query > ADF Quick Query Panel**.
4. If you do not already have a results component, then drop the data collection associated with the view criteria as a component.

5. In the Property Inspector for the quick query panel, copy the value of the **Id** field.
6. In the Property Inspector for the results component (for example, a table), paste or enter the value into the **PartialTriggers** field.

31.4.3 How to Set the Quick Query Layout Format

The default layout of the form is horizontal. You can change the layout option using the Property Inspector.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a query search form. For more information, see Section 31.4, "Creating Quick Query Search Forms."

You may also find it useful to understand functionality that can be used with query search forms. For more information, see Section 31.1.4, "Additional Functionality for Search Forms."

To set the layout:

1. In the Structure window, double-click **af:quickQuery**.
2. In the Property Inspector, on the Commons page, select the **Layout** property using the dropdown list to specify **default**, **horizontal**, or **vertical**.

31.4.4 What Happens When You Create a Quick Query Search Form

When you drop a quick query search form onto a page, JDeveloper creates an `af:quickQuery` tag. If you have dropped a quick query with table or tree table, then an `af:table` tag or `af:treeTable` tag is also added.

Under the `af:quickQuery` tag are several attributes and facets that define the quick query properties. Some of the tags are:

- The `id` attribute, which uniquely identifies the quick query. This value should be set to match the results table or component's `partialTriggers` value. JDeveloper will automatically assign these values when you drop a quick query with table or tree table. If you want to change to a different results component, see Section 31.4.2, "How to Create a Quick Query Search Form and Add a Results Component Later."
- The `layout` attribute, which specifies the quick query layout to be default, horizontal, or vertical.
- The `end` facet, which specifies the component to be used to display the **Advanced** link (that changes the mode from quick query to the query). For more information about creating this function, see the "Using Query Components" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

31.4.5 What Happens at Runtime: Quick Query

At runtime, the quick query search form displays a single search criteria field with a dropdown list of selectable search criteria items. If there is only one searchable criteria item, then the dropdown list box will not be rendered. An input component that is compatible with the selected search criteria type will be displayed, as shown in Table 31–3. For example, if the search criteria type is date, then `inputDate` will be rendered.

Table 31–3 Quick Query Search Criteria Field Components

Attribute Type	Rendered Component
DATE	af:inputDate
VARCHAR	af:inputText
NUMBER	af:inputNumberSpinBox

If the **Default List Type** control hint in the view object has been declared as an LOV or a selection list component, the search criteria field component appears as shown in Table 31–1 in Section 31.1.2, "List of Values (LOV) Input Fields."

In addition, a **Search** button is rendered to the right of the input field. If the end facet is specified, then any components in the end facet are displayed. By default, the end facet contains an **Advanced** link.

31.5 Creating Standalone Filtered Search Tables from Named View Criteria

A filtered table can be created standalone or as the results table of a query or quick query search form. Filtered table searches are based on Query-by-Example and use the QBE text or date input field formats. The input validators are turned off to allow for entering characters such as > and <= to modify the search criteria. For example, you can enter `>1500` as the search criteria for a number column. Wildcard characters may also be supported. If a column does not support QBE, the search criteria input field will not render for that column.

The filtered table search criteria input values are used to build the query `WHERE` clause with the `AND` operator. If the filtered table is associated with a query or quick query search panel, the composite search criteria values are also combined to create the `WHERE` clause.

> **Note:** If the filtered table is used with a query component in a search form and the search region is using an existing named criteria, the results of the query will be filtered by all the view criteria rows in an iterative manner. For example, a filtered table has two view criteria rows: `PersonId` and `DeptId`. The first view criteria row has `PersonId > 1`. When the user enters > 100 in the filter field, then the second view criteria row `DeptId` is used to accept input to further filter the results. This process iterates through all the view criteria rows until the final query result is reached.

Figure 31–22 shows a query search form with a filtered results table. When the user enters a QBE search criteria, such as `>100` for the `PersonId` field, the query result is the `AND` of the query search criteria and the filtered table search criteria.

Figure 31–22 Query Search Form with Filtered Table

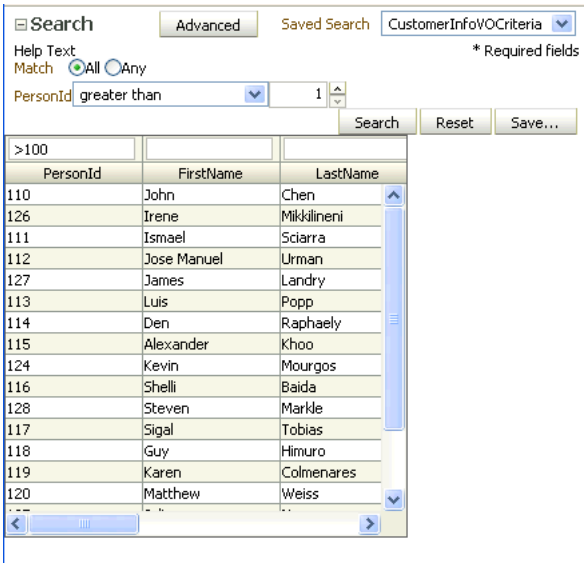


Table 31–4 lists the acceptable QBE search operators that can be used to modify the search value.

Table 31–4 Query-by-Example Search Criteria Operators

Operator	Description
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
AND	And
OR	Or

You use query search forms for complex searches, but you can also perform simple QBE searches using the filtered table. You can create a standalone ADF-filtered table without the associated search panel and perform searches using the QBE-style search criteria input fields. For more information about filtered tables, see Section 31.5.1, "How to Create Filtered Table and Query-by-Example Searches."

You can set the QBE search criteria for each filterable column to be a case-sensitive or case-insensitive search using the `filterFeature` attribute of `af:column` in the `af:table` component. For more information, see the "Enable Filtering in Tables" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

31.5.1 How to Create Filtered Table and Query-by-Example Searches

When creating a table, you can make almost any table a filtered table by selecting the filtering option if the option is enabled. There are three ways to create a standalone filtered table:

- You can drop a table onto a page from the Component Palette, bind it to a data collection, and set the filtering option. For more information, see the "Using Query Components" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
- You can create a filtered table by dragging and dropping a data collection onto a page and setting the filtering option. For more information, see Section 27.2.1, "How to Create a Basic Table."
- You can also create a filtered table or a read-only filtered table by dropping named criteria onto a page. You can use either the implicitly created named criteria **All Queriable Attributes** or any declaratively created named view criteria. The resulting filtered table will have a column for each searchable attribute and an input search field above each column.

Before you begin:

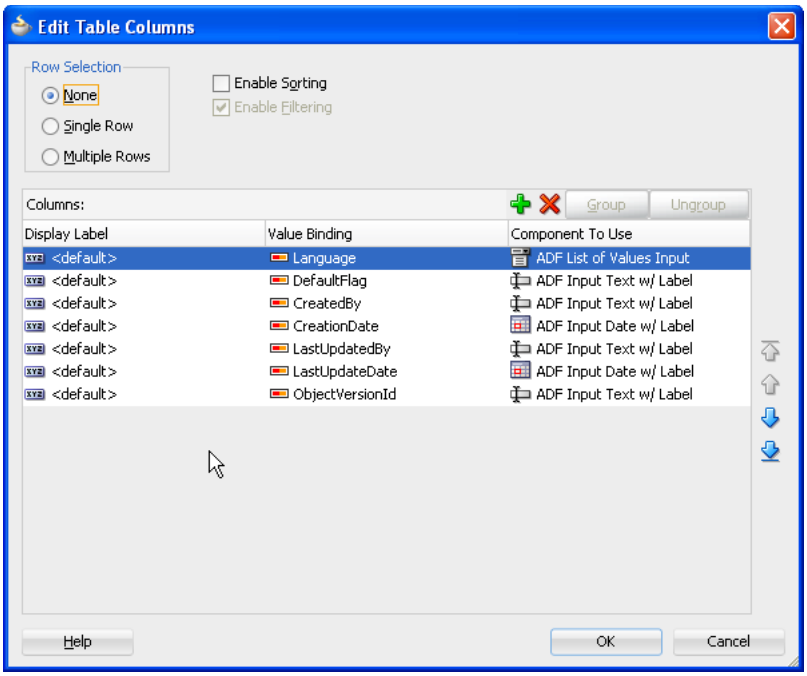
It may be helpful to have an understanding of the options that are available to you when you create a query search form. For more information, see Section 31.5, "Creating Standalone Filtered Search Tables from Named View Criteria."

You may also find it useful to understand functionality that can be used with query search forms. For more information, see Section 31.1.4, "Additional Functionality for Search Forms."

To create a filtered table using named view criteria:

1. From the Data Controls panel, select the data collection and expand the **Named Criteria** node to display a list of named view criteria.
2. Drag the named view criteria item and drop it onto the page or onto the Structure window.
3. From the context menu, choose **Create > Tables > ADF Filtered Table** or **Create > Tables >ADF Read-Only Filtered Table**.
4. In the Edit Table Columns dialog, you can rearrange any column and select table options. Because the table is created by JDeveloper during quick query creation, the filtering option is automatically enabled and not user-selectable, as shown in Figure 31–23.

Figure 31–23 Edit Table Columns Dialog for Filtered Table



32

Using More Complex Databound ADF Faces Components

This chapter describes how to add the ADF Faces Calendar and Carousel components to your pages.

This chapter includes the following sections:

- Section 32.1, "About More Complex Databound ADF Faces Components"
- Section 32.2, "Using the ADF Faces Calendar Component"
- Section 32.3, "Using the ADF Faces Carousel Component"

32.1 About More Complex Databound ADF Faces Components

ADF Faces calendar and ADF Faces carousel are complex components that you can use to include calendar functions or display rotating images in your application. The ADF Faces calendar displays activities in daily, weekly, monthly, or list views for a given provider. The calendar is configurable to only display some of the views. The calendar includes a toolbar with built-in functionality that allows a user to change the view (between daily, weekly, monthly, or list), go to the previous or next day, week, or month, and return to today. The toolbar is customizable and allows you to choose which buttons and text to display, and you can also add buttons or other components.

The ADF Faces carousel displays images in a revolving loop that the user can select by using the slider or clicking another image. It can be configured to have either a horizontal or vertical orientation. You can use other components in conjunction with the carousel. You can add a toolbar or menu bar, and then add buttons or menu items that allow users to perform actions on the current object.

32.1.1 Complex Databound ADF Faces Components Use Cases and Examples

The ADF Faces calendar can be used whenever you want to add a calendar feature to your application. You will need to have the relevant data in your data store that represents the content provided by the calendar, such as the date, time, title, location, and owner. You can use the familiar patterns of entity objects and view objects to model the data and then use drag and drop from the Data Controls panel to create the calendar.

The carousel component gives the user the ability to view an image from a series of images. The user can see partial views of the images before and after the image being viewed and can scroll through each image in the sequence. The user can do so using a slider or navigation buttons. The carousel is useful for showing objects that require a

highly visual presentation. For example, it can be used to display a photographic collection or merchandise in a catalog.

32.1.2 Additional Functionality of Complex Databound ADF Faces Components

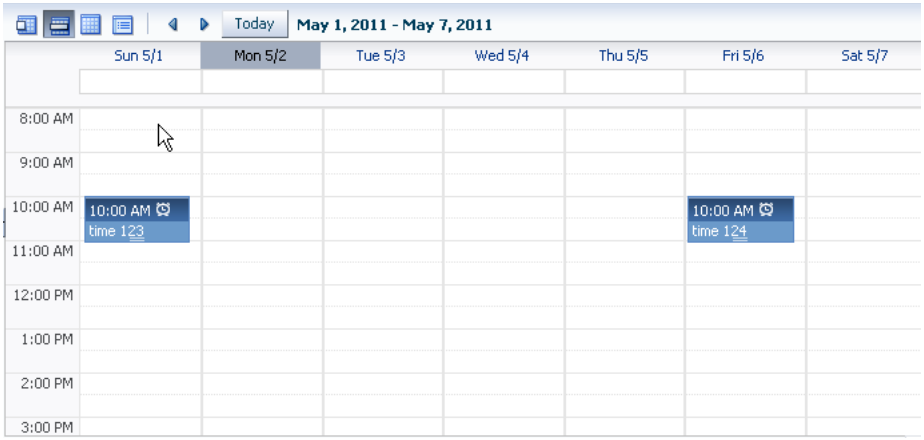
You may find it helpful to understand other ADF features before you configure or use the ADF Model layer. Additionally, you may want to read about what you can do with your model layer configurations. Following are links to other functionality that may be of interest.

- You can customize the calendar for individual users so that the calendar appears in the selected configuration when that user accesses the calendar. For more information, see Chapter 40, "Allowing User Customizations at Runtime."
- Calendars are based on entity objects. For information on creating entity objects, see Chapter 4, "Creating a Business Domain Layer Using Entity Objects."
- For more information about partial page rendering and the partialTriggers attribute, see the "Rerendering Partial Page Content" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
- The carousel component uses a tree binding to iterate over the data. For more information about the tree binding, see Section 27.2.2.1, "Iterator and Value Bindings for Tables."

32.2 Using the ADF Faces Calendar Component

ADF Faces includes a calendar component that displays created activities in daily, weekly, or monthly views. Figure 32–1 shows an ADF Faces calendar in weekly view mode with some sample activities.

Figure 32–1 ADF Faces Calendar



The calendar component also includes the following functionality:

- A toolbar that allows users to switch between monthly, weekly, daily, and list views.

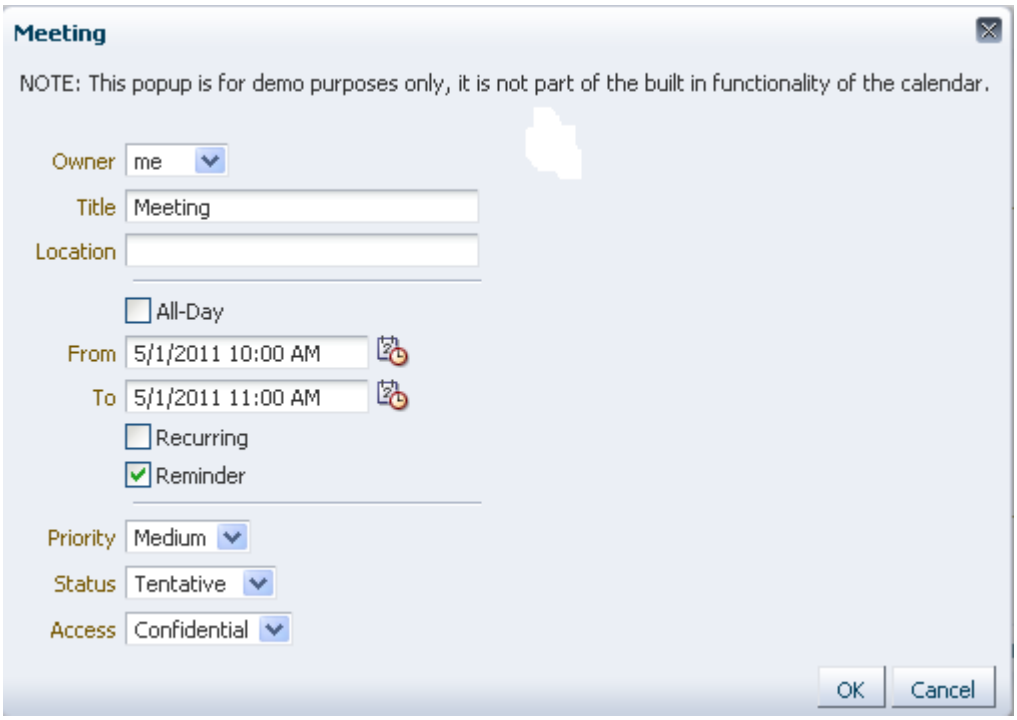
Tip: When these toolbar buttons are used, attribute values on the calendar are changed. You can configure these values to be persisted so that they remain for a particular user whenever they accesses the calendar. For more information, see Chapter 40, "Allowing User Customizations at Runtime."

- Configurable start of the week days and start of the day hours. For example, a calendar's week might start on Sunday and the day might show 8:00 am at the top.
- Configurable styles using skinning keys.

Additionally, you can implement the following functionality using other ADF Faces components and the rich client framework:

- Popup functionality. Components placed in supported faces that respond to certain events and allow the user to act on activities or the calendar. For example, when a user clicks an activity in the calendar, the `CalendarActivityEvent` is invoked and any popup component in the `ActivityDetail` facet is displayed. You might use a dialog component that contains a form where users can view and edit the activity, as shown in Figure 32–2.

Figure 32–2 Edit Dialog for ActivityDetail Facet



- Drag and drop capability: You can add the `calendarDropTarget` tag that allows a user to drag an activity to another place on the calendar. You then implement the functionality so that the time is actually changed on the activity and persisted to the data store.
- Toolbar customization: By default, the toolbar contains buttons that allow the user to switch between the different views, along with previous and next buttons and a button that returns to the current date. The toolbar also displays the current date range (or the date when in day view). You can customize the toolbar by adding facets that contain additional buttons of your choosing.

- Skinning: The calendar uses skinning keys to determine things like colors and icons used. You can extend the skin to change the appearance of the calendar.

Details for configuring the built-in functionality or for implementing additional functionality can be found in the "Using a Calendar Component" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

An ADF Faces Calendar component must be bound to a `CalendarModel` class. This class can be created for you when you use ADF Business Components to manage your calendar's data. For example, say you have data in your data store that represents the details of an activity, such as the date, time, title, location, and owner. When you create an entity object to represent that data, and then a view object to display the data, you can drag and drop the associated collection from the Data Controls panel to create the calendar. JDeveloper will declaratively create the model and bind the view to that model so that the correct data will display when the calendar is launched. However, in order for the model to be created, your entity objects in the data model project with ADF Business Components and your view objects in the same project must contain date-effective attributes. Additionally, the view objects must contain variables that will be used to modify the query to return the correct activities for the given date range.

32.2.1 How to Use the ADF Faces Calendar

Before you can create a calendar on a JSF page, you must first create an entity object with specific attributes that represent attributes on a calendar. You then must create a view object from that entity object, and modify the query to use named bind variables that represent the date range and current time zone to display. This will allow the query to return only the activities that should be displayed in the given view on the calendar.

For example, say you have a database table that represents an activity. It has a column for title, start time, end time, and a reference to a provider object that represents the owner. You would create an entity object and a view object based on that table (ensuring that it meets the requirements, as described in the following steps). To the view object, you would then add named bind variables for the start and end times currently displayed on the calendar, along with the time zone currently in use by the calendar, so that the query returns only those activities that fall within that time range.

Once you add the calendar component to a JSF page, you can configure it, and add further needed functionality.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a calendar. For more information, see Section 32.2, "Using the ADF Faces Calendar Component."

To create an ADF Faces calendar:

1. Create an entity object based on your data source. The entity object must include the attributes shown in Table 32–1. The attributes do not have to use the names shown in the table; they can be named anything. However, they must be of one of the types noted. You will map these attributes to attributes in the `CalendarModel` in a later step.

Table 32–1 Required Attributes for a Calendar

Attribute	Valid Types	Description
Start time	`java.util.Date`, `java.sql.Date`, `oracle.jbo.domain.Date`, `oracle.jbo.domain.TimeStamp`	Start time for the activity
End time	`java.util.Date`, `java.sql.Date`, `oracle.jbo.domain.Date`, `oracle.jbo.domain.TimeStamp`	End time for the activity
ID	`String`	Unique ID
Provider ID	`String`	ID of the provider object that represents the owner of the activity
Title	`String`	Short description of the activity

The entity object can also contain the known (but not required) attributes shown in Table 32–2:

Table 32–2 Optional Attributes for a Calendar

Attribute	Type	Description
Recurring	`String` or `CalendarActivity.Recurring`	Status of recurrence for the activity. Valid values are `SINGLE` (does not recur), `RECURRING`, or `CHANGED` (this activity was part of the recurring activity but has been modified to be different from parent activity).
Reminder	`String` or `CalendarActivity.Reminder`	Whether or not the activity has an associated reminder. Valid values are `ON` or `OFF`.
Time Type	`String` or `CalendarActivity.TimeType`	Type of time associated with the activity. Valid values are `ALLDAY` and `TIME`. Activities that have a value of `ALLDAY` do not have any time associated with them. They are considered to span the entire day. Activities with a value of `TIME` have a specific time duration.
Location	`String`	Location of an activity.
Tags	Set of `Strings` or a semicolon separated list of `Strings`.	Keywords for the activity.

Your entity objects can also contain other attributes that the `CalendarModel` has no knowledge of. You will be able to add these to the model as custom properties in a later step.

For information on creating entity objects, see Chapter 4, "Creating a Business Domain Layer Using Entity Objects."

2. Create an associated view object. In the Query page of the overview editor, create named bind variables for the following:
   - A string that represents the time zone
   - A date that represents the start time for the current date range shown on the calendar.

- A date that represents the end time for the current date range shown on the calendar.

  Tip: Dates in an ADF Faces calendar are "half-open," meaning that the calendar will return all activities that start on or after the start time and before (but not on) the end time.

  For more information about creating named bind variables, see Section 5.10, "Working with Bind Variables."

3. Create an entity object that represents the provider (owner) of activities. The entity object must include the attributes shown in Table 32–3. The attributes do not have to use the names shown in the table; they can be named anything. However, they must be of the type noted. You will map these attributes to attributes in the `CalendarProvider` class in a later step.

Table 32–3 Attributes for a CalendarProvider Class

Attribute	Type	Description
Id	`String`	Unique ID.
Display Name	`String`	The name of the provider that can be displayed in the calendar.

4. Create a view object for the provider.
5. Ensure that the new view objects are part of the application module, and if needed, refresh the Data Controls panel.
6. Create your JSF page, as documented in Section 24.3, "Creating a Web Page."
7. From the Data Controls panel, drag the collection that represents the view object for the activity created in Step 2 and drop it as a **Calendar**.

   Tip: The Calendar option will display in the context menu only if the view object contains the required attributes documented in Table 32–1 and the bind variables described in Step 2.

8. Complete the Calendar Bindings dialog to map the bind variables and attributes to the `CalendarModel` and the `CalendarProvider` classes. For additional help, click **Help** or press **F1**.
9. By default, the calendar will be read-only and will return only those activities currently in the data store. You will need to configure the calendar and implement additional functionality as described in the "Using a Calendar Component" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

   For example, to allow creation of a new activity, you might create an input form in a dialog (as described in the "How to Create a Dialog" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*) using the same data control collection used to create the calendar. For more information about creating input forms, see Section 26.6, "Creating an Input Form."

32.2.2 What Happens When You Create a Calendar

When you drop a collection as a calendar, JDeveloper:

- Defines an iterator binding to the collection of activities, and another iterator binding to the collection of providers.
- Defines an action binding to the `executeWithParams` operation on the activities collection. It is this operation that will be invoked to execute the query to return the activities to display. Because the operation requires parameters to determine the date range and time zone, `NamedData` elements are also created for each of the parameters (created as named bind variables on the view object). For more information about `NamedData` elements, see Section 28.2.2.2, "Using Parameters in a Method."
- Defines a calendar binding. This binding contains a `node` element that represents a row in the collection and maps the data control attributes to the calendar activity's attributes, as defined when using the wizard. The `value` is the data control attribute and the `type` is the calendar attribute. For any custom defined attributes, the `type` will be `custom` and `value` will be the data control attribute. Each row (node) is represented by a `rowKey`, which is the activity ID.

  There is also a `providerDefinition` element that determines the source and mapping of available providers. This mapping allows the calendar model to filter activities based on the state of the provider (either enabled or disabled).

  Tip: To access a custom attribute, use the `CalendarActivity.getCustomAttributes()` method, passing in the name of the attribute as defined by the `value` element.

Example 32–1 shows the page definition code for a calendar.

Example 32–1 Page Definition Code for a Calendar Binding

```
<executables>
    <iterator Binds="ActivityView1" RangeSize="-1"
              DataControl="AppModuleDataControl" id="ActivityView1Iterator"/>
    <iterator Binds="EmployeesView1" RangeSize="25"
              DataControl="AppModuleDataControl" id="EmployeesView1Iterator"/>
  </executables>
  <bindings>
    <action IterBinding="ActivityView1Iterator" id="ExecuteWithParams"
            RequiresUpdateModel="true" Action="executeWithParams">
      <NamedData NDName="startTime"
                 NDValue="#{bindings.ActivityView1.startDate}"
                 NDType="oracle.jbo.domain.Date"/>
      <NamedData NDName="endTime" NDValue="#{bindings.ActivityView1.endDate}"
                 NDType="oracle.jbo.domain.Date"/>
      <NamedData NDName="timeZone"
                 NDValue="#{bindings.ActivityView1.timeZoneId}"
                 NDType="java.lang.String"/>
    </action>
    <calendar IterBinding="ActivityView1Iterator" id="ActivityView1"
              xmlns="http://xmlns.oracle.com/adf/faces/binding"
              ActionBindingName="ExecuteWithParams">
      <nodeDefinition DefName="model.ActivityView">
        <AttrNames>
          <Item Type="id" Value="Id"/>
          <Item Type="providerId" Value="ProviderId"/>
          <Item Type="title" Value="Title"/>
          <Item Type="startTime" Value="StartTime"/>
          <Item Type="endTime" Value="EndTime"/>
        </AttrNames>
      </nodeDefinition>
```

```
      <providerDefinition IterBindingName="EmployeesView1Iterator">
        <AttrNames>
          <Item Type="id" Value="EmployeeId"/>
          <Item Type="displayName" Value="FirstName"/>
        </AttrNames>
      </providerDefinition>
    </calendar>
  </bindings>
```

JDeveloper inserts code onto the JSF page that binds the calendar value to the `CalendarModel` class, as shown in Example 32–2.

Example 32–2 JSF Page Code for a Calendar

```
<af:form>
  <af:calendar value="#{bindings.ActivityView1.calendarModel}"/>
</af:form>
```

The `CalendarModel` class uses `CalendarActivityDefinition` class to access the calendar binding.

32.2.3 What Happens at Runtime: How the Calendar Binding Works

When the calendar is accessed, the `executeWithParams` operation is invoked, with the value of the `startDate` and `endDate` parameters determined by the value of the calendar component's `view` and `activeDay` attributes. For example, if the `view` attribute is set to `month` and the `activeDay` is set to the current date (say, February 6, 2009), then the value for the `startDate` would be `February 1, 2009` and the `endDate` value would be `February 28, 2009`. By default, the time zone value is taken from the `time-zone` setting in the `trinidad-config.xml` file (for more information, see the "Configuration in trinidad-config.xml" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*). Therefore, the query would be restricted to return only activities that fall within that date range.

When the query returns data, because the calendar component is bound to the `CalendarModel`, the `CalendarModel` uses the `CalendarActivityDefinition` class to access the calendar binding class and map the values from the data source to the calendar, using the mappings provided by the binding.

32.3 Using the ADF Faces Carousel Component

You can display images in a revolving carousel, as shown in Figure 32–3. Users can change the image at the front by using either the slider at the bottom or by dragging another image to the front.

Figure 32–3 Carousel Component

Instead of containing a child `carouselItem` component for each image to be displayed, and then binding these components to the individual images, the `carousel` component is bound to a complete collection and repeatedly renders one `carouselItem` component by stamping the value for each item, similar to the way a tree stamps out each row of data. As each item is stamped, the data for the current item is copied into a property that can be addressed using an EL expression using the `carousel` component's `var` attribute. Once the carousel has completed rendering, this property is removed or reverted back to its previous value. Carousels contain a `nodeStamp` facet, which is a holder for the `carouselItem` component used to display the text and short description for each item, and is also the parent component to the image displayed for each item. For more information about the carousel component, see the "Displaying Images in a Carousel" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

32.3.1 How to Create a Databound Carousel Component

When using a carousel component in a Fusion web application, you create the component using the Data Controls Panel. You also use a managed bean to handle the carousel spin event and for other logic you may need to display your items.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create a calendar. For more information, see Section 32.3, "Using the ADF Faces Carousel Component."

You will need to complete these tasks:

1. Create a view object for the collection to be displayed in the carousel.
2. Define the view object with the following minimum set of attributes:
   - Title, which will be displayed below the image in the carousel

- Short description used for text displayed when the user mouses over the image

To create a databound carousel component:

1. From the Data Controls panel, drag the collection for the view object on to the page and select **Carousel** from the context menu.
2. In the Property Inspector, in the Behavior section, bind the **CarouselSpinListener** to a handler method that handles the spinning of the carousel when you need logic to be executed when the carousel spin is executed. Example 32–3 shows the handler methods that might be used to handle the display of product images for the Products view object used to create the carousel:

Example 32–3 Handler for the CarouselSpinEvent

```
public void handleCarouselSpin(CarouselSpinEvent event)
{
  RichCarousel carousel = getCarousel();
  carousel.setRowKey(event.getNewItemKey());
  detailNodeItem = (JUCtrlHierNodeBinding)carousel.getRowData();
}
public JUCtrlHierNodeBinding getDetailNodeItem()
{
//   Get the initial item
  if(detailNodeItem == null)
  {
    RichCarousel carousel = getCarousel();

    Object oldKey = carousel.getRowKey();
    try
      {
         Object key = carousel.getCurrentItemKey();
          getCarousel().setRowKey(key);
          detailNodeItem = (JUCtrlHierNodeBinding)carousel.getRowData();
       }
    finally
     {
       carousel.setRowKey(oldKey);
      }
    }

  return detailNodeItem;
}
```

3. In the Advanced section of the Property Inspector, click the dropdown menu for the **Bindings** attribute and select Edit. In the Edit Property: Binding dialog, select the managed bean used in Step 2. Create a new property called `carousel`. This will allow the handler methods to access the carousel object.
4. In the Structure window, expand the `carousel` component and the `nodeStamp` facet, then select the `carouselItem` component.
5. Bind the `CarouselItem` component's `text` attribute to the associated property in the data model using the variable value set on the carousel's `var` attribute, which by default is set to `item`. So the value of the `carouselItem`'s `text` attribute would be `item.title` (given that `title` is the property used to access the text used for the carousel items on the data model).

   If you were using the `Products` view object, the value would be `#{item.ProductName}`.

6. In the Advanced section of the Property Inspector, click the dropdown menu for the **Bindings** attribute and select Edit. In the Edit Property: Binding dialog, select the managed bean used in Step 2. Create a new property called `carouselItem`.

7. In the ADF Faces page of the Component Palette, from the Common Components panel, drag an **Image** and drop it as a child to the `carouselItem`.

   In the Insert Image dialog, enter the path to the source for the images, being sure to use the variable for the item in the carousel. For example, the path to the image files for the products would normally be:

   ```
   /imageservlet?detail=#{Products.ProductId}
   ```

   For an image in the carousel, you would use:

   ```
   /imageservlet?detail=#{item.ProductId}
   ```

   For information about setting other attributes of the `carousel` and `carouselItem` components, see the "How to Create a Carousel" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

8. If you want to provide additional information about the items in the carousel, you can drag and drop the same view object onto the page (for example, as a form). For the components in the form to redisplay the information for the current item displayed once the carousel is spun, you need to set the `partialTrigger` attribute of the component containing the form to the carousel component's ID.

   For example, the form that displays the information for each item in Figure 32–3 is contained in a `panelBox` component.The `partialTrigger` attribute for the `panelBox` component is set to `c1`, which is the carousel component's ID. This means that whenever the `carouselItem` invokes the `CarouselSpinEvent`, the `panelBox` will be refreshed, causing it to display information about the item that was just made current. For more information about partial page rendering and the partialTriggers attribute, see the "Rerendering Partial Page Content" chapter of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

   Example 32–4 shows the page code for the carousel displayed in Figure 32–3.

Example 32–4 Partial Trigger Updates the Form to Match the Displayed Carousel Item

```
<af:carousel
        currentItemKey="#{bindings.Products.treeModel.rootCurrencyRowKey}"
  value="#{bindings.Products.treeModel}" var="item"
  id="c1"
 carouselSpinListener="#{carBean.handleCarouselSpin}">
   <f:facet name="nodeStamp">
     <af:carouselItem id="ci1" text="#{item.ProductName}"
                      binding="#{carBean.carouselItem}">
       <af:image source="/imageservlet?detail=#{item.ProductId}"
                 id="i1"/>
     </af:carouselItem>
   </f:facet>
 </af:carousel>
 <af:panelBox text="PanelBox1" id="pb1" partialTriggers="c1">
   <af:panelFormLayout id="pfl1">
     <af:panelLabelAndMessage label="#{bindings.ProductName.hints.label}"
                              id="plam2">
       <af:outputText value="#{bindings.ProductName.inputValue}"
                      id="ot4"/>
```

```
      </af:panelLabelAndMessage>
.
.
.
</af:panelBox>
```

32.3.2 What Happens When You Create a Carousel

When you drop a collection from the Data Controls panel as a carousel, a tree value binding is created. A tree consists of a hierarchy of nodes, where each subnode is a branch off a higher-level node.

The tree binding iterates over the data exposed by the iterator binding. The carousel wraps the result set from the iterator binding in a `treeModel` object, which is an extension of the `collectionModel`. The `collectionModel` allows each item in the collection to be available within the carousel component using the `var` attribute. For more information about the tree binding, see Section 27.2.2.1, "Iterator and Value Bindings for Tables."

JDeveloper adds both a `carousel` component and it's child `carouselItem` component onto the page, as shown in Example 32–5.

Example 32–5 Page Code for a Carousel Component

```
<af:carousel
        currentItemKey="#{bindings.Products.treeModel.rootCurrencyRowKey}"
        value="#{bindings.Products.treeModel}" var="item"
        id="c1"
        carouselSpinListener="#{carBean.handleCarouselSpin}">
   <f:facet name="nodeStamp">
     <af:carouselItem id="ci1" text="#{item.ProductName}"/>
   </f:facet>
 </af:carousel>
```

The carousel value is bound to the `treeModel` for the associated collection, and the `currentItemKey` attribute of the carousel is bound to the `rootCurrencyRowKey` of the binding object. In this example, the carousel iterates over the items in the `Products` iterator binding. The iterator binding binds to a `rowKeySet` that keeps track of the current product. By default, the `currentItemKey` attribute of the carousel is bound to the `rootCurrencyRowKey` of the binding object, which causes the product currently displayed at the front of the carousel to be the root and the current item. The `carouselItem` component accesses the current data object for the current item presented to the carousel tag using the `item` variable.

33

Creating Databound ADF Data Visualization Components

This chapter describes how to use ADF data controls to create ADF Data Visualization components that visually represent business data. Data visualization components allow you to display and analyze data through a wide variety of graphs, several kinds of gauges, a pivot table and pivot filter bar, geographic and thematic maps with multiple layers of information, several kinds of Gantt charts, and a hierarchy viewer.

This chapter includes the following sections:

- Section 33.1, "About ADF Data Visualization Components"
- Section 33.2, "Creating Databound Graphs"
- Section 33.3, "Creating Databound Gauges"
- Section 33.4, "Creating Databound Pivot Tables"
- Section 33.5, "Creating Databound Geographic Maps"
- Section 33.6, "Creating Databound Thematic Maps"
- Section 33.7, "Creating Databound Gantt Charts"
- Section 33.8, "Creating Databound Hierarchy Viewers"

33.1 About ADF Data Visualization Components

ADF Data Visualization components provide extensive graphical and tabular capabilities for visually displaying and analyzing business data. Each component needs to be bound to data before it can be rendered since the appearance of the components is dictated by the data that is displayed. This chapter describes how to bind each component to a data source

Both graph and gauge components render graphical representations of data. However, graphs, which support more than 50 types of charts, allow you to evaluate multiple data points on multiple axes in a variety of ways. Many graph types assist in the comparison of results from one group with the results from another group. In contrast, gauges focus on a single data point and examine that point relative to minimum, maximum, and threshold indicators to identify problems.

The pivot table component produces a grid that supports multiple layers of data labels on the row edge or the column edge of the grid. An optional pivot filter bar represents a page edge that filters the available pivot table data. This component also provides the option of automatically generating subtotals and totals for grid data. Pivot tables let you pivot data layers from one edge to another to obtain different views of your data. For example, a pivot table might initially display total sales data for products

within regions on the row edge, broken out by years on the column edge. If you pivot region and year at runtime, then you end up with total sales data for products within years, broken out by region. At runtime, end users can click buttons that appear in the inner column labels to sort rows in ascending or descending order.

The geographic map component represents business data spatially, enabling you to superimpose multiple layers, also referred to as *themes*, of information on a single map. For example, a map of the United States might use a color theme that provides varying color intensity to indicate the popularity of a product within each state, a pie chart theme that shows sales within product category, and a point theme that identifies the exact location of each warehouse. When all three themes are superimposed on the United States map, you can easily evaluate whether there is sufficient inventory to support the popularity level of a product in specific locations. Geographic maps require a connection to an Oracle MapViewer service, and optionally, a geocoder service to display geographical and political detail.

Thematic map components represents business data as patterns in stylized areas or associated markers and does not require a connection to an Oracle MapViewer service. Thematic maps focus on data without the geographic details in a geographic map. The thematic map is packaged with prebuilt base maps including a USA base map, a world base map, and base maps for continents and regions of the world including EMEA and APAC. Each base map includes several sets of regions and one fixed set of cities. A set of regions or cities is referred to as a layer. Each layer can be bound to a data collection and stylized to represent the data with color and pattern fills, or a data marker, or both. At runtime, only one map layer and its associated data can be displayed at a time, unless the thematic map has been enabled for drilling.

There are three types of ADF Gantt chart components: the project Gantt chart (which focuses on project management), the scheduling Gantt chart, and the resource utilization Gantt chart (both of which focus on resource management). Each Gantt chart displays the following regions combined with a splitter:

- List region content: The left side of the splitter provides a list of tasks (for the project Gantt chart) or a list of resources (for the resource utilization and scheduling Gantt charts). This region can display any number of additional columns of related information, using a tree table model.
- Chart region content: The right side of the splitter consists of an area in which task progress, resource utilization, or resource progress is graphed over time. The ability of the Gantt chart to zoom in or out on its time axis lets you view management information across the desired time period.

The ADF hierarchy viewer component produces an interactive graphic that displays hierarchical data as a set of linked shapes. The shapes and links correspond to the elements and relationships in the data. For example, a hierarchy viewer component might be used to generate an organizational chart based on employee data. At runtime, end users can pan and zoom the graphic and expand, select, and navigate the management hierarchy that the graphic displays.

Each Data Visualization component needs to be bound to data before it can be rendered since the appearance of the components is dictated by the data that is displayed. This chapter describes how to bind each component to a data source.

The prefix `dvt:` occurs at the beginning of each data visualization component name indicating that the component belongs to the ADF Data Visualization Tools (DVT) tag library.

33.1.1 End User and Presentation Features

Visually compelling data visualization components enable end users to understand and analyze complex business data. The components are rich in features that provide out-of-the-box interactivity support. For detailed descriptions of the end user and presentation features for each component, see the following:

- Graph components: "End User and Presentation Features of Graphs" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
- Gauge components: "End User and Presentation Features of Gauges" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
- Pivot table and pivot filter bar components: "End User and Presentation Features of Pivot Tables" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
- Geographic and thematic map components: "End User and Presentation Features of Maps" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
- Gantt chart components: "End User and Presentation Features of Gantt Charts" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
- Hierarchy view components: "End User and Presentation Features of Hierarchy Viewers" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

33.1.2 Data Visualization Components Use Cases and Examples

For detailed descriptions of each data visualization use cases and examples, see the following:

- Graph components: "Graph Component Use Cases and Examples" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
- Gauge components: "Gauge Component Use Cases and Examples" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
- Pivot table and pivot filter bar components: "Pivot Table and Pivot Filter Bar Component Use Cases and Examples" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
- Geographic and thematic map components: "Map Component Use Cases and Examples" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
- Gantt chart components: "Gantt Chart Component Use Cases and Examples" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
- Hierarchy view components: "Hierarchy Viewer Component Use Cases and Examples" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

33.1.3 Additional Functionality for Data Visualization Components

You may find it helpful to understand other ADF Faces features before you data bind your data visualization components. Additionally, once you have added a data visualization component to your page, you may find that you need to add functionality such as validation and accessibility. Following are links to other functionality that data visualization components use:

- Partial page rendering: You may want a DVT component to refresh to show new data based on an action taken on another component on the page. For more information, see the "Rerendering Partial Page Content" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework.*
- Personalization: Users can change the way the DVT components display at runtime, those values will not be retained once the user leaves the page unless you configure your application to allow user customization. For more information, see the "Allowing User Customization on JSF Pages" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework.*
- Accessibility: You can make your DVT components accessible. For more information, see the "Developing Accessible ADF Faces Pages" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework.*
- Skins and styles: You can customize the appearance of DVT components using an ADF skin that you apply to the application or by applying CSS style properties directly using a style-related property (`styleClass` or `inlineStyle`). For more information, see the "Customizing the Appearance Using Styles and Skins" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework.*
- Placeholder data controls: If you know the DVT components on your page will eventually use ADF data binding, but you need to develop the pages before the data controls are ready, then you should consider using placeholder data controls, rather than manually binding the components. Using placeholder data controls will provide the same declarative development experience as using developed data controls. For more information, see Chapter 17, "Designing a Page Using Placeholder Data Controls."

33.2 Creating Databound Graphs

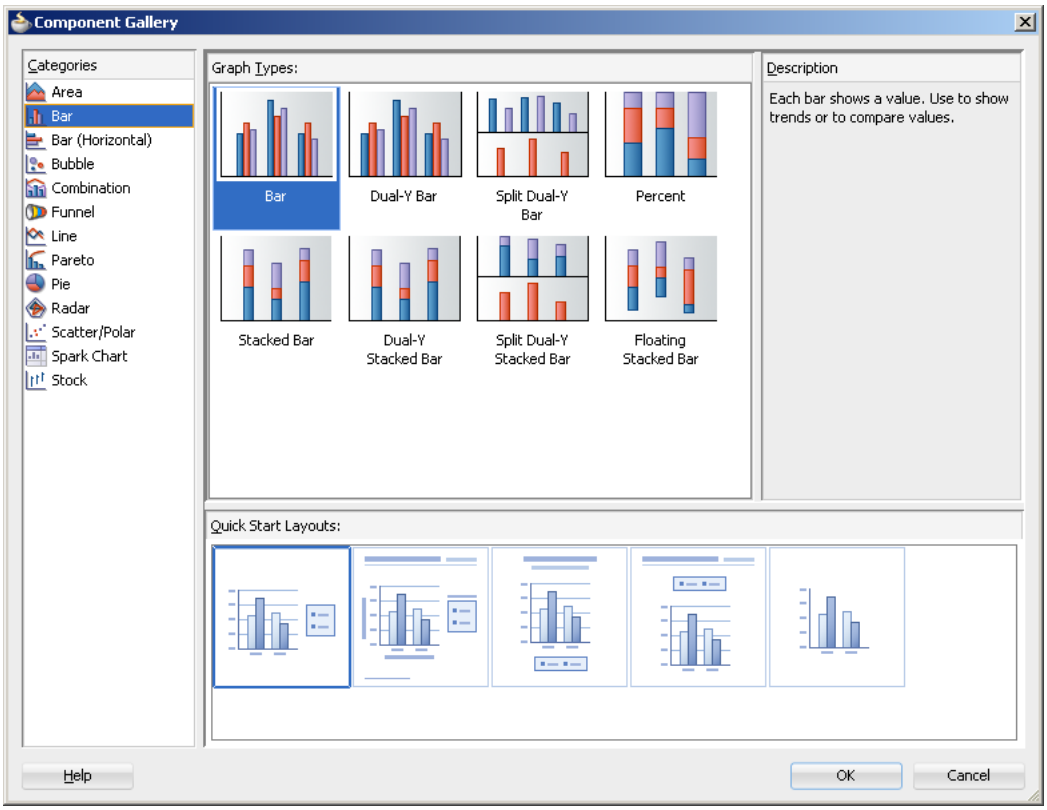
When you create a graph using a data collection inserted from the Data Controls panel, a Component Gallery allows you to choose from a wide number of graph categories, graph types, and layout options. Graph categories group together one or more types of graph. For example, the Bar category includes the following types of graphs:

- Bar
- Dual-Y Bar
- Split Dual-Y Bar
- Percent
- Stacked Bar
- Dual-Y Stacked Bar

- Split Dual-Y Stack Bar
- Floating Stacked Bar

Explore the Component Gallery that appears when you create a graph to view available graph categories, types, and descriptions for each one. Figure 33–1 shows the Component Gallery that appears for ADF graphs when you use the Data Controls panel.

Figure 33–1 ADF Graphs Component Gallery from Data Controls Panel



Note: Some graph types require very specific kinds of data. If you bind a graph to a data collection that does not contain sufficient data to display the graph type requested, then the graph is not displayed and a message about insufficient data appears.

You can also create a graph by dragging a graph component from the Component Palette. This approach allows you the option of designing the graph user interface before binding the component to data. A Component Gallery appears to view graph types, descriptions, and quick layout options.

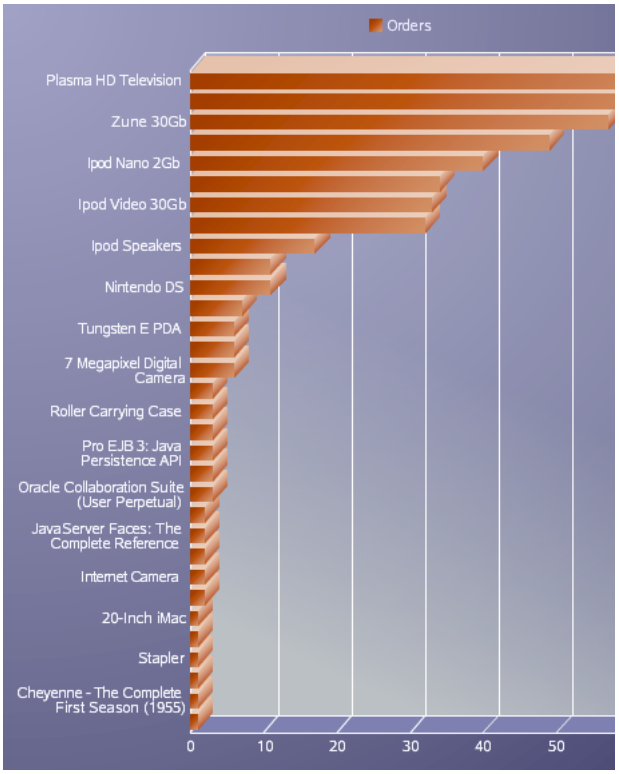
Table 33–1 lists the categories that appear in the Component Gallery for graphs. Each category has one or more graph types associated with it.

Table 33–1 ADF Graph Categories in the Component Gallery

Category	Description
Area	Creates a graph in which data is represented as a filled-in area. Use area graphs to show trends over time, such as sales for the last 12 months. Area graphs require at least two groups of data along an axis. The axis is often labeled with time periods such as months.
Bar	Creates a graph in which data is represented as a series of vertical bars. Use to examine trends over time or to compare items at the same time, such as sales for different product divisions in several regions.
Bar (Horizontal)	Creates a graph that displays bars horizontally along the y-axis. Use to provide an orientation that allows you to show trends or compare values.
Bubble	Creates a graph in which data is represented by the location and size of round data markers (bubbles). Use to show correlations among three types of values, especially when you have a number of data items and you want to see the general relationships. For example, use a bubble graph to plot salaries (x-axis), years of experience (y-axis), and productivity (size of bubble) for your work force. Such a graph allows you to examine productivity relative to salary and experience.
Combination	Creates a graph that uses different types of data markers (bars, lines, or areas) to display different kinds of data items. Use to compare bars and lines, bars and areas, lines and areas, or all three.
Funnel	Creates a graph that is a visual representation of data related to steps in a process. The steps appear as vertical slices across a horizontal cylinder. As the actual value for a given step or slice approaches the quota for that slice, the slice fills. Typically a funnel graph requires actual values and target values against a stage value, which might be time. For example, use this component to watch a process (such as a sales pipe line) move towards a target across the stage of the quarters of a fiscal year.
Line	Creates a graph in which data is represented as a line, as a series of data points, or as data points that are connected by a line. Line graphs require data for at least two points for each member in a group. For example, a line graph over months requires at least two months. Typically a line of a specific color is associated with each group of data such as the Americas, Europe, or Asia. Use to compare items over the same time.
Pareto	Creates a graph in which data is represented by bars and a percentage line that indicates the cumulative percentage of bars. Each set of bars identifies different sources of defects, such as the cause of a traffic fatality. The bars are arranged by value, from the largest number to the lowest number of incidents. A pareto graph is always a dual-y graph in which the first y-axis corresponds to values that the bars represent and the second y-axis runs from 0 to 100% and corresponds to the cumulative percentage values. Use the pareto graph to identify and compare the sources of defects.
Pie	Creates a graph in which one group of data is represented as sections of a circle causing the circle to look like a sliced pie. Use to show relationship of parts to a whole such as how much revenue comes from each product line.
Radar	Creates a graph that appears as a circular line graph. Use to show patterns that occur in cycles, such as monthly sales for the last three years.
Scatter/Polar	Creates a graph in which data is represented by the location of data markers. Use to show correlation between two different kinds of data values such as sales and costs for top products. Scatter graphs are especially useful when you want to see general relationships among a number of items.
Sparkchart	Creates a simple, condensed graph that displays trends or variations, often in the column of a table, or inline with text.
Stock	Creates a graph in which data shows the high, low, and closing prices of a stock. Each stock marker displays three separate values.

Figure 33–2 shows the horizontal bar graph that appears in the Hot Items Statistics page of the `StoreFrontModule` application. This graph displays the quantity of orders for each product so that you can easily identify the items that have been ordered most frequently.

Figure 33–2 Hot Items Statistics Graph



For information about customizing graphs after the data binding is completed, see the "Using ADF Graph Components" chapter in *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

33.2.1 How to Create a Graph

Graphs are based on data collections. Using data controls in Oracle ADF, JDeveloper makes this a declarative task. You drag and drop a collection from the Data Controls panel onto the JSF page and use a dialog to bind the data collection attributes in the graph.

The attributes in a data collection can be data values or categories of data values. Data values are numbers represented by markers, like bar height, or points in a scatter graph. Categories of data values are members represented as axis labels, or appear as additional properties in a tooltip. The role that an attribute plays in the bindings (either data values, or identifiers) is determined by both its data type (Graph requires numeric data values) and where it gets mapped (e.g., Bars vs. X Axis).

To create a databound graph:

1. From the Data Controls panel, select a collection.

   For example, to create the bar graph that displays the order count for each product in the Hot Items Statistics page of the `StoreFrontModule` application, select the

`ProductOrdersCount` collection. Figure 33–3 shows the `ProductOrdersCount` collection in the Data Controls panel.

Figure 33–3 Data Collection for Counting Product Orders

2. Drag the collection onto a JSF page and, from the context menu, choose **Graphs** to display the Component Gallery.
3. Select a graph category and a graph type from the Component Gallery and click **OK**.

   For information about the graph categories and graph types that appear in the Component Gallery, see Table 33–1.

   The name of the dialog and the input field labels that appears depends on the category and type of graph that you select. For example, if you select **Bar (Horizontal)** as the category and **Bar** as the type, then the name of the dialog that appears is Create Horizontal Bar Graph and the input field is labeled **Bars**.
4. Do the following in the dialog to configure the graph to display data:
   - Drag attributes from the **Available** list to the **Bars** or **X Axis** input fields, depending on where you want the values for the attributes to appear at runtime.
   - In the **Attribute Labels** table, accept the default value or select a value from the dropdown list in the **Label** field to specify the label that appears at runtime.

     If an attribute represents data values, then the choices in the **Label** field determine how that attribute's label is rendered in the graph:

     – **Use Attribute Name**: Select to render the value as a string using the label from the UIHints for that attribute in the underlying `ViewObject`. This is the default selection.

     – **No Label**: Select to render no label. This choice is useful if there is a single metric and you want to provide your own descriptive text on the page to describe the resulting graph.

     – **Select Text Resource**: Select to open a Select Text Resource dialog to select or add a text resource to use for the label. The text resource is a translatable string from an application resource bundle. If you need help, press **F1** or click **Help.**

     – **Expression Builder**: Select to open the Expression Builder dialog to create an expression to be executed at runtime for the label. If you need help, press **F1** or click **Help**.

     If the attribute represents a category of data values, then the choices are:

  - **Use Attribute Name**: Select to render the value as a string using the label from the UIHints for that attribute in the underlying `ViewObject`. This is the default selection.
  - From the dropdown list, choose an alternate attribute for the label. For example, use Employee Names for labels instead of Employee IDs.

- If you want to change the display of the data represented in the bars to the Y-Axis, click the **Swap Bars with Y-Axis** button. This action switches the hidden data layer between the graph series and groups, not a simple manual swap of the bars and y-axis attributes.
- If you want to change from the default selection of **Typed Attributes** to **Name-Value Pairs** to configure how data points are stored in a collection, then click the **Change Data Shape** button. A dialog appears that presents you with the following options:
  - Typed Attributes

    Each kind of data point in the collection is represented by a different attribute. This option is also valid when there is only a single kind of data point in the graph.

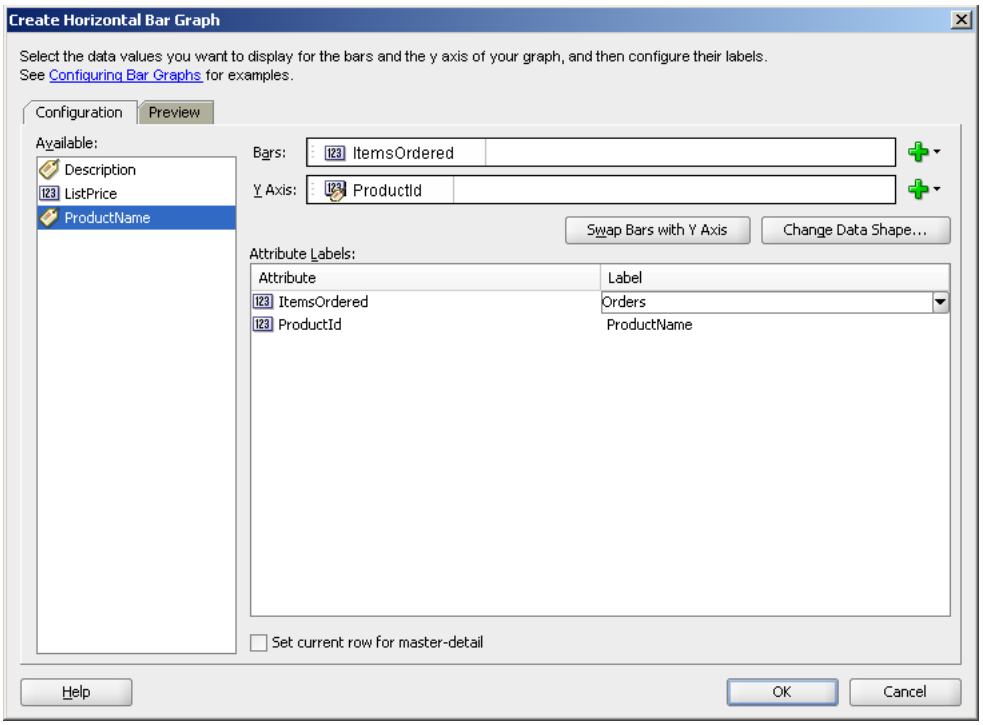
    For example, if you have data points for **Estimated Value** and **Actual Value**, then select **Typed Attributes** only if you have one attribute for the estimated value and a second attribute for the actual value.
  - Name-Value Pairs

    Indicates that there are two or more kinds of data points represented by exactly two attributes; a `Name` attribute that specifies the kind of data point, and a `Value` attribute that specifies the data value.

    For example, the `Name` attribute might have the value `EST` for a `Value` attribute that represents an estimated value, or the `Name` attribute might have a value `ACT` for a `Value` attribute that represents an actual value.
- Select **Set the current row for master-detail** to use the graph's row selection listener to enable clicks on a bar, slice, or other graph data element to update the data in another ADF component. For more information see Section 33.2.3, "What You May Need to Know About Using a Graph's Row Selection Listener for Master-Detail Processing."

Figure 33–4 shows the Create Horizontal Bar Graph dialog that generates a graph using data from the `ItemsOrdered` attribute in the `ProductOrdersCount` data collection. The `ProductName` attribute provides labels for the displayed data.

Figure 33–4 Create Horizontal Bar Graph Dialog for ProductOrdersCount



5. Optionally, click the **Preview** tab to display a live preview of the bar graph and its data. If necessary, go back to the **Configuration** tab so that you can adjust the bar graph specifications.
6. Click **OK**.

After completing the data binding dialog, you can use the Property Inspector to specify settings for the graph attributes and you can also use the child tags associated with the graph tag to customize the graph further.

Note: Use the `emptyText` attribute to specify the text to display when there is no data to return in the graph. The default message is **No data to display**. The attribute accepts HTML for formatting the message and an EL expression that evaluates to the viewable property of the data. If the graph is not viewable (for example, if there are authorization restrictions set against the graph), it displays **Access Denied**.

33.2.2 What Happens When You Use the Data Controls Panel to Create a Graph

Dropping a graph from the Data Controls panel has the following effect:

- Creates the bindings for the graph and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

The data binding XML that JDeveloper generates varies depending on the type of graph you select. The XML represents the physical model of the specific graph type you create. Example 33–1 shows the bindings that JDeveloper generated in the page definition file where a horizontal bar graph was created using data from the `ItemsOrdered` attribute in the `ProductOrdersCount` data collection.

Example 33–1 Binding XML for an ADF Bar (Horizontal) Graph

```
<graph IterBinding="ProductOrdersCountIterator" id="ProductOrdersCount"
       xmlns="http://xmlns.oracle.com/adfm/dvt" type="BAR_HORIZ_CLUST">
  <graphDataMap convert="false" leafOnly="true">
    <series>
      <data>
        <item value="ItemsOrdered"/>
      </data>
    </series>
    <groups>
      <item value="ProductName"/>
    </groups>
  </graphDataMap>
</graph>
```

Example 33–2 shows the code generated for a horizontal bar graph when you drag the `ProductOrdersCount` data collection onto a JSF page.

Example 33–2 JSF Code for an ADF Bar (Horizontal) Graph

```
<dvt:horizontalBarGraph id="horizontalBarGraph1"
                        value="#{bindings.ProductOrdersCount.graphModel}"
                        subType="BAR_HORIZ_CLUST"
                        threeDEffect="true"
                        imageHeight="450" imageWidth="500">
  <dvt:background>
    <dvt:specialEffects/>
  </dvt:background>
  <dvt:graphPlotArea/>
  <dvt:seriesSet>
    <dvt:series/>
  </dvt:seriesSet>
  <dvt:o1Axis/>
  <dvt:y1Axis/>
  <dvt:legendArea automaticPlacement="AP_NEVER"
                  position="LAP_BOTTOM"/>
</dvt:horizontalBarGraph>
```

33.2.3 What You May Need to Know About Using a Graph's Row Selection Listener for Master-Detail Processing

You can use the row selection listener of a graph (which serves as a master view) to enable clicks on a bar, slice, or other graph data element to update the data in another ADF component (which serves as a detail view). For example, a click on a bar that represents sales for a given product in a graph might cause the display of the detailed sales data for that product in a pivot table.

The following requirements must be met to achieve this master-detail processing declaratively:

1. You must use the same data control to provide data for both views as follows:
   a. Bind the graph as a row set to the parent collection in the data control, for example, `DepartmentsView`.
   b. Bind the other ADF view (such as a table or pivot table) to the dependent detail collection in the data control, for example `EmployeesView`.

2. Select **Set the current row for master-detail** in the Create <type> Graph dialog to automatically set a value for the `clickListener` attribute of the graph tag and use the `processClick` method that is already part of the graph binding.

   For example, if the `value` attribute of the graph tag is `value="#{bindings.myGraph.graphModel}"`, then the `clickListener` attribute is set to `clickListener="#{bindings.myGraph.graphModel.processClick}"`.

You do not have to write Java code for handling clicks on data elements in the graph. The `processClick` event on the graph binding recognizes click events on data component in a graph and performs the necessary processing.

33.2.4 What You May Need to Know About Using Name-Value Pairs

When you configure a graph using data controls, by default the data binding dialog supports typed attributes, where each kind of data point in the collection is represented by a different attribute. For example, if you have data points for the sales and costs of multiple products, use typed attributes only if you have one attribute for the sales data of each product, and a second attribute for the cost of each product in the data collection.

Typed attributes data shape is also valid when there is only a single kind of data point in the graph. Figure 33–5 shows a typed attribute data shape.

Figure 33–5 Typed Attribute Data Shape

Product	Sales	Costs
A	10	80
B	20	120
C	30	150
D	40	200

You can select **Change Data Shape** in the data binding dialog to change the dialog to support name-value pairs, where there are two or more kinds of data points represented by exactly two attributes; a `Name` attribute that specifies the kind of data point, and a `Value` attribute that specifies the data value. In this case, each dedicated attribute has values that identify each kind of data point. For example, the Metrics attribute might have values for Sales and Costs for each product in a data collection. Figure 33–6 shows a data collection in a name-value pairs data shape.

Figure 33–6 Name-Value Pairs Data Shape

Product	Metric	Value
A	Sales	100
A	Costs	80
B	Sales	200
B	Costs	120

When using a name-value pairs data shape, specify these values in the graph data binding dialog:

- Name Attribute: the attribute in the data collection representing the `Name` attribute, when its unique values categorize data points into different metrics.
- Value Attribute: the attribute in the data collection representing the numeric data points plotted on the graph. Each data point is categorized as belonging to a particular metric by the corresponding `Name` attribute value.
- Specify the values for the `Value` attribute in the relevant fields for the graph type as defined in the dialog, for example, **Bubble Colors** and **Bubble Tooptips** for a

bubble graph. Values can be attributes in the data collection, or data values, representing a unique value of the `Name` attribute to be used at runtime to categorize data points into specific metrics. You must specify one or more data values, although you are not required to select all the metrics available in the `Name` attribute.

You must add one or more data values by selecting **Data Value** from the field **Add** dropdown list. Each data value represents a unique value of the `Name` Attribute to be used at runtime to categorize data points into specific metrics. You are not required to select all the metrics available in the `Name` Attribute.

After specifying attributes or data values for the input field, you can right click on any attribute to display a context menu for actions such as **Move Right**, **Move Left**, **Delete**, or **Edit Data Value**.

For example, you can create a stock graph that displays open, high, low, and close stock values bound to a data collection with a single numeric column of data in a name-value pairs shape as displayed in Figure 33–7.

33.2.5 How to Create a Graph Using Name-Value Pairs

You can create a graph where the data collection is arranged in a name-value pairs data shape. For example, stock values may be arranged in a single numeric column of data with multiple data points and multiple attributes to designate these points as displayed in the data collection in Figure 33–7.

Figure 33–7 Stock Values in Name-Value Pairs Data Shape

Symbol	MarketDate	Measure	Data
ORCL	2/12/2007	OPEN	16.65
ORCL	2/12/2007	HIGH	16.71
ORCL	2/12/2007	LOW	16.51
ORCL	2/12/2007	CLOSE	16.65
ORCL	2/12/2007	VOLUME	31930300
ORCL	2/13/2007	OPEN	16.71
ORCL	2/13/2007	HIGH	16.74
ORCL	2/13/2007	LOW	16.53
ORCL	2/13/2007	CLOSE	16.62
ORCL	2/13/2007	VOLUME	27438100
ORCL	2/14/2007	OPEN	16.74
ORCL	2/14/2007	HIGH	16.94
ORCL	2/14/2007	LOW	16.68
ORCL	2/14/2007	CLOSE	16.77
ORCL	2/14/2007	VOLUME	42296000
ORCL	2/15/2007	OPEN	16.84
ORCL	2/15/2007	HIGH	16.92

Before You Begin:

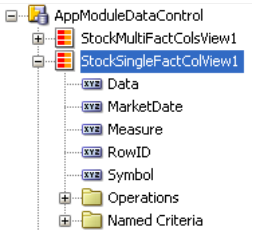
What you may need to know about n-v pairs

To create a graph using a name-value pairs data shape:

1. From the Data Controls panel, select a collection.

   In the example to create a graph that displays the open, high, low, and close stock values over a week's time, select the collection `StockSingleFactColView1`. Figure 33–8 shows the `StockSingleFactColView1` collection in the Data Controls panel.

Figure 33–8 Data Collection for Stock Values in Single Column



2. Drag the collection onto a JSF page and, from the context menu, choose **Graphs** to display the Component Gallery.
3. In the Component Gallery, select the Stock graph category, then the Open-Hi-Lo-Close Candle with Volume graph type, and click **OK**.
4. Do the following to change the dialog to support a name-value pairs data shape:
   - Click **Change Data Shape**.
   - In the Change Chart Data Shape dialog, select **Name-Value Pairs**, and click OK.
   - If prompted, confirm your selection.
5. Do the following in the modified dialog to configure the graph to display data:
   - For Name Attribute, select the attribute category **Measures** from the dropdown list.
   - For Value Attribute, specify the exact Name Attribute values corresponding to the **Open**, **Hi**, **Lo**, and **Close** data attribute from the dropdown list.
   - In the **Chart Values** table, accept the default value or select a value from the dropdown list in the **Label** field to specify the label that appears at runtime.

     If an attribute represents data values, then the choices in the **Label** field determine how that attribute's label is rendered in the graph:

     – **Use Attribute Name**: Select to render the value as a string using the label from the UIHints for that attribute in the underlying `ViewObject`. This is the default selection.

     – **No Label**: Select to render no label. This choice is useful if there is a single metric and you want to provide your own descriptive text on the page to describe the resulting graph.

     – Select Text Resource: Select to open a Select Text Resource dialog to select or add a text resource to use for the label. The text resource is a translatable string from an application resource bundle. If you need help, press **F1** or click **Help.**

     – Expression Builder: Select to open the Expression Builder dialog to create an expression to be executed at runtime for the label. If you need help, press **F1** or click **Help**.

     If the attribute represents a category of data values, then the choices

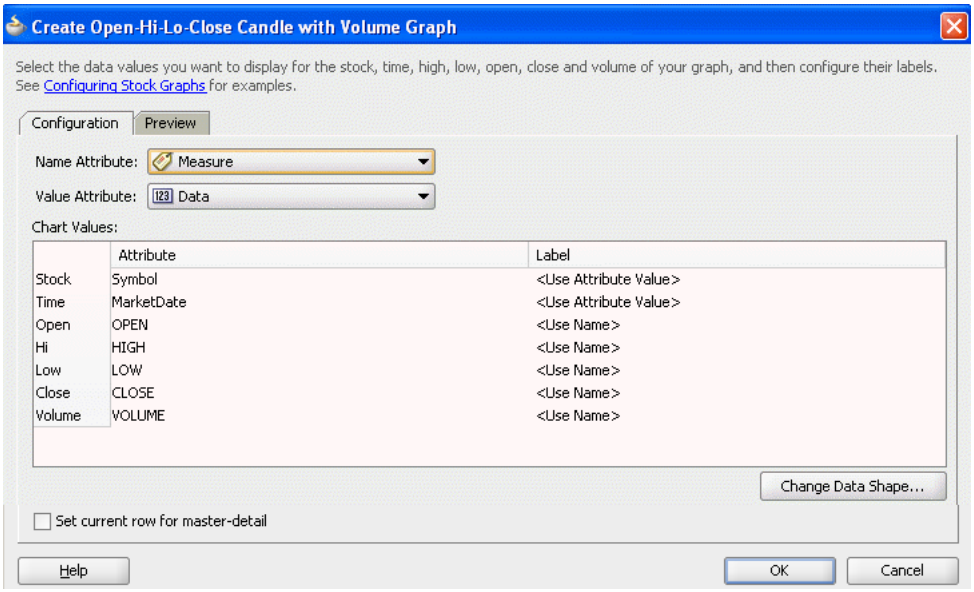
     – **Use Attribute Name**: Select to render the value as a string using the label from the UIHints for that attribute in the underlying `ViewObject`. This is the default selection.

     – From the dropdown list, choose an alternate attribute for the label. For example, use Employee Names for labels instead of Employee IDs.

- **Set the current row for master-detail**: Select to use the graph's row selection listener to enable clicks on a bar, slice, or other graph data element to update the data in another ADF component. For more information see Section 33.2.3, "What You May Need to Know About Using a Graph's Row Selection Listener for Master-Detail Processing."

Figure 33–9 shows the Create Stock Graph dialog that generates a graph using the attribute in the collection.

Figure 33–9 Create Stock Graph for StockSingleFactColView1

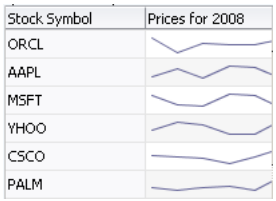


6. Optionally, click the **Preview** tab to display a live preview of the stock graph and its data. If necessary, go back to the **Configuration** tab so that you can adjust the stock graph specifications.
7. Click **OK**.

33.2.6 How to Create a Databound Spark Chart

Spark charts are simple, condensed graphs that display trends or variations, often in the column of a table, or inline with text. Line, bar, and area spark charts require a single series of data values. Figure 33–10 shows an example of a line spark chart in a table column.

Figure 33–10 Line Spark Chart in Table of Stock Prices



Floating bar spark charts require two series of data values, one for the float offset, and one for the bar value. Figure 33–11 shows an example of a floating bar sparkchart.

Figure 33–11 Floating Bar Spark Chart

In a simple UI-first development scenario you can insert a sparkchart using the Component Palette and bind it to data afterwards.

You can provide data to sparkcharts in any of the following ways:

- Specify data statically in child `dvt:sparkItem` tags. Example 33–3 shows an example of providing static data to a sparkchart.

Example 33–3 Static Data in Sparkchart

```
<dvt:sparkChart>
  <dvt:sparkItem value="20"/>
  <dvt:sparkItem value="15"/>
  <dvt:sparkItem value="30"/>
</dvt:sparkChart>
```

- Specify data using EL Expression in child `dvt:sparkItem` tags. Example 33–4 shows an example of providing data using EL Expressions.

Example 33–4 EL Expression Data in Sparkchart

```
<af:table value="#{sample.tableModel}" var="row">
  <af:column>
    <dvt:sparkChart>
      <dvt:sparkItem value="#{row.col1}"/>
      <dvt:sparkItem value="#{row.col2}"/>
      <dvt:sparkItem value="#{row.col3}"/>
    </dvt:sparkChart>
  </af:column>
</af:table>
```

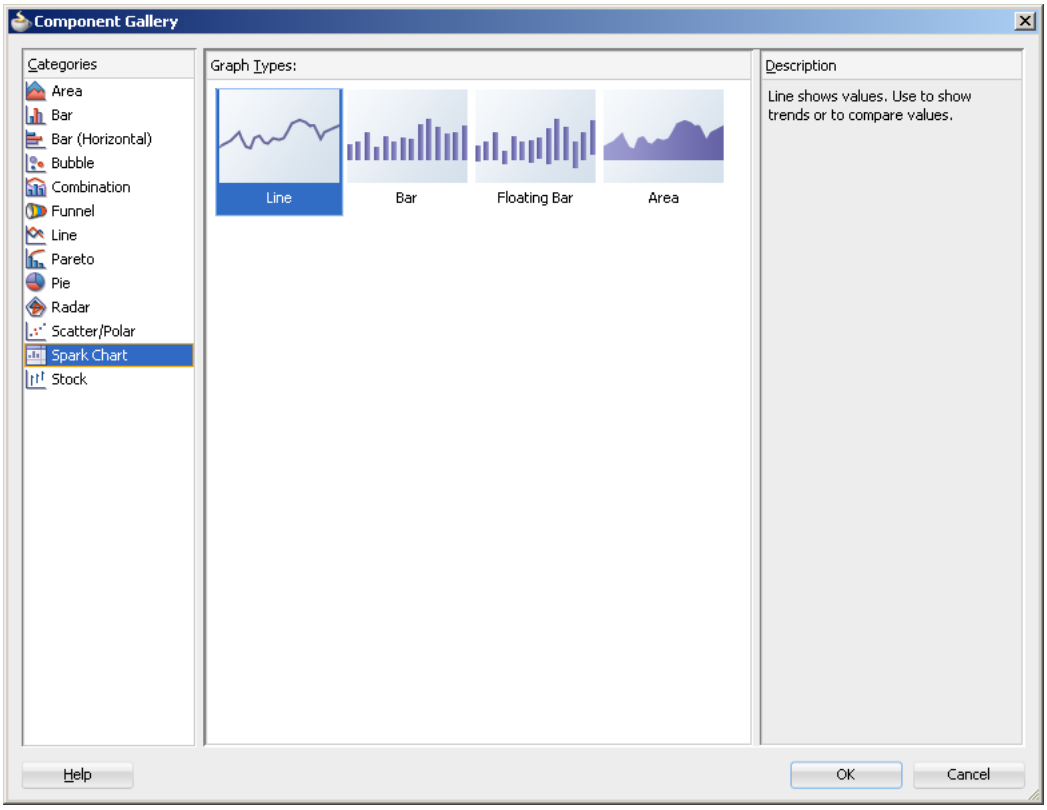
- Specify data using a child accessor in a table. Example 33–5 shows an example of using af:iterator to provide sparkchart data from the data collection model, `row.stockValues`.

Example 33–5 Sparkchart Data in Table Child Accessor

```
<af:table value="#{sample.tableModel}" var="row">
  <af:column>
    <dvt:sparkChart>
      <af:iterator value="#{row.stockValues}" var="data">
        <dvt:sparkItem value="#{data.closeValue}"/>
      </af:iterator>
    </dvt:sparkChart>
  </af:column>
</af:table>
```

You can also create a sparkchart by inserting a data control from the Data Controls Panel. Figure 33–12 shows the Component Gallery that displays when you drag a sparkchart component onto your page from the Data Controls panel.

Figure 33–12 Create Sparkchart Component Gallery

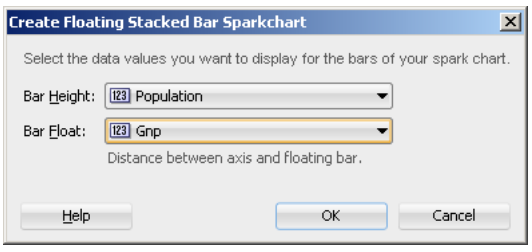


A binding dialog prompts you to specify the value you wish to display for the selected sparkchart type. Line, bar, and area spark charts require a single series of data values, for example the changing value of a stock. Floating bar sparkcharts require two series of data values, one for the float offset, and one for the bar value. For example, in the Create Floating Stacked Bar Sparkchart dialog you specify:

- **Bar Height**: Use to select the data value to use for the bar value.
- **Bar Float**: Use to select the data value to use for the float offset, the distance between the axis and the floating bar.

Figure 33–13 shows a completed Create Floating Stacked Bar Sparkchart dialog.

Figure 33–13 Sparkchart Binding Dialog



33.3 Creating Databound Gauges

A gauge plots one data point with indication of whether the data point falls in an acceptable or unacceptable range. One databound gauge component can create a

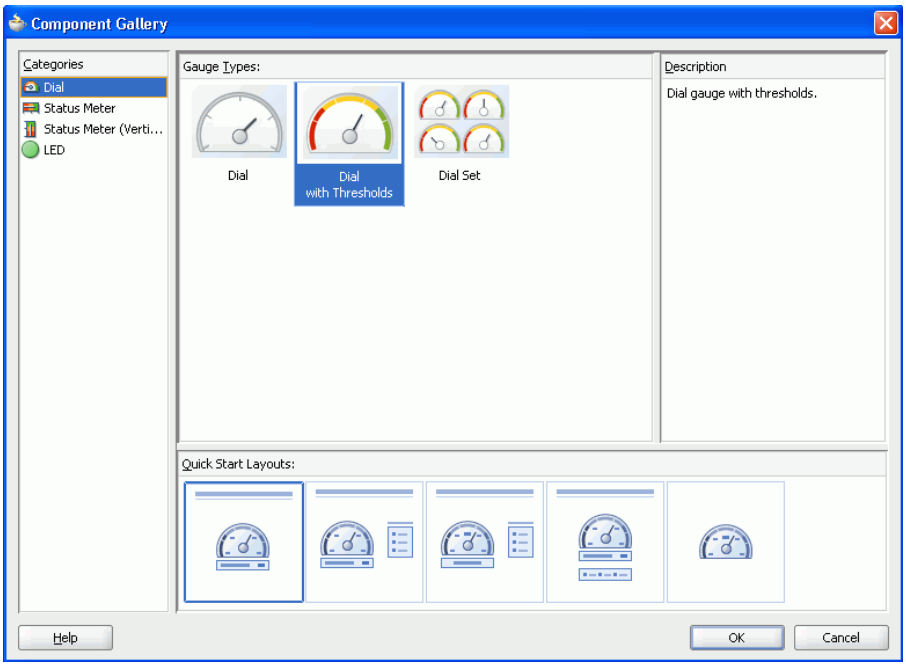
single gauge or an entire set of gauges, depending on the number of rows in the data collection used. In a collection, each row contains the values for a single gauge.

The Component Gallery for gauges allows you to choose from the following categories of gauges:

- Dial: Indicates the metric value of a task along a configurable arc.
- Status Meter: Indicates the progress of a task or the level of some measurement along a rectangular bar.
- Status Meter (Vertical): Indicates the progress of a task or the level of some measurement along a rectangular bar.
- LED: Depicts graphically a measurement such as a key performance indicator (KPI). Several styles of graphics are available for LED gauges such as arrows that indicate good (up arrow), fair (left- or right-pointing arrow), and poor (down arrow).

Each of these categories contains a number of different types of gauge. Explore the Component Gallery that appears when you create a gauge to view all available gauge and category types, and descriptions for each one. Figure 33–14 shows the Component Gallery that appears for ADF gauges.

Figure 33–14 ADF Gauges Component Gallery



The data binding process is essentially the same regardless of which type of gauge you create. Only the metric value (that is, the measurement that the gauge is to indicate) is required. However, if a row in a data collection contains range information such as maximum, minimum, and thresholds, then these values can be bound to the gauge to provide dynamic settings. If information that you want to use in a gauge's upper or lower labels is available in the data collection, then you can bind these values to the gauge also.

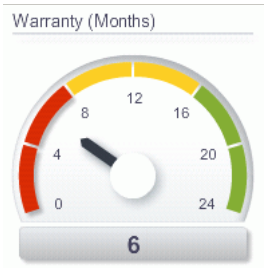
For information about customizing a gauge after the data binding is completed, see the "Using ADF Gauge Components" chapter in *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

33.3.1 How to Create a Databound Dial Gauge

You can use the ADF gauge component to create a dial gauge against a background that specifies ranges of values (also called *thresholds*) that vary from poor to excellent. The gauge indicator specifies the current value of the metric while the graphic allows you to evaluate the status of that value easily.

Figure 33–15 shows a single dial gauge that appears if you create a gauge from the collection that stores `WarrantyPeriodMonths` data. Because only one gauge appears, this data collection must contain only a single row of data. The value of the metric (which is 6) appears in a label below the gauge. The range of values in the gauge is displayed as 0 to 24. Threshold ranges are identified at 8, 16, and 24 and are filled with the colors red for poor (below 8), yellow for adequate (from 8 to 16), and green for superior (above 16).

Figure 33–15 The Warranty in Months Dial Gauge



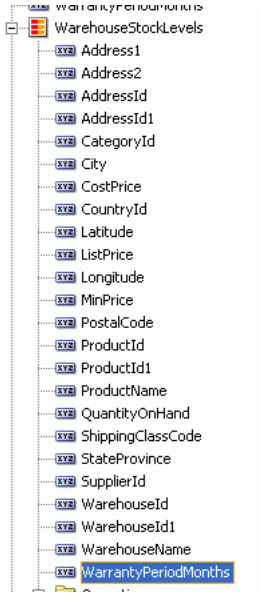
To create a dial gauge using a data control, you bind the gauge component to a collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel.

To create a databound dial gauge:

1. From the Data Controls panel, select a collection.

   For example, to create a dial gauge in the `StoreFrontModule` application to display the current warranty period in months for a product in a particular warehouse, you would select the `WarehouseStockLevels` collection. Figure 33–16 shows the `WarehouseStockLevels` collection in the Data Controls panel.

Figure 33–16 Data Collection with Warranty Period for a Product



2. Drag the collection onto a JSF page and, from the context menu, choose **Gauges**.
3. In the Component Gallery dialog, choose the category, type of gauge, and quick start layout, and then click **OK**.
4. In the Configuration tab of the dialog, do the following:
   - In the **Metric** box, select the column in your data collection that contains the actual value that the gauge is to plot. This is the only required value in the dialog.
   - In the **Minimum** box, if your data collection stores a minimum value for the gauge range, select the column that contains this value.
   - In the **Maximum** box, if your data collection stores a maximum value for the gauge range, select the column that contains this value.
   - In the **Top Label** box, if your data collection stores a value that you want to display in the top label of the gauge, select the column that contains this value.
   - In the **Bottom Label** box, if your data collection stores a value that you want to display in the bottom label of the gauge, then select the column that contains this value.
   - In the **Threshold Attributes** list, if you want to specify threshold values, click the **Add icon** to insert a row for each threshold and specify the value in that row. Do not create a threshold equal to the maximum value for the gauge because the gauge automatically treats the maximum value as a threshold setting.

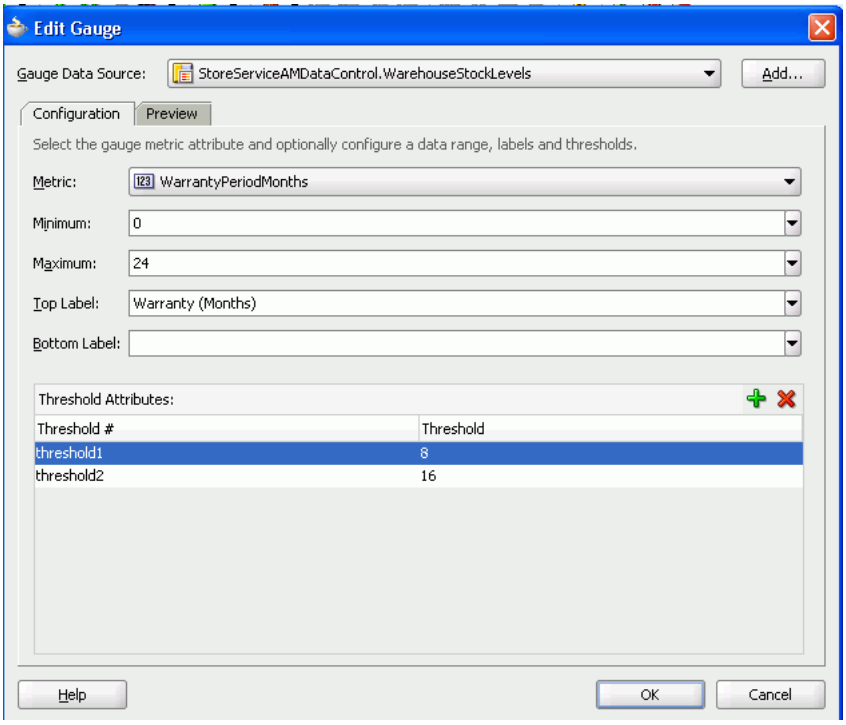
   Note: Gauge Top, Bottom, and Threshold labels can be configured to specify a text resource from a resource bundle using the Select Text Resource dialog, or use the EL Expression builder to evaluate the label text at runtime. Use the dropdown menu for the label field to open both dialogs.

5. Optionally, click the **Preview** tab to display a live preview of the gauge and its data. If necessary, go back to the **Configuration** tab so that you can adjust the gauge specifications.
6. Click **OK**.

You can examine and adjust the gauge bindings for `WarehouseStockLevels` in the Edit Gauge dialog. Figure 33–17 shows the dialog that appears when you click the **Edit** icon in the Property Inspector for the `gauge` component.

> **Note:** The data source and metric data values are required. Other data values in the dialog are optional and can be specified in the gauge component through the Property Inspector.

Figure 33–17 Edit Gauge Dialog



In the Property Inspector, after you complete the binding of the gauge, you can set values for additional attributes in the gauge tag and its child tags to customize the component.

33.3.2 What Happens When You Create a Dial Gauge from a Data Control

Dropping a gauge from the Data Controls panel has the following effect:

- Creates the bindings for the gauge and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

Example 33–6 shows the bindings that JDeveloper generated for the dial gauge that displays warranty in months for a product in a warehouse. This code example shows that the gauge metric receives its value dynamically from the

WarrantyPeriodMonths column in the data collection and that the remaining data values have static settings.

Example 33–6 Bindings for a Dial Gauge

```
<gauge IterBinding="WarehouseStockLevelsIterator" id="WarehouseStockLevels"
       xmlns="http://xmlns.oracle.com/adfm/dvt">
  <gaugeDataMap>
    <item type="threshold" value="8" valueType="literal"/>
    <item type="threshold" value="16" valueType="literal"/>
    <item type="metric" value="WarrantyPeriodMonths"/>
    <item type="minimum" value="0" valueType="literal"/>
    <item type="maximum" value="24" valueType="literal"/>
    <item type="topLabel"
          value="${adfBundle['view.ViewControllerBundle'].
                 WARRANTY_(MONTHS))" valueType="literal"/>
  </gaugeDataMap>
</gauge>
```

Example 33–7 shows the code that JDeveloper generated in the JSF page for a dial gauge. The <dvt:thresholdSet> element specifies one <dvt:threshold> element for each threshold. Colors for the threshold ranges default to red, yellow, and green as specified by the values for the fillColor attributes. The <dvt:indicator> element specifies IT_NEEDLE as the indicator to use. This renders a needle at runtime. The default value for <dvt:indicator> renders a line (IT_LINE).

Example 33–7 Code on the JSF Page for an ADF Dial Gauge

```
<dvt:gauge id="gauge1"
           value="#{bindings.WarehouseStockLevels.gaugeModel}"
           gaugeType="DIAL" imageFormat="FLASH">
  <dvt:gaugeBackground>
    <dvt:specialEffects fillType="FT_GRADIENT">
      <dvt:gradientStopStyle/>
    </dvt:specialEffects>
  </dvt:gaugeBackground>
  <dvt:thresholdSet>
    <dvt:threshold text="Low" fillColor="#d62800"/>
    <dvt:threshold text="Medium" fillColor="#ffcf21"/>
    <dvt:threshold text="High" fillColor="#84ae31"/>
  </dvt:thresholdSet>
  <dvt:gaugeFrame/>
  <dvt:indicator type="IT_NEEDLE"/>
  <dvt:indicatorBase/>
  <dvt:gaugePlotArea/>
  <dvt:tickLabel/>
  <dvt:tickMark/>
  <dvt:topLabel/>
  <dvt:bottomLabel/>
  <dvt:metricLabel position="LP_WITH_BOTTOM_LABEL"/>
</dvt:gauge>
```

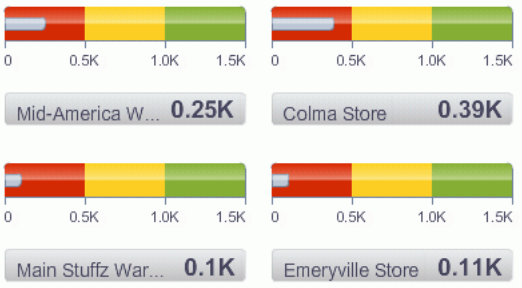
33.3.3 How to Create a Databound Status Meter Gauge Set

You can use the ADF gauge component to create a status meter gauge where the inner rectangle shows the current level of a measurement against the ranges marked in the

outer rectangle. The graphic of the status meter gauge allows you to evaluate the condition or progress of a measurement easily.

Figure 33–18 shows a set of status meter gauges that represent the inventory levels in a number of warehouses. This set of gauges results from binding one gauge component to a data collection (`WarehouseStockLevels`). This data collection contains a row of data for each warehouse. Each row produces one gauge in the set. Notice that all gauges in the set share the same range values of minimum (0) and maximum (1500) with thresholds set at 500 and 1000 and 1500. Each gauge in the set displays the name of the warehouse that it represents and the stock metric for that warehouse in its bottom label.

Figure 33–18 The Warehouse Inventory Status Meter Gauge Set



To create a status meter gauge set using a data control, you bind the gauge component to a data collection that contains multiple rows of data. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel.

To create a databound status meter gauge:

1. From the Data Controls panel, select a collection.

   For example, to create a status meter gauge in the `StoreFrontModule` application that displays the quantity of stock on hand in a warehouse, you select the `WarehouseStockLevels` collection.

2. Drag the collection onto a JSF page and, from the context menu, choose **Gauges**.

3. In the Component Gallery, choose the following:

   - **Status Meter** or **Status Meter (Vertical)** in the **Categories** list
   - The type of gauge that you want to create
   - The quick start layout for the gauge at runtime

4. Click **OK**.

5. In the Create Gauge dialog that appears, select values as described in the following list:

   - Select an attribute binding from the **Metric** dropdown list. This attribute binding contains the actual value that the gauge is to plot.
   - Input a minimum value in the **Minimum** field if your data collection stores a minimum value for the gauge range.
   - Input a maximum value in the **Maximum** field if your data collection stores a maximum value for the gauge range.

- Write or select a value in the **Top Label** field if you want to display a label on top of the gauge at runtime.
- Write or select a value in the **Bottom Label** field if you want to display a label below the gauge at runtime.
- Click the **Add** icon to insert a row for each threshold and specify the value for that threshold if you want to specify threshold values in the **Threshold Attributes** list. Do not create a threshold equal to the maximum value for the gauge because the gauge automatically treats the maximum value as a threshold setting.

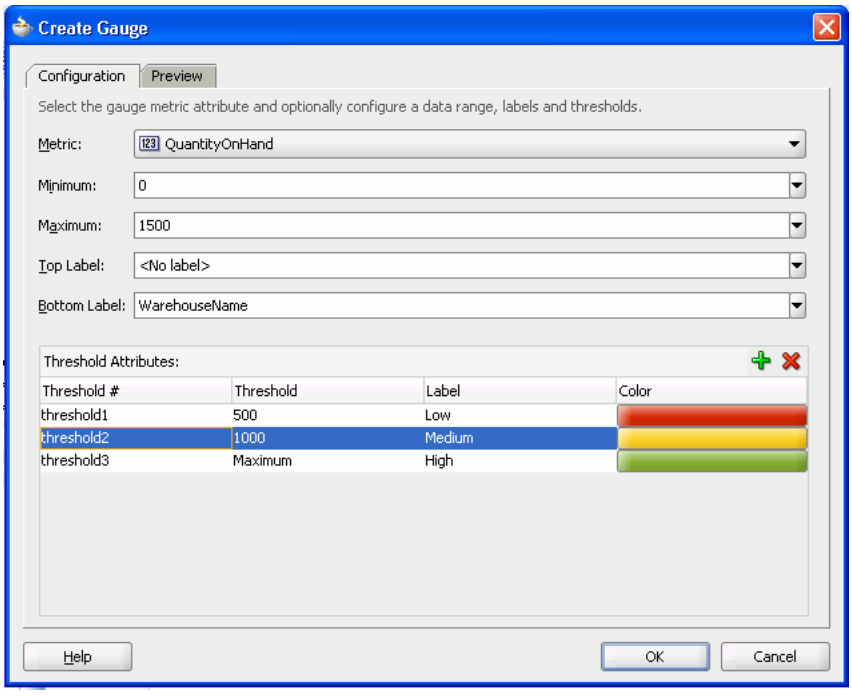
Note: Gauge Top, Bottom, and Threshold labels can be configured to specify a text resource from a resource bundle using the Select Text Resource dialog, or use the EL Expression builder to evaluate the label text at runtime. Use the dropdown menu for the label field to open both dialogs.

---

6. Optionally, click the **Preview** tab to display a live preview of the gauge and its data. If necessary, click the **Configuration** tab so that you can adjust the gauge specifications.

   Figure 33–19 shows the settings for the set of status meter gauges that appears in Figure 33–18. In addition to setting values for the required metric value, this dialog sets values for thresholds and for the name of the warehouse to appear in the gauge's bottom label.

Figure 33–19 Create Gauge Dialog for Warehouse Inventory Gauge Set



7. Click **OK**.

In the Property Inspector, after you complete the binding of the gauge, you can set values for additional attributes in the gauge tag and its child tags to customize the component as needed.

33.3.4 What Happens When You Create a Status Meter Gauge from a Data Control

Dropping a gauge from the Data Controls panel has the following effect:

- Creates the bindings for the gauge and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

Example 33–8 shows the row set bindings that were generated for the status meter gauge set that displays inventory levels for each warehouse as illustrated in Figure 33–19. This example shows the value binding created between the gauge metric attribute and the `QuantityOnHand` value in the data collection. It also shows the value binding between the `Bottom Label` attribute and the `WarehouseName` value in the data collection.

Example 33–8 Bindings Code for the ADF Status Meter Gauge Set

```
<gauge IterBinding="WarehouseStockLevelsIterator" id="WarehouseStockLevels"
       xmlns="http://xmlns.oracle.com/adfm/dvt">
  <gaugeDataMap>
    <item type="threshold" value="500" valueType="literal"/>
    <item type="threshold" value="1000" valueType="literal"/>
    <item type="metric" value="QuantityOnHand"/>
    <item type="minimum" value="0" valueType="literal"/>
    <item type="maximum" value="1500" valueType="literal"/>
    <item type="bottomLabel" value="WarehouseName"/>
  </gaugeDataMap>
</gauge>
```

Example 33–9 shows the code generated on the JSF page for the status meter gauge set that shows inventory levels for warehouses. The `gaugeSetColumnCount` attribute specifies that gauges should be displayed in two columns. The code also specifies three thresholds: Low, Medium, and High. For brevity, the value of the `inlineStyle` attribute has been omitted.

Example 33–9 Code on the JSF Page for the ADF Status Meter Gauge Set

```
<dvt:gauge id="gauge1"
           value="#{bindings.WarehouseStockLevels.gaugeModel}"
           gaugeType="STATUSMETER" imageFormat="FLASH">
  <dvt:gaugeBackground>
    <dvt:specialEffects fillType="FT_GRADIENT">
      <dvt:gradientStopStyle/>
    </dvt:specialEffects>
  </dvt:gaugeBackground>
  <dvt:thresholdSet>
    <dvt:threshold text="Low" fillColor="#d62800"/>
    <dvt:threshold text="Medium" fillColor="#ffcf21"/>
    <dvt:threshold text="High" fillColor="#84ae31"/>
  </dvt:thresholdSet>
  <dvt:indicatorBar/>
  <dvt:gaugePlotArea/>
  <dvt:tickLabel/>
  <dvt:tickMark/>
  <dvt:topLabel/>
```

```
  <dvt:bottomLabel/>
  <dvt:metricLabel position="LP_WITH_BOTTOM_LABEL"/>
</dvt:gauge>
```

33.4 Creating Databound Pivot Tables

The ADF pivot table displays data in a grid layout with unlimited layers of nested rows and columns. The pivot table supports an optional pivot filter bar, representing a page edge that filters the available pivot table data. The pivot table has the following structure:

- Column edge: The horizontal axis above the pivot table containing one or more layers of information in the pivot table.
- Row edge: The vertical axis to the left of the pivot table containing one or more layers of information in the pivot table.
- Page edge: The optional pivot filter bar containing zero or more layers of information for filtering the display of data in the pivot table.
- Data body: One or more measures, or data values, displayed in the cells of the pivot table.

Figure 33–20 shows a Sales pivot table that displays data values for sales and units in the data body, a geography data layer on the column edge, and year and product data layers on the row edge. A pivot filter bar displays a channel filter on the page edge.

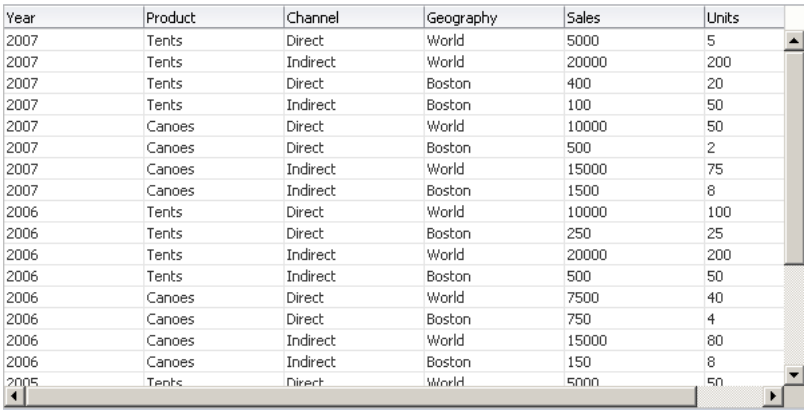
Figure 33–20 Sales Pivot Table

Filters: Direct

		World		Boston		Total Geography	
		Sales	Units	Sales	Units	Sales	Units
▷ 2005		8750	35	500	9	9250	44
▽ 2006		17500	70	1000	15	18500	85
	Tents	10000	100	250	25	10250	125
	Canoes	7500	40	750	4	8250	44
▽ 2007		15000	28	900	11	15900	39
	Canoes	10000	50	500	2	10500	52
	Tents	5000	5	400	20	5400	25
Total Across Years		41250	265	2400	68	43650	333

The pivot table example aggregates data based on a CSV file of data shown in Figure 33–21.

Figure 33–21 Pivot Table CSV Data



Year	Product	Channel	Geography	Sales	Units
2007	Tents	Direct	World	5000	5
2007	Tents	Indirect	World	20000	200
2007	Tents	Direct	Boston	400	20
2007	Tents	Indirect	Boston	100	50
2007	Canoes	Direct	World	10000	50
2007	Canoes	Direct	Boston	500	2
2007	Canoes	Indirect	World	15000	75
2007	Canoes	Indirect	Boston	1500	8
2006	Tents	Direct	World	10000	100
2006	Tents	Direct	Boston	250	25
2006	Tents	Indirect	World	20000	200
2006	Tents	Indirect	Boston	500	50
2006	Canoes	Direct	World	7500	40
2006	Canoes	Direct	Boston	750	4
2006	Canoes	Indirect	World	15000	80
2006	Canoes	Indirect	Boston	150	8
2005	Tents	Direct	World	5000	50

A Create Pivot Table wizard provides declarative support for data-binding and configuring the pivot table. In the wizard pages you can:

- Specify the initial layout of the pivot table
- Associate and configure a pivot filter bar
- Specify alternative labels for the data layers
- Configure insert or filter drilling
- Define aggregation of data values
- Configure category and data sorting
- View a live data preview of the pivot table

As you lay out the pivot table in the first page of the wizard, corresponding entries are initialized in the following wizard pages. You can use the **Back** and **Next** buttons to adjust the pivot table as you go through the wizard pages. You can also skip configuration options in later wizard pages by clicking **Finish**.

For information about customizing a pivot table after data binding is completed, see the "Using ADF Pivot Table Components" chapter in *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

33.4.1 How to Create a Pivot Table

To create a pivot table using a data control, you bind the pivot table component to a collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel.

> **Tip:** You can also create a pivot table by dragging a pivot table component from the Component Palette. This approach allows you the option of designing the pivot table user interface before binding the component to data.

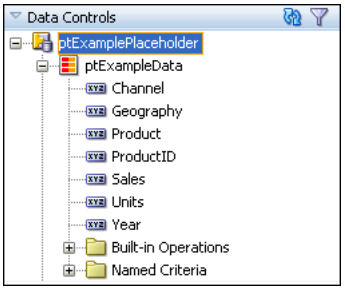
To demonstrate the creation of the sample Sales pivot table shown in Figure 33–20, a placeholder data control was created and data types defined for the CSV file shown in Figure 33–21. For more information, see Section 17.4, "Using Placeholder Data Controls."

To create a databound pivot table:

1. From the Data Controls panel, select a collection.

For example, to create a pivot table that displays sales and units sold by year for each product, you could select the `ptExampleData` defined for the `ptExamplePlaceholder` collection in the Data Controls panel, as shown in Figure 33–22.

Figure 33–22 Data Collection for Product Sales by Year



2. Drag the data collection onto a JSF page and, from the context menu, choose **Tables** > **ADF Pivot Table**.
3. In the Select Display Attributes page of the Create Pivot Table wizard, specify the initial layout of the pivot table by doing the following:
   a. If you want to associate a pivot filter bar with your pivot table, select **Create Pivot Filter Bar**. Optionally, you can drag attributes from the **Available Attributes** list to the page edge to configure the initial display of filters; otherwise, an empty pivot filter bar is created.

      Note: You can add or remove a pivot filter bar after completing the wizard by selecting the `pivotTable` component in the Property Inspector, clicking the **Edit** icon, and checking or unchecking the **Create Pivot Filter Bar** option. You can also drag and drop a `pivotFilterBar` component from the Component Palette adjacent to a pivot table and the bindings will be done for you.

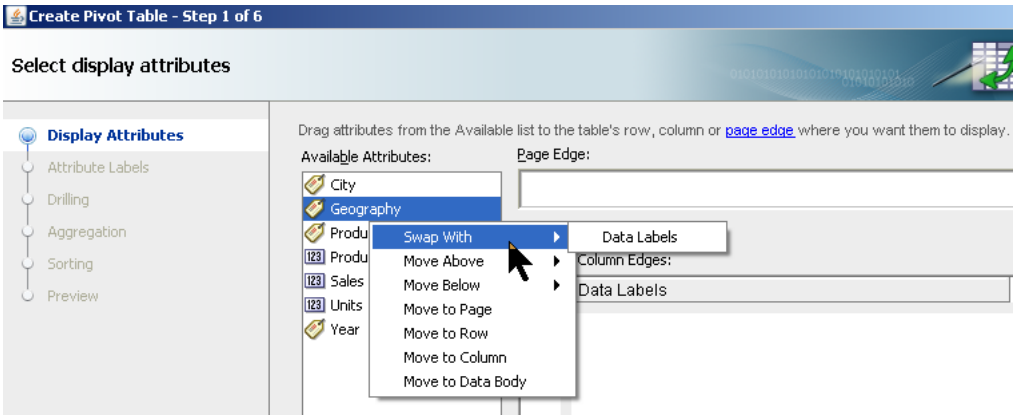
   b. For the initial layout, select the attributes for the pivot table's columns, rows, page edge, and data body by dragging the attributes from the **Available Attributes** list to the pivot table layout.

      In the pivot table layout, **Data Labels** refers to a layer of the pivot table that identifies the data in the cells (data values), and that appears as header labels in the row, column, or page edge. Labels for attributes that you drag to the data body of the pivot table appear in the data labels layer.

      You can drag data labels to any location on the row, column, or page edge. You can also drag attributes to different locations on the same edge or on another edge.

      As an alternative to using a drag operation to place or move attributes in the layout, you can right-click the attribute or use **Shift+F10** to display a context menu of options. Figure 33–23 shows the context menu options for the Geography attribute.

Figure 33–23 Display Attributes Context Menu



Note: Potential drill paths between attributes are defined as you lay out multiple attributes on the row, column, and page edges. These drill paths can later be enabled to support pivot table drilling at runtime.

c. If you want to change from the default selection of **Typed Attributes** to **Name-Value Pairs** to configure how data points are stored in a collection, then click the **Change Data Shape** button. A dialog appears that presents you with the following options:

– **Typed Attributes**

Each kind of data point in the collection is represented by a different attribute. This option is also valid when there is only a single kind of data point in the pivot table.

For example, if you have data points for **Estimated Value** and **Actual Value**, then select **Typed Attributes** only if you have one attribute for the estimated value and a second attribute for the actual value.

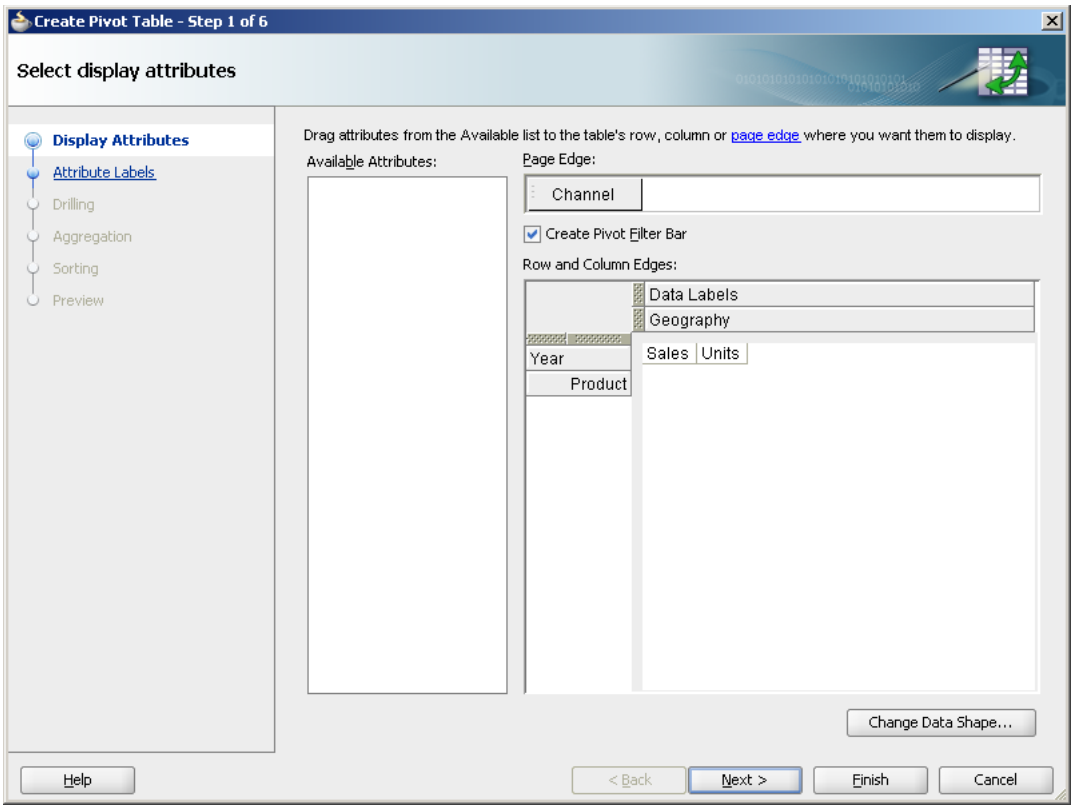
– **Name-Value Pairs**

Indicates that there are two or more kinds of data points represented by exactly two attributes; a `Name` attribute that specifies the kind of data point, and a `Value` attribute that specifies the data value.

For example, the `Name` attribute might have the value `EST` for a `Value` attribute that represents an estimated value, or the `Name` attribute might have a value `ACT` for a `Value` attribute that represents an actual value.

For example, to specify the initial layout of the Sales pivot table shown in Figure 33–20, you would drag the Channel attribute to the page edge, Year and Product attributes to the row edge, Sales and Units attributes to the data body (Data Labels), Geography to the column edge, and select **Create Pivot Filter Bar**, as shown in Figure 33–24.

Figure 33–24 Select Display Attributes Page of Create Pivot Table Wizard



4. If you want to specify alternative labels or values for the attributes laid out in the Select Display Attributes page of the wizard, use the Specify Attribute Labels page to do the following:

   a. To specify alternative labels for data values in the **Data Values** area, change the default `Use Data Attribute Name` text label stamped in the header cell for the attribute at runtime. You can enter the text directly, select `No Label` to suppress the header cell as in the case of using a single data value for the pivot table, specify a text resource from a resource bundle, or use the EL Expression builder to evaluate the label text at runtime.

   Note: If you configured **Project Properties** > **Resource Bundle** page to **Automatically Synchronize Bundle**, then you can type an alternate label string, and the design time code will create a translatable Text Resource for you.

   ---

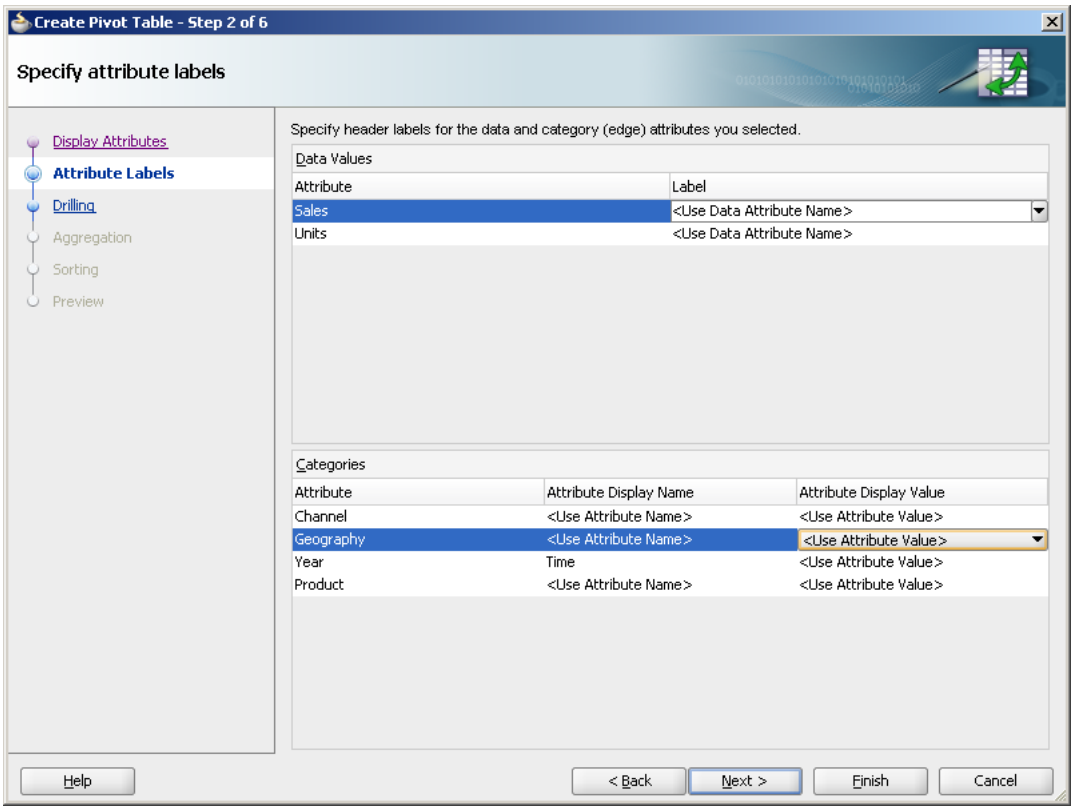
   b. To specify alternative labels for attribute categories in the **Categories** area, change the default `Use Attribute Name` text label stamped in the header cell for the attribute at runtime in the **Attribute Display Name** column. You can enter the text directly, specify a text resource from a resource bundle, or use the EL Expression builder to evaluate the label text at runtime. The label displays in the pivot handle at runtime.

Note: If you configured **Project Properties** > **Resource Bundle** page to **Automatically Synchronize Bundle**, then you can type an alternate label string, and the design time code will create a translatable Text Resource for you.

You can also specify an alternative value for an attribute category by selecting a different attribute in the **Attribute Display Value** column. For example, you might use a `ProductId` attribute in the data collection to lay out the pivot table, but you want the `ProductName` attribute values to appear in the pivot table header at runtime to make the information more readable.

For example, to specify an alternate label for the Year attribute of the Sales pivot table shown in Figure 33–20, you would enter text (Time) in the **Attribute Display Name** field, as shown in Figure 33–25.

Figure 33–25 Specify Attribute Labels Page of Create Pivot Table Wizard



5. If you want to expose drill operations in the pivot table at runtime, use the Configure Drilling page of the Create Pivot Table wizard to enable one of the following options:

   - Select **Insert Drilling** to provide a collapsed or expanded view of the detail data while preserving the sibling and aggregate data. At runtime, a drill icon is displayed in the parent attribute display label.

     Use **Insert Parent Row** to specify whether the aggregate total for the parent attribute will be displayed before or after the child attributes in the expanded view.

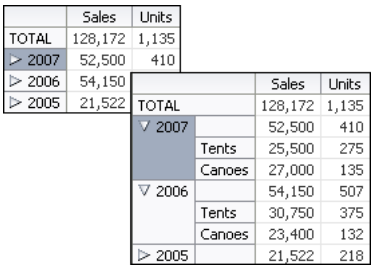
     To enable insert drilling you must also:

– Select the drill paths to enable. Drill paths are configured based upon the layout of the attributes in the Select Display Attributes page of the wizard.

– Configure aggregation in the Configure Aggregation page of the wizard.

For example, Figure 33–26 shows a pivot table using insert drilling to expand the view for the Year data layer. The aggregated value of Sales (52,500 in 2007, 544,150 in 2006) and Units (410 in 2007, 507 in 2006) for each year is displayed in the row above the products.

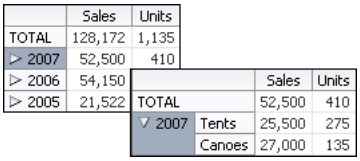
Figure 33–26 Pivot Table with Insert Drilling Enabled



- Select **Filter Drilling** to provide a collapsed or expanded view of the detail data without preserving the sibling or aggregate data. At runtime, a drill icon is enabled in the parent attribute display label.

  Filter drilling focuses the view on the details of the data layer attribute. For example, Figure 33–27 shows a pivot table using filter drilling to expand the view of the Year (2007) data layer, displaying the total Sales (52,500) and Units (410), while filtering out both the data for the other years and the aggregated total for all the years.

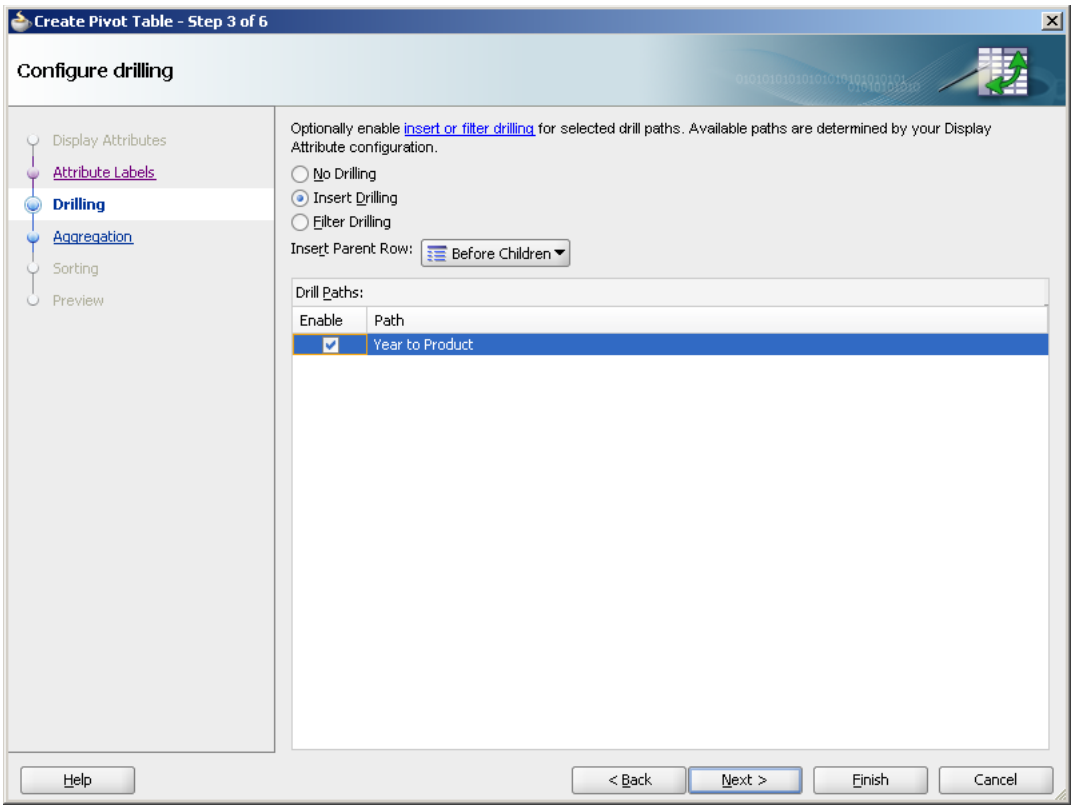
Figure 33–27 Pivot Table with Filter Drilling Enabled



  To enable filter drilling you must select the drill paths to enable. Drill paths are configured based upon the layout of the attributes in the Select Display Attributes page of the wizard.

For example, to enable the insert drilling for the Sales pivot table shown in Figure 33–20, complete the Configure Drilling page of the wizard, as shown in Figure 33–28.

Figure 33–28 Configure Drilling Page of Create Pivot Table Wizard



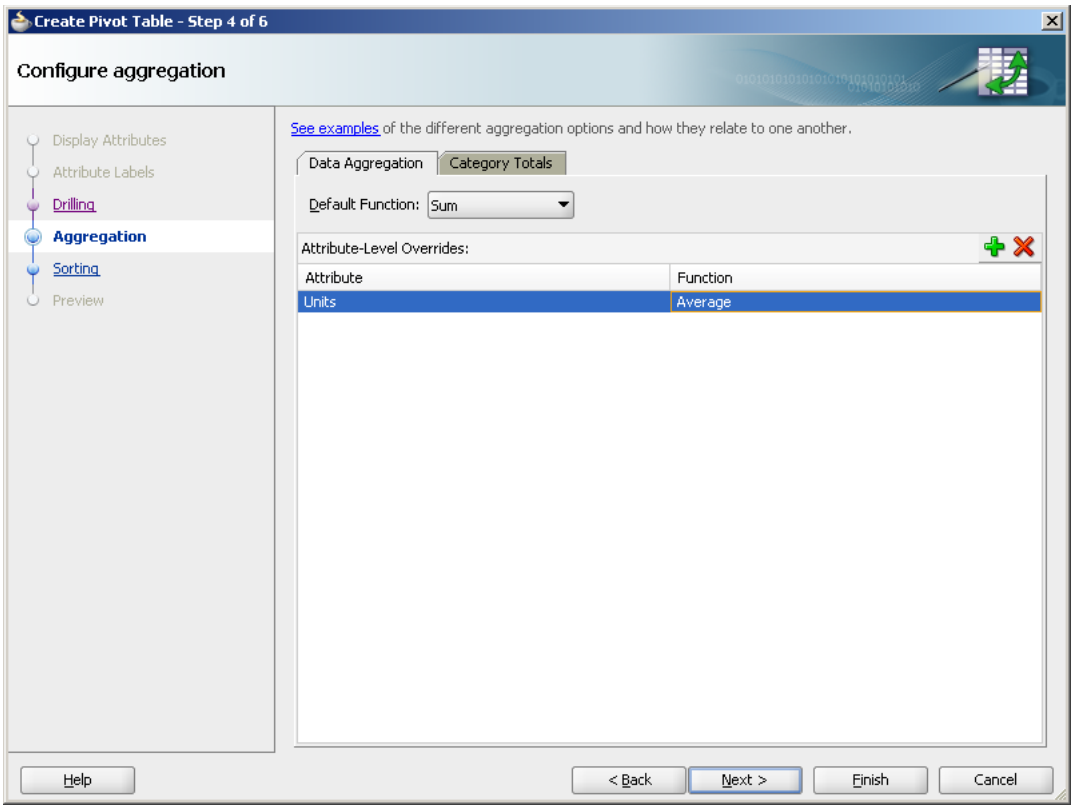
6. If you want to define how data is aggregated in totals and subtotals for the pivot table, use one or both of the Configure Aggregation pages of the Create Pivot Table wizard.

   By default, if the attributes displayed in the pivot table do not uniquely identify each row in the data collection, the data from duplicate rows is aggregated to collapse that data into a single pivot table cell. You can also override the default aggregate type for a particular data item.

   - If you want to specify how data is aggregated in the pivot table, in the Data Aggregation page, do the following:
     - If you want to change the default aggregation method for handling duplicate rows, use the **Default Function** dropdown list to specify the value. Valid values are `Sum`, `Average`, `Count`, `Maximum`, `Minimum`, `Standard Deviation`, `Median`, and `Variance`.
     - If you want to override the default aggregate type for a specific data value, click the **Add** icon to insert a row for the available attributes. Then, in the **Function** column for each attribute, select the mathematical operation that you want to use for the aggregation. Available options are `Sum`, `Average`, `Count`, `Maximum`, `Minimum`, `Standard Deviation`, `Median`, and `Variance`. This attribute is useful only when you have multiple data values (such as Sales and Units) bound to your pivot table.

   For example, to override the default aggregation type for the Units data value in the Sales pivot table shown in Figure 33–20, use the **Add** icon to add the Units attribute and select Average in the **Function** column in the Data Aggregation page, as shown in Figure 33–29.

Figure 33–29 Data Aggregation Page of Create Pivot Table Wizard



- You can also define totals and subtotals for attribute categories added to the column, row, or page edges in the pivot table. In the Categories Totals page, use the **Add** icon to insert each attribute or select **Aggregate All** to add all available attributes, and do the following:
  - In the **Attribute** column, select the attribute that you want to total.
  - In the **Function** column, select the mathematical operation that you want to use for the aggregation. Available options are `Sum`, `Average`, `Count`, `Maximum`, `Minimum`, `Standard Deviation`, `Median`, and `Variance`.
  - In the **Insert Total** column, select the value that indicates where you want the aggregate display to appear relative to the item referenced in the **Attribute** column. Valid values are: `Before`, `After`, or `Replace`.
  - In the **Total Label** column, enter the text that you want to use as a label for the aggregation. You can enter the text directly, specify a text resource from a resource bundle, or use the EL Expression builder to evaluate the label text at runtime.

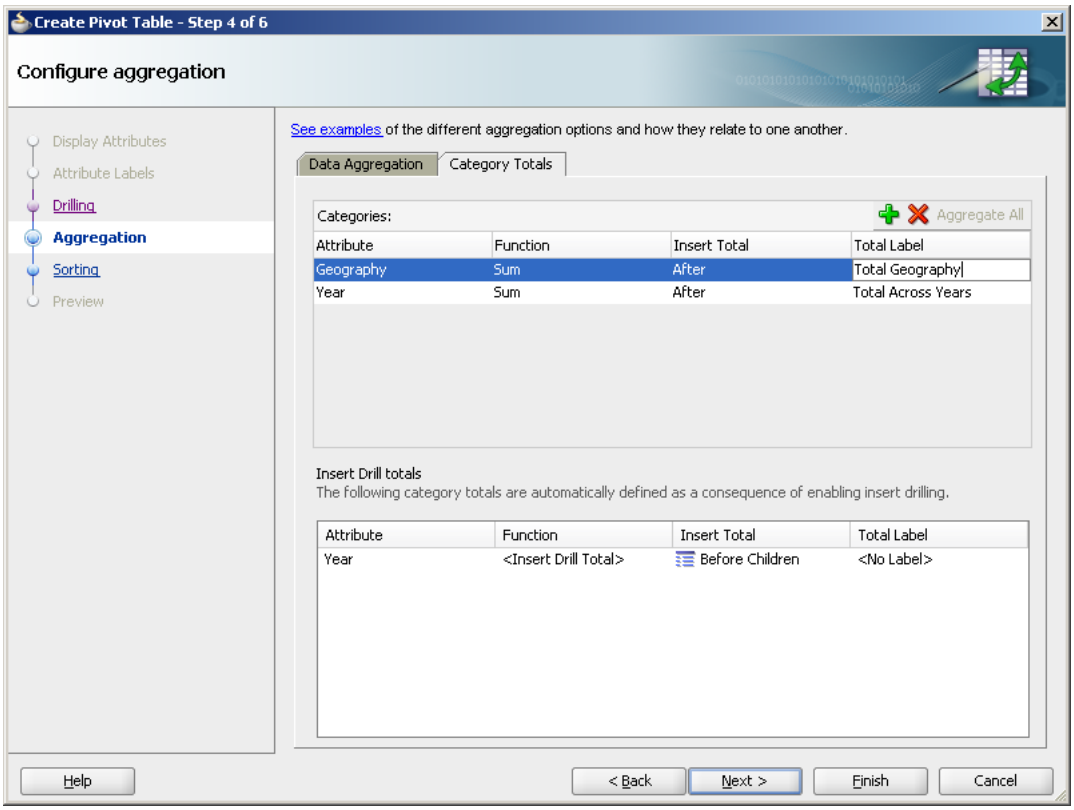
Note: If you configured **Project Properties** > **Resource Bundle** page to **Automatically Synchronize Bundle**, then you can type an alternate label string, and the design time code will create a translatable Text Resource for you.

Note: The read-only **Insert Drill Totals** table displays the category totals automatically defined as a consequence of enabling insert drilling on the pivot table.

For example, to define totals for the Geography and Year data layers in the Sales pivot table shown in Figure 33–20, select `Sum` in the **Function** column and `After` in the **Insert Total** column, and enter text (Total Geography and Total Year) in the **Total Labels** column respectively for each attribute in the Categories Totals page, as shown in Figure 33–30.

In the resulting pivot table at runtime, expanding a particular Year value will automatically preserve the aggregate total computed from its child value based on the layout and configuration of the insert drill option in the previous wizard page.

Figure 33–30 Categories Totals Page of the Create Pivot Table Wizard



7. If you want to configure sorting in the pivot table, use one or both of the Configure Sorting pages in the Create Pivot Table wizard.

   By default, a pivot table initially sorts data based on values in the outer row data layer. You can specify sort order on the data layer of any row, column, or page edge, called a *category sort*. At runtime, when the data layer is pivoted to a different edge, the specified category sort order is honored.

   You cannot specify a category sort of data labels (data values), although you can order the attributes mapped to the data body in the Select Display Attributes page of the wizard. For example, Figure 33–24 shows a pivot table layout with data values for Sales and Units. While you cannot specify a category sort of these

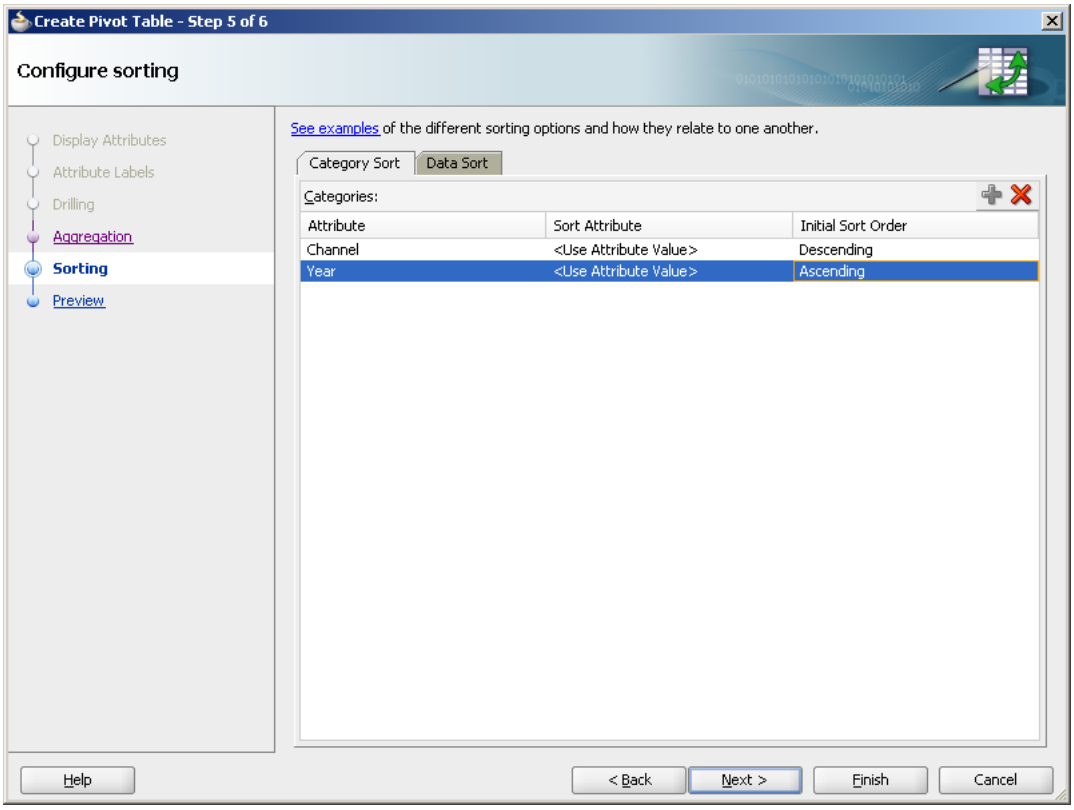
measures, you can specify the order in which the values will appear in the data body of the pivot table at runtime, shown in Figure 33–20.

You can also specify an initial sort order of the data values in the data body when the pivot table is rendered, called a *data sort*.

- To configure sorting by category, in the Category Sort page, use the **Add** icon to add the attribute for each row, column, or page edge you wish to configure, and do the following:
  - In the **Sort Attribute** column, accept the default `Use Attribute Value` to specify an alphabetical sort based on the actual values in the pivot table header, or customize the sort order by specifying an alternate sort order attribute from the dropdown list. For example, if the underlying query included a rank calculation for ranking products by profitability, you could choose to see products ordered by (ProductRank, Descending).
  - In the **Initial Sort Order** column, select the initial direction of the sort. Valid values are `ASCENDING` or `DESCENDING`.

  For example, Figure 33–31 shows the Category Sort page of the wizard configured to display the Channel data layer descending on the column edge and the Year data layer ascending on the row edge.

Figure 33–31 Category Sort Page of Create Pivot Table Wizard



At runtime, the pivot table displays as shown in Figure 33–32.

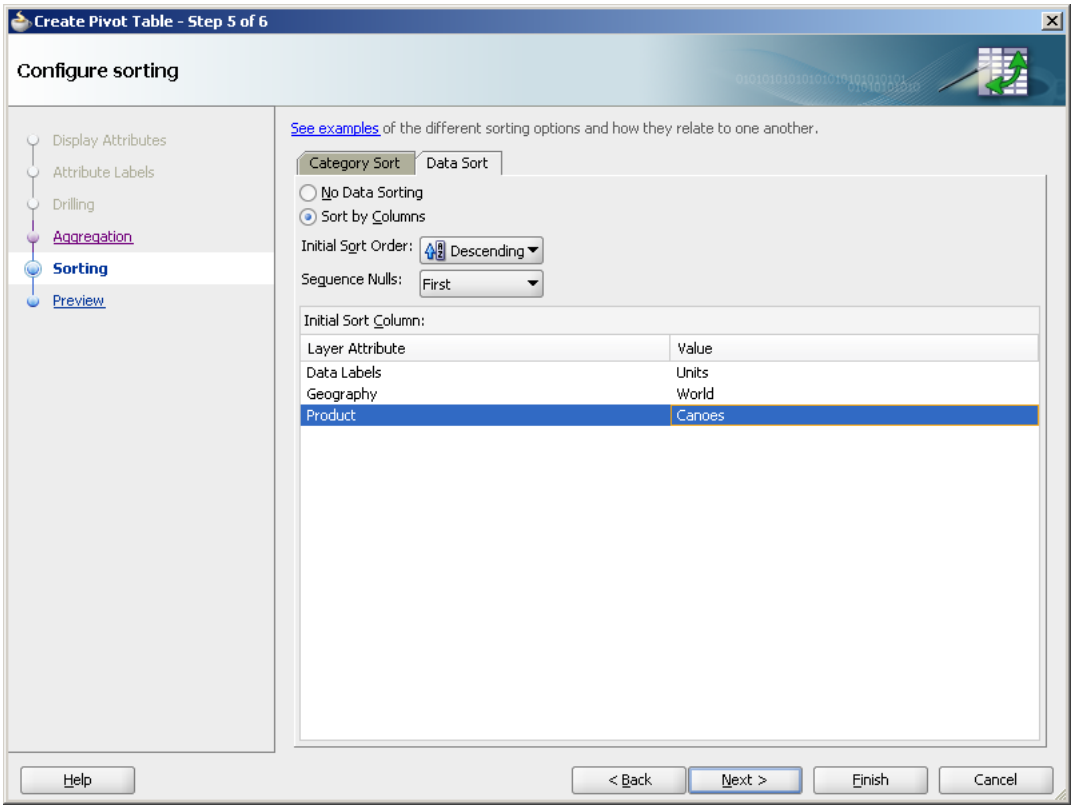
Figure 33–32 Category Sort Example

	Sales		Units	
	Indirect	Direct	Indirect	Direct
2005	12272	9250	131	87
2006	35650	18500	338	169
2007	36600	15900	333	77

- To configure data sorting, in the Data Sort page, do the following:
  - Select **Sort by Columns** to specify an initial sort order of the data when the pivot table is rendered.
  - In the **Initial Sort Order** dropdown list select the initial direction of the sort. Valid values are `ASCENDING` and `DESCENDING`.
  - In the **Sequence Nulls** dropdown list, select `First` if you want the null values to appear at the beginning of a sort and select `Last` if you want the null values to appear at the end of the sort.
  - In the **Initial Sort Column** table, specify a data value in the **Value** column for each data layer displayed in the **Layer Attribute** column.

  For example, Figure 33–33 shows the Data Sort page configured to sort the Channel data layer grouped by Year, based upon Units/World/Canoes data values.

Figure 33–33 Data Sort Page of the Create Pivot Table Wizard



At runtime, the pivot table initially renders as shown in Figure 33–34.

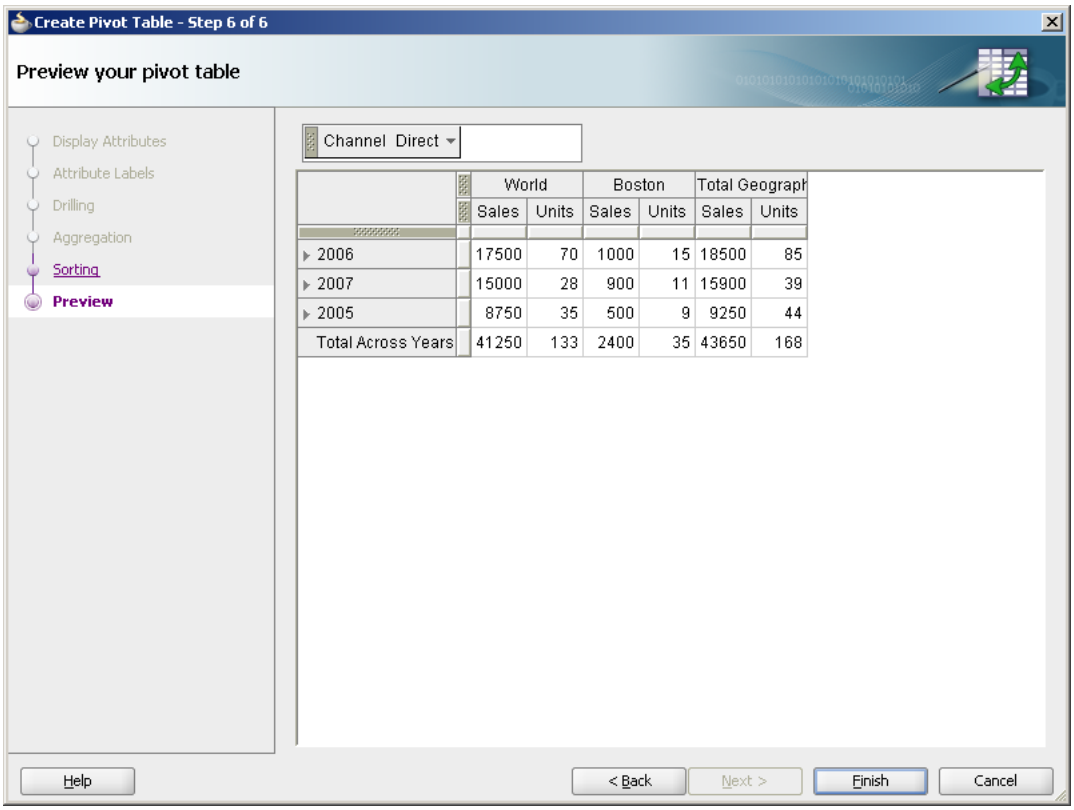
Figure 33–34 Data Sort Example

		Units			
		World		Boston	
		Tents	Canoes	Tents	Canoes
2007	Indirect	200	75	50	8
	Direct	5	50	20	2
2006	Indirect	200	80	50	8
	Direct	100	40	25	4
2005	Direct	50	20	15	2
	Indirect	100	10	20	1

8. In the Preview Your Pivot Table page of the Create Pivot Table wizard, see a live preview of the data that will be displayed in the pivot table. The preview does not require that you compile and run code. If you are not satisfied with the preview, alter the settings in the binding wizard pages and return again to the preview page to verify that your data looks as desired.

   Figure 33–35 shows the Preview Your Pivot Table page of the wizard for the Sales pivot table shown in Figure 33–20.

Figure 33–35 Live Data Preview of Pivot Table



33.4.2 What Happens When You Use the Data Controls Panel to Create a Pivot Table

Dropping a pivot table from the Data Controls panel has the following effect:

- Creates the bindings for the pivot table and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

33.4.2.1 Bindings for Pivot Tables

When you create a pivot table from the Data Controls panel, the page definition file is updated with the bindings. Example 33–10 shows the row set bindings that were generated for the pivot table that displays product sales and units sold within geography by year. The pivot table data map contains the following elements:

- `<columns>`: Defines each column item in the appropriate sequence
- `<rows>`: Defines each row item in the appropriate sequence
- `<pages>`: Defines the items to be included in the pivot filter bar
- `<aggregatedItems>`: Defines the totals and subtotals of items
- `<hierarchies>`: Defines the potential drill paths between two items
- `<sorts>`: Defines category sorts and the initial sort order of pivot table data

The default data aggregation method for duplicate rows is specified in the `<data>` element. For more information about aggregating duplicates, see Section 33.4.3, "What You May Need to Know About Aggregating Attributes in the Pivot Table."

For more information about sorting operations, see Section 33.4.4, "What You May Need to Know About Specifying an Initial Sort for a Pivot Table."

Example 33–10 Binding XML for the ADF Pivot Table

```
<pivotTable IterBinding="ptExampleDataIterator" id="ptExampleData"
            xmlns="http://xmlns.oracle.com/adfm/dvt"
            ChangeEventPolicy="ppr">
  <pivotTableDataMap>
    <columns>
      <item value="Geography"
            itemLabel="${adfBundle['view.ViewControllerBundle'].
            LOCATION"/>
        <data aggregateDuplicates="true" defaultAggregateType="SUM">
          <item value="Sales"/>
          <item value="Units" aggregateType="AVERAGE"/>
  </data>
    </columns>
    <rows>
      <item value="Year"/>
    </rows>
    <pages>
      <item value="Channel"/>
    </pages>
    <aggregatedItems>
      <item aggregateLocation="AFTER" aggregateType="SUM" value="Geography"
            aggregateLabel="${adfBundle['view.ViewControllerBundle'].
            TOTAL_GEOGRAPHY"/>
      <item aggregateLocation="AFTER" aggregateType="SUM" value="Year"
            aggregateLabel="${adfBundle['view.ViewControllerBundle'].
            TOTAL_ACROSS_YEARS}"/>
    </aggregatedItems>
    <drills type="INSERT"/>
    <hierarchies>
      <item value="Year" location="BEFORE">
         <child value="Product"
                label="${adfBundle['view.ViewControllerBundle'].PRODUCT"/>
      </item>
    </hierarchies>
    <sorts>
```

```
      <categorySort item="Channel" direction="DESCENDING"/>
      <categorySort item="Year" direction="ASCENDING"/>
      <qdrSliceSort direction="DESCENDING" edge="rows" grouped="true"
                    nullsFirst="true">
        <item name="Geography" value="World"/>
      </qdrSliceSort>
    </sorts>
  </pivotTableDataMap>
</pivotTable>
```

33.4.2.2 Code on the JSF Page for a Pivot Table and Pivot Filter Bar

When the pivot table is created using the Data Controls panel, the necessary code is added to the page. Example 33–11 shows the code generated on the JSF page for the sales pivot table and associated pivot filter bar.

Example 33–11 XML Code on a JSF Page for the Pivot Table and Pivot Filter Bar

```
<dvt:pivotFilterBar id="pt1pivotFilterBar"
                    value="#{bindings.ptExampleData.pivotFilterBarModel}"
                    modelName="pt1Model"/>
<dvt:pivotTable id="pt1" value="#{bindings.ptExampleData.pivotTableModel}"
                         modelName="pivotTableModel"
                         var="cellData" varStatus="cellStatus">
  <dvt:headerCell>
    <af:switcher facetName="#{cellData.layerName}" defaultFacet="Default" id="s1">
      <f:facet name="DataLayer">
        <af:outputText value="#{cellData.label}" id="ot1"/>
      </f:facet>
      <f:facet name="Year">
        <af:outputText value="#{cellData.dataValue}" id="ot2">
          <af:convertNumber groupingUsed="false"
                            pattern="#{bindings.ptExampleData.
                            hints.Gnp.format}"/>
        </af:outputText>
      </f:facet>
      <f:facet name="Product">
        <af:outputText value="#{cellData.dataValue}" id="ot3"/>
      </f:facet>
      <f:facet name="Default">
        <af:outputText value="#{cellData.dataValue}" id="ot4"/>
      </f:facet>
    </af:switcher>
  </dvt:headerCell>
  <dvt:dataCell>
    <af:switcher facetName="#{cellStatus.members.DataLayer.value}"
                 defaultFacet="Default" id="s2">
      <f:facet name="Sales">
        <af:outputText value="#{cellData.dataValue}" id="ot5">
          <af:convertNumber groupingUsed="false"
                            pattern="#{bindings.ptExampleData.
                            hints.Population.format}"/>
        </af:outputText>
      </f:facet>
      <f:facet name="Default">
        <af:outputText value="#{cellData.dataValue}" id="ot6"/>
      </f:facet>
    </af:switcher>
  </dvt:dataCell>
```

```
</dvt:pivotTable>
```

33.4.3 What You May Need to Know About Aggregating Attributes in the Pivot Table

If the attributes that you choose to display in your pivot table do not uniquely identify each row in your data collection, then you can aggregate the data from duplicate rows to collapse that data into a single pivot table cell.

For example, if the rows in the data collection shown in Figure 33–20 also contained a store identification, then the data rows from all stores in a given combination of Product, Channel, and Geography would have to be collapsed into a single cell in the pivot table.

The pivot table has the following optional data binding attributes available for controlling the calculation of duplicate data rows:

- `aggregateDuplicates`: Boolean property of the `<data>` element that determines whether special processing is enabled at binding runtime to aggregate data values in duplicate rows. If this attribute is not specified, then `false` is assumed.
- `defaultAggregateType`: String property of the `<data>` element that specifies a default aggregation method for handling duplicates. Valid values are `SUM`, `AVERAGE`, `COUNT`, `MIN`, `MAX`, `STDDEV`, `MEDIAN`, `VARIANCE`. If `aggregateDuplicates` is `true` and `defaultAggregateType` is unspecified, then `SUM` is assumed.
- `aggregateType`: String property of an `<item>` element that enables you to override the default aggregate type for a particular data item. This attribute is useful only when you have multiple data values (such as Sales and Units) bound to your pivot table.

33.4.3.1 Default Aggregation of Duplicate Data Rows

By default, the pivot table uses the SUM operation to aggregate the data values of duplicate data rows in a data collection to produce a single cell value in the pivot table. This means that the `aggregateDuplicates` attribute is set to `true` and the `defaultAggregateType` is assumed to be `SUM`.

The `<data>` element shown in Example 33–10 is an example of such default aggregation.

33.4.3.2 Custom Aggregation of Duplicate Rows

If you want the pivot table to use a different mathematical operation to aggregate the data values of duplicate rows, then you set the `defaultAggregateType` to the desired operation.

Example 33–12 shows a data element with the `defaultAggregateType` set to `SUM`. This operation would be appropriate if you want to see the total of sales from all stores for each unique combination of Product, Channel, and State.

Example 33–12 Binding XML for Custom Aggregation of Duplicate Rows

```
<pivotTable IterBinding="SalesPivotTable1Iterator" id="SalesPivotTable11"
            xmlns="http://xmlns.oracle.com/adfm/dvt">
  <pivotTableDataMap>
    <columns>
      <data aggregateDuplicates="true" defaultAggregateType="SUM">
        <item value="Sales"/>
```

```
      </data>
      <item value="Geography"/>
    </columns>
    <rows>
      <item value="Channel"/>
      <item value="Product"/>
    </rows>
    <aggregatedItems>
      <item aggregateLocation="After" aggregateType="AVERAGE"
            value="Product" aggregateLabel="Average"/>
    </aggregatedItems>
  </pivotTableDataMap>
</pivotTable>
```

If you have a pivot table with multiple data values (such as sales and the average size of a store in square feet) and you want to sum the sales data values in duplicate rows, but you want to average the square feet data values, then do the following:

- On the `<data>` element, set the `defaultAggregateType` to `SUM`.
- On the `<item>` element for the square feet attribute, set the `aggregateType` to `AVERAGE`.

Example 33–13 shows the `<columns>` elements wrapped by a `PivotTableDataMap` element. The `<data>` element contains the default attributes for aggregation. These apply to all data items that do not have a specific custom `aggregateType` attribute specified.

Example 33–13 Data and Item Elements for Multiple Custom Aggregations

```
<columns>
   <data aggregateDuplicates="true" defaultAggregateType="SUM">
     <item value="Sales" label="Total Sales"/>
     <item value="StoreSqFeet" label="Avg Sq Feet" aggregateType="AVERAGE"/>
   </data>
   <item value="State"/>
</columns>
```

33.4.4 What You May Need to Know About Specifying an Initial Sort for a Pivot Table

By default, a pivot table initially sorts data based on values in the outer row data layer. You can specify sort order on the data layer of any row, column, or page item, called a *category sort*. At runtime, when the data layer is pivoted to a different edge, the specified category sort order is honored. Insert a `categorySort` element inside the `sorts` element and set values for the attributes as described in Table 33–2.

Table 33–2 Attribute Values for categorySort Element

Attribute	Description
`item`	Specify the column, row, or page item for which you are setting the category sort. A value for this attribute is required.
`direction`	Specify the initial direction of the sort. Valid values are `ASCENDING` and `DESCENDING`. A value for this attribute is required.

You can also specify the initial sort order of the data values in the data body when the pivot table is rendered, called a *data sort*. You can change the default behavior by inserting a `sorts` element inside the `pivotTableDataMap` element of a pivot table

binding in the page definition file. Insert a `qdrSliceSort` element inside the `sorts` element and set values for the attributes as described in Table 33–3.

Table 33–3 Attribute Values for qdrSliceSort Element

Attribute	Description
`direction`	Specify the initial direction of the sort. Valid values are `ASCENDING` and `DESCENDING`. A value for this attribute is required.
`edge`	Specify `columns` or `rows` to determine which edge sorts data. A value for this attribute is required.
`grouped`	Specify `true` if you want to sort slices within their parent or `false` if you want to sort across the entire edge. A value for this attribute is optional. The default value is `false`.
`nullsFirst`	Specify `true` if you want null values to appear at the beginning of a sort and `false` if you want null values to appear at the end of a sort. The default value is `false`. A value for this attribute is optional.

Insert one or more `item` tags inside the `qdrSliceSort` tag. An item tag specifies the slice on the opposite edge from which the values to be sorted should be obtained. For example, if sorting rows based upon the data, then you must specify an item tag for each layer on the column edge. Set values for the attributes as described in Table 33–4.

Table 33–4 Attribute Values for item Tag

Attribute	Description
`name`	Specify the name of the layer to sort on. Typically, this is the column name in the row set. Specify `DataLayer` to identify the layer that contains the data columns in a row set (for example, Sales, Costs, and so on).
`value`	Specify the value of the specified layer on the desired slice.

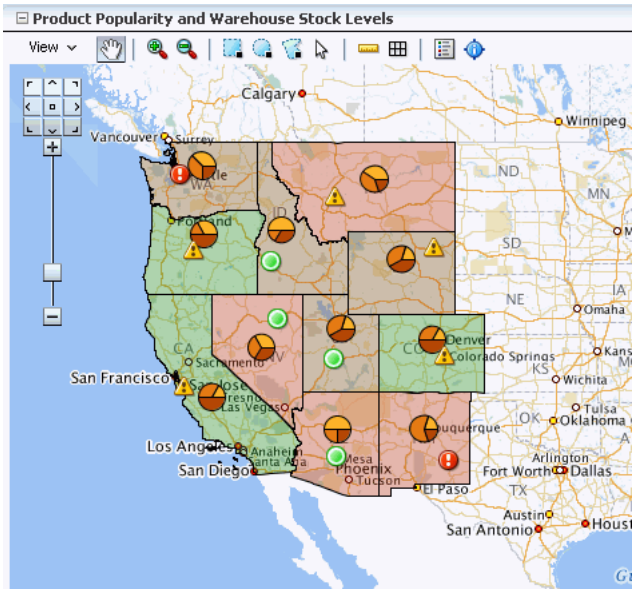
33.5 Creating Databound Geographic Maps

An ADF geographic map is an ADF Data Visualization component that provides the functionality of Oracle Spatial within Oracle ADF. This component allows users to represent business data on a geographic map and to superimpose multiple layers of information (known as *themes*) on a single map. These layers can be represented as any of the following themes: bar graph, pie graph, color, point, and predefined theme.

Figure 33–36 shows a geographic map component that uses a base map for a region in the United States with the following themes:

- Color theme: For the selected product, this theme colors states based on product popularity. The colors range from green (which represents the highest popularity for that product) to red (which represents the lowest popularity for that product).
- Pie graph theme: This theme displays a pie graph in each state to indicate the popular product categories in that state. In this example, the pie graph shows the following product categories as pie slices: Media, Office, and Electronics.
- Point theme: This theme identifies warehouses as points. For each point, it displays an icon to indicate the inventory level at that warehouse for the selected product. A separate icon is displayed for each of the following ranges of inventory: low inventory, medium inventory, and high inventory.

Figure 33–36 Geographic Map with Color Theme, Pie Graph Theme, and Point Theme for a Product



A geographic map component differs from other ADF Data Visualization components as you do not need to put multiple maps on a page to display multiple sets of data. This contrasts to components such as graphs where you can put multiple graphs on a page. Instead, you show how multiple sets of data relate to each other spatially or, for a specific point, you display different attributes layered in separate themes.

The geographic map component itself is not bound to data. However, each map *theme* has its own data bindings.

A base map forms the background on which the ADF geographic map component layers the themes that developers create.

In Oracle Spatial, administrators create base maps that consist of one or more themes. The administrator controls the visibility of the base map themes. When you zoom in and out on a base map, various base map themes are hidden or displayed. At the ADF geographic map component level, you cannot use zoom factor to control the display of the themes created by the administrator on the base map.

When you overlay themes on the ADF geographic map, you can control the visibility of your themes by setting the `maxZoom` and `minZoom` properties of the components related to these themes. At runtime, you can also hide or display your custom themes by using the View menu of the Map toolbar or by using other ADF components that you create on the page.

For information about customizing a geographic map after data-binding is completed, see the "Using ADF Geographic Map Components" chapter in *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

33.5.1 How to Create a Geographic Map with a Point Theme

To create a geographic map, you first configure the map (that is, select a base map and provide URLs for processing) and then bind a theme of the map to a data collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel for the theme that you want to create.

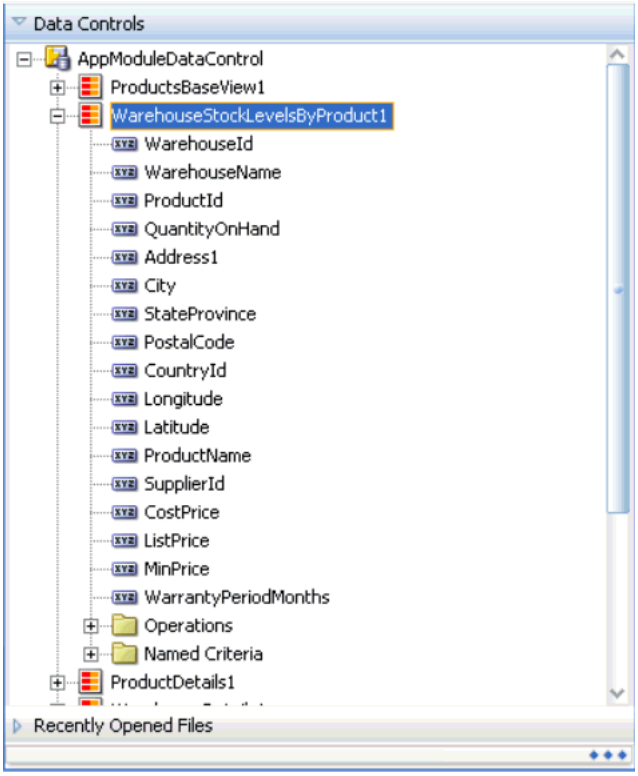
When you create a map point theme, you have the option of customizing the style of the points that appear in the map. For each different point style, you can use a `mapPointStyleItem` tag.

To create a geographic map with a databound point theme:

1. From the Data Controls panel, select a collection.

   Figure 33–37 shows an example where you could select the `WarehouseStockLevelsByProduct1` collection in the Data Controls panel to create a geographic map with a point theme that displays an image to represent the quantity on hand for each warehouse point.

Figure 33–37 Data Collection for Warehouse Stock Levels



2. Drag the collection onto a JSF page and, from the context menu, choose **Geographic Map** > **Map and Point Theme**.
3. If you have not yet configured a map on the page, then in the ensuing Create Geographic Map dialog, click the **New** icon to display the Create Geographic Map Configuration dialog and do the following:
   a. In the **Id** field enter the unique identifier for the map configuration.
   b. In the **MapViewer URL** field enter the URL for the Oracle MapViewer service.
   c. In the **Geocoder URL** field select the URL for the Geocoder Web service that converts street addresses into latitude and longitude coordinates for mapping.

   Note: The Geocoder URL is needed only if you do *not* already have longitude and latitude information for addresses.

d. Click **OK** to dismiss the dialog and return to the Create Geographic Map dialog.

4. In the Maps page, you select the base map for the geographic map component and provide other settings to use with the map by doing the following:

   a. From the **Data Source** list select the collection of maps from which you will choose a base map.

   b. From the **Base Map** list select the map that will serve as the background for the geographic map component.

   c. To specify values for the **StartingX** field and the **StartingY** field click on the image of the map to center it within the Preview window.

      You can use the arrows in the map navigator in the upper left-hand corner to move the map in the appropriate direction.

   d. Optionally use the sliding arrow in the Preview window to adjust the zoom factor of the map.

   e. Click **OK** to dismiss the dialog and to display the Create Point Map Theme dialog.

5. In the **Theme Id** field enter the unique identifier for the point map theme.

6. In the **Location** section, specify whether the point location is to be specified by **Address** or by a pair of *x* and *y* coordinates (Longitude and Latitude).

   The choice you select for location will determine which controls appear in the Location section.

   Tip: Using *x* and *y* coordinates is a more efficient way to present data on the map rather than using the Address controls, which must be converted by a Geocoder to *x* and *y* coordinates. If the data collection has more than 100 rows, use *x* and *y* coordinates for better performance.

7. For the *x* and *y* point location, select attributes from the data collection that represents the following items:

   - X (Longitude): The horizontal location of the point on the map.
   - Y (Latitude): The vertical location of the point on the map.
   - Label: The labels for the points in the top section of the information window, which is displayed when you click a point.

8. In the Point Data section, provide the following information that identifies the data associated with the point, its label, and optionally the style for the point:

   - In the **Data** field, select the data column that is associated with the point, such as `QuantityOnHand`.
   - In the **Label** field, enter the text that will appear in the information window before the data value when you click a point. You can enter a text resource to use for the label. The text resource can be a translatable string from a resource bundle or an EL expression executed at runtime. Use the dropdown list to open a Select Text Resource or Expression Builder dialog. If you need help, press **F1** or click **Help**.
   - Optionally, in the **Category** field, select a data column to use for finding the appropriate style for a point. If you select a value for Category, that value is stored in the binding for this point theme and then matched against the

`itemValue` attribute of the `mapPointStyleItem` tags that you create for this point theme.

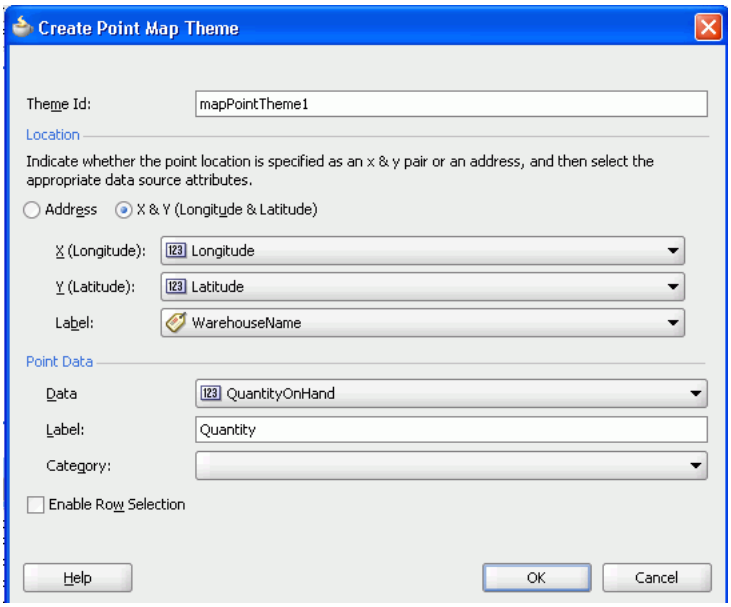
Note: If your data does not have a column that you want to use as a category for finding the style of a point, you can also use `mapPointStyleItem` tags to define styles related to data ranges (such as high, medium, and low) that are matched to the values in the column that you select in the Data field. For more information, see Section 33.5.2, "How to Create Point Style Items for a Point Theme".

---

9. Select the **Enable Row Selection** Select if you want to enable master-detail relationships. This is useful when the data collection for the geographic map is a master in a master-detail relationship with a detail view that is displayed in another UI component on the page. Selecting this options enables both `selectionListenter` and `clickListener` attributes.
10. Click **OK**.

Figure 33–38 shows the Create Point Map Theme dialog for a geographic map with a point theme that displays an image representing quantity on hand for each warehouse point.

Figure 33–38 Create Point Map Theme Dialog for Warehouse Inventory Levels



33.5.2 How to Create Point Style Items for a Point Theme

There are a variety of options available for creating point style items for use in a given map point theme. These are:

- A single image for all data points
- Separate images for each data point category
- Images that represent low, medium, and high data value ranges

After you create the data binding for a map point theme, you have the option of selecting a single built-in image that should be used for all points in that map theme.

In the Property Inspector, you can make this selection in the `builtInImage` attribute of the `mapPointTheme` tag. The default value for this attribute is an orange ball.

Alternatively, if you specify a value for Category in the Create Point Map Theme dialog, then you should also create a set of point style items to determine a separate image that represents data points in each category. In this case, you do not use the minimum and maximum values in the point style item tags. Instead, you set the `itemValue` attribute of point style item tags to a value that matches entries in the data column that you specified for Category.

In a point theme for a geographic map, if you do not specify a value for Category, you can still use the `mapPointStyleItem` child tags of the `mapPointTheme` tag to specify ranges of values (such as low, medium, and high) and the images that are to represent these ranges. If you do this, then each point will be represented by an image that identifies the range in which the data value for that point falls.

The following procedure assumes that you have already created a geographic map with a point theme.

To add point style items to a map point theme to represent low, medium, and high data value ranges:

1. In the Structure window, right-click the **dvt:mapPointTheme** tag and choose **Insert inside the dvt:mapPointTheme** > **Point Style Item**.
2. In the Point Style Item Property Inspector, set values as described Table 33–5, " Properties for Point Style Item".

Table 33–5 Properties for Point Style Item

For this property	Set this value
`Id`	Specify a unique ID for the point style item.
`MinValue`	Specify the minimum value in a data range that you define.
`MaxValue`	Specify the maximum value in a data range that you define.
`ShortLabel`	Specify text to appear when a user hovers over the point item. For example, if you define a point item for low inventory, then enter `Low Inventory` as the value for this property.
`ImageURL`	Specify the URL to the image file or select it from the dropdown list. At runtime, the image you specify appears on the map to represent the data range identified by the **MinValue** and **MaxValue** properties. Alternatively, you can select one of a number of predefined images referenced by the **BuiltInImage** dropdown list that appears in the **Other** section.
`HoverImageURL`	Specify the URL to the image file or select it from the dropdown list. At runtime, the image you specify appears when a user hovers over the point item.
`SelectedImageURL`	Specify the URL to the image file or select it from the dropdown list. At runtime, the image you specify appears when a user selects the point item.

3. If you defined a data value range for a low data value range in Steps 1 and 2, then repeat Steps 1 and 2 to define medium and high data value ranges with appropriate values.

Note: The use of `mapPointStyleItem` child tags to customize the style of points is a declarative approach that lets you provide custom point images. For information about using a callback to provide not only custom images but also custom HTML, see Section 33.5.4, "What You May Need to Know About Adding Custom Point Style Items to a Map Point Theme".

33.5.3 What Happens When You Create a Geographic Map with a Point Theme

Dropping a geographic map and a point theme (which in this case would be the initial theme added to the map) from the Data Controls panel has the following effect:

- Creates the bindings for the point theme and adds the bindings to the page definition file
- Adds the necessary tags to the JSF page for the geographic map component
- Adds the necessary point theme child tags within the geographic map tag to the JSF page

33.5.3.1 Binding XML for a Point Theme

Example 33–14 shows the row set bindings that were generated for the point theme of the geographic map.

Example 33–14 Point Theme Binding XML

```
<mapTheme IterBinding="WarehouseStockLevelsByProduct1Iterator"
          id="WarehouseStockLevelsByProduct1"
          xmlns="http://xmlns.oracle.com/adfm/dvt">
  <mapThemeDataMap convert="false" mapThemeType="point">
    <data>
      <item value="QuantityOnHand"
            label="${adfBundle['view.ViewControllerBundle'].PRODUCT_QUANTITY"/>
    </data>
      <item type="lat_long" latitude="Latitude"
            longitude="Longitude"
            label="${adfBundle['view.ViewControllerBundle'].WAREHOUSE_NAME"/>
  </mapthemeDataMap>
</mapTheme>
```

33.5.3.2 XML Code on the JSF Page for a Geographic Map and Point Theme

Example 33–15 shows the XML code generated on the JSF page for the geographic map and its point theme. Notice the code for the three kinds of point style settings based on data value.

The initial point style setting (`ps0`) applies to values that do not exceed 500. This point style displays an image for very low inventory and provides corresponding tooltip information.

The second point style setting (`ps1`) applies to values between 500 and 1000. This point style displays an image for low inventory and provides corresponding tooltip information.

The final point style setting (`ps2`) applies to values between 1000 and 1600. This point style displays an image for high inventory and provides corresponding tooltip information.

Example 33–15 Geographic Map and Point Theme XML Code on the JSF Page

```
<dvt:map id="map1"
         mapServerConfigId="mapConfig1"
         inlineStyle="width:850px;height:490px"
         startingX="-96.0"
         baseMapName="ELOCATION_MERCATOR.WORLD_MAP"
         startingY="37.0" zoomBarPosition="WEST"
         showScaleBar="false"
  <dvt:mapPointTheme id="mapPointTheme1"
                     shortLabel="#{viewcontrollerBundle.WAREHOUSE_STOCK_LEVELS}"
                     value="#{bindings.WarehouseStockLevelsByProduct1.geoMapModel}"
                     rendered="#{AppState.showPointTheme}">
    <dvt:mapPointStyleItem id="ps0"
                           minValue="0"
                           maxValue="500"
                           imageURL="/images/low.png"
                           selectedImageURL="/images/lowSelected.png"
                        shortLabel="#{viewcontrollerBundle.VERY_LOW_
                            INVENTORY}"/>
    <dvt:mapPointStyleItem id="ps1"
                           minValue="500"
                           maxValue="1000"
                           imageURL="/images/medium.png"
                           selectedImageURL="/images/mediumSelected.png"
                           shortLabel="#{viewcontrollerBundle.LOW_INVENTORY}"/>
    <dvt:mapPointStyleItem id="ps2"
                           minValue="1000"
                           maxValue="1600"
                           imageURL="/images/regularGreen.png"
                           selectedImageURL="/images/regularGreenSelected.png"
                           shortLabel="#{viewcontrollerBundle.HIGH_INVENTORY}"/>
  </dvt:mapPointTheme>
</dvt:map>
```

33.5.4 What You May Need to Know About Adding Custom Point Style Items to a Map Point Theme

If you want to provide custom HTML as well as custom images for map points, then you can use the `customPointCallback` attribute of the `dvt:mapPointTheme` tag to accomplish this customization.

> **Important:** If you set the `customPointCallback` attribute for a map point theme, the map ignores any `dvt:mapPointStyleItem` child tags because the callback overrides these tags.

To use a callback to customize the style of map points:

1. Write a method in Java to perform the desired point customization.
2. Store this method in a managed bean for the map.

   For more information about managed beans, see the "Creating and Using Managed Beans" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.
3. After you finish data-binding the map point theme, use the Property Inspector to specify a reference to the managed bean method in the `customPointCallback` attribute of the `dvt:mapPointTheme` tag.

For example, if the managed bean is named `MapSampleBean` and the method is named `setCustomPointStyle`, then the reference becomes `#{mapSampleBean.CustomPointStyle}`.

33.5.5 How to Add a Databound Color Theme to a Geographic Map

When you create a geographic map, you can choose to create themes (point, color, and graph) in any sequence that you wish.

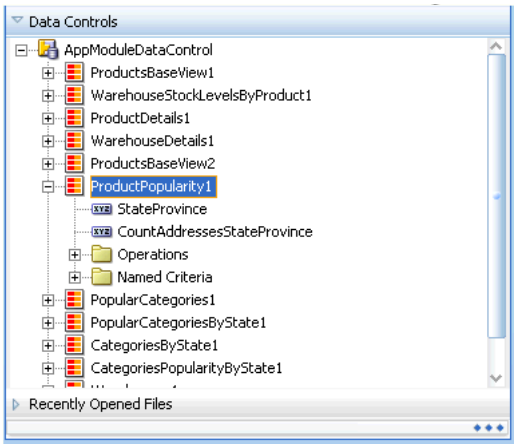
The following procedure assumes that a geographic map has already been configured and, therefore, the map component does not display the dialog for configuring the map. Instead, only the dialog for creating the color theme appears.

To add a databound color theme to a geographic map:

1. From the Data Controls panel, select a collection.

   Figure 33–39 shows an example where you could select the `ProductPopularity1` collection in the Data Controls panel to create a color map theme that shows product popularity by the color of regions (for example, states).

Figure 33–39 Data Collection for Product Popularity by State

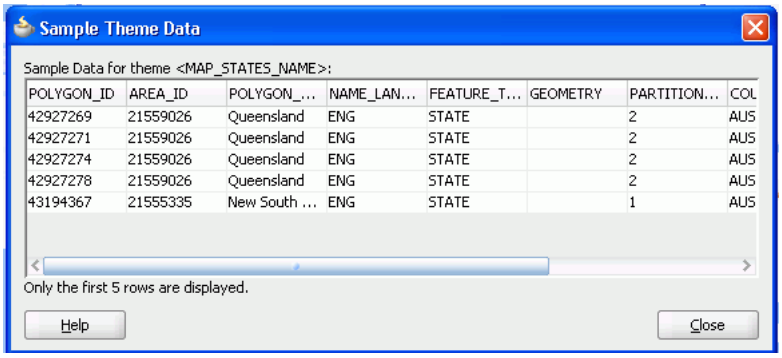


2. Drag the collection onto a JSF page which already contains a geographic map component and, from the context menu, choose **Geographic Map** > **Color Theme**.
3. In the ensuing Create Color Map Theme dialog, enter a unique identifier for the map theme in the **Id** field.
4. In the **Base Map Theme** section, identify the base map color theme to use for the geographic map by doing the following:
   a. In the **Name** field, select the name of the base map theme.
   b. For **Location**, select the location column in the data collection that should be matched to the location column in the base map theme that you selected.
   c. Optionally, click **View Sample Theme Data** to display the Sample Theme Data dialog, in which you can examine the first several rows of the actual data so that you can identify the appropriate location column.

      For example, if you want to view the data for a region that consists of states in the United States map, you might select MAP_STATES_NAME as shown in Figure 33–40.

Note: It is possible for an administrator of Oracle Spatial to disable the display of sample data. If this button is not available, then consult the administrator for guidance.

Figure 33–40 Sample Theme Data for Regions or States



5. In the **Appearance** section, specify the look of the color theme as follows:

   a. In **Data Bucket Count**, enter the number of groups for the data in this geographic map. Each group is coded with a color. After specifying this number, you can provide colors for the minimum value and the maximum value. The colors for the other values are chosen automatically using an RGB algorithm.

   b. In **Minimum Value Color**, select the color for the minimum value.

   c. In **Maximum Value Color**, select the color for the maximum value.

   Note: If you want to specify an exact color for each data bucket, see Section 33.5.7, "What You May Need to Know About Customizing Colors in a Map Color Theme".

6. In the **Data** section, provide the following information about the data in the collection:

   a. For **Location**, select the column in the data collection that should match the values in the location column that you selected from the base map theme.

   b. For **Location Label**, select the column in the data collection that contains the labels associated with the values in the location column. These labels are shown in the information window that is displayed when you click or hover over a color.

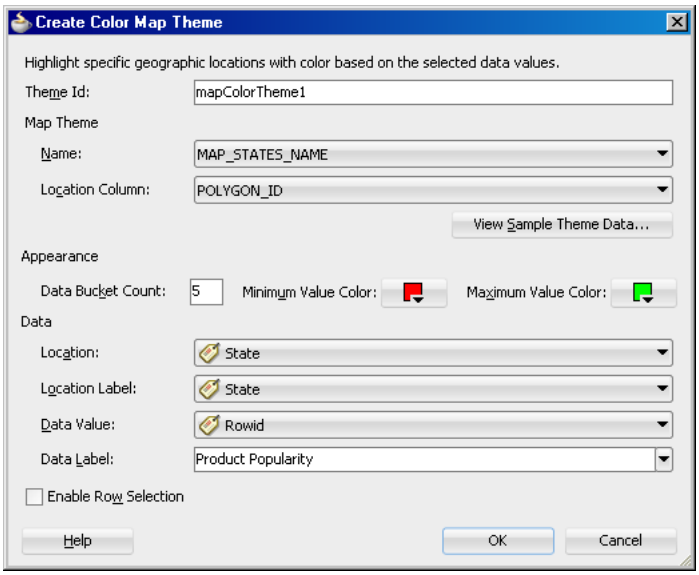
   c. For **Data Label**, enter the label to use for describing the data in the information window and the tooltip that is displayed when you click or hover over a color. For example, the information window might include a label before the data value, such as **Product Popularity**. You can also enter a text resource to use for the data label. The text resource can be a translatable string from a resource bundle or an EL expression executed at runtime for a dynamic label. Use the dropdown list to open a Select Text Resource or Expression Builder dialog. If you need help, press **F1** or click **Help**.

7. Use **Enable Row Selection** only if you want to enable master-detail relationships. This is useful when the data collection for the map theme is a master in a

master-detail relationship with a detail view that is displayed in another UI component on the page.

Figure 33–41 shows the Create Color Map Theme dialog for the product popularity by state color theme.

Figure 33–41 Create Color Map Theme for Product Popularity By State



33.5.6 What Happens When You Add a Color Theme to a Geographic Map

Dropping a color theme from the Data Controls panel to an existing geographic map has the following effect:

- Creates the bindings for the color theme and adds the bindings to the page definition file
- Adds the necessary color theme child tags within the geographic map tag to the JSF page

33.5.6.1 Binding XML for a Color Theme

Example 33–16 shows the row set bindings that were generated for the color theme of the geographic map.

Example 33–16 Color Theme Binding XML

```
<mapTheme IterBinding="ProductPopularity1Iterator" id="ProductPopularity1"
          xmlns="http://xmlns.oracle.com/adfm/dvt">
  <mapThemeDataMap mapThemeType="color">
    <data>
      <item value="CountAddressesStateProvince"
            label="${adfBundle['view.ViewControllerBundle'].POPULARITY"/
    </data>
      <item type="location" value="StateProvince"
            label="${adfBundle['view.ViewControllerBundle'].STATE_PROVINCE"/
   </mapThemeDataMap>
</mapTheme>
```

33.5.6.2 XML Code on the JSF Page for a Color Theme

Example 33–17 shows the XML code generated on the JSF page for a color theme that represents product popularity in different states on the United States map.

Example 33–17 Color Theme XML Code on the JSF Page

```
<dvt:mapColorTheme id="mapColorTheme1"
     themeName="MAP_STATES_NAME"
     shortLabel="#{viewcontrollerBundle.PRODUCT_POPULARITY}"
     value="#{bindings.ProductPopularity1.geoMapModel}"
     locationColumn="POLYGON_NAME"
     minColor="#ff0000"
     maxColor="#008200"
     bucketCount="5"/>
</dvt:mapColorTheme>
```

33.5.7 What You May Need to Know About Customizing Colors in a Map Color Theme

While you are data-binding a map color theme, you can specify only a minimum color and a maximum color for the data buckets. The map uses an algorithm to determine the colors of the buckets between the minimum and maximum. However, after the data-binding is finished, you have the option of specifying the exact color to be used for each data bucket.

In the Object Inspector, for the `dvt:mapColorTheme` tag you can use the `colorList` attribute to specify the color for each bucket. You can either bind a color array to this attribute or you can specify a string of colors using a semicolon separator.

For example, if the value of this attributes is set to: `#ff0000;#00ff00;#0000ff`, then the color of the first bucket is red, the second bucket is green, and the third bucket is blue.

33.5.8 How to Add a Databound Pie Graph Theme to a Geographic Map

When you create a geographic map, you can choose to create themes (point, color, and graph) in any sequence that you wish. However, only one graph theme (pie or bar) can be visible at a time on the ADF geographic map component.

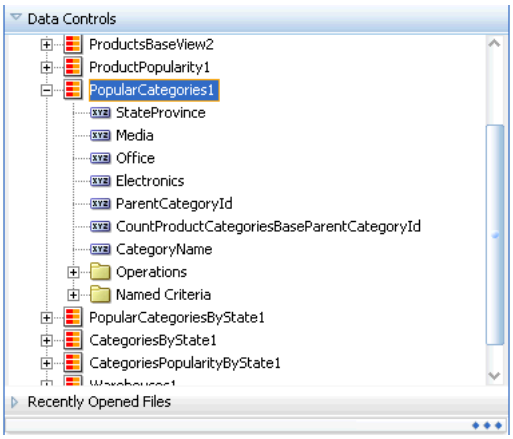
The following procedure assumes that a geographic map has already been configured and, therefore, the map component does not display the dialog for configuring the map. Instead, only the dialog for creating the pie graph theme appears.

To add a databound pie graph theme to a geographic map:

1. From the Data Controls panel, select a collection.

   Figure 33–42 shows an example where you could select the `PopularCategories1` collection to create a pie bar theme in an existing geographic map component to represent the popular product categories within a state.

Figure 33–42 Data Collection for Popular Product Categories by State

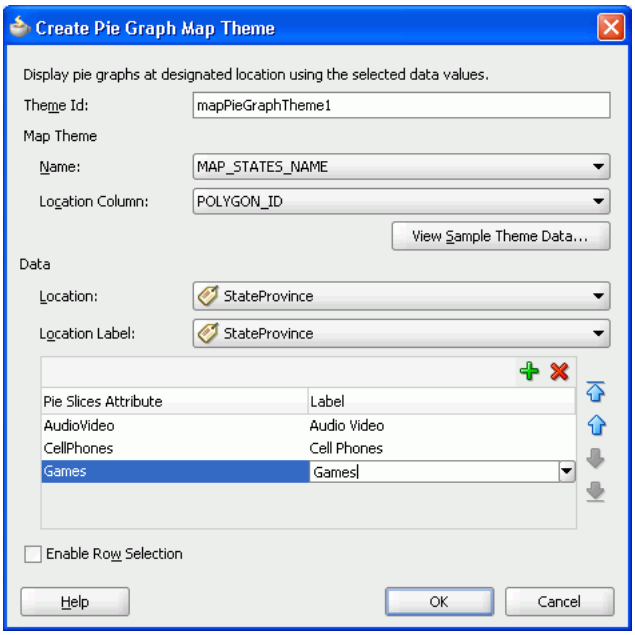


2. Drag the collection onto a JSF page and, from the context menu, choose **Create** > **Pie Graph Theme**.
3. In the ensuing Create Pie Graph Theme Binding dialog, do the following to identify the new theme and the base map theme elements that you want to work with:
   a. For **Theme Id**, enter a unique identifier for the pie graph theme that you are creating.
   b. In the **Base Map Theme** section, select the name of the base map and the region in which you want to place the pie graphs.
4. In the **Appearance** section, under **Data**, do the following:
   a. For **Location**, select the location column in the data collection that should be matched to the location column in the base map theme that you selected.

      If needed, click **View Sample Theme Data** to examine the first several rows of the actual data so that you can identify the appropriate location column.
   b. For **Location Label**, select the column in the data collection that contains labels for the locations in the data collection.
   c. In the grid for **Series Attributes**, enter each attribute that contains values that you want represented in the pie graph that you are creating.
   d. Beside each series attribute, enter text that should be used as a label for the data values in the series attribute.
5. Select **Enable Row Selection** only if you want to enable the selection of rows in a related component. You select this component when the page contains a component that is linked to a data collection that is related to the geographic map that you are creating.
6. Click **OK**.

Figure 33–43 shows the completed Create Pie Graph Map Theme dialog for the product popularity by state pie graph theme.

Figure 33–43 Create Pie Graph Map Theme for Product Popularity by State



33.5.9 What Happens When You Add a Pie Graph Theme to a Geographic Map

Dropping a pie graph theme from the Data Controls panel to an existing geographic map has the following effect:

- Creates the bindings for the pie graph theme and adds the bindings to the page definition file
- Adds the necessary pie graph theme code to the JSF page within the map XML

33.5.9.1 Binding XML for a Pie Graph Theme

Example 33–18 shows the row set bindings that were generated for the pie graph theme of the geographic map.

Example 33–18 Pie Graph Theme Binding XML

```
<mapTheme IterBinding="PopularCategoriesByState1Iterator"
        id="PopularCategoriesByState1"
        xmlns="http://xmlns.oracle.com/adfm/dvt">
  <mapThemeDataMap mapThemeType="pieChart">
    <item type="location" value="StateProvince" label="StateProvince"/>
    <data>
      <item value="AudioVideo"
            label="${adfBundle['view.ViewControllerBundle'].AUDIO_VIDEO}"/>
      <item value="CellPhones"
            label="${adfBundle['view.ViewControllerBundle'].CELL_PHONES}"/>
      <item value="Games"
            label="${adfBundle['view.ViewControllerBundle'].GAMES}"/>
    </data>
  </mapThemeDataMap>
</mapTheme>
```

33.5.9.2 Code on the JSF Page for a Pie Graph Theme

Example 33–19 shows the XML code generated on the JSF page for the pie graph theme of the geographic map.

Example 33–19 Pie Graph Theme Code on the JSF Page

```
<dvt:mapPieGraphTheme id="mapPieGraphTheme1"
     themeName="MAP_STATES_NAME"
     shortLabel="#{viewcontrollerBundle.POPULAR_CATEGORIES}"
     pieRadius="10"
     styleName="comet"
     value="#{bindings.PopularCategoriesByState1.geoMapModel}"
     locationColumn="POLYGON_ID"/>
</dvt:mapPieGraphTheme>
```

33.6 Creating Databound Thematic Maps

A thematic map represents business data as patterns in stylized areas or associated markers and does not require a connection to a remote Oracle MapViewer service. Thematic maps focus on data without the geographic details in a geographic map.

The thematic map is packaged with prebuilt base maps including a USA base map, a world base map, and base maps for continents and regions of the world including EMEA and APAC. Each base map includes several sets of regions and one fixed set of cities. A set of regions or cities is referred to as a layer. Each layer can be bound to a data collection and stylized to represent the data with color and pattern fills, or a data marker, or both. At runtime, only one map layer and its associated data can be displayed at a time, unless the thematic map has been enabled for drilling.

The data displayed in a thematic map is based on data collections. Using ADF data controls, JDeveloper makes data binding a declarative task. You drag and drop a collection from the Data Controls panel onto the JSF page and use a Component Gallery to select the base map and map layers on which to display the data. You can then use a Layer Browser and binding dialogs to bind data collection attributes to the data layers in the thematic map.

Stamping is used to associate map layers with a row of data in a data collection. Using stamping, each row of data in the data model can be identified by a style, for example a color or pattern; a marker, for example a circle or square; or an image. When you use stamping, child components are not created for every area, marker, or image in a thematic map. Rather, the content of the component is repeatedly rendered, or stamped, once per data attribute, such as the rows in a data collection.

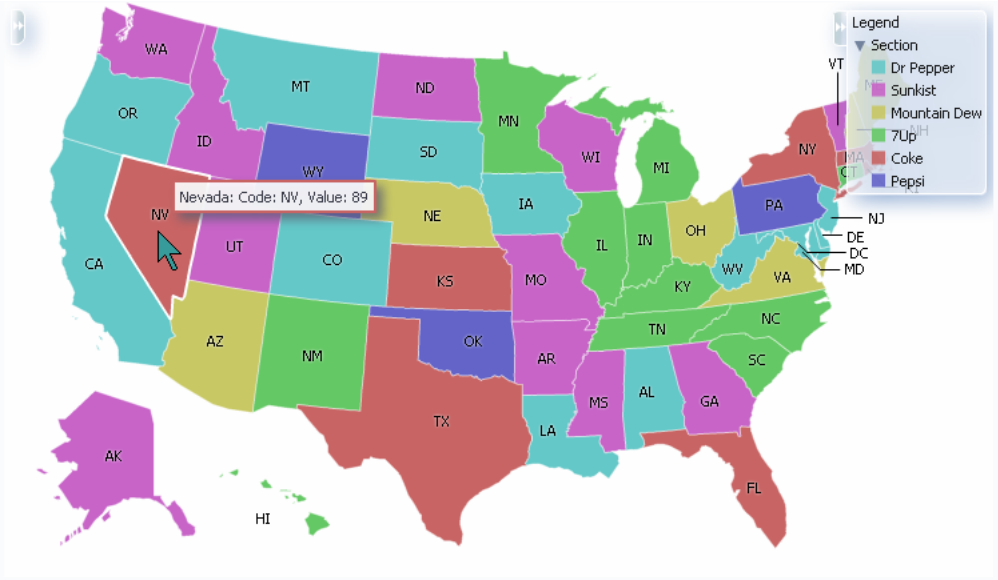
For detailed information about thematic map end user and presentation features, use cases, tag structure, and adding special features to thematic maps, see the "Using Thematic Map Components" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

33.6.1 How to Create a Thematic Map Using ADF Data Controls

You can create a thematic map that displays business data associated with a regional layer in the base map, stylizing the map areas to visually illustrate the data.

For example, you could use a USA base map with a states map layer to display the preferences in each state for a particular brand of soft drink using colors as displayed in Figure 33–44. The example illustrates thematic map default features including a data bound legend and labels associated with the stylized areas when you use the Data Controls panel and thematic map binding dialogs.

Figure 33–44 Thematic Map Displaying Product Preference by US State



The `thematicMap` component uses a model to access the data in the underlying list. The specific model class is `oracle.adf.view.rich.model.CollectionModel`. You can also use other model instances, for example, `java.util.List`, `java.util.Array`, and `javax.faces.model.DataModel`. The data layer will automatically convert the instance into a `CollectionModel`.

Before you begin:

It may be helpful to have an understanding of how thematic map attributes and thematic map child tags can affect functionality. For more information, see the "Configuring Thematic Maps" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

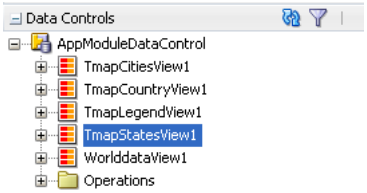
You should already have an ADF data control or ADF managed bean that represents the data you wish to display on the thematic map.

To create a thematic map using the Data Controls panel:

1. From the Data Controls panel, select a data collection.

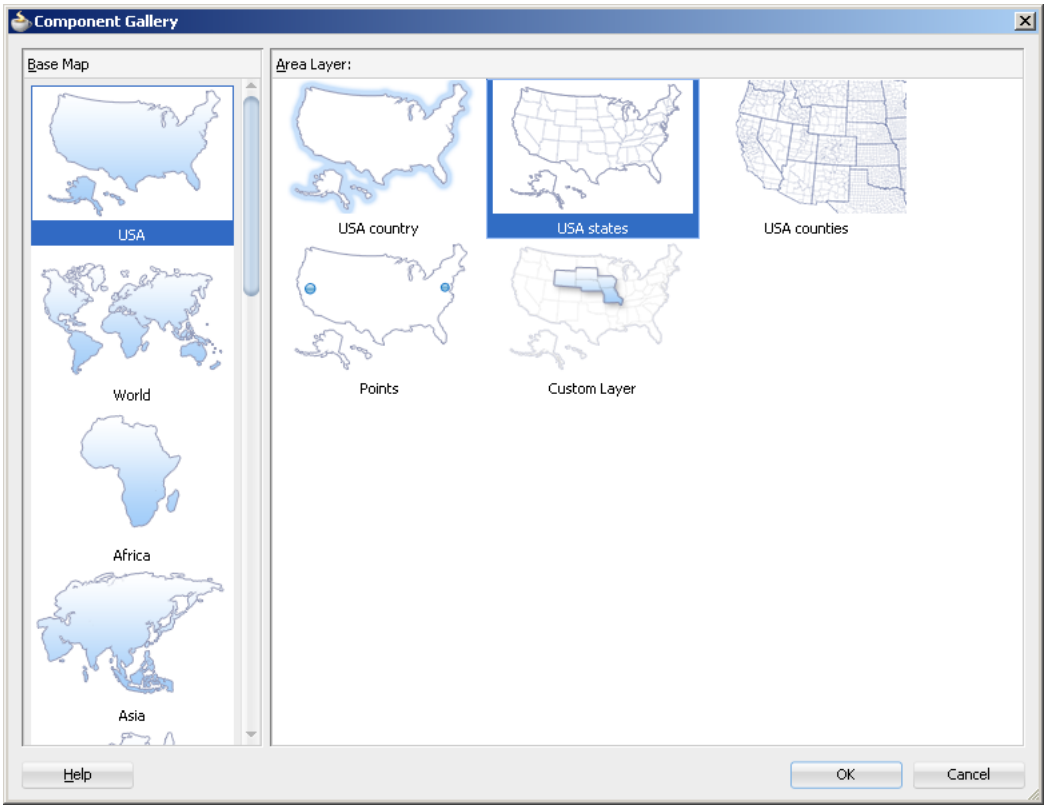
   Figure 33–45 shows an example where you could select the `TmapStatesView1` collection in the Data Controls panel to create a thematic map using colors to represent the preference in each state for a particular product.

Figure 33–45 Data Collection for Product Preference by State



2. Drag the collection onto a JSF page and, from the context menu, choose **Thematic Map**.
3. In the Component Gallery, select the map layer associated with the base map you want to configure for displaying data. In the example, a states map layer in the US base map is selected. Figure 33–46 shows the Component Gallery with the USA states map layer selected.

Figure 33–46 Thematic Map Component Gallery



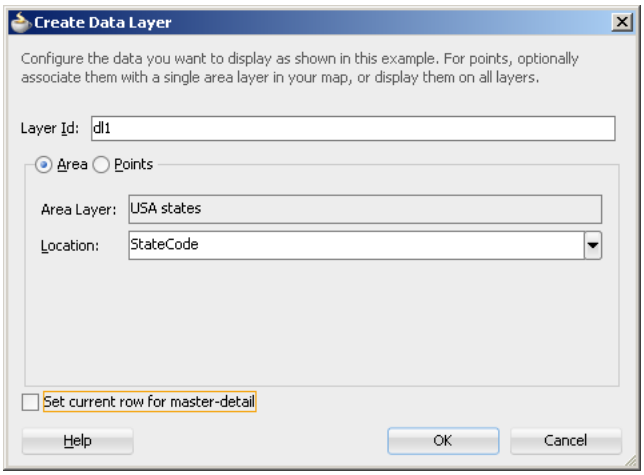
By default, the Create Data Layer dialog opens for adding an area or point data layer to the selected map layer. In the example, an area data layer is configured. For information about adding and configuring point data layers, see Section 33.6.4, "How to Add Data Layers to Thematic Maps."

4. In the Create Data Layer dialog, enter the following:
   - **Layer Id**: Enter a unique name for the data layer you are defining. By default, a unique, consecutively numbered id is entered, `dl1`, `dl2`, and so on.
   - **Area**: Select to add an area data layer to the map layer.
   - **AreaLayer**: References the map layer to which you are adding a data layer. In the example, a **USA States** map layer.
   - **Location**: Select the attribute that represents the column in the data model that determines the location of the data for the areas in the data layer. The locations are Ids of the regions from the base map for which the data is being displayed. For more information, see Section 33.6.3, "What You May Need to Know About Base Map Location Ids."
   - **Set current row for master-detail**: Select if you want to enable master-detail relationships. This is useful when the data collection for the thematic map is a

master in a master-detail relationship with a detail view that is displayed in another UI component on the page. For more information, see Section 33.6.6, "What You May Need to Know About Configuring Master-Detail Relationships."

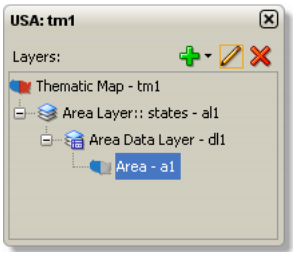
Figure 33–47 shows the completed Create Data Layer dialog.

Figure 33–47 Create Area Data Layer Dialog



An area data layer representing the `areaDataLayer` component, and an area, representing the `area` component is added in the Layer Browser hierarchy. Figure 33–48 shows the expanded Layer Browser after adding an area data layer and area to the map layer.

Figure 33–48 Thematic Map Layer Browser



5. In the Layer Browser, expand the area data layer, select the area to be stylized, and click the **Edit** icon to open the Configure Area dialog.

Note: You configure an area with a default stamp across all areas in the thematic map layer, or you can you can use a child `attributeGroups` tag to generate the style attribute type automatically based on categorical groups in the data set. If the same style attribute is set in both the `area` tag, and by an `attributeGroups` tag, the `attributeGroups` style type will take precedence. For more information, see Section 33.6.7, "Styling Areas, Markers, and Images to Display Data."

---

6. In the Configure Area dialog, Attribute Groups page, enter the following:

- **Grouping Rules**: Use this table to specify the styling of categorical groups of data in a data collection. Use the **Add** icon to add a row to the table for configuring rules for a categorical group and use the **Delete** icon to remove any row selected in the table. Each grouping rule is represented as a `attributeGroups` component, and assigned a unique, consecutively numbered Id, `ag1`, `ag2`, and so on.

  For each row added to the table, enter the following:

  – **Group by Value**: Enter or use the dropdown list to select the attribute representing the column in the data set by which you wish to group the data values. For example, `Category` represents the product preference by state of the different soft drinks.

  Note: The selected attribute should consist of discrete values that can be categorized. For example, a range of numeric values between 40 and 45, are not automatically grouped.

  – **Area Properties**: Use the dropdown list to select the property to use for styling that area. Areas can be styled using **color**, **pattern**, **opacity**, or any combination of these valid values. Choose **Select multiple attributes** from the dropdown list for a dialog to specify any combination of values.

    The default style values that are generated for each property are defined using CSS style properties in the ADF skin. Each `attributeGroups` type has a default ramp defined in the skin, and these can be customized by setting the index-based properties to the desired values. For more information, see Section 33.6.7.4, "What You May Need to Know About Default Style Values for Attribute Groups."

  – **Legend Label**: Enter text or use the dropdown list to select the attribute representing the text to use for the categorical group in the thematic map legend. You can also select **Expression Builder** from the dropdown list to create an EL expression to specify the legend text. For more information, see Section 33.6.8, "Creating Databound Legends."

- **Value-Specific Rules**: Click to open the **Match Rules** and **Exception Rules** tables used to specify a finer detail for one or more data values in categorical groups in a data set. For example, use a match rule to style every state with a preference of Dr Pepper with an aqua, `#00ffff`, color, instead of a predefined range of colors.

  Note: Any match or exception rule specified in these tables will override the settings defined in the **Grouping Rules** table.

- **Match Rules**: Use to specify the style rule matched to one or more data values in a group of data in a data collection. Use the **Add** icon to add a row to the table for configuring a match rule for a categorical group and use the **Delete** icon to remove any row selected in the table. Each match rule is represented as a `attributeMatchRule` component, and assigned a unique, consecutively numbered Id, `amr1`, `amr2`, and so on. The property and property value is defined in a child `f:attribute` tag. For example:

```
<dvt:attributeMatchRule id="amrl" group="Mountain Dew">
  <f:attribute name="color" value="#ffff00"/>
</dvt:attributeMatchRule>
```

For each row added to the table, enter the following:

– **Group Value**: Enter the exact value for a **Group By Value** attribute that will trigger this Match Rule to execute. In the example, soft drink preference data collection attribute, values include `Dr Pepper`, `7 Up`, and so on.

– **Property**: Use the dropdown list to select the property to use for styling that data value. Areas can be styled using **color**, **pattern**, or **opacity** values. The property selected here must match one of the property types listed in the **Area Properties** for the attribute **Grouping Rules**.

– **Property Value**: Enter or use the dropdown list to assign a value to the property. If the value provided by the match override is also in the prebuilt ramp returned by the **Grouping Rules**, then that value will only be used by the overrides and will be skipped in the prebuilt ramp.

  Valid values for **color** are RGB hexidecimal colors.

  Valid values for **pattern** include a choice of twelve prebuilt patterns, for example, **smallChecker**, **largeDiamond**, **smallDiagonalRight**, **large-Crosshatch**. If fill color is specified, the pattern displays in that color on the default white background.

  Valid values for **opacity** range from **0.0** for transparent to **1.0** for opaque.

- **Exception Rules**: Use to specify one or more exception to the style rules for categorical groups in the data set. Use the **Add** icon to add a row to the table for configuring an exception rule and use the **Delete** icon to remove any row selected in the table. Each exception rule is represented as an `attributeExceptionRule` component, and assigned a unique, consecutively numbered Id, `aer1`, `aer2`, and so on. The property and property value is defined in a child `f:attribute` tag. For example:

```
<dvt:attributeExceptionRule id="aer1" condition="#{row.name=='TX'}"
                            label="Texas">
  <f:attribute name="color" value="#ff00ff"/>
</dvt:attributeExceptionRule>
```

  For each row added to the table, enter the following:

  – **Condition**: Enter an EL expression, or use the dropdown list to open an Expression Builder dialog to create an EL expression that replaces the style property value with another when certain conditions are met. For example:

```
#{row.Sales gt 100000}
```

  – **Property**: Use the dropdown list to select the property to use for styling that data value. Areas can be styled using **color**, **pattern**, or **opacity** values. The property selected here must match one of the property types listed in the **Area Properties** for the attribute **Grouping Rules**.

  – **Property Value**: Enter or use the dropdown list to assign a value to the property. If the value provided by the match override is also in the prebuilt ramp returned by the **Grouping Rules**, then that value will only be used by the overrides and will be skipped in the prebuilt ramp.

    Valid values for **color** are RGB hexidecimal colors.

Valid values for **pattern** include a choice of twelve prebuilt patterns, for example, **smallChecker**, **largeDiamond**, **smallDiagonalRight**, **large-Crosshatch**. If fill color is specified, the pattern displays in that color on the default white background.

Valid values for **opacity** range from **0.0** for transparent to **1.0** for opaque.

– **Legend Label**: Enter a text resource to use for the legend label. The text resource can be a translatable string from a resource bundle or an EL expression executed at runtime. Use the dropdown list to open a Select Text Resource or Expression Builder dialog. If you need help, press **F1** or click **Help**.

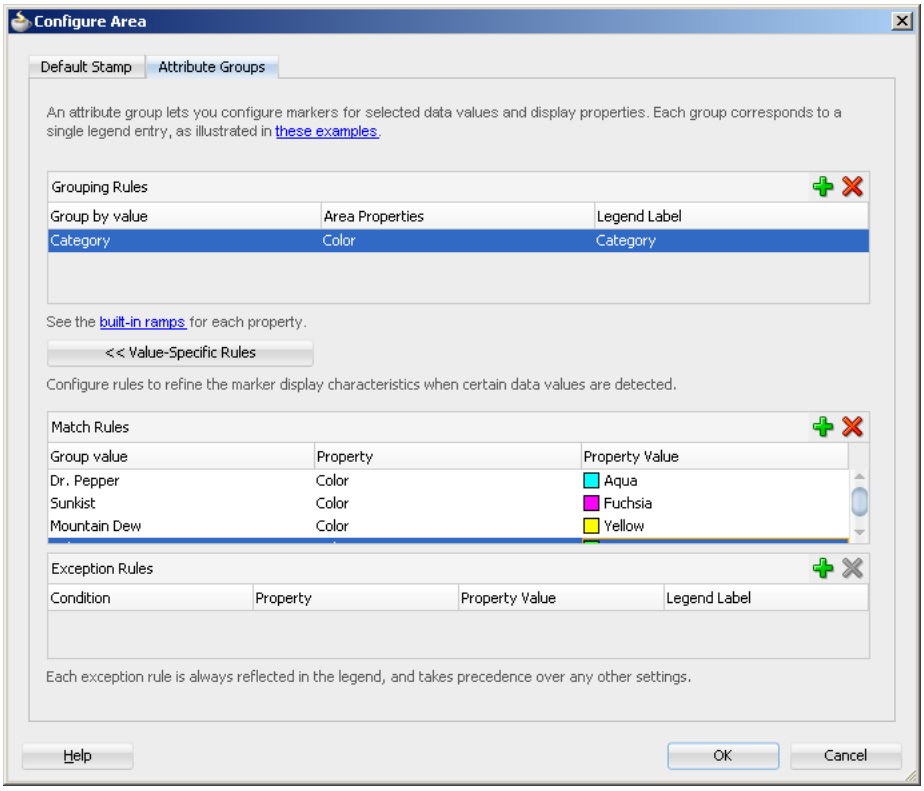
Note: the text resource option is only available for a fixed area. For row-varying areas, use an EL expression to retrieve a row-varying key to look up the text resource in a resource bundle, for example:

```
#{viewController.ResourceBundle[row.label]}
```

- **Messages**: Review and clear as necessary any alerts related to the configuration of the area.

Figure 33–49 shows the completed Configure Area dialog. The warning in the message pane alerts the user that the default area color specified in the Default Stamp page of the dialog will be overwritten by the color specified in the Grouping Rules and any value-specific overrides specified in the Attribute Groups page.

Figure 33–49 Configure Area Dialog Attribute Groups Page



You can add additional map layers representing available regions in the geographical hierarchy of the base map, and associate area or point data layers using the same data collection you used to create the thematic map. The Layer Browser represents the logical structure of the map layers, area and point data layers, and stylized areas and markers. Use the Layer Browser to:

- Add additional map layers in the base map geographical hierarchy to the thematic map.
- Define a custom map layer in the geographical hierarchy of the base map, using lower level regions to aggregate the regions in the custom layer. For more information, seeSection 33.6.9, "How to Define a Custom Map Layer."
- Add, edit, or delete area or point (global or map layer specific) data layers. For more information, see Section 33.6.4, "How to Add Data Layers to Thematic Maps."
- Add, edit, or delete stylized areas or markers. For more information, see Section 33.6.7, "Styling Areas, Markers, and Images to Display Data."

After creating a thematic map using data controls, you can customize the default map labels, legend display, and add interactivity and animation effects. For more information, see in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

33.6.2 What Happens When You Use Data Controls to Create a Thematic Map

When you use ADF data controls to create a thematic map, JDeveloper:

- Defines the bindings for the thematic map in the page definition file of the JSF page, and
- Inserts code in the JSF page for the DVT thematic map components.

Example 33–20 shows the bindings defined in the page definition file of the JSF page for the example thematic map in Figure 33–44.

Example 33–20 XML Bindings in Page Definition File

```
<bindings>
  <tree IterBinding="TmapStatesView1Iterator" id="TmapStatesView1">
    <nodeDefinition DefName="model.TmapStatesView">
      <AttrNames>
        <Item Value="Abbrev"/>
        <Item Value="Category"/>
        <Item Value="Color"/>
        <Item Value="Data"/>
        <Item Value="Id"/>
        <Item Value="Name"/>
      </AttrNames>
    </nodeDefinition>
  </tree>
</bindings>
```

Example 33–21 shows the code inserted in the JSF page for the example thematic map in Figure 33–44.

Example 33–21 JSF Code for Thematic Map Components

```
<dvt:thematicMap id="tm1" basemap="usa">
  <dvt:areaLayer layer="states" id="al1">
    <dvt:areaDataLayer id="dl1"
```

```
          value="#{bindings.TmapStatesView1.collectionModel}" var="row">
        <dvt:areaLocation name="#{row.Name}" id="al2">
          <dvt:area id="a1">
            <dvt:attributeGroups id="ag1" value="#{row.Category}" type="color"
                                 label="#{row.Category}">
              <dvt:attributeMatchRule id="amr1" group="Mountain Dew">
                <f:attribute name="color" value="#ffff00"/>
              </dvt:attributeMatchRule>
              <dvt:attributeMatchRule id="amr2" group="Sunkist">
                <f:attribute name="color" value="#ff00ff"/>
              </dvt:attributeMatchRule>
              <dvt:attributeMatchRule id="amr3" group="Dr Pepper">
                <f:attribute name="color" value="#00ffff"/>
              </dvt:attributeMatchRule>
              <dvt:attributeMatchRule id="amr4" group="7 Up">
                <f:attribute name="color" value="#00ff00"/>
              </dvt:attributeMatchRule>
              <dvt:attributeMatchRule id="amr5" group="Pepsi">
                <f:attribute name="color" value="#0000ff"/>
              </dvt:attributeMatchRule>
              <dvt:attributeMatchRule id="amr6" group="Coke">
                <f:attribute name="color" value="#ff0000"/>
              </dvt:attributeMatchRule>
            </dvt:attributeGroups>
          </dvt:area>
        </dvt:areaLocation>
      </dvt:areaDataLayer>
    </dvt:areaLayer>
    <dvt:legend id="l1">
      <dvt:legendSection source="al1:dl1:ag1" id="ls1"/>
    </dvt:legend>
  </dvt:thematicMap>
```

33.6.3 What You May Need to Know About Base Map Location Ids

Each base map provided for the thematic map component has two or more prebuilt map layers that represent a set of regions. For example, the `world` base map includes a map layer for `continents` and another layer for `countries`. The regions in the lower level map layers are aggregated to make up the next level in the geographical hierarchy. The map layer is specified in the `layer` attribute of the `areaLayer` component.

When you are binding your data collection to a thematic map, you must provide a column in the data model that specifies the location of the area or point data using the map location Ids of the regions from the base map for which the data is being displayed. Area locations are specified in the `name` attribute of the `areaLocation` component, and point locations are specified in the `pointName` attribute for the `pointLocation` component when its `type` attribute is set to `pointName`.

For the United States base map, the locations Ids are determined by the following naming rules:

- `country` layer: `USA`
- `states` layer: Use the two-letter postal abbreviation. For example, the location Id for Massachusetts is `MA`, and the location Id for Texas is `TX`.
- `counties` layer: Use the `states` layer location Id, followed by an underscore, and then the name of the county, all in capital letters with underscores replacing

characters that are not letters. For example, the location Id for Middlesex county in Massachusetts is `MA_MIDDLESEX`, and the location Id for Red River county in Texas is `TX_RED_RIVER`.

- `cities` layer: Use the `states` layer location Id, followed by an underscore, and then the name of the city, all in capital letters with underscores replacing characters that are not letters. For example, the location Id for the id for Boston, Massachusetts is `MA_BOSTON`, and the location Id for San Antonio, Texas is `TX_SAN_ANTONIO`.

For all other base maps, location Ids are determined by the following naming rules:

- `continents` layer: `AF` (Africa), `AS` (Asia), `AU` (Australia), `EU` (Europe), `NA` (North America), and `SA` (South America) for the `world`, `africa`, `asia`, `australia`, `europe`, `northAmerica`, and `southAmerica` base maps.
- `worldRegions` layers: `APAC` (Asia-Pacific), `EMEA` (Europe and the Middle East), `LAT` (Latin America), `NA` (United States and Canada) for the `worldRegions`, `apac`, `emea`, `latinAmerica`, and `usaAndCanada` base maps.
- `countries` layer: Use the ISO 3166-1 alpha-3 country codes. For example, the location Id for Canada is `CAN`, and the location Id for China is `CHN`.
- `cities` layer: Use the three-letter `countries` location ID, followed by an underscore, and then the name of the city, all in capital letters with underscores replacing characters that are not letters. For example, the location Id for Tornonto, Canada is `CAN_TORONTO`, and the location Id for Los Angeles, United States is `USA_LOS_ANGELES`.

You can download a comma-separated value (CSV) file for each of the prebuilt map layers with a complete listing of all the thematic map base map location Ids. Find these links in the tag documentation for the `areaLocation` component, `name` attribute. To access tag documentation for the data visualization components, select the component in the Structure window and click the help button in the Property Inspector.

33.6.4 How to Add Data Layers to Thematic Maps

You use data layers to associate map layers with a data collection. Using stamping, each row of data in the data model can be identified by a style, for example a color or pattern; a marker, for example a circle or square; or an image. When a map layer is displayed at runtime, the data appears as stylized areas, markers, or images.

Map layers can display data using an area data layer and/or one or more point data layers. Area data layers can be styled using areas, markers, or images. Point data layers can be styled using markers or images.

When a point data layer is associated with the base map as a direct child of `thematicMap` instead of as a child of a specific map layer, the data displays at all times and is called a *global point layer*.

Before you begin:

It may be helpful to have an understanding of how thematic map attributes and thematic map child tags can affect functionality. For more information, see the "Configuring Thematic Maps" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

You should already have a thematic map on your page. If you do not, follow the instructions in this chapter to create a thematic map. For more information, see Section 33.6.1, "How to Create a Thematic Map Using ADF Data Controls."

To add an area data layer to a map layer:

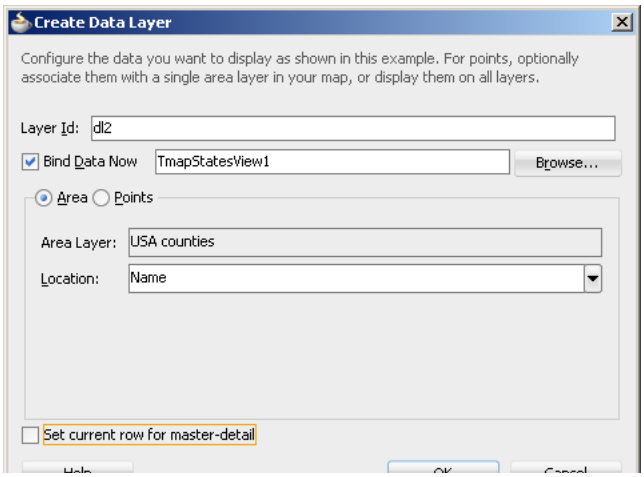
1. Select the thematic map in the Visual Editor.
2. In the Layer Browser, select the map layer to which you wish to bind a row in the data collection. If the Layer Browser is not open in the Visual Editor, right-click inside the map and choose **Open Layer Browser**.
3. From the **Add** icon dropdown list, choose **Add Data Layer** to open the Create Data Layer dialog.
4. In the Create Data Layer dialog, enter the following:
   - **Layer Id**: Enter a unique name for the data layer you are defining. By default, a unique, consecutively numbered id is entered, `dl1`, `dl2`, and so on.
   - **Bind Data Now**: Select and click **Browse** to open the Picker dialog > Data Controls Definitions page. Select the data collection in the ADF data controls you are using to data bind your data layer.

     Note: Alternatively, you can use the Expression Builder page to select an ADF managed bean you are using to databind your area data layer and areas. You can also use the Expression Builder dialog in each remaining field.

   - **Area**: Select to add an area data layer to the map layer.
   - **AreaLayer**: References the map layer to which you are adding a data layer.
   - **Location**: Select attribute that represents the column in the data model that determines the location of the data for the areas in the data layer. The locations are Ids of the regions from the base map for which the data is being displayed. For more information, see Section 33.6.3, "What You May Need to Know About Base Map Location Ids."
   - **Set current row for master-detail**: Select if you want to enable master-detail relationships. This is useful when the data collection for the thematic map is a master in a master-detail relationship with a detail view that is displayed in another UI component on the page. For more information, see Section 33.6.6, "What You May Need to Know About Configuring Master-Detail Relationships."

Figure 33–50 shows the completed Create Area Data Layer dialog.

Figure 33–50 Create Area Data Layer Dialog



By default, an area data layer representing the `areaDataLayer` component, and an area, representing the `area` component is added in the Layer Browser hierarchy. Example 33–22 shows the code added to the JSF page when you add an area data layer to a map layer.

Example 33–22 Area Data Layer Code in JSF Page

```
<dvt:thematicMap id="tm1" basemap="usa">
  <dvt:areaLayer layer="counties" id="al1">
    <dvt:areaDataLayer id="dl2"
                       value="#{bindings.TmapStatesView11.collectionModel}"
                       var="row">
      <dvt:areaLocation name="#{row.Name}" id="al2">
        <dvt:area id="a1"/>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
...
</dvt:thematicMap>
```

To add a point data layer to a map layer:

1. Select the thematic map in the Visual Editor.
2. In the Layer Browser, select the map layer to which you wish to bind a data layer. If the Layer Browser is not open in the Visual Editor, right-click inside the map and choose **Open Layer Browser**.
3. From the **Add** icon dropdown list, choose **Add Data Layer** to open the Create Data Layer dialog.

> **Note:** If you wish to create a global point layer that displays at all times on the thematic map, choose **Create Global Point Layer**.

4. In the Create Data Layer dialog, enter the following:
   - **Layer Id**: Enter a unique ID for the data layer you are defining. By default, a unique, consecutively numbered id is entered, `dl1`, `dl2`, and so on.

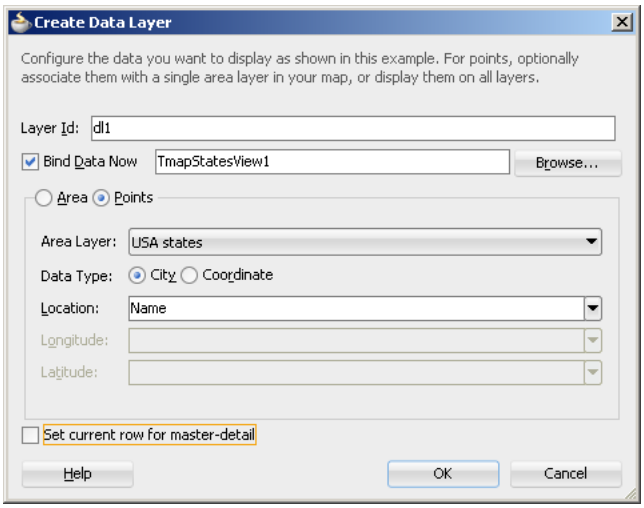
- **Bind Data Now**: Select and click **Browse** to open the Picker dialog > Data Controls Definitions page. Select the data collection in the ADF data controls you are using to data bind your data layer.

  Note: Alternatively, you can use the Expression Builder page to select an ADF managed bean you are using to databind your point data layer and markers. You can also use the Expression Builder dialog in each remaining field.

- **Points**: Select to add a point data layer to the map layer.
- **AreaLayer**: Use the dropdown list to select the map layer to which you are associating a data layer. If you select **All**, the point data layer will be configured as a global point layer and display at all times. If you select an available map layer, the point data layer will only display with that map layer is displayed.
- **Data Type**: Choose one of the following:
  - **City**: Select to use a `pointName` data type representing the named points, such as cities, in the data collection that map to named points in the base map.
  - **Coordinates**: Select to use a `pointXY` data type representing the columns in the data collection that join `pointX` and `pointY` to define the point locations.
- **Location**: Available if a City Data Type is specified. Use the dropdown list to select the attribute that represents the column in the data model that determines the location of the data for the points in the data layer. The locations are IDs of the points from the base map for which the data is being displayed. For more information, see Section 33.6.3, "What You May Need to Know About Base Map Location Ids."
- **Longitude**: Available if a Coordinates Data Type is specified. Use the dropdown list to select the attribute in the data collection that represents the longitude, or `pointX` of the marker.
- **Latitude**: Available if a Coordinates Data Type is specified. Use the dropdown list to select the attribute in the data collection that represents the latitude, or `pointY` of the markers.
- **Set current row for master-detail**: Select if you want to enable master-detail relationships. This is useful when the data collection for the thematic map is a master in a master-detail relationship with a detail view that is displayed in another UI component on the page. For more information, see Section 33.6.6, "What You May Need to Know About Configuring Master-Detail Relationships."

Figure 33–51 shows the completed Create Point Data Layer dialog for a `pointName` data type.

Figure 33–51 Create Point Data Layer Dialog



By default, a point data layer representing the `pointDataLayer` component, and a marker, representing the `marker` component is added in the Layer Browser hierarchy. Example 33–23 shows the code added to the JSF page when you add a point data layer to a map layer.

Example 33–23 Point Data Layer Code in JSF Page

```
<dvt:thematicMap id="tm1" basemap="usa">
  <dvt:areaLayer layer="states" id="al1">
    <dvt:pointDataLayer id="dl1"
                        value="#{bindings.TmapCitiesView11.collectionModel}"
                        var="row">
      <dvt:pointLocation type="pointName" pointName="#{row.City}" id="pl1">
        <dvt:marker id="m2"/>
      </dvt:pointLocation>
    </dvt:pointDataLayer>
  </dvt:areaLayer>
...
</dvt:thematicMap>
```

Example 33–24 shows the code added to the JSF page when you add a global point data layer to a thematic map.

Example 33–24 Global Point Data Layer Code in JSF Page

```
<dvt:thematicMap>
  <areaLayer layer="states" id="al1"/>
  <dvt:pointDataLayer id="dl2"
                      value="#{bindings.TmapCitiesView12.collectionModel}"
                      var="row">
    <dvt:pointLocation type="pointXY" pointX="#{row.Longitude}"
                       pointY="#{row.Latitude}" id="pl2">
      <dvt:marker id="m3"/>
    </dvt:pointLocation>
  </dvt:pointDataLayer>
...
</dvt:thematicMap>
```

After adding the area or point data layer to the thematic map, you style the data using areas, markers, or images. For more information, see Section 33.6.7, "Styling Areas, Markers, and Images to Display Data."

33.6.5 How to Configure Drilling in Thematic Maps

A thematic map with related data views in different map layers can be configured for drilling between the higher and lower level data views. For example, a thematic map can display data for sales category by USA state drilled down to USA county data as illustrated in Figure 33–62. When a state or county is selected, drilling up or down is initiated through a context menu choice or the Control Panel.

The following requirements must be met to achieve thematic map area drilling declaratively:

- For each map layer (`areaLayer`) in the drilling hierarchy, you must bind its child `areaDataLayer` with a data control that defines the related data for that map layer.
- Each `areaDataLayer` in the map layer drill hierarchy must have its `selectionMode` attribute set to `single` or `multiple`.
- You must configure the `area` in the lower level map layer drill hierarchy to display data using the same default stamp or categorical attribute style used in the higher level map layer `area` component.
- The `thematicMap` component `drilling` attribute must be set to `on`.

Note: If drilling is enabled for the thematic map, drilling between a custom layer and the map layer used to aggregate the custom layer is available.

Before you begin:

It may be helpful to have an understanding of how thematic map attributes and thematic map child tags can affect functionality. For more information, see the "Configuring Thematic Maps" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

You should already have a thematic map on your page. If you do not, follow the instructions in this chapter to create a thematic map. For more information, see Section 33.6.1, "How to Create a Thematic Map Using ADF Data Controls."

You should already have data controls that define the data model for each of the map layers in the drill hierarchy.

To configure a thematic map area for drilling:

1. In the Structure window, select the **dvt:thematicMap** component.
2. In the Property Inspector, expand the **Behavior** section. Use this section to set the following attributes:
   - **Drilling**: Use to enable drilling the area data view between thematic map layers. From the dropdown list select **on** to enable drilling. The default value is **off**.
   - **MaintainDrill**: Optionally, use to specify an optional **true** value for maintaining the drilled state of a previously drilled area when a new area is drilled. The default value is **false**.

- **DrillBehavior**: Optionally, use to specify an optional **zoomToFit** effect on the area being drilled. The default value is **none**.

3. In the Layer Browser, select each **Area Layer** component in the desired drilling hierarchy and do the following:
   - Select the **Add icon** and choose **Add Data Layer** to open the Create Data Layer dialog. Complete the dialog to add an area data layer and bind the data layer to the data control for that map layer. If you need help, press **F1** or click **Help**.

     Note: If the **Area Data Layer** is already present, select the **Edit icon** to confirm binding to the data control.

   - In the Property Inspector, expand the **Behavior** section and set the **SelectionMode** attribute to **single** or **multiple**.
4. In the Layer Browser, select each **Area Data Layer** component in the desired drilling hierarchy and do the following:
   - Select the **Add icon** and choose **Add Area** to open the Configure Area dialog. Complete the dialog to define a default stamp or use attribute groups to style the area for that map layer. If you need help, press **F1** or click **Help**.

     Note: If the **Area** is already present, select the **Edit icon** to confirm the styling of the area.

Figure 33–25 shows sample code for drilling enabled for USA states and counties map layers with an area styled to display data about sales categories.

Example 33–25 Sample Code for Configuring Drilling

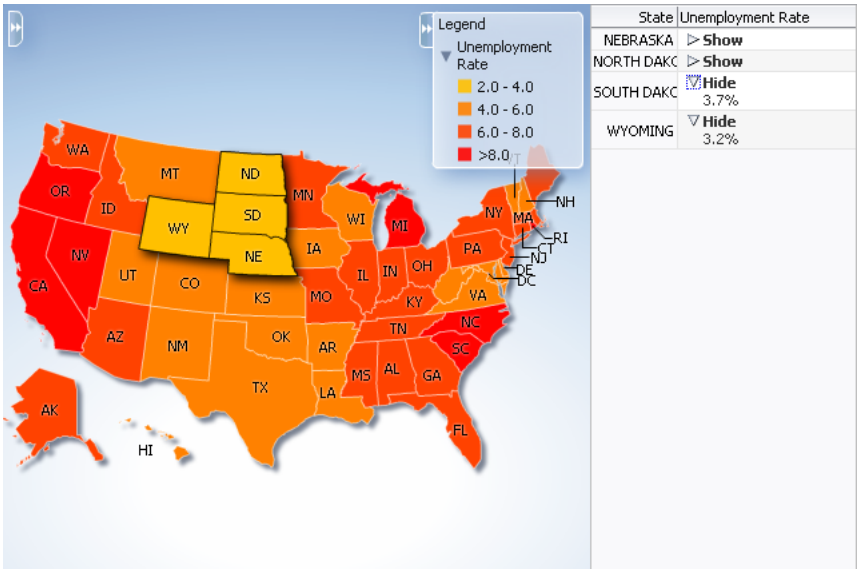
```
<dvt:thematicMap id="thematicMap"
                 basemap="usa"
                 drilling="on"
                 maintainDrill="true"
                 drillBehavior="zoomToFit"
                 animationOnDisplay="none"
  <dvt:areaLayer id="al1" layer="states">
    <dvt:areaDataLayer id="adl1"
                       selectionMode="single"
                       contentDeliver="immediate"
                       value="#{row.state}"
                       var="row"
                       var="rowStatus">
      <dvt:areaLocation id="areaLocS" name="#{row.stname}">
        <dvt:area id="a1" fillColor="#{row.state}"/>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
  <dvt:areaLayer id="al2" layer="counties">
    <dvt:areaDataLayer id="adl2"
                       selectionMode="single"
                       contentDelivery="immediate"
                       value="#{row.county}"
                       var="row"
                       varStatus="rowStatus">
      <dvt:areaLocation id="areaLocC" name="#{row.coname}">
```

```
        <dvt:area id="a2" fillColor="#{row.county}"/>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
...
</dvt:thematicMap>
```

33.6.6 What You May Need to Know About Configuring Master-Detail Relationships

You can configure a thematic map to display its associated data in another UI component on the page such as a table. In this configuration the data collection for the thematic map is a master in a master-detail relationship with a detail view in another UI component. Figure 33–52 shows a thematic map displaying unemployment rates by state. Users can select one or multiple states to display the data detail view in the table.

Figure 33–52 Thematic Map Master and Detail Table View



The following requirements must be met to achieve this master-detail processing declaratively:

- You must use the same data collection to provide data for both views as follows:
  - Bind the thematic map `areaDataLayer` or `pointDataLayer` to the data collection whose attributes represent the data to be styled by areas or markers in the thematic map layer.
  - Bind the other ADF component (such as a table) to same data collection.
- Select **Set the current row for master-detail** in the Create Data Layer dialog to automatically set a value for the `selectionListener` attribute of the thematic map `areaLayer` component and use the `processSelection` method that is already part of the thematic map binding.

  For example, if the `value` attribute of the thematic map area data layer component is `value="#{stateData.employmentData}"`, then the `selectionListener` attribute is set to:

  ```
  selectionListener="#{stateData.employmentData.processSelection}".
  ```

- Ensure that the `selectionMode` attribute on the `areaDataLayer` or `pointDataLayer` component is set to `single` or `multiple`, depending on the requirements for the thematic map.

33.6.7 Styling Areas, Markers, and Images to Display Data

An area data layer is used to associate map layers with a data collection. Using stamping, each row of data in the data model can be identified by a style, for example a color or pattern; a marker, for example a circle or square; or an image.

A point data layer is used to associate a set of points on a map with a data collection. The data point can be specified by a named point in a map layer, for example, cities in the US map, or by longitude and latitude. Using stamping, each row of data in the data model can be identified by a marker, for example a circle or square, or an image.

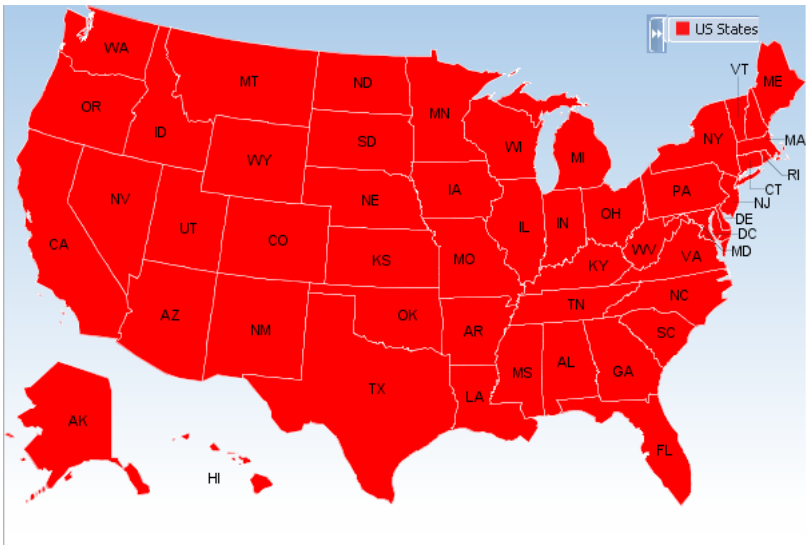
33.6.7.1 How to Style Areas to Display Data

You configure an area with a default stamp across all areas in the thematic map `areaLayer`, or you can use a child `attributeGroups` tag to generate the style attribute type automatically based on categorical groups in the data set. If the same style attribute is set in both the `area` tag, and by an `attributeGroups` tag, the `attributeGroups` style type will take precedence.

By default, when you add an area data layer, a configurable area is added to the Layer Browser. You can use the Layer Browser to add additional areas or markers to an area data layer, or markers to a point data layer. The Layer Browser reflects the logical structure of the map layers, area or point data layers, and areas or markers configured in the thematic map.

For example, using the default stamp, you can style all the states in the `states` layer of the `usa` base map with the color red as illustrated in Figure 33–53.

Figure 33–53 Thematic Map with Area Default Stamp Configured



Before you begin:

It may be helpful to have an understanding of how thematic map attributes and thematic map child tags can affect functionality. For more information, see the "Configuring Thematic Maps" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

You should already have a thematic map on your page. If you do not, follow the instructions in this chapter to create a thematic map. For more information, see Section 33.6.1, "How to Create a Thematic Map Using ADF Data Controls."

To style areas using a default stamp:

1. In the Layer Browser, select the area at the area location inside the map layer you are configuring.
2. Choose the **Edit** icon to open the Configure Area dialog.
3. In the Configure Area dialog, Default Stamp page, enter the following:
   - **Location**: By default, the attribute that represents the column in the data model that determines the location of the data for the areas in the data layer. The locations are Ids of the regions from the base map for which the data is being displayed. This read-only field displays the EL expression that maps the stamped area to a region in the base map. For more information, see Section 33.6.3, "What You May Need to Know About Base Map Location Ids."
   - **Color**: Optionally, from the dropdown list select the fill color for the area. Valid values are RGB hexidecimal. Choose **Custom Color** from the dropdown list to open a Color Picker dialog
   - **Pattern**: Optionally, from the dropdown list select one of twelve prebuilt patterns to style the area, for example, **smallChecker**, **largeDiamond**, **smallDiagonalRight**, **largeCrosshatch**. If fill color is specified, the pattern displays in that color on the default white background.
   - **Opacity**: Optionally, specify the opacity of the fill color of the area. Valid values range from **0.0** for transparent to **1.0** for opaque.
   - **Include in Legend**: Select and use the search icon to open a Select Text Resource dialog to select or create an application text resource to use for the legend text. The text resource can be a translatable string from a resource bundle or an EL expression executed at runtime. If you need help, press **F1** or click **Help**.

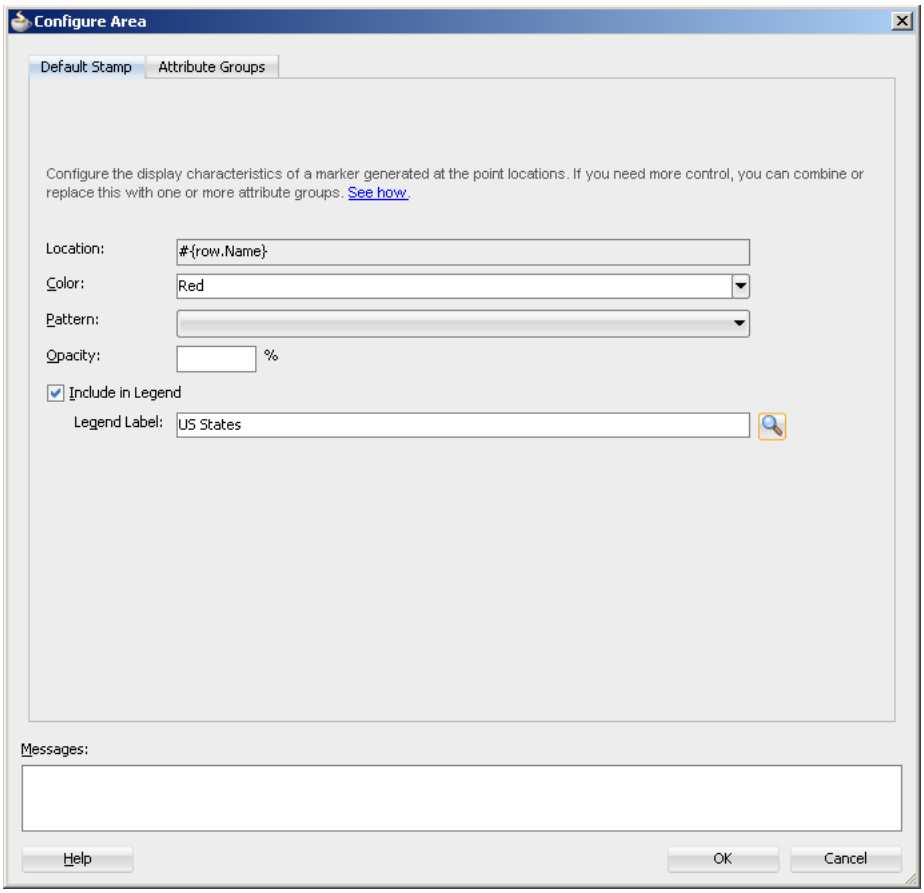
     Note: the text resource option is only available for a fixed area. For row-varying areas, use an EL expression to retrieve a row-varying key to look up the text resource in a resource bundle, for example:

     ```
     #{viewController.ResourceBundle[row.label]}
     ```

   - **Messages**: Review and clear as necessary any alerts related to the configuration of the area.

Figure 33–54 shows the Configure Area dialog for the Default Stamp page.

Figure 33–54 Configure Area Dialog Default Stamp Page



Example 33–26 shows the code inserted in the JSF page for the configured area.

Example 33–26 Sample Code for Area Default Stamp Configuration

```
<dvt:thematicMap id="tm1" basemap="usa">
  <dvt:areaLayer layer="states" id="al1">
    <dvt:areaDataLayer id="dl1"
                       value="#{bindings.TmapStatesView1.collectionModel}"
                       var="row">
      <dvt:areaLocation name="#{row.Name}" id="al2">
        <dvt:area id="a1" fillColor="#ff0000">
          <f:attribute name="legendLabel"
                value="#{Bundle.US_STATES}"/>
        </dvt:area>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
</dvt:thematicMap>
```

If you wish to style areas using categorical groups of data in the data collection, use the Attribute Groups page of the Configure Area dialog. For details about using the Attribute Groups page including sample code, see the use case described in Section 33.6.1, "How to Create a Thematic Map Using ADF Data Controls."

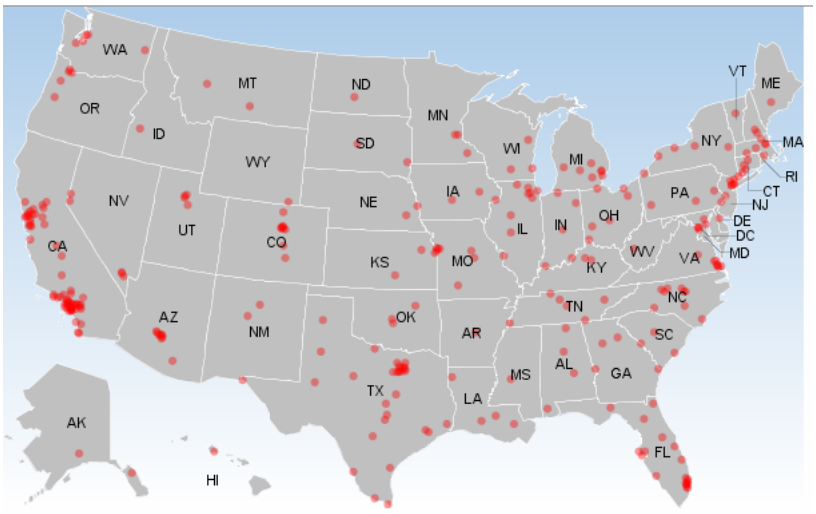
33.6.7.2 How to Style Markers to Display Data

You configure a marker with a default stamp across all markers in the thematic map layer, or you can use a child `attributeGroups` tag to generate the style attribute type automatically based on categorical groups in the data set. If the same style attribute is set in both the `marker` tag, and by an `attributeGroups` tag, the `attributeGroups` style type will take precedence.

By default, when you add a point data layer, a configurable marker element is added to the Layer Browser. You can use the Layer Browser to add additional areas or markers to an area data layer, or markers to a point data layer. The Layer Browser reflects the logical structure of the map layers, area or point data layers, and areas or markers configured in the thematic map.

For example, using the default stamp, you can identify all the predefined cities in the states layer of the USA base map with a red circle as illustrated in Figure 33–55. In the example, markers are styled on a point data layer. Styling markers on an area data layer is similar.

Figure 33–55 Thematic Map with Marker Default Stamp Configured



Before you begin:

It may be helpful to have an understanding of how thematic map attributes and thematic map child tags can affect functionality. For more information, see the "Configuring Thematic Maps" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

You should already have a thematic map on your page. If you do not, follow the instructions in this chapter to create a thematic map. For more information, see Section 33.6.1, "How to Create a Thematic Map Using ADF Data Controls."

To style markers using a default stamp:

1. In the Layer Browser, select the marker at the point location inside the map layer you are configuring
2. Select the **Edit** icon to open the Configure Marker dialog.
3. In the Configure Marker dialog, Default Stamp page, enter the following:

- **Location**: By default, the attribute that represents the column in the data model that determines the location of the data for the markers in the data layer. The locations are Ids of the points from the base map for which the data is being displayed. This read-only field displays the EL expression that maps the stamped marker to a point in the base map. For more information, see Section 33.6.3, "What You May Need to Know About Base Map Location Ids."
- **Color**: Optionally, from the dropdown list select the fill color for the marker. Valid values are RGB hexidecimal. Choose **Custom Color** from the dropdown list to open a Color Picker dialog.
- **Pattern**: Optionally, from the dropdown list select one of twelve prebuilt patterns to style the marker, for example, **smallChecker**, **largeDiamond**, **smallDiagonalRight**, **largeCrosshatch**. If fill color is specified, the pattern displays in that color on the default white background.
- **Opacity**: Optionally, specify the opacity of the fill color of the marker. Valid values range from **0.0** for transparent to **1.0** for opaque.
- **Shape**: Optionally, select the shape of the marker. Valid values are **circle** (default), **square**, **plus**, **diamond**, **triangleUp**, **triangleDown**, and **human**.

  Optionally, use the **Custom Shape** field to specify the `shapePath` value to a Scalable Vector Graphics (SVG) file to use for the marker. Enter the path or use the **Search** icon to open the Select SVG File dialog and navigate to the file location. This option is only available if the default shape is selected in the **Shape** field.

  Shapes can also be specified using CSS style properties. Predefined marker shapes can be overwritten, and the paths to SVG files for custom markers can also be defined without using the `shapePath` attribute. For more information, see Section 33.6.7.3, "What You May Need to Know About Styling Markers."
- **Size**: Optionally, enter a percentage by which to scale the marker from its default size for **ScaleX** (horizontal), and **ScaleY** (vertical). The percentages are then scaled to a float. For example, you can double the marker width by setting **ScaleX** to `200`, written to the tag as `2.0`, and halve the height by setting **ScaleY** to `50`, written to the tag as `0.5`.
- **Include in Legend**: Enter a text resource to use for the legend label. The text resource can be a translatable string from a resource bundle or an EL expression executed at runtime. Use the dropdown list to open a Select Text Resource or Expression Builder dialog. If you need help, press **F1** or click **Help**.

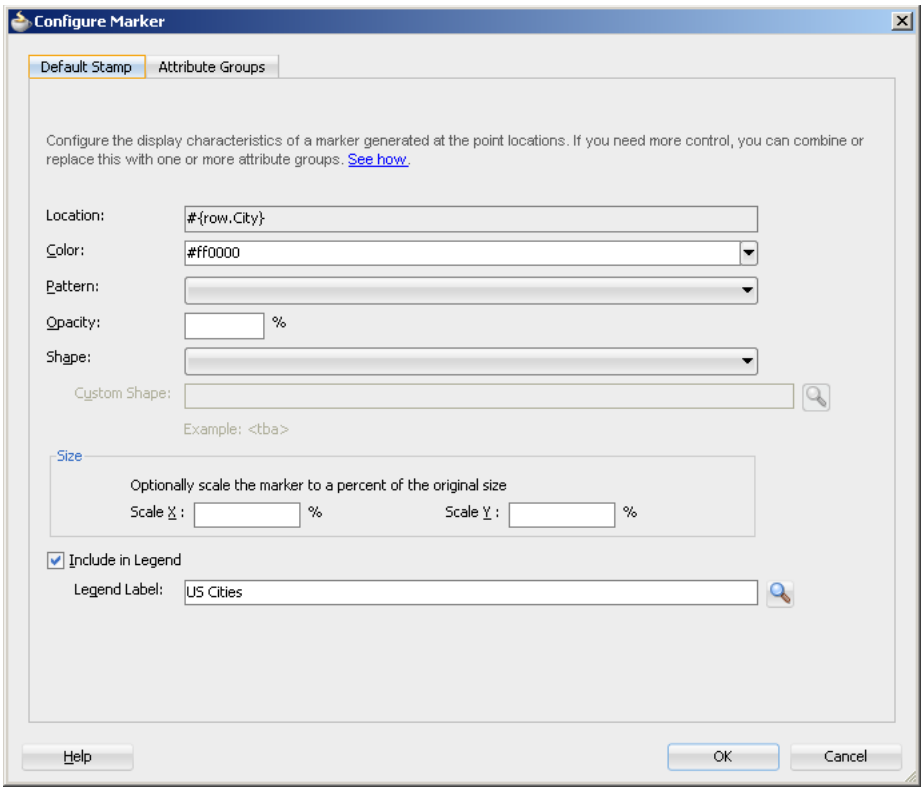
  Note: the text resource option is only available for a fixed area. For row-varying areas, use an EL expression to retrieve a row-varying key to look up the text resource in a resource bundle, for example:

  ```
  #{viewController.ResourceBundle[row.label]}
  ```

- **Messages**: Review and clear any messages related to the configuration of the marker.

Figure 33–56 shows the Configure Marker dialog, Default Stamp page.

Figure 33–56 Configure Marker Dialog Default Stamp Page



Example 33–27 shows the code inserted in the JSF page for the configured marker.

Example 33–27 Sample Code for Marker Default Stamp Configuration

```
<dvt:thematicMap id="tm1" basemap="usa">
  <dvt:areaLayer layer="states" id="al1">
    <dvt:pointDataLayer id="dl2"
                        value="#{bindings.TmapCitiesView1.collectionModel}"
                        var="row">
      <<dvt:pointLocation type="pointName" pointName="#{row.City}" id="pl1">
        <dvt:marker id="m2" fillColor="#ff0000">
          <f:attribute name="legendLabel"
                value="#{Bundle.US_CITIES}"/>
        </dvt:marker>
      </dvt:pointLocation>
    </dvt:pointDataLayer>
  </dvt:areaLayer>
</dvt:thematicMap>
```

If you wish to style markers using categorical groups of data in the data collection, use the Attribute Groups page of the Configure Marker dialog. You can configure markers for a point data layer or an area data layer.

To style markers using categorical groups of data:

1. In the Layer Browser, select the marker at the marker location inside the map layer you are configuring.
2. Select the **Edit** icon to open the Configure Marker dialog.
3. In the Configure Marker dialog, Attribute Groups page, enter the following:

- **Grouping Rules**: Use this table to specify the styling of categorical groups of data in a data collection. Use the **Add** icon to add a row to the table for configuring rules for a categorical group and use the **Delete** icon to remove any row selected in the table. Each grouping rule is represented as a `attributeGroups` component, and assigned a unique, consecutively numbered Id, `ag1`, `ag2`, and so on.

  For each row added to the table, enter the following:

  – **Group by Value**: Enter or use the dropdown list to select the attribute representing the column in the data set by which you wish to group the data values.

  Note: The selected attribute should consist of discrete values that can be categorized. For example, a range of numeric values between 40 and 45, are not automatically grouped.

  – **Marker Properties**: Use the dropdown list to select the property to use for styling that marker. Markers can be styled using **color**, **pattern**, **opacity**, **scaleX**, **scaleY**, or any combination of these valid values. Choose **Select multiple attributes** from the dropdown list for a dialog to specify any combination of values.

    The default style values that are generated for each property are defined using CSS style properties in the ADF skin. Each `attributeGroups` type has a default ramp defined in the skin, and these can be customized by setting the index-based properties to the desired values. For more information, see Section 33.6.7.4, "What You May Need to Know About Default Style Values for Attribute Groups."

  – **Legend Label**: Enter text or use the dropdown list to select the attribute representing the text to use for the categorical group in the thematic map legend. You can also select **Expression Builder** from the dropdown list to create an EL expression to specify the legend text. For more information, see Section 33.6.8, "Creating Databound Legends."

- **Value-Specific Rules**: Click to open the **Match Rules** and **Exception Rules** tables used to specify a finer detail for one or more data values for categorical groups in a data set.

  Note: Any match or exception rule specified in these tables will override the settings defined in the **Grouping Rules** table.

- **Match Rules**: Use to specify the style rule matched to one or more data values in a group of data in a data collection. Use the **Add** icon to add a row to the table for configuring a match rule for a categorical group and use the **Delete** icon to remove any row selected in the table. Each match rule is represented as a `attributeMatchRule` component, and assigned a unique, consecutively numbered Id, `amr1`, `amr2`, and so on. The property and property value is defined in a child `f:attribute` tag. For example:

```
<dvt:attributeMatchRule id="amr1" group="Mountain Dew">
  <f:attribute name="color" value="#ffff00"/>
</dvt:attributeMatchRule>
```

  For each row added to the table, enter the following:

- **Group Value**: Enter the exact value for the **Group by Value** attribute that will trigger the Match Rule to execute.
- **Property**: Use the dropdown list to select the property to use for styling that data value. Markers can be styled using **color**, **pattern**, **opacity**, **scaleX**, **scaleY**, or any combination of these valid values. The property selected here must match one of the property types listed in the **Marker Properties** for the attribute **Grouping Rules**.
- **Property Value**: Enter or use the dropdown list to assign a value to the property. If the value provided by the match override is also in the prebuilt ramp returned by the **Grouping Rules**, then that value will only be used by the overrides and will be skipped in the prebuilt ramp.

  Valid values for **color** are RGB hexidecimal colors.

  Valid values for **pattern** include a choice of twelve prebuilt patterns, for example, **smallChecker**, **largeDiamond**, **smallDiagonalRight**, **large-Crosshatch**. If fill color is specified, the pattern displays in that color on the default white background.

  Valid values for **opacity** range from **0.0** for transparent to **1.0** for opaque.

  Valid values for **scaleX** and **scaleY** are percentages that are then scaled to a float.

- **Exception Rules**: Use to specify one or more exceptions to the style rules for categorical groups in the data set. Use the **Add** icon to add a row to the table for configuring an exception rule and use the **Delete** icon to remove any row selected in the table. Each exception rule is represented as an `attributeExceptionRule` component, and assigned a unique, consecutively numbered Id, `aer1`, `aer2`, and so on. The property and property value is defined in a child `f:attribute` tag. For example:

```
<dvt:attributeExceptionRule id="aer1" condition="#{row.name=='TX'}"
               label="Texas">
  <f:attribute name="color" value="#ff00ff"/>
</dvt:attributeExceptionRule>
```

  For each row added to the table, enter the following:

  - **Condition**: Enter an EL expression, or use the dropdown list to open an Expression Builder dialog to create an EL expression that replaces the style property value with another when certain conditions are met. For example:

    ```
    #{row.Sales gt 100000}
    ```

  - **Property**: Use the dropdown list to select the property to use for styling that data value. Markers can be styled using **color**, **pattern**, **opacity**, **scaleX**, or **scaleY**. The property selected here must match one of the property types listed in the **Marker Properties** for the attribute **Grouping Rules**.
  - **Property Value**: Enter or use the dropdown list to assign a value to the property. If the value provided by the match override is also in the prebuilt ramp returned by the **Grouping Rules**, then that value will only be used by the overrides and will be skipped in the prebuilt ramp

    Valid values for **color** are RGB hexidecimal colors.

Valid values for **pattern** include a choice of twelve prebuilt patterns, for example, **smallChecker**, **largeDiamond**, **smallDiagonalRight**, **large-Crosshatch**. If fill color is specified, the pattern displays in that color on the default white background.

Valid values for **opacity** range from **0.0** for transparent to **1.0** for opaque.

Valid values for **scaleX** and **scaleY** are percentages ranging from **100** to **500**. The percentages are then scaled to a float, ranging from **1.0** to **5.0**.

– **Legend Label**: Enter a text resource to use for the legend label. The text resource can be a translatable string from a resource bundle or an EL expression executed at runtime. Use the dropdown list to open a Select Text Resource or Expression Builder dialog. If you need help, press **F1** or click **Help**.

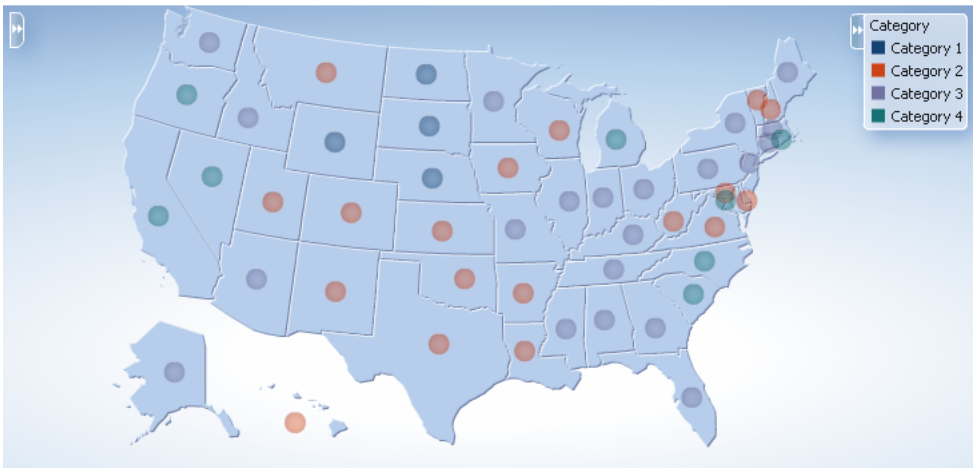
Note: the text resource option is only available for a fixed area. For row-varying areas, use an EL expression to retrieve a row-varying key to look up the text resource in a resource bundle, for example:

```
#{viewController.ResourceBundle[row.label]}
```

- **Messages**: Review and clear as necessary any alerts related to the configuration of the marker.

For example, you can use markers in an area data layer to display categorical groups of data using colors as illustrated in Figure 33–57.

Figure 33–57 Marker Attribute Groups by Color



Example 33–28 shows the sample code for marker attribute groups by color.

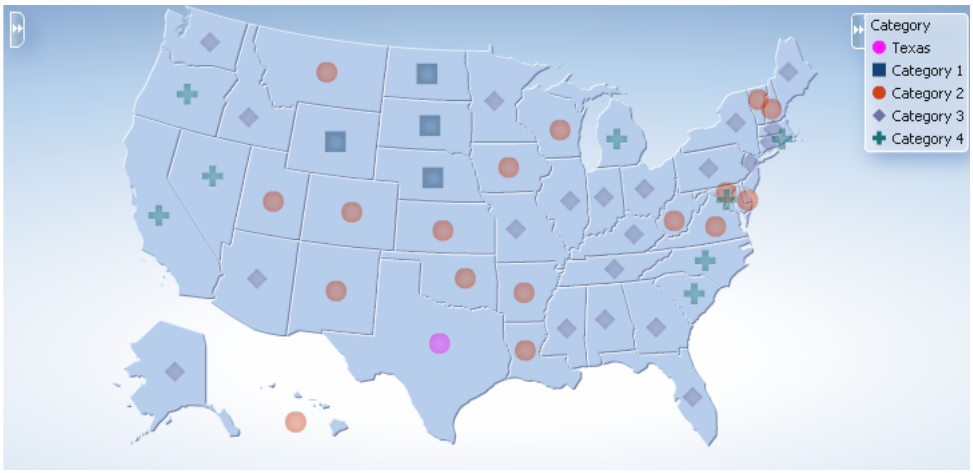
Example 33–28 Sample Code for Marker Attribute Groups by Color

```
<dvt:thematicMap basemap="usa" id="tm1"
  <dvt:areaLayer layer="states" id="al1" labelDisplay="off">
    <dvt:areaDataLayer id="dl1" var="row" value="#{stateData.colorModel}">
      <dvt:areaLocation id="al2" name="#{row.name}">
        <dvt:marker id="m1"
                    scaleX="3.0"
                    scaleY="3.0"
                    shape="circle">
```

```
          <dvt:attributeGroups id="ag1" type="color" value="#{row.category}"
                               label="#{row.category}"/>
        </dvt:marker>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
  <dvt:legend id="l1">
    <f:facet name="separator"/>
    <dvt:legendSection id="ls1" label="Category" source="ag1"/>
  </dvt:legend>
</dvt:thematicMap>
```

You can also use markers to display categorical groups of data using multiple attributes such as color and shape, and display an exception to the grouping rules when certain conditions are met. Figure 33–58 shows a thematic map with categorical groups using color and shape with an exception for the state of Texas.

Figure 33–58 Multiple Marker Attribute Groups with Exception Rule



Example 33–29 shows sample code for multiple marker attribute groups with exception rule.

Example 33–29 Sample Code for Multiple Marker Attribute Groups with Exception Rule

```
<dvt:thematicMap basemap="usa" id="tm2"
  <dvt:areaLayer layer="states" id="al1" labelDisplay="off">
    <dvt:areaDataLayer id="dl1" var="row" value="#{stateData.colorModel}">
      <dvt:areaLocation id="al2" name="#{row.name}">
        <dvt:marker id="m1"
               scaleX="3.0"
               scaleY="3.0"
               shape="circle">
          <dvt:attributeGroups id="ag1" type="shape color" value="#{row.category}"
               label="#{row.category}">
            <dvt:attributeExceptionRule id="aer1" condition="#{row.name=='TX'}"
               label="Texas">
              <f:attribute name="color" value="#ff00ff"/>
            </dvt:exceptionRule>
          </dvt:attributeGroups>
        </dvt:marker>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
```

```
  <dvt:legend id="l1">
    <f:facet name="separator"/>
    <dvt:legendSection id="ls1" label="Category" source="ag1"/>
  </dvt:legend>
</dvt:thematicMap>
```

33.6.7.3 What You May Need to Know About Styling Markers

Thematic maps support a predefined set of seven shapes (`circle`, `square`, and so on) that can be specified using the `shape` attribute in the `marker` component. For custom markers, the `shapePath` attribute can be used to specify the path of an SVG file that will get displayed in place of a predefined shape.

Marker shapes can be also specified through CSS style properties in an ADF skin. Using thematic map style properties, predefined marker shapes can be overwritten, and the paths to SVG files for custom markers can be defined without using the `shapePath` attribute. When using style properties, the `shape` attribute in the `marker` component is used for defining both predefined and custom shapes.

A predefined shape will be overwritten if a global or component-specific style property for that shape is specified in the ADF skin. For example, you can overwrite the predefined `circle` shape by specifying the `newCircle.svg` file in the thematic map component style property as follows:

```
af|dvt-thematicMap::shape-circle{
  -tr-path: url(/resources/path/newCircle.svg);
}
```

In the JSF page, the `marker` component `shape` attribute is set as follows:

```
<dvt:marker id="m1" shape="circle"/>
```

To specify a custom shape in the `marker` component `shape` attribute, you must use a prefix of `custom` in the shape style property name. For example, if the custom shape is named `customName`, then the ADF skin file should define either a global `.AFDVTShapeCustomName:alias` style property, or the thematic map specific `af|dvt-thematicMap::shape-customName` with the `-tr-path` property pointing to the SVG file as follows:

```
af|dvt-thematicMap::shape-customName{
  -tr-path: url(/resource/path/newCShape.svg);
}
```

In the JSF page, the marker component shape attribute is set as follows:

```
<dvt:marker id="m1" shape="customName"/>
```

For information about using ADF skins and style properties, see the "Customizing the Appearance Using Styles and Skins" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

33.6.7.4 What You May Need to Know About Default Style Values for Attribute Groups

The `attributeGroups` component is used to generate stylistic property values such as colors or shapes based on categorical grouping of the data in a data collection. Based on the attribute representing the column in the data model to group by, the `attributeGroups` component can generate style values for each unique value, or group, in the data.

The type of stylistic properties to generate values for is specified by the `type` attribute of the `attributeGroups` component. Supported types for `area` components are `color`, `pattern`, and `opacity`. Supported types for `marker` components are `color`, `shape`, `pattern`, `opacity`, `scaleX`, and `scaleY`. These types can be combined in a space-delimited list to generate multiple stylistic properties for each unique data value.

The default style values that are generated are defined using CSS style properties in the ADF skin. Each `attributeGroups` type has a default ramp defined in the ADF skin that can be customized by setting the index-based selectors to the desired values. Example 33–30 show sample code for using CSS style properties to specify attribute groups using CSS style properties.

Example 33–30 Attribute Groups CSS Style Properties

```
af|dvt-attributeGroups::shape1{
  -tr-shape: square;
}
af|dvt-attributeGroups::shape2{
  -tr-shape: square;
}
...
af|dvt-attributeGroups::color1{
  -tr-fill-color: #003366;
}
```

The default ramps for each attribute groups type are displayed in Table 33–6.

Table 33–6 Default Ramps for Thematic Map Attribute Groups

Type	Default Ramps
`color`	j`#003366` (blue), `#CC3300` (red), `#666699` (lavender), `#0006666` (emerald), `#FF9900` (orange yellow), `#993366` (purple), `#99CC33` (lime green), `#624390` (violet,), `#669933` (green), `#FFCC33` (yellow), `#006699` (turquoise blue), and `#EBEA79` (pale yellow).
`shape`	`square`, `circle`, `diamond`, `plus`, `triangleDown`, `triangleUp`, and `human`
`pattern`	`smallDiagonalLeft`, `smallDiagonalRight`, `smallTriangle`, `smallChecker`, `smallChecker`, `smallCrosshatch`, `smallDiamond`, `largeDiagonalLeft`, `largeDiagonalRight`, `largeTriangle`, `largeChecker`, `largeCrosshatch`, `largeDiamond`
`opacity`	`0.25`, `0.50`, `0.75`, `1.0`
`scaleX` and `scaleY`	`1.0`, `2.0`, `3.0`, `4.0`, `5.0`

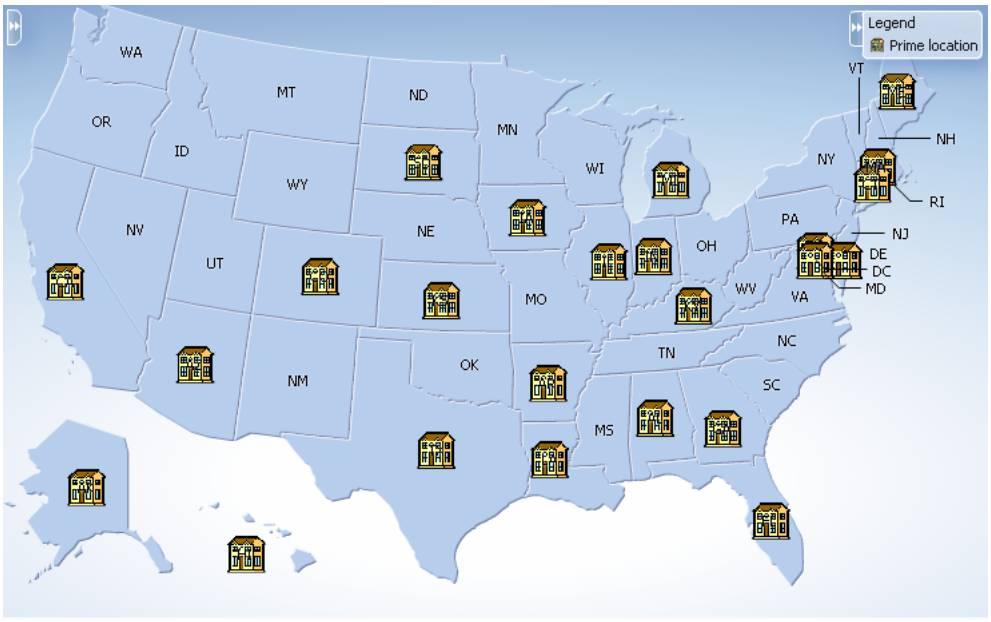
For information about using ADF skins and style properties, see the "Customizing the Appearance Using Styles and Skins" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

33.6.7.5 How to Style Images to Display Data

You can use images to style data displayed in a thematic map. Using stamping, each row of data in the data model can be identified by an image. Images can be associated with either an area or point data layer.

For example, in an area data layer you can use a house image to identify prime locations on the states map layer of a US base map as illustrated in Figure 33–59.

Figure 33–59 Thematic Map Styling Data with Images



Before you begin:

It may be helpful to have an understanding of how thematic map attributes and thematic map child tags can affect functionality. For more information, see the "Configuring Thematic Maps" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

You should already have a thematic map on your page. If you do not, follow the instructions in this chapter to create a thematic map. For more information, see Section 33.6.1, "How to Create a Thematic Map Using ADF Data Controls."

To use an image to display data:

1. In the Layer Browser, select the area or point data layer where you are configuring an image to display data.
2. In the Structure window, right-click the area or point data layer child `areaLocation` or `pointLocation` component and select **insert inside dvt:areaLocation** or **insert inside dvt:pointLocation > Image** to open the Insert Image dialog.
3. In the Insert Image dialog, enter the following:
   - **Source**: Enter the URI specifying the location of the image source. You can use the dropdown menu to choose **Edit** and open an Edit Property: Source dialog or the Expression Builder to specify the location of the image source.
   - **ShortDesc**: Enter the short description of the image used as the `alt` text for screen reader users.

Example 33–31 shows a code sample for using images to display data.

Example 33–31 Code Sample for Image Configuration

```
<dvt:thematicMap id="thematicMap" imageFormat="flash" basemap="usa"
                 summary="Thematic map showing the important real estate markets">
  <dvt:legend label="Legend">
    <dvt:legendSection source="areaLayer:dataLayer:img1"/>
```

```
  </dvt:legend>
  <dvt:areaLayer id="areaLayer" layer="states">
    <dvt:areaDataLayer id="dataLayer" contentDelivery="immediate"
                       value="#{tmapBean.colorModel}"
                       var="row"varStatus="rowStatus">
      <dvt:areaLocation id="dataLoc" name="#{row.name}">
        <af:image id="img1" source="/resources/images/geoMap/mansion.gif"
                            rendered="#{row.category == 'category1'}"
                            shortDesc="House image">
          <f:attribute name="legendLabel" value="Prime location"/>
        </af:image>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
</dvt:thematicMap>
```

33.6.7.6 What You May Need to Know About SVG Files

Scalable Vector Graphics (SVG) is the supported file format for creating custom shapes for thematic map markers.

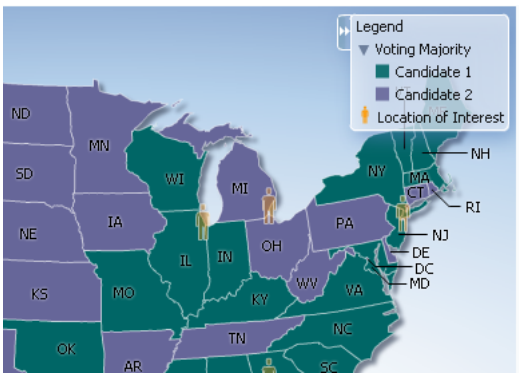
SVG features that are not supported by custom shapes include:

- Image tags within the SVG file. Everything must be declared using SVG's vector shapes.
- Pattern fills
- Gradients on strokes

33.6.8 Creating Databound Legends

Legends provide an explanatory table of the thematic map's styled data in symbol and label pairs. Thematic map legend components (`legend`) support symbols for color, shape, custom shape, fill pattern, opacity, images and size. One or more child legend item components (`legendSection`) are sourced from thematic map `area`, `marker`, `attributeGroups`, or `af:image` components stamped to style the data displayed in the map. The legend section structure supports control over content ordering and appearance. You can wrap legend items into a disclosable section using a `showLegendSection` component. Figure 33–60 shows a legend with a disclosable section for map areas and a marker.

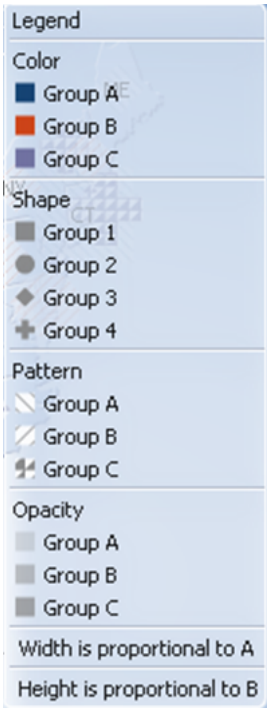
Figure 33–60 Legend with Disclosable Section and Marker



Legend items sourced from the `attributeGroups` component automatically split area or marker attribute types into different sections. You can specify a separator facet

to draw separators between legend sections. Figure 33–61 shows a legend with attribute groups for color, shape, fill pattern, opacity, and size with separators between each section.

Figure 33–61 Legend with Attribute Groups



Legends can be displayed in both Flash (default) and PNG image formats and both formats support bi-directional locales. When rendered in a PNG format, for example when printing the thematic map, disclosable sections in the legend are not supported, and legend items display as disclosed.

When you create a thematic map using the Data Controls panel and the thematic map binding dialogs, the legend data bindings are created for you. If you configure an area or marker as a default stamp across all areas in the thematic map, you can assign a static text resource to a fixed area or marker for the legend.

For default stamps displaying row-varying data, you can use an EL expression to assign the legend text and optionally, a managed bean to retrieve a row-varying key to look up the text resource in a resource bundle. Example 33–32 shows a code sample to generate legend entries for area and marker stamps. The code illustrates a disclosable section for a row-varying area and a fixed marker with an assigned text resource.

Example 33–32 Code Sample for Legend Area and Marker Stamps

```
<dvt:thematicMap id="tm1" basemap="usa" ...>
  <dvt:legend label="Legend">
    <dvt:showLegendGroupLabel label="Voting Majority">
      <dvt:legendSection id="ls1" source="a11:ad11:areaStamp"/>
    </dvt:showLegendGroupLabel>
    <dvt:legendSection id="ls2" source="a11:ad11:fixedMarker">
  </dvt:legend>
  <dvt:areaLayer id="a11" layer="states">
    <dvt:areaDataLayer id="ad11"...>
      <dvt:areaLocation id="aloc1" ,,,>
        <dvt:area id="areaStamp">
```

```
                fillColor="#{row.value > 50 ? tmapLegendBean,color1 :
                tmapLegendBean.color2}"
          <f:attribute name="legendLabel" value="#{row.value > 50 ?
                        'Candidate 2' : 'Candidate 1'}" />
        <dvt:marker id="fixedMarker" shape="human" fillColor=""#FF9900"
                                     scaleX="3" scaleY="3">
          <f:attribute name="legendLabel"
                       value="#{Bundle.Office_Locations}"/>
        </dvt:marker>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:area>
</dvt:thematicMap>
```

If you configure an area or marker to use an attributes group to specify the styling of categorical groups of data in a data collection, you can use an EL expression to assign the legend text and optionally, a managed bean to retrieve a row-varying key to look up the text resource in a resource bundle.

If you specify a match rule for the attributes group, the legend text is specified in the `group` attribute. If you specify an exception rule, you can specify a text resource from the application resource bundle. Example 33–33 shows sample code for a legend with attribute groups, including both a match rule and an exception rule, and specifying a separator between legend sections.

Example 33–33 Sample Code for Legend with Attribute Groups

```
<dvt:thematicMap id="tm1" basemap="usa" ...>
  <dvt:legend id="l1" label="Legend">
    <f:facet name="separator"
      <af:separator/>
    <dvt:legendSection id="ls1" source="al1:adl1:attributeGroupColor" />
    <dvt:legendSection id="ls2" source="al1:adl1:attributeGroupShape" />
    <dvt:legendSection id="ls3" source="al1:adl1:attributeGroupPattern" />
    <dvt:legendSection id="ls4" source="al1:adl1:attributeGroupOpacity" />>>
  </dvt:legend>
  <dvt:areaLayer id="al1" layer="states">
    <dvt:areaDataLayer id="adl1" value=" " var=" " ...>
      <dvt:areaLocation id="dataLoc" name="#{row.name}">
        <dvt:marker id="m1"... >
          <dvt:attributeGroups id="attributeGroupColor" type="color"
                 label="#{row.category1}" value="#{row.category1}" />
            <dvt:attributeMatchRule id="amrl" group="Mountain Dew">
              <f:attribute name="color" value="#ffff00"/>
          </dvt:attributeMatchRule>
          <dvt:attributeGroups id="attributeGroupShape" type="shape"
                 label="#{row.category2}" value="#{row.category2}" />
            <dvt:attributeExceptionRule id="aer1" condition="#{row.name=='TX'}"
                 label="#{viewcontroller.Texas}">
              <f:attribute name="shape" value="human"/>
            </dvt:exceptionRule>
          <dvt:attributeGroups id="attributeGroupShape" type="pattern"
                 label="#{row.category3}" ... />
          <dvt:attributeGroups id="attributeGroupShape" type="opacity"
                 label="#{row.category4}" ... />
        </dvt:marker>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
</dvt:thematicMap>
```

You configure thematic map legend areas or markers in a custom region in the same way as any other map layer. shows sample code for a custom map layer legend.

Example 33–34 Sample Code for Legend for Custom Map Layer

```
<dvt:thematicMap>
  <dvt:legend>
    <dvt:legendSection label="Sales Regions" source="customAreaStamp"/>
      </dvt:legend>
      ...
  <dvt:customAreaLayer id="crl1">
  ...
  </dvt:customAreaLayer>
  <dvt:areaLayer layer="crl1">
    <dvt:areaDataLayer var="row">
      <dvt:areaLocation name="#{row.name}" id="al1">
        <dvt:area id="customAreaStamp" fillColor="#{row.color}">
          <f:attribute name="legendLabel" value="#{row.name}"/>
        </dvt:area>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
</dvt:thematicMap
```

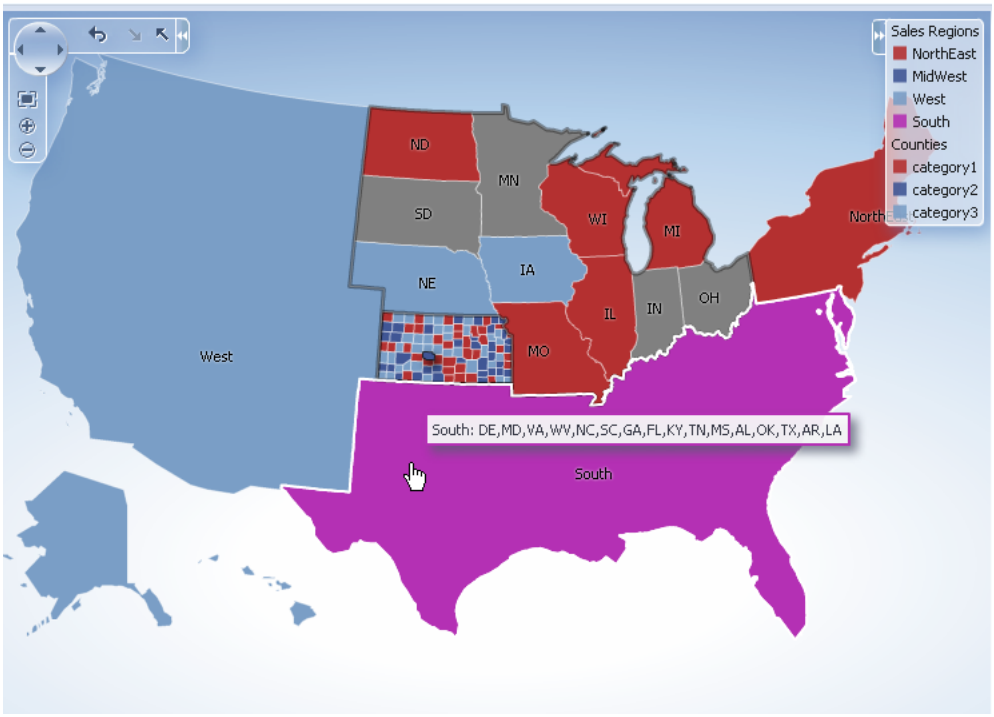
You can customize the appearance of thematic map legends using ADF skins. For more information, see the "Customizing the Appearance Using Styles and Skins" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*

33.6.9 How to Define a Custom Map Layer

You can define a custom map layer from your own regional data and insert it into the natural geographical hierarchy of a thematic map. The custom layer is created by extending a predefined map layer and aggregating the lower level regions to form the new regions in the custom layer. After defining a custom map layer, it is used in the same way as any other map layer.

For example, you could define geographical regions for the NorthEast, Midwest, West and South in the United States in a US Regions custom map layer as illustrated in Figure 33–62. In the figure, the US Regions custom layer is extended from the states layer in the US base map. The new areas in the layer are aggregated from the states in the states layer. The label for the South US Region lists the states in that region. The MidWest US Region is drilled down to display the sales categories in the lower level counties region. For more information about drilling, see Section 33.6.5, "How to Configure Drilling in Thematic Maps."

Figure 33–62 US Regions Custom Map Layer



The `customAreaLayer` component uses a model to access the data in the underlying list. The specific model class is `oracle.adf.view.rich.model.CollectionModel`. You can also use other model instances, for example, `java.util.List`, `java.util.Array`, and `javax.faces.model.DataModel`. The `customAreaLayer` will automatically convert the instance into a `CollectionModel`.

Before you begin:

It may be helpful to have an understanding of how thematic map attributes and thematic map child tags can affect functionality. For more information, see the "Configuring Thematic Maps" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

You should already have an ADF data control or ADF managed bean that defines the aggregated areas in the predefined base map you are extending.

You should already have a thematic map on your page. If you do not, follow the instructions in this chapter to create a thematic map. For more information, see Section 33.6.1, "How to Create a Thematic Map Using ADF Data Controls."

How to add and configure a custom map layer:

1. Select the thematic map in the Visual Editor.
2. In the Layer Browser, from the **Add** icon dropdown list, choose **Add Custom Layer** to open the Create Custom Layer dialog. If the Layer Browser is not open in the Visual Editor, right-click in the map and choose **Open Layer Browser**.
3. In the Create Custom Layer dialog, enter the following:
   - **Bind Data Now**: Select and click **Browse** to open the Picker dialog > Data Controls Definitions page. Select the data collection in the ADF data controls you are using to data bind your custom layer and areas.

Note: Alternatively, you can use the Expression Builder page to select an ADF managed bean you are using to databind your custom layer and areas. You can also use the Expression Builder dialog in each remaining field.

- **Layer Id**: Enter a unique identifier for the `customAreaLayer` component. For example, if you divide the US into aggregate regions (Northeast, Midwest, West, and South), then you might define them with corresponding Ids (NE, MW, W, and S).
- **Extends**: Use the search icon to display the built-in map layers that can be used to aggregate areas for the custom layer. Select the map layer that the custom layer will extend.
- **Area List**: Use the dropdown list to select the data collection attribute representing the list of lower level map regions that are used to aggregate the areas in the custom map layer. The comma separated list of values aggregate the regions in the may layer defined in the in the **Extends** attribute.
- **Area Id**: Use the dropdown list to select the data collection attribute representing the unique identifier of lower level map regions that are used to aggregate the areas in the custom map layer. By default, a unique identifier, `ca1, ca2`, and so on, is used in the for the `customArea` component.
- **Area Label**: Use the dropdown list to select the data collection attribute representing the names of lower level map regions that are used to aggregate the areas in the custom map layer. At runtime the label will display the comma separated list of the aggregated regions.

Figure 33–63 shows the completed Create Custom Layer dialog.

Figure 33–63 Create Custom Layer Dialog

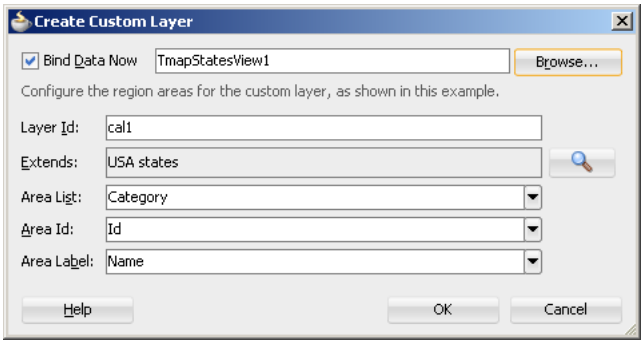
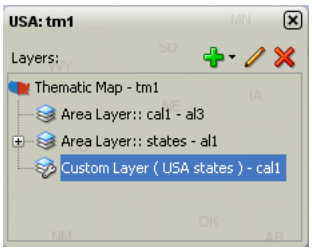


Figure 33–64 shows the Layer Browser after defining a custom layer. In the layer structure, the custom layer `ca11` is referenced in the map layer `a11`, where you add area or point data layers to display data. For more information, see Section 33.6.4, "How to Add Data Layers to Thematic Maps."

Figure 33–64 Custom Map Layer in Layer Browser



Example 33–35 shows the code inserted in the JSF page

Example 33–35 Custom Map Layer Code

```
<dvt:thematicMap>
...
  <dvt:areaLayer layer="states" id="al1"/>
  <dvt:areaLayer layer="cal1" id="al3"/>
  <dvt:customAreaLayer id="cal1"
                       value="#{bindings.TmapStatesView.collectionModel}"
                       var="row"
                       extendsLayer="states">
    <dvt:customArea areaId="#{row.RowID}" areaList="#{row.RowID}"
                    label="#{row.RowID}" id="ca1"/>
  </dvt:customAreaLayer>
...
 </dvt:thematicMap>
```

After configuring and adding a custom layer to the map layer hierarchy, you can then use the map layer in the same way as any other map layer. For example, to create the thematic map illustrated in Figure 33–62, you will need to do the following:

- Configure the thematic map to support drilling. For more information, see Section 33.6.5, "How to Configure Drilling in Thematic Maps."

  Note: If drilling is enabled for the thematic map, drilling between a custom layer and the map layer used to aggregate the custom layer is available without configuring data display for either layer.

- Add and configure an area data layer for the county data view. For more information, see Section 33.6.4, "How to Add Data Layers to Thematic Maps."
- Add and style area components to display the related data for each map layer. For more information, see Section 33.6.7, "Styling Areas, Markers, and Images to Display Data."
- Configure the thematic map legend. For more information, see Section 33.6.8, "Creating Databound Legends."

Example 33–36 shows sample code for using a custom layer in a thematic map.

Example 33–36 Sample Code for Thematic Map with Custom Layer

```
<dvt:thematicMap id="thematicMap" imageFormat="flash" basemap="usa" drilling="on"
                 maintainDrill="true"
                 controlPanelBehavior="initExpanded" summary="US Custom Regions">
  <dvt:customAreaLayer id="crl1" value="#{tmapRegions.collectionModel}" var="row"
                       varStatus="rowStatus"
```

```
                    extendsLayer="states">
    <dvt:customArea areaId="#{row.name}" label="#{row.name}"
                    areaList="#{row.regions}" id="ca1"/>
  </dvt:customAreaLayer>
  <dvt:areaLayer id="custom" layer="crl1">
    <dvt:areaDataLayer contentDelivery="immediate"
                    value="#{tmapRegions.collectionModel}"
                    selectionMode="single"
                    var="row" varStatus="rowStatus" id="adl1">
      <dvt:areaLocation name="#{row.name}" id="al1">
        <dvt:area fillColor="#{row.color}" shortDesc="#{row.regions}"
                  id="a1" value="#{row.name}"/>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
  <dvt:areaLayer id="areaLayerS" layer="states"
    <dvt:areaDataLayer id="dataLayerS" selectionMode="multiple"
                    contentDelivery="immediate"
                    value="#{tmapStates.collectionModel}" var="row"
                    varStatus="rowStatus">
      <dvt:areaLocation id="areaLocS" name="#{row.name}">
        <dvt:area id="area1S" fillColor="#{row.color}"></dvt:area>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
  <dvt:areaLayer id="areaLayer" layer="counties">
    <dvt:areaDataLayer id="dataLayer" selectionMode="single"
                    contentDelivery="immediate"
                    value="#{tmapCounty.collectionModel}" var="row"
                    varStatus="rowStatus">
      <dvt:areaLocation id="areaLoc" name="#{row.name}">
        <dvt:area id="area1" fillColor="#{row.color}"
                             value="#{row.category}"></dvt:area>
      </dvt:areaLocation>
    </dvt:areaDataLayer>
  </dvt:areaLayer>
  <dvt:legend label="Sales Regions" id="l1">
    <dvt:legendSection source="custom:adl1:a1" id="ls1"/>
    <dvt:legendSection label="Counties" source="areaLayer:dataLayer:areaLoc"
                    id="ls3"/>
  </dvt:legend>
</dvt:thematicMap>
```

33.7 Creating Databound Gantt Charts

A Gantt chart is a type of bar graph (with time on the horizontal axis). It is used in planning and tracking projects to show tasks or resources in a time frame with a distinct beginning and end.

When you create a Gantt chart, you can choose from the following types:

- Project

  A project Gantt chart lists tasks vertically and shows the duration of each task as a bar on a horizontal time line.

- Resource Utilization

A resource utilization Gantt chart shows graphically whether resources are over or under allocated. It shows resources vertically while showing their allocation and, optionally, capacity on the horizontal time axis.

- Scheduling

  A scheduling Gantt chart is based on manual scheduling boards and shows resources vertically with corresponding activities on the horizontal time axis. Examples of resources include people, machines, or rooms.

For information about customizing Gantt chart charts after data-binding is completed, see the "Using ADF Gantt Chart Components" chapter in *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

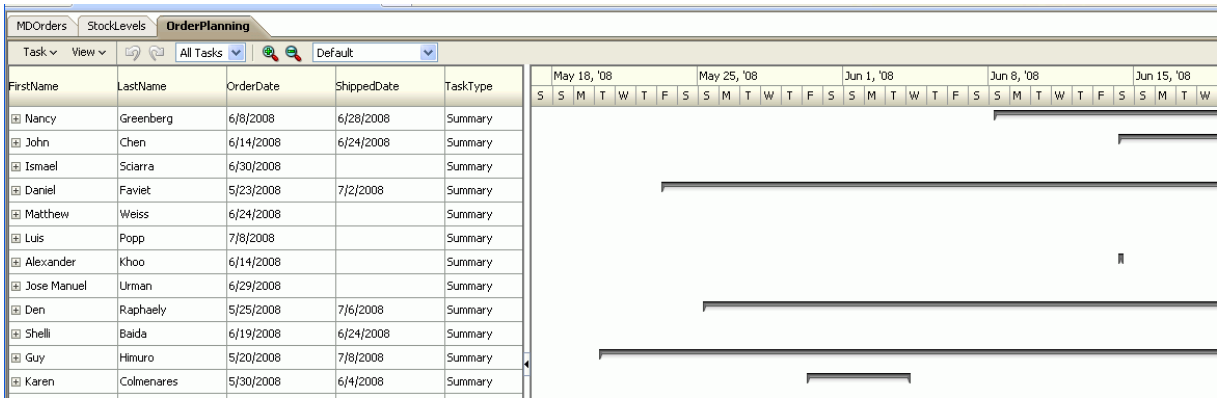
33.7.1 How to Create a Databound Project Gantt Chart

For a project Gantt chart, you must specify values for tasks. Optionally, you can specify values for split tasks, subtasks, recurring tasks, and dependencies between tasks, if your data collection has accessors for this additional information.

The project Gantt chart is displayed with default values for overall start time and end time and for the major and minor time axis values. In a project Gantt chart, the setting for the major time axis defaults to weeks and the setting for the minor time axis defaults to days.

Figure 33–65 shows a project Gantt chart in which each task is an order to be filled. The list region on the left side of the splitter shows columns with the name of the person who is responsible for the order and columns for the order date and shipped date. In the chart region on the right side of the splitter, the Gantt chart displays a horizontal bar from the order date to the ship date for each order.

Figure 33–65 The Order Shipping Project Gantt Chart



To create a project Gantt chart using a data control, you bind the project Gantt chart component to a data collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel.

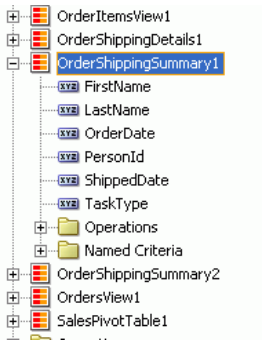
> **Tip:** You can also create a project Gantt chart by dragging a project Gantt chart component from the Component Palette and completing the Create Project Gantt dialog. This approach allows you to design the Gantt chart user interface before binding the component to data.

To create a databound project Gantt chart:

1. From the Data Controls panel, select a data collection. For a Gantt chart, you can select a row set collection (which produces a flat list of tasks) or a basic tree collection (which produces a hierarchical list of tasks).

   Figure 33–66 shows an example where you could select the `OrderShippingSummary1` collection in the Data Controls panel to create a project Gantt chart that displays the progress of order shipping.

Figure 33–66 Data Collection for Shipping Orders



2. Drag the collection onto a JSF page and, from the context menu, choose **Gantt** > **Project**.

3. In the ensuing Create Project Gantt dialog, you do the following to connect task-related controls in the pages at the top of the dialog with corresponding columns in the data collection:

   a. In the Tasks page at the top of the dialog, you select the columns in the data collection that correspond to each of the following controls: **Task Id**, **Start Time**, and **End Time**. By clicking **Show More** in the dialog, you can select additional columns in the data collection that correspond to the following controls: **Actual Start**, **Actual End**, **% Complete**, **Completed Through**, **Critical**, and Is **Container**.

      Optionally, you can select a column in the data collection to map to the task type. If you do not bind **Task Type** to a column in your data collection, then all tasks default to **Normal**. The task type controls the appearance of the task bar when it is rendered in the Gantt chart.

      A project Gantt chart component supports the following task types that you configure the data collection to return: Summary, Normal, and Milestone.

   b. If the data collection has an accessor for subtasks, you have the option of using the Subtasks page in the dialog to select the subtasks accessor and to select the columns in the data collection that correspond to each of the following controls: **SubTask Id**, **Start Time**, and **End Time**. Optionally, you can select a column in the data collection to map to the subtask type. If you do not bind **SubTask Type** to data, then all subtasks default to **Normal**.

      A project Gantt chart component supports the following task types that you configure the data collection to return: Summary, Normal, and Milestone.

      If you do not bind subtasks, then the Gantt chart cannot render a hierarchy view of tasks. If you bind subtasks, then you can drill from tasks to subtasks in the hierarchy view of the Gantt chart.

c. If the data collection has an accessor for dependent tasks, you have the option of using the **Dependent Tasks** page in the dialog to select the dependent task accessor and to select the columns in the data collection that correspond to the following controls: **Dependency Type**, **From Task Id**, and **To Task Id**.

   Dependent tasks are linked by their dependency between their finish and start times.

   A project Gantt chart component supports the following dependency types that you configure the data collection to return: `fs` (for finish to start), `ss` (for start to start), `ff` (for finish to finish), and `sf` (for start to finish).

d. If the data collection has an accessor for split tasks, you have the option of using the Split Tasks page in the dialog to select **Split Tasks** accessor and to select the columns in that data collection that correspond to each of the following controls: **SplitTask Id**, **Start Time**, and **End Time**.

e. If the data collection has an accessor for recurring tasks, you have the option of using the **Recurring Tasks** page in the dialog to select the recurring tasks accessor and to select the columns in that data collection that correspond to each of the following controls: **Recurring Task Id**, **Type**, **Start Time**, and **End Time**.

f. You can use the **Appearance** page in the dialog to specify the attributes that correspond to the **Label** of the task bar, and up to three icons to associate with the task bar.

4. In the **Table Columns** section, you specify the columns to appear in the list region of the Gantt chart. Specify one row of information for each column that is to appear. Use the **New** icon to add new rows. Use the arrow icons to arrange the rows in the exact sequence that you want the columns to appear in the Gantt chart list.

   Note: The first row that you specify in the **Table Columns** section designates the nodestamp column for the list region. The nodestamp column is the one that you can expand or collapse when you have a subtask collection.

   For each row, you provide the following specifications:

   - **Display Label**: Select the values for the headers of the columns in the Gantt chart list. If you select **<default>**, then the text for the header is automatically retrieved from the data binding.
   - **Value Binding**: Select the columns in the data collection to use for the columns in the Gantt chart list. The available values are the same as those for the tasks group.
   - **Component to Use**: Select the type of component to display in the cell of the Gantt chart list. The default is the ADF Output Text component.

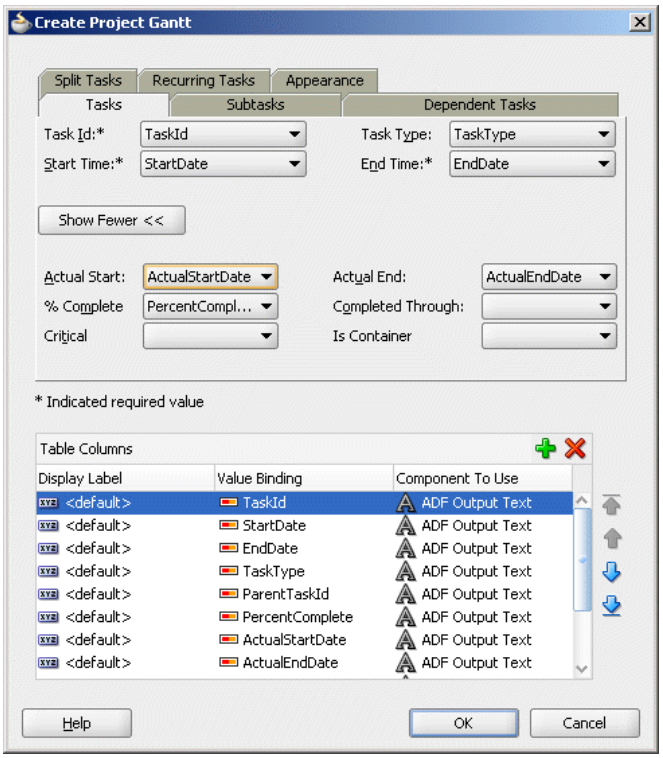
5. Click **OK**.

6. If you want to include a legend in the Gantt chart, right-click the project Gantt chart node in the Structure window and choose **Insert inside dvt:projectGantt** > **Legend**.

   The legend shows information about each symbol and color coded bar that is used to represent different task types. It also shows detailed information about the task that is selected in the Gantt chart.

Figure 33–67 shows the dialog used to create the project Gantt dialog from the data collection for shipping orders.

Figure 33–67 Create Project Gantt Dialog for Orders Shipped



After completing the data binding dialog, you can use the Property Inspector to specify values for additional attributes for the project Gantt chart.

33.7.2 What Happens When You Create a Project Gantt Chart from a Data Control

Dropping a project Gantt chart from the Data Controls panel has the following effect:

- Creates the bindings for the Gantt chart and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

Example 33–37 displays the row set bindings that were generated for the project Gantt chart that displays orders shipped. This code example shows that there are nodes defined for tasks and subtasks, along with attributes. There are also nodes defined for dependent tasks, split tasks, and recurring tasks but no attributes.

Example 33–37 Bindings for a Project Gantt Chart

```
<bindings>
  <gantt IterBinding="OrderShippingSummary2Iterator" id="GanttProjectView1"
         xmlns="http://xmlns.oracle.com/adfm/dvt">
    <ganttDataMap>
      <nodeDefinition DefName="oracle.fod.model.OrderShippingSummary"
                      type="Tasks">
        <AttrNames>
          <Item Value="TaskId" type="taskId"/>
          <Item Value="StartDate" type="startTime"/>
          <Item Value="EndDate" type="endTime"/>
```

```
          <Item Value="TaskType" type="taskType"/>
        </AttrNames>
      </nodeDefinition>
      <nodeDefinition type="subTasks">
        <AttrNames/>
      </nodeDefinition>
      <nodeDefinition type="Dependents">
        <AttrNames/>
      </nodeDefinition>
      <nodeDefinition type="SplitTasks">
        <AttrNames/>
      </nodeDefinition>
      <nodeDefinition type="RecurringTasks">
        <AttrNames/>
      </nodeDefinition>
  </ganttDataMap>
 </gantt>
</bindings>
```

Example 33–38 shows the code generated on the JSF page for the ADF project Gantt chart. This tag code contains settings for the overall start and end time for the project Gantt chart. It also shows the default time axis settings for the major axis (in weeks) and the minor axis (in days). Finally, it lists the specifications for each column that appears in the list region of the Gantt chart. For brevity, the code in the `af:column` elements for `OrderStatusCode` and `ShippedDate` has been omitted.

Example 33–38 Code on the JSF Page for a Project Gantt Chart

```
<dvt:projectGantt id="projectGantt1"
                  value="#{bindings.OrderShippingSummary2.projectGanttModel}"
                  var="row"
                  startTime="2011-04-15" endTime="2011-07-14">
  <f:facet name="major">
    <dvt:timeAxis scale="weeks" id="ta1"/>
  </f:facet>
  <f:facet name="minor">
    <dvt:timeAxis scale="days" id="ta2"/>
  </f:facet>
  <f:facet name="nodeStamp">
    <af:column
        sortProperty="#{bindings.OrderShippingSummary2.hints.FirstName.name}"
        sortable="false"
        headerText="#{bindings.OrderShippingSummary2.hints.FirstName.label}"
        id="c1">
      <af:outputText value="#{row.FirstName}" id="ot1"/>
    </af:column>
  </f:facet>
    <af:column sortProperty="LastName" sortable="false"
             headerText="#{bindings.OrderShippingSummary2.hints.LastName.label}">
      <af:outputText value="#{row.LastName}"/>
    </af:column>
    <af:column sortProperty="OrderDate" sortable="false"
             headerText="#{bindings.OrderShippingSummary2.hints.OrderDate.label}">
       <af:outputText value="#{row.OrderDate}">
         <af:convertDateTime
             pattern="#{bindings.OrderShippingSummary2.hints.OrderDate.format}"/>
       </af:outputText>
     </af:column>
     <af:column sortProperty="ShippedDate" sortable="false"
         headerText="#{bindings.OrderShippingSummary2.hints.ShippedDate.label}">
```

```
      <af:outputText value="#{row.ShippedDate}">
      <af:convertDateTime
        pattern="#{bindings.OrderShippingSummary2.hints.ShippedDate.format}"/>
      </af:outputText>
    </af:column>
    <af:column sortProperty="TaskType" sortable="false"
        headerText="#{bindings.OrderShippingSummary2.hints.TaskType.label}">
      <af:outputText value="#{row.TaskType}"/>
    </af:column>
</projectGantt>
```

33.7.3 What You May Need to Know About Summary Tasks in a Project Gantt Chart

A summary task shows start time and end time for a group of tasks (which are usually subtasks). A summary task cannot be moved or extended. However, your application should recalculate the summary task times when the dates of any subtask changes.

To detect a change in task duration, register an event handler by specifying a method binding expression on the `DataChangeListener` attribute of the Gantt chart component. When an action occurs that potentially changes data in the Gantt chart, the event fired is of type `oracle.adf.view.faces.bi.event.DataChangeEvent`. This event contains information about the data changes and is fired to the registered event listener. The listener is responsible for validating the new values and updating the underlying data model.

When the update is completed, the Gantt chart is refreshed with either the old data (if the update failed) or with the new data. The Gantt chart uses partial page rendering so that only the Gantt chart and not the entire page is refreshed.

33.7.4 What You May Need to Know About Percent Complete in a Project Gantt Chart

Percent complete can be represented as an inner bar that indicates what percentage of a task is complete. The length of the inner bar is calculated based on percent complete returned by the data object.

The data binding dialog for the project Gantt chart does not provide a control in which you can enter the percentage complete value, but this value is required to render a percent complete. However, the Gantt chart data object does contain a `percentComplete` attribute.

To provide the percent complete integer, you must add a new attribute mapping in the `nodeDefinition` for type `Tasks`. For example, if your data collection has a column named PercentDone, then the attribute mapping would appear as follows: `<Item Value="PercentDone" type="percentComplete"/>`.

Example 33–39 shows the percent complete attribute mapping added to the data binding code for the `nodeDefinition` of type `Tasks` in the project Gantt chart binding displayed in Example 33–37.

Example 33–39 Bindings for Project Gantt Chart with Percent Complete

```
<bindings>
  <nodeDefinition DefName="oracle.fod.model.OrderShippingSummary"
                  type="Tasks">
    <AttrNames>
      <Item Value="PersonId" type="taskId"/>
      <Item Value="OrderDate" type="startTime"/>
      <Item Value="TaskType" type="taskType"/>
      <Item Value="ShippedDate" type="endTime"/>
```

```
      <Item Value="PercentDone" type="percentComplete"/>
    </AttrNames>
  </nodeDefinition>
</bindings>
```

Another attribute (`completedThrough`) exists that allows you to specify a date rather than a percentage. The Gantt chart data object calculates the percentage complete based on the date that the `completedThrough` attribute references. For example, if your data collection has a column named `PercentDone`, then the attribute mapping would appear as follows: `<Item Value="PercentDone" type="completedThrough"/>`.

33.7.5 What You May Need to Know About Variance in a Project Gantt Chart

Variance can be rendered within two horizontal bars. One bar represents the base (or original) start and end time for the task. The second bar represents the actual start and end time for the task. You enter the binding information for the base start time and end time in the data binding dialog for a project Gantt chart.

The data binding dialog for Gantt chart does not provide controls in which you can enter the actual start time and actual end time for the Gantt chart, but these values are required to render variance. However, the Gantt chart data object does contain the following attributes: `actualStart` and `actualEnd`.

To provide the actual start and actual end time, you must add two new attribute mappings in the `nodeDefinition` for type `Tasks`. For example, if your data collection has columns named `ActualStartDate` and `ActualEndDate`, then the attribute mappings would appear as shown in Example 33–40.

Example 33–40 Attribute Mappings for Actual Start and Actual End

```
<Item Value="ActualStartDate" type="actualStart"/>
<Item Value="ActualEndDate" type="actualEnd"/>
```

Example 33–41 shows the actual start and actual end attribute mappings added to the data binding code for the `nodeDefinition` of type `Tasks` for a project Gantt chart.

Example 33–41 Bindings for Project Gantt Chart with Actual Start and Actual End

```
<nodeDefinition DefName="oracle.fod.model.OrderShippingSummary"
                type="Tasks">
  <AttrNames>
    <Item Value="PersonId" type="taskId"/>
    <Item Value="OrderDate" type="startTime"/>
    <Item Value="TaskType" type="taskType"/>
    <Item Value="ShippedDate" type="endTime"/>
    <Item Value="ActualStartDate" type="actualStart"/>
    <Item Value="ActualEndDate" type="actualEnd"/>
  </AttrNames>
</nodeDefinition>
```

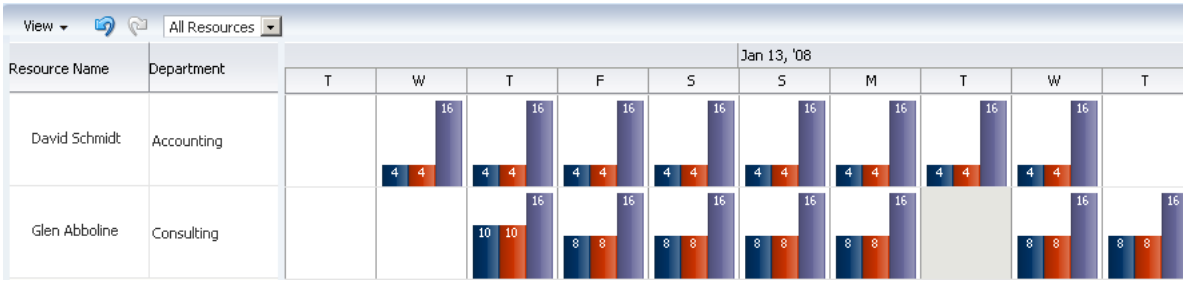
33.7.6 How to Create a Databound Resource Utilization Gantt Chart

For a resource utilization Gantt chart, you must supply identification for resources, identification for time, and start and end times for resource usage. Optionally, you can provide data values for subresources.

The resource utilization Gantt chart is displayed with default values for the major and minor time axis values. In a resource utilization Gantt chart, the setting for the major time axis defaults to weeks and the setting for the minor time axis defaults to days.

Figure 33–68 shows a resource utilization Gantt chart that lists each resource and an associated calendar that can display when the resource is in use.

Figure 33–68 Resource Utilization Gantt Chart



To create a resource utilization Gantt chart using a data control, you bind the resource utilization component to a data collection. JDeveloper allows you to do this declaratively by dragging a collection from the Data Controls panel and dropping it on a JSF page.

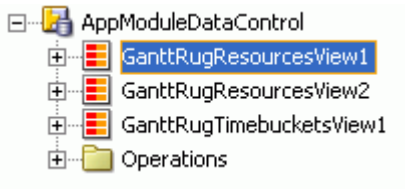
> **Tip:** You can also create a resource utilization Gantt chart by dragging a resource utilization Gantt chart component from the Component Palette and completing the Create Resource Utilization Gantt dialog. This approach gives you the option of designing the Gantt chart user interface before binding the component to data.

To create a resource utilization Gantt chart:

1. From the Data Controls panel, select a data collection. For a Gantt chart, you can select a row set collection or a basic tree collection.

   Figure 33–69 shows an example where you could select the `GanttRugResourcesView1` collection in the Data Controls panel to create a resource utilization Gantt chart to display the usage of a resource.

Figure 33–69 Data Collection for Resource Utilization



2. Drag the collection onto a JSF page and, from the context menu, choose **Gantt** > **Resource Utilization**.
3. In the Create Resource Utilization Gantt dialog, connect resource- and time-related controls with corresponding columns in the data collection.
   a. For **Resource Id**, select the column in the data collection that corresponds to the unique identifier of the resource.

b. In the **Time Buckets** page, select a value from the **Bucket Accessor** dropdown list that contains the time buckets assigned to the resource and select a value from the **Bucket Date** dropdown list that corresponds to a unit of time.

c. In the **Bucket Metrics** list, you can optionally specify attributes that appear as bars within the time bucket. Each attribute that you specify in the **Bucket Metrics** list must be of type Number as the value of the attribute is used to calculate the height of the bar.

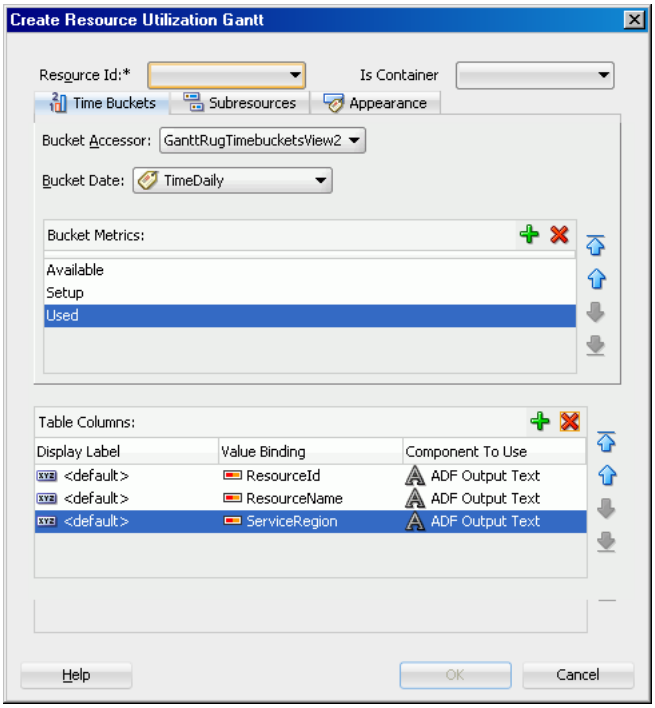
d. In the **Table Columns** list, specify the column(s) to appear in the list region of the Gantt chart resource utilization on the left side of the splitter. Specify one row of information for each column that is to appear. Use the New icon to add new rows. Use the arrow icon to arrange the rows in the exact sequence that you want the columns to appear in the resource utilization list. For each row, provide values for **Display Label**, **Value Binding**, and **Component to Use**.

e. If the data collection has an accessor for subresources, you can use the **Subresources** page to select a subresources accessor and a resource ID.

4. Click **OK**.

Figure 33–70 shows the dialog used to create a resource utilization Gantt chart from the data collection for resources available for a project.

Figure 33–70 Create Resource Utilization Gantt Chart



33.7.7 What Happens When You Create a Resource Utilization Gantt Chart

Dropping a resource utilization Gantt chart from the Data Controls panel onto a JSF page has the following effects:

- Creates bindings for the resource utilization Gantt chart and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

Example 33–42 shows the row set bindings that were generated for the resource utilization Gantt chart illustrated in Figure 33–70.

Example 33–42 Binding XML for a Resource Utilization Gantt Chart

```
<gantt IterBinding="GanttRugResourcesView2Iterator"
       id="GanttRugResourcesView2"
       xmlns="http://xmlns.oracle.com/adfm/dvt">
  <ganttDataMap>
    <nodeDefinition DefName="model.GanttRugResourcesView" type="Resources">
      <AttrNames>
        <Item Value="ResourceId" type="resourceId"/>
    </AttrNames>
  <Accessors>
    <Item Value="GanttRugTimebucketsView2" type="timeBuckets"/>
  </Accessors>
    </nodeDefinition>
    <nodeDefinition type="TimeBuckets"
                    DefName="model.GanttRugTimebucketsView">
      <AttrNames>
        <Item Value="TimeDaily" type="time"/>
        <Item type="metric" Value="Available"/>
        <Item type="metric" Value="Setup"/>
        <Item type="metric" Value="Used"/>
      </AttrNames>
    </nodeDefinition>
    <nodeDefinition type="Subresources">
      <AttrNames/>
    </nodeDefinition>
  </ganttDataMap>
</gantt>
```

Example 33–43 shows the code generated on the JSF page for the resource utilization Gantt chart. This tag code contains settings for the overall start and end time for the resource utilization Gantt chart. These settings have to be edited manually. The code also shows the time axis settings for the major time axis (in weeks) and the minor time axis (in days). Finally, it lists the specifications for each column to appear in the list region of the resource utilization Gantt chart.

Example 33–43 Code on the JSF Page for a Resource Utilization Gantt Chart

```
<dvt:resourceUtilizationGantt id="resourceUtilizationGantt1"
                              value="#{bindings.GanttRugResourcesView2.
                                  resourceUtilizationGanttModel}"
                              var="row"
                              metrics="#{bindings.GanttRugResourcesView2.metrics}"
                              taskbarFormatManager="#{bindings.
                                  GanttRugResourcesView2.
                                  resourceUtilizationGanttTaskbarFormatManager}"
                              startTime="2011-02-03"
                              endTime="2011-02-29">
  <f:facet name="major">
    <dvt:timeAxis scale="weeks"/>
  </f:facet>
  <f:facet name="minor">
    <dvt:timeAxis scale="days"/>
  </f:facet>
  <f:facet name="nodeStamp">
    <af:column sortProperty="ResourceId" sortable="false"
```

```
            headerText="#{bindings.GanttRugResourcesView2.hints.ResourceId.label}">
        <af:outputText value="#{row.ResourceId}"/>
      </af:column>
    </f:facet>
    <af:column sortProperty="ResourceName" sortable="false"
          headerText="#{bindings.GanttRugResourcesView2.hints.ResourceName.label}">
      <af:outputText value="#{row.ResourceName}"/>
    </af:column>
    <af:column sortProperty="ServiceRegion" sortable="false"
          headerText="#{bindings.GanttRugResourcesView2.hints.ServiceRegion.label}">
      <af:outputText value="#{row.ServiceRegion}"/>
    </af:column>
  </dvt:resourceUtilizationGantt>
```

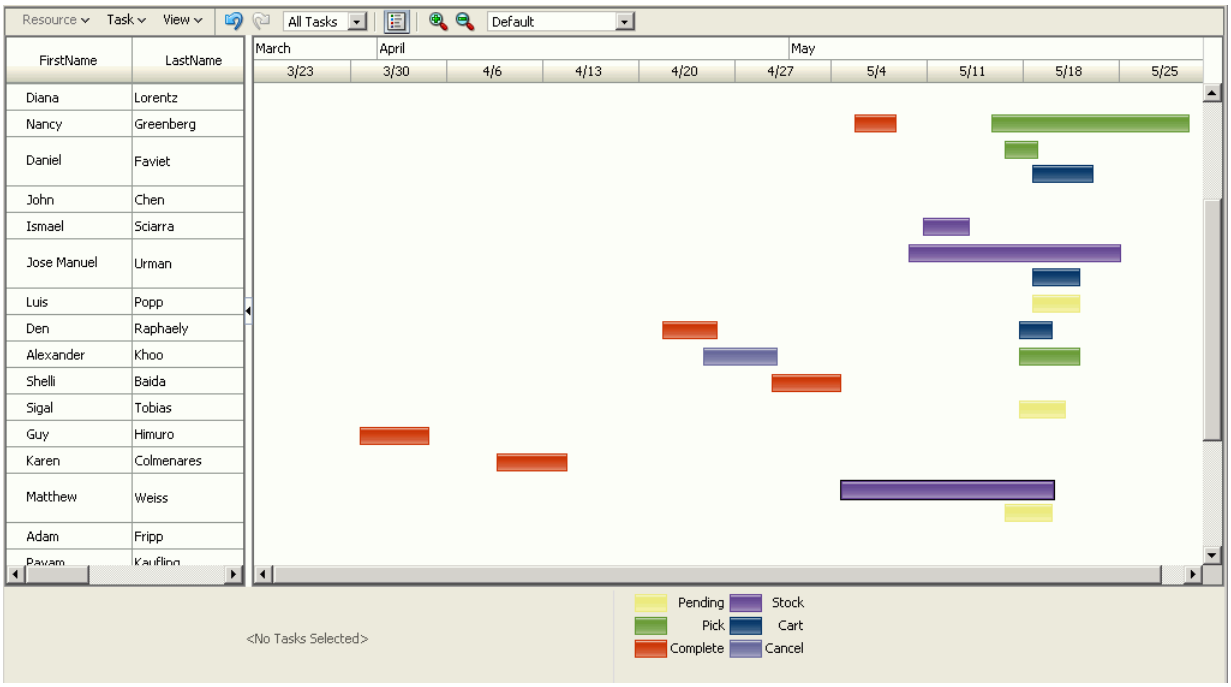
33.7.8 How to Create a Databound Scheduling Gantt Chart

For a scheduling Gantt chart, you must supply identification for resources, identification for tasks, and start and end times for tasks. Optionally, you can provide data values for subresources, recurring tasks, split tasks, and dependencies between tasks.

The scheduling Gantt chart is displayed with default values for overall start and end time and for the major and minor time axis values. In a scheduling Gantt chart, the setting for the major time axis defaults to weeks and the setting for the minor time axis defaults to days.

Figure 33–71 shows a scheduling Gantt chart that lists each resource and all the orders for which that resource is responsible. In contrast to a project Gantt chart, the scheduling Gantt chart shows all the tasks for a given resource on the same line, while the project Gantt chart lists each task on a separate line.

Figure 33–71 The Scheduling Gantt Chart for Order Shipping



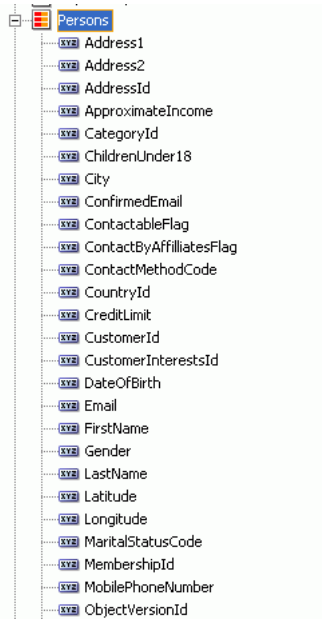
To create a scheduling Gantt chart using a data control, you bind the `schedulingGantt` tag to a data collection. JDeveloper allows you to do this declaratively by dragging and dropping a collection from the Data Controls panel.

To create a databound scheduling Gantt chart:

1. From the Data Controls panel, select a data collection. For a Gantt chart, you can select a row set collection or a basic tree collection.

   Figure 33–72 shows an example where you could select the `Persons` data collection to create a scheduling Gantt chart that displays the orders that each resource is responsible for.

Figure 33–72 Data Collection for Resources



2. Drag the collection onto a JSF page and, from the context menu, choose **Gantt**, then **Scheduling**.
3. In the ensuing Create Scheduling Gantt dialog, you do the following to connect resource- and task-related controls at the top of the dialog with corresponding columns in the data collection:
   a. For **Resource Id**, select the column in the data collection that corresponds to the unique identifier of the resource.
   b. In the Tasks page select a value from the **Task Accessor** dropdown list that contains the tasks assigned to the resource. Select the columns in the data collection that correspond to each of the following controls: **Task Id**, **Start Time**, and **End Time**. You can optionally specify a data column to map to task type. If you do not bind task type to data, then all task types default to **Normal**.
   c. If the data collection has an accessor that holds dependent tasks, you have the option of using the Dependent Tasks page in the dialog to select the dependent tasks accessor and to select the columns in that data collection that correspond to each of the following controls: **Dependency Type**, **From Task Id**, and **To Task Id**.

d. If the data collection has an accessor for split tasks, you have the option of using the Split Tasks page in the dialog to select the split tasks accessor and to select the columns in that data collection that correspond to each of the following controls: **Split Task Id**, **Start Time**, and **End Time**.

e. If the data collection has an accessor for recurring tasks, you have the option of using the Recurring Tasks page in the dialog to select the Recurring Tasks accessor and to select the columns in that data collection that correspond to each of the following controls: **Recurring Task Id**, **Type**, **Start Time**, and **End Time**.

f. If the data collection has an accessor for subresources (lower-level resources), you have the option of using the Subresouces page to specify the appropriate accessor and to select the data column that contains the unique identifier of the subresource.

   For example, a manager might be a resource and his direct reports might be subresources. If data contains subresources, then you can drill in a resource to locate subresources.

4. In the **Table Columns** section, you specify the columns that will appear in the list region of the Gantt chart on the left side of the splitter. Specify one row of information for each column that is to appear. Use the **New** icon to add new rows. Use the arrow icon to arrange the rows in the exact sequence that you want the columns to appear in the Gantt chart list. For each row, you provide the following specifications:

   - **Display Label:** Select the values for the headers of the columns in the Gantt chart list. If you select `<default>`, then the text for the header is automatically retrieved from the data binding.
   - **Value Binding:** Select the columns in the data collection to use for the column in the Gantt chart list. The available values are the same as those for the tasks group.
   - **Component to Use:** Select the type of component to display in the cell of the Gantt chart list. The default is the **ADF Output Text** component.

5. Click **OK** to dismiss the Create Scheduling Gantt dialog.

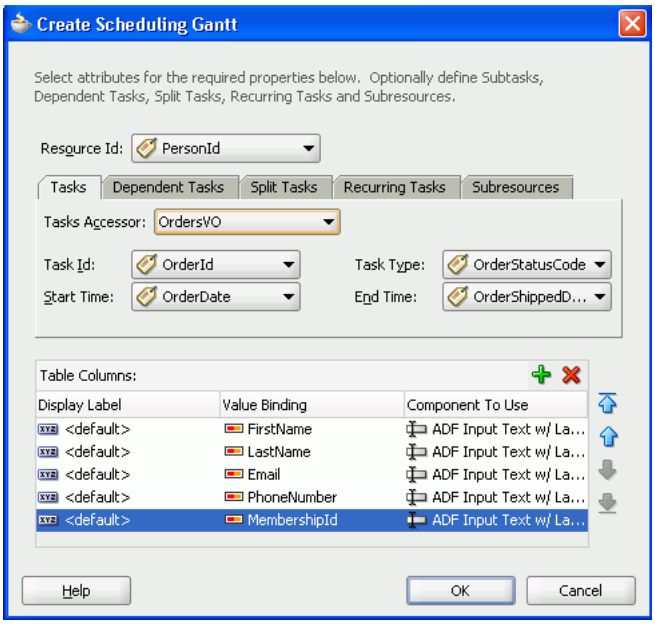
6. Select the `dvt:schedulingGantt` element in the Structure window of the JSF page and set dates for the following attributes of the `dvt:schedulingGantt` element in the Property Inspector:

   - `StartTime`
   - `EndTime`

   The dates that you specify determine the initial view that appears in the scheduling Gantt chart at runtime.

Figure 33–73 shows the dialog used to create the scheduling Gantt chart from the data collection for resources responsible for shipping orders.

Figure 33–73 Create Scheduling Gantt Dialog



33.7.9 What Happens When You Create a Scheduling Gantt Chart

Dropping a scheduling Gantt chart from the Data Controls panel has the following effect:

- Creates the bindings for the Gantt chart and adds the bindings to the page definition file
- Adds the necessary code for the UI components to the JSF page

Example 33–44 shows the row set bindings that were generated for the scheduling Gantt chart that displays resources and orders shipped.

Example 33–44 Binding XML for a Scheduling Gantt Chart

```
<gantt IterBinding="PersonsIterator" id="Persons"
       xmlns="http://xmlns.oracle.com/adfm/dvt">
  <ganttDataMap>
    <nodeDefinition DefName="oracle.fodemo.storefront.store.queries.PersonsVO"
                    type="Resources">
      <AttrNames>
        <Item Value="PersonId" type="resourceId"/>
      </AttrNames>
      <Accessors>
          <Item Value="OrdersVO" type="tasks"/>
      </Accessors>
    </nodeDefinition>
    <nodeDefinition type="Tasks"
                    DefName="oracle.fodemo.storefront.store.queries.OrdersVO">
      <AttrNames>
        <Item Value="OrderId" type="taskId"/>
        <Item Value="OrderDate" type="startTime"/>
        <Item Value="OrderStatusCode" type="taskType"/>
        <Item Value="OrderShippedDate" type="endTime"/>
      </AttrNames>
    </nodeDefinition>
    <nodeDefinition type="Dependents">
```

```
      <AttrNames/>
    </nodeDefinition>
    <nodeDefinition type="SplitTasks">
      <AttrNames/>
    </nodeDefinition>
    <nodeDefinition type="RecurringTasks">
      <AttrNames/>
    </nodeDefinition>
    <nodeDefinition type="Subresources">
      <AttrNames/>
    </nodeDefinition>
  </ganttDataMap
</gantt>
```

Example 33–45 shows the code generated on the JSF page for the scheduling Gantt chart. This tag code contains settings for the overall start and end time for the scheduling Gantt chart. It also shows the time axis settings for the major time axis (in months) and the minor time axis (in weeks). Finally, it lists the specifications for each column that appears in the list region of the Gantt chart. For brevity, the code in the `af:column` elements for `MembershipId`, `Email`, and `PhoneNumber` has been omitted.

Example 33–45 Code on the JSF Page for a Scheduling Gantt Chart

```
<dvt:schedulingGantt id="schedulingGantt1"
                     value="#{bindings.Persons.schedulingGanttModel}"
                     var="row" startTime="2011-03-29"
                     endTime="2011-05-30"
                     taskbarFormatManager="#{GanttBean.taskbarFormatManager}">
  <f:facet name="major">
    <dvt:timeAxis scale="months"/>
  </f:facet>
  <f:facet name="minor">
    <dvt:timeAxis scale="weeks"/>
  </f:facet>
  <f:facet name="nodeStamp">
    <af:column sortProperty="FirstName" sortable="false"
               headerText="#{bindings.Persons.hints.FirstName.label}">
      <af:outputText value="#{row.FirstName}"/>
   </af:column>
 </f:facet>
   <af:column sortProperty="LastName" sortable="false"
              headerText="#{bindings.Persons.hints.LastName.label}">
      <af:outputText value="#{row.LastName}"/>
    </af:column>
...
  <dvt:ganttLegend/>
</dvt:schedulingGantt>
```

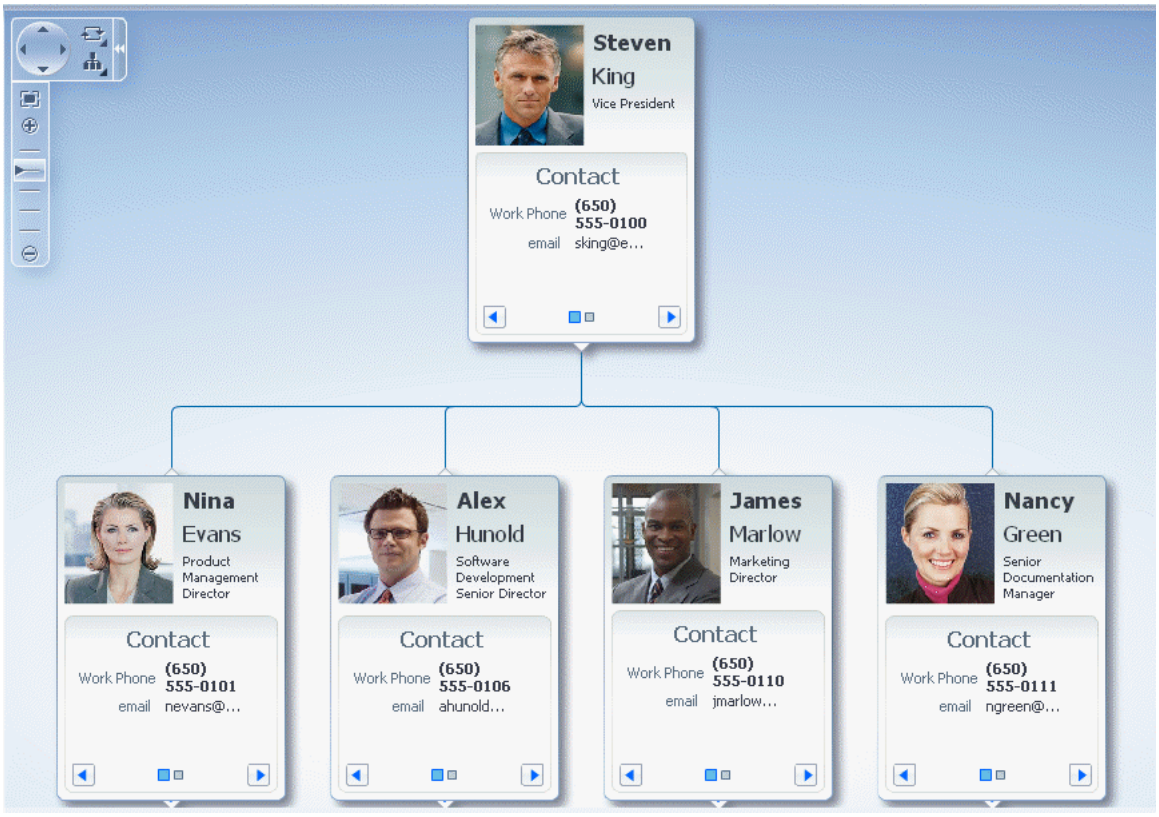
33.8 Creating Databound Hierarchy Viewers

A hierarchy viewer is an ADF Data Visualization component that visually displays data where parent-child relationships exist within the data. This component is useful where you want to display organization charts, network diagrams, social networks, or similar visual displays.

Hierarchy viewers use a shape called a *node* to reference the data in a hierarchy. The shape and content of the nodes is configurable, as well as the visual layout of the nodes. Nodes can display multiple views in a panel card.

Figure 33–74 shows a runtime view of a hierarchy viewer component that renders an organization chart.

Figure 33–74 Hierarchy Viewer Component Rendering an Organization Chart



Each hierarchy viewer component (`dvt:hierarchyViewer`) that you create can include:

- One or more node elements (`dvt:node`)
- One or more link elements (`dvt:link`)

The optional panel card element (`dvt:panelCard`) can be used in conjunction with the hierarchy viewer component. The panel card provides a method to dynamically switch between multiple sets of content referenced by a node element using animation by, for example, horizontally sliding the content or flipping a node over.

For detailed information about hierarchy viewer end user and presentation features, use cases, tag structure, and adding special features to hierarchy viewers, see the "Using ADF Hierarchy Viewer Components" chapter in *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

33.8.1 How to Create a Hierarchy Viewer Using ADF Data Controls

Hierarchy viewers are based on data collections where a master-detail relationship exists between one or more detail collections and a master data collection. Using data controls in Oracle ADF, JDeveloper makes this a declarative task. You drag and drop a

data collection from the Data Controls panel that generates one or more root nodes onto a JSF page.

Before you begin:

It may be helpful to have an understanding of how hierarchy viewer attributes and hierarchy viewer child tags can affect functionality. For more information, see the "Configuring Hierarchy Viewers" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

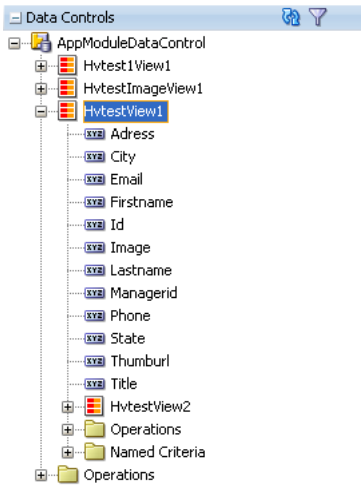
You should already have an ADF data control or ADF managed bean that represents the data you wish to display in the hierarchy viewer.

To create a hierarchy viewer using the Data Controls panel:

1. From the Data Controls panel, select a collection.

   Figure 33–75 shows an example where you could select the `HvtestView1` collection in the Data Controls panel to create a hierarchy viewer representing the personnel data in an organizational chart.

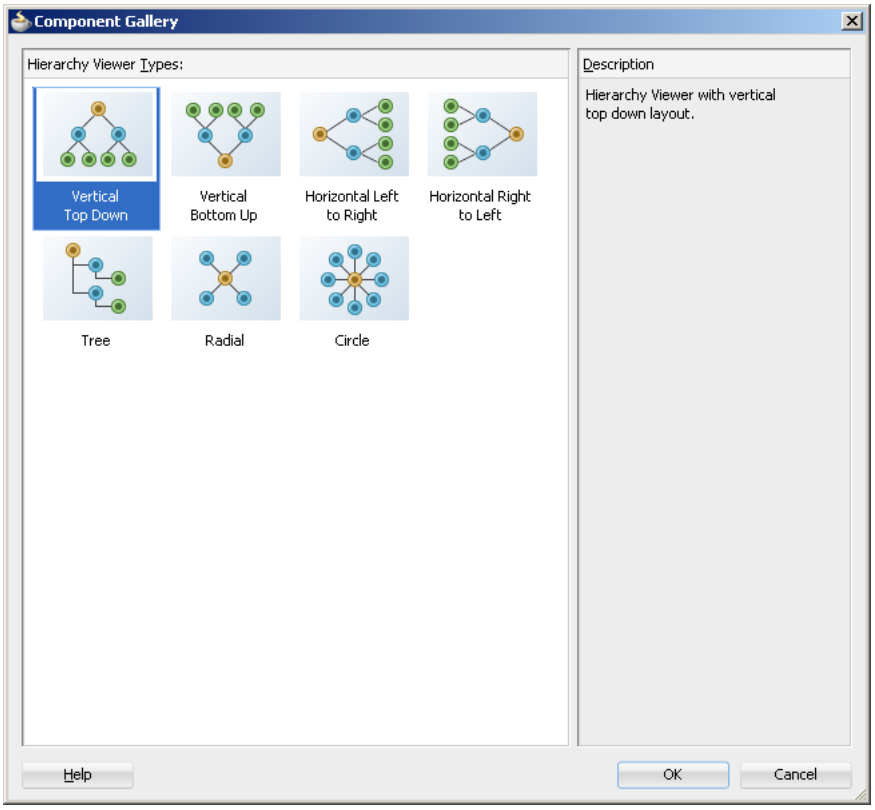
Figure 33–75 Data Collection for Personnel Organizational Chart



2. Drag the collection onto a JSF page and, from the context menu, choose **Hierarchy Viewer.**

3. From the Component Gallery, select the layout of the hierarchy viewer you want to create. Figure 33–76 shows the Component Gallery with the vertical top down layout selected.

Figure 33–76 Hierarchy Viewer Component Gallery



4. In the Create Hierarchy Viewer dialog, enter the following:

   - **Hierarchy**: Select the collections you want to include as nodes in the runtime diagram. By default, the root collection is selected.
   - **Node**: For each collection that you select in the Hierarchy list, configure the attributes in the Title Area, and the title and attributes in each panel card, using one or more of the zoom levels available to you. By default, the node for the 100% zoom level follows an algorithm that:
     - Assigns the first three attributes in the data collection to the **Title Area**.
     - Assigns the next two attributes to the first panel card.
     - Assigns the following two attributes to the second panel card, and so on until all attributes are assigned to a panel card.

     Click the **New** icon to add a new panel card, attribute, or image to the node, relative to your selection in the node. After selecting an existing element, use the arrow icons (**Up**, **Down**, **Top**, **Bottom**) to reorder the element, or use the **Delete** icon to delete the selected element.

     Select a panel card, attribute, or image to configure one or more of the following for that element:

     - **Text**: Available for panel card elements. Not available for the **Title Area** element. Enter text for the panel card title in the hierarchy, or choose **Select Text Resource** from the dropdown list to open a dialog to select or add a text resource to use for the title. The text resource is a translatable string from an application resource bundle. If you need help, press **F1** or click **Help**.

- **Attribute**: Available for attribute and image elements. From the dropdown list, select the attribute in the data collection that represents the value you wish to display in the hierarchy viewer node.
- **Label**: Available for attributes. Not available for default attributes assigned to the Title Area. From the dropdown list, select the attribute in the data collection that represents the label you wish to display for that element in the hierarchy viewer node. You can use the default **Use Attribute Name** to render the value as a string using the label from the UIHints for that attribute in the underlying `ViewObject`, select **No Label** to render no label for the attribute, or choose **Select Text Resource** to open a Select Text Resource dialog to select or add a text resource to use for the label. The text resource is a translatable string from an application resource bundle.
- **Component**: Available for attributes. Select the type of component to use to display the data in the node. Valid values are: **ADF Output Text/w label**, **ADF Output Text**, **ADF Output Formatted Text w/ Label**, and **ADF Output Formatted Text**. For more information about using these components, see the "Using Output Components" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Use the 75%, 50%, and 25% pages to specify the expanded display of the hierarchy at each page level. Select **Add Zoom Level** to enable the page level. By default, the algorithms to assign node attributes and panel cards in the node are similar to the 100% zoom level.

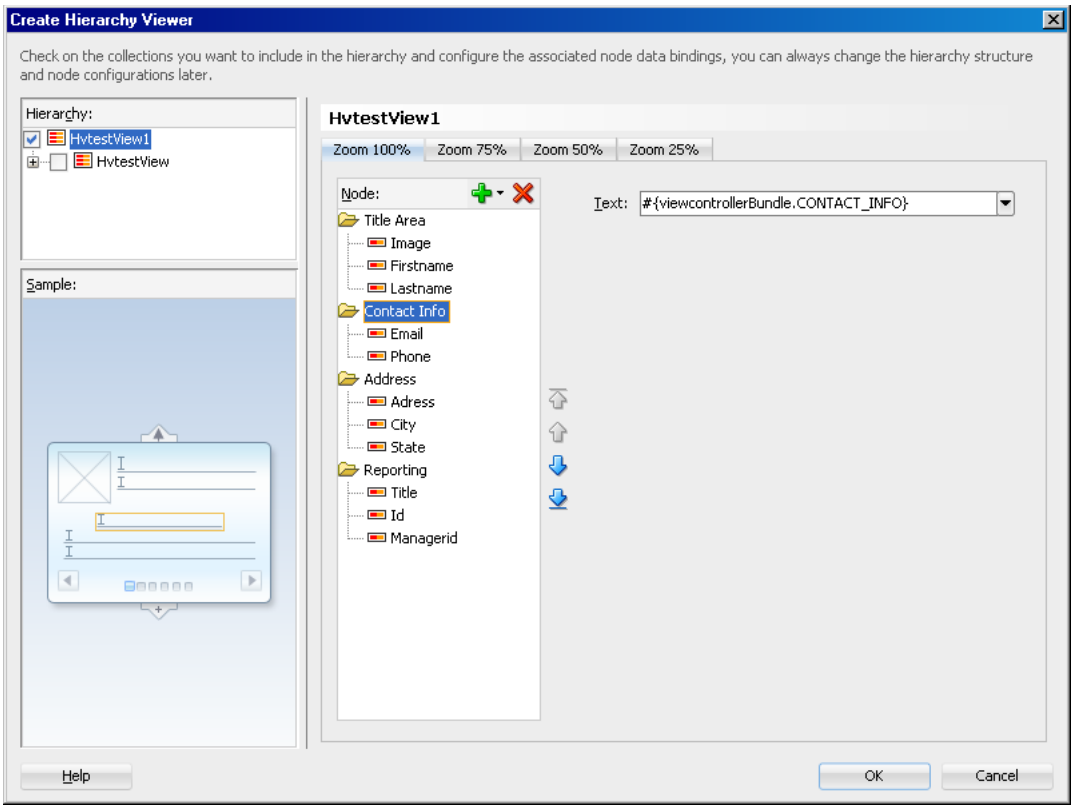
Note: The hierarchy viewer component defines four zoom levels. You cannot modify these zoom levels or create new zoom levels. The default zoom level is 100%.

- **Sample**: A non-configurable display of the sample outline of the hierarchy viewer node. Areas such as Title Area and attributes are highlighted in the sample when selected in the node area.

5. Click **OK**.

Figure 33–77 shows the Create Hierarchy Viewer dialog that appears if you create a hierarchy viewer using data from a data collection named `HvtestView1`.

Figure 33–77 Create Hierarchy Viewer Dialog



After completing the Create Hierarchy Viewer dialog, you can use the Property Inspector to specify settings for the hierarchy viewer attributes and you can also use the child tags associated with the hierarchy viewer tag to customize the hierarchy viewer further.

When editing the hierarchy viewer in the visual editor, the display approximates the runtime display and functionality. You can move between panel cards, and work with stamped elements within the panel cards and title area to rearrange attributes or bind new attributes imported from the Data Controls panel.

33.8.2 What Happens When You Create a Databound Hierarchy Viewer

Creating a hierarchy viewer from Data Controls panel has the following effect:

- Creates the bindings for the hierarchy viewer in the page definition file of the JSF page
- Adds the necessary tags to the JSF page for the hierarchy viewer component

Example 33–46 displays bindings that JDeveloper generated for a hierarchy viewer component. The rules for populating the nodes of the master-detail tree are defined as a node definition. The example shows that two node definitions were generated. Each of these node definitions references a view object and associated attributes. The code example also references an accessor `HvtestView`.

Example 33–46 Bindings for a Hierarchy Viewer

```
<tree IterBinding="HvtestView1Iterator" id="HvtestView1">
  <nodeDefinition DefName="model.HvtestView"> Name="HvtestView10">
    <AttrNames>
```

```
    <Item Value="Adress"/>
    <Item Value="Email"/>
    <Item Value="Lastname"/>
    <Item Value="Thumburl"/>
    <Item Value="Firstname"/>
    <Item Value="Title"/>
    <Item Value="Managerid"/>
    <Item Value="Id"/>
    <Item Value="Phone"/>
    <Item Value="City"/>
    <Item Value="State"/>
  </AttrNames>
  <Accessors>
    <Item Value="HvtestView"/>
  </Accessors>
  </nodeDefinition>
  <nodeDefinition DefName="model.HvtestView">
  <AttrNames>
    <ittem Value="Adress"/>
    <Item Value="Email"/>
    <Item Value="Lastname"/>
    <Item Value="Thumburl"/>
    <Item Value="Firstname"/>
    <Item Value="Title"/>
    <Item Value="Managerid"/>
    <Item Value="Id"/>
    <Item Value="Phone"/>
    <Item Value="City"/>
    <Item Value="State"/>
  </AttrNames>
  <Accessors>
   <Item Value="ManageridHvtestView"/>
  </Accessors>
  </nodeDefinition>
</tree>
```

Example 33–47 shows the code generated on the JSF page that is associated with the page definition file in Example 33–46. For brevity, the code in the facet elements named `zoom75`, `zoom50`, and `zoom25` has been omitted. Some code from elements such as `<af:panelGroupLayout>`, `<af:spacer/>`, and `<af:separator/>` has also been omitted.

The example shows a hierarchy viewer component that references the `HvtestView1` tree binding. It includes a node `(dvt:node)` component that in turn includes a panel card component `(dvt:panelCard)`. The panel card component defines `slide_horz` as the effect to use when changing the display of content referenced by the node.

Once you create your hierarchy viewer, you can modify the layout of the component or add additional components, such as a panel card, using the Create Hierarchy dialog illustrated in Figure 33–77. To open the dialog, use the **Edit** icon in the Property Inspector for the `hierarcyViewer` component. You can also customize the layout of a hierarchy viewer component directly in the code, in the visual editor, or by setting values in the Property Inspector. You can add additional components, such as panel card, using the Component Palette.

Example 33–47 Code on the JSF Page for a Hierarchy Viewer

```
<dvt:hierarchyViewer inlineStyle="width:100%;height:600px;" id="hv1" var="node"
                     value="#{bindings.HvtestView1.treeModel}"
                 selectionListener="#{bindings.HvtestView1.
```

```
                    treeModel.makeCurrent}"
                featuresOff="nodeDetail" layout="hier_vert_top"
                levelFetchSize="#{bindings.HvtestView1.rangeSize}"
                summary="Emplyee Organizational Chart"/>
<dvt:link linkType="orthogonalRounded" id="l1"/>
<dvt:node type="model.HvtestView" width="233" height="233" id="n1">
  <f:facet name="zoom100">
    <af:image source="#{node.Image}" inlineStyle="width:85px;height:120px;"
                   inlineStyle="font-weight:bold;font-size:14px"
                   id="ot1"/>
    <af:outputText value="#{node.Firstname}"
                   inlineStyle="font-size:14px"
                   id="ot2"/>
    <af:outputText value="#{node.Lastname}"
                   inlineStyle="font-style:italic;font-size:14px"
                   id="ot3"/>
    <dvt:panelCard effect="slide_horz" id="pc1">
      <af:showDetailItem text="#{viewcontrollerBundle.CONTACT_INFO}"
                         id="sdi1">
      <af:panelFormLayout inlineStyle="padding:5px;" id="pfl1">
          <af:goLink text="#{viewcontrollerBundle.EMAIL}"
                     destination="mailto:#{node.bindings.Email.inputValue}"/>
        <af:panelLabelAndMessage label="#{viewcontrollerBundle.PHONE_NUMBER}"
                                 labelStyle="font-size:14px;color:#5A6A7A"
                                 id="plam3">
          <af:outputText value="#{node.Phone}"
                         inlineStyle="font-size:14px;color:#383A47" id="ot5"/>
        </af:panelLabelAndMessage>
        </af:panelFormLayout>
      </af:showDetailItem>
      <af:showDetailItem text="#{viewcontrollerBundle.REPORTING}" id="sdi2">
        <af:panelFormLayout inlineStyle="padding:5px;" id="pfl2">
          <af:outputText value="#{node.Title}"
                         inlineStyle="font-size:14px"
                         id="ot6"/>
        <af:panelLabelAndMessage label="#{viewcontrollerBundle.MANAGER}"
                                 labelStyle="font-size:14px"
                                 id="plam4">
          <af:outputText value="#{node.Managerid}"
                         inlineStyle="font-size:14px;color:#383A47" id="ot7">
            <af:convertNumber groupingUsed="false"
                       pattern="#{bindings.HvtestView1.hints.Managerid.format}"/>
          </af:outputText>
         </af:panelLabelAndMessage>
         <af:panelLabelAndMessage label="#{viewcontrollerBundle.EMPLOYEE_ID}"
                                  labelStyle="font-size:14px" id="plam5">
           <af:outputText value="#{node.Id}"
                          inlineStyle="font-size:14px" id="ot8">
             <af:convertNumber groupingUsed="false"
                            pattern="#{bindings.HvtestView1.hints.Id.format}"/>
           </af:outputText>
         </af:panelLabelAndMessage>
         </af:panelFormLayout>
           </af:showDetailItem>
           <af:showDetailItem text="#{viewcontrollerBundle.ADDRESS}" id="sdi3">
           <af:panelFormLayout inlineStyle="padding:5px;" id="pfl3">
           <af:panelLabelAndMessage label="#{viewcontrollerBundle.ADDRESS}"
                                    labelStyle="font-size:14px" idd="plam6">
           <af:outputText value="#{node.Adress}"
                          inlineStyle="font-size:14px" id="ot9"/>
```

```
            <af:panelLabelAndMessage>
            <af:panelLabelAndMessage label="#{viewcontrollerBundle.STATE}"
                                     labelStyle="font-size:14px" id-"palm7">
              <af:outputText value="#{node.State}"
                             inlineStyle="font-size:14px" id="ot10"/>
            </af:panelLabelAndMessage>
            <af:panelLabelAndMessage label="#{viewcontrollerBundle.CITY}"
                                     labelStyle="font-size:14px" id="plam8">
              <af:outputText value="#{node.City}"
                             inlineStyle="font-size:14px" id="ot11"/>
            </af:panelLabelAndMessage>
          </af:panelFormLayout>
        </af:showDetailItem>
      </dvt:panelCard>
    </f:facet>
  </dvt:node>
</dvt:hierarchyViewer>
```

33.8.3 How to Create a Databound Search in a Hierarchy Viewer

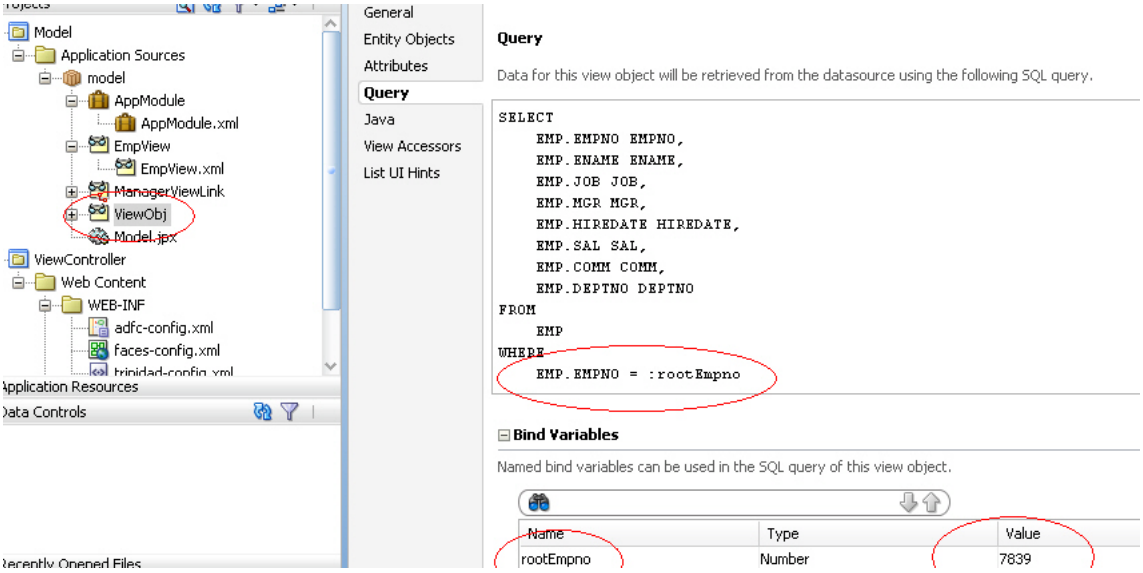
The search function in a hierarchy viewer is based on the searchable attributes or columns of the data collection that is the basis of the hierarchy viewer data model. Using a query results collection defined in data controls in Oracle ADF, JDeveloper makes this a declarative task. You drag and drop an **ExecuteWithParams** operation into an existing hierarchy viewer component on the page.

Before you begin:

1. You must have a databound hierarchy viewer component present on your page.
2. Verify the query that retrieves the root node in the hierarchy viewer.

   For example, Figure 33–78 shows retrieving the root node by `EMPNO` column.

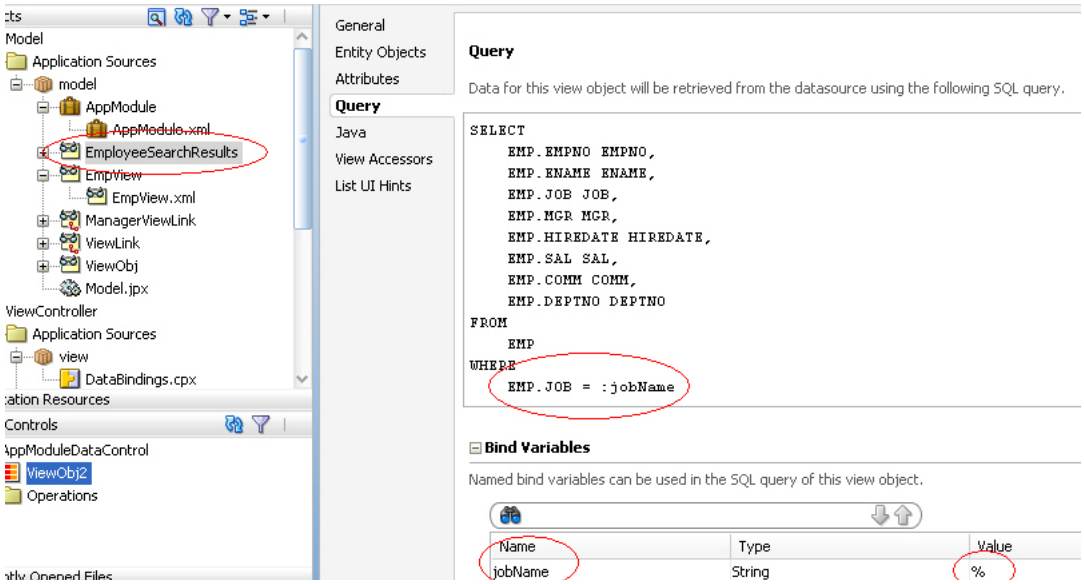
Figure 33–78 Root Node Query



3. Create a view object that performs the search.

For example, Figure 33–79 shows the `EmployeesSearchResults` view object that performs the search based on the `Job` column in the data collection with a default value of `%` for matching any value, and a comparison value of = specifying an exact match against the column.

Figure 33–79 EmployeeSearchResults View Object



For information about creating a view object, see Section 5.2.1, "How to Create an Entity-Based View Object."

To create a databound search with a hierarchy viewer:

1. From the Data Controls panel, select the collection that corresponds to the query results and expand the **Operations** node to display the **ExecuteWithParams** operation.
2. Drag the **ExecuteWithParams** operation and drop it onto the hierarchy viewer in the visual editor or onto the component in the Structure window. Alternatively, you can drag the parameter onto the hierarchy viewer.
3. In the Create Hierarchy Viewer Search dialog that displays, use the **Add** icon to specify the list of results to display in the Search Results panel, and specify the following for each result:
   a. **Display Label**: Select the values for the headers of the nodes in the hierarchy. If you select `<default>`, then the text for the header is automatically retrieved from the data binding.
   b. **Value Binding:** Select the columns in the data collection to use for nodes in the tree for the hierarchy viewer.
   c. **Component to Use**: Select the type of component to display in the node. The default is the ADF Output Text component.

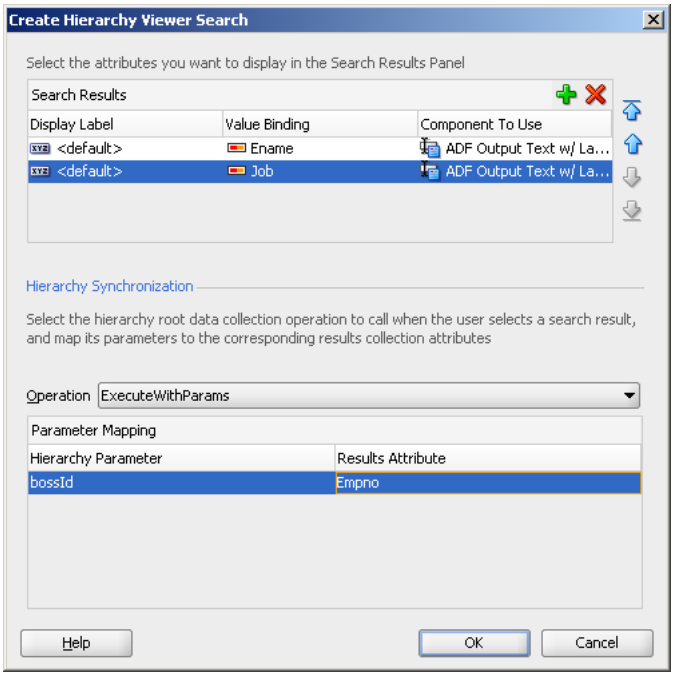
   After selecting an existing field, use the arrow icons (**Up**, **Down**, **Top**, **Bottom**) to reorder the results or use the **Delete** icon to delete that result.
4. In the **Operation** dropdown list select the hierarchy root data collection to use when a search result is selected. Valid values include:

- `removeRowWithKey`: Uses the row key as a `String` converted from the value specified by the input field to remove the data object in the bound data collection.
- `setCurrentRowWithKey`: Sets the row key as a `String` converted from the value specified by the input field. The row key is used to set the currency of the data object in the bound data collection.
- `setCurrentRowWithKeyValue`: Sets the current object on the iterator, given a key's value.
- `ExecuteWithParams`: Sets the values to the named bind variables passed as parameters.

5. In the **Parameter Mapping** table, use the dropdown list in the **Results Attribute** column to select the results collection attribute to map to the parameter displayed in the **Hierarchy Parameter** column.

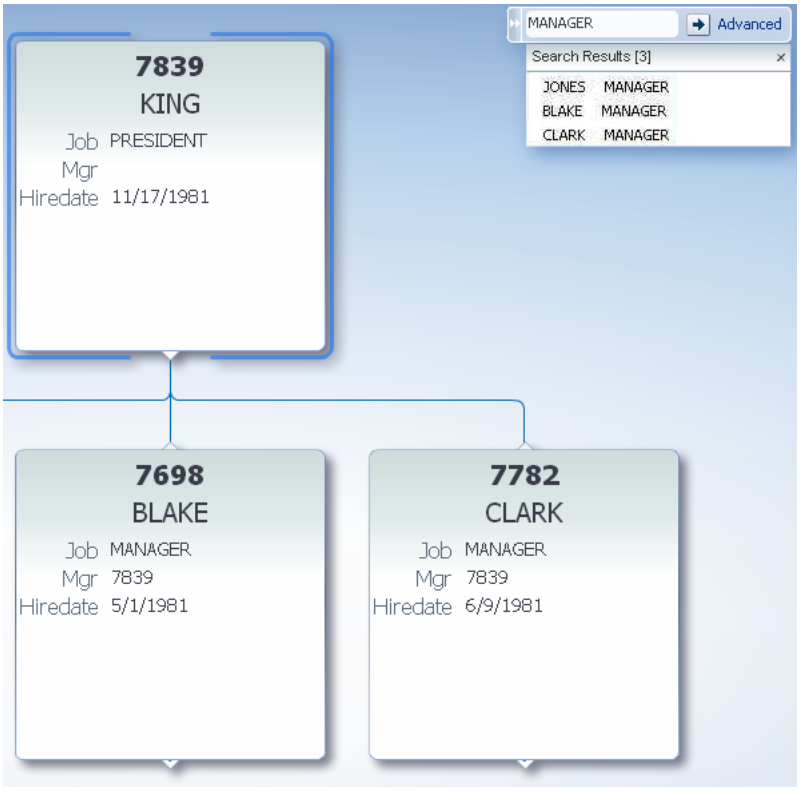
Figure 33–80 shows the Create Hierarchy Viewer Search dialog that appears if you create a hierarchy viewer using data from a data collection named `EmployeesSearchResults1`.

Figure 33–80 Create Hierarchy Viewer Search Dialog



At runtime, the search results are displayed in a table by `Ename` and `Job`. Figure 33–81 shows the resulting search results panel when the user enters "MANAGER" in the search box.

Figure 33–81 Hierarchy Viewer Search Results



34

Using Contextual Events

This chapter describes how to create, publish, and subscribe to contextual events to facilitate communications between regions.

This chapter includes the following sections:

- Section 34.1, "About Creating Contextual Events"
- Section 34.2, "Creating Contextual Events Declaratively"
- Section 34.3, "Creating Contextual Events Manually"
- Section 34.4, "Creating Contextual Events Using Managed Beans"
- Section 34.5, "Creating Contextual Events Using JavaScript"
- Section 34.6, "Creating the Event Map Manually"
- Section 34.7, "Registering a Custom Dispatcher"

34.1 About Creating Contextual Events

Often a page or a region within a page needs information from somewhere else on the page or from a different region. While you can pass parameters to obtain that information, doing so makes sense only when the parameters are well known and the inputs are EL-accessible to the page. Parameters are also useful when a task flow may need to be restarted if the parameter value changes.

However, suppose you have a task flow with multiple page fragments that contain various interesting values that could be used as input on one of the pages in the flow. If you were to use parameters to pass the value, the task flow would need to surface output parameters for the union of each of the interesting values on each and every fragment. Instead, for each fragment that contains the needed information, you can define a contextual event that will be raised when the page is submitted. The page or fragment that requires the information can then subscribe to the various events and receive the information through the event.

In the StoreFront module, contextual events are used in the customer registration page to display the appropriate informational topic. The user registration page `register.jspx` contains two regions. One region contains the customer registration task flow `customer-registration-task-flow`, and the other contains the informational topic task flow `help-task-flow`. A contextual event is passed from the customer registration region to the informational topic region so that the informational topic task flow can display the information topic. At design time, the event name, producer region, consumer region, consumer handler, and other information is stored in the event map section of the page definition file, as shown in Example 34–1.

Example 34–1 Event Map in the registerPageDef.xml File

```
<eventMap xmlns="http://xmlns.oracle.com/adfm/contextualEvent">
    <event name="queueHelpTopic">
      <producer region="*">
        <consumer region="helptaskflow1"
                  handler="helpPageDef.findHelpTextById">
         <parameters>
            <parameter name="helpTopicId" value="${payLoad}"/>
          </parameters>
        </consumer>
      </producer>
    </event>
</eventMap>
```

At runtime, when users enter the customer registration task flow, they progress through a series of view activities from **Basic Information** to **Address** and then to **Payment Options** by entering data and clicking the **Next** button. When the user clicks **Next**, a contextual event with a `payLoad` parameter is broadcasted by the customer registration task flow. This event is then consumed by the information task flow and its handler, the `helpPageDef.findHelpTextById()` method. The consuming method uses the `payLoad` parameter to determine which information topic text to display. In the event map, you can specify EL expressions to bind the input parameters to variables and parameters of the page.

Events are configured in the page definition file for the page or region that will raise the event (the producer). In order to associate the producer with the consumer that will do something based on the event, you create an event map also in the page definition (when using events between regions, the page definition file which holds both the regions contains the event map). If the consuming page is in a dynamic region, the event map should be in the page definition file of the consuming page and the producer's attribute region set to "*". The attribute region is set to "*" because at design time, the framework cannot determine the relative path to the producer.

You can raise a contextual event for an action binding, a method action binding, a value attribute binding, or a range binding (table, tree, or list binding). You also can conditionally fire an event and conditionally handle an event using EL expressions.

For action and method action bindings, the event is raised when the action or method is executed. The `payLoad` contains the method return value. You can also raise a contextual event from an ADF Faces event such as clicking a button or selecting from a menu. An `eventBinding` is created in the page definition to define the event.

For a value attribute binding, the event is triggered by the binding container and raised after the attribute is set successfully. The `payLoad` is an instance of `DCBindingContainerValueChangeEvent`, which provides access to the new and old value, the producer iterator, the binding container, and the source. If the `payLoad` property is changed to point to a custom data object, then the `payLoad` reference will return the object instead. Example 34–2 shows a value change event inside an attribute value binding associated with an input component. The event, `valueChangeEvent`, will be dispatched when the user changes the value of `LAST_NAME` in the page.

Example 34–2 Value Attribute Event in the Page Definition File

```
<attributeValues IterBinding="DeptView1Iterator" id="Dname"
     xmlns="http://xmlns.oracle.com/adfm/jcuimodel">
    <events xmlns="http://xmlns.oracle.com/adfm/contextualEvent">
        <event name="valueChangeEvent"/>
```

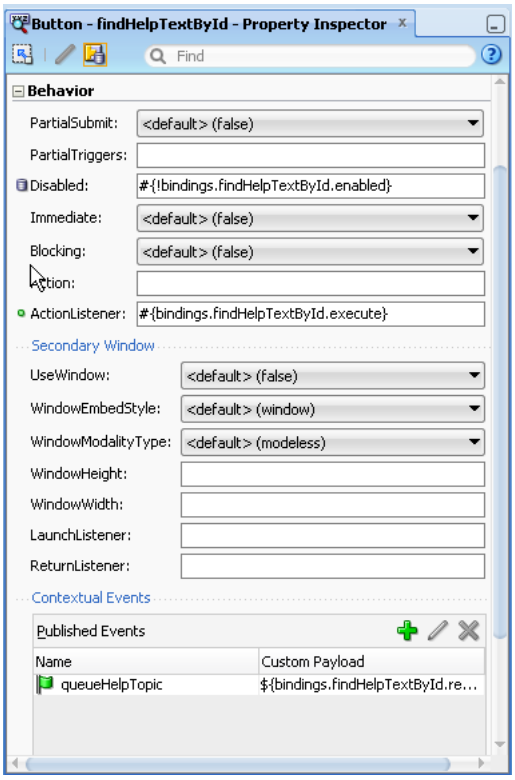
```
     </events>
     <AttrNames xmlns="http://xmlns.oracle.com/adfm/uimodel">
        <Item Value="LAST_NAME"/>
      </AttrNames>
</attributeValues>
</bindings>
<eventMap xmlns="http://xmlns.oracle.com/adfm/contextualEvent">
    <event name="valueChangeEvent">
      <producer region="LAST_NAME">
        <consumer region="" handler="consumeEvent"/>
      </producer>
    </event>
</eventMap>
```

For an range binding (tree, table, list), the event is raised after the currency change has succeeded. The `payLoad` is an instance of `DCBindingContainerValueChangeEvent`, which provides access to the new and old value, the producer iterator, the binding container, and the source.

Value attribute binding and range binding contextual events may also be triggered by navigational changes. For example, if you create an event inside a tree table binding, the event will be dispatched when the user selects a different node of the tree in the page.

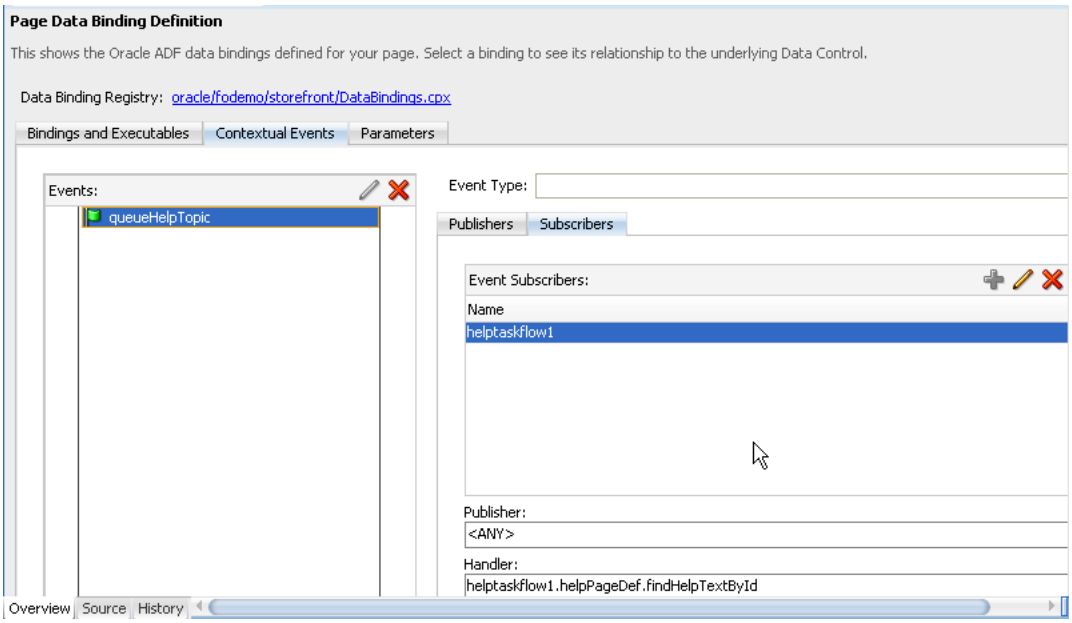
You use the Contextual Events section under the **Behavior** section in the Property Inspector to create, publish, and subscribe to contextual events. The Contextual Events panel will only appear when you select eligible components or regions in the page, as shown in Figure 34–1.

Figure 34–1 Contextual Events Panel in the Property Inspector



You can also subscribe to contextual events using the overview editor for page definition file's **Contextual Events** tab **Subscriber** section, as shown Figure 34–2

Figure 34–2 Page Definition Contextual Events Tab



Contextual events are not the same as the business events that can be raised by ADF Business Components or the events raised by UI components. For a description of these types of events, see the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*. Contextual events can be used, however, in association with UI events. In this case, an action listener that is invoked due to a UI event can, in turn, invoke a method action binding that then raises the event.

The contextual events framework provides the communications between regions within a page. The framework provides the page with the ability to map events that are produced and consumed by the various regions on the page. You can use JDeveloper to declaratively publish events using the page definition file. Similarly, you can declaratively subscribe to those events from the page definition file. You can pass parameters with the event and implement handlers to respond to an event. Other ways to create contextual events include using managed beans and using JavaScript. Note that contextual events do not require regions to be refreshed for the region to uptake parameters.

34.1.1 Contextual Events Use Cases and Examples

There are three kinds of communication patterns between a region and the parent page: parent to region, region to parent, and region to region. The contextual events framework is the most powerful communication implementation that works for all three of these scenarios. Also note that contextual events do not require a region to be refreshed in order to consume input parameters.

You should use contextual events to communicate between ADF regions. While it is possible to use ADF task flow parameters to communicate between regions, doing so may create direct dependencies between the regions. ADF task flow parameters may be fine for static regions, but using contextual events will allow you to implement independent communications between regions.

For example, a page may contain a region with a form for entering employee information. When the user updates the employee Id and presses the submit button, a value change contextual event is raised and the employee Id value is also passed as payLoad. On the same page, another region subscribes to and consumes the event and displays departmental information about the selected employee based on the payLoad information passed with the contextual event.

34.1.2 Additional Functionality for Contextual Events

You may find it helpful to understand other ADF features before you work with contextual events. Following are links to other functionality that may be of interest.

- You can also use ADF task flow parameters to communicate between ADF regions. For more information about using task flows and regions, see Chapter 21, "Using Task Flows as Regions."

34.2 Creating Contextual Events Declaratively

You create contextual events by first creating and publishing the event on the producer based on a method action, action, value attribute, or list binding. On the consumer, you subscribe to the event and create a handler to process the event.

Typically, you create a parent page with regions that contain task flows and view activities. You create contextual events in one region to be published for consumer event handlers in the other region. For more information about using task flows and regions, see Chapter 21, "Using Task Flows as Regions."

Note: You can also publish an action contextual event from code (for example, from within a managed bean). You can use `getBindingContainer().getEventDispatcher` which returns an instance of `EventDispatcher`. `EventDispatcher` has public APIs which you can use to programmatically raise contextual events.

34.2.1 How to Publish Contextual Events

You use the Property Inspector to create contextual events in the producer's page.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 34.2, "Creating Contextual Events Declaratively."

You may also find it helpful to understand functionality that can be used with contextual events. For more information, see Section 34.1.2, "Additional Functionality for Contextual Events."

You will need to complete this task:

Decide on the type of component you want to use to raise the contextual event. If you plan to use a method action binding, you must have already created the method to be dropped onto the page

To create a contextual event:

1. In the producer page, drag and drop a component from the Data Controls panel to the page that will trigger the event.

The component must have a method action, action, value attribute, or list binding. In the StoreFront module, the `setHelpId()` method from the Data Controls panel was added to the page.

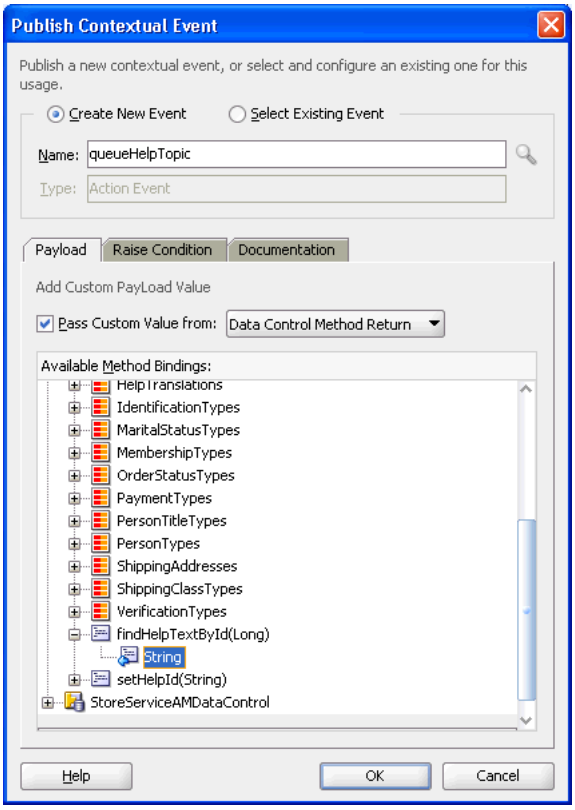
2. In the Property Inpector expand the **Behavior** section.
3. In the **Publish Events** section, click the **Add** icon.
4. In the Publish Contextual Events dialog:
   a. Select **Create New Event**.
   b. Enter the name of the event.
   c. If you are using a table, tree, or tree table, a **Node** field appears for you to enter the node that will publish the event.
   d. Select **Pass Custom Value From** if you want to pass payload data to the consumer.
   e. If you are passing payload data, select the type of data from the dropdown list.

      For example, if you want to pass an attribute value from the producer page to the consumer page, you can select **Page Data** and select the attribute from the tree structure.
   f. You can conditionally raise the event by entering an EL expression in the **Raise Condition** tab.

      For instance, entering an expression such as `${bindings.LAST_NAME.inputValue == 'KING'}` will cause the event to be raised only if the customer's last name is KING.
   g. Click **OK**.

      The event is created on the page, but it is not ready for publishing until it is associated with the component binding.

Figure 34–3 Publish a Contextual Event



34.2.2 How to Subscribe to and Consume Contextual Events

You use the overview editor for page definition files on the parent page to subscribe to contextual events.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 34.2, "Creating Contextual Events Declaratively."

You may also find it helpful to understand functionality that can be used with contextual events. For more information, see Section 34.1.2, "Additional Functionality for Contextual Events."

You will need to complete this task:

Create a contextual event in the page definition file, as described in Section 34.2.1, "How to Publish Contextual Events."

To subscribe and consume the event:

1. In the consuming page, add the components that may respond to the event.

   In the StoreFront module, the `findHelpTextById` method handler return value, **String**, is dropped onto the page as an `outputText` component to display the help information.

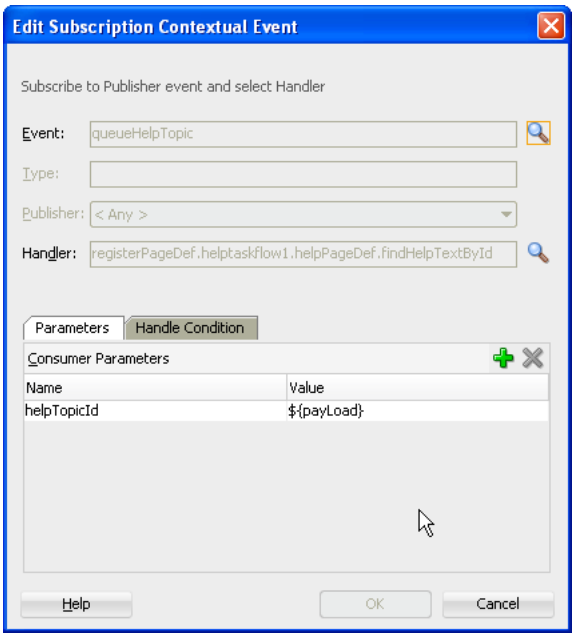
2. Create a handler to process the event and its payload data.

   In the StoreFrontModule example, the `findHelpTextById` handler method was created in the `LookupServiceAMDataControl` module.

3. In the overview editor for the consumer page definition's **Bindings and Executables** tab **Bindings** section, click the **Add** icon.
4. In the Insert Item dialog, select **methodAction** and click **OK**.
5. In the Create Action Binding dialog:
   a. Select the data collection where you have created your handler.
   b. From the **Operation** dropdown list, select the handler.
   c. Click **OK**.
6. In the overview editor for the consumer page definition's **Bindings and Executables** tab **Bindings** section, click the **Add** icon.
7. In the Insert Item dialog, select **attributeValue** and click **OK**.
8. In the Create Attribute Binding dialog:
   a. From the **Data Source** dropdown list, select **Variable**.
   b. Select the return value of the handler as the **Attribute**.
   c. Click **OK**.
9. In the overview editor **Contextual Events** tab, click **Subscribers** and click the **Add** icon in the **Event Subscribers** section.
10. In the Subscribe to Contextual Event dialog, click the **Search** icon.
11. In the Select Contextual Events dialog, select the event you want to subscribe to from the tree and click **OK**.
12. In the Subscribe to Contextual Event dialog:
    a. Select the producer or **<Any>** from the **Publisher** dropdown list. A contextual event can have more than one producer.

       Selecting **<Any>** will allow the consumer to subscribe to any producer producing the selected event. In the page definition file, the `producer` attribute will be set to the wildcard "*". If your producer is in a dynamic region, you should set this field to **<Any>** so that the subscriber can consume from any producer.
    b. Click the **Search** icon next to the **Handler** field.
    c. In the Select Handler dialog, select the event handler from the tree and click **OK**.
    d. If the handler requires parameters, select the **Parameters** tab, click **Add**, and enter name-value pair as parameters.
    e. If you want to conditionally handle the event, select the **Handle** tab, and enter an EL Expression that determines the conditions under which the handler will process the event.
    f. Click **OK**.

Figure 34–4 Subscribe to a Contextual Event



Note: You can edit the event map by right-clicking the page definition in the Structure window and choosing **Edit Event Map**. You can also edit event attributes in the page definition file or in the Property Inspector.

34.2.3 What Happens When You Create Contextual Events

When you create an event for the producer, JDeveloper adds an `events` element to the page definition file. Each event name is added as a child. Example 34–3 shows the event on the `setHelpId` method action binding in the `account_basicinformationPageDef` page definition file of the StoreFront module. This is the page definition for the Basic Information view of the customer registration task flow.

Example 34–3 Event Definition for the Producer

```
<methodAction id="setHelpId"
                  InstanceName="LookupServiceAMDataControl.dataProvider"
                  DataControl="LookupServiceAMDataControl"
                  RequiresUpdateModel="true" Action="invokeMethod"
                  MethodName="setHelpId" IsViewObjectMethod="false"
                  ReturnName="LookupServiceAMDataControl.
                      methodResults.setHelpId_
                      LookupServiceAMDataControl_dataProvider_
                      setHelpId_result">
                  <NamedData NDName="usage" NDValue="CREATE_PROFILE"
                       NDType="java.lang.String"/>
 <events xmlns="http://xmlns.oracle.com/adfm/contextualEvent">
                  <event name="queueHelpTopic"/>
 </events>
```

When the method action binding is invoked, the event is broadcasted to its consumers.

When you configure an event map, JDeveloper creates an event map entry in the corresponding page definition file. Example 34–4 shows the event map on the `registerPageDef` page definition file that maps the `queueHelpTopic` event from the `customerregistrationtaskflow1` region to the `helptaskflow1` region. It also maps the `helpPageDef.findHelpTextById` handler method bindings that is defined in the `helpPageDef` page definition file. The consumer invokes a method that determine the information text to display based on the parameters that are passed into it. The mapping is in the `registerPageDef` page definition, as that is the parent container for both the `customerregistrationtaskflow1` and the `helptaskflow1` regions.

Example 34–4 Event Map in the Parent Page Definition File

```
<eventMap xmlns="http://xmlns.oracle.com/adfm/contextualEvent">
    <event name="queueHelpTopic">
      <producer region="*">
        <consumer region="helptaskflow1"
                  handler="helpPageDef.findHelpTextById">
         <parameters>
            <parameter name="helpTopicId" value="${payLoad}"/>
          </parameters>
        </consumer>
      </producer>
    </event>
</eventMap>
```

34.2.4 How to Control Contextual Events Dispatch

You can control the dispatch of contextual events to child regions at the application level or at the page level.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 34.2, "Creating Contextual Events Declaratively."

You may also find it helpful to understand functionality that can be used with contextual events. For more information, see Section 34.1.2, "Additional Functionality for Contextual Events."

To disable event dispatch:

1. To disable event dispatch at the application level, set the `dynamicEventSubscriptions` property to `false` in the `adf-config.xml` file, as shown in Example 34–5.

   You can disable the event dispatch to regions that have an event map with producers as wildcards.

Example 34–5 Disabling Contextual Event Dispatch at the Application Level Using adf-config.xml

```
<?xml version="1.0" encoding="windows-1252" ?>
<adf-config xmlns="http://xmlns.oracle.com/adf/config"
     xmlns:cef="http://xmlns.oracle.com/adfm/contextualEvent">
     <cef:DynamicRegionEventsConfig dynamicEventSubscriptions="false">
     </cef:DynamicRegionEventsConfig>
</adf-config>
```

2. To disable event dispatch at the individual page level, set the `dynamicEventSubscriptions` property to `false` in the associated page definition file, as shown in Example 34–6.

   Contextual events will not be passed to the page and any of its children.

Example 34–6 Disabling Contextual Event Dispatch for a Page Using the Page Definition File

```
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="11.1.1.52.8" id="viewBPageDef" Package="view.pageDefs"
                DynamicEventSubscriptions="false">
```

34.2.5 What Happens at Runtime: Contextual Events

If both the event producer and the consumer are defined in the same page definition file, then after the corresponding page is invoked and the binding container is created, the event is raised when:

- The corresponding method or action binding is executed
- A value binding is set successfully
- A range binding currency is set successfully

For a method binding, the result of the method execution forms the payload of the event, and the event is queued. In the Invoke Application phase of the JSF lifecycle, all the queued events will be dispatched. The event dispatcher associated with the binding container checks the event map (also in the binding container, as it is part of the same page definition file) for a consumer interested in that event and delivers the event to the consumer.

When the producer and consumer are in different regions, the event is first dispatched to any consumer in the same container, and then the event propagation is delegated to the parent binding container. This process continues until the parent or the topmost binding container is reached. After the topmost binding container is reached, the event is again dispatched to child-binding containers that have regions with pages that have producer set to wildcard "* ".

34.3 Creating Contextual Events Manually

You create contextual events by first creating the event on the producer. You then determine the consumer of the event, and map the producer to the consumer.

34.3.1 How to Create Contextual Events Manually

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 34.3, "Creating Contextual Events Manually."

You may also find it helpful to understand functionality that can be used with contextual events. For more information, see Section 34.1.2, "Additional Functionality for Contextual Events."

You will need to complete this task:

Create a contextual event that has a method binding, action binding, value attribute binding, or list binding on the producer page.

To create a contextual event:

1. In the Application Navigator, open the page definition file that contains the binding for the producer of the event.

   A producer must have an associated binding that will be used to raise the event. For example, if a method or operation will be the producer, the associated action binding or method action binding will contain the event.

2. In the Structure window, right-click the binding for the producer and choose **Insert inside** ***binding name*** **> events** or **Insert inside** ***binding name*** **> Contextual Events > events**.

3. In the Structure window, right-click the **events** element just created, and choose **Insert inside events > event**.

4. In the Insert event dialog, enter a name for the event in the **name** field, and click **Finish**.

   The event is now created. By default, any return of the associated method or operation will be taken as the payload for the event and stored in the EL-accessible variable `${payLoad}`. You now need to map the event to the consumer, and to configure any payload that needs to be passed to the consumer.

5. Open the page definition that contains the binding for the consumer.

   The binding container represented by this page provides access to the events from the current scope, including all contained binding containers (such as task flow regions). If regions or other nested containers need to be aware of the event, the event map should be in the page definition of the page in the consuming region.

6. In the Structure window, right-click the topmost node that represents the page definition, and choose **Edit Event Map**.

   Note: If the producer event comes from a page in an embedded dynamic region, you may not be able to edit the event map using the Event Map Editor. You can manually create the event map by editing the page definition file or use **insert inside** steps, as described in Section 34.3, "Creating Contextual Events Manually."

7. In the Event Map Editor, click the Add icon to add an event entry.

8. In the Add New EventMap Entry dialog, do the following:

   a. Use the **Producer** dropdown menu to choose the producer.

   b. Use the **Event Name** dropdown menu to choose the event.

   c. Use the **Consumer** dropdown menu to choose the consumer. This should be the actual method that will consume the event.

   d. If the consuming method or operation requires parameters, click the **Add** icon.

      In the **Param Name** field, enter the name of the parameter expected by the method. In the **Param Value** field, enter the value. If this is to be the payload from the event, you can access this value using the `${payLoad}` expression. If the payload contains many parameters and you don't need them all, use the ellipses button to open the Expression Builder dialog. You can use this dialog to select specific parameters under the **payload** node.

You can also click the **Parameters** ellipses button to launch the selection dialog.

e. Click **OK**.

9. In the Event Map Editor, click **OK**.

34.4 Creating Contextual Events Using Managed Beans

You can publish an action contextual event from code such as from within a managed bean. You bind the producer component to the method in the managed bean, as shown in Example 34–7.

In this example, the producer is a command button that invokes an action binding and the consumer is an `outputText` component that displays a string. They are both on the same page.

Example 34–7 Event Producer and Event Consumer on the JSF

```
<af:form id="f1">
     <af:commandButton value="eventProducerButton1" id="cb1"
                       action="#{MyBean.myActionPerformed}"
                       />
     <af:panelLabelAndMessage label="#{bindings.return.hints.label}"id="plam1">
          <af:outputText value="#{bindings.return.inputValue}" id="ot1"/>
     </af:panelLabelAndMessage>
</af:form>
```

The page definition file contains the method action bindings for the producer, the consumer, and the event map, as shown in Example 34–8.

Example 34–8 Page Definition with Event Producer, Event Consumer, and Event Map

```
<executables>
    <variableIterator id="variables">
      <variable Type="java.lang.String" Name="eventConsumer_return"
                IsQueriable="false" IsUpdateable="0"
                DefaultValue="${bindings.eventConsumer.result}"/>
    </variableIterator>
</executables>
<bindings>
     <methodAction id="eventProducer"
                  InstanceName="AppModuleDataControl.dataProvider"
                  DataControl="AppModuleDataControl" RequiresUpdateModel="true"
                  Action="invokeMethod" MethodName="eventProducer"
                  IsViewObjectMethod="false"
                  ReturnName="AppModuleDataControl.methodResults.eventProducer_
                       AppModuleDataControl_dataProvider_eventProducer_result">
            <events xmlns="http://xmlns.oracle.com/adfm/contextualEvent">
                  <event name="myEvent"/>
            </events>
     </methodAction>
     <methodAction id="eventConsumer" RequiresUpdateModel="true"
                  Action="invokeMethod" MethodName="eventConsumer"
                  IsViewObjectMethod="false" DataControl="AppModuleDataControl"
                  InstanceName="AppModuleDataControl.dataProvider"
                  ReturnName="AppModuleDataControl.methodResults.eventConsumer_
                        AppModuleDataControl_dataProvider_eventConsumer_result">
              <NamedData NDName="str" NDValue="test" NDType="java.lang.String"/>
    </methodAction>
```

```
    <attributeValues IterBinding="variables" id="return">
        <AttrNames>
             <Item Value="eventConsumer_return"/>
        </AttrNames>
    </attributeValues>
</bindings>
<eventMap xmlns="http://xmlns.oracle.com/adfm/contextualEvent">
    <event name="myEvent">
      <producer region="eventProducer">
        <consumer region="" handler="eventConsumer">
          <parameters>
            <parameter name="test" value="${payLoad}"/>
          </parameters>
        </consumer>
      </producer>
    </event>
</eventMap>
```

When the button is pressed, the `myActionPerformed` method is invoked and calls the following methods to generate the contextual event with "myString" as the payLoad:

```
BindingContainer bc =
  (BindingContainer)fc.getApplication().evaluateExpressionGet(fc,"#{bindings}",
   BindingContainer.class);
JUCtrlActionBinding actionBnd =
  (JUCtrlActionBinding)bc.getControlBinding("eventProducer");
...
((DCBindingContainer)bc).getEventDispatcher().queueEvent(actionBnd.
   getEventProducer(),"myString");
((DCBindingContainer)bc).getEventDispatcher().processContextualEvents();
   return null;
```

34.5 Creating Contextual Events Using JavaScript

Every action and method binding that is accessible from a managed bean can be invoked from JavaScript. ADF Faces provides an `af:serverListener` operation component that can be used to call a managed bean method from client-side JavaScript. To invoke this component using the referenced managed bean method, use the `BindingContext` object to look up the current `BindingContainer` and to access the `OperationBinding` or a `JUEventBinding` binding. The `af:serverListener` component can also be used to send a message `payload` from the browser client to the managed bean method.

34.6 Creating the Event Map Manually

Under most circumstances, you can create the event map using the Event Map Editor as described in Section 34.3, "Creating Contextual Events Manually." However, in situations such as when the producer event is from a page in an embedded dynamic region, the Event Map Editor at design time cannot obtain the necessary information to create an event map.

34.6.1 How to Create the Event Map Manually

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 34.6, "Creating the Event Map Manually."

You may also find it useful to understand functionality that can be used with contextual events. For more information, see Section 34.1.2, "Additional Functionality for Contextual Events."

To create the event map manually:

1. In the Application Navigator, open the page definition that contains the binding for the consumer.
2. In the Structure window, right-click the topmost node that represents the page definition, and choose **Insert inside *pagedef name* > eventMap**.
3. In the Structure window, select the **eventMap** node, right-click and choose **Insert inside eventMap > event**.
4. In the Insert Event dialog, enter the name of the event and click **OK**.

   Repeat steps 3 and 4 to add more events.
5. Select the **event** node, right-click and choose **Insert inside event > producer**.
6. In the Insert Producer dialog, enter the name of the binding that is producing this event and click **OK**.

   You can also enter the name of the region which has the event producer, in which case all the consumers specified under this tag can consume the event. You can also enter "*" to denote that this event is available for all consumers under this tag.
7. Select the **producer** node, right-click, and choose **Insert inside producer > consumer**.
8. In the Insert Consumer dialog, enter the name of the handler that will consume the event and click **OK**.

   Repeat steps 7 and 8 to add more consumers.
9. If there are parameters being passed, add the parameter name and value.
   a. Select the **consumer** node, right-click, and choose **Insert inside consumer > parameters**.
   b. Select the **parameters** node, right-click, and choose **Insert inside parameters > parameter**.
   c. In the Insert Parameter dialog, enter the name of the parameter and the value of the parameter and click **OK**. The value can be an EL expression.

      Repeat this step to add more parameters.

34.7 Registering a Custom Dispatcher

By default, the contextual event framework uses `EventDispatcherImpl` to dispatch events that would traverse through the regions. You can create a custom event dispatcher to override the default event dispatcher to provide custom behaviors. After

you have created the custom event dispatcher, you must register it in the `Databindings.cpx` file to override the default dispatcher.

34.7.1 How to Register a Custom Dispatcher

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 34.7, "Registering a Custom Dispatcher."

You may also find it useful to understand functionality that can be used with contextual events. For more information, see Section 34.1.2, "Additional Functionality for Contextual Events."

To register a custom event dispatcher:

1. Create a custom event dispatcher Java class based on the `EventDispatcher` class.
2. Register the custom event dispatcher in the `Databindings.cpx` file with a fully qualified name using the following format:

   ```
   EventDispatcher="package_name.CustomEventDispatcher_name"
   ```

   Example 34–9 shows the code for a custom event dispatcher called `NewCustomEventDispatcher` created in package `NewPackage`.

Example 34–9 Adding Custom Event Dispatcher in the Databindings.cpx File

```
<Application xmlns="http://xmlns.oracle.com/adfm/application"
             version="11.1.1.51.60" id="DataBindings" SeparateXMLFiles="false"
             Package="project3" ClientType="JClient"
             EventDispatcher="NewPackage.NewCustomEventDispatcher">
```

3. Create the event in the producer's page definition. For more information, see Section 34.2, "Creating Contextual Events Declaratively," or Section 34.3, "Creating Contextual Events Manually."
4. Create the event map in the consumer region if the consumer is in a dynamic region. If the consumer is not in a dynamic region, you can also specify the event map in the parent page which holds both the producer and consumer regions. For more information, see Section 34.6, "Creating the Event Map Manually."

Part VI

Completing Your Application

Part V contains the following chapters:

- Chapter 35, "Enabling ADF Security in a Fusion Web Application"
- Chapter 36, "Testing and Debugging ADF Components"
- Chapter 37, "Refactoring a Fusion Web Application"
- Chapter 38, "Reusing Application Components"
- Chapter 39, "Customizing Applications with MDS"
- Chapter 40, "Allowing User Customizations at Runtime"
- Chapter 41, "Deploying Fusion Web Applications"

35

Enabling ADF Security in a Fusion Web Application

This chapter describes how you can enable ADF Security in the Fusion web application to define security policies for Oracle Application Development Framework (Oracle ADF) resources and to restrict the user's ability to view web pages associated those resources.

This chapter includes the following sections:

- Section 35.1, "About ADF Security"
- Section 35.2, "ADF Security Process Overview"
- Section 35.3, "Enabling ADF Security"
- Section 35.4, "Creating Application Roles"
- Section 35.5, "Defining ADF Security Policies"
- Section 35.6, "Creating Test Users"
- Section 35.7, "Creating a Login Page"
- Section 35.8, "Testing Security in JDeveloper"
- Section 35.9, "Preparing the Secure Application for Deployment"
- Section 35.10, "Disabling ADF Security"
- Section 35.11, "Advanced Topics and Best Practices"

35.1 About ADF Security

The ADF Security framework is the preferred technology to provide authentication and authorization services to the Fusion web application. ADF Security is built on top of the Oracle Platform Security Services (OPSS) architecture, which itself is well-integrated with Oracle WebLogic Server. While other security-aware models exist that can handle user login and resource protection, ADF Security is ideally suited to provide declarative, permission-based protection for ADF bounded task flows, for top-level web pages that use ADF bindings (pages that are not contained in a bounded task flow), and at the lowest level of granularity, for rows of data defined by ADF entity objects and their attributes. In this document, these specific resources that the ADF Security framework protects are known as *ADF security-aware resources*.

You enable ADF Security for Fusion web applications when you run the Configure ADF Security wizard, as described in Section 35.3, "Enabling ADF Security." The wizard configures ADF Security for the entire Fusion web application, so that any web page associated with an ADF security-aware resource is protected by default. This

means that after you enable ADF Security, your application is locked down so that the pages are considered secure by default.

After you enable ADF Security you must grant users access rights so that they may view the web pages of the Fusion web application. Access rights that you grant users are known as a *security policy* that you specify for the page's corresponding ADF security-aware resource. Ultimately, it is the security policy on the ADF resource that controls the user's ability to enter a task flow or view a web page.

Because ADF Security is based on Java Authentication and Authorization Service (JAAS), security policies identify the principal (the user or application role), the ADF resource, and the permission (an operation defined by the resource's ADF permission class). For example, the StoreFront module of the Fusion Order Demo application secures the web pages contained by the `checkout-task-flow` task flow to grant access only to logged-in users (also known as *authenticated users*). At runtime, the ADF Security framework performs authorization checking against the task flow's security policy to determine the user's right to complete the view operation. In this case, the security policy must grant the view permission to the user if they are to complete the checkout process.

To simplify the task of defining security policies for users and ADF resources, ADF Security defines a containment hierarchy that lets you define one security policy for the ADF bounded task flow and its contains web pages. In other words, when you define the security policy at the level of the bounded task flow, you protect the flow's entry point and then all pages within that flow are secured by the policy it defines. Additionally, instead of granting access to individual users, you group users into application roles and grant the view permission to the role.

Specifically, you will define security policies in the Fusion web application for the following ADF security-aware resources to make web pages accessible to users:

- ADF bounded task flow protects the entry point to the task flow, which in turn controls the user's access to the pages contained by the flow

  For example, a series of web pages may guide new customers through a registration process and the bounded task flow controls page navigation for the process. For a description of bounded task flows, see Section 18.1.2, "About Bounded Task Flows."

  The unbounded task flow is not an ADF security-aware component and thus does not participate in authorization checks. When you need to protect the constituent pages of an unbounded task flow, define grants for the page definition files associated with the pages instead.

- ADF page definition files associated with web pages

  For example, a page may display a summary of best selling products with data coordinated by the ADF bindings of the page's associated ADF page definition file. For a description of page definitions and ADF bindings, see Section 13.7, "Working with Page Definition Files."

- ADF entity objects and attributes of entity objects that reference rows of data and help define collections for display in the user interface

  For example, a web page may display an ADF Faces table component that displays columns that ADF bindings map to the attributes of an entity object as its data source. In the case of entity objects, enabling ADF Security does not automatically secure entity objects rows. The data will remain accessible to users until you define a security policy to explicitly protect the entity object or its attributes. For a description of entity objects, see Section 4.1, "About Entity Objects."

JDeveloper tools support iterative development of security so you can easily create, test, and edit security policies that you create for ADF resources. You can proceed to create test users in JDeveloper and run the application in Integrated WebLogic Server to simulate how end users will access the secured resources. This chapter describes how to configure the repository of user identities and login credentials known as the *identity store*.

> **Note:** References to the identity store in this chapter are always in the context of *test* user identities that you create for the purpose of running in Integrated WebLogic Server. Typically, you would not migrate these users to the staging environment when you deploy to Oracle WebLogic Server, as described in Section 35.9, "Preparing the Secure Application for Deployment."

To avoid a situation where you have enabled ADF Security but have not yet defined security policies to grant access to test users, the Configure ADF Security wizard lets you grant temporary view rights to all existing ADF resources (a view permission grant will be added to the security policy for each ADF resource). This wizard option gives you the choice to disable automatic grants and proceed to define security policies for ADF resources as you create each resource or to enable automatic view grants and gradually replace these grants with security policies that you define. To understand iterative security development choices, see Section 35.2, "ADF Security Process Overview."

> **Tip:** Before you enable ADF Security and define security policies for the ADF security-aware resources, you will want to understand the rules that govern ADF authorization checking. Understanding these rules will help you to implement the security you intend. For a discussion of these rules, see Section 35.1.2, "ADF Security Use Cases and Examples."

35.1.1 Integration of ADF Security and Java Security

The ADF Security model for securing Fusion web application resources is not based on the URL mapping of a security constraint as exemplified by the Java EE security model. In actual practice, security constraints are not feasible for securing a JavaServer Faces (JSF) web application where page navigation is not supported by specific page URLs. For example, when the user navigates to the next page in a task flow, the URL remains the same throughout the flow. As each new page is displayed, there is no means to trigger a URL-based security constraint.

Instead, ADF Security implements a Java Authentication and Authorization Service (JAAS) security model. The JAAS model is policy-based since JAAS is built on the existing Java security model and integrates with any JAAS implementation, including the Oracle Platform Security Services (OPSS) implementation of the JAAS service. Whereas applications that utilize URL security constraints are security-unaware because they rely on the Java EE container to manage security, Fusion web applications require an explicit call to the ADF Security framework to authorize access to resources based on user-defined policies. Thus, when you enable ADF Security and define access policies for ADF resources, your application is security-aware.

ADF Security simplifies the implementation of a JAAS authorization model. This implementation minimizes the work needed to create a security-aware application by exposing security policies on ADF resources in a declarative fashion and performing authorization checks on these resources at runtime.

The policy store in JDeveloper is file-based and contains a list of entries known as *grants*, which define the security policy for the ADF resource. The grant entry includes all the permissions granted to the user to perform operations on the protected resource, for instance, accessing a web page associated with an ADF bounded task flow. Permissions are granted in the policy store to an application role principal.

ADF Security expands on the JAAS model by allowing you to define grants using the actions specified by the ADF Security framework permission classes. These classes are specific to the ADF resource and map the actions to an operation supported by the resource. The policy store for the Fusion web application therefore contains grants that specify:

- One or more permissions that associate an action defined by the resource's permission class with an instance of the ADF resource in the application (currently, only the view action is supported for bounded task flows and page definitions resources)
- The grantee, which is an application role defined by your application that you populate with member users or, optionally, enterprise roles for whom you wish to confer the same access rights

In the case of entity objects, the permission class defines read, delete, and update actions.

For a description of the ADF permission classes and supported actions, see Appendix C, "ADF Security Permission Grants."

35.1.2 ADF Security Use Cases and Examples

The use of ADF Security enables web applications to easily adjust to real-world business security requirements, because rather than securing paths to application resources, you secure the view operation on ADF resources with JAAS. JAAS-based ADF Security provides:

- Declarative security support for ADF resources, such as the bounded task flow

  Because Java EE security is URL-based or page-based, it is not possible to have a navigation control without custom code. With ADF Security, you can control whether or not the user can enter a task flow. Thus, a single security policy for a task flow can control access to multiple web pages. Additionally, because declarative security lets you secure the ADF resource, not the access path, you can reuse the ADF resource elsewhere in the application and it will remain secured.

- Simplified permission assignment by using application roles that allow for the inheritance of permissions

  While Java EE security roles that are used by Java EE security constraints are flat, JAAS permissions are granted to application roles, which can be nested and may be mapped to enterprise roles that the Oracle WebLogic Server domain defines.

- Utility methods for use in EL expressions to access ADF resources in the security context

  You can use the ADF Security EL expression utility methods to determine whether the user is allowed to perform a known operation. For example, you can determine whether the user is allowed to view a particular task flow.

Additionally, JDeveloper enables you to quickly create test users and passwords to test security in Integrated WebLogic Server. When you are ready to deploy to Oracle WebLogic Server, you can migrate the application-specific authorization policies to the

server and the administrator can configure the application to use an LDAP user repository.

Table 35–1 summarizes the effect that enabling ADF Security has on the application and the various ADF security-aware resources. For further discussion about how you can work most effectively with ADF Security, see Section 35.11.4, "Best Practices for Working with ADF Security."

Table 35–1 Summary of ADF Security-Aware Resources

ADF Resource	How ADF Enforces Security	How to Grant Access
Bounded task flows in all user interface projects	Protected by default. Requires a grant to allow users to enter the bounded task flow.	Define the grant for the task flow. Do *not* define grants for individual page definition files associated with the web pages of the bounded task flow.
Page definition files in all user interface projects	Protected by default. Requires a grant to allow users to view the page associated with the page definition.	If the web page is part of a bounded task flow, define the grant for the task flow. Define the grant for the page definition *only* when the web page is contained by an unbounded task flow. Note that the unbounded task flow is not an ADF security-aware component and allows no grants.
Entity objects in the data model project	Not protected by default. Requires a grant to prevent access by users.	Define a grant on the entity object to protect data only if you need to control access at the level of the entire data collection. The data displayed by *all* components in the user interface that reference the protected entity object will be protected. Use entity-level security carefully. Instead, consider defining security at the level of the entity attribute. Note that grants in the data model project are saved as metadata on the entity object itself and do not appear in the ADF policy store.
Attributes of entity objects in the data model project	Not protected by default. Requires a grant to prevent access by users.	Define a grant on the entity object attribute to protect data when you need to control access at the level of the columns of the data collection. The data displayed by *all* components in the user interface that reference the protected entity attribute will be protected. Note that grants in the data model project are saved as metadata on the entity object itself and do not appear in the ADF policy store.

35.1.3 Additional Functionality for ADF Security

You may find it helpful to understand other ADF features before you start working with ADF Security. Following are links to other functionality that may be of interest.

- To understand the security features of Oracle Platform Security Services, see the *Oracle Fusion Middleware Security Guide*.

35.2 ADF Security Process Overview

You work in JDeveloper when you want to secure the ADF resources of your Fusion web application. ADF Security will protect your application's bounded task flows and any web pages contained in an unbounded task flow. You enable this protection by running the Configure ADF Security wizard and later by defining ADF security policies to define user access rights for each resource.

As you create the user interface for your application, you may run the Configure ADF Security wizard at any time. You may choose to:

- Iterate between creating web pages in the UI project and defining security policies on their associated ADF resources
- Complete all of the web pages in the UI project and then define security policies on their associated ADF resources

> **Note:** Before you proceed to secure the Fusion web application, you should become familiar with the ADF security model, as described in Section 35.1.2, "ADF Security Use Cases and Examples."

The iterative design and test process is supported by a variety of design time tools.

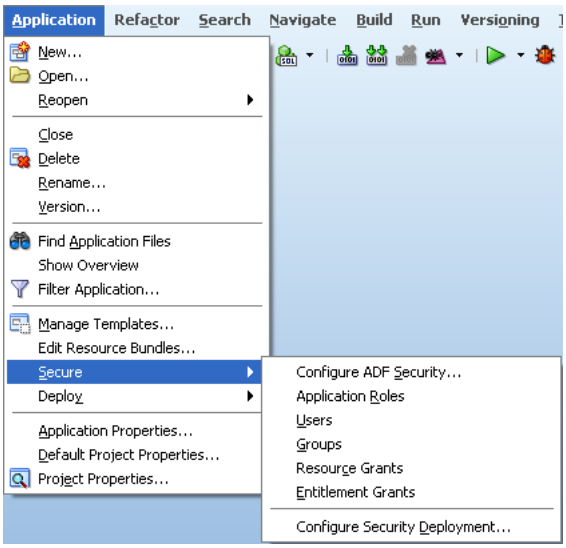
Each time you create a new bounded task flow or ADF page definition file in your user interface projects, the new ADF resource will be visible in the `jazn-data.xml` file overview editor. You use the overview editor to define security policies for ADF resources associated with web pages for the entire application. You can also use the overview editor to sort ADF resources and easily view those that have no security policy yet defined.

You use another editor to provision a few test users in the ADF identity store. The identity store you create in JDeveloper lets you define user credentials (user ID and password). The editor also displays the relationship between users you create and the application roles that you assign them to for the purpose of conferring the access rights defined by ADF security policies.

At design time, JDeveloper saves all policy store and identity store changes in a single file for the entire application. In the development environment, this is the `jazn-data.xml` file. After you configure the `jazn-data.xml` file using the editors, you can run the application in Integrated WebLogic Server and the contents of the policy store will be added to the domain-level store, the `system-jazn-data.xml` file, while the test users will be migrated to the embedded LDAP server that Integrated WebLogic Server uses for its identity store. The domain-level store allows you to test the security implementation by logging on as test users that you have created.

You access all design time tools for security under the main menu **Application > Secure** menu, as shown in Figure 35–1.

Figure 35–1 Accessing the ADF Security Design Time Tools



Design Phase

To enable ADF Security and set up the policy store in JDeveloper:

1. Enable ADF Security for the application by running the Configure ADF Security wizard.

   The wizard configures files that integrate the security framework with OPSS on Oracle WebLogic Server.

2. Create an ADF security-aware resource, such as a bounded task flow with constituent web pages (or regions) or a top-level web page (or region) that is designed using ADF bindings.

   Note: After you run the Configure ADF Security wizard, any web page associated with an ADF security-aware resource will be protected. This means that you must define security policies to make the web pages accessible *before* you can run the application and test security.

3. Associate the ADF security-aware resource with one or more application roles that you create.

   Application roles you create are specific to the application and let you confer the same level of access to a set of users (also known as *member users*). In the test phase you will create some users and add them as members to the application roles you created.

4. Grant view permission to the ADF security-aware resource and each of its associated application roles.

   The grant confers access rights to the application role's member users. Without the grant, the user would not be able to access the ADF security-aware resource. In the test phase, you will create some users and add them to your application roles.

Testing Phase

To provision the identity store and test security using Integrated WebLogic Server:

1. Create some users and, optionally, create their enterprise roles.

   You will log in to the application using the user ID and password you define. An *enterprise role* is a logical role that lets you group users and associate these groups

with application roles. The enterprise role is not needed for testing. For more information, see Section 35.4.3, "What You May Need to Know About Enterprise Roles and Application Roles."

2. Associate the users you created and, optionally, the enterprise roles, with one or more application roles.

   A member user may belong to more than one application role when you wish to confer the access right granted to multiple application roles.

3. Optionally, replace the default login page with a custom login page.

   The default login page generated by the Configure ADF Security wizard cannot utilize ADF Faces components. It is provided only as a convenience for testing ADF security policies.

4. Run the application in JDeveloper and access any ADF security-aware resource.

   The first time you attempt to access an ADF security-aware resource, the security framework will prompt you to log in.

5. Log in and check that you are able to access the page and its resources as you intended.

   After you log in, the security framework checks the user's right to access the resource. For example, if you receive an unexpected 401 unauthorized user error, verify that you have created grants as suggested in Section 35.11.4, "Best Practices for Working with ADF Security."

Preparation for Staging

To prepare the secure application for deployment to Oracle WebLogic Server in a staging or production environment:

1. Remove any grants to the `test-all` role for all ADF security-aware resources and replace with grants that you define.

   Because ADF resources are secure by default, developers testing the application will be granted view access only after security policies are defined. The Configure ADF Security wizard gives you the option to generate grants to the `test-all` role that will make all ADF resources accessible. To avoid compromising enterprise security, you must eventually replace all temporary grants to the `test-all` role with explicit grants that you define.

2. Remove all user identities that you created.

   JDeveloper must not be used as an identity store provisioning tool, and you must be careful not to deploy the application with user identities that you create for testing purposes. Deploying user identities with the application introduces the risk that malicious users may gain unintended access. Instead, rely on the system administrator to configure user identities through the tools provided by the domain-level identity management system.

3. Confirm that the application roles shown in the policy store are the ones that you want an administrator to eventually map to domain-level groups.

4. Decide whether or not you what to define a security constraint to protect ADF Faces resource files.

   Resource files including images, style sheets, and JavaScript libraries are files that the Fusion web application loads to support the individual pages of the application. These files are not secured by ADF Security, but you can secure their Java EE access paths if you require all users to be authenticated before they can access the application.

5. Migrate the finalized policy store and credentials store to the target server.

   Application policies and credentials can be automatically migrated to the domain policy store when the application is deployed to a server in the Oracle WebLogic environment. Support to automatically migrate these stores is controlled by the target server's configuration. If Oracle Enterprise Manager is used to perform the deployment outside of JDeveloper, then the migration configuration settings can be specified in that tool. For information about migrating the `jazn-data.xml` policies and the `cwallet.sso` credentials, see the *Oracle Fusion Middleware Security Guide*.

35.3 Enabling ADF Security

To simplify the configuration process which allows ADF Security to integrate with OPSS, JDeveloper provides the Configure ADF Security wizard. The wizard is the starting point for securing the Fusion web application using ADF Security. The wizard is an application-level tool that, once run, will enable ADF Security for all user interface projects that your application contains.

> **Note:** Because the Configure ADF Security wizard enables ADF Security for all user interface projects in the application, after you run it, users will be required to have authorization rights to view any web page contained by a bounded task flow and all web pages associated with an ADF page definition. Therefore, after you run the wizard, the application is essentially locked down until you define security policies to grant view rights to the user. For an overview of the process, see Section 35.2, "ADF Security Process Overview."

35.3.1 How to Enable ADF Security

The Configure ADF Security wizard allows you to choose to enable authentication and authorization separately. You may choose to:

- Enable *only* user authentication.

  Although ADF Security leverages Java EE container-managed security for authentication, enabling authentication means that you want to use the ADF authentication servlet to support user login and logout, but that you intend to define container-managed security constraints to secure web pages.

- Enable user authentication and also enable authorization.

  Enabling authorization means you intend to control access to the Fusion web application by creating security policies on ADF resources.

The ADF Security framework supports these two choices to give you the option to implement Java EE Security and still be able to support login and logout using the ADF authentication servlet. The benefit of enabling the ADF authentication servlet is that the servlet will automatically prompt the user to log in the first time the application is accessed. The ADF authentication servlet also allows you to redirect the user to a defined start page after successful authentication. You will also be able to manage the page redirect when the user logs out of the application. These redirect features provided by ADF Security are not available using only container-managed security.

> **Best Practice:** Because Java EE security constraints cannot interact with the task flow to secure the current page of a task flow, container-managed security is not a useful solution when your application is designed with ADF task flows. When you use ADF task flows, select the **ADF Authentication and Authorization** option in the Configure ADF Security wizard. This option will allow you to define security policies to protect the task flows of your application.

Because ADF Security delegates authentication to the web container, when you run the Configure ADF Security wizard, the wizard prompts you to configure the authentication method that you want the web container to use. The most commonly used types of authentication are HTTP Basic Authentication and Form-Based Authentication. Basic authentication uses the browser login dialog for the user to enter a user name and password. Note that with basic authentication, the browser caches credentials from the user, thus preventing logout. Basic authentication is useful when you want to test the application without requiring a custom login page. Form authentication allows the application developer to specify a custom login UI. If you choose Form-based authentication, you can also use the wizard to generate a simple login page. The default login page is useful for testing your application with Integrated WebLogic Server.

> **Note:** Because the generated login page is a simple JSP or HTML file, you will not be able to modify it with ADF Faces components. For information about replacing the default login page with a custom login page that uses ADF Faces components, see Section 35.7, "Creating a Login Page."

Before you begin:

It may be helpful to have an understanding of the Configure ADF Security wizard. For more information, see Section 35.3, "Enabling ADF Security."

To enable ADF Security for the application:

1. In the main menu, choose **Application** and then **Secure > Configure ADF Security**.
2. In the ADF Security page, leave the default **ADF Authentication and Authorization** option selected. Click **Next**.

   When you run the wizard with the default option selected, your application will enforce authorization for ADF security-aware resources. Enforcing authorization for ADF resources means that you intend to define security policies for these resources to make the web pages of your application accessible. Until you do so, all pages that rely on the ADF bounded task flows and ADF page definitions will remain protected.

   The other two wizard options to configure ADF Security should not be used when you want to enable ADF Security. Those options allow you to temporarily disable ADF Security and run your application without security protection, as described in Section 35.10, "Disabling ADF Security."

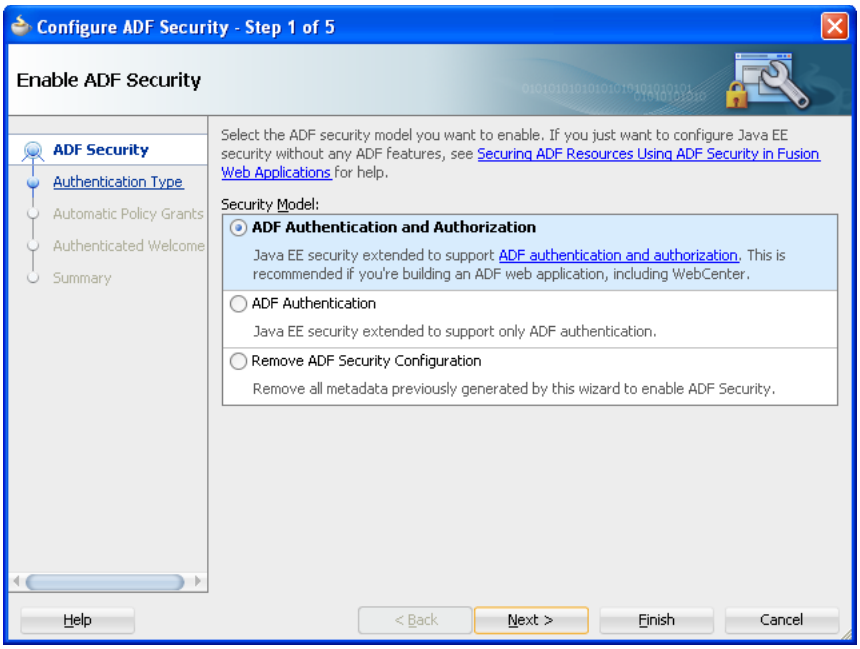
   Specifically, the first page of the wizard lets you choose among three options, with the default option set to enable **ADF Authentication and Authorization**, as shown in Figure 35–2:

   - **ADF Authentication and Authorization** (default) enables the ADF authentication servlet so that you can redirect to a configured web page when

the user logs in and logs out. This option also enables ADF authorization to enforce authorization checking against security policies that you define for ADF resources. This option assumes that you will define application roles and assign explicit grants to those roles to manage access to ADF security-aware resources.

- **ADF Authentication** enables the ADF authentication servlet to require the user to log in the first time a page in the application is accessed and supports page redirect by mapping the Java EE application root "/" to the a Java EE security constraint that will trigger user authentication. Since the wizard disables ADF authorization, authorization checking is not performed, whether or not security policies exist for ADF resources. Once the user is logged in, all web pages containing ADF resources will be available to the user.
- **Remove ADF Security Configuration** disables the ADF authentication servlet and prevents ADF Security from checking policy grants without altering the existing policy store. In this case, you may require users to log in, become authenticated, and test access rights against URL security constraints using standard Java EE security. Note that running the wizard with this option disables fine-grained security against ADF resources.

Figure 35–2 Using the Configure ADF Security Wizard to Enable Security-Aware Resources



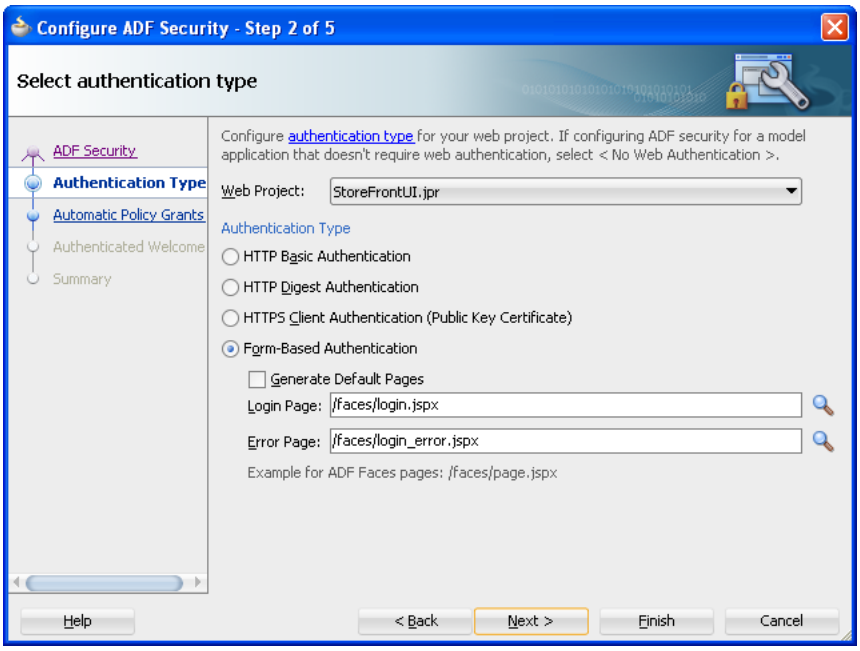
3. In the Authentication Type page, select the authentication type that you want your application to use when the user submits their login information. Click **Next**.

   A known issue prevents the ADF authentication servlet from working with Basic type authentication and allows a user to access resources after logout. Use form-based authentication instead of basic authentication. For details about this issue, see Section 35.7.5, "What You May Need to Know About ADF Servlet Logout and Browser Caching."

   If you select **Form-based Authentication**, you can also select **Generate Default Pages** to allow the wizard to generate a default login and error page. By default the wizard generates the login and error pages at the top level of the user interface

project, as shown in Figure 35–3. If you want to change the location, specify the full path relative to the user interface project.

Figure 35–3 Using the Configure ADF Security Wizard to Generate a Simple Login Page



4. In the Automatic Policy Grants page, leave the default **No Automatic Grants** option selected. Click **Next**.

   When you select **No Automatic Grants**, you must define explicit grants that are specific to your application. The `test-all` application role provides a convenient way to run and test application resources without the restricted access that ADF authorization enforces. However, it increases the risk that your application may leave some resources unprotected.

   Alternatively, you can use the wizard to grant to the `test-all` application role. When you enable grants to the `test-all` role, you can postpone defining explicit grants to ADF resources until you are ready to refine the access policies of your application. If you decide to enable automatic grants, do not let application development progress too far and the content of the application become well-established before you replace grants to the `test-all` role with the your application's explicit grants. The explicit grant establishes the necessary privilege (for example, `view` on a page) to allow users to access these resources. For more information about the `test-all` role, see Section 35.8.3, "How to Use the Built-In test-all Application Role."

5. In the Authenticated Welcome page, select **Redirect Upon Successful Authentication** to direct the user to a specific web page after they log in. Click **Next**.

   If you leave the **Redirect Upon Successful Authentication** option unselected, the user will be returned to the page from which the login was initiated. However, when the user presses **Ctrl-N** or **Ctrl-T** to open a new browser window or tab, they will receive a 403 or 404 error unless a welcome page definition appears in the application's `web.xml` file. You can use this option to specify a welcome page so the definition appears in the application's `web.xml` file.

Note that if the web page you specify contains ADF Faces components, you must define the page in the context of `/faces/`. For example, the path for `adffaces_welcome.jspx` would appear in the **Welcome Page** field as `/faces/adffaces_welcome.jspx`.

For details about specifying other redirect options, see Section 35.7.4, "How to Redirect a User After Authentication."

6. In the Summary page, review your selections and click **Finish**.

35.3.2 What Happens When You Enable ADF Security

After you run the Configure ADF Security wizard with the default **ADF Authentication and Authorization** option selected in the ADF Security page, you will have:

- Enabled ADF authentication to prompt the user to log in and to allow page redirects
- Enabled ADF authorization checking so that only authorized users will have access to ADF resources

The wizard updates all security-related configuration files and ensures that ADF resources are secure by default. Table 35–2 shows which files the Configure ADF Security wizard updates.

Table 35–2 Files Updated for ADF Authentication and Authorization

File	File Location	Wizard Configuration
`web.xml`	`/public_html/WEB-INF` directory relative to the user interface project	▪ Defines the Oracle `JpsFilter` filter to set up the OPSS policy provider. The filter defines settings that indicate that your servlet has special privileges. It is important that the `JpsFilter` be the first filter definition in the `web.xml` file. ▪ Adds the Oracle `adfAuthentication` servlet definition to require the user to log in the first time the application is accessed. ▪ When you select the **ADF Authentication and Authorization** option in the wizard, maps the `adfAuthentication` servlet to a security constraint that will trigger user authentication dynamically. ▪ When you select the **ADF Authentication** option in the wizard, maps the Java EE application root "/" to the `allPages` security constraint that will trigger user authentication dynamically. ▪ Sets the authentication method for the Login configuration to handle user login. ▪ Defines required security roles, including the role `valid-users`, which is used to trigger the security constraint that enables dynamic authentication.
`adf-config.xml`	`/.adf/META-INF` directory relative to the web application workspace	▪ Defines the JAAS security context. ▪ Enables the use of the ADF authentication servlet (the `authenticationRequire` parameter in the `<JaasSecurityContext>` element is set to `true`). This parameter is used in J2SE applications to trigger displaying a login dialog. ▪ Enables the use of ADF Security security policies for authorization checking (the `authorizationEnforce` parameter in the `<JaasSecurityContext>` element is set to `true`).
`jps-config.xml`	`/src/META-INF` directory relative to the web application workspace	▪ Defines the Oracle Platform Security Services context for the credential store. ▪ Defines the Oracle Platform Security Services context for the identity store and references the `jazn-data.xml` file as the service provider. ▪ Defines the Oracle Platform Security Services context for the policy store and references the `jazn-data.xml` file as the service provider. ▪ Defines the Oracle Platform Security Services context for the anonymous user, which contains the anonymous service instance and the anonymous login module.
`weblogic.xml`	`/public_html/WEB-INF` directory relative to the web application workspace	▪ Maps the `valid-users` security role to the Oracle Platform Security Services principal `users`.
`jazn-data.xml`	`./src/META-INF` directory relative to the web application workspace	▪ Sets the default `jazn.com` realm name for the XML identity store that you configure for use with Integrated WebLogic Server.

Because authentication is delegated to the web container, the wizard updates the `web.xml` file to enable authentication by the ADF authentication servlet. It defines servlet mapping for the ADF authentication servlet and adds two Java EE security constraints, `allPages` and `adfAuthentication`, to the `web.xml` file, as shown in Example 35–1.

Example 35–1 ADF Authentication Descriptors in the web.xml FIle

```
<servlet>
   <servlet-name>adfAuthentication</servlet-name>
   <servlet-class>
        oracle.adf.share.security.authentication.AuthenticationServlet
   </servlet-class>
   <load-on-startup>1</load-on-startup>
</servlet>
...
<servlet-mapping>
   <servlet-name>adfAuthentication</servlet-name>
   <url-pattern>/adfAuthentication</url-pattern>
</servlet-mapping>
...
<security-constraint>
   <web-resource-collection>
        <web-resource-name>allPages</web-resource-name>
        <url-pattern>/</url-pattern>
   </web-resource-collection>
   <auth-constraint>
        <role-name>valid-users</role-name>
   </auth-constraint>
</security-constraint>
<security-constraint>
   <web-resource-collection>
        <web-resource-name>adfAuthentication</web-resource-name>
        <url-pattern>/adfAuthentication</url-pattern>
   </web-resource-collection>
   <auth-constraint>
        <role-name>valid-users</role-name>
   </auth-constraint>
</security-constraint>
```

Because the `allPages` constraint maps to the '`/`' URL, it protects the Java EE application root. This mapping enables the ADF authentication servlet to trigger user authentication dynamically even before ADF Security is accessed. When the user first accesses the application, it forces the container to challenge the user for the user name and password. Then when the user accesses a page protected by ADF Security, there is no longer a need to authenticate the user and no need to redirect to the ADF authentication servlet.

> **Note:** You can remove the `allPages` constraint from the `web.xml` file if you prefer to provide a login link or button to explicitly trigger login. You could also have a link or button to perform logout. For details about creating a custom component to perform login and logout, see Section 35.7, "Creating a Login Page." If you keep the constraint to allow dynamic authentication, because it covers everything under the Java EE application root, your login page may not display supporting resources at runtime, as described in Section 35.7.2.5, "Ensuring That the Login Page's Resources Are Accessible."

Because every user of the application is required to be able to log in, the security constraint defined against the `adfAuthentication` resource allows all users to access this web resource. As such, the security role associated with the constraint must encompass all users. To simplify this task, the Java EE `valid-users` role is defined. The `weblogic.xml` file maps this role to an implicit `users` group defined by Oracle WebLogic Server. This mapping ensures that every user will have this role because Oracle WebLogic Server configures all properly authenticated users as members of the `users` group, as described in Section 35.3.7, "What You May Need to Know About the valid-users Role."

> **Note:** The `adfAuthentication` resource constraint provides the definition of a single standard URL pattern against the ADF authentication servlet. Your web pages can provide an explicit login or logout link that references the ADF authentication servlet URL pattern. This explicit login scenario is an alternative to generating a simple login form in the Configure ADF Security wizard and relying on ADF authentication to prompt the user to log in. For details about handling the explicit login scenario, see Section 35.7, "Creating a Login Page."

To enable authorization, the wizard updates the `adf-config.xml` file and sets the `authorizationEnforce` parameter in the `<JaasSecurityContext>` element to `true`, as shown in Example 35–2.

Example 35–2 AuthorizationEnforce Flag Enabled in the adf-config.xml FIle

```
<JaasSecurityContext
  initialContextFactoryClass="oracle.adf.share.security.JAASInitialContextFactory"
  jaasProviderClass="oracle.adf.share.security.providers.jps.JpsSecurityContext"
                             authorizationEnforce="true"
                             authenticationRequire="true"/>
```

When authorization is enabled, the ADF security context gets the user principals from the `HttpServletRequest` once the user is authenticated by the container. The user submits a user name and password and that data is compared against the data in the identity store where user information is stored. If a match is found, the originator of the request (the user) is authenticated. The user name is then stored in the ADF security context, where it can be accessed to obtain other security-related information (such as the group the user belongs to) in order to determine authorization rights. For details about accessing the ADF security context, see Section 35.11.3, "Getting Information from the ADF Security Context."

35.3.3 What Happens When You Generate a Default Form-Based Login Page

The wizard-generated login and error pages are simple HTML pages that are added to the top-level folder of your user interface project. The generated login page defines a form with the standard `j_security_check` action. The form takes the user name and password as input to the `j_SecurityCheck` method within the container's security model.

The wizard updates the `web.xml` file to specify Form-based authentication and identify the location of the pages, as shown in Example 35–3.

Example 35–3 Wizard-Generated Login Page Definition Added to in the web.xml FIle

```
<login-config>
   <auth-method>FORM</auth-method>
   <form-login-config>
       <form-login-page>/login.html</form-login-page>
       <form-error-page>/error.html</form-error-page>
   </form-login-config>
</login-config>
```

35.3.4 What You May Need to Know About the Configure ADF Security Wizard

The first time you run the Configure ADF Security wizard and enable authentication and authorization, you secure ADF resources at the level of the application. Additionally, you select specific project-level settings for the user interface project, including the authentication type and the authentication welcome. The wizard adds these web application settings to the `web.xml` file in the project you select. When your application contains multiple user interface projects and `web.xml` files, you can return to the wizard and configure these settings in the `web.xml` file for another user interface project that you select.

35.3.5 What You May Need to Know About ADF Authentication

Java EE container-managed security defines a standard method for login. There is no standard way to log out and stay within a secured application, so while the login process is delegated to the container, the logout process is handled by the ADF authentication servlet itself.

On the first access to a page that relies on an ADF security-aware resource, if there is no subject defined, OPSS is configured by the `JPSFilter` to create a subject containing the `anonymous` user principal and the `anonymous-role` role principal. With this role principal, the unauthenticated user will be able to access public web pages that are not associated with any ADF security-aware resources (including ADF bounded task flows or page definitions).

In the case of pages associated with ADF security-aware resources, you must explicitly grant `view` permission to `anonymous-role` to make the page accessible to the anonymous user. For details about granting privileges to the anonymous user, see Section 35.5.1, "How to Make an ADF Resource Public."

35.3.6 What You May Need to Know About the Built-In test-all Role

The Configure ADF Security wizard lets you enable automatic grants to the built-in `test-all` application role for the purpose of granting view permission to all ADF security-aware resources in your application. Without a permission grant, either an automatic view grant or an explicit grant that you define, ADF Security authorization checking enforcement would prevent you from being able to run the application and

access its resources. You can run the wizard with the `test-all` application role feature enabled and then gradually replace automatic view grants with explicit grants. Be aware that you must not deploy the application with grants to the `test-all` application role in place, since this feature makes all ADF resources public. If you choose to enable the built-in `test-all` application role in the wizard, see Section 35.9.1, "How to Remove the test-all Role from the Application Policy Store," before deploying your application.

35.3.7 What You May Need to Know About the valid-users Role

The `valid-users` role is a Java EE security role defined by ADF Security to ensure that all users will access the `adfAuthentication` servlet web resource defined in the `web.xml` file. The Configure ADF Security wizard updates the `weblogic.xml` file to map this ADF Security role to the `users` principal, as shown in Example 35–4. This mapping ensures that every user will have this role, because Oracle WebLogic Server configures all properly authenticated users as members of the `users` group.

Example 35–4 valid-users Role Mapping in the weblogic.xml FIle

```
<security-role-assignment>
   <role-name>valid-users</role-name>
   <principal-name>users</principal-name>
</security-role-assignment>
```

The `users` principal name is a special principal defined by OPSS that serves a purpose similar to an enterprise role. At runtime, the principal is added automatically to a successfully authenticated subject by OPSS. From a security perspective, the `valid-users` role supports ADF authentication only in the case where you need to control access to web resources using security constraints alone. The end result of this mapping relies entirely on Java EE security and does not involve JAAS Permissions.

35.4 Creating Application Roles

You create application roles to represent the policy requirements of the application and to define groups of users with the same view permission rights. The application roles that you create in the application policy store are specific to your application. For example, in the context of the work flow, there may be application roles such as `fod-customer`, `fod-productSpecialist`, `fod-supervisor`, and `fod-admin`, where `fod` identifies these roles as specific to the Fusion Order Demo application.

At runtime, the access rights are conferred on the user through the application role for which the user is defined as a member. Thus, before you can define security policies, the policy store must contain the application roles that you intend to issue grants to. This can be an application role that you define (such as `fod-users`) or it can be one of the two built-in application roles defined by OPSS: `authenticated-role` or `anonymous-role`. JDeveloper provides the built-in application roles to let you make ADF resources public, as described in Section 35.5.1, "How to Make an ADF Resource Public."

After you create the application role, you will:

- Grant permissions to the application roles, as described in Section 35.5, "Defining ADF Security Policies."
- Associate test users with each application role, as described in Section 35.6, "Creating Test Users."

Best Practice: The ADF Security framework enforces a role-based access control mechanism with permissions granted either to application roles or to individual users. Although you may only need to test security and therefore might not need to create groups of users, you should still create application roles (with at least one user member). Later when you define security polices on the ADF resources, the overview editor for the application policy store will allow you to select an application role for the grant.

35.4.1 How to Create Application Roles

JDeveloper lets you add application roles to the policy store of the `jazn-data.xml` file, which appears in the **Descriptors/META-INF** node of the Application Resources panel.

Note: When you create application roles, be sure to add the new application roles to the policy store, not the identity store. Roles that you add to the identity store define enterprise security roles and provide a way to conveniently group users in the identity store. For more details about enterprise roles, see Section 35.4.3, "What You May Need to Know About Enterprise Roles and Application Roles."

To create application roles in the policy store of the `jazn-data.xml` file, you use the Application Roles page of the overview editor for the `jazn-data.xml` file. This editor lets you view the relationship between identity store members and the application roles you create.

Before you begin:

It may be helpful to have an understanding of application roles. For more information, see Section 35.4, "Creating Application Roles."

To create application roles:

1. In the main menu, choose **Application** and then **Secure > Application Roles**.
2. In the Application Roles page of the `jazn-data.xml` overview editor, select the policy store for your application from the **Security Policy** dropdown list.

   The policy store that JDeveloper creates in the `jazn-data.xml` file is automatically based on the name of your application.
3. In the **Roles** list, click the **New** icon.
4. In the **Name** field, enter the name of the role and click any other field to add the application role to the policy store.
5. If you have already setup test users in the identity store, you can map users and roles, as described in Section 35.6.3, "How to Associate Test Users with Application Roles."

35.4.2 What Happens When You Create Application Roles

When you add an application role to the policy store, JDeveloper updates the `jazn-data.xml` file located in the `src/META-INF` directory relative to the application workspace. Application roles are defined in `<app-role>` elements under `<policy-store>`, as shown in Example 35–5. Because the policy store `<application>` element names the application, at runtime all application roles that

you create will be visible to your application only. Other web applications may define a policy store with their own set of application roles.

Example 35–5 Application Role Definition in the Policy Store

```
<policy-store>
     <applications>
         <application>
            <name>StoreFrontModule</name>
            <app-roles>
               <app-role>
                  <name>fod-users</name>
                  <display-name>FOD Users</display-name>
                  <class>oracle.security.jps.service.policystore.
                                                        ApplicationRole</class>
               </app-role>
               ...
            </app-roles>
            <jazn-policy>
              ...
            </jazn-policy>
         </application>
    </applictions>
</policy-store>
```

35.4.3 What You May Need to Know About Enterprise Roles and Application Roles

An *enterprise role* is a role that is maintained in the domain policy store (as opposed to an application policy store). Enterprise roles are available to every application deployed in the domain and defined in the domain identity store, typically represented by an ordered list of authenticators configured in the domain.

An *application role* is a role used by a Fusion web application. It is specific to the application, defined by the application policy, and not necessarily known to the Java EE container. Application roles are scoped in the sense that they can contain only users and roles defined in the application. Application roles must be mapped to enterprise roles.

You use the overview editor for the `jazn-data.xml` file to create enterprise roles to group users that you add to the identity store. You can use this mechanism to assign entire groups of users to application roles that you have defined for the purpose of conferring access rights defined by ADF security policies, as described in Section 35.6.3, "How to Associate Test Users with Application Roles."

However, Integrated WebLogic Server does not require you to create enterprise roles to run the application within JDeveloper. For the purpose of testing the application, it may be sufficient to create a few test users and assign them directly to application roles. When you run the application in JDeveloper, the users and any enterprise roles you defined will be created in the default security provider (which is embedded LDAP for Integrated WebLogic Server).

Typically, when you deploy the application for staging, you will migrate only the policy store to the target server. You can configure JDeveloper deployment options so that the identity store, including test users and enterprise roles, is not migrated, as described in Section 35.8.1, "How to Configure, Deploy, and Run a Secure Application in JDeveloper."

After you deploy the secure application, Oracle Fusion Middleware will merge your application's policy store with the policies of the domain-level policy store. To

complete this task, the administrator for the Oracle WebLogic Server will eventually map the application roles of your policy store to the existing domain-level enterprise roles. This application role mapping at the domain level allows enterprise users to access application resources according to the ADF security policies you have defined. The domain-level application role mapping by the administrator also allows you to develop the ADF security policies of your application without requiring any knowledge of the identity store in the production environment.

35.5 Defining ADF Security Policies

Authorization relies on a policy store that is accessed at runtime and that contains permissions that grant privileges to execute predefined actions, like `view`, on a specified object. Initially, after you run the Configure ADF Security wizard, the policy store defines no grants. And, because the default wizard option **ADF Authentication and Authorization** enables authorization checking, the web pages of your application that rely on the ADF security-aware resources will be inaccessible to users. You must use JDeveloper to define explicit grants for the resources that you want to permit users to access.

> **Best Practice:** When you run the Configure ADF Security wizard with the default option **ADF Authentication and Authorization** selected, you will lock down the web pages of your application. This affords the most protection to the Fusion web application possible since you will define explicit grants to allow users to access only the pages you intend. For a discussion of this guideline and others, see Section 35.11.4, "Best Practices for Working with ADF Security."

Before you can define security policies, the policy store for your application must contain the application roles that you intend to issue grants to. This can be an application role that you define (such as `fod-users`) or it can be one of the two built-in application roles defined by OPSS: `authenticated-role` or `anonymous-role`. You use application roles to classify users, so that each member of the same role possesses the same access rights. As such, the security policy names the application role as the principal of the grant, rather than specific users. For details about defining application roles, see Section 35.4, "Creating Application Roles."

For the user interface project, you use the `jazn-data.xml` file overview editor to secure ADF resources, including ADF task flows and ADF page definitions. You open the editor on the `jazn-data.xml` file by double-clicking the `jazn-data.xml` file (located in the Application Resources panel) or by choosing **Secure > Resource Grants** from the **Application** menu in the main menu.

Note that when you open the `jazn-data.xml` file, the overview editor provides additional editor pages that you use to create test users, enterprise roles, and application roles.

For the data model project, you do not secure entity objects or their attributes using the `jazn-data.xml` file overview editor. Instead, you set metadata directly on these objects to manage whether or not the databound UI component displays the data. For details about granting permissions for row-level security, see Section 35.5.10, "How to Define Policies for Data."

35.5.1 How to Make an ADF Resource Public

It is a common requirement that some web pages be available to all users, regardless of their specific access privileges. For example, the home page should be seen by all

visitors to the site, while a corporate site should be available only to those who have identified themselves through authentication.

In both cases, the page may be considered public, because the ability to view the page is not defined by the users' specific permissions. Rather, the difference is whether the user is anonymous or a known identity.

In the ADF security model, you differentiate between the absence of security and public access to content by granting access privileges to the `anonymous-role` principal. The anonymous role encompasses both known and anonymous users, thus permission granted to `anonymous-role` allows access to a resource by unauthenticated users, for example, guest users. To provide access to authenticated users only, the policy must be defined for the `authenticated-role` principal.

Note: For details about creating a public home page which contains links to other pages in the application, see Section 35.7.3, "How to Create a Public Welcome Page."

Before you begin:

It may be helpful to have an understanding of ADF security policies. For more information, see Section 35.5, "Defining ADF Security Policies."

You will need to complete these tasks:

1. Create bounded task flows as described in Section 18.2, "Creating a Task Flow."
2. Create web pages with an ADF page definition file as described in Section 13.7, "Working with Page Definition Files."
3. Run the Configure ADF Security wizard as described in Section 35.3, "Enabling ADF Security."
4. Create application roles as described in Section 35.4, "Creating Application Roles."

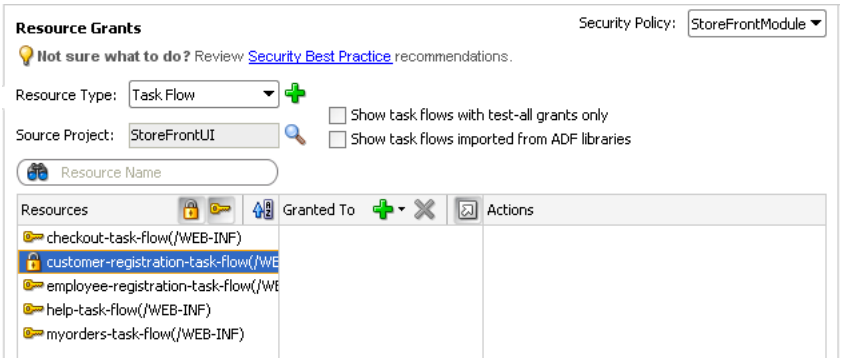
To grant public access to ADF security-aware resources:

1. In the main menu, choose **Application** and then **Secure > Resource Grants**.
2. In the Resource Grants page of the `jazn-data.xml` file overview editor, select one of the following resources from the **Resource Types** dropdown list:
   - **Task Flow** when you want to make a bounded task flow public. The application displays the web pages under the permission you define for the task flow itself. Thus, all constituent web pages of the bounded task flow will become public.
   - **Web Page** when you want to make individual web pages public. Typically, these pages are defined by an unbounded task flow and are top-level pages in the application, such as a home page.
3. In the **Resources** column, select the ADF resource for which you want to grant access rights.

   The resource you select should display the lock icon in the first column next to the resource name. The lock icon indicates that the resource has no security policy defined and therefore is "locked"—which means it remains inaccessible to users until you define a grant. For example, in Figure 35–4, the ADF resource `customer-registration-task-flow` (a bounded task flow) shows the lock icon since no grant has been made.

Tip: Click the key toggle icon in the header for the overview editor's first column to hide or show resources that already have grants and display only the resources without grants. The key icon indicates that the resource has a grant that will make the resource accessible to users with sufficient access rights.

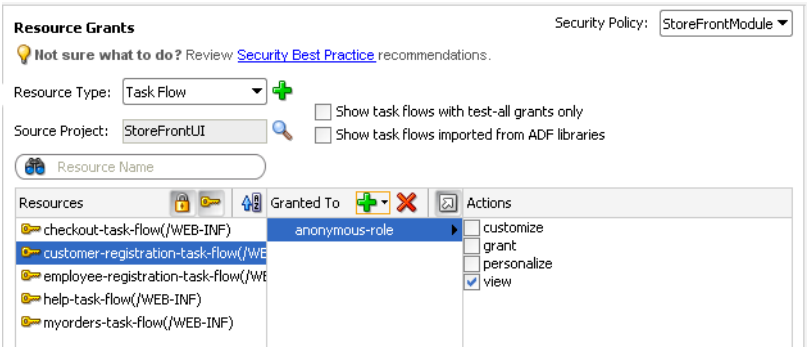
Figure 35–4 Selecting an ADF Security-Aware Resource in the Overview Editor



4. In the **Granted to** column, click the **Add Grantee** icon and choose **Add Application Role**.
5. In the Select Application Roles dialog, select one of these built-in application roles:
   - **anonymous-role** means the resource will be accessible to anyone who visits the site. A grant to this role is necessary if you want to make a web page associated with an ADF security-aware resource accessible before a user logs in. For example, you would grant to `anonymous-role` for a task flow that manages customer registration.
   - **authenticated-role** means the resource will be accessible only to authenticated users (ones who visit the site and log in). For example, you would grant to `authenticated-role` for an employee registration task flow.
6. In the Select Application Roles dialog, click **OK**.
7. In the Resource Grants page of the overview editor, in the **Actions** column, leave the **View** action selected.

   By default, the overview editor shows **View** selected, as shown in Figure 35–5. The view action is the only action currently supported for Fusion web applications.

Figure 35–5 Granting to anonymous-role in the Overview Editor



35.5.2 What Happens When You Make an ADF Resource Public

When you define a security policy, the `jazn-data.xml` file overview editor updates the `jazn-data.xml` file located in the `/src/META-INF` node relative to the web application workspace.

The overview editor writes the policy information to the `<policy-store>` section of the file. The security policy, or grant, contains both a grantee and one or more permissions. The grantee is the application role that the policy is being defined for—in this case, the anonymous role. Each permission defines the resource being secured and the action that can be performed against that resource.

Example 35–6 shows a security policy in the `jazn-data.xml` file that makes a customer registration task flow public. The grant to `anonymous-role` contains a single view permission for a bounded task flow, `customer-registration-task-flow`. With this grant, all users will be able to enter the customer registration task flow and complete the customer registration process. Additional grants to the anonymous role may be made and will appear in the `<permissions>` section of the anonymous role grant.

Example 35–6 Grants to anonymous-role in the Application-Level Policy Store

```
<policy-store>
   ...
   <jazn-policy>
      <grant>
         <grantee>
            <principals>
                <principal>
                   <class>oracle.security.jps.internal.core.
                                          principals.JpsAnonymousRoleImpl</class>
                   <name>anonymous-role</name>
                </principal>
            </principals>
         </grantee>
         <permissions>
            <permission>
                  <class>oracle.adf.controller.security.TaskFlowPermission</class>
                  <name>/WEB-INF/customer-registration-task-flow.xml#
                                            customer-registration-task-flow</name>
                  <actions>view</actions>
            </permission>
            ...
         </permissions>
      ...
      </grant>
      ...
   </jazn-policy>
</policy-store>
```

35.5.3 What Happens at Runtime: How the Built-in Roles Are Used

The `anonymous-role` and `authenticated-role` names are special roles defined by Oracle Platform Security Services (OPSS).

When you run the Configure ADF Security wizard, the wizard configures the `JpsFilter` definition in the `web.xml` file to enable support for the anonymous role. The enabled anonymous role allows ADF Security to support browsing of the site by anonymous users—those users who have not yet logged in. In contrast, the

authenticated role is not declared and is always recognized by default. ADF Security supports both of these roles.

When an end user first accesses an ADF security-aware resource, the system creates a subject and populates it with the anonymous role principal. As long as the ADF security-aware resource being accessed has the view grant to anonymous role, the user is permitted access. If the anonymous role is not a grantee of the ADF resource, the user is prompted to log in. After logging in, the authenticated role is added to the subject. The wizard also adds the `JpsFilter` definition to the `web.xml` file, where `remove.anonymous.role` set to `false` ensures that the anonymous role principal is available even after the user logs in. With the authenticated role principal, the user may access resources that have an explicit grant to the authenticated role.

35.5.4 How to Define Policies for ADF Bounded Task Flows

You define the access policy for an ADF bounded task flow by creating permission grants in the Resource Grants page of the `jazn-data.xml` file overview editor. The grants you create will appear as metadata in the policy store section of the `jazn-data.xml` file. This metadata defines a permission target (in this case, the bounded task flow definition name) for which you have issued grants to authorize the members of a specific application role.

> **Best Practice:** Do not create permission grants for the individual web pages of a bounded task flow. When the user accesses the bounded task flow, security for all pages will be managed by the permissions you grant to the task flow. And, because the contained web pages (with associated page definitions) will be inaccessible by default, ADF Security prevents users from directly accessing the pages of the task flow. This supports a well-defined security model for task flows that enforces a single entry point for all users. For further information about implementing security policies, see Section 35.11.4, "Best Practices for Working with ADF Security."

You can sort the task flows in the overview editor by clicking the toggle buttons in the **Task Flow** header, as described in Table 35–3.

Table 35–3 Resource Grant Toggle Buttons for Bounded Task Flows

Button	Toggle Action	Description
	Shows/hides bounded task flows with no grants	Represents a bounded task flow with no permission grants defined. The web pages that the task flow calls will not be accessible to any user.
	Shows/hides bounded task flows with grants	Represents a bounded task flow with one or more permission grants defined. The web pages that the task flow calls will be accessible to users who are members of the application role that received the grant.

The list of available actions displayed by the overview editor is defined by the task flow permission class (`oracle.adf.controller.security.TaskFlowPermission`). The permission class maps these actions to the operations supported by the task flow. Table 35–4 shows the actions displayed by JDeveloper for ADF bounded task flows.

Note that the `view` action is the only action currently supported for Fusion web applications. Do not select `customize`, `grant`, or `personalize` actions—they are reserved for future use in task flow security.

Table 35–4 Secured Actions of ADF Bounded Task Flows

Grantable Action	Effect on the User Interface
`view`	Controls who can read and execute a bounded task flow in a Fusion web application. This is the only operation that the task flow supports.
`customize`	Reserved for future use. This action is not checked at runtime.
`grant`	Reserved for future use. This action is not checked at runtime.
`personalize`	Reserved for future use. This action is not checked at runtime.

To define a grant for the task flow security policy, use the Resource Grants page of the overview editor for the `jazn-data.xml` file.

Before you begin:

It may be helpful to have an understanding of ADF security policies. For more information, see Section 35.5, "Defining ADF Security Policies."

You will need to complete these tasks:

1. Create bounded task flows as described in Section 18.2, "Creating a Task Flow."

   Best Practice: If you are creating bounded task flows in separate UI projects of the same application, you will want to assign unique task flow definition names. This is necessary because a grant's task flow definition name is scoped in the `jazn-data.xml` policy store by path (for example, `/WEB-INF/`*`mytaskflow`*`-definition.xml#`*`mytaskflow`*`-definition`). Therefore creating bound task flows with unique definition names is the only way to impose project-level scoping of the grants.

2. Run the Configure ADF Security wizard as described in Section 35.3, "Enabling ADF Security."
3. Create application roles as described in Section 35.4, "Creating Application Roles."

To define a permission grant on an ADF bounded task flow:

1. In the main menu, choose **Application** and then **Secure > Resource Grants**.
2. In the Resource Grants page of the `jazn-data.xml` file overview editor, select **Task Flow** from the **Resource Types** dropdown list.

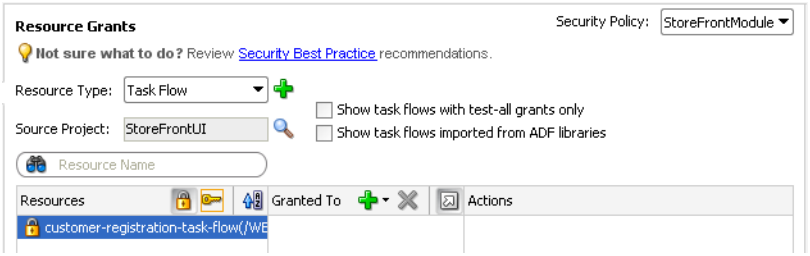
   The overview editor displays all the task flows that your application defines. Task flows are defined by task flow definition files (`.xml`) that appear in the Web Content/Page Flows node of the user interface project.

3. In the **Resources** column, select the task flow for which you want to grant access rights.

   The first time you make a grant to a bounded task flow, the first column should display the **Resources without any grants** icon (represented by the "lock" icon) next to the task flow name. The overview editor displays the lock icon to indicate that a resource has no security policy defined and therefore is "locked"—which means it remains inaccessible to users until you define a grant.

Tip: Click the **Resources with grants** icon (represented by the "key" icon) in the header for the **Resources** column to hide all task flows that already have grants. This will display only task flows without grants, as shown in Figure 35–6. Additionally, you can type a partial task flow name in the search field to display only the task flows with character-matching names.

Figure 35–6 Hiding Task Flows with Grants in the Overview Editor



4. In the **Granted to** column, click the **Add Grantee** icon and choose **Add Application Role**.
5. In the Select Application Roles dialog, select the application role that you want to make a grantee of the permission.

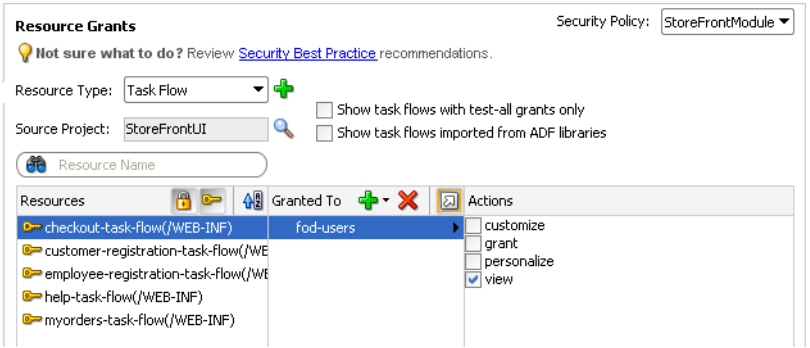
   The Select Application Roles dialog displays application roles from the `jazn-data.xml` file. It also displays the built-in OPSS application roles, **anonymous-role** and **authenticated-role**, as described in Section 35.5.3, "What Happens at Runtime: How the Built-in Roles Are Used."

   If you do not see application roles that are specific to your application, create the role as described in Section 35.4, "Creating Application Roles."
6. In the Select Application Roles dialog, click **OK**.
7. In the Resource Grants page of the overview editor, in the **Actions** column, leave the **View** action selected.

   By default, the overview editor shows **view** selected, as shown in Figure 35–7. The view action is the only action currently supported for Fusion web applications. Do not select **customize**, **grant**, or **personalize** actions—they are reserved for future use and will not be checked by ADF Security at runtime.

   The `TaskFlowPermission` class defines task flow—specific actions that it maps to the task flow's operations, as described in Table 35–4.

Figure 35–7 Granting to an Application Role for a Bounded Task Flow Definition in the Overview Editor



8. You can repeat these steps to make additional grants as desired.

   The same task flow definition can have multiple grants made for different application roles. The grants appear in the policy store definition of the `jazn-data.xml` file, as described in Section 35.5.6, "What Happens When You Define the Security Policy."

35.5.5 How to Define Policies for Web Pages That Reference a Page Definition

You define the access policy for an ADF page definition by creating permission grants in the Resource Grants page of the `jazn-data.xml` file overview editor. The grants you create will appear as metadata in the policy store section of the `jazn-data.xml` file. This metadata defines a permission target (in this case, the page definition name) for which you have issued grants to authorize the members of a specific application role.

> **Best Practice:** Create permission grants for the individual web page only when the page is not a constituent of a bounded task flow. Page-level security is checked for pages that have an associated page definition binding file *only* if the page is directly accessed or if it is accessed in an *unbounded* task flow. For further information about implementing security policies, see Section 35.11.4, "Best Practices for Working with ADF Security."

You can sort the web page definition resources in the overview editor by clicking the toggle buttons in the **Resources** header, as described in Table 35–5.

Table 35–5 Resource Grant Toggle Buttons for Web Page Definitions

Button	Toggle Action	Description
	Shows/hides top-level pages with no grants	Represents a page definition with no permission grants defined for a web page that is contained in an unbounded task flow. The web page will not be accessible to any user.
	Shows/hides top-level pages with grants	Represents a page definition with one or more permission grants defined for a web page that is contained in an unbounded task flow. The web page will be accessible to users who are members of the application role that received the grant.
	Shows/hides pages included in a bounded task flow	Represents a page definition associated with a web page that also is contained in a bounded task flow. Do not grant to these web page definitions. Instead, define a security policy for the bounded task flow.
	Shows/hides unsecurable pages (with no page definition)	Represents a web page with no page definition defined that is contained in an unbounded task flow. (Pages like this that are contained by a bounded task flow are secured by the bounded task flow's permission.) The web page will be accessible to all users since it is not secured by an associated ADF security-aware resource. Optionally, you can secure the page by adding an empty page definition file, as described in Section 35.5.8, "What You May Need to Know About Defining Policies for Pages with No ADF Bindings."

The list of available actions displayed by the overview editor is defined by the region permission class (`oracle.adf.share.security.authorization.RegionPermission`). The permission class maps these actions to the operations supported by the ADF page definition for the web page. Table 35–6 shows the actions displayed by JDeveloper for ADF page definitions.

Note that the `view` action is the only action currently supported for Fusion web applications.

Table 35–6 Securable Actions of ADF Page Definitions

Grantable Action	Effect on the User Interface
`view`	Controls who can view the page. This is the only operation that the page definition supports.

To define a grant for the page definition security policy, use the Resource Grants page of the overview editor for the `jazn-data.xml` file.

Before you begin:

It may be helpful to have an understanding of ADF security policies. For more information, see Section 35.5, "Defining ADF Security Policies."

You will need to complete these tasks:

1. Create the top-level web pages with an ADF page definition file as described in Section 13.7, "Working with Page Definition Files."

   Best Practice: If you are creating top-level web pages in separate UI projects of the same application, you will want to assign unique page file names. This is necessary because a grant's page definition name is scoped in the `jazn-data.xml` policy store by package (for example, `view.pageDefs.`*`mytoppage`*`PageDef`). Therefore creating top-level pages with unique file names is the only way to impose project-level scoping of the grants.

2. Run the Configure ADF Security wizard as described in Section 35.3, "Enabling ADF Security."
3. Create application roles as described in Section 35.4, "Creating Application Roles."

To define a permission grant on an ADF page definition:

1. In the main menu, choose **Application** and then **Secure > Resource Grants**.
2. In the Resource Grants page of the `jazn-data.xml` file overview editor, select **Web Page** from the **Resource Types** dropdown list.

   The Resource Grants page of the overview editor displays all web pages, including those that have an associated ADF page definition. This includes any web page that uses ADF data bindings or any web page for which you have created an empty page definition. Page definitions are defined by `PageDef.xml` files that appear in the Application Sources node of the user interface project.
3. In the **Page Definition** column, select the page definition for which you want to grant access rights.

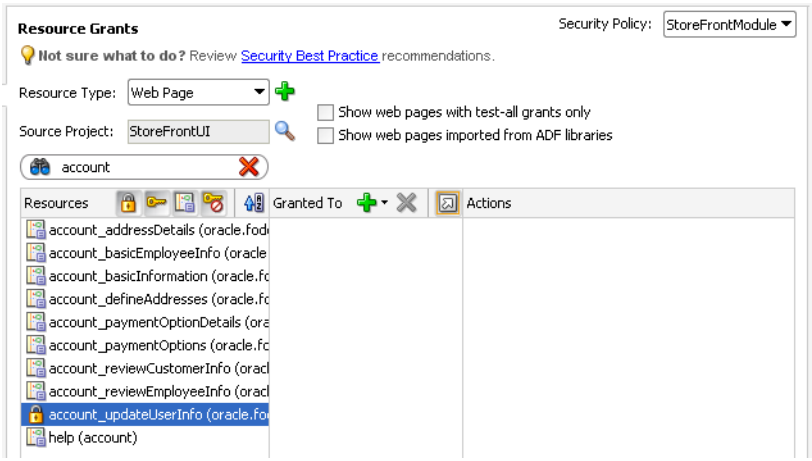
   The first time you make a grant to a page definition, the first column should display the **Resource without any grants** icon (represented by the "lock" icon) next to the page definition name. The editor displays the lock icon to indicate that a resource has no security policy defined and therefore is "locked"—which means it remains inaccessible to users until you define a grant. For example, the page definition `account_updateUserInfo` shown in Figure 35–8 displays the lock icon since no grant has been made. Other page definitions in Figure 35–8 show the

Page included in bounded task flow icon because they are not top-level pages and thus are securable by the containing bounded task flow.

Do not create grants for individual web page definitions that display the **Page included in bounded task flow** icon. Security policies for the associated web pages are secured by their bounded task flow. For example, in Figure 35–8, the page definition associated with the `account_addressDetails.jsff` region will be secured by the containing bounded task flow.

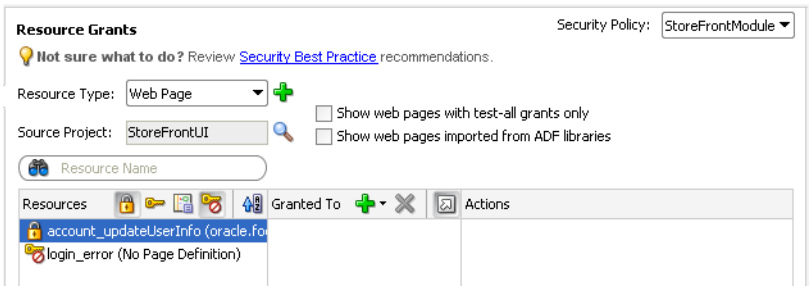
> **Tip:** You can type a partial page definition name in the search field to display only the page definitions with character-matching names. For example, a search on the word `account` would display only the page definitions that begin with the word `account`, as shown in Figure 35–8.

Figure 35–8 Matching Page Definitions by Name in the Overview Editor



> **Tip:** You can click the **Resources with grants** icon (represented by the "key" icon) to hide all page definitions that already have grants. Confirm that the **Show pages included in a bounded task flow** toggle button in the header for the **Resources** column is toggled off to hide all page definitions that are included in a bounded task flow (by default, it is set to hide these pages). This will display top-level pages that have no grants (and unsecurable pages, if any), as shown in Figure 35–9.

Figure 35–9 Hiding Web Pages with Grants in the Overview Editor



4. In the **Granted to Roles** column, click the **Add Application Role** icon.

5. In the Select Application Roles dialog, select the application role that you want to make a grantee of the permission.

   The Select Application Roles dialog displays application roles from the `jazn-data.xml` file. It also displays the built-in OPSS application roles, **anonymous-role** and **authenticated-role**, as described in Section 35.5.3, "What Happens at Runtime: How the Built-in Roles Are Used,"

   If you do not see application roles that are specific to your application, create the role as described in Section 35.4, "Creating Application Roles."

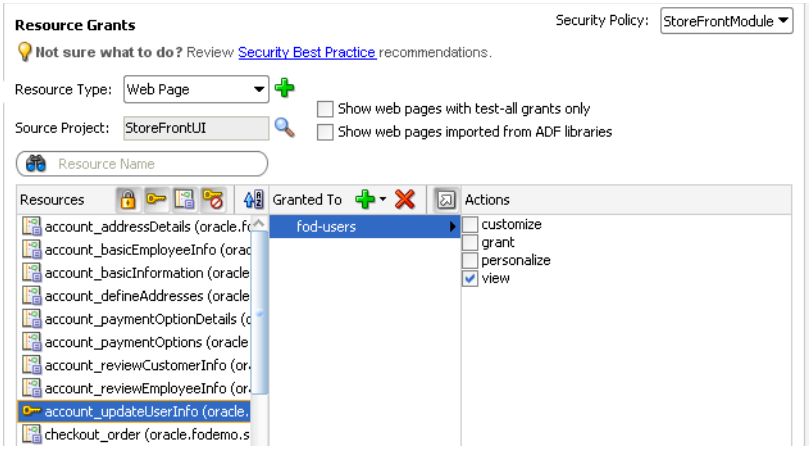
6. In the Select Application Roles dialog, click **OK**.

7. In the Resource Grants page of the overview editor, in the **Actions** column, leave the **View** action selected.

   By default, the overview editor shows **view** selected, as shown in Figure 35–10. The view action is the only action currently supported for Fusion web applications.

   The `RegionPermission` class defines page definition—specific actions that it maps to the page's operations, as described in Table 35–6.

Figure 35–10 Granting to an Application Role for an ADF Page Definition in the Overview Editor



8. You can repeat these steps to make additional grants as desired.

   The same page definition can have multiple grants made for different application roles. The grants appear in the policy store definition of the `jazn-data.xml` file, as described in Section 35.5.6, "What Happens When You Define the Security Policy."

35.5.6 What Happens When You Define the Security Policy

When you define a security policy, the `jazn-data.xml` file overview editor updates the `jazn-data.xml` file located in the `/src/META-INF` node relative to the web application workspace.

The overview editor writes the policy information to the `<policy-store>` section of the file. The security policy, or grant, contains both a grantee and one or more permissions. The grantee is the application role that the policy is being defined for. Each permission defines the resource being secured and the action that can be performed against that resource.

Example 35–7 shows a security policy in the `jazn-data.xml` file that secures a checkout task flow and secures a top-level web page that lets users update account information. The grant to the `fod-users` application role contains a view permission for a bounded task flow, `checkout-task-flow`, and a view permission on the web page with the `account_updateUserInfoPageDef` page definition. With this grant, only users who are authenticated as members of the `fod-users` application role will be able to enter the checkout task flow or view the user information update page.

For the web page, notice that permission has been defined on the `account_updateUserInfoPageDef` page definition created for the user information update page (`updateUserInfo.jspx`). Also, note that this is a top-level web page that is not already secured by a bounded task flow.

Example 35–7 Grants in the Application-Level Policy Store

```
<policy-store>
    ...
    <jazn-policy>
        <grant>
            <grantee>
                <principals>
                    <principal>
                        <class>oracle.security.jps.service.policystore.ApplicationRole</class>
                         <name>fod-users</name>
                     </principal>
                 </principals>
             </grantee>
             <permissions>
                  <permission>
                      <class>oracle.adf.controller.security.TaskFlowPermission</class>
                      <name>/WEB-INF/checkout-task-flow.xml#checkout-task-flow</name>
                      <actions>view</actions>
                  </permission>
                   <permission>
                        <class>oracle.security.jps.service.policystore.ApplicationRole</class>
                        <name>oracle.fodemo.storefront.pageDefs.acct_updateUserInfoPageDef</name>
                        <actions>view</actions>
                  </permission>
                  ...
             </permissions>
        </grant>
        ...
    </jazn-policy>
</policy-store>
```

35.5.7 What Happens at Runtime: How ADF Security Policies Are Enforced

Grants that you make for ADF resources are standard JAAS Permissions. When you enable ADF Security in your application, Oracle Platform Security Service (OPSS) running in Oracle WebLogic Server will utilize the grants to allow authorization. In authorization mode, ADF Security uses fine-grained authorization, implemented with JAAS Permissions to perform security checks for access rights to pages. The ADF Security enforcement logic checks to see whether the user, represented by the JAAS subject, has the right permissions to access the resource.

The subject contains the user's principals, which include a user principal that contains their name (could be anonymous, before logging on, or some user name after logging on), and their list of role principals, which would include `authenticated-role` and

some number of other roles that are obtained from the policy and identity stores. The principal is created to represent all of the user's memberships in application roles defined in the policy store. In turn, each application role may have multiple Permissions associated with them. These are the ADF security policies that are created through the overview editor for the `jazn-data.xml` file.

> **Note:** ADF security policies are scoped by application. This scoping allows two applications to refer to the same permission target, without producing unintentional results. You are not required to name application resources to impose application scoping of the policy store information.

Before you run the application using Integrated WebLogic Server, you will need to provision the identity store with test users and add these users to the application roles that you want to configure. The application roles can define members that are specific users or groups of users (also known as *enterprise roles*), as described in Section 35.6, "Creating Test Users."

Then at runtime, whether the current user has view permission on the page they are trying to access will be determined by the context of the page:

- If the page is an activity of a bounded task flow, the task flow controller determines the permission.
- If the page is a top-level page with an associated page definition file, the ADF model determines the permission.

Oracle Platform Security Services then checks to see whether the subject contains the roles that have the corresponding permissions needed to access the page. If the user is authorized, then the task flow is entered.

In the case of a bounded task flow and top-level pages (defined by an unbounded task flow), if the user is not authorized, ADF Controller throws an exception and passes control to an exception handler that the task flow configuration specifies. For details about specifying an error page, see Section 35.7.4, "How to Redirect a User After Authentication."

35.5.8 What You May Need to Know About Defining Policies for Pages with No ADF Bindings

The default Configure ADF Security wizard option **ADF Authentication and Authorization** enables authorization checking and secures a web page whenever the page is associated with an ADF security-aware resource. Therefore, after you run the wizard, a web page will not be secured if both of these conditions exist:

- The page does not display databound ADF Faces components and therefore no ADF page definition exists for the page.
- The page is not a constituent page of a bounded ADF task flow. (Any page that the user accesses as a process of a bounded task flow is checked under the permission of the task flow.)

JDeveloper will generate an ADF page definition file for you whenever you design a web page using the Data Controls panel to create databound ADF Faces components (for details, see Section 13.7, "Working with Page Definition Files"). However, if your web page does not use ADF bindings, you can still create an *empty* page definition file by right-clicking the web page in the user interface project and choosing **Go to Page**

Definition. The page definition file can remain empty because the page does not need to work with ADF bindings to support databound ADF Faces components.

Once you associate a web page with an ADF page definition file, empty or not, authorization checking will be enforced when the user accesses the associated web page. You can define security policies for the page as you would any other ADF page definition. For details about making grants to an empty ADF page definition, see Section 35.5.5, "How to Define Policies for Web Pages That Reference a Page Definition."

To create an empty page definition that you can define security policies for:

1. In the Application Navigator, locate the web page you want to secure, right-click the node and choose **Go to Page Definition**.
2. In the confirmation dialog, click **Yes** to create a new page definition for the page.

   The page definition will be added to the `pageDefs` package.

35.5.9 How to Use Regular Expressions to Define Policies on Groups of Resources

When you want to define a grant that applies to multiple resources at once, you can create patterns as defined by the `java.util.regex.Pattern` class to form a regular expression that gets evaluated at runtime. For example, to match a grant to a set of resources, you can enter the expression `.*` (specifies any character zero or more times) on the name of the permission. ADF Security does not support the use of regular expressions on other security objects, such as the principal name.

You might use this feature to group bounded task flows that would have the same permissions into their own subfolders of `WEB-INF` and define the grant for the entire folder, as shown in Example 35–8. In this case, the expression uses the dot character (defined as, any character) followed by the asterisk quantifier (defined as, zero or more times).

Example 35–8 Task Flow Permission for an Entire Folder Defined in the Application-Level Policy Store

```
...
<grant>
   <grantee>
      <principals>
         <principal>
            <class>oracle.security.jps.service.policystore.ApplicationRole</class>
            <name>anonymous-role</name>
         </principal>
      </principals>
   </grantee>
   <permissions>
       <permission>
          <class>oracle.adf.controller.security.TaskFlowPermission</class>
          <name>/WEB-INF/.*</name>
          <actions>view</actions>
       </permission>
   </permissions>
</grant>
```

As the overview editor for the `jazn-data.xml` file does not support the use of regular expressions in the user interface, you must edit the file directly. Do not edit the policy store of the `system-jazn-data.xml` file directly. Instead, add grants using

regular expressions to the `jazn-data.xml` file. These grants will then be merged to the policy store when you run or deploy the application.

The use of more complex regular expressions enables you to define business rules in the policy, thus creating a very targeted set of permissions. For example, you can grant the view permission on all page definitions and deny specific page definitions at the same time by defining an exclusion set in your regular expression. Example 35–9 shows how the view permission is granted to `anonymous-role` for all pages except those for which the page definition name starts with `custom`.

Example 35–9 Using Regular Expressions and Metacharacters to Define a Policy Grant

```
<grant>
   <grantee>
      <principals>
         <principal>
            <class>oracle.security.jps.service.policystore.ApplicationRole</class>
            <name>anonymous-role</name>
         </principal>
      </principals>
   </grantee>
   <permissions>
      <permission>
         <class>oracle.adf.share.security.authorization.RegionPermission</class>
         <name>[^(custom)].*</name>
         <actions>view</actions>
      </permission>
   </permissions>
</grant>
```

Table 35–7 shows some of the basic regular expression metacharacters that you can use in your policy definitions.

Table 35–7 Description of Metacharacters

Metacharacter	Description
[abc]	a, b, or c (included in list)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a to z or A to Z, inclusive (range)
[a-d[m-p]]	a to d, or m to p ~= [a-dm-p](union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, without b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p
.*	Any number of arbitrary characters (note this expression uses a dot and an asterisk together)

35.5.10 How to Define Policies for Data

ADF entity objects in the data model project are security-aware, meaning that predefined resource-specific permissions exist that a developer can grant. Additionally, you can secure just the individual attributes of entity objects.

Entity objects that you secure restrict users from updating data displayed by any web page that renders a UI component bound by an ADF binding to the data accessed by the secured entity object. Additionally, when you secure an entity object, you effectively secure any view object in the data model project that relies on that entity

object. As such, entity objects that you secure define an even broader access policy that applies to all UI components bound to this set of view objects.

Before you begin:

It may be helpful to have an understanding of ADF security policies. For more information, see Section 35.5, "Defining ADF Security Policies."

You may also find it helpful to have an understanding of entity-based view objects. For details about entity-based view objects, see Section 5.2, "Populating View Object Rows from a Single Database Table."

To secure row data using ADF entity objects:

1. Define a permission map for the specific actions of the entity object or the attributes of the entity object that you want to secure.
2. Grant the permission to an application role that you have added to the policy store.

35.5.10.1 Defining Permission Maps on ADF Entity Objects

You can secure operations of the entity objects or their individual attributes.

In the data model project, you use the overview editor for the entity object to define a permission map for the specific actions allowed by the entity object. The metadata consists of a permission class, a permission name, and a set of actions mapped to binding operations.

The list of available operations displayed by the overview editor is defined by the entity object permission class (`oracle.adf.share.security.authorization.EntityPermission`). The permission class maps the operations supported by the entity object to actions. Table 35–8 shows the securable operations of the entity object.

Table 35–8 Securable Operations of ADF Business Components

ADF Component	Securable Operation	Expected Mapped Action	Corresponding Implementation
ADF Business Components entity objects	`read`	`Read`	View the rows of a result set that has been restricted by a `WHERE` clause fragment.
	`update`	`Update`	Update any attribute of the bound collection.
	`removeCurrentRow`	`Delete`	Delete a row from the bound collection.
ADF Business Components attributes of entity objects	`update`	`Update`	Update a specific attribute of the bound collection.

To secure all row-level data that the entity object accesses, use the overview editor for the entity object.

Before you begin:

It may be helpful to have an understanding of ADF security for entity objects. For more information, see Section 35.5.10, "How to Define Policies for Data."

You will need to complete this task:

Create entity objects in the data model project as described in Chapter 4, "Creating a Business Domain Layer Using Entity Objects."

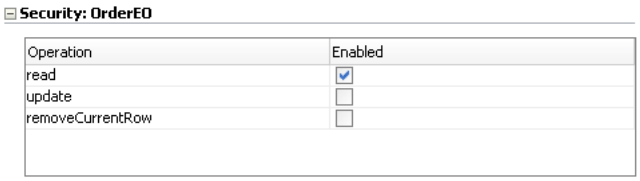
To secure an operation on an entity object:

1. In the data model project displayed in the Application Navigator, double-click the entity object that you want to secure.
2. In the overview editor, click the **General** navigation tab.
3. In the General page, expand the **Security** section and select the operations you want to secure for the entity object.

   The **Security** section displays the securable operations that the `EntityPermission` class defines. The class maps the entity object—specific actions to the entity object's operations, as described in Table 35–8.

   For example, to enable **read** permission, select it as shown in Figure 35–11. The permissions appear in the XML definition of the entity object.

Figure 35–11 Permission Enabled on read Operation for an ADF Entity Object



To secure individual columns of data that the entity object accesses, use the Attributes page of the overview editor for the entity object.

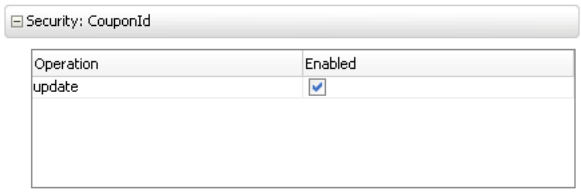
To secure an operation on an entity object attribute:

1. In the data model project displayed in the Application Navigator, double-click the entity object that defines the attribute you want to secure.
2. In the overview editor, click the **Attributes** navigation tab.
3. In the Attributes page, select the desired attribute, and then expand the **Security** section and select the operation you want to secure for the entity object attribute.

   The **Security** section displays the securable operations that the `EntityAttributePermission` class defines. The class maps the entity object-specific actions to the entity object's operations, as described in Table 35–8.

   For example, to enable **update** permission, select it as shown in Figure 35–12. The permission map appears in the XML definition of the entity object.

Figure 35–12 Permission Enabled on update Operation for an ADF Entity Object Attribute



35.5.10.2 Granting Permissions on ADF Entity Objects

Once a permission target is configured, any data that derives from entity objects or their attributes remains unsecured until you explicitly define policy grants for the entity object's permission target.

To define the access policy for an existing entity object permission target, use the Edit Authorization dialog.

Before you begin:

It may be helpful to have an understanding of ADF security for entity objects. For more information, see Section 35.5.10, "How to Define Policies for Data."

You will need to complete these tasks:

1. Run the Configure ADF Security wizard as described in Section 35.3, "Enabling ADF Security."
2. Create application roles as described in Section 35.4, "Creating Application Roles."
3. Define the permission target for the entity object or the attributes of the entity object as described in Section 35.5.10.1, "Defining Permission Maps on ADF Entity Objects."

To define the access policy for an entity object:

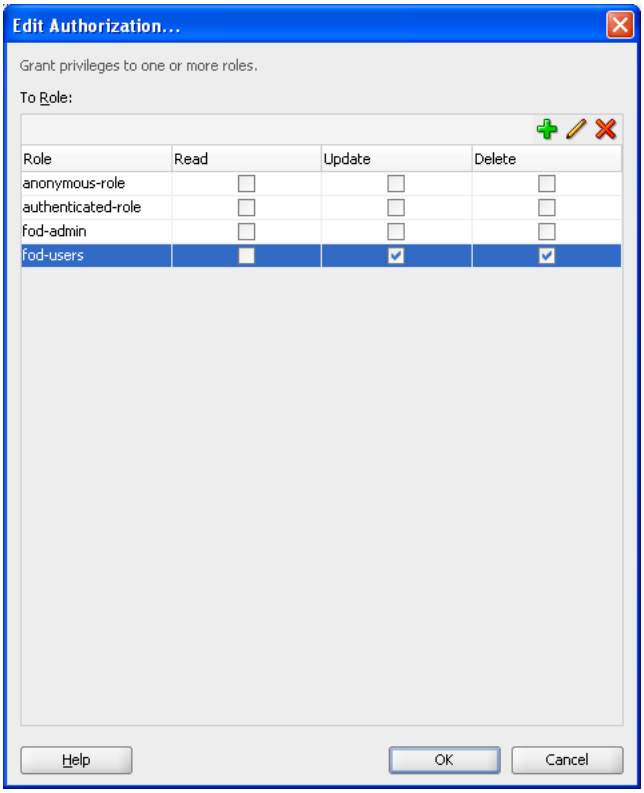
1. In the data model project displayed in the Application Navigator, locate the entity object and select it.
2. In the Structure window, locate the selected entity object or entity object attribute, then right-click the node and choose **Edit Authorization**.

   The Edit Authorization dialog displays the available actions of the entity object (or attribute), as described in Table 35–8. The dialog also displays the application roles from the `jazn-data.xml` policy store. The built-in application roles **anonymous-role** and **authenticated-role** will appear with application roles that the application developer created.
3. In the dialog, select the action that you want to grant to a specific application role.

   For example, to grant **Update** and **Delete** privileges to the **fod-users** application role, select those actions, as shown in Figure 35–13.

   The grant to the application role appears in the `jazn-data.xml` file.

Figure 35–13 Defining the Access Policy for an Entity Object



35.5.11 How to Aggregate Resource Grants as Entitlement Grants

You define end user access for individual securable Oracle ADF artifacts by creating resource grants. However, for ease of administration and maintenance, resource grants can also be defined as entitlement grants, in which case multiple securable application artifacts are aggregated into a named security group that can be granted to application roles using a single statement. For example, when you want to authorize access to the data exposed in the application by ADF Business Components entity objects, multiple entity objects may require the same security policy. Instead of individually granting access privileges to each entity object, you can group the privileges as a set in an entitlement and grant them all at once.

You create entitlement grants in the Entitlements Grants page of the `jazn-data.xml` file overview editor. The grants you create will appear as metadata in the policy store section of the `jazn-data.xml` file. This metadata defines an entitlement (identified in the XML definition as `<permission-set>`) comprised of resource instance /action pairs that you select. This entitlement is a grantable entity that you then grant to an application role.

The list of resource types appears in the security policy overview editor. The resource type you select filters the resource instances defined within the projects of your application's workspace. The resource type selection also determines the list of available actions displayed by the overview editor. For example, when you select the **Task Flow Permission** resource type, the overview editor will display all of the task flows in the user interface projects that you select and also displays the **view** action that you can associate with the available ADF bounded task flow resources.

Table 35–9 lists the resource types displayed in JDeveloper and identifies the associated resource and supported actions for each type.

Table 35–9 Resource Types of Securable Oracle ADF Artifacts

Resource Type	Supports These Resources and Actions
`Task Flow`	Defines view actions on ADF bounded task flows in a user interface project that you select.
`Web Page`	Defines view actions on regions and web pages backed by ADF page definition files in a user interface project that you select.
`ADF Entity Object`	Defines read, update, and delete actions on entity objects in a data model project that you select.
`ADF Entity Object Attribute`	Defines update actions on entity object attributes of a specific entity object in a data model project that you select.

To define an entitlement grant for a securable Oracle ADF artifact, use the Entitlement Grants page of the overview editor for the `jazn-data.xml` file. This editor is also called the security policy overview editor.

Before you begin:

You will need to complete this task:

- Create application roles as described in Section 35.4, "Creating Application Roles."

To define an entitlement grant for an Oracle ADF artifact:

1. In the main menu, choose **Application** and then **Secure > Entitlement Grants**.
2. In the Entitlement Grants page of security policy overview editor, click the **Add Entitlements** icon in the **Entitlements** list.

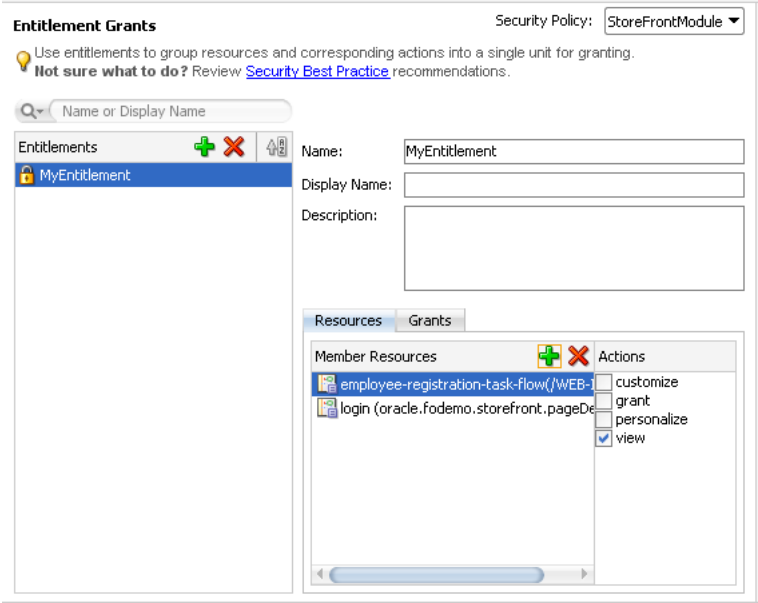
   The overview editor displays all the resources that your application defines.
3. In the Resources section, click the **Add Member Resource** icon to add a member resource to the entitlement.
4. In the Select Resources dialog, select the resource from the **Resource Type** dropdown and then select the desired project in the **Source Projects** list.

   The dialog displays all the projects in your application workspace.
5. In the **Available Resources** list, select the resource from and click the **Add** icon.

   The dialog displays all the resources define by your selected project.
6. In the **Actions** lists, select the desired action for the selected resource.

   Figure 35–14 shows the overview editor with the **View** action selected for the task flow and added to **MyEntitlement**.

Figure 35–14 Adding a Bounded Task Flow as a Resource in an Entitlement Grant



7. Add other desired resources to the list.
8. In the Grants section of the security policy overview editor, click the **Add Role Grants** icon to grant the entitlement to an application role.
9. In the Select Application Roles dialog, select one or more custom application roles.

   The dialog displays all the application roles from the `jazn-data.xml` file. You must not add a grant to a predefined application role (also called *duty roles* in the terminology of Oracle Fusion Applications). Only select custom application roles that either you created in JDeveloper or that were created by an IT security manager for this purpose.
10. Click **OK**.
11. You can repeat these steps to add other resources and make grants on those resources to the same entitlement for the same custom application role.

35.5.12 What Happens After You Create an Entitlement Grant

When you use the security policy editor in JDeveloper to create an entitlement grant, JDeveloper modifies the source for the application policy store in the `jazn-data.xml` file. The policy store section of the file contains a `<resource-type>` definition (that identifies the actions supported for resources of the selected type), a `<resource>` definition (to identify the resource instance that you selected from your application and mapped to a resource type), a `<permission-set>` definition (to define the resources and actions to be granted as an entitlement), and a `<grant>` definition with one or more entitlements (defined in the XML as a permission set) granted to the desired application roles (the grantee).

As Example 35–10 shows, entitlement-based security policies in the Oracle Fusion application are defined in the `<jazn-policies>` element and consist of one or more entitlements granted to a single application role.

Example 35–10 Entitlement-Based Policy Definition in the jazn-data.xml File

```
<?xml version="1.0" ?>
```

```
<jazn-data>
  <policy-store>
    <applications>
      <application>
        <name>MyApp</name>

        <app-roles>
        <app-role>
          <name>AppRole</name>
          <display-name>AppRole display name</display-name>
          <description>AppRole description</description>
          <guid>F5494E409CFB11DEBFEBC11296284F58</guid>
          <class>oracle.security.jps.service.policystore.ApplicationRole</class>
        </app-role>
      </app-roles>

      <role-categories>
        <role-category>
          <name>MyAppRoleCategory</name>
          <display-name>MyAppRoleCategory display name</display-name>
          <description>MyAppRoleCategory description</description>
        </role-category>
      </role-categories>

      <!-- resource-specific OPSS permission class definition -->
      <resource-types>
        <resource-type>
          <name>APredefinedResourceType</name>
          <display-name>APredefinedResourceType display name</display-name>
          <description>APredefinedResourceType description</description>
          <provider-name>APredefinedResourceType provider</provider-name>
          <matcher-class>oracle.security.jps.ResourcePermission</matcher-class>
          <actions-delimiter>,</actions-delimiter>
          <actions>write,read</actions>
        </resource-type>
      </resource-types>

      <resources>
        <resource>
          <name>MyResource</name>
          <display-name>MyResource display name</display-name>
          <description>MyResource description</description>
          <type-name-ref>APredefinedResourceType</type-name-ref>
        </resource>
      </resources>

      <!-- entitlement definition -->
      <permission-sets>
        <permission-set>
          <name>MyEntitlement</name>
          <display-name>MyEntitlement display name</display-name>
          <description>MyEntitlement description</description>
          <member-resources>
            <member-resource>
              <type-name-ref>APredefinedResourceType</type-name-ref>
              <resource-name>MyResource</resource-name>
              <actions>write</actions>
            </member-resource>
          </member-resources>
        </permission-set>
```

```
      </permission-sets>

      <!-- Oracle function security policies -->
      <jazn-policy>
        <!-- function security policy is a grantee and permission set -->
        <grant>
          <!-- application role is the recipient of the privileges -->
          <grantee>
            <principals>
              <principal>
                <class>
                   oracle.security.jps.service.policystore.ApplicationRole
                </class>
                <name>AppRole</name>
                <guid>F5494E409CFB11DEBFEBC11296284F58</guid>
              </principal>
            </principals>
          </grantee>

          <!-- entitlement granted to an application role -->
          <permission-set-refs>
            <permission-set-ref>
              <name>MyEntitlement</name>
            </permission-set-ref>
          </permission-set-refs>
        </grant>
      </jazn-policy>
    </application>
  </applications>
 </policy-store>
</jazn-data>
```

35.6 Creating Test Users

JDeveloper provides editors to help you create both the identity and the policy stores. You create both repositories in an application-specific `jazn-data.xml` file. The editor for the identity store section of the file lets you enter the list of valid user IDs and their assigned passwords. The same editor lets you create application roles and assign the test users or enterprise roles as members of the application roles. Once defined, this information appears in the policy store section of the `jazn-data.xml` file.

35.6.1 How to Create Test Users in JDeveloper

You seed the identity store of your application with a temporary set of users to simulate the actual users' experience in your production environment. When you run the application in Integrated WebLogic Server, you can log in as any test user and be conferred access rights to view the secure ADF resources of your application.

You can use the identity store to organize users into enterprise roles. Because you typically will configure JDeveloper's deployment options to prevent migrating the identity store to a staging environment, enterprise roles that you create in the `jazn-data.xml` file are for convenience only. For more details about the use of enterprise roles, see Section 35.4.3, "What You May Need to Know About Enterprise Roles and Application Roles."

Caution: If you choose to deploy the identity store to your standalone server, you must not create users and enterprise roles in your local identity store that are already configured for Oracle WebLogic Server. For example, if you were to deploy the identity store with the user `weblogic` and enterprise role `Administrators`, you would overwrite the default administration configuration on the target server. For a complete list of global roles that Oracle WebLogic Server installs by default, see the *Oracle Fusion Middleware Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

To enable the user to view resources, you make grants against application roles rather than against the users who are the members of those roles. Therefore, after you seed the identity store with test users, you must associate each user or enterprise role group with an application role. This association confers the access rights defined by ADF security policies to users. For details about conferring access rights to users, see Section 35.6.3, "How to Associate Test Users with Application Roles."

Before you begin:

It may be helpful to have an understanding of the identity store. For more information, see Section 35.6, "Creating Test Users."

To create test users and groups:

1. In the main menu, choose **Application** and then **Secure > Users**.
2. In the Users page of the `jazn-data.xml` file overview editor, select the realm for your application from the **Realm** dropdown list and perform the following steps.

   JDeveloper automatically creates the default realm `jazn.com`.

   a. In the **Users** list, click the **New User** icon.

   b. In the **Name** field, enter the user name.

      You should avoid choosing a user name already configured for Oracle WebLogic Server (for example, do not enter `webcenter`). For the list of user names installed by Oracle WebLogic Server, see the *Oracle Fusion Middleware Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

   c. In the **Password** field, enter the password for the user and click any other field to add the password to the identity store.

      The password must contain at least eight characters and at least one of the characters must be a special character (such as !, %, ^, &, $ and so on).

3. Optionally, in the `jazn-data.xml` file overview editor, click the **Enterprise Roles** navigation tab, and select the realm for your application from the **Realm** dropdown list and perform the following steps.

   You create enterprise roles only when you want to organize users into groups that you will add to an application role. For the purpose of creating test users to run the application using Integrated WebLogic Server, you do not need to create enterprise role groups.

   a. In the **Enterprise Roles** list, click the **New Role** icon.

   b. In the **Name** field, enter the name of the enterprise role and click any other field to add the role to the identity store.

If you create enterprise role groups, you should avoid choosing a role name that is already configured for Oracle WebLogic Server (for example, do not enter `Administrators`). For a complete list of the default group names installed by Oracle WebLogic Server, see the *Oracle Fusion Middleware Securing Resources Using Roles and Policies for Oracle WebLogic Server*.

35.6.2 What Happens When You Create Test Users

When you provision the identity store with user identities and enterprise role groups, JDeveloper updates the `jazn-data.xml` file located in the `/src/META-INF` node relative to the web application workspace.

The dialog writes the user information to the `<jazn-realm>` section of the file corresponding to the identity store. Each user identity has a user name and a user login password. Each enterprise role contains one or more member users.

Example 35–11 shows the identity store in the `jazn-data.xml` file with two users and two enterprise roles. The users `ahunold` and `sking` are both members of the `fod-users` enterprise role, while only `sking` is a member of the `fod-admin` enterprise role.

Example 35–11 Users and Enterprise Roles in the Application-Level Identity Store

```
<jazn-data>
   <jazn-realm default="jazn.com">
      <realm>
         <name>jazn.com</name>
         <users>
            <user>
               <name>sking</name>
               <guid>09FC5C61F68111DCAF1DB790A6B3BAC5</guid>
               <credentials>{903}A0VQ5ozADte7EKIJzcTi6xMZ7YDpRXY5</credentials>
            </user>
            <user>
               <name>ahunold</name>
               <guid>09FEA650F68111DCAF1DB790A6B3BAC5</guid>
               <credentials>{903}/SQSrKZYCLW068VJpHaodILd48mJI47w</credentials>
            </user>
            ...
         </users>
         <roles>
            <role>
               <name>fod-users</name>
               <guid>0A084340F68111DCAF1DB790A6B3BAC5</guid>
               <members>
                  <member>
                     <type>user</type>
                     <name>sking</name>
                  </member>
                  <member>
                     <type>user</type>
                     <name>ahunold</name>
                  </member>
                  ...
               </members>
            </role>
            <role>
               <name>fod-admin</name>
               <guid>0A0CFE30F68111DCAF1DB790A6B3BAC5</guid>
               <members>
```

```
                <member>
                   <type>user</type>
                   <name>sking</name>
                </member>
                ...
             </members>
          </role>
          ...
       </roles>
    </realm>
    ...
 </jazn-realm>
</jazn-data>
```

35.6.3 How to Associate Test Users with Application Roles

Because the ADF Security framework enforces a role-based access control mechanism with permissions granted to application roles, you define a set of roles in the policy store that are specific to your application. For example, in the context of the work flow, there may be roles such as customer, product specialist, supervisor, and administrator.

After you create an application role, you can proceed to associate users that you created in the identity store with one or more roles. At runtime, users who are members of an application role will be conferred the access rights of their application roles. You can assign a user to more than one application role when you want to confer the right of multiple resource grants to a particular user.

For example, one authenticated user might belong to the supervisor role and an employee role, while another user might belong only to the employee role. The security policy for a bounded task flow that permits customer records to be browsed and edited may confer view permission to the supervisor role and limit view permission to the browse page for the employee role. Thus, grants to application roles support multiple levels of access. If the authenticated user is not a member of an application role with a view permission grant for the target ADF resource, the security framework will return an unauthorized user message.

Before you begin:

It may be helpful to have an understanding of the identity store. For more information, see Section 35.6, "Creating Test Users."

You will need to complete these tasks:

1. Run the Configure ADF Security wizard as described in Section 35.3, "Enabling ADF Security."
2. Create application roles as described in Section 35.4, "Creating Application Roles."
3. Define security policies for ADF security-aware resources as described in Section 35.5, "Defining ADF Security Policies."
4. Create test users, and, optionally, create enterprise role groups as described in Section 35.6.1, "How to Create Test Users in JDeveloper."

To associate users with application roles:

1. In the main menu, choose **Application** and then **Secure > Application Roles**.
2. In the Application Roles page of the `jazn-data.xml` file overview editor, select the policy store for your application from the **Security Policy** dropdown list.

The policy store that JDeveloper creates in the `jazn-data.xml` file are automatically based on the name of your application.

3. In the **Roles** list, select an existing application role and complete these tasks as appropriate:

   a. In the **Mappings** section, click the **Add User or Role** icon dropdown menu and choose **Add User**, then in the Select Users dialog select the previously created user from the list and click **OK**.

   b. Optionally, if you have defined enterprise roles in the identity store, in the **Mappings** section, click the **Add User or Role** icon dropdown menu and choose **Add Enterprise Role**, then in the Select Enterprise Roles dialog select the previously created enterprise role from the list and click **OK**.

35.6.4 What Happens When You Configure Application Roles

When you associate users with application roles, JDeveloper updates the `jazn-data.xml` file located in the `/src/META-INF` node relative to the web application workspace.

The dialog writes the user information to the `<policy-store>` section of the file. Each application role contains one or more member users or enterprise roles.

Example 35–12 shows the policy store in the `jazn-data.xml` file with the `fod-users` application role, which contains two members, `sking` and `ahunold`.

Example 35–12 Users Associated with Application Roles in the Application-Level Policy Store

```
<policy-store>
     <applications>
         <application>
            <name>StoreFrontModule</name>
            <app-roles>
               <app-role>
                  <name>fod-users</name>
                  <display-name>FOD Users</display-name>
                  <class>oracle.security.jps.service.policystore.ApplicationRole</class>
                  <members>
                     <member>
                        <class>oracle.security.jps.internal.core.principals.JpsXmlUserImpl</class>
                        <name>sking</name>
                     </member>
                     <member>
                        <class>oracle.security.jps.internal.core.principals.JpsXmlUserImpl</class>
                        <name>ahunold</name>
                     </member>
                     ...
                  </members>
               </app-role>
               ...
            </app-roles>
            <jazn-policy>
              ...
            </jazn-policy>
         </application>
    </applictions>
</policy-store>
```

35.7 Creating a Login Page

ADF Security allows for implicit and explicit authentication:

- In an *implicit authentication* scenario, if a user who is not yet authenticated tries to access a web page associated with ADF security-aware resources that are not granted to `anonymous-role`, then authentication is triggered dynamically. After the user successfully logs in, another check will be done to verify whether the authenticated user has view access granted on the requested page's ADF security-aware resource.
- In an *explicit authentication* scenario, your application has a public page that displays a login link, which, when clicked, triggers an authentication challenge to log in the user. The login link may optionally specify some other target page that should be displayed (assuming the authenticated user has access) after the successful authentication.

The implicit authentication scenario is handled for you by default when you run the Configure ADF Security wizard, as described in Section 35.3.5, "What You May Need to Know About ADF Authentication."

To handle the explicit authentication scenario you need to replace the `allPages` security constraint added to the `web.xml` file by the Configure ADF Security wizard and trigger an authentication challenge using the security constraint defined for the ADF authentication servlet.

> **Note:** This procedure for creating the user interface for a JSF-based login page relies on APIs that are specific to Oracle WebLogic Server and the login process that it supports. Only use this procedure when you will deploy the Fusion web application to Oracle WebLogic Server.

To explicitly handle user authentication:

1. Create a login link component and add it to the public home web page for your application.
2. Create a managed bean using API specific to Oracle WebLogic Server to handle the login attempts by the user.
3. Create a JSF login page using ADF Faces components.
4. Configure the container-managed deployment descriptor (`web.xml` file) to use the ADF Faces login page.

35.7.1 How to Create a Login Link Component and Add it to a Public Web Page

You can create a standard login link component that can be added to any page in your application to enable users to authenticate or subsequently log off. This component keeps track of the authenticated state of the user and returns the appropriate login or logout URLs and icons. The login link component will redirect users back to a specific page once they are authenticated. Hence, using this login link component provides you with a single, consistent object.

To the unauthenticated user, the login link component will look similar to Figure 35–15.

Figure 35–15 Component Before User Logs In

Then when the user clicks the login link and logs in as a user with the appropriate credentials, the component will look similar to Figure 35–16.

Figure 35–16 Component After User Logs In

Before you begin:

It may be helpful to have an understanding of ADF authentication. For more information, see Section 35.7, "Creating a Login Page."

You will need to complete this task:

Copy your login and logout image files (GIF, JPG, or PNG files) to the `public_html` directory of your project.

Note: The images used should reference the appropriate skin image if your application uses skins. For more information about skins, see the "Customizing the Appearance Using Styles and Skins" chapter in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

To create the login link component and add it to a page:

1. In the Application Navigator, double-click the web page that will display the component.
2. In the ADF Faces page of the Component Palette, select **Go Image Link** and drag it onto the page.
3. In the Structure window, right-click **af:goImageLink** and choose **Go to Properties**.
4. In the Property Inspector, set the **Text** field for the Go Image Link component to render the link text specified by a conditional Expression Language (EL) expression.

   For example, you can enter an EL expression similar to this:

   ```
   #{securityContext.authenticated ? &quot;Click to log out&quot; :
                                     &quot;Click to log in&quot;}
   ```

   The `authenticated` property of the `securityContext` bean will evaluate to true and render the log out option if the current user is authenticated. Otherwise, the link is rendered with the login option.
5. In the Property Inspector, set the **Destination** field for the Go Image Link component to forward the user to the URL specified by a conditional EL expression.

   For example, you can enter an EL expression similar to this:

```
#{securityContext.authenticated ? &quot;/adfAuthentication?logout=true&amp;
                                    end_url=/faces/welcome.jspx&quot; :
                &quot;/adfAuthentication?success_url=/faces/welcome.jspx&quot;}
```

When the user clicks the link, the URL parameters `end_url` and `success_url` forward the destination to the ADF authentication servlet. The `authenticated` property of the `securityContext` bean will evaluate to true and forward to the welcome page if the current user is authenticated. When the user is not authenticated, there is no need to forward to the login page because the ADF authentication servlet triggers log in, which is handled by the container-managed security configuration. Note that log out is handled by the ADF authentication servlet which invalidates the session. Although log out is handled by ADF Security, the browser cache must be cleared to complete the process. For a description of a known issue with Basic type authentication and browser caching, see Section 35.7.5, "What You May Need to Know About ADF Servlet Logout and Browser Caching."

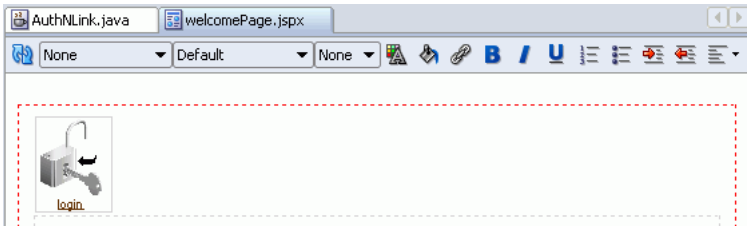
6. In the Property Inspector, set the **Icon** field for the Go Image Link component to render the link component image specified by a conditional EL expression.

   For example, this expression renders the link component image as the lock GIF if the user is not authenticated; otherwise, renders the image with the key GIF:

   ```
   #{securityContext.authenticated ? "/images/lock.gif" : "/images/key.gif"}
   ```

   Figure 35–17 shows how the login link component appears when added to the global menu facet of the page.

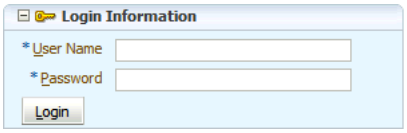
Figure 35–17 Login Link Component on the Page



35.7.2 How to Create a Login Page

The default login form that is generated for you when you run the Configure ADF Security wizard is provided as a convenience for testing your application within JDeveloper. The default form does not allow you to customize the page using ADF Faces components to match the user interface of the application. You can replace the default form with an ADF Faces-based login page that enables you to include customizable components, as shown in Figure 35–18.

Figure 35–18 Login Page



However, if designing a login page with ADF Faces components is not a requirement, then a simple JSP or HTML login page can be also used. For details about generating a

simple login page when running the Configure ADF Security wizard, see Section 35.3.1, "How to Enable ADF Security."

Web applications typically have a notion of public pages and allow for explicit as well as implicit authentication. This means that users can log in to the application by clicking the login link before they navigate to secured content, or they can navigate to a secured page, which will redirect them to the login page for the application. For more information about implicit and explicit authentication, see Section 35.8.4, "What Happens at Runtime: How ADF Security Handles Authentication."

35.7.2.1 Creating Login Code for the Backing Bean

Before you create the login page as an ADF Faces page, you need to create a managed bean to handle login attempts. Container-based authentication relies on the `j_SecurityCheck` method within the container's security model. The login method that your bean defines use this method to enforce authentication. You will add this bean to the `adfc-config.xml` file and register it with `request` scope.

> **Note:** Note that the backing bean you will create in this procedure relies on APIs that are specific to Oracle WebLogic Server and the login process that it supports. Only use this procedure when you will deploy the Fusion web application to Oracle WebLogic Server.

Before you begin:

It may be helpful to have an understanding of the login page. For more information, see Section 35.7.2, "How to Create a Login Page."

You will need to complete this task:

- Create a new library that contains the following classpath entries and add it to the user interface project where you create the backing bean for login:
  - `modules/com.bea.core.weblogic.security.auth_1.1.0.0.jar`
  - `oracle.wlserver_10.3/server/lib/wls-api.jar`

  This library will allow the user interface project to compile and resolve the following import statements from the backing bean code sample.

```
import weblogic.security.SimpleCallbackHandler;
import weblogic.security.services.Authentication;
import weblogic.servlet.security.ServletAuthentication;
```

  For details about creating a library and adding it to a project, see the "Working with Applications and Projects" chapter in the *Oracle Fusion Middleware User Guide for Oracle JDeveloper*.

To create and register a backing bean for login:

1. In the Application Navigator, right-click the user interface project node and choose **New**.
2. In the New Gallery, expand **General**, select **Java** and then **Java Class**, and click **OK**.
3. In the Create Java Class dialog, enter the name for the login page backing bean class file and disable the default options to generate **Constructors from Superclass** and **Implement Abstract Methods**, and click **OK**.

For convenience, you might name the backing bean based on the name of your login page, for example, ***LoginPageName*.java**.

4. In the Applications Navigator, expand the **Application Sources** node and double-click the new ***LoginPageName*.java** backing bean.
5. In the source editor, create two private fields by adding the following in the declaration section of the *LoginPageName*.java file:

```
private String _username;
private String _password;
```

6. Generate or create public accessors for both field.

   You can right-click in the source editor and choose **Generate Accessors** to add the following public accessors to the file:

```
public void setUsername(String _username) {
    this._username = _username;
}

public String getUsername() {
    return _username;
}

public void setPassword(String _password) {
    this._password = _password;
}

public String getPassword() {
    return _password;
}
```

7. Add a doLogin() method to this Java class to handle user attempts to log in:

```
1 public String doLogin() {
2   String un = _username;
3   byte[] pw = _password.getBytes();
4   FacesContext ctx = FacesContext.getCurrentInstance();
5   HttpServletRequest request =
6     (HttpServletRequest)ctx.getExternalContext().getRequest();
7   CallbackHandler handler = new SimpleCallbackHandler(un, pw);
8   try {
9     Subject mySubject = Authentication.login(handler);
10     ServletAuthentication.runAs(mySubject, request);
11     ServletAuthentication.generateNewSessionID(request);
12     String loginUrl = "/adfAuthentication?success_url=/faces" +
13       ctx.getViewRoot().getViewId();
14     HttpServletResponse response =
15       (HttpServletResponse)ctx.getExternalContext().getResponse();
16     sendForward(request, response, loginUrl);
17   } catch (FailedLoginException fle) {
18     FacesMessage msg = new FacesMessage(FacesMessage.SEVERITY_ERROR,
19                                         "Incorrect Username or Password",
20                                         "An incorrect Username or Password" +
21                                         " was specified");
22     ctx.addMessage(null, msg);
23   } catch (LoginException le) {
24     reportUnexpectedLoginError("LoginException", le);
25   }
26   return null;
27 }
```

The `doLogin()` method performs the following tasks:

Lines 5-6 get an object encapsulating the HTTP request from the `FacesContext`.

Line 7 creates a `CallbackHandler`, which is an object that retrieves information for security operations. A `SimpleCallbackHandler` allows security operations to retrieve the username and password that were passed to its constructor; other `CallbackHandler` implementations can obtain the username and password from another source.

Line 9 creates a `Subject`, which is an object that encapsulates credentials, from the information provided by the `CallbackHandler`.

Line 10 attempts to log the user issuing the request in using the credentials encapsulated by the `Subject`.

Line 11 ensures that the session ID for the session is changed after the user is successfully authenticated. This is necessary to prevent leaving the application open to a session fixation attack, which would be a security vulnerability.

Lines 12-13 construct a URL to which to forward the user. The `getViewId()` call returns the path, from the context root, to the currently rendered page. So, if line executes when the user is on the `home.jspx` page, `forwardURL` will be set to `/adfAuthentication?success_url=/faces/home.jspx`.

Lines 14-15 retrieve an object which encapsulates the HTTP response from the `FacesContext`.

Line 16 calls a method, `sendForward()`, which you will implement later in this section to forward the user to the URL specified in Lines 12-13.

Lines 17-22 handle a `FailedLoginException`, which is the exception thrown when the credentials supplied are incorrect. The lines handle the exception by adding a new message to the `FacesContext`.

Lines 23-24 handle a `LoginException`, which can be thrown by many different problems with a login. For example, exceptions can result from incorrect credentials or attempts to log into a locked account or uses of an expired password. `FailedLoginException` is a subclass of `LoginException`, but since a `FailedLoginException` will be caught by Line 17, these lines will only be executed when there are login problems other than incorrect credentials. `reportUnexpectedLoginError()` is a method which you will implement to deal with miscellaneous problems with the login process.

Line 26 returns null so that the ADF Controller will not attempt to follow a control flow case.

8. Import the following classes:

```
javax.faces.application.FacesMessage
javax.faces.context.FacesContext
javax.security.auth.Subject
javax.security.auth.callback.CallbackHandler
javax.security.auth.login.FailedLoginException
javax.security.auth.login.LoginException
javax.servlet.http.HttpServletRequest
javax.servlet.http.HttpServletResponse
weblogic.security.SimpleCallbackHandler
weblogic.security.services.Authentication
weblogic.servlet.security.ServletAuthentication
```

9. Create stubs for the methods `sendForward()` and `reportUnexpectedLoginError()`.

10. Add a `sendForward()` method with its actions:

```
1 private void sendForward(HttpServletRequest request,
2                          HttpServletResponse response,
3                          String forwardUrl){
4   FacesContext ctx = FacesContext.getCurrentInstance();
5   RequestDispatcher dispatcher = request.getRequestDispatcher(forwardUrl);
6   try {
7     dispatcher.forward(request, response);
8   } catch (ServletException se) {
9     reportUnexpectedLoginError("ServletException", se);
10   } catch (IOException ie) {
11     reportUnexpectedLoginError("IOException", ie);
12   }
13   ctx.responseComplete();
14 }
```

    The `sendForward()` method performs the following tasks:

    Lines 5 creates a `RequestDispatcher`, which forwards a response to a particular URI.

    Line 7 uses the `RequestDispatcher` to forward the current HTTP response to the URL.

    Line 8-9 handle a `ServletException`, which is thrown when the resource at the `RequestDispatcher`'s target URI throws an exception.

    Line 10-11 handle an `IOException`, which is thrown when the request cannot be read or the response cannot be written to.

    Lines 13 marks the HTTP response as complete so that the browser can finish rendering it.

11. Import the following classes:

```
javax.servlet.RequestDispatcher
javax.servlet.ServletException
java.io.IOException
```

12. Implement a `reportUnexpectedLoginError()` method:

```
private void reportUnexpectedLoginError(String errType, Exception e){
  FacesMessage msg =
    new FacesMessage(FacesMessage.SEVERITY_ERROR, "Unexpected error
                                                      during login",
                     "Unexpected error during login (" + errType + "),
                        please consult logs for detail");
  FacesContext.getCurrentInstance().addMessage(null, msg);
  e.printStackTrace();
}
```

    This `reportUnexpectedLoginError()` method adds a summary error message to the `FacesContext`, and then prints the full stack trace of the exception to the console.

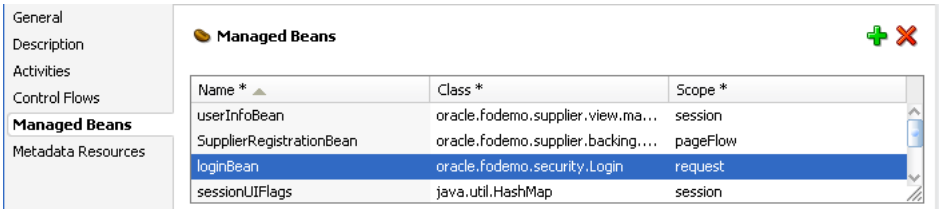
13. Save the Java file.

14. In the Application Navigator, double-click the **adfc-config.xml** node in the **WEB-INF** node.

15. In the editor window for the `adfc-config.xml` file, click the **Overview** tab.
16. In the overview editor for task flows, click the **Managed Beans** navigation tab.
17. In the Managed Beans page, in the **Managed Beans** section click the **Add** icon and enter a name for the bean, enter the fully qualified class name, and select `request` scope.

    For example, the class name might look like `oracle.foddemo.security.Login`, as shown in Figure 35–19.

Figure 35–19 Login Bean Registered in adfc-config.xml File



18. Save all.

35.7.2.2 Creating an ADF Faces-Based Login Page

A simple login page that utilizes ADF Faces layout components and ADF Faces user interface components includes two input fields and a button. You must bind the properties of these UI components to the login handler methods that you defined in the managed bean for the login page.

Before you begin:

It may be helpful to have an understanding of the login page. For more information, see Section 35.7.2, "How to Create a Login Page."

You will need to complete this task:

Create the managed bean to handle the user's login attempts as described in Section 35.7.2.1, "Creating Login Code for the Backing Bean."

To create the ADF Faces-based login page:

1. In the Application Navigator, right-click the user interface project node and choose **New**.
2. In the New Gallery, expand **Web Tier**, select **JSF** and then **JSF Page**, and click **OK**.
3. In the Create JSF Page dialog, select **Create as XML Document (*.jspx)**.
4. In the **File Name** field, specify a name for your login page. For example, enter `LoginPage.jspx`.

   Select no other options and do not select the option to expose UI components in a managed bean. You will manually bind the components to the managed bean you created for the login page.
5. Click **OK**.
6. Save the page.
7. In the ADF Faces page of the Component Palette, from the Layout panel, drag and drop a **Panel Box** onto the Structure window below the **af:form** node.
8. In the Property Inspector, set the **Text**, **Horizontal**, **Width**, and **Height** fields.

For example, to recreate the login page shown in the Master Price List module of the Fusion Order Demo application, you would enter:

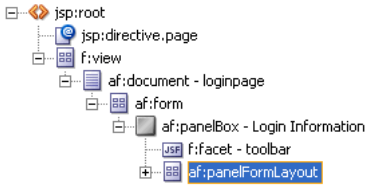
Text set to `Login Information`

Icon set to `/images/key_ena.png`

Width/**Height** set to `300`/`200` pixels

9. From the Component Palette, drag and drop a **Panel Form Layout** below the **af:panelBox** node in the Structure window, as shown in Figure 35–20.

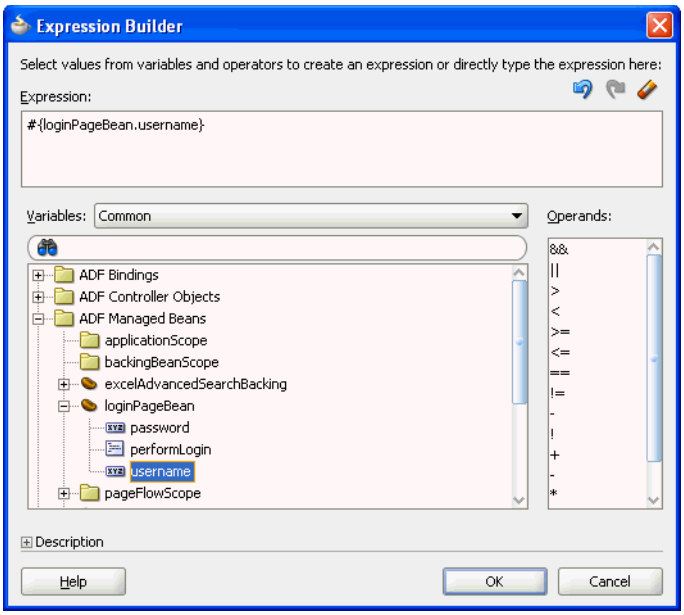
Figure 35–20 Login Page Structure with Panel Form Layout



10. From the Component Palette, in the ADF Faces page, drag and drop a **Input Text** onto the **Panel Form Layout** node for the username field and another **Input Text** for the password field.
11. In the Property Inspector for the input fields, set the **Label** fields to `Username` and `Password` and set both fields' **Behavior - Required** field to **true**.
12. In the Property Inspector, set the password field's **Appearance - Secret** field to **true**.
13. To handle processing of the values for the two input fields, perform these steps for each field:
    a. In the Structure window, select one of the input fields (for example, select the **af:inputText - Username** node), then in the Property Inspector click the **Property Menu** dropdown menu to the right of the **Value** field and choose **Expression Builder**.
    b. In the Expression Builder, expand **ADF Managed Beans** and expand your login bean, then select the expression value corresponding to your bean's handler method.

       For example, if you selected the **username** input field in the Structure window, then you would select **username** from the Expression Builder dialog, as shown in Figure 35–21.

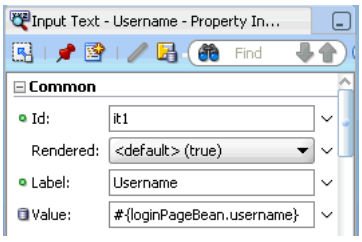
Figure 35–21 username Selection in Expression Builder Dialog



c. Click **OK**.

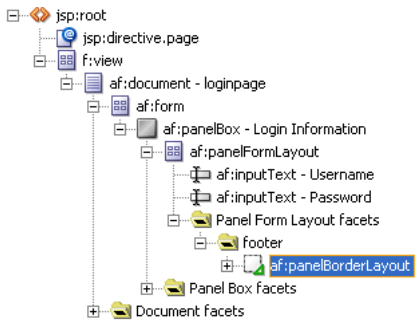
The expression shown in the Property Inspector binds the field to the managed bean you created for the login page. For example, for the username input field, the expression is similar to `#{loginPageBean.username}` as shown in Figure 35–22.

Figure 35–22 username Value in Property Inspector



14. From the Component Palette, drag and drop a **Panel Border Layout** inside the **footer** node of the **af:panelBox** node, as shown in Figure 35–23.

Figure 35–23 Login Page Structure with Panel Border Layout

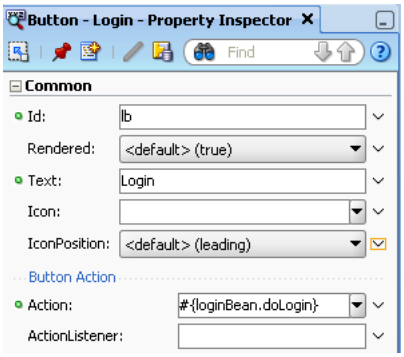


15. In the JSP/HTML visual editor, delete the panel border layout facets labeled **End**, **Top**, and **Bottom**. Leave only the **Start** facet.
16. In the Structure window, expand **Panel Border Layout facets** and select the **start** node, and then from the Component Palette drag and drop a **Button**.
17. In the Property Inspector, set the button component's **Text** field to `Login`.
18. To handle processing of the login button action, perform these steps.
    a. In the Structure window, select **af:commandButton** under the **start** node, then in the Property Inspector click the **Property Menu** dropdown menu to the right of the **Action** field and choose **Expression Builder**.
    b. In the Expression Builder, expand **ADF Managed Beans** and expand your login bean, then select the expression value corresponding to your login method.

       For example, if you created a method in the login bean named `doLogin()`, then you would select **doLogin** in the Expression Builder dialog.
    c. Click **OK**.

       The expression shown in the Property Inspector binds the button to the managed bean you created for the login page. For example, for the login button, the expression is similar to `#{loginPageBean.doLogin}` as shown in Figure 35–24.

Figure 35–24 login Action in Property Inspector



19. Save the page.

35.7.2.3 Configuring the web.xml File for an ADF Faces-Based Login Page

Because the login page is called directly from the container, it is not part of the ADF Faces navigation process. As such, you must force a call to the ADF Faces servlet when calling the login page.

You can accomplish this in the Authentication Type page of the Configure ADF Security wizard when you configure ADF Security, or in the `web.xml` file directly. If you have already run the Configure ADF Security wizard, you can use the following procedure to confirm that the `web.xml` file has been updated as described.

Before you begin:

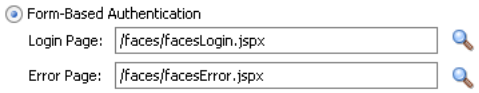
It may be helpful to have an understanding of the login page. For more information, see Section 35.7.2, "How to Create a Login Page."

To reference a login page as part of the ADF Faces lifecycle:

1. In the Application Navigator, expand the **WEB-INF** node and double-click **web.xml**.
2. In the overview editor, click the **Security** navigation tab.
3. In the Security page, expand the **Login Authentication** section, and set the login page to include a reference to the ADF Faces servlet such that the login page can be part of the ADF Faces lifecycle `/faces/ADFlogin.jspx` page.

   When you add a page using the file browser, the path entered in the `web.xml` file will not specify `/faces`. Modify the entry so that the path references the servlet mapping path for the ADF Faces servlet. For example, if the URL pattern specified by the mapping is `/faces/*`, then your path should look like `/faces/`*`yourpage`*`.jspx`, as shown in Figure 35–25.

Figure 35–25 Adding a Reference to the Faces Servlet in the Login Configuration



35.7.2.4 Ensuring That the Login Page Is Public

Because the application is secured by ADF Security, all web pages defined within bounded task flows and any web page defined by an ADF page definition will be inaccessible by default. Since all users must be allowed to log on, the login page should remain publicly accessible, and thus you should add no databound components to the page. As long as the login page uses no databound components, then it will be accessible by default.

No further steps are required to ensure that the container will always redirect to the defined authentication point before allowing access to the page (which in this case is the authentication page).

35.7.2.5 Ensuring That the Login Page's Resources Are Accessible

When you run the ADF Security wizard and choose the **ADF Authentication** option (because you do not want to enable ADF authorization) if your application uses a custom login page or a custom error page, you may need to edit the default Java EE security constraint added to the `web.xml` file by ADF Security. As shown in Figure 35–12, the default URL pattern (`/*`) defined in the `allPages` security constraint covers everything under the Java EE application root, meaning that the resource files (such as images, style sheets, or JavaScript library) used by the login page are also included.

Example 35–13 Java EE Security Constraint for ADF Authentication Only

```
<security-constraint>
   <web-resource-collection>
      <web-resource-name>allPages</web-resource-name>
      <url-pattern>/*</url-pattern>
   </web-resource-collection>
   <auth-constraint>
      <role-name>valid-users</role-name>
   </auth-constraint>
</security-constraint>
```

If the pages you create use resources, such as images, CSS files, or JavaScript libraries, the default `allPages` security constraint in the `web.xml` file will prevent those resources from loading at runtime. To allow your application to display those resources, you should save the resources in a folder of their own and then edit the `allPages` security constraint so that the resources folder is not contained in the URL pattern.

Note that this resource issue does not apply when you run the ADF Security wizard and choose the **ADF Authentication and Authorization** option (the default). Specifically, in that case, the default generated constraint is on the ADF Authentication servlet and the constraint (`/adfAuthentication`) excludes any resource files.

35.7.3 How to Create a Public Welcome Page

Because web applications are generally secured, there is always a need for a starting point or home page for unauthenticated users. To create this public welcome page, you create an ADF Faces page to act as the entry point for the application, which contains links to other pages within the application. However, only links to public pages should be rendered to unauthenticated users and, conversely, links to secured pages should be rendered only after the user has logged in and has the appropriate privileges to view the target page.

> **Best Practice:** When the user presses **Ctrl-N** or **Ctrl-T** to open a new browser window or tab and no welcome page is defined in the application's `web.xml` file, the browser will display a 403 or 404 error. To prevent this error, you must specify a welcome page definition in the application's `web.xml` file. You can create this definition when you run the Configure ADF Security wizard. For details about running the wizard, see Section 35.3.1, "How to Enable ADF Security."

35.7.3.1 Ensuring That the Welcome Page Is Public

After you have created a regular ADF Faces page, the page will, by default, be public and accessible by unauthenticated users. If, however, you have associated the welcome page with an ADF resource, for example, by dropping databound ADF Faces components into the welcome page using the Data Controls panel, then ADF Security will secure the page by default. You can make any ADF resource publicly accessible using the `jazn-data.xml` file overview editor to grant a view privilege on the resource to the provided `anonymous-role`. For details about the `anonymous-role` see, Section 35.5.2, "What Happens When You Make an ADF Resource Public."

35.7.3.2 Adding Login and Logout Links

You can add login and logout links to your public welcome page so that users can explicitly log in and out while they are in the application. While Java EE container-managed security supports the concept of authentication when accessing a secured resource, there is no standard way to log out and stay within a secured application. However, it is a common practice in web applications to allow the user to stay on the same page if that page is public or to return the user to the welcome page if that page is secured. While adding the login and logout links to each page would let the user end their login session anywhere within the application (and return to the welcome page), having these links on the welcome page enables users to explicitly authenticate on entering the application.

For example, you can create an ADF Faces panel group with three components, including an output text area, an image, and a go link. To render the appropriate login or logout link, you can use an EL expression that evaluates the user's authentication

status. Specifically, you can use `securityContext.authenticated` to access the ADF security context, as shown in Example 35–14. The expression evaluates to `true` or `false` and, in this example, the result determines which login/logout image and link to display.

Example 35–14 ADF Faces Components and EL Expressions to Render Login/Logout Link

```
<af:panelGroupLayout inlineStyle="width:100%; height:15px;" id="ptpgl3">
     <af:spacer width="7" height="10" id="pts2"/>
     <af:outputText value="Welcome #{securityContext.userName}!"
                          inlineStyle="font-weight:bold; width:100px" id="ptot2"
                          rendered="#{securityContext.authenticated}"/>
     <af:image source='#{securityContext.authenticated ? "/images/lock.gif" : "/images/key.gif"}'
               id="pti2" inlineStyle="width:16px; height:16px;"
               shortDesc="switchable icon"/>
     <af:goLink text="#{securityContext.authenticated ? &quot;Logout&quot; : &quot;Login&quot;}"
                destination="#{securityContext.authenticated ?
                       &quot;/adfAuthentication?logout=true&amp;end_url=/faces/welcome.jspx&quot; :
                            &quot;/adfAuthentication?success_url=/faces/welcome.jspx&quot;}"
                inlineStyle="color:White; font-size:14px; font-weight:bold;"/>
      <f:facet name="separator">
         <af:spacer width="5" height="10" id="pts1"/>
      </f:facet>
</af:panelGroupLayout>
```

As an alternative to rendering the link directly within a page, you can create a login link component with the login and logout links that you can add to a page template, as described in Section 35.7.1, "How to Create a Login Link Component and Add it to a Public Web Page."

35.7.3.3 Hiding Links to Secured Pages

Since an anonymous user should not have access to any secured pages, any navigation component on the welcome page that points to a secured page should be hidden from view based on the following two criteria:

- Is the user authenticated with a known user identity?
- Does the specified user identity have permission to view the target?

If either of these criteria has not been met, the `rendered` attribute of any navigation component on a public page that points to a secured resource must have its `rendered` attribute set to `false`, thus hiding it from the anonymous user. To enforce these rules within your welcome page, see Section 35.11.1, "Using Expression Language (EL) with ADF Security."

35.7.4 How to Redirect a User After Authentication

After the user accesses a secured web page and logs in, the ADF authentication servlet will redirect back to the original page that initiated the login request. With ADF Security authentication enabled, the ADF authentication servlet automatically passes the original page as the ADF authentication `success_url` parameter on the URL. Typically, this is the desired behavior; however, you can also specify the desired authentication redirect page when you want to display an explicit login link in your page, as shown in Example 35–15.

Example 35–15 Explicit Login Link with success_url in a Web Page

```
<af:goLink text="Login" destination="/adfAuthentication?success_url=/faces/pagename.jspx"/>
```

Additionally, you can specify the `success_url` parameter as an `<init-param>` within the `web.xml` file to handle any cases where it is not possible to redirect to the original page. Thus, when the user accesses the secured web page and gets redirected to log in, the framework automatically passes the original page as the `success_url` parameter on the URL, which supersedes any `web.xml` setting. Therefore, in practice the only scenario in which an `<init-param>` setting in `web.xml` takes effect is when the user explicitly types the `adfAuthentication` URL into the browser.

In cases where the user is authenticated but not authorized to view a web page, you can redirect the ADF authentication servlet to an error page in your application. Error handling in Fusion web applications is under the control of the ADF Controller exception handler unless you have created an application that does not use a task flow in its design. For example, in an unbounded task flow, where you have defined an unbounded task flow with a top-level welcome page and a browse page (secured through its ADF page definition), you would see an error page from the application, named `authorizationErrorPage.jspx`, specified in the `adfc-config.xml` file, as shown in Example 35–16.

Example 35–16 Error Page Redirect for Applications That Use Task Flows

```
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2">
  <exception-handler>authorizationErrorPage</exception-handler>
  <view id="welcomePage">
    <page>/welcomePage.jspx</page>
  </view>
  <view id="browse">
    <page>/browse.jspx</page>
  </view>
  <view id="authorizationErrorPage">
    <page>/authorizationErrorPage.jspx</page>
  </view>
  <control-flow-rule>
    <from-activity-id>welcomePage</from-activity-id>
    <control-flow-case>
      <from-outcome>goToSecuredPage</from-outcome>
      <to-activity-id>browse</to-activity-id>
    </control-flow-case>
  </control-flow-rule>
</adfc-config>
```

For details about how to specify an error page as a view activity for the ADF Controller exception handler, see Section 22.4, "Handling Exceptions in Task Flows."

In cases where the user is not authenticated and an authorization failure occurs, the framework redirects to the ADF authentication servlet, which in turn triggers a Java EE constraint that prompts for login. In this case, container-managed security relies on the login page and error page that you specify in the `<login-config>` element of the `web.xml` file.

If you create a Fusion web application without utilizing task flows, then you can specify an `<init-param>` setting in `web.xml` for the ADF binding filter, as shown in Example 35–17. In this case, when no task flow is present in the application, page authorization checking is handled by the ADF binding filter, and the `unauthorizedErrorPage` parameter will be passed to the ADF binding request handler.

Note: The `unauthorizedErrorPage` parameter feature is provided for compatibility with previous releases where ADF Controller was not available. In Fusion web applications, when you need to redirect users to an error page, you use the task flow exception handler to specify the error page, as shown in Example 35–16.

Example 35–17 Error Page Redirect for Applications That Don't Use Task Flows

```
<filter>
    <filter-name>adfBindings</filter-name>
    <filter-class>oracle.adf.model.servlet.ADFBindingFilter</filter-class>
    <init-param>
        <param-name>unauthorizedErrorPage</param-name>
        <param-value>faces/authorizationErrorPage.jspx</param-value>
    </init-param>
</filter>
```

35.7.5 What You May Need to Know About ADF Servlet Logout and Browser Caching

When basic type authentication is in effect as specified in the application's `web.xml` file, the browser caches authentication credentials. This is a known issue with basic authentication that prevents the ADF authentication servlet from completing log out and allows users to access resources after logout. In this scenario, in order to complete the logout session, it is necessary to close the browser and restart a new browser session.

To ensure the ADF authentication servlet completes logout and prevents a user from being able to access resources after logout, use form-based authentication instead of basic authentication. You can select form-based authentication when you run the Configure ADF Security wizard, as described in Section 35.3.1, "How to Enable ADF Security."

35.8 Testing Security in JDeveloper

Integrated WebLogic Server enables you to run the application directly within JDeveloper and determine whether or not to migrate security objects, including the application policies, users, and credentials that your application defines. By default, all security objects are migrated to Integrated WebLogic Server each time you run the application.

35.8.1 How to Configure, Deploy, and Run a Secure Application in JDeveloper

JDeveloper is configured by default to deploy the security objects from your application repositories to Integrated WebLogic Server each time you run the application. You can change this behavior by selecting security deployment options in the Application Properties dialog to:

- Decide whether to overwrite the domain-level policies with those from the application `jazn-data.xml` file
- Decide whether to overwrite the system credentials from the application's `cwallet.sso` file
- Decide whether to migrate the identity store portion of the `jazn-data.xml` file to the domain-level identity store

If you make no changes to the deployment settings, each time you run the application, JDeveloper will overwrite the domain-level security policies and system credentials. Additionally, JDeveloper will migrate new user identities you create for test purposes and update existing user passwords in the embedded LDAP server that Integrated WebLogic Server uses for its identity store. However, if you prefer to run the application without updating the existing security objects in Integrated WebLogic Server, you have this option.

Before you begin:

It may be helpful to have an understanding of using Integrated WebLogic Server. For more information, see Section 35.8, "Testing Security in JDeveloper."

To configure security deployment and run the application in JDeveloper:

1. In the main menu, choose **Application** and then **Secure > Configure Security Deployment**.
2. In the Application Properties dialog, in the Deployment page, in the **Security Deployment Options** section, select the security objects that you want to deploy to Integrated WebLogic Server.

   By default, each time you run the application, JDeveloper will overwrite the application policies and system credentials at the domain level with those from the application. If you prefer not to overwrite either of these repositories, deselect **Application Policies** or **Credentials**. When deselected, JDeveloper will merge only new polices or credentials into the domain-level stores.

   By default, each time you run the application, JDeveloper will migrate new user identities you create for test purposes and update existing user passwords in the embedded LDAP server that Integrated WebLogic Server uses for its identity store. You can disable migration of the application identity store by deselecting **Users and Groups**.
3. Click **OK**.
4. In the Application Navigator, right-click the user interface project that contains the secured web pages and choose **Run**.

   When you choose **Run** on the user interface project, JDeveloper will run the application using the default run target you configured for the project. For example, you can configure a task flow activity as the run target to start your application. To configure the default run target, see Section 18.5, "Testing Task Flows."

   The Configure Default Domain dialog displays the first time you run the application and start a new domain in Integrated WebLogic Server. Use the dialog to define an administrator password for the new domain. Passwords you enter can be eight characters or more and must have a numeric character.

35.8.2 What Happens When You Configure Security Deployment Options

When you run the application using Integrated WebLogic Server, JDeveloper migrates the security policies and credentials to the domain level based on security deployment configuration settings specified in the Application Properties dialog. During the deployment process, JDeveloper updates the `weblogic-application.xml` file that it adds to the deployment archive file with the Application Properties settings, as shown in Example 35–18. Note that these settings are not added to the `weblogic-application.xml` file in the application source directory and thus are not visible.

Example 35–18 Default Security Deployment Settings in the Archive weblogic-application.xml File

```
<application-param>
    <param-name>jps.credstore.migration</param-name>
    <param-value>OVERWRITE</param-value>
</application-param>
<application-param>
    <param-name>jps.policystore.migration</param-name>
    <param-value>OVERWRITE</param-value>
</application-param>
```

The `OVERWRITE` value allows you to modify the security policies and credentials in your application and redeploy either to Oracle WebLogic Server running in development mode or to Integrated WebLogic Server (set up to run in development mode by default).

> **Note:** When you eventually deploy to a production environment, the migration settings in the `weblogic-application.xml` file are ignored; it would be considered a security vulnerability to allow existing policies and credentials to be overwritten. For information about deploying to a production environment, see Section 35.9, "Preparing the Secure Application for Deployment."

JDeveloper also updates the `weblogic-application.xml` file with OPSS lifecycle listeners, as shown in Example 35–19. To initiate the migration process before the application runs, the lifecycle listeners observe the migration settings for policies and credentials and overwrite the security objects at the domain level.

Example 35–19 Security Migration Listeners in the Archive weblogic-application.xml File

```
<listener>
    <listener-class>
        oracle.security.jps.wls.listeners.JpsApplicationLifecycleListener
    </listener-class>
</listener>
<listener>
    <listener-class>
        oracle.security.jps.wls.listeners.JpsAppVersionLifecycleListener
    </listener-class>
</listener>
```

During the migration process, JDeveloper maps the Oracle Platform Security Services (OPSS) application role member classes to the Integrated WebLogic Server member classes and migrates the users to WebLogic Server identity store `users` and migrates the roles to Integrated WebLogic Server identity store groups. In Oracle WebLogic Server, `users` is an implicit group equivalent to OPSS `authenticated-role`.

Example 35–20 Application Role Fragment in the system-jazn-data.xml File

```
<app-roles>
   <app-role>
      <name>fod-users</name>
      <guid>FFFF394F696E786F4134485764511002</guid>
      <display-name/>
      <description/>
      <class>oracle.security.jps.service.policystore.ApplicationRole</class>
      <members>
```

```
            <member>
               <name>fod-users</name>
               <class>weblogic.security.principal.WLSGroupImpl</class>
            </member>
         </members>
      </app-role>
</app-roles>
```

Identity store migration is not controlled by the application lifecycle listener settings in the `weblogic-application.xml` file. Instead, an Oracle WebLogic Mbean handles migrating the identities when running in Integrated WebLogic Server or when deploying from JDeveloper. If the user already exists, the Mbean will not migrate the entire user definition. Only the user password will be updated.

35.8.3 How to Use the Built-In test-all Application Role

When you run the Configure ADF Security wizard, you can enable the option to add the `test-all` application role to the policy store in the `jazn-data.xml` file. When you enable this option, you also specify the scope of grants to the application role for your application:

- Select **Grant to Existing Objects Only** when you want JDeveloper to grant view rights to the `test-all` application role and you want this policy to apply to all the ADF task flows and web pages that appear in your user interface project at the time you run the wizard.
- Select **Grant to All Objects** when you want JDeveloper to grant view rights to the `test-all` application role and you want this policy to apply to all existing and future ADF task flows and web pages that developers will create in the user interface project. Note that the wizard displays the option **Grant to New Objects** after you run the wizard the first time with the **Grant to All Objects** option selected.

After you run the wizard, the `test-all` role appears in the `jazn-data.xml` file and is visible in the `jazn-data.xml` file overview editor. You will not need to populate the `test-all` role with test users since the wizard assigns the built-in application role `anonymous-role` to the `test-all` role. In this case, all users will automatically have the `anonymous-role` principal and will be permitted to access the application.

Note: Before you deploy the application, you must remove all occurrences of the `test-all` role from the policy store, as described in Section 35.9.1, "How to Remove the test-all Role from the Application Policy Store." This will prevent unauthorized users from accessing the web pages of your application.

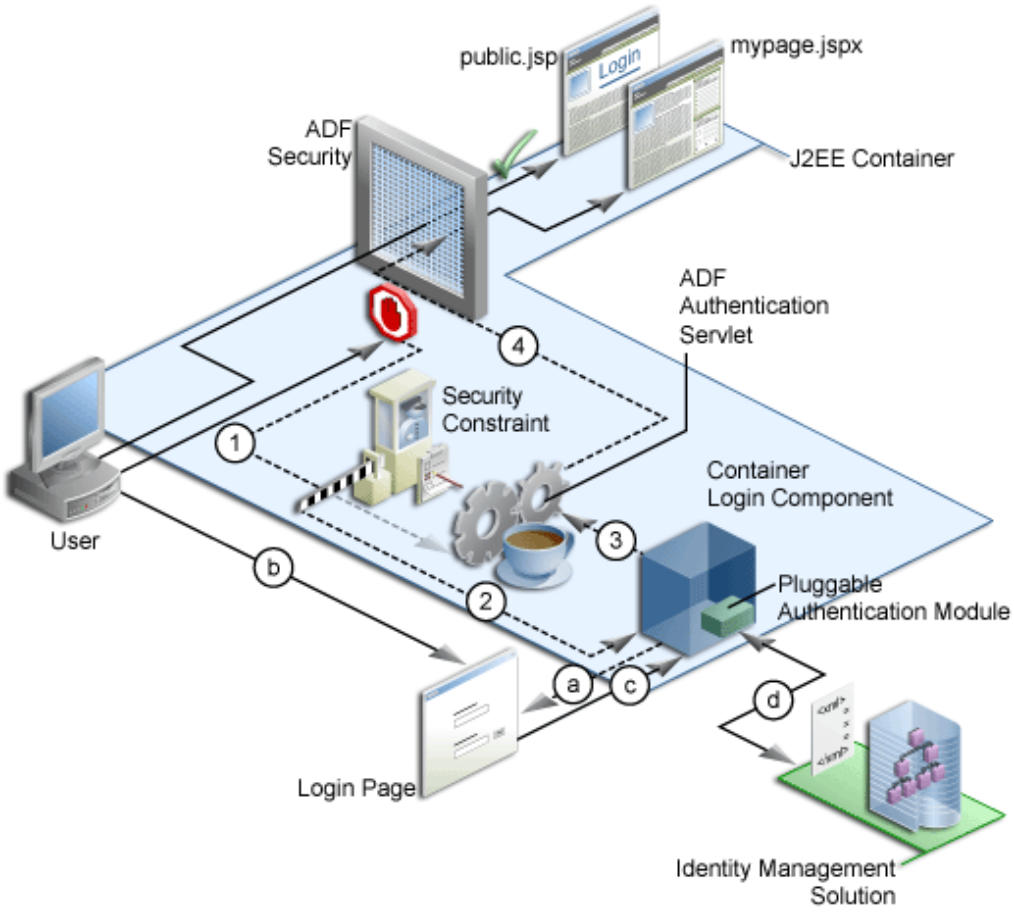
You can rerun the wizard and disable automatic grants at any time. Once disabled, new ADF task flows and web pages that you create will not utilize the `test-all` role and will therefore require that you define explicit grants, as described in Section 35.5, "Defining ADF Security Policies."

35.8.4 What Happens at Runtime: How ADF Security Handles Authentication

When you test the application in JDeveloper using Integrated WebLogic Server, the identity store is migrated to the embedded LDAP server, with information stored in Oracle Internet Directory.

Figure 35–26 illustrates the authentication process when users attempt to access an ADF bounded task flow or any web page containing ADF bindings (such as `mypage.jspx`) without first logging in. Authentication is initiated implicitly because the user does not begin login by clicking a login link on a public page. In the case of the secured page, no grants have been made to the anonymous user.

Figure 35–26 ADF Security Implicit Authentication



In Figure 35–26, the implicit authentication process assumes that the resource does not have a grant to `anonymous-role`, that the user is not already authenticated, and that the authentication method is Form-based authentication. In this case, the process is as follows:

1. When the bounded task flow or web page (with ADF bindings) is requested, the ADF bindings servlet filter redirects the request to the ADF authentication servlet (in the figure, Step 1), storing the logical operation that triggered the login.
2. The ADF authentication servlet has a Java EE security constraint set on it, which results in the Java EE container invoking the configured login mechanism (in the figure, Step 2). Based on the container's login configuration, the user is prompted to authenticate:
   a. The appropriate login form is displayed for form-based authentication (in the figure, Step 2a).
   b. The user enters his credentials in the displayed login form (in the figure, Step 2b).

c. The user posts the form back to the container's `j_security_check()` method (in the figure, Step 2c).

d. The Java EE container authenticates the user, using the configured pluggable authentication module (in the figure, Step 2d).

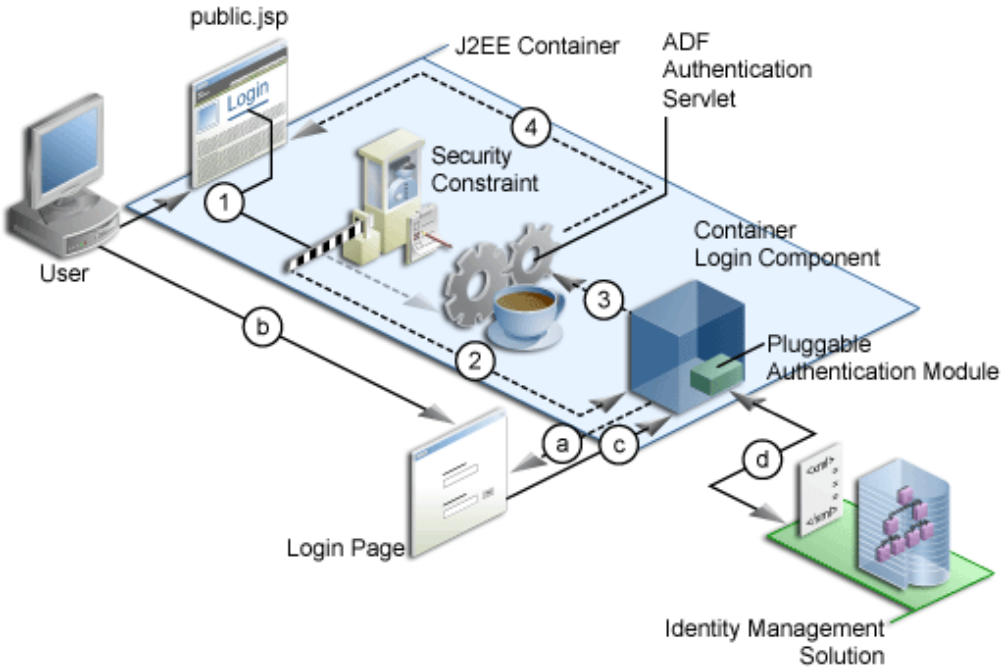
3. Upon successful authentication, the container redirects the user back to the servlet that initiated the authentication challenge, in this case, the ADF authentication servlet (in the figure, Step 3).

4. On returning to the ADF authentication servlet, the servlet subsequently redirects to the originally requested resource (in the figure, Step 4).

   Whether or not the resource is displayed will depend on the user's access rights and on whether authorization for ADF Security is enforced, as explained in Section 35.8.5, "What Happens at Runtime: How ADF Security Handles Authorization."

Figure 35–27 illustrates the explicit authentication process when the user becomes authenticated starting with the login link on a public page.

Figure 35–27 ADF Security Explicit Authentication



In an explicit authentication scenario, an unauthenticated user (with only the `anonymous` user principal and `anonymous-role` principal) clicks the **Login** link on a public page (in the figure, Step 1). The login link is a direct request to the ADF authentication servlet, which is secured through a Java EE security constraint in the `web.xml` file.

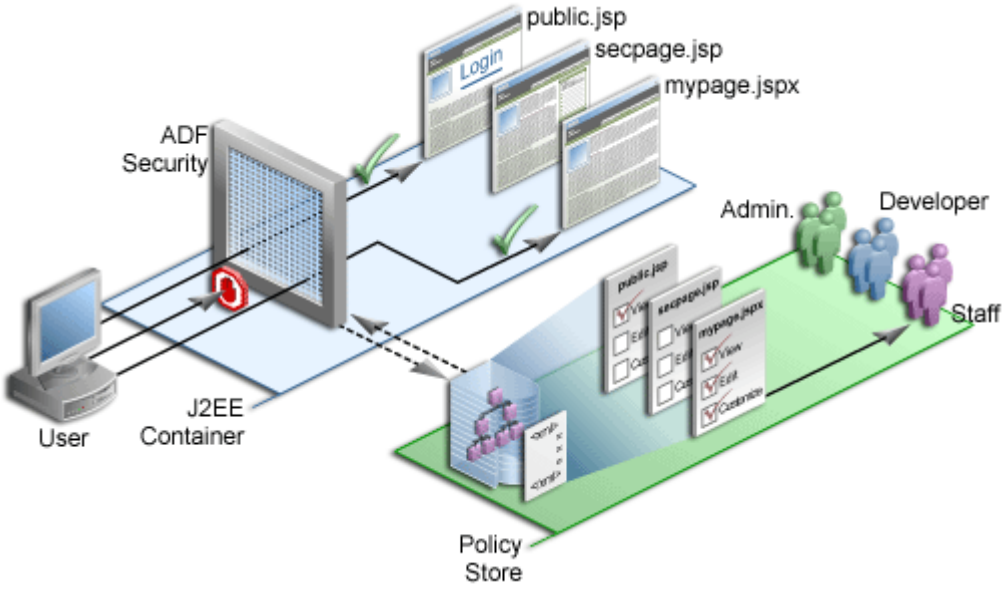
In this scenario, the current page is passed as a parameter to the ADF authentication servlet. As with the implicit case, the security constraint redirects the user to the login page (in the figure, Step 2). After the container authenticates the user, as described in Step a through Step d in the implicit authentication case, the request is returned to the ADF authentication servlet (in the figure, Step 3), which subsequently returns the user to the public page, but now with new user and role principals in place.

35.8.5 What Happens at Runtime: How ADF Security Handles Authorization

When ADF authorization is enabled, the ADF bounded task flows and web pages outside of a task flow that have an ADF page definition will be secure by default. When a user attempts to access these web pages, ADF Security checks to determine whether the user has been granted access in the policy store. If the user is not yet authenticated, and the page is not granted to the `anonymous-role`, then the application displays the login page or form. If the user has been authenticated, but does not have permission, a security error is displayed. If you do not configure the policy store with appropriate grants, the pages will remain protected and therefore stay unavailable to the authenticated user.

Figure 35–28 illustrates the authorization process.

Figure 35–28 ADF Security Authorization



The user is a member of the application role *staff* defined in the policy store. Because the user has not yet logged in, the security context does not have a subject (a container object that represents the user). Instead, Oracle Platform Security Services provides ADF Security with a subject with the `anonymous` user principal (a unique definition of the user) and the `anonymous-role` principal.

With the `anonymous-role` principal, typically the user would be able to access only pages not defined by ADF resources, such as the `public.jsp` page, whereas all pages that are defined either by an ADF task flow or outside of a task flow using an ADF page definition file are secure by default and unavailable to the user. An exception to this security policy would be if you were to grant `anonymous-role` access to ADF resources in the policy store. In this case, the user would not be allowed immediate access to the page defined by an ADF resource.

When the user tries to access a web page defined by an ADF resource, such as `mypage.jspx` (which is specified by an ADF page definition, for example), the ADF Security enforcement logic intercepts the request and because all ADF resources are secured by default, the user is automatically challenged to authenticate (assuming that the `anonymous-role` is not granted access to the ADF resource).

After successful authentication, the user will have a specific subject. The security enforcement logic now checks the policy store to determine which role is allowed to

view `mypage.jspx` and whether the user is a member of that role. In this example for `mypage.jspx`, the view privilege has been granted to the staff role and because the user is a member of this role, they are allowed to navigate to `mypage.jspx`.

Similarly, when the user tries to access `secpage.jsp`, another page defined by ADF resources, for which the user does not have the necessary view privilege, access is denied.

Users and roles are those already defined in the identity store of the resource provider. Application roles are defined in the policy store of the `jazn-data.xml` file.

35.9 Preparing the Secure Application for Deployment

After testing in JDeveloper using Integrated WebLogic Server, you will eventually want to deploy the application to a standalone server. Initially, the server you target will be your staging environment where you can continue development testing using that server's identity store before deploying to the production environment. Thus, you will typically not migrate the test users you created to run with Integrated WebLogic Server. The steps you perform to migrate security policies and system credentials (from the `cwallet.sso` file) to standalone Oracle WebLogic Server will depend on the configured mode of the target server and whether you deploy using JDeveloper or a tool outside of JDeveloper.

> **Note:** For details about deploying from JDeveloper to a development environment, see Section 41, "Deploying Fusion Web Applications."

When the target server is configured for *development mode*, you can deploy directly from JDeveloper. In this case, JDeveloper automatically handles the migration of the policy store, system credentials, and identity store (users and groups) as part of the deployment process. Application security deployment properties are configured by default to allow the deployment process to overwrite the domain-level policy store and the system credentials. Additionally, the identity store deployment property is configured by default to migrate the identity store consisting of your test users. You can change this default deployment behavior in the Application Properties dialog, as described in Section 35.8.1, "How to Configure, Deploy, and Run a Secure Application in JDeveloper."

> **Note:** Note that migration of system credentials to Oracle WebLogic Server running in development mode will be performed only if the target server is configured to permit credential overwrite. For details about configuring Oracle WebLogic Server to support overwriting of system credentials, see the *Oracle Containers for J2EE Security Guide*.

When the target server is configured for *production mode*, you typically handle the migration task outside of JDeveloper using tools like Oracle Enterprise Manager. For details about using tools outside of JDeveloper to migrate the policy store to the domain-level in a production environment, see the *Oracle Containers for J2EE Security Guide*. Note that Oracle WebLogic Server running in production mode does not support the overwriting of system credentials under any circumstances.

Before you deploy the application, you will want to remove the `test-all` application role if you enabled the automatic grants feature in the Configure ADF Security wizard. Because the `test-all` role makes all ADF resources public, its presence increases the

risk that your application may leave some resources unprotected. You must therefore remove the role before you migrate application-level policy store.

Additionally, when you prepare to deploy the application to Oracle WebLogic Server, you will want to remove the test identities that you created in the `jazn-data.xml` file. This will ensure that users you created to test security policies are not migrated to the domain-level identity store.

> **Best Practice:** If you deploy your application to the standalone environment, you must not migrate users and enterprise roles in your local identity store that are already configured for Oracle WebLogic Server. For example, if you were to deploy the identity store with the user `weblogic` and enterprise role `Administrators`, you would overwrite the default administration configuration on the target server. To ensure you avoid all possible conflicts, you can disable migration of the identity store as described in Section 35.9.2, "How to Remove Test Users from the Application Identity Store."

35.9.1 How to Remove the test-all Role from the Application Policy Store

The `jazn-data.xml` file overview editor provides the facility to display all resources with view grants made to ADF Security's built-in `test-all` role. You can use this feature in the overview editor to delete the `test-all` role grant and replace it with a grant to the roles that your application defines.

Alternatively, you could delete the `test-all` role using the overview editor for the `jazn-data.xml` file, by selecting the `test-all` role in the Application Roles page of the editor and clicking the **Delete Application Role** button. However, when you remove the `test-all` role this way, you will still need to create a grant to replace the ones that you delete. Because the overview editor lets you combine both of these tasks, the following procedure describes its usage.

Before you begin:

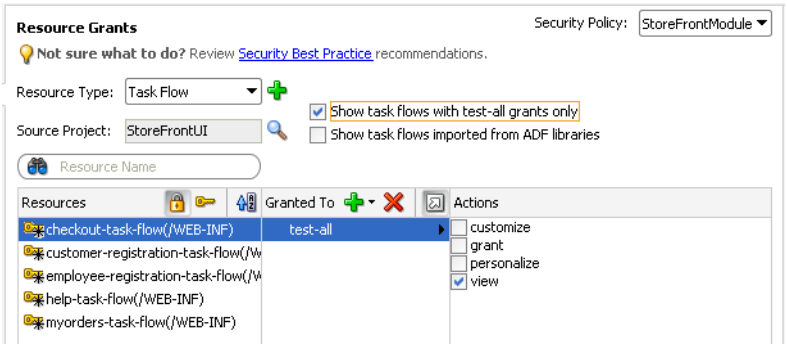
It may be helpful to have an understanding of the Oracle WebLogic Server. For more information, see Section 35.9, "Preparing the Secure Application for Deployment."

To remove the test-all application role and substitute custom application roles:

1. In the main menu, choose **Application** and then **Secure > Resource Grants**.
2. In the Resource Grants page of the `jazn-data.xml` file overview editor, select **Task Flow** from the **Resource Types** dropdown list and then select the **Show task flows with test-all grants only** checkbox to view the list of task flows with grants to this built-in role.

   If no grant exists for the `test-all` role, then the **Resources** list in the overview editor will appear empty. The `test-all` role is defined only when enabled in the Configure ADF Security wizard. If it is enabled, you will see those task flows with `test-all` grants listed, as shown in Figure 35–29.

Figure 35–29 Showing Task Flows with test-all Grants in the Overview Editor



3. In the **Resources** column, select the first task flow in the list.
4. In the **Granted to** column, select **test-all** and click the **Remove Grantee** icon.
5. In the **Granted to** column, click the **Add Grantee** icon and choose **Add Application Role** and then use the Select Application Roles dialog to add the desired role.
6. Repeat these steps to remove the `test-all` role and substitute your own application role for all remaining task flows.
7. In the Resource Grants page of the overview editor, select **Web Page** from the **Resource Types** dropdown list and repeat these steps to remove the `test-all` role for all web pages and their ADF page definitions.
8. With the **Show task flows/web pages with test-all grants only** checkbox selected, verify that the overview editor displays no resources with `test-all` grants.

35.9.2 How to Remove Test Users from the Application Identity Store

The standalone Oracle WebLogic Server that you will deploy to will have its own identity stored already configured. To ensure that you do not migrate test users and enterprise role groups you created in JDeveloper to the domain level, you should remove the test user realm from the `jazn-data.xml` file.

Alternatively, if you are deploying from JDeveloper, you can disable the migration of users and groups by deselecting the **Users and Groups** option in the Application Properties dialog, as described in Section 35.8.1, "How to Configure, Deploy, and Run a Secure Application in JDeveloper."

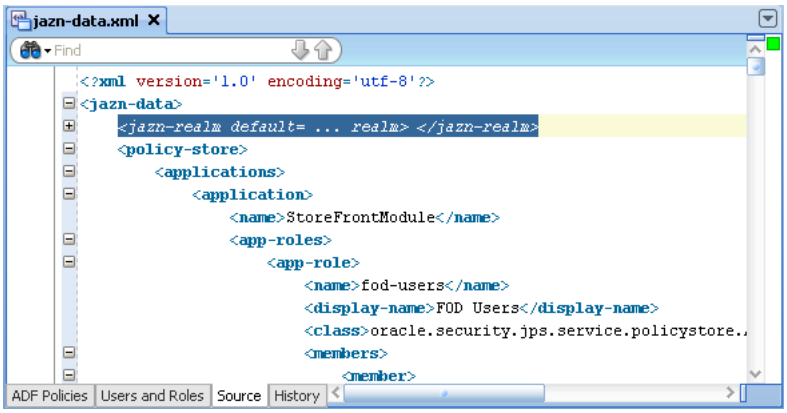
Before you begin:

It may be helpful to have an understanding of the Oracle WebLogic Server. For more information, see Section 35.9, "Preparing the Secure Application for Deployment."

To remove test users and enterprise role groups from the identity store:

1. In the main menu, choose **Application** menu and then **Secure > Users**.
2. In the editor window for the `jazn-data.xml` file, click the **Source** tab.
3. In the source for the `jazn-data.xml` file, click the **-** icon to the left of the **<jazn-realm>** element so the entire element appears collapsed as shown in Figure 35–30.

Figure 35–30 Selecting the <jazn-realm> Element in the XML Editor



4. With the element selected, press **Delete** and save the file.

35.9.3 How to Secure Resource Files Using a URL Constraint

Resource files, including images, style sheets, and JavaScript libraries are files that the Fusion web application loads to support the individual pages of the application. ADF Security does not secure these files. Although securing such files is not a requirement for securing the web pages of the application, in some cases it may be desirable to protect resource files to fully harden your application. Please note that JavaScript files are downloaded to the browser and, as such, can be read at that time.

To serve up resource files, the ADF Faces framework relies on `org.apache.myfaces.trinidad.webapp.ResourceServlet`, which delegates to a resource loader. The Fusion web application's `web.xml` file contains a servlet mapping that maps the resource servlet to URL patterns. By default, JDeveloper uses the patterns `/adf/*` for MyFaces Trinidad Core, and `/afr/*` for ADF Faces.

Because ADF Security does not protect the resource servlet, you can protect the Java EE access paths for the resource servlet with a security constraint. To implement your own security for the resource files in the `web.xml` file, define a security constraint for the `/adflib/*`, `/adf/*` and `/afr/*` paths and assign a specific security role to them. This security constraint will require all users to be authenticated before they can access the first page of the Fusion web application.

Example 35–21 shows the security role `resource_role` and the constraint with the URL patterns mapped to this role. After you define the security role, the administrator for the target server must map this role to an Oracle WebLogic Server enterprise role (user group) or create an enterprise role with the same name (in which case no mapping is required).

Example 35–21 URL Constraint in web.xml File for Resource Files

```
<security-role>
    <description>Java EE role to map to an enterprise role. All users who will
      be allowed to run Fusion web app must be members of that role.
    </description>
    <role-name>resource_role</role-name>
</security-role>

<security-constraint>
    <web-resource-collection>
      <web-resource-name>resources</web-resource-name>
      <url-pattern>/adflib/*</url-pattern>
```

```
      <url-pattern>/adf/*</url-pattern>
      <url-pattern>/afr/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
      <role-name>resource_role</role-name>
    </auth-constraint>
</security-constraint>
```

35.10 Disabling ADF Security

JDeveloper allows you to disable ADF Security when you want to temporarily run the application without enforcing authorization checks against the application policy store. This will allow you to run the application and access all resources without the protection provided by existing security policies.

35.10.1 How to Disable ADF Security

To disable ADF Security at the level of your application, run the wizard and choose one of these options:

- **ADF Authentication** disables ADF authorization but leaves the ADF authentication servlet enabled. This option will require the user to log in the first time a page in the application is accessed by mapping the Java EE application root "/" to the a Java EE security constraint that will trigger user authentication. ADF resources will not be security-aware because authorization checking is not enforced. Thus, once the user is logged in, all web pages containing ADF resources will be available to the user.
- **Remove ADF Security Configuration** disables the ADF authentication servlet and disables authorization checking on ADF resources. In this case, you can run the application with no user authentication and no security for ADF resources in place.

You may select either option with the intention of reenabling ADF Security at any time. The wizard specifically does not alter the application policy store that contains the security policies that application developers defined for ADF resources. This means that you can return to the wizard at any time, select the **ADF Authentication and Authorization** option, and reenable ADF Security against your application's existing policy store and identity store.

Before you begin:

It may be helpful to have an understanding of disabling ADF Security. For more information, see Section 35.10, "Disabling ADF Security."

To disable ADF authorization checking:

1. In the main menu, choose **Application** and then **Secure > Configure ADF Security**.
2. In the ADF Security page, select either the **ADF Authentication** option or the **Disable ADF Security Configuration** option. Click **Next**.

   After you run the wizard with either of these options, the ADF resources of your user interface projects will no longer be security-aware.
3. Click **Finish**.

35.10.2 What Happens When You Disable ADF Security

If you run the wizard with the **Remove ADF Security Configuration** option selected, it removes the ADF-specific metadata in the `web.xml` file and `adf-config.xml` file, as described in Table 35–2. Similarly, running the wizard with the **ADF Authentication** option selected sets the `authorizationEnforce` parameter in the `<JaasSecurityContext>` element of the `adf-config.xml` file to `false`, as shown in Example 35–22.

Example 35–22 AuthorizationEnforce Flag Disabled in the adf-config.xml FIle

```
<JaasSecurityContext
  initialContextFactoryClass="oracle.adf.share.security.JAASInitialContextFactory"
  jaasProviderClass="oracle.adf.share.security.providers.jps.JpsSecurityContext"
                              authorizationEnforce="false"
                              authenticationRequire="true"/>
```

35.11 Advanced Topics and Best Practices

After you have completed the process of enabling ADF Security, you may want to customize your application to work with ADF Security in the user interface. For example, you can use Expression Language (EL) to render UI components in the web page based on evaluation of custom permissions that you define just for a group of UI components. Additionally, you can define methods within a managed bean to expose information, such as the user name and role membership, in your application.

35.11.1 Using Expression Language (EL) with ADF Security

You can use Expression Language (EL) to evaluate the policy directly in the UI, while the use of Java enables you to evaluate the policy from within a managed bean. ADF Security implements several convenience methods for use in EL expressions to access ADF resources in the security context. For example, you can use the EL expression convenience methods to determine whether the user is allowed to access a particular task flow. Good security practice dictates that your application should hide resources and capabilities for which the user does not have access. And for this reason, if the user is not allowed access to a particular task flow, you would evaluate the user's permission grant to determine whether or not to render the navigation components that initiate the task flow.

> **Note:** The ability to evaluate a policy is limited to the current request. For this reason, it is important to understand where the policy evaluation occurs, because evaluating the policy at anything other than the request scope can lead to unexpected results.

35.11.1.1 How to Evaluate Policies Using EL

The use of EL within a UI component allows for the component's attribute values to be defined dynamically, resulting in modification of the UI component at runtime. In the case of securing resources, the UI component attribute of interest is the `rendered` attribute, which allows you to show and hide components based on available permissions. By default, the `rendered` attribute is set to `true`. By dynamically changing this value based on the permission, you can set the UI component to be shown or hidden. For example, if the user has the appropriate permission, the `rendered` attribute should be set to `true` so that the UI component is shown. If they do not have permission, the attribute should be set to `false` and the UI component hidden from view.

To evaluate a policy using EL, you must use the ADF Security methods in the `securityContext` EL namespace. These methods let you access information in the ADF security context for a particular user or ADF resource.

Table 35–10 shows the EL expression that is required to determine whether a user has the associated permission. If the user has the appropriate permission, the EL expression evaluates to `true`; otherwise, it returns `false`.

Table 35–10 EL Expression to Determine View Permissions on ADF Resources

Expression	Expression action

`#{securityContext.taskflowViewable['MyTaskFlow']}`
For example:
`#{securityContext.taskflowViewable ['/WEB-INF/audit-expense-report.xml#audit-expense-report']}`
	Where *`MyTaskFlow`* is the WEB-INF node-qualified name of the task flow being accessed. Returns `true` if the user has access rights. Returns `false` if the user does not have sufficient access rights.

`#{securityContext.regionViewable['MyPagePageDef']}`	Where *`MyPagePageDef`* is the qualified name of the page definition file associated with the web page being accessed. Returns `true` if the user has access rights. Returns `false` if the user does not have sufficient access rights.

Note: In the case of page permission, the value of the page definition can be specified dynamically by using late-binding EL within a managed bean, as described in Section 35.3.7, "What You May Need to Know About the valid-users Role."

Table 35–11 shows the EL expression that lets you get general information from the ADF security context not related to a particular ADF resource. For example, you can access the current user name when you want to display the user's name in the user interface. You can also check whether the current user is a member of certain roles or granted certain privileges. Your application may use this result to dynamically hide or show menus.

Table 35–11 EL Expression to Determine User Information in the ADF Security Context

Expression	Expression Action
`#{securityContext.userName}`	Returns the user name of the authenticated user.
`#{data.adfContext.enterpriseName}`	Returns the enterprise name of the authenticated user. The enterprise name is an alias that the user knows for themselves and can use to login.
`#{data.adfContext.enterpriseId}`	Returns the enterprise ID of the authenticated user.
`#{securityContext.authenticated}`	Returns `true` if the user is logged in. Returns `false` if the user is not logged in. This is useful for rendering a dynamic link for login/logout, or for rendering a "Welcome, *username*" message when the user has been authenticated. For an example that uses this expression, see Section 35.7.3.2, "Adding Login and Logout Links."

Table 35–11 (Cont.) EL Expression to Determine User Information in the ADF Security Context

Expression	Expression Action
`#{securityContext.userInRole['roleList']}`	Where *`roleList`* is a comma-separated list of role names. Returns `true` if the user is in at least one of the roles. Returns `false` if the user is in none of the roles, or if the user is not currently authenticated.
`#{securityContext.userInAllRoles['roleList']}`	Where *`roleList`* is a comma-separated list of role names. Returns `true` if the user is in all of the roles. Returns `false` if the user is not in all of the roles, or if the user is not currently authenticated.
`#{securityContext.userGrantedPermission['permission']}`	Where *`permission`* is a string containing a semicolon-separated concatenation of `permissionClass=<class>;target=<artifact_name>;action=<action>`. Returns `true` if the user has access rights. Returns `false` if the user does not have sufficient access rights. Note that the convenience methods `taskflowViewable` and `regionViewable` shown in Table 35–10 provide the same functionality.

`#{securityContext.userGrantedResource['resource']}`	Where *`resource`* is a string containing a semicolon-separated concatenation of `resourceName=<name>;resourceType=<type>;action=<action>`. Returns `true` if the user has access rights. Returns `false` if the user does not have sufficient access rights.
	
	You can use this expression to test the permission grant in the `rendered` attribute of a resource that is not contained in a task flow (like an ADF Faces panel). This provides an alternative to creating a custom permission class that must be packaged with the application.
	
	For example, when you want to show or hide a panel in a page based on the permission granted to that resource, the expression might look like:
	
	`#{securityContext.userGrantedResource ['resourceName=myPanel1; resourceType=myLayoutPanel; action=myAction']}`
	
	In the policy store, a grant to the resource has a `<permission>` definition like:
	
	`<permission> <class>oracle.security.jps. ResourcePermission</class> <name>resourceType=myLayoutPanel, resourceName=myPanel1</name> <actions>myAction</actions> </permission>`

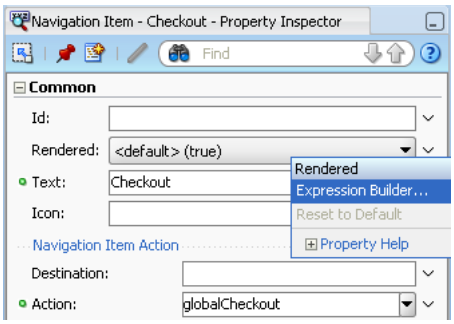
Before you begin:

It may be helpful to have an understanding of using EL. For more information, see Section 35.11.1, "Using Expression Language (EL) with ADF Security."

To associate the rendering of a navigation component with a user's granted permissions on a target task flow or page definition:

1. In the Application Navigator, double-click the page.
2. Select the component that is used to navigate to the secured page.
3. In the Property Inspector, click the **Property Menu** dropdown menu displayed to the right of the **Rendered** field and choose **Expression Builder**, as shown in Figure 35–31.

Figure 35–31 Binding the Rendered Property to Data



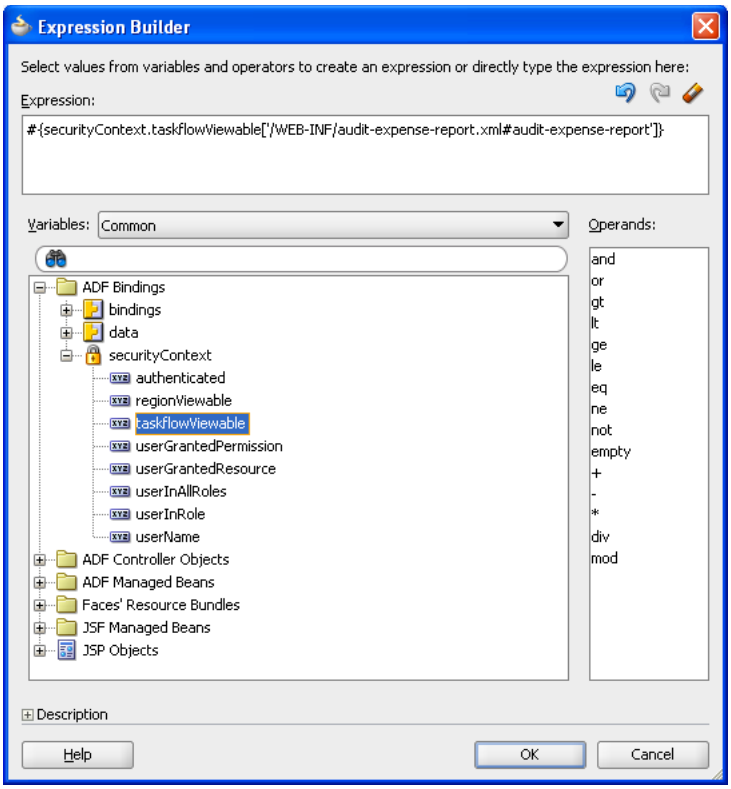
4. In the Expression Builder, expand the **ADF Bindings - securityContext** node and select the appropriate EL value, then in the **Expression** field, enter the qualified name of the ADF resource that the user will attempt to access.

   For example, as shown in Figure 35–32, to limit access to a task flow that your application displays, you would create an expression like:

   ```
   #{securityContext.taskflowViewable
        ['/WEB-INF/audit-expense-report.xml#audit-expense-report']}
   ```

   In this example, the expression determines the user's access rights to view the target task flow `audit-expense-report`. If the user has the access rights, then the expression evaluates to `true` and the `rendered` attribute receives the value `true`.

Figure 35–32 Defining EL in the Expression Builder Dialog



Tip: In the Expression Builder dialog, expand **Description** for additional information about any security EL method you select.

5. Click **OK**.

When you run the application, the component will be rendered or hidden based on the user's ability to view the target page.

35.11.1.2 What Happens When You Use the Expression Builder Dialog

When you use the Expression Builder to define an expression for the `rendered` attribute, JDeveloper updates the component definition in the open `.jspx` file. The component's `rendered` attribute appears with an expression that should evaluate to either `true` or `false`, as shown in Example 35–23. In this example, the component is a navigation link with the link text `Checkout` defined by another expression. The page that contains the navigation link renders the component only when the user has sufficient rights to access the checkout task flow.

Example 35–23 EL Expression in Source for .jspx File

```
<af:commandNavigationItem
   text="#{res['global.nav.checkout']}"
   action="globalCheckout"
   id="cni3"
   rendered="#{securityContext.taskflowViewable
                      ['/WEB-INF/checkout-task-flow.xml#checkout-task-flow']}"
/>
```

35.11.1.3 What You May Need to Know About Delayed Evaluation of EL

The ability to evaluate a security permission is scoped to the request. If you want to evaluate permissions to access a target page from a managed bean that is scoped to a higher level than request (for example, a global menu that is backed by a managed bean), you must implement delayed EL evaluation (late-binding). By passing in the target page as a managed property of the bean, you ensure that the EL expression is evaluated only after the required binding information is available to the managed bean. Because EL is evaluated immediately when the page is executed, placing the EL expression directly in the properties of a UI component, backed by a managed bean, would result in an out-of-scope error.

Example 35–24 shows a property (`authorized`) of a managed bean that returns `true` or `false` based on a user's ability to view a named target page. In this case, the `_targetPageDef` variable is a managed property containing the name of the target page. Within the UI, the EL expression would reference the `authorized` property.

Example 35–24 Delayed EL Evaluation in a Managed Bean

```
public boolean isAuthorized()
{
 if (_targetPageDef != null) {
  FacesContext fctx = FacesContext.getCurrentInstance();
  ADFContext adfCtx = ADFContext.getCurrent();
  SecurityContext secCtx = adfCtx.getSecurityContext();
  boolean hasPermission = secCtx.hasPermission(new RegionPermission
     (_targetPageDef, RegionPermission.VIEW_ACTION));
     if (hasPermission) {
        return hasPermission;
     }
     else {
        fctx.addMessage(null, new FacesMessage (
        FacesMessage.SEVERITY_WARN, "Access Permission not defined! " , null));
        return false;
     }
}
```

35.11.2 How to Evaluate Policies Using Custom JAAS Permissions and EL

You can use the value `userGrantedPermission` in the ADF Security EL namespace described in Table 35–10 to determine whether to render UI elements in your page. The expression you create can evaluate custom permission grants for the authenticated user. A custom permission is a JAAS Permission class that you create using the Create JAAS Permission dialog. The dialog helps you create a class that extends the `oracle.adf.share.security.authorization.ADFPermission` class to ensure that the permission can be used by ADF Security.

Custom permissions in the Fusion web application give you additional flexibility to define security policies. For example, you might name a custom permission to correspond to the UI element you want to protect. Once you create the permission, you use the `jazn-data.xml` file overview editor to create a security policy for the ADF resource by granting permission to the application roles that your `jazn-data.xml` policy store defines.

Best Practice: Custom ADF permission classes let you extend ADF Security to define custom actions for use in grants. This gives you the flexibility to define security policies to manage the user's ability to view UI components without having to overload the built-in actions defined by the ADF resources' permission classes. Be aware that you do not need to create custom permissions to manage access to web pages. This level of access is provided by the default ADF Security view permission that you work with in the `jazn-data.xml` file overview editor.

To evaluate policies using custom JAAS permissions:

1. Create the custom JAAS Permission class.
2. Create the ADF Security policy using the custom permission.
3. Associate the rendering of a UI component with a user's granted custom permission.

35.11.2.1 Creating the Custom JAAS Permission Class

You use the Create JAAS Permission dialog to create the custom JAAS Permission class. JDeveloper adds the custom JAAS Permission class to the package you specify. For example, the `oracle.fodemo.storefront.store.view` package of the Fusion Order Demo application defines the custom permission class `AccountPermission.java` shown in Example 35–25.

Example 35–25 Custom JAAS Permission Class

```
package oracle.fodemo.storefront.store.view;

import oracle.adf.share.security.authorization.ADFPermission;
import oracle.adf.share.security.authorization.PermissionActionDescriptor;
import oracle.adf.share.security.authorization.PermissionTargetDescriptor;

public class AccountPermission extends ADFPermission {
  private static final PermissionActionDescriptor[] actions =
   {new PermissionActionDescriptor("view", "view")};
  private static final PermissionTargetDescriptor[] targets =
   {new PermissionTargetDescriptor("attributeValue", "Attribute")};

  public AccountPermission(String name, String actions) {
    super(name, actions);
  }

  public static PermissionActionDescriptor[] getPermissionActionDescriptors() {
    return actions;
  }

  public static PermissionTargetDescriptor[] getPermissionTargetDescriptors() {
    return targets;
  }
}
```

Before you begin:

It may be helpful to have an understanding of the ADF Permission class. For more information, see Section 35.11.2, "How to Evaluate Policies Using Custom JAAS Permissions and EL."

To create the custom JAAS Permission class:

1. In the Application Navigator, right-click the project where you want to create the custom JAAS Permission class and choose **New**.
2. In the New Gallery, select **All Items** and then **JAAS Permission**, and click **OK**.
3. In the Create JAAS Permission dialog, enter the name of the permission and the fully qualified package name.

   The permission name you choose can be a generic name.
4. In the **Actions** list, enter the name that you want to use for the action grant.

   The action name can be a specific name that helps you to identify the permission's purpose. You can add more than one action to the list when you want the permission to apply to the same component, but for different purposes. For example, you might allow a manager and an employee to both view a page menu, but you might want the manager to also be able to choose specific menu items.
5. In the **Targets** list, leave the selection **Attribute** unchanged.

   You will specify the actual target name when you create the policy in the policy store using the custom JAAS Permission's action.
6. Click **OK**.

35.11.2.2 Creating the ADF Security Policy Using a Custom Permission

You use the overview editor for the `jazn-data.xml` file to create the ADF Security policy based on a custom permission. The finished source should look similar to the custom permission grant defined in the policy store for the Fusion Order Demo application, as shown in Example 35–26.

Example 35–26 Custom Permission Grant Definition

```
<grant>
  <grantee>
    <principals>
      <principal>
         <class>oracle.security.jps.service.policystore.ApplicationRole</class>
         <name>fod-admin</name>
      </principal>
    </principals>
  </grantee>
  <permissions>
     <permission>
       <class>oracle.fodemo.storefront.store.view.AccountPermission</class>
       <name>AccountPermission</name>
       <actions>view</actions>
     </permission>
   </permissions>
</grant>
```

The `<permission>` target name can be any name that helps you to identify the permission. For example, a permission that lets users view their account number might be named `AccountPermission`.

The actions that you can enter are those that your custom JAAS Permission class defines. For example, in the Fusion Order Demo application the class `oracle.fodemo.storefront.store.view.AccountPermission` defines the single action `view`.

Before you begin:

It may be helpful to have an understanding of the ADF Permission class. For more information, see Section 35.11.2, "How to Evaluate Policies Using Custom JAAS Permissions and EL."

You will need to complete this task:

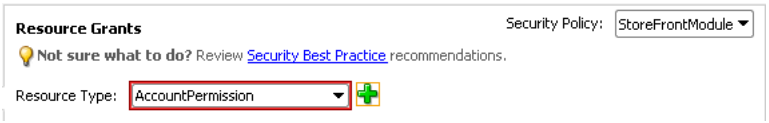
Create the custom JAAS permission class, as described in Section 35.11.2.1, "Creating the Custom JAAS Permission Class."

To create the ADF security policy using the custom permission:

1. In the main menu, choose **Application** and then **Secure > Resource Grants**.
2. In the Resource Grants page of the `jazn-data.xml` file overview editor, select the custom resource from the **Resource Types** dropdown list.

   The overview editor displays all custom resources. Initially, the custom resource will not have a resource type associated with it and the editor highlights this, as shown in Figure 35–33.

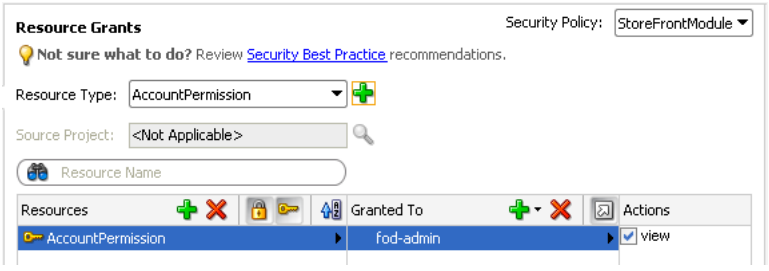
Figure 35–33 Displaying a Custom Resource in the Overview Editor



3. Next to the **Resource Type** field, click the **New Resource Type** icon.
4. In the Create Resource Type dialog, enter the name and action and click **OK**.
5. In Resource Grants page of the overview editor, in the **Granted to** column, click the **Add Grantee** icon and choose **Add Application Role**.
6. In the Select Application Roles dialog, select the application role and click **OK**.
7. In the Resource Grants page of the overview editor, in the **Actions** column, select desired action.

   The overview editor displays the custom permission grant, as shown in Figure 35–34.

Figure 35–34 Creating a Custom Permission Grant in the Overview Editor



35.11.2.3 Associating the Rendering of a UI Component with a Custom Permission

You use the Expression Builder dialog that you display for the UI component display property to define an EL expression. For example, in the Fusion Order Demo application, the page `myOrders.jpx` defines the `userGrantedPermission` expression on the `value` attribute of the `af:outputText#ot18` text field, as shown in Example 35–27. In this case, the expression tests whether the user has permission

and then either displays the account number (through `bindings.AccountNumber.inputValue`) or, when the user does not have permission, displays `XXXXXXXXXXXX` in place of the account number. Because the expression is not defined on the text field's `rendered` attribute, the page always displays the field.

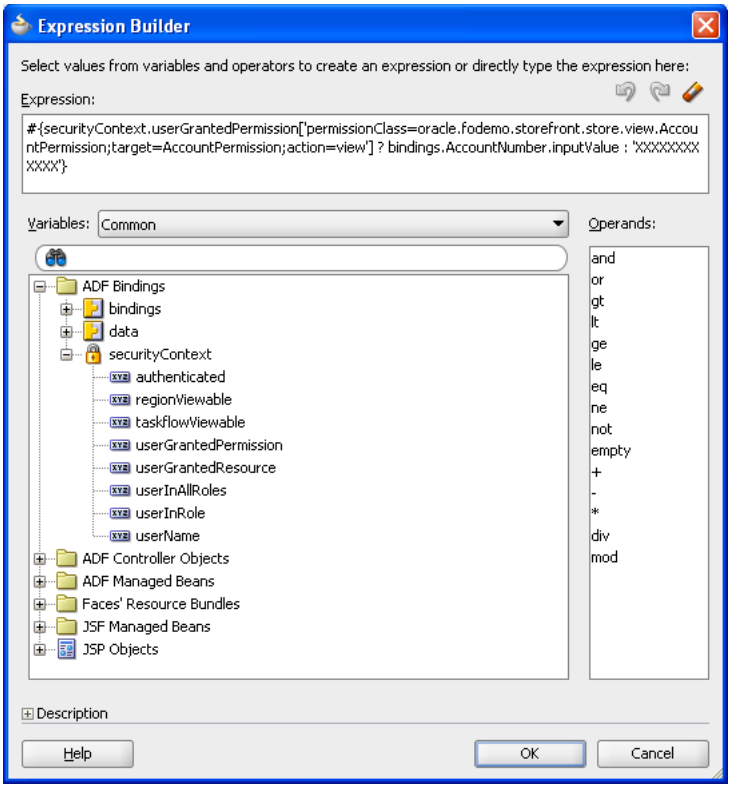
When you run the application, the component will be rendered or hidden based on the user's ability to view the target page.

Example 35–27 Custom Permission Expression on Value Attribute

```
#{securityContext.userGrantedPermission
       ['permissionClass=oracle.fodemo.storefront.store.view.AccountPermission;
         target=AccountPermission;action=view']
     ? bindings.AccountNumber.inputValue : 'XXXXXXXXXXXX'}
```

Figure 35–35 shows how the expression appears in the Expression Builder dialog.

Figure 35–35 Defining EL in the Expression Builder Dialog



Before you begin:

It may be helpful to have an understanding of the ADF Permission class. For more information, see Section 35.11.2, "How to Evaluate Policies Using Custom JAAS Permissions and EL."

You will need to complete these tasks:

1. Create the custom JAAS permission class, as described in Section 35.11.2.1, "Creating the Custom JAAS Permission Class."
2. Create the ADF security policy using the custom permission, as described in Section 35.11.2.2, "Creating the ADF Security Policy Using a Custom Permission."

To associate the rendering of a UI component with a user's granted custom permission:

1. In the Application Navigator, double-click the page.
2. Select the component that is used to navigate to the secured page.
3. In the Property Inspector, click the **Property Menu** dropdown menu displayed to the right of the **Rendered** field and choose **Expression Builder**.
4. In the Expression Builder, expand the **ADF Bindings - securityContext** node and select **userGrantedPermission**, and then, in the **Expression** field, enter a concatenated string that defines the permission.

   Enter the permission string as a semicolon-separated concatenation of `permissionClass=`*`qualifiedClassName`*`;target=`*`artifactName`*`;action=`*`actionName`*. For example, to protect an account number that a text field displays in a page, you would enter an expression similar to the one shown in Example 35–28, where the permission for `userGrantedPermission` is the same name as the custom JAAS permission grant.

Example 35–28 Custom JAAS Permission Expression

```
#{securityContext.userGrantedPermission
       ['permissionClass=oracle.fodemo.storefront.store.view.AccountPermission;
         target=AccountPermission;action=view']}
```

   In Example 35–28, the expression evaluates the permission based on the custom JAAS permission definition named `AccountPermission` that you added to the application policy store.

5. Click **OK**.

35.11.3 Getting Information from the ADF Security Context

The implementation of security in a Fusion web application is by definition an implementation of the security infrastructure of the ADF Security framework. As such, the security context of the framework allows access to information that is required as you define the policies and the overall security for your application.

35.11.3.1 How to Determine Whether Security Is Enabled

Because the enforcement of ADF Security can be turned on and off at the container level independent of the application, you should determine whether ADF Security is enabled prior to making authorization checks. You can achieve this by calling the `isAuthorizationEnabled()` method of the ADF security context, as shown in Example 35–29.

Example 35–29 Using the isAuthorizationEnabled() Method of the ADF Security Context

```
if (ADFContext.getCurrent().getSecurityContext().isAuthorizationEnabled()){
  //Authorization checks are performed here.
}
```

35.11.3.2 How to Determine Whether the User Is Authenticated

As the user principal in a Fusion web application is never `null` (that is, it is either `anonymous` for unauthenticated users or the actual user name for authenticated users), it is not possible to simply check whether the user principal is `null` to determine if the user has logged on or not. As such, you must use a method to take into account that a user principal of `anonymous` indicates that the user has not

authenticated. You can achieve this by calling the `isAuthenticated()` method of the ADF security context, as shown in Example 35–30.

Example 35–30 Using the isAuthenticated() Method of the ADF Security Context

```
// ============ User's Authenticated Status =============
private boolean _authenticated;
public boolean isAuthenticated() {
_authenticated = ADFContext.getCurrent().getSecurityContext().isAuthenticated();
    return _authenticated;
}
```

35.11.3.3 How to Determine the Current User Name, Enterprise Name, or Enterprise ID

Fusion web applications support the concept of public pages that, while secured, are available to all users. Furthermore, components on the web pages, such as portlets, require knowledge of the current user identity. As such, the user name in a Fusion web application will never be `null`. If an unauthenticated user accesses the page, the user name `anonymous` will be passed to page components. When the Fusion web application registers an enterprise name for the user, the enterprise name may also be obtained. The enterprise name is an alias that the user knows for themselves and can use to login.

You can determine the current user's name by evaluating the `getUserName()` method of the ADF security context, as shown in Example 35–31. This method returns the string `anonymous` for all unauthenticated users and the actual authenticated user's name for authenticated users.

Example 35–31 Using the getUserName() Method of the ADF Security Context

```
// ============ Current User's Name/PrincipalName =============
     public String getCurrentUser() {
      _currentUser = ADFContext.getCurrent().getSecurityContext().getUserName();
         return _currentUser;
     }
```

Because the traditional method for determining a user name in a Faces-based application (`FacesContext.getCurrentInstance().getExternalContext().getRemoteUser()`) returns `null` for unauthenticated users, you need to use additional logic to handle the public user case if you use that method.

You can determine the current user's enterprise name by evaluating the `getEnterpriseName()` method of the ADF context, as shown in Example 35–32.

Example 35–32 Using the getEnterpriseName() Method of the ADF Context

```
// ============ Current User's Enterprise Name =============
public String getEnterpriseName() {
  _enterpriseName = ADFContext.getCurrent().getEnterpriseName();
  return _enterpriseName;
}
```

You can determine the current user's enterprise ID by evaluating the `getEnterpriseId()` method of the ADF context, as shown in Example 35–33. This method returns the string `anonymous` for all unauthenticated users and the actual authenticated user's name for authenticated users.

Example 35–33 Using the getEnterpriseId() Method of the ADF Context

```
// ============ Current User's Enterprise ID =============
public String getEnterpriseId() {
    _enterpriseId = ADFContext.getCurrent().getEnterpriseId();
    return _enterpriseId;
}
```

35.11.3.4 How to Determine Membership of a Java EE Security Role

As Fusion web applications are JavaServer Faces-based applications, you can use the `isUserInRole(roleName)` method of the Faces external context, as shown in Example 35–34, to determine whether a user is in a specified role. Because ADF Security is based around JAAS policies, you should not need to use Java EE security roles to secure pages associated with ADF security-aware resources based on role membership. However, you might use the method to check the role for a page that is not associated with an ADF security-aware resource.

In this example, a convenience method (`checkIsUserInRole`) is defined. The use of this method within a managed bean enables you to expose membership of a named role as an attribute, which can then be used in EL.

Example 35–34 Using the isUserInRole(roleName)) Method of the Faces Context

```
public boolean checkIsUserInRole(String roleName){
        return
(FacesContext.getCurrentInstance().getExternalContext().isUserInRole(roleName));
}

public boolean isCustomer() {
        return (checkIsUserInRole("fod-users"));
 }
```

35.11.3.5 How to Determine Permission Using Java

To evaluate the security policies from within Java, you can use the `hasPermission` method of the ADF security context. This method takes a permission object (defined by the resource and action combination) and returns `true` if the user has the corresponding permission.

In Example 35–35, a convenience function is defined to enable you to pass in the name of the page and the desired action, returning `true` or `false` based on the user's permissions. Because this convenience function is checking page permissions, the `RegionPermission` class is used to define the permission object that is passed to the `hasPermission` method.

Example 35–35 Using the hasPermission() Method to Evaluate Access Policies

```
private boolean TestPermission (String PageName, String Action)  {
  Permission p = new RegionPermission("view.pageDefs." + PageName + "PageDef",
                                      Action);
  if (p != null) {
     return ADFContext.getCurrent().getSecurityContext().hasPermission(p);
 }
 else {
     return (true);
 }
```

As it is possible to determine the user's permission for a target page from within a backing bean, you can use this convenience method to dynamically alter the result of a Faces navigation action. In Example 35–36, you can see that a single command button

can point to different target pages depending on the user's permission. By checking the view permission from the most secured page (the manager page) to the least secured page (the public welcome page), the command button's backing bean will apply the appropriate action to direct the user to the page that corresponds to their permission level. The backing bean that returns the appropriate action is using the convenience method defined in Example 35–35.

Example 35–36 Altering a Page Navigation Result Based on an Authorization Check

```
//CommandButton Definition
<af:commandButton text="Goto Your Group Home page"
  binding="#{backing_content.commandButton1}"
  id="commandButton1"

  action="#{backing_content.getSecureNavigationAction}"/>

//Backing Bean Code
    public String getSecureNavigationAction() {
      String ActionName;
      if (TestPermission("ManagerPage", "view"))
        ActionName = "goToManagerPage";
      else if (TestPermission("EmployeePage", "view"))
        ActionName = "goToEmployeePage";
      else
        ActionName = "goToWelcomePage";
      return (ActionName);
    }
```

35.11.4 Best Practices for Working with ADF Security

These best practices summarize the rules that govern enforcement of security by the ADF Security framework. Understanding these best practices will help you to secure the application to allow users to access the web pages you intend.

Do build your application with ADF Security enabled from the start.

When you enable security, you essentially lock down the application and you will be required to make explicit permission grants to specific ADF security-aware resources you create. Knowing about these resources and making grants on them as you build the application will enable you to iteratively test security to ensure that you structure your application in a way that achieves the desired result.

Do define permission grants for bounded task flows.

Pages that the user accesses within the process of executing a bounded task flow will not be individually permission-checked and will run under the permission grants of the task flow. This means that any page that you add to the task flow should not have its own page definition-level security defined. Upon requesting a flow, the user will be allowed either to view all the pages of the task flow or to view none of the pages, depending on their level of access.

Do not define permission grants for individual pages of a bounded task flow.

It is important to realize that task flows do not prevent users from accessing pages directly. Any web page that is located in a directory that is publicly accessible can be reached from a browser using a URL. To ensure that pages referenced by a bounded task flow cannot be accessed directly, remove all permission grants that exist for their associated page definition file. When pages require additional security within the

context of a bounded task flow, wrap those pages in a sub-task flow with additional grants defined on the nested task flow.

Do use task flows to reduce the number of access points exposed to end users.

When you use task flows you can reduce the number of access points that you expose to end users. For example, configure an unbounded task flow to display one page that provides navigation to the remaining pages in your application. Use bounded task flows for the remaining pages in the application. By setting the URL Invoke property of these bounded task flows to `url-invoke-disallowed`, your application has one access point (the page on the unbounded task flow). For more information about the URL Invoke property, see Section 19.6.4, "How to Call a Bounded Task Flow Using a URL."

Do define permission grants for individual pages outside of a bounded task flow.

Page-level security is checked for pages that have an associated page definition binding file *only* if the page is directly accessed or if the page is accessed in an *unbounded* task flow. There is a one-to-one relationship between the page definition file and the web page it secures.

If you want to secure a page that uses no ADF bindings, you can create an empty page definition binding file for the page.

Do define custom permissions to render UI component based on the user's access rights.

Custom ADF permission classes let you extend ADF Security to define custom actions for use in grants. This gives you the flexibility to define security policies to manage the user's ability to view UI components without having to overload the built-in actions defined by the ADF resources' permission classes.

Do define entity object attribute permissions to manage the user's access rights to row-level data displayed by UI components.

Entity objects and entity object attributes both define permission classes that let you define permissions for the read, update, and delete operations that the entity object initiates on its data source. In the case of these data model project components, you must explicitly grant permissions to an application role in order to opt into ADF Security authorization. However, once you enable authorization for an entity object, all rows of data defined by the entity object will be protected by the grant. At this level of granularity, your table component would render in the web page either with all data visible or with no data visible—depending on the user's access rights. As an alternative to securing the entire collection, you can secure individual columns of data. This level of granularity is supported by permissions you set on the individual attributes of entity objects. When entity objects are secured, users may see only portions of the data that the table component displays.

Do use task flow or page-level permission grants to avoid exposing row-level create/insert operations to users with view-only permission.

The correct way to control access to a page that should allow only certain users to update new rows in a table is to use task flow or page-level permission grants. However, as an alternative, it is possible to secure table buttons corresponding to particular operations by specifying an EL expression to test the user's access rights to view the button. When the custom permission is defined and the `userGrantedPermission` expression is set on the `Rendered` property of the button, only users with sufficient privileges will see the button. This may be useful in a

case where the user interface displays a page that is not restricted and view-only permission for row-level data is defined for the entity object. In this case, when viewed by the user, the Delete button for the editable table associated with the entity object will appear disabled. However, in the case of an input table, the user interface does not disable the button for the `CreateInsert` operation even though the user may not have update permission.

Do not use JDeveloper as a user identity provisioning tool.

JDeveloper must not be used as an identity store provisioning tool, and you must be careful not to deploy the application with user identities that you create for testing purposes. Deploying user identities with the application introduces the risk that malicious users may gain unintended access. Instead, always rely on the system administrator to configure user identities through the tools provided by the domain-level identity management system. You should delete all users and groups that you create in the `jazn-data.xml` file before deploying the application.

Do not allow users to access a web page by its file name.

When you deploy the Fusion web application, you should always permit users to access the web page from a view activity defined in the ADF Controller configuration file. Do not allow users to access the JSPX file directly by its physical name (for example, similar to the file name `AllDepartments.jspx`.

Assuming the view activity is named `AllDepartments`, then there are two ways to call the page:

1. `localhost:7101/myapp/faces/AllDepartments`
2. `localhost:7101/myapp/faces/AllDepartments.jspx`

The difference is that the call 1) is in the context of the ADF Controller task flow, which means that navigation on the page will work and any managed beans that are referenced by the page will be properly instantiated. The call in 2) also serves the page, however, the page may not function fully. This may be considered a security breach.

To prevent direct JSPX file access, move the JSPX file under the `/public_html/WEB-INF` directory so that direct file access is no longer possible. To access a document, users will have to call its view activity name.

Note that this suggestion does not protect documents that are unprotected in ADF Security. It only helps to lock down access to the physical file itself.

Thus, the following security guidelines still apply:

1. Apply ADF Security permissions to all JSPX documents associated with view activities defined in the `adfc-config.xml` file.
2. Move all JSPX documents in the user interface project under the `/public_html/WEB-INF` directory to prevent direct file access.
3. Limit the pages in the `adfc-config.xml` to the absolute minimum and place all other pages into bounded task flows.
4. Make bounded task flows inaccessible from direct URL access (which is the default configuration setting for new task flows).
5. Apply ADF Security permissions to bounded task flows.

36

Testing and Debugging ADF Components

This chapter describes the process of debugging your user interface project. It describes several JDeveloper tools that can be used to help debug an application. It contains debugging procedures with breakpoints using the ADF Declarative Debugger. Finally, it explains how to write and run regression tests for your ADF Business Components-based business services.

This chapter includes the following sections:

- Section 36.1, "About ADF Debugging"
- Section 36.2, "Correcting Simple Oracle ADF Compilation Errors"
- Section 36.3, "Correcting Simple Oracle ADF Runtime Errors"
- Section 36.4, "Reloading Oracle ADF Metadata in Integrated WebLogic Server"
- Section 36.5, "Validating ADF Controller Metadata"
- Section 36.6, "Using the ADF Logger"
- Section 36.7, "Using the Oracle ADF Model Tester for Testing and Debugging"
- Section 36.8, "Using the ADF Declarative Debugger"
- Section 36.9, "Setting ADF Declarative Breakpoints"
- Section 36.10, "Setting Java Code Breakpoints"
- Section 36.11, "Regression Testing with JUnit"

36.1 About ADF Debugging

Like any debugging task, debugging the web application's interaction with Oracle Application Development Framework (Oracle ADF) is a process of isolating specific contributing factors. However, in the case of web applications, generally this process does not involve compiling Java source code. Your web pages contain no Java source code, as such, to compile. In fact, you may not realize that a problem exists until you run and attempt to use the application. For example, these failures are only visible at runtime:

- A page not found servlet error
- The page is found, but the components display without data
- The page fails to display data after executing a method call or built-in operation (like Next or Previous)
- The page displays, but a method call or built-in operation fails to execute at all
- The page displays, but unexpected validation errors occur

The failure to display data or to execute a method call arises from the interaction between the web page's components and the ADF Model layer. When a runtime failure is observed during ADF lifecycle processing, the sequence of preparing the model, updating the values, invoking the actions, and, finally, rendering the data failed to complete.

Fortunately, most failures in the web application's interaction with Oracle ADF result from simple and easy-to-fix errors in the declarative information that the application defines or in the EL expressions that access the runtime objects of the page's ADF binding container.

In your databound Fusion web application, you should examine the declarative information and EL expressions as likely contributing factors when runtime failures are observed. To understand editing the declarative files, see Section 36.2, "Correcting Simple Oracle ADF Compilation Errors," and Section 36.3, "Correcting Simple Oracle ADF Runtime Errors."

One of the most useful diagnostic tools is the ADF Logger. You use this logging mechanism in JDeveloper to capture runtime traces messages. With ADF logging enabled, JDeveloper displays the application trace in the Message Log window. The trace includes runtime messages that may help you to quickly identify the origin of an application error. Read Section 36.6, "Using the ADF Logger," to configure the ADF Logger to display detailed trace messages.

Supported Oracle ADF customers can request Oracle ADF source code from Oracle Worldwide Support. This can make debugging ADF Business Components framework code a lot easier. Read Section 36.8.1, "Using ADF Source Code with the Debugger," to understand how to configure JDeveloper to use the Oracle ADF source code.

If the error cannot be easily identified, you can utilize the ADF Declarative Debugger in JDeveloper to set breakpoints. When a breakpoint is reached, the execution of the application is paused and you can examine the data that the ADF binding container has to work with, and compare it to what you expect the data to be. Depending on the types of breakpoints, you may be able to use the step functions to move from one breakpoint to another. For more information about the debugger, read Section 36.8, "Using the ADF Declarative Debugger."

JDeveloper provides integration with JUnit for your Fusion web application through a wizard that generates regression test cases. Read Section 36.11, "Regression Testing with JUnit," to understand how to write test suites for your application.

36.2 Correcting Simple Oracle ADF Compilation Errors

When you create web pages and work with the ADF data controls to create the ADF binding definitions in JDeveloper, the Oracle ADF declarative files you edit must conform to the XML schema defined by Oracle ADF. When an XML syntax error occurs, the JDeveloper XML compiler immediately displays the error in the Structure window.

Although there is some syntax checking during design time, the JDeveloper compiler is currently limited by an inability to resolve EL expressions. EL expressions in your web pages interact directly with various runtime objects in the web environment, including the web page's ADF binding container. At present, errors in EL expressions can be observed only at runtime. Thus, the presence of a single typing error in an object-access expression will not be detected by the compiler, but will manifest at runtime as a failure to interact with the binding container and a failure to display data in the page. For information about debugging runtime errors, see Section 36.3, "Correcting Simple Oracle ADF Runtime Errors."

> **Tip:** The Expression Builder is a dialog that helps you build EL expressions by providing lists of objects, managed beans, and properties. It is particularly useful when creating or editing ADF databound EL expressions because it provides a hierarchical list of ADF binding objects and their valid properties from which you can select. You should use the Expression Builder to avoid introducing typing errors. For details, see Section 13.8, "Creating ADF Data Binding EL Expressions."

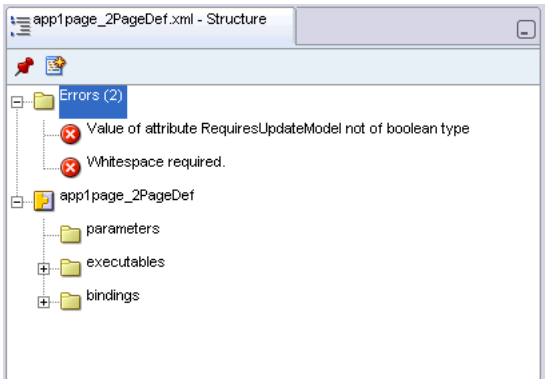
Example 36–1 illustrates two simple compilation errors contained in a page definition file: `tru` instead of `true` and `id="CountryCodesView1Iterator"/` instead of `id="CountryCodesView1Iterator"/>` (that is, the ID is missing a closing angle bracket).

Example 36–1 Sample Page Definition File with Two Errors

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="11.1.1.42.54" id="app1page_2PageDef"
                Package="project1.pageDefs">
  <parameters/>
  <executables>
    <iterator Binds="CountryCodesView1" RangeSize="10"
              DataControl="AppModuleDataControl"
              id="CountryCodesView1Iterator"/
  </executables>
  <bindings>
    <action id="Find" RequiresUpdateModel="tru" Action="3"
            IterBinding="CountryCodesView1Iterator"/>
```

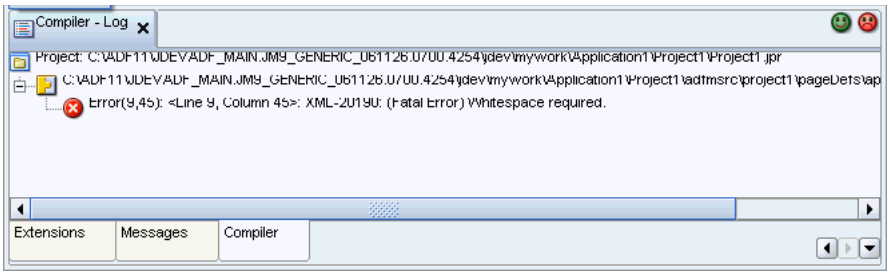
During compilation, the Structure window displays the XML errors in the page, as shown in Figure 36–1.

Figure 36–1 The Structure Window Displays XML Errors



The Compiler-Log window also displays the compilation errors in the page, as shown in Figure 36–2.

Figure 36–2 The Compiler Window Displays XML Compile Errors



To view and correct schema validation errors:

1. From the main menu, choose **View > Structure** to open the Structure window or **View > Log** to open the Log Window.
2. In either window, double-click the error message to open the file in the XML editor.
3. In the XML editor, locate the highlighted lines.

   The highlighted lines will be lines with errors.
4. Correct any errors.

   After an error has been corrected, the corresponding error message will be automatically removed from the Structure window.
5. Optionally, you can recompile the project by choosing **Run > Make** and checking to see whether the compiler still produces the error message.

36.3 Correcting Simple Oracle ADF Runtime Errors

Failures of the ADF Model layer cannot be detected by the JDeveloper compiler, in part because the page's data-display and method-execution behavior relies on the declarative ADF page definition files. The ADF Model layer utilizes those declarative files at runtime to create the objects of the ADF binding container.

To go beyond simple schema validation, you will want to routinely run and test your web pages to ensure that none of the following conditions exists:

- The project dependency between the data model project and the user interface project is disabled.

  By default, the dependency between projects is enabled whenever you create a web page that accesses a data control in the data model project. However, if the dependency is disabled and remains disabled when you attempt to run the application, an internal servlet error will be generated at runtime:

  ```
  oracle.jbo.NoDefException: JBO-25002: Definition
  model.DataControls.dcx of type null not found
  ```

  To correct the error, double-click the user interface project, and select the **Dependencies** node in the dialog. Make sure that the ***ModelProjectName*.jpr** option appears selected in the panel.
- Page definition files have been renamed, but the `DataBindings.cpx` file still references the original page definition file names.

  While JDeveloper does not permit these files to be renamed within the IDE, if a page definition file is renamed outside of JDeveloper and the references in the

DataBindings.cpx file are not also updated, an internal servlet error will be generated at runtime:

```
oracle.jbo.NoDefException: JBO-25002: Definition
oracle.<path>.pageDefs.<pagedefinitionName> of type Form
Binding Definition not found
```

To correct the error, open the DataBindings.cpx file and use the source editor to edit the page definition file names that appear in the <pageMap> and <pageDefinitionUsages> elements.

- The web page file (.jsp or .jspx) has been renamed, but the DataBindings.cpx file still references the original file name of the same web page.

  The page controller uses the page's URL to determine the correct page definition to use to create the ADF binding container for the web page. If the page's name from the URL does not match the <pageMap> element of the DataBindings.cpx file, an internal servlet error will be generated at runtime:

  ```
  javax.faces.el.PropertyNotFoundException: Error testing
  property <propertyname>
  ```

  To correct the error, open the DataBindings.cpx file and use the source editor to edit the web page file names that appear in the <pageMap> element.

- Bindings have been renamed in the web page EL expressions, but the page definition file still references the original binding object names.

  The web page may fail to display information that you expect to see. To correct the error, compare the binding names in the page definition file and the EL expression responsible for displaying the missing part of the page. Most likely the mismatch will occur on a value binding, with the consequence that the component will appear but without data. Should the mismatch occur on an iterator binding name, the error may be more subtle and may require deep debugging to isolate the source of the mismatch.

- Bindings in the page definition file have been renamed or deleted, and the EL expressions still reference the original binding object names.

  Because the default error-handling mechanism will catch some runtime errors from the ADF binding container, this type of error can be very easy to find. For example, if an iterator binding named findUsersByNameIter was renamed in the page definition file, yet the page still refers to the original name, this error will display in the web page:

  ```
  JBO-25005: Object name <iterator> for type Iterator Binding
  Definition is invalid
  ```

  To correct the error, right-click the name in the web page and choose **Go to Page Definition** to locate the correct binding name to use in the EL expression.

- EL expressions were written manually instead of using the expression picker dialog and invalid object names or property names were introduced.

  This error may not be easy to find. Depending on which EL expression contains the error, you may or may not see a servlet error message. For example, if the error occurs in a binding property with no runtime consequence, such as displaying a label name, the page will function normally but the label will not be displayed. However, if the error occurs in a binding that executes a method, an internal servlet error javax.faces.el.MethodNotFoundException: *methodname* will display. Or, in the case of an incorrectly typed property name on the method

expression, the servlet error `javax.faces.el.PropertyNotFoundException: propertyname` will display.

If this list of typical errors does not help you to find and fix a runtime error, you can initiate debugging within JDeveloper to find the contributing factor. For an ADF application, start setting ADF declarative breakpoints to find the problem. Using the ADF Declarative Debugger to set ADF declarative breakpoints is described in Section 36.8, "Using the ADF Declarative Debugger," and Section 36.9, "Setting ADF Declarative Breakpoints." This process involves pausing the execution of the application as it proceeds through the application and examining data. You can also use the ADF Declarative Debugger to set Java code breakpoints, as described in Section 36.10, "Setting Java Code Breakpoints."

36.4 Reloading Oracle ADF Metadata in Integrated WebLogic Server

JDeveloper support for hot reloading of Oracle ADF metadata is an alternative to quitting the running application, editing your project's XML definition files, redeploying, and rerunning the application in Integrated WebLogic Server to view the latest changes.

Changes that you make to the Fusion web application projects will not be picked up automatically by an application that you have deployed to Integrated WebLogic Server. You can, however, reload metadata from the data model project and user interface project any time you want to synchronize the running application with changes you have made to your application's XML definition files.

To reload metadata so your changes are reflected in the deployed Fusion web application, you must recompile the project and refresh the web browser.

Metadata that JDeveloper will hot reload in Integrated WebLogic Server, include:

- In the data model project, changes to the definition files of business components.
- In the user interface project, changes to the binding definitions in the page definition files and changes to task flows in the task flow definition files.

This support makes it possible to make incremental changes and test them.

36.5 Validating ADF Controller Metadata

Basic validation is performed when ADF Controller retrieves metadata. The most serious errors, for example, a task flow that is missing a default activity, result in parsing exceptions.

The `enable-grammar-validation` setting in `adf-config.xml` allows you to validate the grammar in ADF Controller metadata before deploying an application. When `enable-grammar-validation` is set to `true`, ADF Controller metadata is validated against ADF Controller XSDs. For example, invalid characters in ADF Controller metadata, such as a slash (`/`) in a view activity ID, are flagged as exceptions.

By default, `enable-grammar-validation` is set to `false`. For performance reasons, it should be set to `true` only during application development or when troubleshooting an application.

36.6 Using the ADF Logger

If you are not able to easily find the error in either your web page or its corresponding page definition file, you can use the JDeveloper debugging tools to investigate where your application failure occurs.

You configure the logging session by editing the `logging.xml` configuration file using the editor for Oracle Diagnostic Logging Configuration. Logging configuration can be set at any time, even while the application is running in JDeveloper.

Unlike many files in JDeveloper, you cannot directly open the `logging.xml` file. Instead you use menu commands at these locations to open the Oracle Diagnostic Logging Configuration editor:

- Right-click an active server instance in the Application Server Navigator window.
- Click the Actions dropdown menu displayed in the Log window's debugger process panel, after you have started the application in debug mode.

You use the Oracle Diagnostic Logging Configuration editor to set the desired logging level to control the level and number of messages that are displayed. You can set the logging level for both persistent and transient loggers and declare handlers for each logger.

JDeveloper creates diagnostic log files in the Oracle Diagnostic Logging (ODL) format, used by Oracle Fusion Middleware components. Log file naming and the format of the contents of log files conform to an Oracle standard. By default, the diagnostic messages are in text format. For more information about the ODL format, see the *Oracle Application Server Administrator's Guide*.

In the editor, as an alternative to the default ODL format, you can configure the Java Logger to display Java diagnostic messages.

After you have created a log, you can view and filter the log messages with Oracle Diagnostic Log Analyzer. This tool allows you to set filters for different log levels, define message time frames, and search on message text.

You can then use the ADF Declarative Debugger to set breakpoints and examine the the application. For more information, see Section 36.9, "Setting ADF Declarative Breakpoints," and Section 36.10, "Setting Java Code Breakpoints."

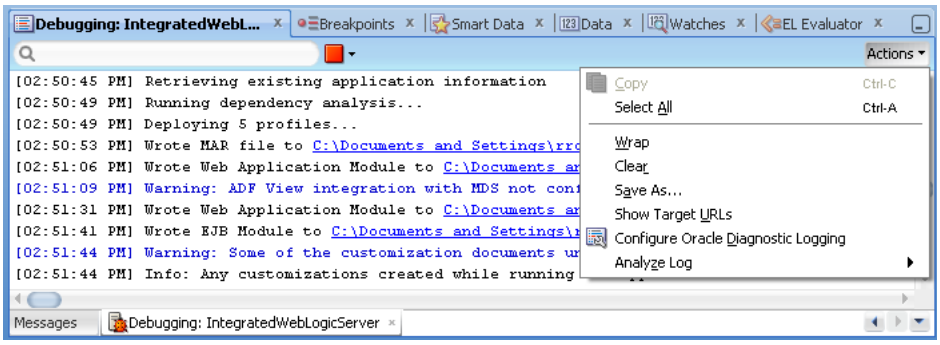
36.6.1 How to Set ADF Logging Levels

You can use the Oracle Diagnostic Logging Configuration editor to configure the logging levels specified in the `logging.xml` configuration file. The file can be configured before and while the application is running in Integrated WebLogic Server. The changes will apply without the need to restart the server.

When Integrated WebLogic Server is running, you can define both persistent and transient loggers. When Integrated WebLogic Server is not running, you can only define persistent loggers. The transient loggers will last only for the session and will not be entered in the `logging.xml` configuration file. If the server is not running, you must explicitly save the configuration changes to the `logging.xml` file for the updates to take effect in the next server run.

You can access the Oracle Diagnostic Logging Configuration editor from the Application Server Navigator or from the Log window, which is shown in Figure 36–3.

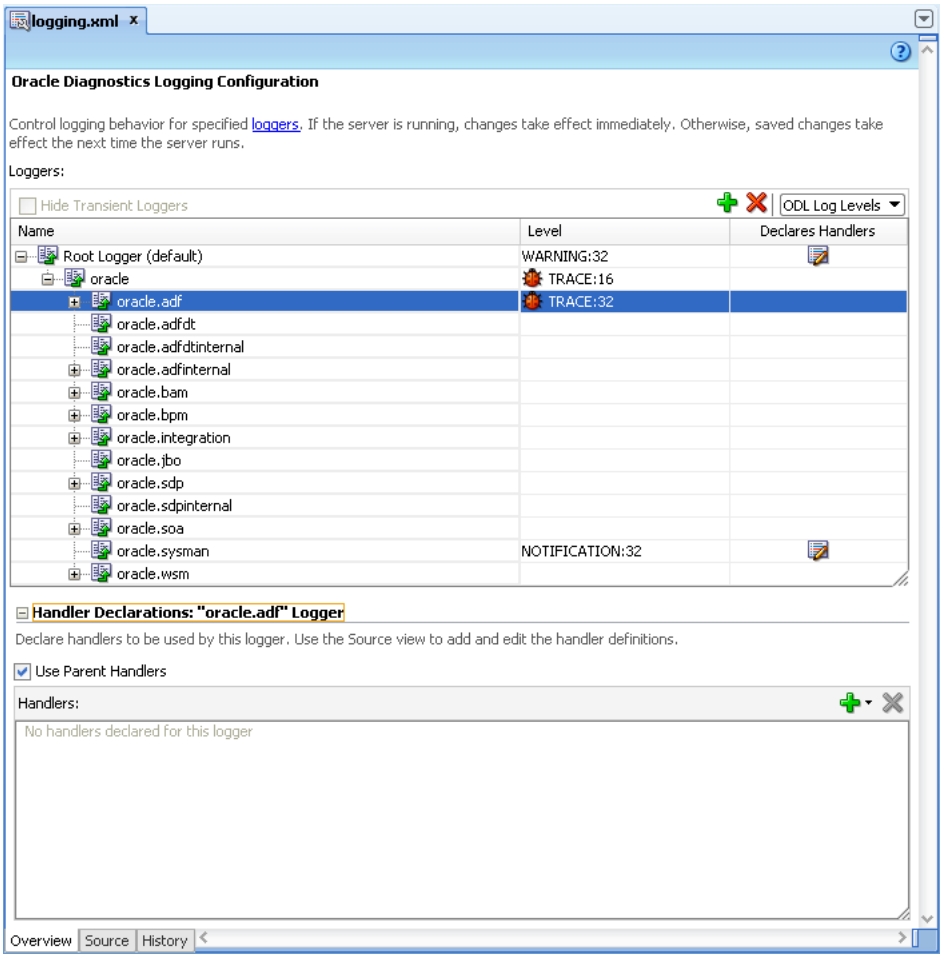
Figure 36–3 Log Window with Toolbar



However, while the server is running, when you access the editor via the Log window menu, then the editor has the ability to add transient loggers.

Figure 36–4 shows the Oracle Diagnostic Logging Configuration while the server is running.

Figure 36–4 Editor for Oracle Diagnostic Logging Configuration



You can only use JDeveloper menu commands to open the `logging.xml` configuration file and launch the editor for Oracle Diagnostic Logging Configuration. However, you may find the following information about the location of the configuration file useful.

Note: You can declare and add log handler definitions by clicking the **Source** tab and entering them in the XML editor.

If you are using Integrated WebLogic Server in JDeveloper on the Windows platform, you can find the `logging.xml` configuration file in a location similar to:

```
C:\Documents and
Settings\username\ApplicationData\JDeveloper\latest_system_
folder\DefaultDomain\config\fmwconfig\servers\DefaultServer
```

The log files for Integrated WebLogic Server are in a location similar to:

```
C:\Documents and
Settings\username\ApplicationData\JDeveloper\latest_system_
folder\DefaultDomain\servers\DefaultServer\logs
```

The log files for a standalone WebLogic Server instance are in a location similar to:

```
$domain_home/servers/your_servername/logs
```

You can configure logging levels before a test run from the Application Server Navigator or during a debug session from the Log window toolbar.

To configure the log levels:

1. In the Application Server Navigator, right-click **IntegratedWebLogicServer** and choose **Configure Oracle Diagnostic Logging for** *server name*.

   Or, from the Log window's debug process panel, choose **Actions - Configure Oracle Diagnostic Logging.** The debug process panel is only visible in the Log window after you have started the application in debug mode.

2. In the editor for Oracle Diagnostics Logging Configuration, select **ODL Log Levels** or **Java Log Levels** for the logger type you want to view.

3. If you want to see persistent loggers only, select **Hide Transient Loggers**.

4. To add a logger:

   a. If the server is running, click the **Add** icon dropdown menu and choose **Add Persistent Logger** or **Add Transient Logger**. If the server is not running, click **Add** to add a persistent logger. You cannot add a transient logger.

   b. In the Add Logger dialog, enter a logger name.

   c. Select the logging level.

   d. Click **OK**.

5. For any logger, including a newly created logger, you can specify its handlers by selecting from a list of available handlers by clicking the **Add** icon in the **Handler Declarations** section.

   Or, you can select **Use Parent Handlers** to assign its parent's handler to the logger. By default, a logger uses its parent's handler.

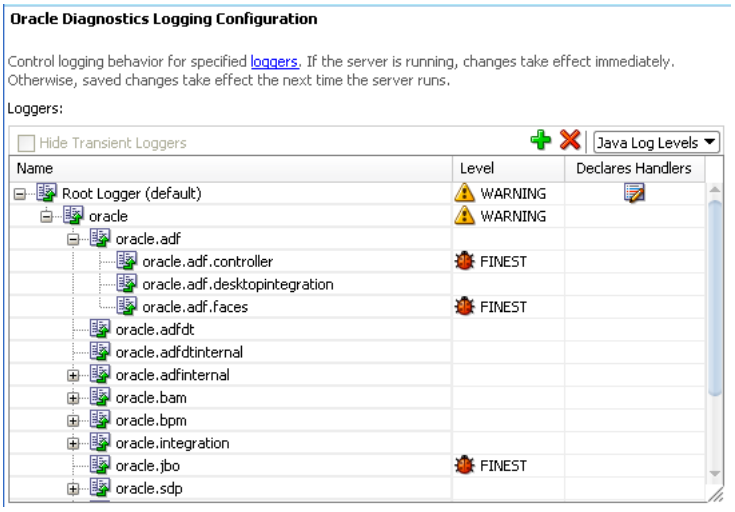
   Note: You can declare and add log handler definitions by clicking the **Source** tab and entering them in the XML editor.

36.6.2 How to Turn On Diagnostic Logging

Even before you use the actual debugger, running the application with loggers set to Java log level FINE, FINER, or FINEST will enable framework diagnostics logging. The debug diagnostic messages can be helpful to see what happens when the problem occurs. To enable debug diagnostic messages, use the Oracle Diagnostics Logging Configuration Editor to configure the desired loggers with a Java log level set to FINEST.

If you have configured diagnostic logging for the supported loggers, JDeveloper will direct debug diagnostics messages to the JDeveloper Log window. Figure 36–5 shows specific loggers configured to enable the most detailed Oracle ADF debug diagnostics.

Figure 36–5 Configuring a Logger for Oracle ADF Debugging



For backward compatibility, the Java system property `jbo.debugoutput` set to the value `console` or `ADFLogger` (to route diagnostics through the standard Logger implementation, which can be controlled in a standard way through the `logging.xml` file) is supported. The easiest way to set this system property while running your application inside JDeveloper is to edit your project properties and in the Run/Debug page, select a run configuration and click **Edit**. Then add the string `-Djbo.debugoutput=console` to the **Java Options** field.

36.6.3 How to Create an Oracle ADF Debugging Configuration

Oracle ADF leverages the Java Logging API (`java.util.logging.Logger`) to provide logging functionality when you run a debugging session. Java Logging is a standard API that is available in the Java platform at `http://download.oracle.com/javase/1.4.2/docs/guide/util/logging/overview.html`.

To create an ADF Model debugging configuration:

1. In the editor for Oracle Diagnostics Logging Configuration, select **ODL Log Levels** or **Java Log Levels** for the logger type you want to view.
2. Expand the **oracle.adf logger** node and select the Java log level **FINEST** for the desired Oracle ADF loggers.
3. Optionally, to enable ADF Business Components diagnostics, for the **oracle.jbo** logger, select the Java log level **FINEST**.

To create an ADF view Javascript logging configuration:

1. In the Application Navigator, double-click the application or project **web.xml** file.
2. In the source editor, add the following elements to the file:

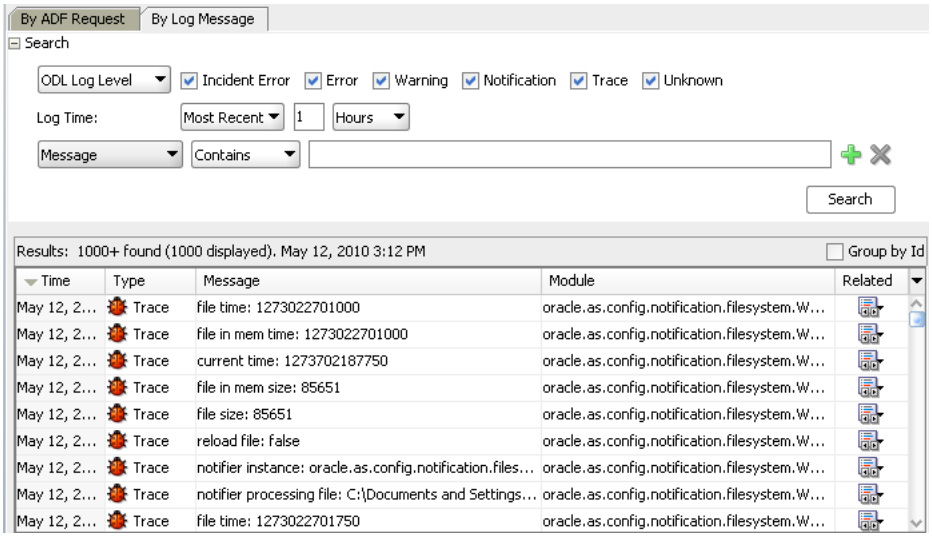
```
<context-param>
  <param-name>
     oracle.adf.view.rich.LOGGER_LEVEL
  </param-name>
  <param-value>
      FINE
  </param-value>
</context-param>
```

36.6.4 How to Use the Log Analyzer to View Log Messages

You can use Oracle Diagnostic Log Analyzer to view the log entries of a log file. The log analyzer allows you to filter the entries by log level, entry type, log time, and entry content (using a query panel). You can also order the messages and show and hide columns for better viewing.

Figure 36–6 shows Oracle Diagnostic Log Analyzer set to view ODL Log levels.

Figure 36–6 Oracle Diagnostic Log Analyzer Displays ODL Log Messages



You can also use the log analyzer on log files created in other test runs. For instance, you can analyze the log sent to you by another developer for another application.

36.6.4.1 Viewing Diagnostic Messages in the Log Analyzer

You can configure logging levels before a test run from the Application Server Navigator or during a debug session from the Log window toolbar. The level you specify will determine the type and quantity of log messages.

In the case of ADF events, all messages are generated for the ODL log at the level `Notification` or for the Java log at the level `Info`. Fewer ADF messages will be generated at the `Incident Error/Severe` and `Error/Warning` levels.

After you select the log level for the messages you wish to view, you can use the Search panel of the By Log Message page to filter the messages to display from the log file.

Note: For further details about search criteria that you can specify to search on ADF-specific messages, see Section 36.6.4.3, "Sorting Diagnostic Messages By ADF Events."

You can start the log analyzer before a test run from the **Tools** menu or during a debug session from the log window toolbar.

Before you begin:

It may be helpful to have an understanding of logging. For more information, see Section 36.6, "Using the ADF Logger."

You will need to complete these tasks:

1. Enable logging, as described in Section 36.6.2, "How to Turn On Diagnostic Logging."
2. Set logging levels, as described in Section 36.6.1, "How to Set ADF Logging Levels."
3. Create a log file, either from your test run or from another source.

To start the log analyzer:

1. From the main menu, choose **Tools > Oracle Diagnostic Log Analyzer**.

   Or, from the Log window **Action** menu, choose **Analyze Log** and then either **Current in Console** or **Open Selected** (to browse log files in the server log directory).
2. In the editor for Oracle Diagnostic Log Analyzer, click the **By Log Message** tab.
3. In the By Log Message page, navigate to the log file or enter the path and name of the log file.

   Tip: The Choose Log File dialog helps you to navigate to the directory that contains the log files generated by JDeveloper. Click the **Browse Log Files** icon next to the **Log** text field, and then click the **Server Logs** icon from the scroll list.
4. From the dropdown list, select either **ODL Log Level** or **Java Log Level**.
5. Select the corresponding checkbox for each type of log entry you want to view. You must select at least one type.

   The available ODL log level types are:

   - Incident Error
   - Error
   - Warning
   - Notification - corresponds to ADF event messages
   - Trace
   - Unknown.

   The available Java log level types are:

   - Severe
   - Warning

- Info - corresponds to ADF event messages
- Config
- Fine
- Finer
- Finest
- Unknown

6. Specify a time period for the entries you want to view. You can select the most recent *period* or a range.
7. To filter the results, use the Search panel to query the log for a text pattern. For additional Search panels, click **Add**. The supported search criteria include:
   - **Detail:** Filters text in statements from the stack where the method was invoked.
   - **Message:** Filters text in the logged messages.
   - **ADF Context Data:** Filters the log for data related to ADF lifecycle phase names, view object names, view object query statements, data control names, binding container names, and iterator binding names logged during the execution of ADF events.
   - **Source Method:** Filters the log by the method where the message is logged. For example, you can filter on the method `execute` to view all messages logged for view object query execution or ADF lifecycle phase execution.
   - **Application:** Filters the log by the application name where the message is logged. This is useful when the application is running in a composite application and you want to view messages for a specific application.
   - **Source Class:** Filters the log by the fully qualified class name of the method where the message is logged. To see more messages, enter a partial package name. For example, you can enter the partial package name `oracle.adf` or the full package name `oracle.jbo` to filter for all classes related to Oracle ADF.
   - **Module:** Filters the log by the fully qualified package name of the class where the message is logged. This is same package as the source class.
   - **Message Id:** Filters the log by the ID of the logged messages. Many messages share the same ID. For example, message ID `ADFC-52008` might have four `INFO` messages and one `Warning` message. You can select **Group by Id** in the log analyzer Results panel to group messages by their common ID.
8. To initiate the filters and display the log messages, click **Search**.
9. To order the results by the message ID, select the **Group by Id** checkbox.
10. To group the messages by time period or by request, in the **Related** column, select either **Related by Time** or **Related by Request**.
11. To show or hide columns in the **Results** section, click the dropdown list to the right of the column headers and select among the list of displayed columns to change the visibility of a column.

36.6.4.2 Using the Log Analyzer to Analyze the ADF Request

Because Oracle instrumented the Oracle ADF source code to generate log messages during the execution of the ADF lifecycle phases, you can use the log analyzer to

investigate the details of the active (or previous) page request in your running application. Specifically, the By ADF Request page of the log analyzer lets you view ADF event messages in a hierarchical list, organized by the sequence of their execution. It also provides a graphical representation of the duration of each event. When you run your application and start the log analyzer with ADF logging configured, you can use this page to quickly identify whether a component of your application is contributing to a performance bottleneck due to unusually long execution times.

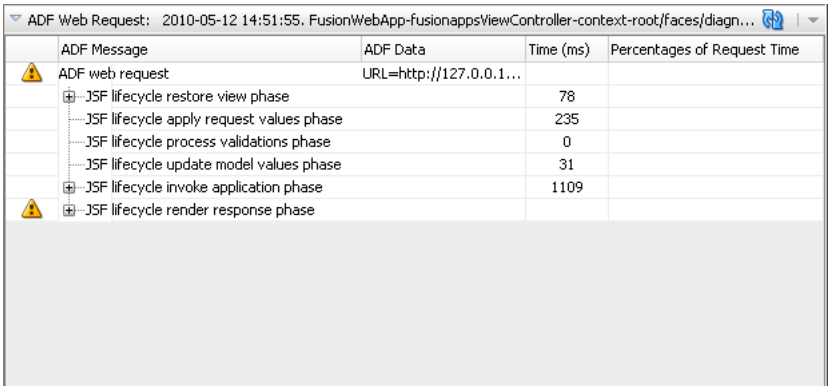
> **Note:** In contrast to the By Log Message page, the By ADF Request page of the log analyzer displays a hierarchical view of ADF event messages. The difference between these two pages is that the By ADF Request page focuses only on ADF page requests made when a page or region is submitted, and it provides the option to examine logged messages from multiple ADF requests. For details about the ADF page lifecycle, see Chapter 25, "Understanding the Fusion Page Lifecycle."

You can use the Search panel of the By ADF Request page to display one or more specific requests from the log file. The Search panel lets you combine any of the following search criteria:

- The number of requests to display
- The timestamp for the request recorded in the log
- The logged-in user name, application name in a composite application, or page name as displayed in the request header

If you search on any of these criteria before the page request is complete, the ADF Web Request panel in the log analyzer displays a yellow triangle symbol to indicate that the phase has not yet completed. Figure 36–7 shows the icon for the JSF lifecycle Render Response phase and the root node for the overall request. To update the ADF Web Request panel with the latest information from the log, you can click on the refresh icon in the panel header.

Figure 36–7 Oracle Diagnostic Log Analyzer Displays ADF Web Request in Progress

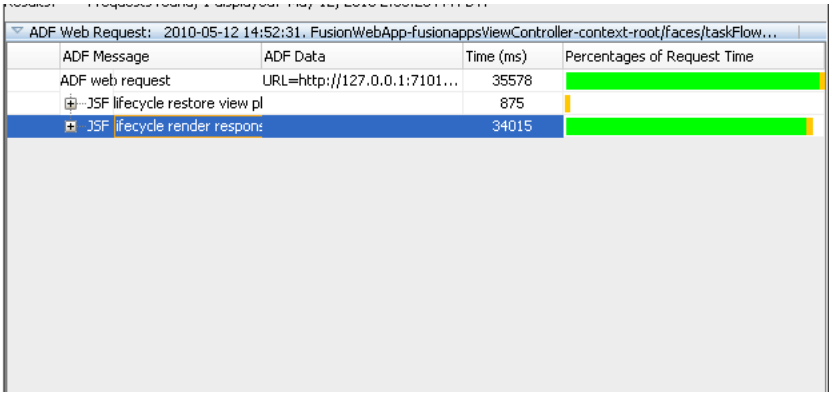


After the request is completed, the log analyzer displays duration bar graphs for each phase to show the percentages of request time that each event contributed to the overall page request. Figure 36–8 shows the ADF Web Request panel with the JSF lifecycle Render Response phase displaying 34015 milliseconds (34 seconds) for the duration of this portion of the overall page request. This example depicts an usually long duration for the render phase to illustrate how the bar graph can help you to

identify a performance bottleneck that may occur during the execution of a page request. The bar graph for the root node of the request event hierarchy (**ADF web request**) displays the total execution time.

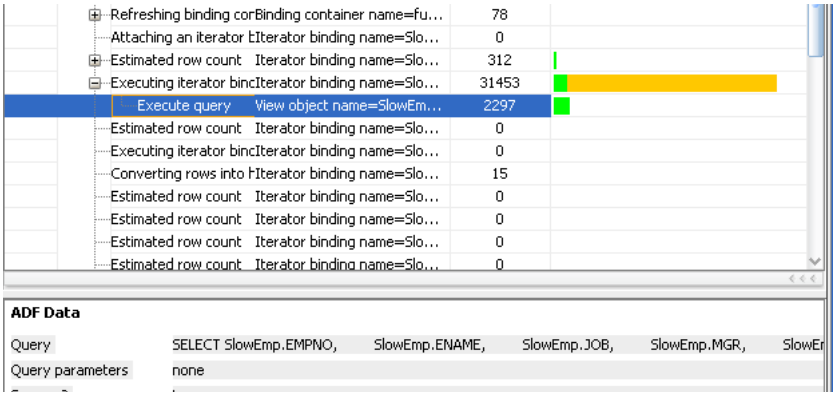
Note: The two colors present in the **Percentage Request Time** bar graph (green and orange) indicate which portion of the request's execution time resulted from ADF source code that was instrumented to generate ADF event messages and which portion resulted from ADF source code that is uninstrumented (and therefore cannot generate ADF event messages). Additionally, note that the individual phases of the request do not sum to equal the total request time. This is due to the fact that only those phases of the lifecycle that are useful are represented in the log analyzer.

Figure 36–8 Oracle Diagnostic Log Analyzer Displays Completed ADF Web Request



To examine the request in more detail, you can expand the tree for any ADF lifecycle node to further investigate where in the application the performance bottleneck occurred. Drilling down and then selecting the ADF event node in the ADF Web Request panel gives you details about the component associated with each ADF event. For instance, expanding the **JSF lifecycle render response phase** node displays all ADF events generated during that phase. Figure 36–9 shows the **JSF lifecycle render response phase** node expanded with a long request duration bar graph for the **Execute iterator binding** node and the **Execute query** node. The **Execute query** node has been selected to reveal detailed ADF data in the bottom portion of the panel, including the view object's name and query statement. By drilling down and selecting the ADF event with the long execution time as indicated by the bar graph, you can obtain, for example, the name of the view object in the data model project that should be tuned for improved performance.

Figure 36–9 Oracle Diagnostic Log Analyzer Displays ADF Event Messages with ADF Data



Before you begin:

It may be helpful to have an understanding of logging. For more information, see Section 36.6, "Using the ADF Logger."

You will need to complete these tasks:

1. Enable logging, as described in Section 36.6.2, "How to Turn On Diagnostic Logging."
2. Set logging levels, as described in Section 36.6.1, "How to Set ADF Logging Levels."

   To log ADF event messages, do not configure an ODL log level that is more restrictive than `Notification` or a Java log level that is more restrictive than `Info` for the following packages:

   - `oracle.adf` will log events generated by source code for the ADF Model data binding layer and ADF Controller source code.
   - `oracle.jbo` will log events generated by source code executed for ADF Business Components.
   - `oracle.adfinternal` will log events generated by source code executed from Oracle ADF internal classes.

     Tip: The default log level for the Root Logger displayed by the editor for Oracle Diagnostics Logging Configuration ensures that ADF event messages are logged.

3. Create a log file, either from your test run or from another source.

To display ADF request messages in the log analyzer:

1. From the main menu, choose **Tools > Oracle Diagnostic Log Analyzer**.

   Or, from the Log window **Action** menu, choose **Analyze Log** and then either **Current in Console** or **Open Selected** (to browse log files in the server log directory).

2. In the editor for Oracle Diagnostic Log Analyzer, click the **By ADF Request** tab.
3. In the By ADF Request page, specify how many of the most recent request you want to display.

   The default displays only the most recent request.

4. Specify a time period for the entries you want to view. You can select the most recent *period* or a range.
5. To filter the request to display, use the Search panel to query the log for a text pattern. For additional Search panels, click **Add**. The supported search criteria include:
   - The logged-in user name. This can be a specific user or `anonymous` for unauthenticated users.
   - The application name. This is useful when the application is running in a composite application and you want to view messages for a specific application.
   - A JSF page name without the file name extension. The log records the page name in the request header. If the submitted page is `orderSummary.jspx`, the logger records `orderSummary` in the request header. The request header also appears in the collapsible ADF Web Request panel header of the By ADF Request page.
6. To initiate the filters and display the ADF event messages for the ADF request, click **Search**.
7. In the ADF Web Request panel, expand the completed ADF request and look for ADF events that display long execution times as indicated by the Request Time bar graphs.

   If the page request has not completed, the yellow triangle symbols are displayed in the ADF Web Request panel. Wait a moment and then click the **Refresh** icon in the ADF Web Request panel.
8. Select the desired ADF event and examine the ADF Data panel for details about the ADF component associated with the ADF event.
9. Examine the component in your application and determine whether optimization is possible.

36.6.4.3 Sorting Diagnostic Messages By ADF Events

Oracle instrumented the Oracle ADF source code to generate log messages during the execution of the ADF lifecycle phases and during operations executed in the ADF Model data binding layer, ADF Controller source, and ADF Business Components source. Combined, the log analyzer refers to these messages as ADF events. You can use the log analyzer to investigate ADF events in your running application. The By Log Message page of the log analyzer lets you view ADF event messages in a flat list, organized by time of execution, with the option to switch to the By ADF Request page to view the ADF events in a hierarchical list, organized by the sequence of their execution.

ADF event messages contain useful information that helps you identify which ADF components in your application generated the event. For example, you can search the log for ADF event messages to identify the components related to displaying data in the page, executing queries, or initiating actions:

- Executing iterator binding: Displays the names of the iterators executed to manage displaying data in the page. This can be useful for diagnosing slow query updates.
- Execute query: Displays the name of the view object associated with the executed query. This can be useful when you want to view the query statement, bind parameters, and name of the view object.

- Executing method binding: Displays the names of the Java methods executed on the bound data source. This can be useful for diagnosing slow method execution.

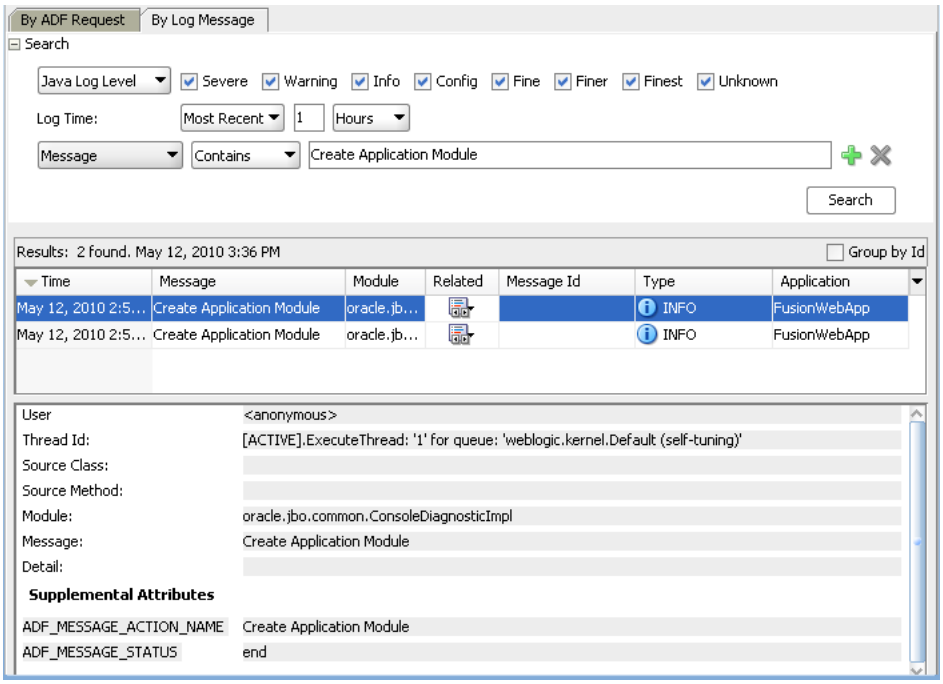
After you display an ADF event message in the log analyzer, you can organize the event in the context of other logged messages. You can select options from the **Related** column to display:

- All messages leading up to the ADF event (related by time)
- All messages in the same web request as the ADF event (related by request)
- Only ADF event messages in the same web request (related by ADF request)

> **Tip:** The **Related by ADF Request** option displays detailed ADF data for the ADF event messages. This is the view to use, for example, when you want to display the query statement associated with the `Execute query` message.

Figure 36–10 shows the log analyzer search result for the ADF event message `Create Application Module`. The Results panel displays all messages that match the search criteria and the bottom panel displays detailed information about the component.

Figure 36–10 Oracle Diagnostic Log Analyzer Displays ADF Event Messages



When you select **Related by ADF Request** in the **Related** column of the Results panel, the log analyzer switches to display the By ADF Request page with the ADF event messages arranged hierarchically to show their execution dependencies. The By ADF Request page of the log analyzer is the preferred way to diagnose performance issues. For details about the By ADF Request page, see Section 36.6.4.2, "Using the Log Analyzer to Analyze the ADF Request." In the By Log Message page, the elapsed time is information that you can leave visible or hide from the Results panel.

Before you begin:

It may be helpful to have an understanding of logging. For more information, see Section 36.6, "Using the ADF Logger."

You will need to complete these tasks:

1. Enable logging, as described in Section 36.6.2, "How to Turn On Diagnostic Logging."
2. Set logging levels, as described in Section 36.6.1, "How to Set ADF Logging Levels."

   To log ADF event messages, do not configure an ODL log level that is more restrictive than `Notification` or a Java log level that is more restrictive than `Info` for the following packages:

   - `oracle.adf` will log events generated by source code for ADF Model data binding layer and ADF Controller source code.
   - `oracle.jbo` will log events generated by source code executed for ADF Business Components.
   - `oracle.adfinternal` will log events generated by source code executed from Oracle ADF internal classes.

     Tip: The default log level for the Root Logger displayed by the editor for Oracle Diagnostics Logging Configuration ensures ADF event messages are logged.

3. Create a log file, either from your test run or from another source.

To display messages related by ADF events:

1. From the main menu, choose **Tools > Oracle Diagnostic Log Analyzer**.

   Or, from the Log window **Action** menu, choose **Analyze Log** and then either **Current in Console** or **Open Selected** (to browse log files in the server log directory).
2. In the editor for Oracle Diagnostic Log Analyzer, click the **By Log Message** tab.
3. In the By Log Message page, select the desired logger type, log levels, and log time.

   To search the log for ADF event messages, you must minimally select log level **Notification** (for ODL log level) or **Info** (for Java log level).
4. Choose the search criteria **Message** and **Contains**, and then enter any of the following ADF event messages and click **Search**.

   You can also filter the log on these additional ADF event messages:

   - Executing iterator binding - this can be useful for diagnosing slow query updates.
   - Executing method binding - this can be useful for diagnosing slow method execution.
   - Execute query - this can be useful when you want to view the query statement, bind parameters, and name of the view object.
   - Refreshing binding container
   - Attaching an iterator binding to a datasource

- Converting rows into hierarchical nodes
- Estimated row count
- Get LOV list
- Filter LOV list
- Validate Entity
- Lock Entity's Parent
- Lock Entity
- Before posting the entity's changes
- Posting the entity's changes
- Posting in batches
- Before committing the entity's changes
- After committing the entity's changes
- Before rolling back the entity's changes
- After rolling back the entity's changes
- Entity notifying an event
- Entity notification name
- Removing Entity
- Updating audit columns
- Applying Effective Date change
- Entity DML
- Entity read all attributes
- Create Application Module
- Create nested Application Module
- Passivating Application Module
- Activating Application Module
- Establish database connection
- Commit transaction
- Rollback transaction
- Validate transaction
- Validate value

Examine the bottom portion of the Results panel for the ADF event information.

5. To view a hierarchical sequence of ADF events, with the desired ADF event message selected in the Results panel, choose **Related - Related By ADF Request**.

   The editor for Oracle Diagnostic Log Analyzer displays the By ADF Request page for the selected ADF event. Examine the bottom portion of the Results panel for additional ADF data for the ADF event. For example, you can see the query statement associated with the `Execute query` message in the ADF Data area of the Results panel.

36.6.5 What You May Need to Know About the Logging.xml File

By default, the level is set to `INFO` for all packages of Oracle ADF. Set `level="FINE"` for detailed logging diagnostics.

For the ADF view layer packages `oracle.adf.view.faces` and `oracle.adfinternal.view.faces`, edit these elements:

```
<logger name="oracle.adf" level="FINE"/>
<logger name="oracle.adfinternal" level="FINE"/>
```

For the ADF Model layer packages, edit these elements:

```
<logger name="oracle.adf" level="FINE"/>
<logger name="oracle.jbo" level="FINE"/>
```

For the ADF Controller layer packages, edit these elements:

```
<logger name="oracle.adf.controller" level="FINE"/>
<logger name="oracle.adfinternal.controller" level="FINE"/>
```

Alternatively, you can create a debug configuration in JDeveloper that you can choose when you start a debugging session.

Example 36–2 shows the portion of the `logging.xml` file where you can change the granularity of the log messages. Note in the example that the log for `oracle.adf.faces` has been changed to `FINE` to display more messages.

Example 36–2 Sample Section of the logging.xml Configuration File

```
</logging_configuration>
...
  <loggers>
      <logger name="oracle.adf" level="INFO"/>
      <logger name="oracle.adf.faces" level="FINE"/>
      <logger name="oracle.adf.controller" level="INFO"/>
      <logger name="oracle.bc4j" level="INFO"/>
      <logger name="oracle.adf.portal" level="INFO"/>
      <logger name="oracle.vcr" level="INFO"/>
      <logger name="oracle.portlet" level="INFO"/>
      <logger name="oracle.adfinternal" level="INFO"/>
      <logger name="oracle.adfdt" level="INFO"/>
      <logger name="oracle.adfdtinternal" level="INFO"/>
   </loggers>
</logging_configuration>
```

For the latest information about the different levels of the Java Logging system, go to `http://www.oracle.com/technetwork/java/index.html`. Normally, the Java logging system supports the following levels:

- SEVERE
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST

36.6.6 What You May Need to Know About ADF Logging and Oracle WebLogic Server

After you have deployed the Fusion web application to Oracle WebLogic Server, the operations performed by the application are logged directly to the Managed Server where the application is running:

`DOMAIN_HOME/servers/`*`server_name`*`/logs/`*`server_name`*`-diagnostic.log`

The log files for the different Managed Servers are also available from the Oracle WebLogic Server Administration Console. To verify the logs, access the Oracle WebLogic Server Administration Console `http://<admin_server_host>:<port>/console` and click **Diagnostics-Log Files**.

This log's granularity and logging properties can be changed using Oracle Enterprise Manager Fusion Middleware Control (Fusion Middleware Control). Fusion Middleware Control is a web browser-based, graphical user interface that you can use to monitor and administer a farm.

When the Fusion web application is deployed to a high availability environment, you can receive warning diagnostic messages specific to high availability by setting the level to `FINE`.

For details about using Fusion Middleware Control to change the log settings of Managed Servers and Oracle ADF, see the *Oracle Application Server Administrator's Guide*.

36.7 Using the Oracle ADF Model Tester for Testing and Debugging

The Oracle ADF Model Tester (also referred to as the tester) is a Java application that you launch from JDeveloper when you want to interact with the business objects of the ADF Business Components data model project. The Oracle ADF Model Tester runs outside of JDeveloper and provides a full UI for testing and examining the data model project. You can run the tester to examine the view instances of the ADF application module, navigate the hierarchical relationship of view links, and execute custom methods from the application module's client interface, view object interface, and view row interface. The tester also interacts with the ADF Declarative Debugger to allow you to set breakpoints on the custom methods of these interfaces.

Additionally, the tester simulates many features that the user interface might expose by allowing you to view, insert, and update the contents of business objects in the database specified by the application module's configuration file (`bc4j.xcfg`). Specifically, you can use the tester to verify many aspects of the data model design, including master-detail relationships between view instance, view instances and their attributes, view instance query result sets, search forms using view criteria, validation rules defined for attribute values, and dropdown lists on LOV-defined attributes (list of values). For more information about ways to interact with the tester to test your business objects, see Section 6.3, "Testing View Object Instances Using the Oracle ADF Model Tester." Additional information about testing with the Oracle ADF Model Tester also appears in sections specific to each business object throughout the chapters in the "Building Your Business Services" part of this book.

36.7.1 How to Run in Debug Mode and Test with the Oracle ADF Model Tester

Often you will find it useful to analyze and debug custom code in the service methods of your client interface implementation classes. When you use the Oracle ADF Model Tester, you can do this without needing to run the application with the user interface. You can use the Oracle ADF Model Tester as a testing tool to complement your debugging process.

Before you begin:

Set breakpoints in the custom methods of your client interface, as described in Section 36.9, "Setting ADF Declarative Breakpoints."

To launch the Oracle ADF Model Tester and go into debug mode:

1. In the Application Navigator, right-click the desired application module and choose **Debug**.
2. In the Oracle ADF Model Tester, open the method testing panel for the desired client interface, as described in Section 9.9.4, "How to Test Custom Service Methods Using the Oracle ADF Model Tester."
3. In the method panel, select the desired method from the dropdown list, enter values to pass as method parameters, and click **Execute**.

   Return to JDeveloper to step through your code using the ADF Declarative Debugger. When you complete method execution, the method panel displays the return value (if any) and test result. The result displayed in the Oracle ADF Model Tester will indicate whether or not the method executed successfully.

36.7.2 How to Run the Oracle ADF Model Tester and Test with a Specific Configuration

When you right-click the application module in the Application Navigator and choose **Run** or **Debug**, JDeveloper will run the Oracle ADF Model Tester using the default configuration defined for the application module. If you want to test your business components with a different application module configuration (which can specify a different data source and its own set of runtime parameters), you can do so from the **Configurations** page in the overview editor for the application module.

To run the Oracle ADF Model Tester with a specific configuration:

1. In the Application Navigator, double-click the application module that you want to test.
2. In the overview editor, click the **Configurations** navigation tab, and select the configuration from the list.

   Although the metadata changes that you make are not involved in compiling the project, the compile step is necessary to copy the metadata to the class path and to allow the Oracle ADF Model Tester to reload it.
3. Right-click the selected configuration and choose **Run** or **Debug** to launch the Oracle ADF Model Tester.

36.7.3 What Happens When You Run the Oracle ADF Model Tester in Debug Mode

JDeveloper lets you run the Oracle ADF Model Tester in two modes: either in debug mode or non-debug mode. When run in debug mode, the tester interacts with the ADF Declarative Debugger so that you execute custom methods using breakpoints you insert in custom code of the client interfaces. For instance, if you set a breakpoint on a method in the client interface and execute that method in the tester, then in debug mode, you can step through the code before the tester returns a success/fail result. In non-debug mode, the tester will immediately return a result to indicate whether the method executed successfully. Additionally, in either debug or non-debug mode, the tester can display runtime artifacts from the system catalog created at runtime for the application module.

36.7.4 How to Verify Runtime Artifacts in the Oracle ADF Model Tester

When you want to run the Oracle ADF Model Tester, but do not require the use of the ADF Declarative Debugger you can display information about the runtime artifacts from the application module's system catalog. The system catalog displays business object metadata and other information that you may find useful when you need to compare business objects.

To launch the Oracle ADF Model Tester without debugging:

1. In the Application Navigator, right-click the desired application module and choose **Run**.
2. In the Oracle ADF Model Tester, choose **Create - Create SysCat AM**.
3. In the data model tree, expand the **SysCatAMDefs**, right-click **ViewDefs**, and choose **Show Table**.
4. In the data viewer, scroll vertically to locate the desired view instance in the **SCName** (system catalog name) field.

   Exposing the system catalog in the tester allows access to metadata and other information specific to the runtime objects without running the debugger. For example, you can check whether a view instance has a custom Java implementation class or not.

36.7.5 How to Refresh the Oracle ADF Model Tester with Application Changes

The Oracle ADF Model Tester is a highly interactive tool. When you run the tester and determine a change is needed in the data model project, you can return to JDeveloper to edit the desired application module instances and refresh the Oracle ADF Model Tester data model to display the changes. This way you can verify your changes without needing to rerun the tester.

To reload application metadata in the Oracle ADF Model Tester:

1. In the data model project, edit your business objects and save the changes in the JDeveloper.
2. In the Application Navigator, recompile the data model project. For example, you can right-click the data model project in the Application Navigator and choose **Make** to complete the recompile step.

   Although the metadata changes that you make are not involved in compiling the project, the compile step is necessary to copy the metadata to the class path and to allow the Oracle ADF Model Tester to reload it.
3. In the Oracle ADF Model Tester, in the toolbar, click **Reload application metadata**.

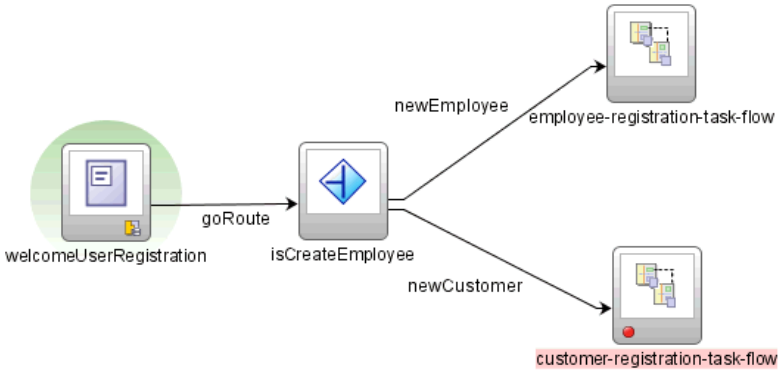
   Alternatively, you can choose **Reload Application** from the **File** menu of the Oracle ADF Model Tester.

36.8 Using the ADF Declarative Debugger

The ADF Declarative Debugger provides declarative breakpoints that you can set at the ADF object level (such as task flows, page definition executables, method and action bindings, ADF lifecycle phases), as well as standard Java breakpoints. ADF declarative breakpoints provide a high-level object view for debugging ADF applications. For example, you can break before a task flow activity to see what parameters would be passed to the task flow, as shown in Figure 36–11. To perform the

same function using only Java breakpoints would require you to know which class or method to place the breakpoint in. ADF declarative breakpoints should be the first choice for ADF applications.

Figure 36–11 ADF Declarative Breakpoint on a Task Flow Activity



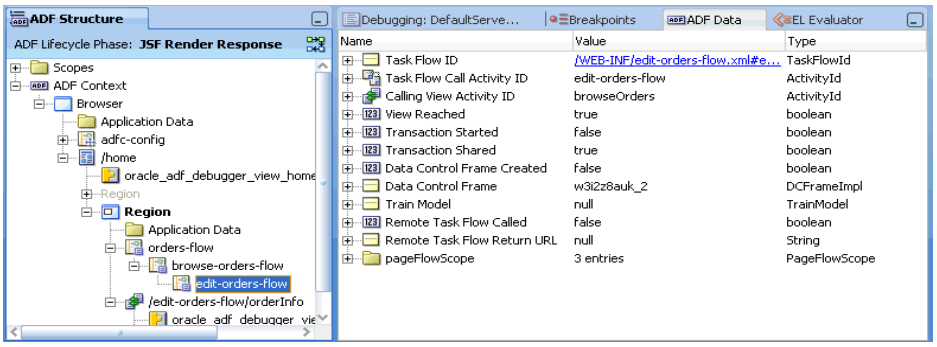
The ADF Declarative Debugger also supports standard Java code breakpoints. You can set Java code breakpoints in any ADF application. You may be able to use Java code breakpoints when an ADF declarative breakpoint does not break in the place you want.

The ADF Declarative Debugger is built on top of the Java debugger, so it has the features and behaviors of the Java debugger. But instead of needing to know the Java class or method, you can set ADF declarative breakpoints in visual editors.

The ADF Declarative Debugger provides standard debugging features such as the ability to examine variable and stack data. When an application pauses at any breakpoint (ADF Declarative or Java code breakpoint), you can examine the application status using a variety of windows. You can check where the break occurs in the Breakpoints window. You can check the call stack for the current thread using the Stack window. When you select a line in the Stack window, information in the Data window, Watches window, and all Inspector windows is updated to show relevant data. You can use the Data window to display information about arguments, local variables, and static fields in your application.

The ADF Structure window displays the runtime structure of the project. The ADF Data window automatically changes its display information based on the selection in the ADF Structure window. For example, if a task flow node is selected, the ADF Data window displays the relevant debugging information for task flows, as shown in Figure 36–12.

Figure 36–12 ADF Structure Window and ADF Data Window for a Task Flow Selection



You can mix ADF declarative breakpoints with Java code breakpoints as needed in your debugging session. Although you can use step functions to advance the application from Java code breakpoint to Java code breakpoint, the step functions for ADF declarative breakpoints have more constraints and limitations. For more information about using step functions on ADF declarative breakpoints, see Table 36–3.

For information on how to use ADF declarative breakpoints, see Section 36.9, "Setting ADF Declarative Breakpoints."

For information on how to use Java breakpoints on classes and methods, see Section 36.10, "Setting Java Code Breakpoints."

In a JSF application (including Fusion web applications), when a breakpoint breaks, you can use the EL Evaluator to examine the value of an EL expression. The EL Evaluator has the browse function that helps you select the correct expression to evaluate. For more information, see Section 36.8.4, "How to Use the EL Expression Evaluator."

Whether you plan to use ADF declarative breakpoints or Java breakpoints, you can use the ADF Declarative Debugger with Oracle ADF source code. You can obtain Oracle ADF source code with Debug libraries. For more information about loading source code. see Section 36.8.1, "Using ADF Source Code with the Debugger."

36.8.1 Using ADF Source Code with the Debugger

If you have valid Oracle ADF support, you can obtain complete source code for Oracle ADF by opening a service request with Oracle Worldwide Support. You can request a specific version of the Oracle ADF source code. You may be given download and password information to decrypt the source code ZIP file. Contact Oracle Worldwide Support for more information.

Adding Oracle ADF source code access to your application debugging session will:

- Provide access to the JDeveloper Quick Javadoc feature in the source editor. Without the source code, you will have only standard Javadoc.
- Enhance the use of Java code breakpoints by displaying the Oracle source code that's being executed when the breakpoint is encountered. You can also set breakpoints easier by clicking on the margin in the source code line you want to break on. Without the source code, you will have to know the class, method, or line number in order to set a breakpoint within Oracle code.
- For Java code breakpoints set within the source code, you will be able to see the values of all local variables and member fields in the debugger.

The ADF source code ZIP file may be delivered within an encrypted "outer" ZIP file to protect its contents during delivery. The "outer" ZIP name is sometimes a variant of the service request number.

After you have received or downloaded the "outer" ZIP, unzip it with the provided password to access the actual source code ZIP file. The ADF source code ZIP name should be a variant of the ADF version number and build number. For example, the ADF source ZIP may have a format similar to `adf_vvvv_nnnn_source.zip`, where *vvvv* is the version number and *nnnn* is the build number.

After you have access to the source code ZIP, extract its contents to a working directory.

36.8.2 How to Set Up the ADF Source User Library

You create a name for the source user library and then associate that name with the source zip file.

To add the ADF source zip file to the user library

1. From the main menu, choose **Tools > Manage Libraries**.
2. In the Manage Libraries dialog, with the **Libraries** tab selected, click **New**.
3. In the Create Library window, enter a library name for the source that identifies the type of library.
4. Select the **Source Path** node in the tree structure. Click **Add Entry**.

   Note: Do not enter a value for the class path. You need to provide a value only for the source path.

5. In the Select Path Entry window, browse to the directory where the file was extracted and select the source zip file. Click **Select**.
6. In the Create Library window, verify that the source path entry has the correct path to the source zip file, and deselect **Deployed by Default**. Click **OK**.
7. Click **OK**.

36.8.3 How to Add the ADF Source Library to a Project

After the source library has been added to the list of available user libraries, add it to the project you want to debug.

To add the ADF source zip file to the project:

1. In the Application Navigator, double-click the project or right-click the project and select **Project Properties**.
2. In the Project Properties dialog, select **Libraries and Classpaths**.
3. Click **Add Library**.
4. In the Add Library dialog, under the **Users** node, select the source library you want to add and click **OK**.

   The source library should appear in the **Classpath Entries** section in the Project Properties dialog.
5. Click **OK**.

36.8.4 How to Use the EL Expression Evaluator

When the application is paused at a breakpoint, you can use the EL expression evaluator to enter an EL expression for evaluation. You can enter arbitrary EL expressions for evaluation within the current context. If the EL expression no longer applies within the current context, the value will be evaluated to `null`.

The EL Evaluator is different from the Watches window in that EL evaluation occurs only when stopped at a breakpoint, not when stopped at subsequent debugging steps.

The EL Evaluator is available for debugging any JSF application.

> **Caution:** Be wary when you are evaluating EL expressions that you do not indirectly change application data and therefore the behavior of the application. For example, if you evaluate `#{foo.bar}`, the corresponding `getBar()` method modifies application data.

To use the EL Evaluator:

1. Set a breakpoint in the application.

   The application must be a JSF application. It does not need to be an ADF application.

2. Start the debugging process.

   You can:

   - From the main menu, choose **Run > Debug**.
   - From the Application Navigator, right-click the project, `adfc-config.xml`, `faces-config.xml`, task flow, or page and choose **Debug**.
   - From the task flow diagrammer, right-click an activity and choose **Debug**. Only task flows that do not use page fragments can be run.

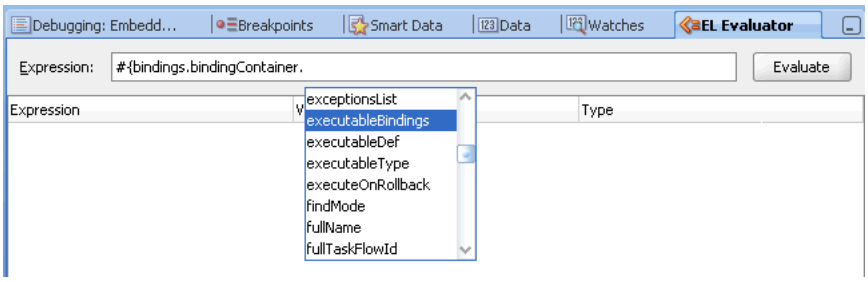
3. When the breakpoint is reached, the EL Evaluator should appear as a tab in the debugger window area. Click the **EL Evaluator** tab to bring it forward. If it does not appear, choose **View > Debugger > EL Evaluator** from the main menu.

   > **Note:** Be sure that the application has actually hit a breakpoint by checking the Breakpoints window or checking that there is an **Execution Point** icon (red right arrow) next to the breakpoint. Depending on where you set the breakpoint, an application may appear to be stopped when in fact it is waiting for user input at the page.

4. Enter an EL expression in the input field.

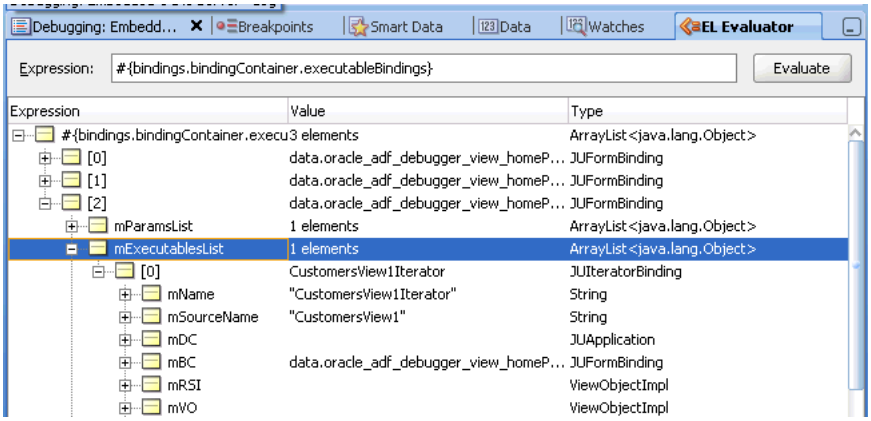
   When you click in the field after entering `#{` or after a period, a discovery function provides a selectable list of expression items, as shown in Figure 36–13. Auto-completion will be provided for easy entry. You can evaluate several EL expressions at the same time by separating them with semicolons.

Figure 36–13 Using the Discovery Function of the EL Evaluator



5. When you finish entering the EL expression, click **Evaluate** and the expression is evaluated, as shown in Figure 36–14.

Figure 36–14 EL Expression Evaluated



36.8.5 How to View and Export Stack Trace Information

If you are unable to determine what the problem is and to resolve it yourself, typically your next step is to ask someone else for assistance. Whether you post a question in the OTN JDeveloper Discussion Forum or open a service request on Metalink, including the stack trace information in your posting is extremely useful to anyone who will need to assist you further to understand exactly where the problem is occurring.

JDeveloper's Stack window makes communicating this information easy. Whenever the debugger is paused, you can view the Stack window to see the program flow as a stack of method calls that got you to the current line. Right-click the Stack window background and choose **Preferences**. You can set the Stack window preference to include the line number information, as well as the class and method name that will be there by default. Finally, the context menu option **Export** lets you save the current stack information to an external text file whose contents you can then post or send to whomever might need to help you diagnose the problem.

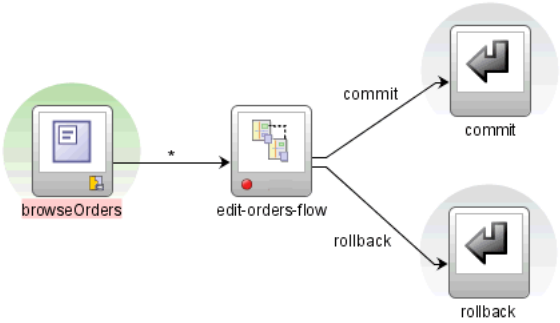
36.9 Setting ADF Declarative Breakpoints

You use the ADF Declarative Debugger features in JDeveloper to declaratively set breakpoints on ADF task flow activities, page definition executables, method, action, and value bindings, ADF Lifecycle phases, and contextual events. Instead of needing to know all the internal constructs of the ADF code, such as method names and class names, you can set breakpoints at the highest level of object abstraction.

You can add breakpoints to task flow activities in the task flow diagrammer or you can launch the Create ADF Task Flow Activity Breakpoint dialog from the Breakpoints window. In the task flow diagrammer, you can select a task flow activity and use the context menu to toggle or disable breakpoints on that activity, or press the F5 button. After the application pauses at the breakpoint, you can view the runtime structure of the objects in the ADF Structure window as a tree structure. The ADF Data window displays a list of data for a given object selected in the ADF Structure window.

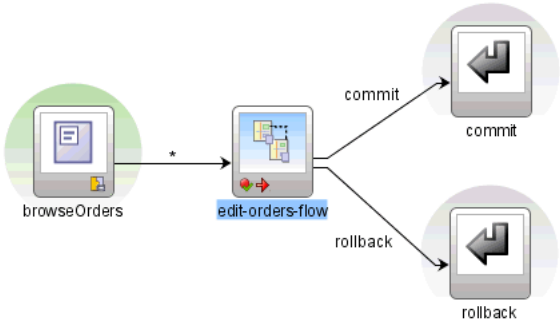
For example, when you set a breakpoint on a task flow call activity in the Browse Orders task flow, a red dot icon appears in the call activity, as shown in Figure 36–15.

Figure 36–15 ADF Declarative Breakpoint on a Task Flow Activity



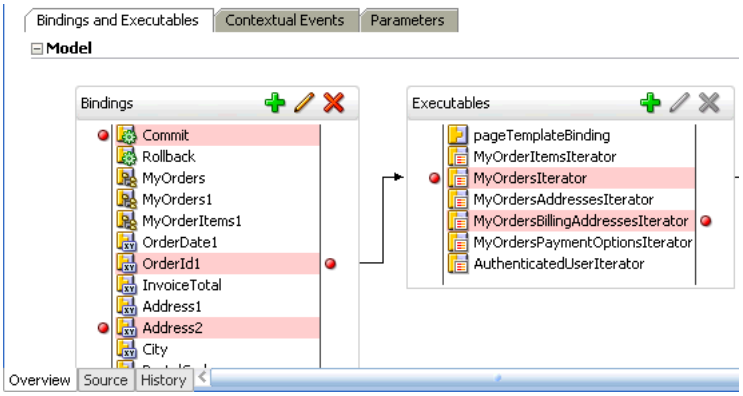
When the breakpoint is reached, the application is paused and the icon changes, as shown in Figure 36–16.

Figure 36–16 Application Paused at an ADF Declarative Breakpoint



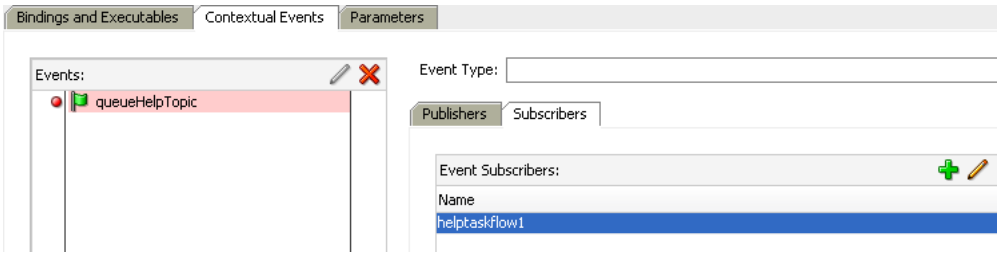
Similarly, you can set Before and After breakpoints in the page definition file. You set breakpoints for supported value bindings (see Table 36–1 for the list of supported value binding) and for executables by clicking on the left or right margin next to the item or by selecting from the context menu. Clicking on the left margin adds a **Before page definition** breakpoint, and clicking on the right margin adds an **After page definition** breakpoint. Again, a red dot icon that indicates the breakpoint is set, as shown in Figure 36–17.

Figure 36–17 ADF Declarative Breakpoints on ADF Bindings in the Page Definition File



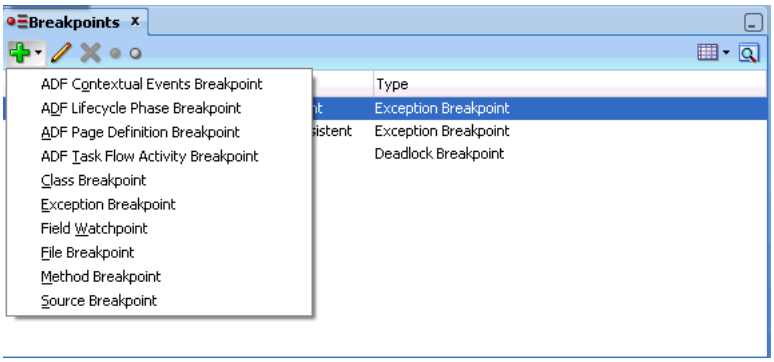
The page definition file also lets you set breakpoints on contextual events that your page or region within a page raises at runtime, as shown in Figure 36–18.

Figure 36–18 ADF Declarative Breakpoint on a Contextual Event in the Page Definition File



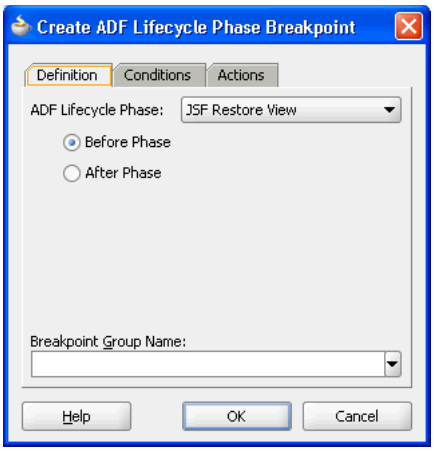
You can also set Before and After breakpoints on all the ADF lifecycle phases. You can launch the Create ADF Lifecycle Phase Breakpoint dialog from the Breakpoints window, as shown in Figure 36–19.

Figure 36–19 Breakpoints Window Add Breakpoint Icon Dropdown Menu



The Create ADF Lifecycle Phase Breakpoint dialog allows you to select different lifecycle breakpoint options, as shown in Figure 36–20.

Figure 36–20 Create ADF Lifecycle Phase Breakpoint Dialog



Alternatively, you can use the ADF Lifecycle Breakpoints dialog from the ADF Structure window or the task flow diagrammer to set ADF lifecycle phase breakpoints. For more information about ADF lifecycle phases, see Chapter 25, "Understanding the Fusion Page Lifecycle."

You can define both ADF declarative breakpoints and standard Java code breakpoints when using the ADF Declarative Debugger. Depending on your debugging scenario, you may only need to use the declarative breakpoints to debug the application. Or you may find it necessary to add additional breakpoints in Java code that are not available declaratively. For information on Java code breakpoints, see Section 36.10.1, "How to Set Java Breakpoints on Classes and Methods." Table 36–1 lists the available ADF Declarative Debugger breakpoint locations.

Table 36–1 ADF Declarative Debugger Breakpoints

ADF Area	Declarative Breakpoint	JDeveloper Editor	JDeveloper Location	JDeveloper Context Menu Command	Description
ADF lifecycle phase	Before ADF lifecycle phase After ADF lifecycle phase	ADF Structure window Task flow diagrammer Breakpoints window	ADF Structure window **ADF Lifecycle Breakpoints** toolbar button Task flow diagrammer Breakpoints window **Add** button		A Before breakpoint pauses debugging before the ADF lifecycle phase. An After breakpoint pauses debugging after the ADF lifecycle phase. The ADF lifecycle JSF Render Response and Prepare Render phase Before and After breakpoints are executed in the following order: ▪ Before `jsfRenderResponse`. ▪ Before `prepareRender`. (`prepareRender` phase executes). ▪ After `prepareRender`. (`jsfRenderResponse` phase executes). ▪ After `jsfRenderResponse`.

Table 36–1 (Cont.) ADF Declarative Debugger Breakpoints

ADF Area	Declarative Breakpoint	JDeveloper Editor	JDeveloper Location	JDeveloper Context Menu Command	Description
ADF page definition - bindings and executables	Before/After executable: ■ Iterator ■ `invokeAction` ■ Region instantiation	Overview editor for page definition files	Page Definition **Bindings and Executables** tab, Executables section Breakpoints window **Add** button	Toggle Breakpoint or F5 Disable Breakpoint	Pauses debugging before or after executable is refreshed. For task flow bindings, this represents two times per lifecycle: first, during `prepareModel` (initial region creation), and then again during `prepareRender` (where dynamic regions swap their corresponding task flow ID).
	Before/After action binding: ■ `methodAction` ■ Built-in operations		Page Definition **Bindings and Executables** tab, Bindings section Breakpoints window **Add** button	Toggle Breakpoint or F5 Disable Breakpoint	Pauses debugging before or after binding is executed.
	Before/After attribute value binding		Page Definition **Bindings and Executables** tab, Bindings section Breakpoints window **Add** button	Toggle Breakpoint or F5 Disable Breakpoint	Pauses debugging before or after the attribute's `setInputValue()` ADF source code method is executed. New values will be the parameters to `setInputValue()`.
	Before/After table binding		Page Definition **Bindings and Executables** tab, Bindings section Breakpoints window **Add** button	Toggle Breakpoint or F5 Disable Breakpoint	Pauses debugging before or after the ADF hierarchical binding's `updateValuesFromRows()` source code method is executed. New values will be the parameters to `updateValuesFromRows()`.
	Before/After tree binding		Page Definition **Bindings and Executables** tab, Bindings section Breakpoints window **Add** button	Toggle Breakpoint or F5 Disable Breakpoint	Pauses in either of these two cases: 1. When making a node selection in the tree, pauses debugging before or after the ADF hierarchical binding's `updateValuesFromRows()` source code method is executed. New values will be the parameters to `updateValuesFromRows()`. 2. When expanding the tree, pauses debugging before or after the ADF tree collection model's `modifyExpanded()` source code method is executed.

Table 36–1 (Cont.) ADF Declarative Debugger Breakpoints

ADF Area	Declarative Breakpoint	JDeveloper Editor	JDeveloper Location	JDeveloper Context Menu Command	Description
ADF page definition - contextual events	Before/After contextual events	Page definition overview editor	Page Definition **Contextual Events** tab, Events section Breakpoints window **Add** button	Toggle Breakpoint or F5 Disable Breakpoint	Pauses debugging either before the event is dispatched or after the event, just before the event is consumed by a subscriber to the event.
ADF task flow	Before activity	Task flow diagrammer	Task flow diagrammer Breakpoints window **Add** button	Toggle Breakpoint or F5 Disable Breakpoint	Pauses debugging before the activity executes within the JSF Invoke Application phase. The activity where the declarative breakpoint is defined has not yet been executed. An exception are view activities; they pause within the JSF Render Response phase after the view activity is executed, but before the new page is rendered. By pausing at that point, the view activity values can be inspected using the ADF Structure and ADF Data windows.

The ADF Declarative Debugger uses the standard debugger icons and notations for setting, toggling, and indicating the status of ADF declarative breakpoints.

When an ADF declarative breakpoint is set, it appears as a red dot icon in the task flow activity, in the page definition breakpoint margins, or in the ADF Lifecycle Breakpoints window, as shown in Figure 36–21, Figure 36–22, Figure 36–23, and Figure 36–24.

Figure 36–21 ADF Declarative Breakpoint Enabled on a Task Flow Activity

Figure 36–22 ADF Declarative Breakpoints Enabled in the Page Definition Executables

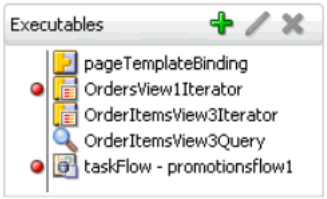


Figure 36–23 ADF Declarative Breakpoint Enabled on a Contextual Event in the Page Definition File

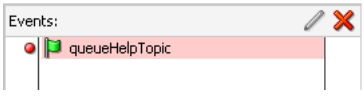
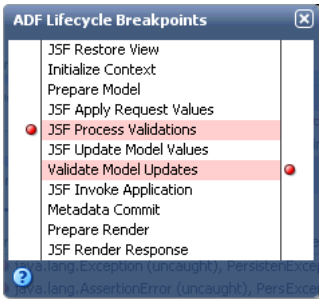


Figure 36–24 ADF Lifecycle Phase Breakpoints Enabled in the ADF Lifecycle Breakpoints Window

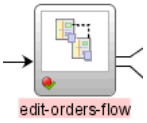


When an ADF task flow or page definition declarative breakpoint is disabled, the red icon becomes a gray icon, as shown in Figure 36–25.

Figure 36–25 ADF Declarative Breakpoint Disabled

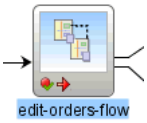
When an ADF task flow declarative breakpoint is active, the red dot icon has a green checkmark, as shown in Figure 36–26.

Figure 36–26 ADF Declarative Breakpoint Active



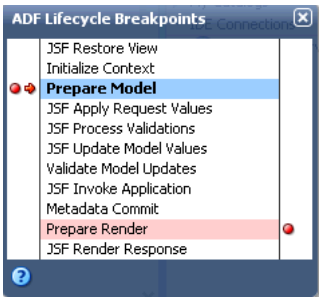
When the application is paused at an ADF declarative breakpoint, an **Execution Point** icon appears, as shown in Figure 36–27.

Figure 36–27 Application Paused at an Execution Point on a Task Flow



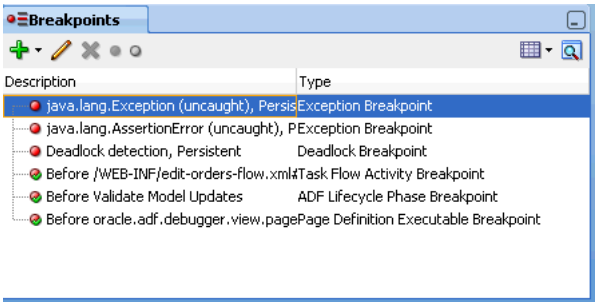
When the application is paused at an ADF lifecycle declarative breakpoint, an **Execution Point** icon appears next to the lifecycle phase in the ADF Lifecycle Breakpoints window, as shown in Figure 36–28. The name of the current ADF lifecycle phase is also displayed in the ADF Structure window.

Figure 36–28 Application Paused at an Execution Point on an ADF Lifecycle Phase



The Breakpoints window list all breakpoints, including ADF declarative breakpoints, as shown in Figure 36–29.

Figure 36–29 Breakpoints Window Showing ADF Declarative and Java Code Breakpoints



The Breakpoints window has a toolbar that includes buttons to add, edit, delete, enable, and disable breakpoints, as shown in Figure 36–29. The **Add Breakpoint** icon dropdown menu includes functions to create and manage ADF contextual events breakpoints, ADF lifecycle phase breakpoints, ADF page definition breakpoints (for ADF bindings and executables), ADF task flow activity breakpoints, and standard Java code breakpoints.

You can use the Breakpoints window to view the location of the ADF declarative breakpoint in its corresponding source file:

- Double-click a contextual event breakpoint to open the overview editor for the corresponding page definition file. You can then click the Contextual Events tab to view the location of the breakpoint.
- Double-click an ADF lifecycle phase breakpoint to open the ADF Lifecycle Breakpoints window which displays all ADF lifecycle execution points.
- Double-click a task flow activity breakpoint to open the task flow diagrammer for the corresponding task flow.
- Double-click an ADF binding breakpoint to open the overview editor for the corresponding page definition file.

To manage how the debugger handles a breakpoint, you can open the Edit ADF Breakpoint dialog for individual breakpoints that appear in the Breakpoints window. Or, you can select multiple ADF declarative breakpoints and customize the behavior of their common fields.

Table 36–2 lists how an ADF declarative breakpoint will appear in the Breakpoints window under the Description and Type columns.

Table 36–2 Breakpoints Window Display of ADF Declarative Breakpoints

Declarative Breakpoint Type	Description Column	Type Column
Before/After contextual event	Before *page definition@event name* After *page definition@event name*	Contextual events breakpoint
Before ADF lifecycle phase	Before *adf lifecycle phase*	ADF lifecycle phase breakpoint
After ADF lifecycle phase	After *adf lifecycle phase*	ADF lifecycle phase breakpoint
Before/After page definition executable: ■ Iterator ■ `invokeAction`	Before *page definition@executable id* After *page definition@executable id*	Page definition executable breakpoint
Before/After page definition action binding: ■ `methodAction` ■ Built-in Operations	Before *page definition@binding id* After *page definition@binding id*	Page definition binding breakpoint
Before/After page definition attribute value binding	Before *page definition@binding id* After *page definition@binding id*	Page definition binding breakpoint
Before/After page definition table binding	Before *page definition@binding id* After *page definition@binding id*	Page definition binding breakpoint
Before/After page definition tree binding	Before *page definition@binding id* After *page definition@binding id*	Page definition binding breakpoint
Before ADF task flow activity	Before *task flow document#task flow id@activity id*	Task flow activity breakpoint

Table 36–3 lists the step commands that can be used with ADF declarative breakpoints.

Table 36–3 ADF Declarative Debugger Step Commands

ADF Debugger Step Commands	Description
Find Execution Point	Supported for declarative breakpoints to display the current execution point open and active within the corresponding editor.
Step Over (F8)	Supported for task flow activity declarative breakpoints to step from activity to activity within a task flow. If user interaction is required (for example, page displayed), once it is received (for example, button selected), processing will resume and then will pause before the next task flow activity. supported for page definition executable breakpoints. The application will step to the next page definition executable breakpoint. Supported for ADF lifecycle phase declarative breakpoints to step to the next Before or After ADF lifecycle phase location.
Step Into (F7)	Supported only for task flow activity declarative breakpoints defined on task flow call activities. Task flow activity declarative breakpoints pause the application just before the activity is executed. The Step Into function provides the ability to pause debugging just prior to executing the called task flow default activity. This action would be the same as placing a task flow activity declarative breakpoint on the called task flow default activity.

Table 36–3 (Cont.) ADF Declarative Debugger Step Commands

ADF Debugger Step Commands	Description
Step Out (Shift F7)	Supported for task flow activity declarative breakpoints to step out of the current called task flow and back into the caller (if any). If user interaction is required (for example, page displayed) once user interaction received (for example, button selected), processing will resume and will pause before the next user interaction or activity within the calling task flow.
Continue Step (Shift F8)	Not supported for declarative breakpoints.
Step to End of Method	Not supported for declarative breakpoints.
Run to Cursor	Not supported for declarative breakpoints.
Pop Frame	Not supported for declarative breakpoints, as it is for Java code, to return to a previous point of execution.

36.9.1 How to Set and Use Task Flow Activity Breakpoints

After you have created a task flow diagram, you can set ADF declarative breakpoints on task flow activities.

To set a breakpoint on a task flow activity:

1. Open the task flow in the task flow diagrammer, or from the Breakpoints window, click the **Add** icon and select **ADF Task Flow Activity Breakpoint**.
2. Set the task flow activity breakpoint.
   - If you use the task flow diagrammer, right-click and choose **Toggle Breakpoint** from the context menu, or press F5.

     A breakpoint icon appears on the task flow activity.
   - If you launched the Create ADF Task Flow Activity Breakpoint dialog from the Breakpoints window, click **Browse** to select a task flow definition, select the task flow from the **Task Flow** dropdown list, select the task flow activity from the **Activity** dropdown list, and click **OK**.
3. Optionally, configure a breakpoint's settings to manage the debugger:
   a. Choose **View > Breakpoints** to open the Breakpoints window.
   b. Select the task flow activity breakpoint you want to configure and click the **Edit** icon.
   c. In the Edit ADF Task Flow Activity Breakpoint dialog, click the **Conditions** tab, specify the conditions which apply to the breakpoint. The conditions must be valid for the breakpoint to occur.
   d. Click the **Actions** tab, specify the actions that you want the debugger to take when the breakpoint occurs and click **OK**.

      For example, the usual action for a breakpoint is to halt the program you are debugging, but you may want the debugger to beep and log information to the Log window without halting the program.
4. Start the debugging process.

   You can:
   - From the main menu, choose **Run > Debug**.

- From the Application Navigator, right-click the project, `adfc-config.xml`, `faces-config.xml`, task flow, or page and choose **Debug**.
- From the task flow diagrammer, right-click an activity and choose **Debug**. Only task flows that do not use page fragments can be run.

5. When the application is paused at a breakpoint, an **Execution Point** icon appears next to the breakpoint icon on the task flow activity. You can examine the application using different debugger windows.

   Note: Be sure that the application has actually hit a breakpoint by checking the Breakpoints window or checking that there is an **Execution Point** icon (red right arrow) next to the breakpoint. Depending on where you set the breakpoint, an application may appear to be stopped when in fact it is waiting for user input at the page.

   The application is paused before the task flow activity executes (except for view activities).

6. The ADF Structure window and the ADF Data window appear by default, as well as several debugger windows. You can examine the runtime structure in the ADF Structure window and its corresponding data in the ADF Data window. See Section 36.9.7, "How to Use the ADF Structure Window," and Section 36.9.8, "How to Use the ADF Data Window."

7. Select a node in the ADF Structure window and view pertinent information in the ADF Data window. Task flow activity declarative breakpoints pause the application just before the task flow activity is executed. You can use the Step Into (F7) function to pause the application just prior to executing the called task flow default activity.

8. Continue debugging the application as required, using the step functions as described in Table 36–3. The key step function is Step Into (F7).

   When the application is paused, you can remove or disable existing breakpoints and set new breakpoints.

36.9.2 How to Set and Use Page Definition Executable Breakpoints

If your page definition has executables, you can set breakpoints to pause the application before or after these executables. For example, you can set breakpoints to pause the application when iterators are refreshed or when `invokeAction` methods are performed.

Note: If you are setting an After iterator breakpoint to pause the application after a view object query has been executed, be aware that the application may pause at this breakpoint multiple times. Also be aware that it may pause at this breakpoint even when the query has not been executed. If you need to know whether the query has been executed, select the relevant ADF Business Components in the ADF Structure window and view their corresponding data in the ADF Data window. For more information on how to use these windows, see Section 36.9.7, "How to Use the ADF Structure Window," and Section 36.9.8, "How to Use the ADF Data Window."

For more information about using Java code breakpoints on view object query execution, see Section 36.10.8, "How to Use Common Oracle ADF Breakpoints."

To set a breakpoint on an executable in the page definition file:

1. In the Application Navigator, double-click the page definition file that contains the executable in which you want to set a breakpoint.
2. In the overview editor, click the **Bindings and Executables** tab, select an executable from the **Executables** list, and click in the breakpoint margin to the left of the item.

   A breakpoint icon appears in the margin next to the item.
3. Optionally, configure a breakpoint's settings to manage the debugger:
   a. Choose **View > Breakpoints** to open the Breakpoints window.
   b. Select the executable breakpoint you want to configure and click the **Edit** icon.
   c. In the Edit ADF Page Definition Binding Breakpoint dialog, click the **Conditions** tab, specify the conditions which apply to the breakpoint. The conditions must be valid for the breakpoint to occur.
   d. Click the **Actions** tab, specify the actions that you want the debugger to take when the breakpoint occurs and click **OK**.

      For example, the usual action for a breakpoint is to halt the program you are debugging, but you may want the debugger to beep and log information to the Log window without halting the program.
4. Start the debugging process.

   You can:

   - From the main menu, choose **Run > Debug**.
   - From the Application Navigator, right-click the project, `adfc-config.xml`, `faces-config.xml`, task flow, or page and choose **Debug**.
   - From the task flow diagrammer, right-click an activity and choose **Debug**. Only task flows that do not use page fragments can be run.
5. When the application is paused at a breakpoint, an **Execution Point** icon appears in the margin next to the breakpoint icon of the executable item. You can examine the application using several debugger windows.

   The application pauses when the executable binding is refreshed. If this is a taskFlow executable, the pause occurs in the `prepareModel` and the `prepareRender` lifecycles.

> **Note:** Be sure that the application has actually hit a breakpoint by checking the Breakpoints window or checking that there is an **Executable Point** icon (red right arrow) next to the breakpoint. Depending on where you set the breakpoint, an application may appear to be stopped when in fact it is waiting for user input at the page.

6. The ADF Structure window and the ADF Data window appear by default, as well as several debugger windows. You can examine the runtime structure in the ADF Structure window and its corresponding data in the ADF Data window. See Section 36.9.7, "How to Use the ADF Structure Window," and Section 36.9.8, "How to Use the ADF Data Window."
7. Select a node in the ADF Structure window and view pertinent information in the ADF Data window.
8. When the application is paused, you can remove or disable existing breakpoints and set new breakpoints.

36.9.3 How to Set and Use Page Definition Action Binding Breakpoints

You can set breakpoints in the page definition file on action bindings and `methodAction` bindings. The application pauses when the binding is executed.

To set a breakpoint on an action binding in the page definition file:

1. In the Application Navigator, double-click the page definition file that contains the binding in which you want to set a breakpoint.
2. In the overview editor, click the **Bindings and Executables** tab, select a `methodAction` binding or built-in operation item from the **Bindings** list, and click in the breakpoint margin to the left of the item.

   A breakpoint icon appears next to the item.
3. Optionally, configure a breakpoint's settings to manage the debugger:
   a. Choose **View > Breakpoints** to open the Breakpoints window.
   b. Select the action binding breakpoint you want to configure and click the **Edit** icon.
   c. In the Edit ADF Page Definition Binding Breakpoint dialog, click the **Conditions** tab, specify the conditions which apply to the breakpoint. The conditions must be valid for the breakpoint to occur.
   d. Click the **Actions** tab, specify the actions that you want the debugger to take when the breakpoint occurs and click **OK**.

      For example, the usual action for a breakpoint is to halt the program you are debugging, but you may want the debugger to beep and log information to the Log window without halting the program.
4. Start the debugging process.

   You can:

   - From the main menu, choose **Run > Debug**.
   - From the Application Navigator, right-click the project, `adfc-config.xml`, `faces-config.xml`, task flow, or page and choose **Debug**.

- From the task flow diagrammer, right-click an activity and choose **Debug**. Only task flows that do not use page fragments can be run.

5. When the application is paused at a breakpoint, an **Execution Point** icon appears next to the breakpoint icon on the action binding item. You can examine the application using several debugger windows.

   The application is paused when the binding is executed.

   Note: Be sure that the application has actually hit a breakpoint by checking the Breakpoints window or checking that there is an **Execution Point** icon (red right arrow) next to the breakpoint. Depending on where you set the breakpoint, an application may appear to be stopped when in fact it is waiting for user input at the page.

6. The ADF Structure window and the ADF Data window appear by default, as well as several debugger windows. You can examine the runtime structure in the ADF Structure window and its corresponding data in the ADF Data window. See Section 36.9.7, "How to Use the ADF Structure Window," and Section 36.9.8, "How to Use the ADF Data Window."
7. Select a node in the ADF Structure window and view pertinent information in the ADF Data window.
8. When the application is paused, you can remove or disable existing breakpoints and set new breakpoints.

36.9.4 How to Set and Use Page Definition Value Binding Breakpoints

If the page definition has one of these values bindings, you can set breakpoints to pause the application:

- Attribute value binding
- Tree value binding
- Table value binding

To set a breakpoint on a value binding in the page definition file:

1. In the Application Navigator, double-click the page definition file that contains the binding in which you want to set a breakpoint.
2. In the overview editor, click the **Bindings and Executables** tab, select an attribute, tree, or table binding from the **Bindings** list, and click on the breakpoint margin to the left of the item. A breakpoint icon appears next to the value binding.
3. Optionally, configure a breakpoint's settings to manage the debugger:
   a. Choose **View > Breakpoints** to open the Breakpoints window.
   b. Select the binding breakpoint you want to configure and click the **Edit** icon.
   c. In the Edit ADF Page Definition Binding Breakpoint dialog, click the **Conditions** tab, specify the conditions which apply to the breakpoint. The conditions must be valid for the breakpoint to occur.
   d. Click the **Actions** tab, specify the actions that you want the debugger to take when the breakpoint occurs and click **OK**.

For example, the usual action for a breakpoint is to halt the program you are debugging, but you may want the debugger to beep and log information to the Log window without halting the program.

4. Start the debugging process.

   You can:

   - From the main menu, choose **Run > Debug**.
   - From the Application Navigator, right-click the project, `adfc-config.xml`, `faces-config.xml`, task flow, or page and choose **Debug**.
   - From the task flow diagrammer, right-click an activity and choose **Debug**. Only task flows that do not use page fragments can be run.

5. When the application is paused at a breakpoint, an **Execution Point** icon appears next to the breakpoint icon on the attribute value binding. You can examine the application using several debugger windows.

   The application pauses before an appropriate method of the ADF source code, as described in Table 36–1. New values will be the parameters that go into the method.

   Note: Be sure that the application has actually hit a breakpoint by checking the Breakpoints window or checking that there is an **Execution Point** icon (red right arrow) next to the breakpoint. Depending on where you set the breakpoint, an application may appear to be stopped when in fact it is waiting for user input at the page.

6. The ADF Structure window and the ADF Data window appear by default, as well as several debugger windows. You can examine the runtime structure in the ADF Structure window and its corresponding data in the ADF Data window. See Section 36.9.7, "How to Use the ADF Structure Window," and Section 36.9.8, "How to Use the ADF Data Window."

7. Select a node in the ADF Structure window and view pertinent information in the ADF Data window.

8. Continue debugging the application as required, using the step functions as described in Table 36–3. The key step function is Step Over (F8).

   When the application is paused, you can remove or disable existing breakpoints and set new breakpoints.

36.9.5 How to Set and Use Page Definition Contextual Event Breakpoints

If the page definition defines contextual events, you can set breakpoints on the contextual events to pause the application.

To set a breakpoint on a contextual event in the page definition file:

1. In the Application Navigator, double-click the page definition file that contains the binding in which you want to set a breakpoint.
2. In the overview editor, click the **Contextual Events** tab, select a contextual event from the **Events** list, and click on the breakpoint margin to the left of the item. A breakpoint icon appears next to the contextual event.
3. Optionally, configure a breakpoint's settings to manage the debugger:

a. Choose **View > Breakpoints** to open the Breakpoints window.

b. Select the contextual event breakpoint you want to configure and click the **Edit** icon.

c. In the Edit ADF Contextual Events Breakpoint dialog, click the **Conditions** tab, specify the conditions which apply to the breakpoint. The conditions must be valid for the breakpoint to occur.

d. Click the **Actions** tab, specify the actions that you want the debugger to take when the breakpoint occurs and click **OK**.

   For example, the usual action for a breakpoint is to halt the program you are debugging, but you may want the debugger to beep and log information to the Log window without halting the program.

4. Start the debugging process.

   You can:

   - From the main menu, choose **Run > Debug**.
   - From the Application Navigator, right-click the project, `adfc-config.xml`, `faces-config.xml`, task flow, or page and choose **Debug**.
   - From the task flow diagrammer, right-click an activity and choose **Debug**. Only task flows that do not use page fragments can be run.

5. When the application is paused at a breakpoint, an **Execution Point** icon appears next to the breakpoint icon on the contextual event. You can examine the application using several debugger windows.

   The application pauses before the contextual event is raised.

   Note: Be sure that the application has actually hit a breakpoint by checking the Breakpoints window or checking that there is an **Execution Point** icon (red right arrow) next to the breakpoint. Depending on where you set the breakpoint, an application may appear to be stopped when in fact it is waiting for user input at the page.

6. The ADF Structure window and the ADF Data window appear by default, as well as several debugger windows. You can examine the runtime structure in the ADF Structure window and its corresponding data in the ADF Data window. See Section 36.9.7, "How to Use the ADF Structure Window," and Section 36.9.8, "How to Use the ADF Data Window."

7. Select a node in the ADF Structure window and view pertinent information in the ADF Data window.

8. Continue debugging the application as required, using the step functions as described in Table 36–3. The key step function is Step Over (F8).

   When the application is paused, you can remove or disable existing breakpoints and set new breakpoints.

36.9.6 How to Set and Use ADF Lifecycle Phase Breakpoints

You can set both Before and After ADF lifecycle phase breakpoints on any of the ADF lifecycle phases. For each phase, you can set Before only, After only, or both. You can set breakpoints on as many phases as you want.

You can create the breakpoint and customize the options using the Create ADF Lifecycle Phase Breakpoint dialog from the Breakpoints window menu. Or You can create breakpoints with the default options using the ADF Lifecycle Breakpoints window. After a lifecycle breakpoint has been set, you can edit the options using the Edit ADF Lifecycle Phase Breakpoint dialog, which is also launched from the Breakpoints window.

You can set ADF lifecycle breakpoints on any of the ADF lifecycle phases:

- JSF Restore View
- Initialize Content
- Prepare Model
- JSF Apply Request Values
- JSF Process Validations
- JSF Update Model Values
- Validate Model Updates
- JSF Invoke Application
- Metadata Commit
- Prepare Render
- JSF Render Response

To set or manage an ADF lifecycle phase breakpoint from the Breakpoints window:

1. Choose **View > Breakpoints** to open the Breakpoints window.
2. Click the **Add** icon and choose **ADF Lifecycle Phase Breakpoint**.
3. In the Create ADF Lifecycle Phase Breakpoint dialog **Definition** tab:
   - Select the ADF lifecycle phase where you want to set a breakpoint
   - Select **Before Phase** or **After Phase** breakpoint
4. In the **Conditions** tab, select the options you want and click **OK**.
5. In the **Actions** tab, select the options you want and click **OK**.

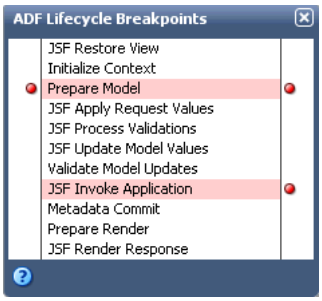
To set an ADF Lifecycle Phase Breakpoint using the breakpoint icon:

1. In the task flow diagrammer or in the ADF Structure window, click the **ADF Lifecycle Breakpoints** icon, as shown in Figure 36–30.

Figure 36–30 ADF Lifecycle Breakpoints Icon

2. In the ADF Lifecycle Breakpoints window, click on the left margin next to the ADF lifecycle phase to set a Before breakpoint, and on the right margin to set an After breakpoint. A red dot icon appears to indicate the breakpoint is set, as shown in Figure 36–31. The breakpoint will be set with the default breakpoint options. To remove the breakpoint, click the red dot icon.

Figure 36–31 Setting Breakpoints in the ADF Lifecycle Breakpoints Window



3. If you want to edit breakpoint options, select the breakpoint in the Breakpoints window and choose the **Edit** icon.

To debug an application using ADF Lifecycle Phase Breakpoints:

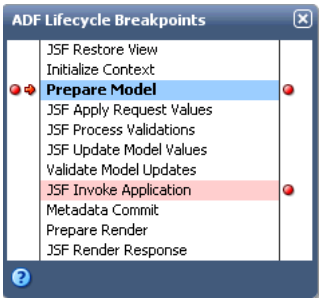
1. Start the debugging process.

   You can:

   - From the main menu, choose **Run > Debug**.
   - From the Application Navigator, right-click the project, `adfc-config.xml`, `faces-config.xml`, task flow, or page and choose **Debug**.
   - From the task flow diagrammer, right-click an activity and choose **Debug**. Only task flows that do not use page fragments can be run.

2. When the application is paused at an ADF lifecycle phase breakpoint, an **Execution Point** icon appears next to the breakpoint icon and the ADF lifecycle phase is in bold in the ADF Lifecycle Breakpoints window, as shown in Figure 36–32. You can examine the application using several debugger windows.

Figure 36–32 Execution Point Displayed in the ADF Lifecycle Breakpoints Window



> **Note:** Be sure that the application has actually hit a breakpoint by checking the Breakpoints window for an breakpoint encounter or checking that there is an **Execution Point** icon (red right arrow) next to the breakpoint. Depending on where you set the breakpoint, an application may appear to be stopped when in fact it is waiting for user input at the page.

3. The ADF Structure window and the ADF Data window appear by default, as well as several debugger windows. You can examine the runtime structure in the ADF Structure window and its corresponding data in the ADF Data window. The current ADF lifecycle phase is displayed at the top of the ADF Structure window.

For more information, see Section 36.9.7, "How to Use the ADF Structure Window," and Section 36.9.8, "How to Use the ADF Data Window."

4. Select a node in the ADF Structure window and view pertinent information in the ADF Data window.
5. Continue debugging the application as required, using the step functions as described in Table 36–3. The key step function is Step Over (F8).

   When the application is paused, you can remove or disable existing breakpoints and set new breakpoints.

36.9.7 How to Use the ADF Structure Window

When the application is paused at a breakpoint, the ADF Structure window displays a tree structure of the ADF runtime objects and their relationships within the application. In particular, it shows the hierarchy of view ports, which represent either the main browser window or contained regions and portlets. When you select different items in the ADF Structure window, the data display in the accompanying ADF Data window changes. For more information about the ADF Data window, see Section 36.9.8, "How to Use the ADF Data Window."

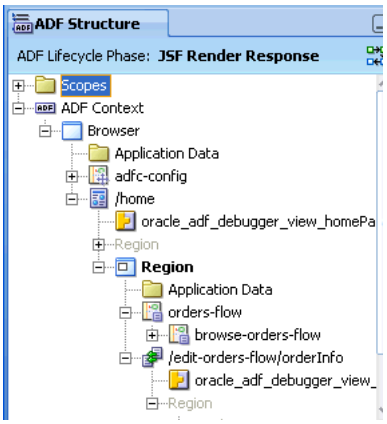
The ADF Structure window and the ADF Data window are shown by default during a debugging session when either of the following is true:

- The project being debugged contains a `WEB-INF/adfc-config.xml` file.
- The project being debugged contains any ADF Faces tag libraries.

You can launch the ADF Structure window by choosing **View > Debugger > ADF Structure** from the main menu. From the ADF Structure window, you can launch the ADF Lifecycle Breakpoints window using the **ADF Lifecycle Breakpoints** icon.

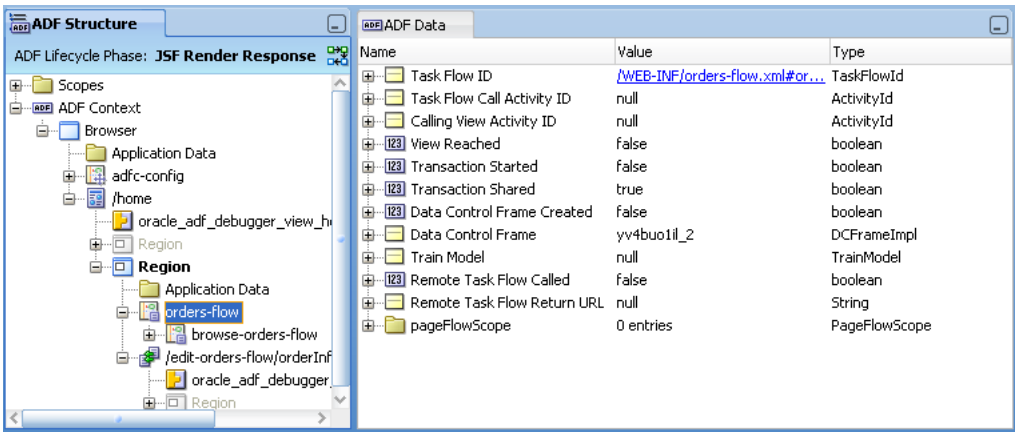
When a breakpoint is encountered, the ADF Structure window displays the ADF lifecycle phase and a tree structure of the runtime objects, as shown in Figure 36–33.

Figure 36–33 ADF Structure Window Showing the Runtime Objects



When you select an item in the ADF Structure window, the data and values associated with that item are displayed in the ADF Data window. Figure 36–34 shows a task flow selected in the ADF Structure window, with its corresponding information displayed in the ADF Data window.

Figure 36–34 ADF Structure Window Selection and ADF Data Window Data



The roots of the hierarchy are the sibling nodes **Scopes** and **ADF Context**. The current view port where processing has stopped appears in bold. Default selections within the tree will be retained from the previous breakpoint, so you can monitor any changes between breakpoints. The ADF object where the ADF declarative breakpoint was defined will be opened in the corresponding JDeveloper editor, either the task flow diagrammer or the overview editor for page definition files.

The ADF Structure tree will be rebuilt each time the application breaks and at subsequent steps to reflect the changed state of the objects. Although the entire tree hierarchy will be displayed, only items within the current view port and its parent view port(s) will be available for selection and further inspection. All other items in the tree hierarchy not in the current context will be dimmed and disabled. You can still use the hierarchy to identify runtime object relationships within the application, but it will be limited to the current context (and its parent view ports).

Table 36–4 lists the different types of items that can be displayed in the ADF Structure window hierarchy tree.

Table 36–4 ADF Structure Window Items

ADF Structure Tree Item	Description
Scopes	Displayed at the top of the ADF Structure hierarchy above its sibling **ADF Context** node. There is only one **Scopes** node in the ADF Structure hierarchy. You can expand the **Scopes** node to show a list of child scope nodes (such as **viewScope** and **pageFlowScope**). If you select a child scope node, the ADF Data window displays the variables and values for that scope.
ADF context	Displayed as the root node of the ADF Structure hierarchy below its sibling **Scopes** node. There will only be one **ADF Context** within the ADF Structure hierarchy.
View port	View ports are an ADF Controller concept. For this reason, view ports appear within the ADF Structure hierarchy only when the application being debugged utilizes ADF Controller. View ports can represent one of the following: ■ Browser: Main browser view ports, also known as root view ports, appear as children of the root **ADF Context**. If multiple browser windows are open during the debugging runtime session, multiple browser view ports are presented within the hierarchy. The label of each browser view port displays the text "Browser". The view port also provides a tooltip for the view port ID similar to the following example: "Root View Port: 999999". ■ Region: Region view ports appear as the children of page or page fragments. They are also known as child view ports. The label of each region view port displays the text "Region". The region also provides a tooltip for the view port ID similar to the following example: "Child View Port: 999999".
ADF task flows	The page flow stack corresponding to each view port appears as a hierarchy of ADF task flows. The initial ADF task flow called for the stack is a direct child of its corresponding view port. The label of each ADF task flow reflects the corresponding ADF task flow display name (if any) or its task flow ID. Region view ports will not display the item in their page flow stack hierarchy for their implied unbounded task flow. The task flow also provides a tooltip displaying the ADF task flow path, and a context menu item to open to the corresponding ADF task flow within the editor workspace. If ADF Controller is not utilized in the application (or if the page is run outside the context of an ADF task flow), ADF task flows will not appear within the hierarchy.
Page	Represents the page (view) currently displayed within a browser view port. Presented along with its associated binding container (if any) as a child. If the application being debugged utilizes ADF Controller, pages will be children of each browser view port. The label of each page reflects its corresponding runtime view ID. The page also provides a tooltip displaying the page path, and a context menu item to open to the corresponding page within the editor workspace. If a visual user interface in not implemented for the application, the page will not appear within the hierarchy.
Page fragment	Represents the page fragment currently displayed within a region view port. Presented along with its associated binding container (if any) as a child. If the application being debugged utilizes ADF Controller, page fragments will be children of each region view port. The label of each page fragment node reflects its corresponding runtime view ID. The page fragment also provides a tooltip displaying the source file page definition path, and a context menu item to open to the corresponding page fragment within the editor workspace.
Binding container	Represents the binding container for the corresponding page or page fragment. The label of each binding container reflects its corresponding file name (page definition file) without the extension. The binding container node will also provide a tooltip displaying the page fragment path. The binding container also appears under current task flows when used to represent task flow activity bindings (for example, method call activity bindings). If ADF Model is not utilized for the application, binding containers will not appear.
Application data	Represents the application data objects (for example, ADF Business Components objects or ADF Business Components business service objects) instantiated within the data control frame for the corresponding view port (or binding container if ADF Controller is not used). Application data objects don't need to be currently instantiated for the Application Data node to appear.

36.9.8 How to Use the ADF Data Window

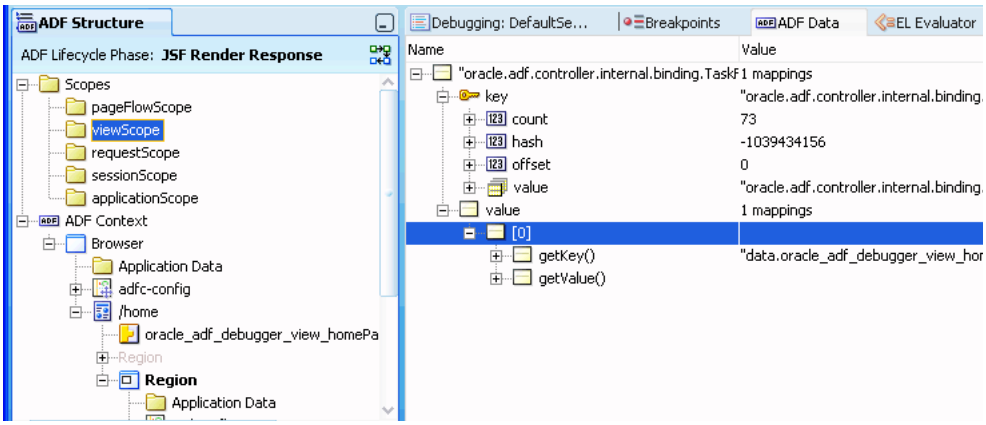
When an application is paused at an ADF declarative breakpoint, the ADF Data window displays relevant data based on the selection in the ADF Structure window. You can launch the ADF Data window by choosing **View > Debugger > ADF Data** from the main menu. The content of the ADF Data window based on the selection in the ADF Structure window is summarized in Table 36–5.

Table 36–5 ADF Data Window Content for an ADF Structure Window Selection

ADF Structure Window	ADF Data Content
Scopes	Displays memory scope values based on the current context. `pageFlowScope` will also appear within ADF task flow content for the `pageFlowScope` values specific to a selected ADF task flow (not necessarily the current context). `viewScope` will also appear within the view port content for the `viewScope` values specific to a selected view port (not necessarily the current context).
ADF context	Displays the ADF context variables and values hierarchy. ADF context variables and values can be inspected by evaluating the `#(data.adfContext)` EL expression in the EL Evaluator.
View port	Displays view port details, including the `viewScope` contents.
Page flow stack entry	Displays information for the selected page flow stack entry, including current transaction status and ADF Model save point status.
Page/page fragment	Displays the page or page fragment UI component tree hierarchy for the selected page or page fragment if the page or page fragment has been rendered.
Binding container	Displays the binding container runtime values, including parameters, bindings, and executables.
Application data	Displays application data objects (for example, ADF Business Components objects) instantiated within the current binding context. If the business service layer is implemented with a technology other than ADF Business Components objects (for example, EJB) the application data objects will be displayed in a more generic form.

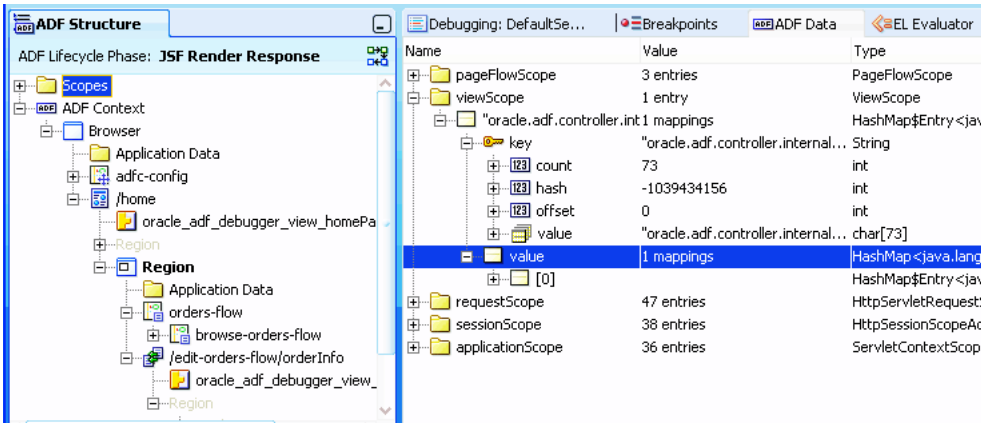
The **Scopes** node in the ADF Structure window can be expanded to show a list of child scope nodes. When a child scope node is selected in the ADF Structure window, the ADF Data window displays the current context values for the selected memory scope, as shown in Figure 36–35.

Figure 36–35 Child Scope Selected in the ADF Structure Window



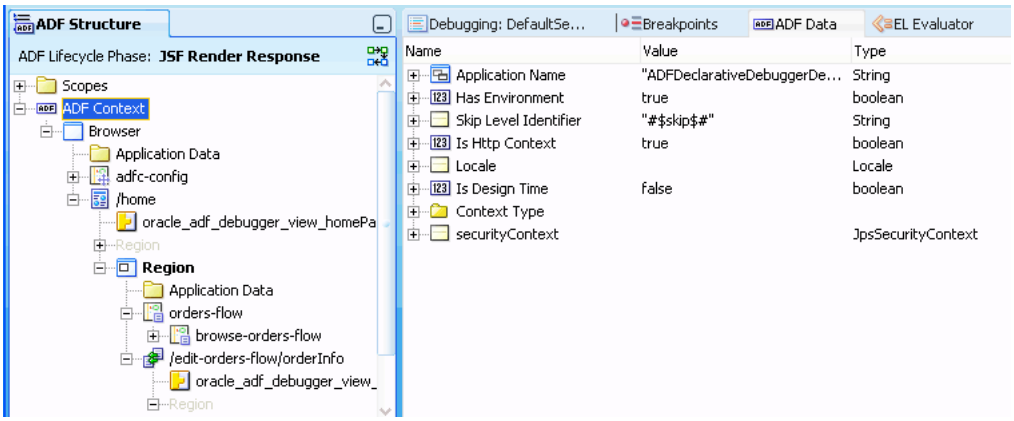
If the **Scopes** node itself is selected, then the full list of memory scopes appears also in the ADF Data windows, which can also be expanded for inspection. Figure 36–36 shows the **Scopes** node selected in the ADF Structure window, and the **viewScope** child node being selected with its values displayed in the ADF Data window. You can inspect the values of `requestScope`, `viewScope`, `pageFlowScope`, `applicationScope`, and `sessionScope` by expanding each corresponding node. `pageFlowScope` will also appear within the ADF Task Flow content to reflect the values of the specific ADF task flow currently selected in the ADF Structure window. `viewScope` will also appear within the view port content to reflect the values of the specific view port currently selected in the ADF Structure window.

Figure 36–36 Scopes Node Selected in the ADF Structure Window



When the ADF context is selected in the ADF Structure window, as shown in Figure 36–37, the current value of the ADF context variables will be displayed in the ADF Data window. You can also inspect ADF context variables and values by evaluating the `#(data.adfContext)` EL expression in the EL Evaluator. For more information, see Section 36.8.4, "How to Use the EL Expression Evaluator."

Figure 36–37 ADF Context Selected for the ADF Data Window



Selecting a view port within the ADF Structure hierarchy will display the view port's current view port details in the ADF Data window, as shown in Figure 36–38. Values displayed for each view port are summarized in Table 36–6.

Figure 36–38 View Port Selected for the ADF Data Window

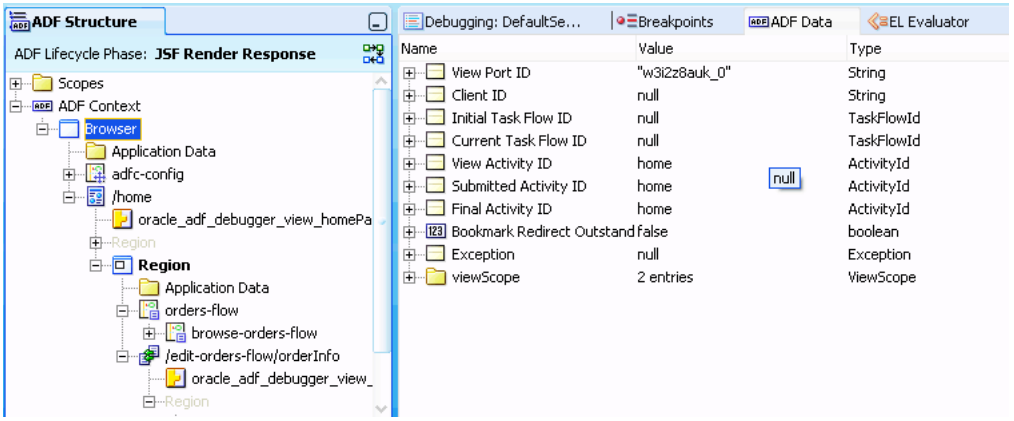


Table 36–6 ADF Data Window Content for View Port

View Port	Description
View port ID	It is displayed
Client ID	It is displayed
Initial task flow ID	Initial ADF task flow on the view ports page flow stack. Not displayed for unbounded task flows. Appears as a link to open the corresponding task flow definition in the editor workspace.
Current task flow ID	Displayed for bounded task flows and not displayed for unbounded task flows. Current ADF task flow on the view port's page flow stack. Appears as a link to open the corresponding task flow definition in the editor workspace.
View activity ID	Current ADF task flow view activity ID. Applicable only if the current ADF task flow activity is a view activity.
Submitted activity ID	ADF task flow activity submitting the current request.
Final activity ID	ADF task flow activity receiving the current request.
Bookmark redirect outstanding	(Boolean)

Table 36–6 (Cont.) ADF Data Window Content for View Port

View Port	Description
Exception	(If any)
View memory scope	View memory scope variables and values for the selected view port.

In the ADF Structure window, each individual ADF task flow within a page flow stack hierarchy is selectable. An ADF task flow selected in the ADF Structure window will display the current task flow information in the ADF Data window, as shown in Figure 36–39. Task flow templates utilized by the selected ADF task flow will be determined by manually navigating to the ADF task flow source file. This is the same way similar functionalities are handled for Java source files. Current information for a selected ADF task flow is summarized in Table 36–7.

Figure 36–39 Task Flow Selected for the ADF Data Window

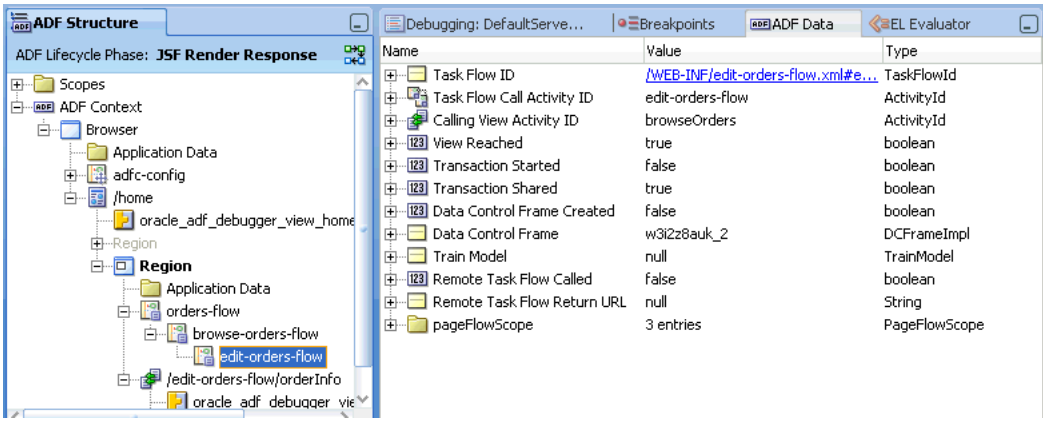


Table 36–7 ADF Data Window Content for Task Flow

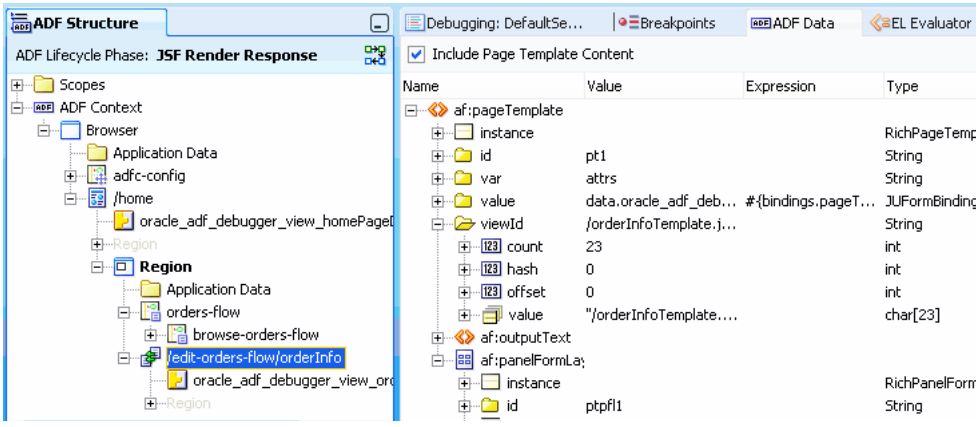
Task Flow	Description
ADF task flow reference	ADF task flow reference
Task flow call activity ID	Task flow activity ID for the calling task flow. Will be `null` for the first ADF task flow within each view port task flow call hierarchy.
Calling view activity ID	The calling view activity of the current view activity displayed by the ADF task flow, if any.
View reached	(Boolean)
Train model	Only applicable if the ADF task flow is created as a train.
Transaction started	(Boolean) Identifies the current status of the ADF task flow transactional state. For example, did the ADF task flow begin a new transaction?
Transaction shared	(Boolean) Identifies the current status of the ADF task flow transactional state. For example, did the ADF task flow join an existing transaction?
Save point	Identifies the current status of the ADF task flow's ADF Model save point creation state. For example, was a model save point created upon ADF task flow entry?.
Remote task flow called	(Boolean)

Table 36–7 (Cont.) ADF Data Window Content for Task Flow

Task Flow	Description
Remote task flow return URL	Applies only when calling an ADF task flow remotely. Identifies the URL for return once the task flow called remotely completes.
Data control frame created	(Boolean)
Data control frame	Name of data control frame associated with the ADF task flow.
Page flow memory scopes	Appears as an expandable node to allow inspection of the values of the page flow memory scopes for the task flow selected in the ADF Structure window. The page flow memory scopes will also be displayed within the ADF Structure window's **Scopes** node. However, the page flow memory scope for the **Scopes** node will always be based on the application's current context, not the selected task flow.

When you select a page or page fragment node in the ADF Structure hierarchy, the corresponding UI component tree is displayed within the ADF Data window, as shown in Figure 36–40. If a page or page fragment is based on a page template, you can include the content coming from the page template outside any facet reference elements by selecting the **Include Page Template Content** checkbox at the top of the ADF Data window. If the page template content is not included, the page or page fragment UI component tree will appear structurally similar to its source file.

Figure 36–40 Page Selected for the ADF Data Window



When you select a binding container in the ADF Structure hierarchy, it displays within the ADF Data window the node selection listed in Table 36–8.

Table 36–8 ADF Data Window Content for Binding Container

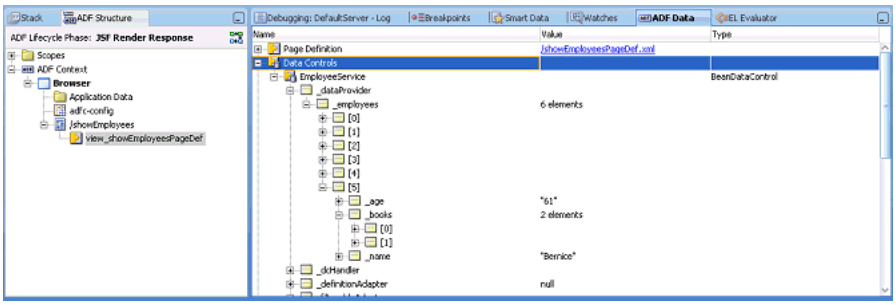
Binding Container	Description
Page definition link	Navigates to the corresponding page definition source file and opens it within the editor workspace.

Table 36–8 (Cont.) ADF Data Window Content for Binding Container

Binding Container	Description
Data Controls	Displays the binding container's data controls. Data controls implemented by ADF Business Components objects and non-ADF Business Components objects will be presented slightly different. ADF Business Components-based data controls will appear similar to the actual business service implementation using row collections. Non-ADF Business Components-based data controls will typically appear as raw member variables similar to what is displayed in the ADF Declarative Debugger Data window. The ADF Data window shows only cached information such as member variables and arrays. Standard debugger functionality can also be used to customize each element. Each ADF Business Components data control will display the following information: ■ Row collections ■ Query string for each row collection ■ Query string with variable substitution for each row collection ■ Application data rows ■ Current row indicator ■ Change indicator ■ Current and original values (if changed within the same request)
Parameters	Current values of all binding container parameters.
Executables	Displays executables showing current row indicators, and current and original values (if changed within the same request). This includes the following types of executables: ■ Iterator - presents corresponding attribute bindings along with their `Refresh` and `RefreshCondition` properties. ■ task flow - current value of the task flow ID assigned to the task flow binding and all of its associated parameter values. Task flow IDs will appear as links navigating to open the corresponding task flow definition source file within the editor workspace. Link text consists of `<task flow source document>#<task flow id>`. ■ Search region - presented similar to iterators, but also displays criteria and criteria with substitution information.
Bindings	Displays value, table, tree, and method bindings. Each binding will display the following information: ■ Associated executables ■ Change indicator ■ Current and original values (if changed within the same request)
Binding container of page template	If the corresponding page or page fragment utilized a page template, the binding container of the page template will appear as a child of page or page fragment binding container content.

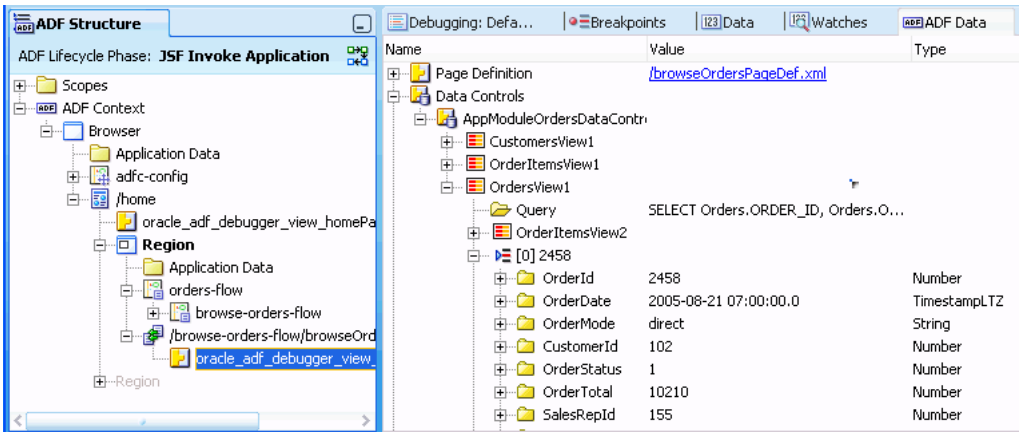
When you select a binding container for an application based on non-ADF Business Components objects, the ADF Data window displays the binding container content, as shown in Figure 36–1.

Figure 36–41 Binding Container (Non-Business Components) Selected for the ADF Data Window



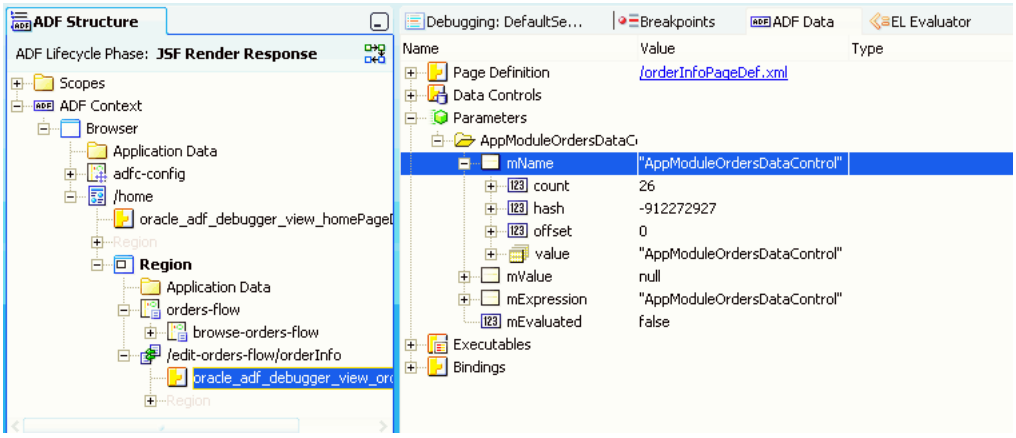
When you select a binding container for an application based on ADF Business Components objects, the ADF Data window displays standard row collection icons, as shown in Figure 36–2.

Figure 36–42 Binding Container (Business Components) Selected for the ADF Data Window



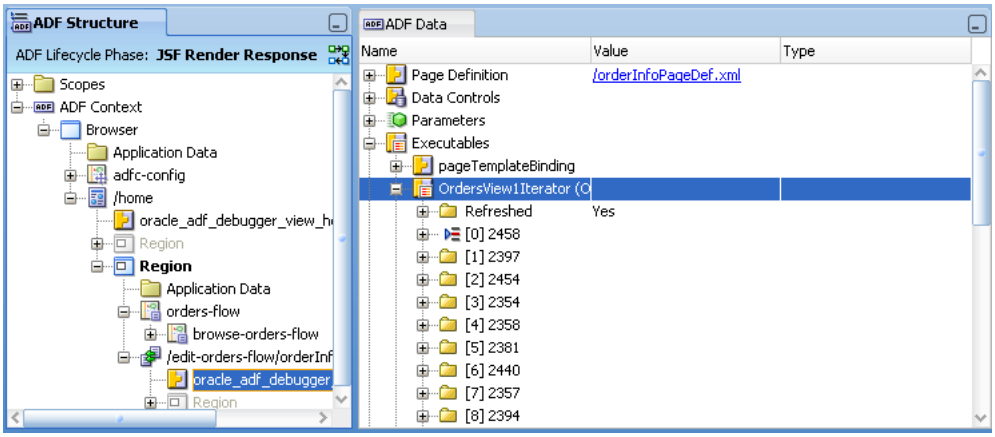
Expanding the Parameters node in the ADF Data window displays information similar to that shown in Figure 36–43.

Figure 36–43 Parameters Selected for the ADF Data Window



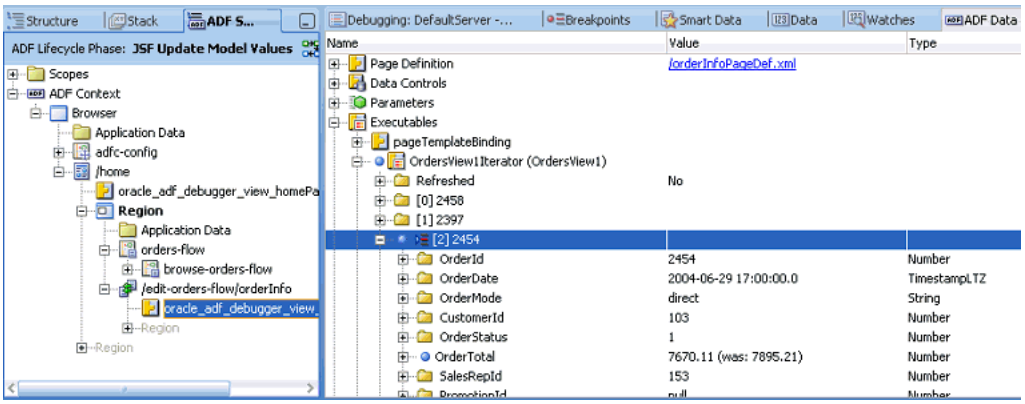
Expanding the Executables node in the ADF Data window displays information similar to that shown in Figure 36–44.

Figure 36–44 Executables Selected for the ADF Data Window



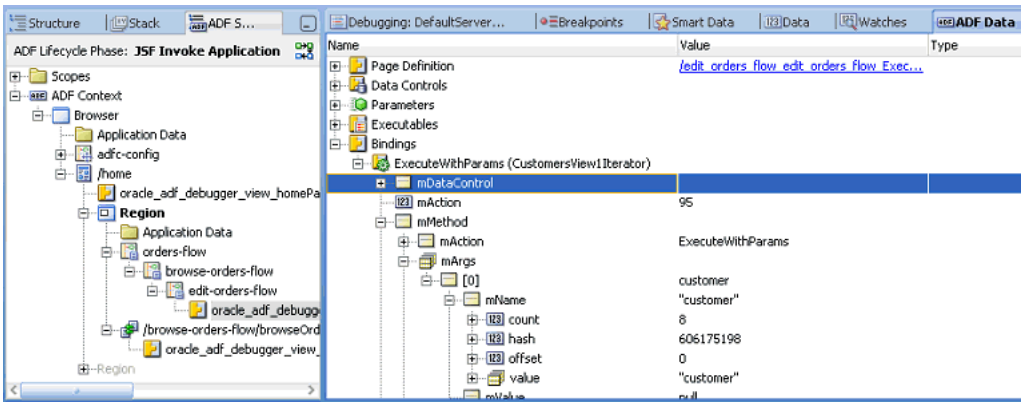
If a value has changed, the changed item will be marked with a blue dot to its left and the previous value is displayed in parenthesis. For instance, suppose the `OrderTotal` value has changed from 7895.81 to 7670.11. The ADF Data window places a blue dot next to `OrderTotal` and its parent `OrdersView1Iterator` and displays the current and previous values in the **Value** column, as shown in Figure 36–45.

Figure 36–45 A Value Change Is Indicated by a Blue Dot in the ADF Data Window



Method binding information is displayed in the ADF Data window similar to what is shown in Figure 36–46.

Figure 36–46 Method Bindings Selected for the ADF Data Window



When you select an Application Data node from the ADF Structure window, the ADF Data window displays the application objects, such as ADF Business Components objects, instantiated within the current data control frame for the corresponding view port (or binding context if ADF Controller is not used).

Business services implemented by ADF Business Components objects display the application data content, as shown in Figure 36–47 and described in Table 36–9.

Figure 36–47 Application Data for ADF Business Components Business Services

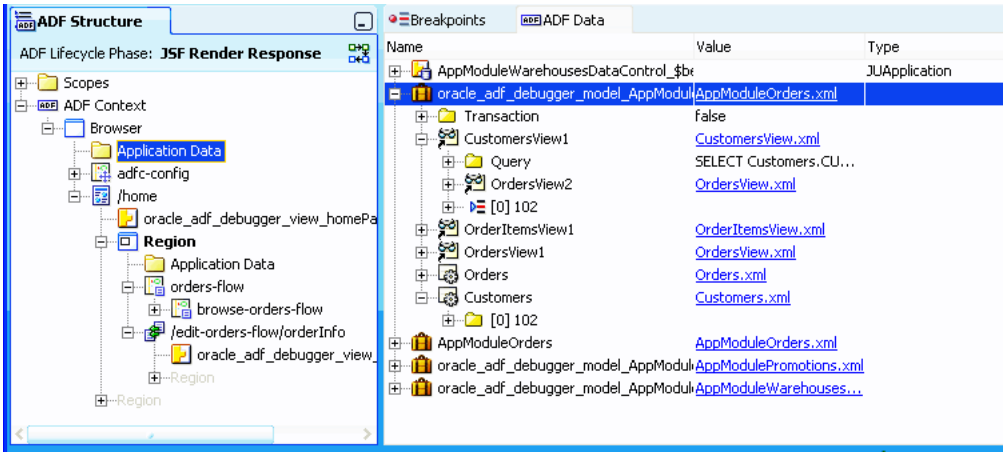


Table 36–9 ADF Data Window Content for Application Data

Binding Container	Description
Application module	The application module(s) of the corresponding view port data control frame will appear within the application data hierarchy as the root node(s). An application module design time icon will be used to identify the node(s). The application module node(s) will provide the following information: ▪ Application module link - link to open the corresponding application module source file within the editor workspace. ▪ Transaction - the application module current transaction status, if applicable. ▪ View objects ▪ Entity objects

Table 36–9 (Cont.) ADF Data Window Content for Application Data

Binding Container	Description
View object	View objects instantiated within the corresponding view port data control frame will appear underneath the corresponding application module root node as subordinate nodes. Design time icons will be used to identify them. Child view objects will appear subordinate to their parent view objects within the hierarchy. Named row sets will appear similar to view objects. Named iterators for a view object will appear similar to child view objects. Each view object node will provide the following information: ■ View object link - link to open the corresponding view object source file within the editor workspace. ■ Query - last executed view object SQL statement. Displays bind variables without value substitution. ■ Query with substitution - last executed view object SQL statement. Displays bind variables with value substitution. ■ Bind variables - last executed view object SQL statement bind variables and their values. ■ View object rows - each row displayed will be identified by its concatenated key values. The current row will be identified with a special icon. ■ Attributes - attributes contained on each row will display their current values along with their originating entity object. Transient attributes will also be displayed. ■ Modifications - changes made within the same request will be identified by a blue dot to left of attribute, row, and view objects node labels. Both the old and new value of the modification will be displayed.
Entity object	Entity objects instantiated within the corresponding view port data control frame will appear underneath the application module root node as subordinate nodes. Design time icons will be used to identify them. Each entity object node will provide the following information: ■ Entity object link - link to open the corresponding entity object source file within the editor workspace. ■ Entity object rows - each row displayed will be identified by its concatenated key values. The current entity-state and post-state of the row (e.g., `STATUS_MODIFIED`) will also be presented. ■ Attributes - attributes contained on each row will display their current values. ■ Modifications - changes made within the same request will be identified by a blue dot to left of attribute, row, and entity objects node labels. Both the old and new value of the modification will be displayed.

Business services implemented by non-ADF Business Components objects display application data content using raw member variables. The format is similar to the display of non-ADF Business Components content for the binding container, as shown in Figure 36–41.

36.9.9 What Happens When You Set an ADF Declarative Breakpoint

When you set an ADF declarative breakpoint, JDeveloper adds the breakpoint to the appropriate class, method, or other construct in the ADF source Java code that

corresponds to the breakpoint. Once the breakpoint is set in the code, the standard Java debugger mechanism pauses application execution when the breakpoint is reached. When the breakpoint is reached, it will be identified by a red dot icon in the Breakpoints window. Depending on the type of declarative breakpoint that was reached, it will also appear as a red dot icon in the task flow activity, in the page definition breakpoint margins, or in the ADF Lifecycle Breakpoints window.

For task flow activity breakpoints, the debugger pauses the application within the JSF Invoke Application phase before the activity where the breakpoint is set. In other words, the activity where the breakpoint is set is not executed.

For task flow view activities, however, the application is paused within the JSF Render Response phase after the view activity is executed, but before the new page is rendered.

For a page definition Before executable breakpoint, the debugger pauses the application when the executable is refreshed. For a page definition Before action binding breakpoint, the debugger pauses the application when the binding is executed. For a page definition Before attribute value binding breakpoint, the debugger pauses the application before the attribute's `setInputValue()` method in the ADF source code is executed.

For a Before lifecycle breakpoint, the debugger pauses the application before it enters the next lifecycle phase. For an After lifecycle breakpoint, the debugger pauses the application after the lifecycle phase and before the next phase.

36.10 Setting Java Code Breakpoints

You can use the ADF Declarative Debugger to set breakpoints on Java classes and methods, as in any standard Java code debugger. You can use Java code breakpoints in combination with ADF declarative breakpoints or by themselves. For most ADF applications, ADF declarative breakpoints will provide enough debugging information to troubleshoot the application. For information about using ADF declarative breakpoints, see Section 36.9, "Setting ADF Declarative Breakpoints." However, you may need to set breakpoints on specific classes or methods for further inspection. Or, you may be debugging a non-ADF application, in which case, you can use Java code breakpoints.

JDeveloper provides a class locator feature that assists you in finding the class you want to break on. If you can obtain Oracle ADF source code, you can enhance your debugging by having access to various ADF classes and methods. For more information about getting ADF source code, see Section 36.8.1, "Using ADF Source Code with the Debugger." If you obtained the ADF source, you can further enhance the debugging experience by using the debug library version of the ADF source, as described in Section 36.10.4, "How to Use Debug Libraries for Symbolic Debugging."

36.10.1 How to Set Java Breakpoints on Classes and Methods

You can set Java breakpoints on your classes and methods. If you have ADF source code, you can set Java breakpoints in the source as well. If you are debugging an ADF application, you should check to see whether ADF declarative breakpoints can be used instead of Java code breakpoints. For more information, see Section 36.9, "Setting ADF Declarative Breakpoints."

Before you attempt to use breakpoints, you should try to run the application and look for missing or incomplete data, actions and methods that are ignored or incorrectly executed, or other unexpected results. If you did not find the problem, create a debugging configuration that will enable the ADF Log and send Oracle ADF messages

to the Log window. For more information, see Section 36.6.3, "How to Create an Oracle ADF Debugging Configuration."

To set Java breakpoints to debug an application:

1. From the main menu, choose **Navigate > Go to Java Type** (or press Ctrl+Minus) and use the dialog to locate the Oracle ADF class that represents the entry point for the processing failure.

   Note: JDeveloper will locate the class from the user interface project with current focus in the Application Navigator. If your workspace contains more than one user interface project, be sure that the one with the current focus is the one you want to debug.

2. Open the class file in the source editor and find the Oracle ADF method call that will enable you to step into the statements of the method.
3. Set a breakpoint on the desired method and run the debugger.
4. When the application stops on the breakpoint, use the Data window to examine the local variables and arguments of the current context.

   Tip: If you are using the **Go to source** context menu command in the Data, Watches, or Smart Data window, you can go back to the execution point by using the back button. You can also access the back button through the **Navigate** menu.

Once you have set breakpoints to pause the application at key points, you can proceed to view data in the Data window. To effectively debug your web page's interaction with the ADF Model layer, you need to understand:

- The ADF page lifecycle and the method calls that get invoked
- The local variables and arguments that the ADF Model layer should contain during the course of application processing

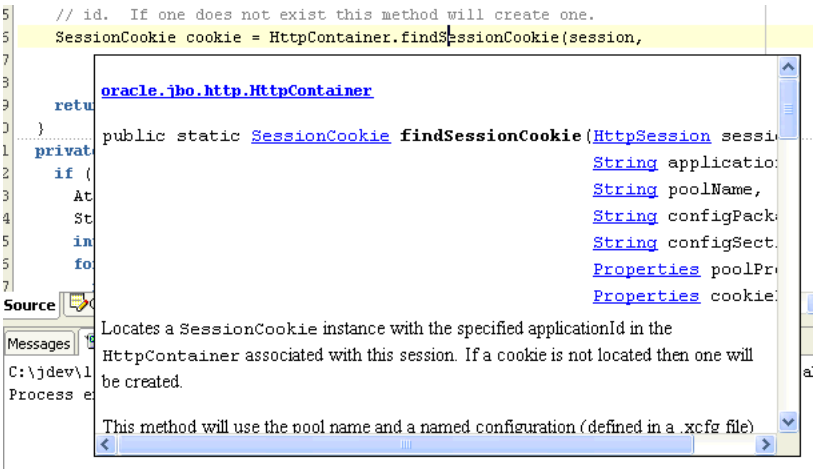
Awareness of Oracle ADF processing will give you the means to selectively set breakpoints, examine the data loaded by the application, and isolate the contributing factors.

Note: JSF web pages may also use backing beans to manage the interaction between the page's components and the data. Debug backing beans by setting breakpoints for them as you would with any other Java class file.

36.10.2 How to Optimize Use of the Source Editor

Once you have added the ADF source library to your project, you have access to the helpful Quick Javadoc feature (Ctrl+D) that the source editor makes available. Figure 36–48 shows Quick Javadoc for a method like `findSessionCookie()`.

Figure 36–48 Using Quick Javadoc on ADF API in the Source Editor



36.10.3 How to Set Breakpoints and Debug Using ADF Source Code

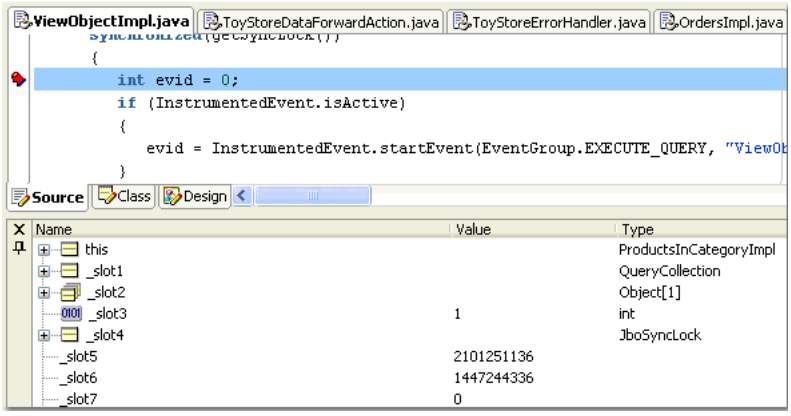
After loading the ADF source code, you can debug any Oracle ADF code for the current project the same way that you do your own Java code. This means that you can press Ctrl+Minus to type in any class name in Oracle ADF, and JDeveloper will open its source file automatically so that you can set breakpoints as desired.

36.10.4 How to Use Debug Libraries for Symbolic Debugging

When debugging Oracle ADF source code, by default you will not see symbol information for parameters or member variables of the currently executing method.

For example, in a debugging session without ADF source code debug libraries, you may see unrecognizable names such as "_slot", as shown in Figure 36–49.

Figure 36–49 Local Symbols Are Hard to Understand Without Debug Libraries

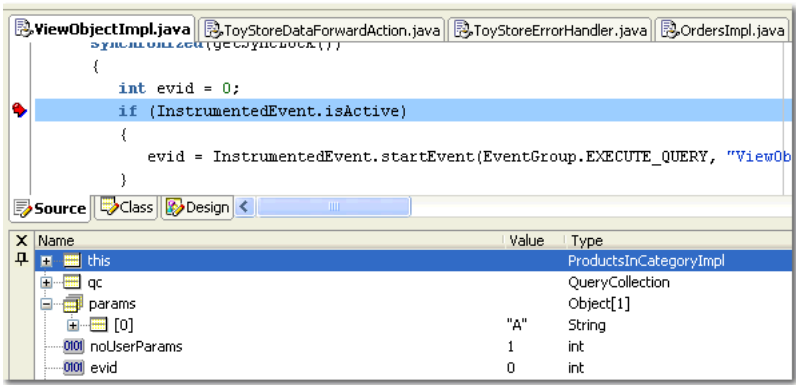


These names are hard to decipher and make debugging more difficult. You can make debugging easier by using the debug versions of the ADF JAR files supplied along with the source while debugging in your development environment.

> **Note:** The supplied debug libraries should not be used in a test or production environment, since they typically have slightly slower runtime performance than the optimized JAR files shipped with JDeveloper.

The debug library JARs are versions of Oracle ADF JARs that have been compiled with additional debug information. When you use these debug JAR files instead of the default optimized JARs, you will see all of the information in the debugger. For example, the variable `evid` is now identified by its name in the debugger, as shown in Figure 36–50.

Figure 36–50 Symbol Information Displayed in the Debugger



Before you replace the standard library JAR, make sure that JDeveloper is not running. If it's currently running, exit from the product before proceeding.

To replace the standard library JARs with the debug library JARs:

1. With JDeveloper closed, make a `backup` subdirectory of all existing optimized JAR files in the `./BC4J/lib` directory of your JDeveloper installation. For example, assuming `jdev11` is the JDeveloper home directory:

   ```
   C:\jdev11\BC4J\lib> mkdir backup
   C:\jdev11\BC4J\lib> copy *.jar backup
   ```

2. For each ADF library that you want debug symbols for while debugging, copy the `_g.jar` version of the matching library over the existing, corresponding library in the `C:\jdev11\BC4J\lib` directory.

   This is safe to do since you made a backup of the optimized JAR files in the `backup` directory in Step 2.

   Since debug libraries typically run a little slower than libraries compiled without debug information, this diagnostic message is to remind you not to use debug libraries for performance timing:

   ```
   ********************************************************************************
   *** WARNING: Oracle BC4J debug build executing - do not use for timing ***
   ********************************************************************************
   ```

3. To change back to the optimized libraries, simply copy the JAR file(s) in question from the `./BC4J/lib/backup` directory back to the `./BC4J/lib` directory.

36.10.5 How to Use Different Kinds of Java Code Breakpoints

You first need to understand the different kinds of Java code breakpoints and where to create them.

To see the debugger Breakpoints window, choose **View > Breakpoints** from the main menu or press Ctrl+Shift+R.

You can create a new Java code breakpoint by choosing **Create Breakpoint** from the context menu in the Breakpoints window. The **Breakpoint Type** dropdown list controls what kind of breakpoint you will create, as shown in Table 36–10.

> **Note:** You can also use the Create Breakpoint dialog to create an ADF lifecycle phase declarative breakpoint. For information about creating ADF declarative breakpoints, see Section 36.9.6, "How to Set and Use ADF Lifecycle Phase Breakpoints."

Table 36–10 Different Types of Java Breakpoints

Breakpoint Type	The Breakpoint Occurs Whenever	Usage
Exception	An exception of this class (or a subclass) is thrown.	An Exception breakpoint is useful when you don't know where the exception occurs, but you know what kind of exception it is (for example, `java.lang.NullPointerException`, `java.lang.ArrayIndexOutOfBoundsException`, `oracle.jbo.JboException`). The checkbox options allow you to control whether to break on caught or uncaught exceptions of this class. The **Browse** button helps you find the fully qualified class name of the exception. The Exception Class combobox remembers the most recently used exception breakpoint classes. Note that this is the default breakpoint type when you create a breakpoint in the breakpoints window.
Source	A particular source line in a particular class in a particular package is run.	You rarely create a source breakpoint in the Create Breakpoint dialog, because it's much easier to create it by first using the **Navigate > Go to Java Type** menu (accelerator Ctrl+Minus), then scrolling to the line number you want — or using **Navigate > Go to Line** (accelerator Ctrl+G) — and finally clicking in the breakpoint margin at the left of the line you want to break on. This is equivalent to creating a new source breakpoint, but it means you don't have to type in the package, class, and line number by hand.
Method	A method in a given class is invoked.	The Method breakpoint is useful for setting breakpoints on a particular method you might have seen in the call stack while debugging a problem. If you have the source, you can set a source breakpoint wherever you want in that class, but this kind of breakpoint lets you stop in the debugger even when you don't have source for a class.

Table 36–10 (Cont.) Different Types of Java Breakpoints

Breakpoint Type	The Breakpoint Occurs Whenever	Usage
Class	Any method in a given class is invoked.	The Class breakpoint can be used when you might know the class involved in the problem, but not the exact method you want to stop on. This kind of breakpoint does not require source. The **Browse** button helps you quickly find the fully qualified class name you want to break on.
Watchpoint	A given field is accessed or modified.	The Watchpoint breakpoint can be used to find a problem if the code inside a class modifies a member field directly from several different places (instead of going through setter or getter methods each time). You can pause the debugger when any field is modified. You can create a breakpoint of this type by using the **Toggle Watchpoint** menu item on the context menu when pointing at a member field in your class's source.

36.10.6 How to Edit Breakpoints for Improved Control

After creating a Java code breakpoint you can edit the breakpoint in the Breakpoints window by right-clicking it and choosing **Edit** in the context menu.

Note: You can use the Edit Breakpoint dialog to edit an ADF declarative breakpoint. However, you cannot edit some of the other information such as the information in the **Definition** tab. You can launch the Edit Breakpoint dialog by choosing **Edit** from the context menu in the Breakpoints window. For information about creating ADF declarative breakpoints, see Section 36.9, "Setting ADF Declarative Breakpoints."

Some of the features you can use by editing your breakpoint are:

- Associate a logical "breakpoint group" name to group this breakpoint with others of the same group name. Breakpoint groups make it easy to enable/disable an entire set of breakpoints in one operation.
- Associate a debugger action to a breakpoint when the breakpoint is hit. The default action is to stop the debugger so that you can inspect the application states, but you can add a sound alert, write information to a log file, and enable or disable group of breakpoints.
- Associate a conditional expression with the breakpoint so that the debugger stops only when that condition is met. The expressions can be virtually any boolean expression, including:
  - `expr ==value`
  - `expr.equals("value")`
  - `expr instanceof.fully.qualified.ClassName`

Note: Use the debugger Watches window to evaluate the expression first to make sure it's valid.

36.10.7 How to Filter Your View of Class Members

You can use the debugger to filter the members that are displayed in the debugger window for any class. In the debugger's Data window, selecting any item and choosing **Preferences** from the context menu brings up a dialog that lets you customize which members appear in the debugger and (more importantly sometimes) which members *don't* appear. You can filter by class type to simplify the amount of scrolling you need to do in the debugger Data window. This is especially useful when you might be interested only in a handful of a class's members.

36.10.8 How to Use Common Oracle ADF Breakpoints

If you loaded Oracle ADF source code, you can use the breakpoints listed in Table 36–11 to debug your application.

By looking at the Stack window when you hit these breakpoints, and stepping through the source, you can get a better idea of what's going on.

Table 36–11 Commonly Used ADF Breakpoints

Breakpoint	Breakpoint Type	Usage
`oracle.jbo.JboException`	Exception	This breakpoint useful for setting a breakpoint on the base class of all ADF Business Components runtime exceptions.
`oracle.jbo.DMLException`	Exception	This is the base class for exceptions originating from the database, like a failed DML operation due to an exception raised by a trigger or by a constraint violation.
`doIt()`	Method	You can also perform the same debugging function by setting an ADF declarative breakpoint on the page definition action binding. See Section 36.9.3, "How to Set and Use Page Definition Action Binding Breakpoints." If you prefer to use this Java breakpoint, you can find it in the `JUCtrlActionBinding` class (`oracle.jbo.uicli.binding` package). This is the method that will execute when any ADF action binding is invoked, and you can step into the logic and look at parameters if relevant.
`oracle.jbo.server.ViewObjectImpl.executeQueryForCollection`	Method	This is the method that will be called when a view object executes its SQL query.
`oracle.jbo.server.ViewRowImpl.setAttributeInternal`	Method	This is the method that will be called when any view row attribute is set.
`oracle.jbo.server.EntityImpl.setAttributeInternal`	Method	You can also perform the same debugging function by setting an ADF declarative breakpoint on the page definition attribute value binding. See Section 36.9.4, "How to Set and Use Page Definition Value Binding Breakpoints." This is the method that will be called when any entity object attribute is set.

36.11 Regression Testing with JUnit

Testing your business services is an important part of your application development process. By creating a set of JUnit regression tests that exercise the functionality provided by your application module, you can ensure that new features, bug fixes, or refactorings do not destabilize your application. JDeveloper's integrated support for creating JUnit regression tests makes it easy test your application. Its integrated support for running JUnit tests means that any developer on the team can run the test suite with a single mouse click, greatly increasing the chances that every team member can run the tests to verify their own changes to the system. Furthermore, by using JDeveloper's integrated support for creating and running Apache Ant build scripts, you can easily incorporate running the tests into your automated build process as well. You can create a JUnit test for your application module, run it, and integrate the tests into an Ant build script.

JDeveloper provides the ability to generate JUnit test cases, test fixtures, and test suites. You can create test cases to test individual Java files containing single or multiple Java classes. You can create JUnit test fixtures that can be reused by JUnit test cases. You can group all these test cases into a JUnit test suite, which you can run together as a unit.

You can also use the JUnit BC4J Test Suite wizard to generate a test suite when there is an application module in the project. The wizard generates a test suite, test fixture, and a test case for each view object in the application module.

You can create a separate project to contain your regression tests or to integrate the test files into an existing project. If you are creating an ADF Business Components test, you should create a separate project for testing.

Creating separate projects for testing has the following advantages:

- The ability to compile the base project without having a dependency on JUnit
- The ability to package the base project for deployment without having to exclude the test classes.

If you are creating separate projects for JUnit testing, you should create directory structures that mirror the structure of the packages being tested. You may want to name the test classes using a naming convention that can easily identify the package being tested. For example, if you are testing `myClass.java`, you can name the test class `myClassTest.java`.

Although having separate projects has many advantages, in certain cases it may be easier to include the tests within the project. For example, the Fusion Order Demo application has a JUnit regression test suite in the `FODCustomization` workspace `Customization Extension` project.

You can use the Create Test wizards in the context of the project to create a JUnit test case, test fixture, or test suite. However, if you do not want to include these tests as part of the deployment, you may want to separate the tests out in their own project.

> **Tip:** If you don't see the Create Test wizards, use JDeveloper's **Help > Check for Updates** feature to install the JUnit Integration extension before continuing.

Each test case class contains a `setUp()` and `tearDown()` method that JUnit invokes to allow initializing resources required by the test case and to later clean them up. These test case methods invoke the corresponding `setUp()` and `tearDown()` methods to prepare and clean up the test fixture for each test case execution. Any time a test in the test case needs access to the application module, it uses the test fixture's

`getApplicationModule()` method. The method returns the same application module instance, saved in a member field of the test fixture class, between the initial call to `setUp()` and the final call to `tearDown()` at the end of the test case.

JDeveloper supports JUnit 4, which allows annotations to be used instead of explicitly having to name the methods `setUp()` and `tearDown()`. These annotations — `@Before`, `@After` — allow you to have multiple setup and teardown methods, including inherited ones if required.

The generated `ExampleModuleConnectFixture` is a JUnit test fixture that encapsulates the details of acquiring and releasing an application. It contains a `setUp()` method that uses the `createRootApplicationModule()` method of the `Configuration` class to create an instance of an application module. Its `tearDown()` method calls the matching `releaseRootApplicationModule()` method to release the application module instance.

Your own testing methods can use any of the programmatic APIs available in the `oracle.jbo` package to work with the application module and view object instances in its data model. You can also cast the `ApplicationModule` interface to a custom interface to have your tests invoke your custom service methods as part of their job. During each test, you will call one or more `assert`*`Xxx`*`()` methods provided by the JUnit framework to assert what the expected outcome of a particular expression should be. When you run the test suite, if any of the tests in any of the test cases contains assertions that fail, the JUnit Test Runner window displays the failing tests with a red failure icon.

The JUnit test generation wizard generates skeleton test case classes for each view object instance in the data model, each of which contains a single test method named `testAccess()`. This method contains a call to the `assertNotNull()` method to test that the view object instance exists.

```
// In ViewInstanceNameTest.java test case class
  public void testSomeMeaningfulName() {
  // test assertions here
  }
```

Each generated test case can contain one or more test methods that the JUnit framework will execute as part of executing that test case. You can add a test to the test case simply by creating a public void method in the class whose name begins with the prefix `test` or use the annotation `@Test`.

36.11.1 How to Obtain the JUnit Extension

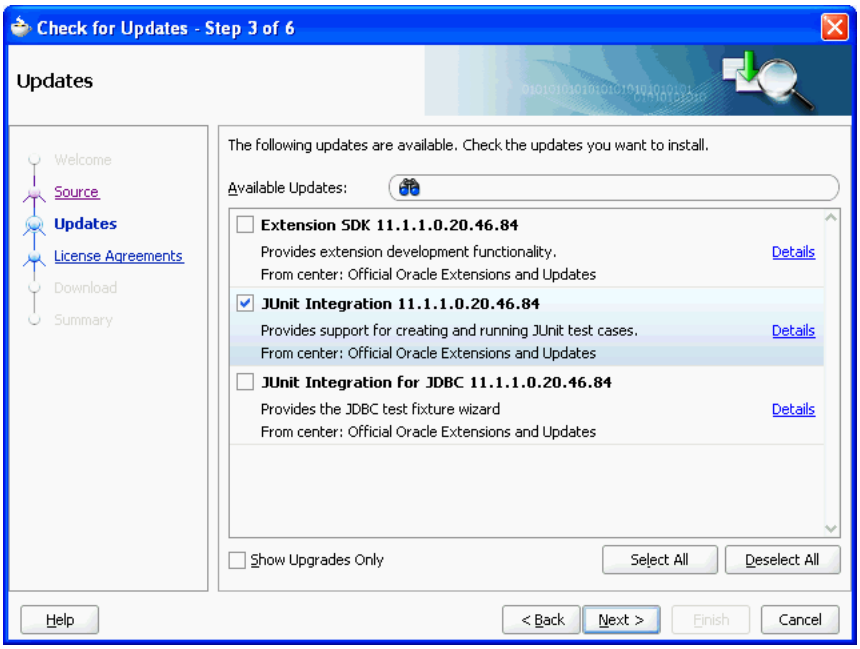
JUnit must be loaded as an extension to JDeveloper before it becomes available and appears in the menu system.

To load the JUnit extension:

1. From the main menu, choose **Help > Check for Updates**.
2. In the Source page of the Check for Updates dialog, select **Search Update Centers** and **Official Oracle Extensions and Updates** and click **Next**.

   If you have the JUnit zip file or if the JUnit selection does not appear in the **Available Updates** list, select **Install From Local File** to load the JUnit zip file.
3. In the Updates page, select **JUnit Integration** and click **Next**, as shown in Figure 36–51.

Figure 36–51 Check for Updates Wizard for Adding JUnit Extension



4. On the License Agreements page, click **I Accept** and click **Finish**.

36.11.2 How to Create a JUnit Test Case

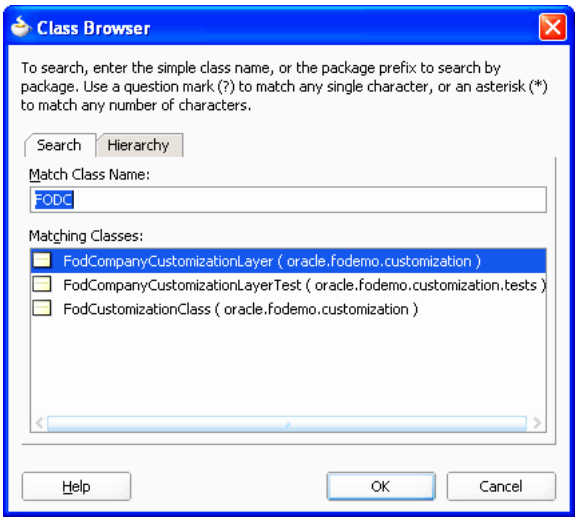
Before you create a JUnit test case, you must have created a project that is to be tested.

To generate a JUnit test case:

1. In the Application Navigator, select the project you want to generate a test case for, right-click and choose **New**.
2. In the New Gallery, expand **General**, select **Unit Tests(JUnit)** and then **Test Case**, and click **OK**.
3. In the Select the Class to Test page of the Create Test Case dialog, enter the class under test or click **Browse**.
4. In the Class Browser dialog, locate the class you want to test or enter the beginning letters in the **Match Class Name** field. The **Match Class** list will be filtered for easier identification.

   For example, entering `FOD` filters the list down to three items, as shown in Figure 36–52.

Figure 36–52 Class Browser for Selecting Class Files to Test

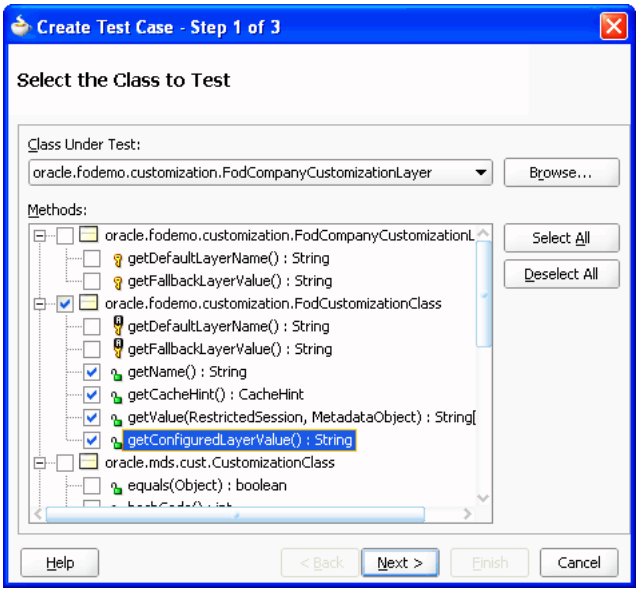


Select the class and click **OK** to close the dialog. Click **Next**.

5. Select the individual methods you want to test, and click **Next**.

   For example, in Figure 36–53, the four methods that are checked are to be tested.

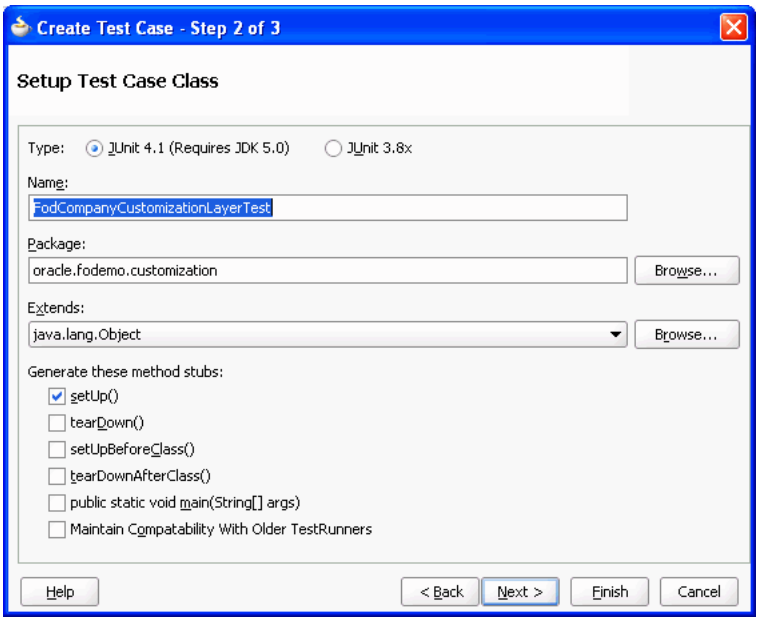
Figure 36–53 Create Test Case Dialog for Selecting Methods to Test



6. In the Setup Test Case Class page, enter the name of the test case, the package, and the class it extends and select the list of built-in functions JUnit will create stubs for. Click **Next**.

   For example, in Figure 36–54, JUnit will create a stub for the `setUp()` method for the `FodCompanyCustomizationLayerTest` test case in the `oracle.fodemo.customization` package.

Figure 36–54 Create Test Case Dialog for Setting Up Classes to Test



7. In the Select Test Fixtures page, select any test fixtures you want to add to the test case or click **Browse**.
8. Make sure that all the test fixtures you want to add to the test case are selected in the list and click **Finish**.

36.11.3 How to Create a JUnit Test Fixture

You should create a JUnit test fixture if you require more than one test for a class or method. A JUnit text fixture allows you to avoid duplicating test code that is needed to initialize testing.

To generate a JUnit test fixture:

1. In the Application Navigator, select the project you want to generate a test fixture for, right-click and choose **New**.
2. In the New Gallery, expand **General**, select **Unit Tests(JUnit)** and then **Test Fixture**, and click **OK**.
3. In the Create Test Fixture dialog, enter the name of the test fixture, the package, and any class it extends.
4. Click **OK**.

36.11.4 How to Create a JUnit Test Suite

Before you create a JUnit test suite, you should have already created JUnit test cases that can be added to the test suite.

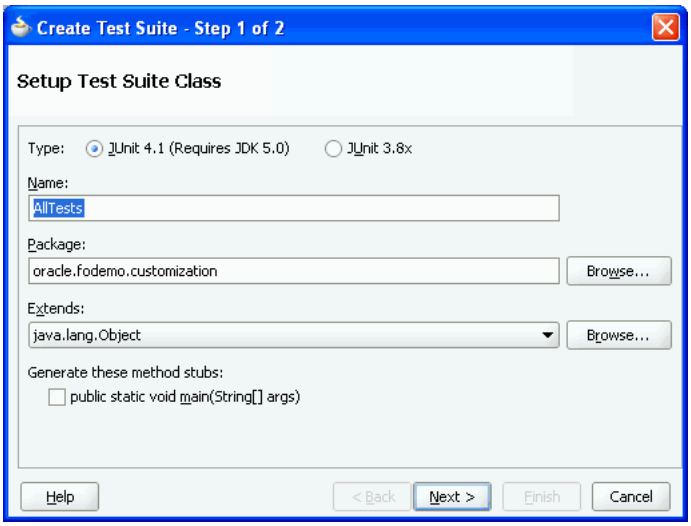
To generate a JUnit test suite:

1. In the Application Navigator, select the project you want to generate a test fixture for, right-click and select **New**.
2. In the New Gallery, expand **General**, select **General** and then **Test Suite**, and click **OK**.

3. In the Setup Test Suite Class page of the Create Test Suite dialog, enter the name of the test suite, the package, and the class it extends. Click **Next**.

   For example, in Figure 36–55, an `AllTests` test suite is created that extends the `java.lang.Object` class.

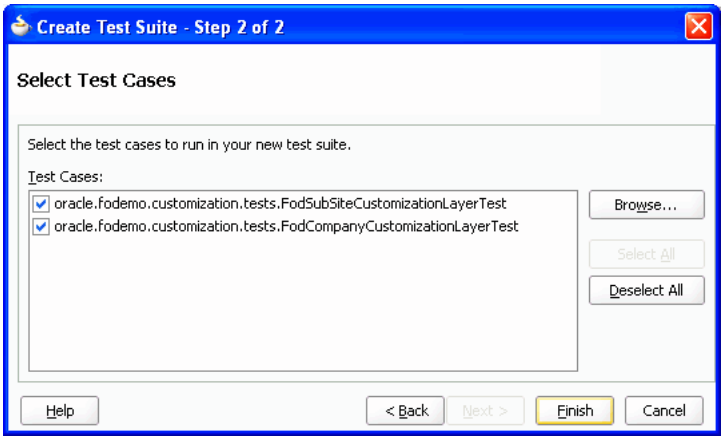
Figure 36–55 Create Test Suite Wizard



4. In the Select Test Cases page of the Create Test Suite dialog, check that all the test cases you want included in the test suite have been selected. The test cases you have created will populate the list. Deselect any test cases that you do not want included. Click **Finish.**

   For example, in Figure 36–56, both test cases are selected to be in the test suite.

Figure 36–56 Selecting Test Cases for a Test Suite



36.11.5 How to Create a Business Components Test Suite

The test fixture that is created is a singleton class to reduce the number of connections. If you want to connect or disconnect for each test case, customize the test case using the JUnit 4 annotations `@Before` and `@After`.

The JUnit BC4J Test Suite wizard will generate tests for each view object in the application module. If the application module does not have exported methods, the

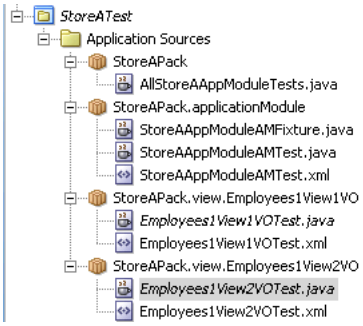
wizard will also generate a test for the application module itself. A generated view object class has the format *view_object*VOTest.java and is placed into a package with the format *package*.view.*viewobject*VO, where *package* is the application module package. A generated application module test has the format *application_module*AMTest.java and is placed into a package with the format *package*.applicationModule. A generated test fixture class has the format *applicationmodule*AMFixture.java and is placed in the same package as the application module test.

The generated all test suite class has the format All*applicationmodule*Test.java and is placed into the package with the same name as the application module package name.

A test case XML file is also generated for each application module or view object test. The XML file contains test methods defined in the application module or view object test cases. It does not include the test methods from the base classes (if any) because there may be too many duplicates.

For instance, after you created a test suite for an application module named StoreAAppModule with view objects Employees1View1 and Employees1View2 in the package StoreAPack, the Application Navigator displays the test hierarchy as shown in Figure 36–19.

Figure 36–57 Business Components Test Suite in the Application Navigator



Before you begin:

Create application modules in the project.

To create a business components test suite:

1. In the Application Navigator, click **New**.

   You will create a separate project for the business components tests.

2. In the New Gallery, expand **General**, select **Projects** and then **Java Projects**, and click **OK**.

3. In the Project Name page of the Create Java Project wizard, enter a name and the directory path for the test project, and click **Next**.

4. In the Project Java Settings page, enter the package name, the directory of the Java source code in your project, and output directory where output class files will be placed, and click **Finish.**

5. In the Application Navigator, double-click the application module you want to test.

6. In the overview editor, click the **Java** navigation tab.

7. In the Java page, click the **Edit** icon for the **Java Class** section.
8. In the Select Java Options dialog, select **Generate Application Module Class** and click **OK**.
9. In the overview editor **Java** tab, click the **Edit** icon in the **Class Interface** section.
10. In the Edit Client Interface dialog, shuttle the methods you want to test to the **Selected** pane, and click **OK**.
11. In the Application Navigator, right-click the test project you have created and choose **New**.
12. In the New Gallery, expand **General**, select **Unit Tests** and then **Business Components Test Suite**, and click **OK**.
13. In the Configure Tests page of the JUnit BC4J Test Suite wizard, select values for the following and click **Next**:
    - **Business Component Project**: Select the project that has the application module you want to test.
    - **Application Module**: Select the application module you want to test.
    - **Configuration**: Choose a local or shared application module.
    - **Test Base Class-Application Module Extends**: You can specify different base cases. The generated test case classes will extend from that base class where all public abstract methods in the base class will have simple and default implementation method bodies.
    - **Test Base Class-View Object Extends**: You can specify which class the view object extends. The generated test case classes will extend from that base class where all public abstract methods in the base class will have simple and default implementation method bodies.
14. In the Summary page, verify the selections and click **Finish**.

36.11.6 How to a Create Business Components Test Fixture

When you create a business components test suite, a business components test fixture is created with it. You can also create Business Components test fixtures independently.

A generated test fixture class has the format *applicationmodule*`AMFixture.java` and put into a package with the format *package*`.applicationModule`, where *package* is the application module package.

Before you begin:

Create application modules in the project.

To create a business components test fixture:

1. In the Application Navigator, click **New**.

   You will create a separate project for the business components tests.
2. In the New Gallery, expand **General**, select **Projects** and then **Java Projects**, and click **OK**.
3. In the Project Name page of the Create Java Project dialog, enter a name and the directory path for the test project, and click **Next**.

4. In the Project Java Settings page, enter the package name and the source and output directories, and click **Finish.**
5. In the Application Navigator, double-click the application module you want to test.
6. In the overview editor, click the **Java** navigation tab and click the **Edit** icon of the **Java Class** section.
7. In the Select Java Options dialog, select **Generate Application Module Class**, and click **OK**.
8. In the overview editor, click the **Edit** icon of the **Class Interface** section.
9. In the Edit Client Interface dialog, shuttle the methods you want to test to the **Selected** pane, and click **OK**.
10. In the Application Navigator, right-click the test project you have created and choose **New**.
11. In the New Gallery, expand **General**, select **Unit Tests** and then **Business Components Test Fixture**, and click **OK**.
12. In the Configure Tests page of the JUnit BC4J Test Fixture wizard, select values for the following and click **Next**:
    - **Business Component Project**: Select the project that has the application module you want to test.
    - **Application Module**: Select the application module you want to test.
    - **Configuration**: Choose a local or shared application module.
13. In the Summary page, verify the test fixture class and click **Finish**.

36.11.7 How to Run a JUnit Test Suite as Part of an Ant Build Script

Apache Ant is a popular, cross-platform build utility for which JDeveloper offers design time support. You can incorporate the automatic execution of JUnit tests and test output report generation by using Ant's built-in `junit` and `junitreport` tasks. Example 36–3 shows a task called `tests` from the `FODCustomizations` Ant `build.xml` file in the `CustomizationExtension` project. It depends on the `build` and `buildTests` targets that Ant ensures have been executed before running the `tests` target.

Example 36–3 Ant Build Target Runs JUnit Test Suite

```
<target name="testCustomizations" depends="compileExtensionClasses">
  <junit printsummary="yes" haltonfailure="yes">
    <classpath refid="customization.classpath">
      <pathelement location="${customization.build.dir}"/>
    </classpath>
    <formatter type="plain"/>
    <test name="oracle.fodemo.customization.tests.AllTests"/>
  </junit>
</target>
```

The `junit` tag contains a nested `test` tag that identifies the test suite class to execute and specifies a directory in which to report the results. The `junitreport` tag allows you to format the test results into a collection of HTML pages that resemble the format of Javadoc.

To try running the JUnit test from Ant, select the `build.xml` file in the Application Navigator, and choose **Run Ant Target >** ***tests*** from the context menu.

37

Refactoring a Fusion Web Application

This chapter describes considerations for renaming, moving, and deleting files, configuration files, objects, attributes, and elements. In most cases, JDeveloper automatically performs refactoring. However, you may need to complete some manual steps to refactor.

This chapter includes the following sections:

- Section 37.1, "About Refactoring a Fusion Web Application"
- Section 37.2, "Renaming Files"
- Section 37.3, "Moving JSF Pages"
- Section 37.4, "Refactoring pagedef.xml Bindings Objects"
- Section 37.5, "Refactoring ADF Business Components"
- Section 37.6, "Refactoring ADF Business Component Object Attributes"
- Section 37.7, "Refactoring Named Elements"
- Section 37.8, "Refactoring ADF Task Flows"
- Section 37.9, "Refactoring the DataBindings.cpx File"
- Section 37.10, "Refactoring Limitations"
- Section 37.11, "Moving the ADF Business Components Project Configuration File (.jpx)"

37.1 About Refactoring a Fusion Web Application

JDeveloper provides refactoring options to rename, move, and delete attributes, named elements, and ADF Business Components objects that your application uses. These refactoring options synchronize your changes with other parts of the application that are dependent on the changes. For example, renaming an ADF Business Components object such as a view object using the **Rename** option renames any references to it in other XML source files.

37.1.1 Refactoring Use Cases and Examples

During the course of development, you will create objects such as entity objects, as needed, to satisfy the needs of the application. Then you might find that you need to subsequently need to delete an object that is no longer used, rename an object to fit a naming convention, or move a set of objects into a different package to consolidate their location. Using JDeveloper, you can refactor these objects so that they are

updated and the references to these objects in other objects are also updated to maintain the integrity of the whole application.

37.2 Renaming Files

You can rename files such as configuration files using the following methods:

- In the Application Navigator, select the file and choose **File** > **Rename** from the main menu.
- In the Application Navigator, right-click the file and choose **Refactor > Rename**.
- In the source editor, select a class name, right-click it, and choose **Rename**.

37.3 Moving JSF Pages

In addition to the other refactoring operations, you can change the package of a JSF page. In the Application Navigator, right-click a JSF page and choose **Refactor** > **Move to Package** to move the page to another package.

Moving the JSF page to another package updates:

- `faces-config.xml` files that reference the page and its package
- ADF task flows containing views associated with the page
- `DataBindings.cpx` mappings to the page

37.4 Refactoring pagedef.xml Bindings Objects

The `pagedef.xml` binding objects that you can refactor include bindings and executables. For more information, see Section 13.7, "Working with Page Definition Files."

Before you begin:

It may be helpful to have an understanding of the options you have for refactoring. For more information, see Section 37.1, "About Refactoring a Fusion Web Application."

In JDeveloper, open the application that contains the objects you want to refactor.

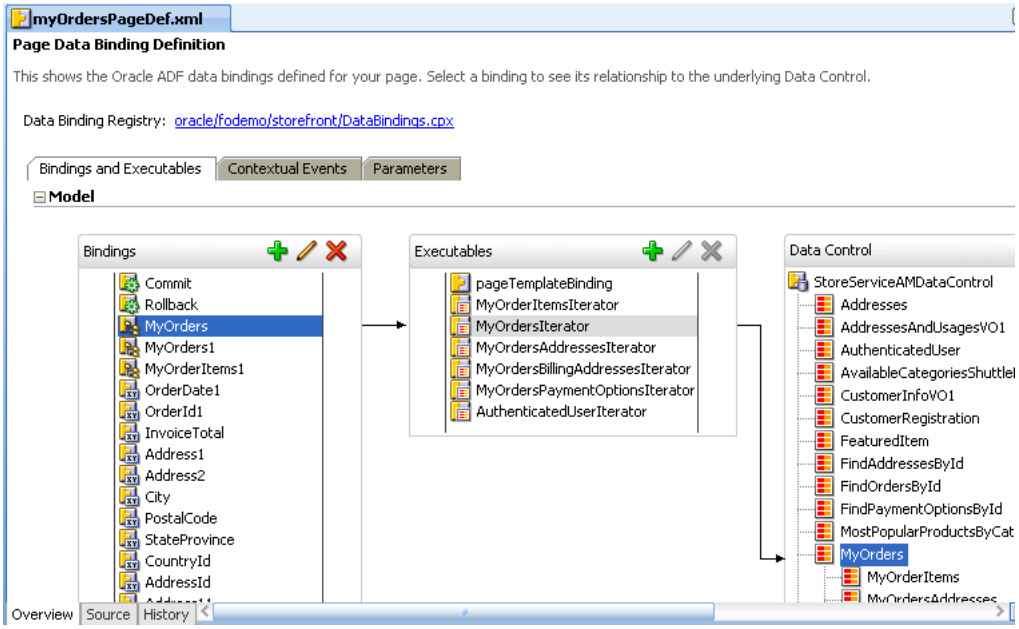
To refactor pagedef.xml binding objects:

1. In the Application Navigator, select the page node on which you have added a bound object such as an ADF Form or selection list.
2. Right-click the page node and choose **Go to Page Definition**.

   If the page does not already have a page definition, the Create Page Definition dialog displays. Click **OK** to create a page definition for the page.
3. In the overview editor, expand the **Model** section.

   Data bindings such as list bindings and iterator bindings defined for the page display under **Bindings** and **Executables**, as shown in Figure 37–1.

Figure 37–1 Page Data Binding Definition Overview Tab



4. Right-click a data binding or executable, choose **Refactor** and a refactoring option such as **Rename** or **Delete**.

5. To display the usages between bindings, executables, and data controls, right-click a binding or executable and choose **Find Usages**.

37.5 Refactoring ADF Business Components

ADF Business Components includes objects such as view objects and entity objects. Table 37–1 shows support for refactoring ADF Business Components.

Table 37–1 Refactoring ADF Business Components

Action	Result
Move	Moves the object to a different package or directory and updates all references.
Delete	JDeveloper shows all dependencies on the object and permits a forced delete. The application may not work at this point. You may need to resolve broken references.
Rename	ADF Business Components objects are defined by an XML file. The XML file has a file name identical to the object name. For example, the name of the XML file for a view object named `Persons1View` is `Persons1View.xml`. Renaming results in changing the `Name` attribute, renaming the XML file, and updating all references. For example, the name of an entity (`Customer`) is stored as an attribute in the XML file (`name=Customer`). The XML file has the same name the entity name (`Customer.xml`).
Find Usages	JDeveloper shows all dependencies on the object.

Note: Refactoring does not cross abstraction layers. For example, when a view object is created based on the `Dept` entity object, it is named `DeptView` by default. Renaming the `Dept` entity object updates the entity usage in `DeptView`, but does not change the name of the view object.

Before you begin:

It may be helpful to have an understanding of the options you have for refactoring. For more information, see Section 37.1, "About Refactoring a Fusion Web Application."

In JDeveloper, open the application that contains the objects you want to refactor.

To refactor ADF Business Components objects:

1. In the Application Navigator, expand the **Projects** node containing the object you want to refactor.
2. Within the project, expand the **Application Sources** node and then the package containing the object you want to refactor.
3. Right-click the object and choose **Refactor > Rename** or **Refactor > Move**.
4. To delete the object, choose **Delete**.

   If the object is used elsewhere in the application, a dialog displays with the following options:

   - **Ignore**: Unresolved usages will remain in the code as undefined references.
   - **View Usages**: Display a preview of the usages of the element in the Compiler log. You can use the log to inspect and resolve the remaining usages.

37.6 Refactoring ADF Business Component Object Attributes

Table 37–1 shows support for refactoring attributes of ADF Business Component entity objects and view objects.

Table 37–2 Refactoring Attributes

Action	Result
Move	Not supported.
Delete	JDeveloper shows all dependencies on the attribute and permits a forced delete. The application may not work at this point. You may need to resolve broken references.
Rename	Attributes share data elements represented in entity and view objects (see Section 4.1, "About Entity Objects" for more information). References to the attribute are updated when you rename the attribute. Renaming results in changing the `Name` attribute, renaming the XML file, and updating all references. Renaming an attribute does not change the data it represents, nor does it rename the underlying table column.
Find Usages	JDeveloper shows all dependencies on the attribute.

Before you begin:

It may be helpful to have an understanding of the options you have for refactoring. For more information, see Section 37.1, "About Refactoring a Fusion Web Application."

In JDeveloper, open the application that contains the attributes you want to refactor.

To refactor attributes:

1. In the Application Navigator, expand the **Projects** node containing the object you want to refactor.
2. Within the project, expand the **Application Sources** node and then the package containing the object you want to refactor.
3. Double-click the object.
4. In the overview editor, select **Attributes**.
5. In the **Name** column, select an attribute.
6. Right-click and choose **Rename** or **Delete**.
7. To delete the attribute, choose **Delete**.

   If the attribute is used elsewhere in the application, a dialog displays with the following options:

   - **Ignore**: Unresolved usages will remain in the code as undefined references.
   - **View Usages**: Display a preview of the usages of the attribute in the Compiler log. You can use the log to inspect and resolve the remaining usages.

37.7 Refactoring Named Elements

Table 37–3 shows support for refactoring named elements in an XML schema. Named elements are any elements in the XML schema that can be referenced by a Name attribute. A named element is not an object or an attribute.

Table 37–3 Refactoring Named Elements

Action	Result
Move	Not supported.
Delete	Not supported.
Rename	One exception to the definition of named elements is the design time element `Attr`, which does have a `Name` attribute. `Attr` is a name-value pair, is not accessible from the code editor, and should not be renamed.
Find Usages	JDeveloper shows all dependencies on the named element.

Before you begin:

It may be helpful to have an understanding of the options you have for refactoring. For more information, see Section 37.1, "About Refactoring a Fusion Web Application."

In JDeveloper, open the application that contains the objects you want to refactor.

To refactor named elements:

1. In the Application Navigator, expand the **Projects** node containing the object you want to refactor.
2. Within the project, expand the **Application Sources** node and then the package containing the object you want to refactor.
3. Double-click the object.
4. In the overview editor, click the **Source** tab.

5. Scroll down to a named element.

   Named elements are indicated by `Name="<element>"` in the source code, for example:

   ```
   <Key Name="PersonsAffContactChk">
   ```

   A named element is not an object or attribute.

6. To rename the element, right-click the element and choose **Rename**.

7. To delete the element, choose **Delete**.

   If the element is used elsewhere in the application, a dialog displays with the following options:

   - **Ignore**: Unresolved usages will remain in the code as undefined references.
   - **View Usages**: Display a preview of the usages of the element in the Compiler log. You can use the log to inspect and resolve the remaining usages.

37.8 Refactoring ADF Task Flows

For more information, see Section 18.6, "Refactoring to Create New Task Flows and Task Flow Templates."

37.9 Refactoring the DataBindings.cpx File

The `DataBindings.cpx` file defines the Oracle ADF binding context for the entire application and provides the metadata from which the Oracle ADF binding objects are created at runtime (see Section A.7.1, "DataBindings.cpx Syntax" for more information). This file is a registry used to quickly find all `.cpx`, `.dcx`, `.jpx`, and `.xcfg` files, which are themselves registries of metadata.

If you rename the `DataBindings.cpx` file to a new name, such as `DataBindingsNew.cpx`, the change is added to the `adfm.xml` file.

Example 37–1 shows the contents of the `adfm.xml` file after `DataBindings.cpx` is refactored to `DataBindingsNew.cpx`.

Example 37–1 Renamed DataBindings.cpx file in adfm.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<MetadataDirectory xmlns="http://xmlns.oracle.com/adfm/metainf"
version="11.1.1.0.0">
<DataBindingRegistry path="adf/sample/view/DataBindingsNew.cpx"/>
</MetadataDirectory>
```

After this change, the application won't run because the data control cannot be found. Example 37–2 shows the old ID in the `DataBindingsNew.cpx` file. To enable the application to access the correct bindings file, update the ID value in the `DataBindingsNew.cpx` file to the new file.

Example 37–2 DataBindingsNew.cpx ID

```
<Application xmlns="http://xmlns.oracle.com/adfm/application"
version="11.1.1.49.28" id="DataBinding" SeparateXMLFiles="false"
Package="adf.sample.view" ClientType="Generic">
```

This should be changed to an ID similar to the one shown in Example 37–3.

Example 37–3 Updated DatabindingNew.cpx ID

```
<Application xmlns="http://xmlns.oracle.com/adfm/application"
version="11.1.1.49.28" id="DataBindingNew" SeparateXMLFiles="false"
Package="adf.sample.view" ClientType="Generic">
```

37.10 Refactoring Limitations

Table 37–4 summarizes the limitations of JDeveloper's refactoring support.

Table 37–4 Refactoring Limitations

Area	Limitation
Database	When a database artifact used in an ADF Business Components object is renamed, the object needs to be updated. This type of refactoring is currently not supported.
Service interface	A service interface defines a contract between two separate pieces of software. For example, the ADF Business Components service interface is responsible for exposing business components to the view and model layers. Changing the name of a service interface can cause a conflict. While developing the application, consider removing the service interface, refactoring the object, and regenerating the service interface. Additionally, if your service interface defines a find operation based on a view criteria that specifies bind variables, changing the number or order of the bind variables in the underlying view criteria will require that you regenerate the service interface. For more information, see Chapter 11, "Integrating Service-Enabled Application Modules."
Java literal references	The Java code generated by ADF Business Components has literal references to the XML metadata. These literal references are updated during refactoring operations. Generated (type safe) methods are also updated. Also, the refactor delete operation is available for local Java variables. However, if the application code directly refers to the metadata, these references are not updated.
Domain	If a domain needs to be renamed or moved, you must create the new domain, then change the type of existing domain usages. For example, you might rename a domain called `EmployeeID` to `EmployeeNumber`. In addition, the entity `Emp` has an attribute called `Empno` that is of type `EmployeeID`. After creating the new domain `EmployeeNumber`, go to the attributes page for the entity, right-click **Empno** and choose **Change Type**. This switches `Empno` from `EmployeeID` to `EmployeeNumber`.
Security	Security policies in the policy store may reference the name of an entity object, attribute, page, or task flow. These policy definitions are not updated in response to the refactoring of the object itself.

Table 37–4 (Cont.) Refactoring Limitations

Area	Limitation
Resource Bundles	Entity object definitions can reference a resource in one or more arbitrary resource bundle (`.properties`) files that you create. You can use this file to define labels for the attributes of entity objects. However, if you rename the `.properties` file you created, JDeveloper will not update the entity object definitions to reflect the new file name. As an alternative to resource bundle files that you create, you can specify a project setting to generate a single, default resource bundle file for the data model project. In this case, JDeveloper will not allow you to rename this generated file. However, if you attempt to change the project-level default resource bundle file, JDeveloper will warn you about the change. The data model project will honor the new ADF Business Components project-level setting for any objects that have not yet been linked to the default resource bundle file; all existing Business Components that have already been linked to the original default file will continue to use it instead.
`.jpx` project configuration file	Renaming the ADF Business Components project configuration file (`.jpx`) is not supported. In previous versions of JDeveloper, ADF Business Components project configuration `.jpx` files were created only in the root package of the `src` directory of a project and were named with the same base name as the project. The ADF Business Components objects (entities, views, and application modules) were all created in the model package (for example, `/model/AppModule.xml`), but the `/Model.jpx` is not. This may cause a reusability problem when attempting to package them in ADF JAR files for use on the class path. There may be name conflicts because several projects are named `Model`.

37.11 Moving the ADF Business Components Project Configuration File (.jpx)

Although refactoring the ADF Business Components project configuration file (`.jpx`) is not supported, you may need to change its name or location to avoid conflicts when sharing your project contents as an ADF library. The `.jpx` file contains configuration information that JDeveloper uses in the design time to allow you to create the data model project with ADF Business Components. If you need to refactor this file, you must do so manually.

Before you begin:

It may be helpful to have an understanding of the options you have for refactoring. For more information, see Section 37.1, "About Refactoring a Fusion Web Application."

To manually move the ADF Business Components project configuration file (.jpx):

1. Move the `.jpx` file to the new source tree location.

   For example, you can move the `Model.jpx` file from `src/Model.jpx` to `src/newpackage/name/here/Model.jpx`.

2. Change the `.jpx` file contents. The `PackageName` attribute of the root element `JboProject` needs to have the correct value.

   For example, you can specify `PackageName="newpackage.name.here"`.

3. Change the `jbo.project` attributes in all `common/bc4j.xcfg` files that contain elements referred to in the `.jpx` file to include the new package name.

For example:

```
<AppModuleConfig name="ScottDeptAMLocal"
ApplicationName="newpackage.name.here.ScottDeptAM"
DeployPlatform="LOCAL" JDBCName="scottdb"
jbo.project="newpackage.name.here.Model">
```

4. Change the JDeveloper project file (the `.jpr` file) contents:

   - Set the new `.jpx` package location. If you later change the default package in the project properties, you will again raise a `NotFound` error for the `.jpx` file.

     For example:

     ```
     <value n="defaultPackage" v="newpackage.name.here"/>
     ```

   - Fix any `ownerURL` elements in the `ownerMap` that contain references to the old location of the `.jpx` file.

     For example:

     ```
     <url n="ownerURL"
     path="src/newpackage/name/here/Model.jpx"/>
     ```

38

Reusing Application Components

This chapter describes how to package certain ADF components into the ADF Library for reuse in applications. Reusable ADF components are application modules, business components (entity objects, view objects, associations), data controls, task flows, page templates, and declarative components.

This chapter includes the following sections:

- Section 38.1, "About Reusable Components"
- Section 38.2, "Common Functionality of Reusable ADF Components"
- Section 38.3, "Packaging a Reusable ADF Component into an ADF Library"
- Section 38.4, "Adding ADF Library Components into Projects"
- Section 38.5, "Removing an ADF Library JAR from a Project"

38.1 About Reusable Components

In the course of application development, certain components will often be used more than once. Whether the reuse happens within the same application, or across different applications, it is often advantageous to package these reusable components into a library that can be shared between different developers, across different teams, and even across departments within an organization.

In the world of Java object-oriented programming, reusing classes and objects is just standard procedure. With the introduction of the model-view-controller (MVC) architecture, applications can be further modularized into separate model, view, and controller layers. By separating the data (model and business services layers) from the presentation (view and controller layers), you ensure that changes to any one layer do not affect the integrity of the other layers. You can change business logic without having to change the UI, or redesign the web pages or front end without having to recode domain logic.

Oracle ADF and JDeveloper support the MVC design pattern. When you create an application in JDeveloper, you can choose many application templates that automatically set up data model and user interface projects. Because the different MVC layers are decoupled from each other, development can proceed on different projects in parallel and with a certain amount of independence.

ADF Library further extends this modularity of design by providing a convenient and practical way to create, deploy, and reuse high-level components. When you first design your application, you design it with component reusability in mind. If you created components that can be reused, you can package them into JAR files and add them to a reusable component repository. If you need a component, you may look into

the repository for those components and then add them into your project or application.

For example, you can create an application module for a domain and package it to be used as the data model project in several different applications. Or, if your application will be consuming components, you may be able to load a page template component from a repository of ADF Library JARs to create common look and feel pages. Then you can put your page flow together by stringing together several task flow components pulled from the library.

An ADF Library JAR contains ADF components and does not, and cannot, contain other JARs. It should not be confused with the JDeveloper library, Java EE library, or Oracle WebLogic shared library.

Table 38–1 lists the reusable components supported by ADF.

Table 38–1 Oracle ADF Reusable Components

Reusable Component	Description
Data control	Any data control can be packaged into an ADF Library JAR. Some of the data controls supported by Oracle ADF include application modules, Enterprise JavaBeans, web services, URL services, JavaBeans, and placeholder data controls.
Application module	When you are using ADF Business Components and you generate an application module, an associated application module data control is also generated. When you package an application module data control, you also package up the ADF Business Components associated with that application module. The relevant entity objects, view objects, and associations will be a part of the ADF Library JAR and available for reuse.
Business components	Business components are the entity objects, view objects, and associations used in the ADF Business Components data model project. You can package business components by themselves or together with an application module.
Task flows and taskflow templates	Task flows can be packaged into an ADF Library JAR for reuse. If you drop a bounded task flow that uses page fragments, JDeveloper adds a region to the page and binds it to the dropped task flow. ADF bounded task flows built using pages can be dropped onto pages. The drop will create a link to call the bounded task flow. A task flow call activity and control flow will automatically be added to the task flow, with the view activity referencing the page. If there is more than one existing task flow with a view activity referencing the page, it will prompt you to select the one to automatically add a task flow call activity and control flow. If an ADF task flow template was created in the same project as the task flow, the ADF task flow template will be included in the ADF Library JAR and will be reusable.
Page templates	You can package a page template and its artifacts into an ADF Library JAR. If the template uses image files and they are included in a directory within your project, these files will also be available for the template during reuse.
Declarative components	You can create declarative components and package them for reuse. The tag libraries associated with the component will be included and loaded into the consuming project.

You can also package up projects that have several different reusable components if you expect that more than one component will be consumed. For example, you can create a project that has both an application module and a bounded task flow. When this ADF Library JAR file is consumed, the application will have both the application module and the task flow available for use. You can package multiple components into one JAR file, or you can package a single component into a JAR file. Oracle ADF and

JDeveloper give you the option and flexibility to create reusable components that best suit you and your organization.

You create a reusable component by using JDeveloper to package and deploy the project that contains the components into a ADF Library JAR file. You use the components by adding that JAR to the consuming project. At design time, the JAR is added to the consuming project's class path and so is available for reuse. At runtime, the reused component runs from the JAR file by reference. For the procedure to add the JAR manually, see Section 38.4.2, "How to Add an ADF Library JAR into a Project Manually." For the procedure to add the JAR using the JDeveloper Resource Catalog, see Section 38.4.1, "How to Add an ADF Library JAR into a Project using the Resource Palette."

Before you proceed to create reusable components, you should review the guidelines for creating reusable components.

38.1.1 Creating Reusable Components

Creating and consuming reusable components should be included in the early design and architectural phases of software projects. You and your development team should consider which components are candidates for reuse, not only in the current applications but also for future applications and including those applications being developed in other departments.

You and your team should decide on the type of repository needed to store the library JARs, where to store them, and how to access them. You should consider how to organize and group the library JARs in a structure that fits your organizational needs. You should also consider creating standardized naming conventions so that both creators and consumers of ADF Library JARs can readily identify the component functionality.

> **Tip:** If, in the midst of development, you and your team find a module that would be a good candidate for reuse, you can use the extensive refactoring capabilities of JDeveloper to help eliminate possible naming conflicts and adhere to reusable component naming conventions.

38.1.1.1 Naming Conventions

When you create reusable components, you should try to create unique and relevant names for the application, project, application module, task flow, connection, or any other file or component. Do not accept the JDeveloper wizard default names such as Application, Project, ViewController, AppModule, `task-flow-defintion.xml`, or Connection. You want to try to have unique names to avoid naming conflicts with other projects, components, or connections in the application. Naming conflicts could arise from components created in the consuming application and those loaded from other JAR files. Table 38–2 lists the objects that you may be required to rename.

Table 38–2 Example Unique and Relevant Names for Reusable Components

Type	JDeveloper Default	Example
Application	Application	`FusionOrderDemo`
Project	Model	`OrderBookingService`
	ViewController	`StoreFrontUI`
	Project	

Table 38–2 (Cont.) Example Unique and Relevant Names for Reusable Components

Type	JDeveloper Default	Example
Package	Various possibilities. For more information, see Section 38.1.1.1.1.	`oracle.foddemo.storefront`
Application module	AppModule	`StoreServiceAMDataControl`
Connection	Connection1	`oracle_apps_foddb`
Task flow	`task-flow-defintion.xml`	`checkout-task-flow.xml`
Page template	`templateDef`.jspx	`StoreFrontTemplate.jspx`
Declarative Component	`componentDef` `componentDef.jspx`	`FODsuperwidgetDef` `FODsuperwidgetDef.jspx`
ADF Library JAR file	adflib<string or 3-digit random number>*N* For more information, see Section 38.1.1.2.	`adflibStoreFrontService`

38.1.1.1.1 Naming Considerations for Packages

Be aware that some components use the default package name of the project without allowing the name to be explicitly set. In this situation, you must take extra care to avoid package name collisions. You can set the package name in the application creation wizard and you should check the names in the Project Properties dialog afterwards. If you don't set the package name, it will default to a variant of the project name, typically with the first letter being lowercase. For example, a project with the name `Project1` will have a default package name of `project1`. You should manually change the package name to a more unique name before you proceed to build the project.

> **Note:** The basic package naming requirement is that ADF metadata registries (`.dcx`, `.cpx`, and so on) are generated based on the project's package name, and you should avoid metadata naming conflicts between projects that will be combined at runtime.

When you are creating a reusable component's web resource files (such as JSPs, HTMLs, and task flows), you should create them in their own relative directories. When the JAR is deployed into another application, there will be less chance for conflict between the reusable component's files and the consuming application's files.

> **Note:** You can override the staleness period setting of 364 days for all static web application resources in ADF Libraries by adding initialization parameters to the `web.xml` file. For more information, see Appendix A.13, "web.xml."

38.1.1.1.2 Naming Considerations for Connections

Often, several modules in an application will connect to the same data source. You should standardize the connection name to the same data source to avoid confusion because there is only one namespace for connections across the application. This would require coordination with other developers, component producers, and component consumers. For example, if `customers` and `suppliers` both have a connection to the same database, the connection name should be standardized to an agreed upon name, such as `orders_db`.

ADF Library JARs (with connections) may be used in different applications with different connection requirements. ADF Library JAR producers should choose connection names that are at least representative of the connection source, if not the actual standardized connection name. Be aware that consumers of the JAR that was created with connections will be required to satisfy the connection requirements when they add the component to the application.

For example, for a database connection, choosing an endpoint host name is usually not appropriate. The most appropriate name is a complete representative for the schema. Acceptable names for connections are `oracle-appsdb` and `oracle-scottdb`. You should realize that if many reusable components use different names for the same logical connection, then the consumer of the component will have to satisfy each one individually with duplicate information. The consumer will have to supply connection details for several different connection names, when in fact they all refer to the same instance.

38.1.1.1.3 Naming Considerations for Applications with EJB Projects

If an application has both EJB projects and a web application project with data binding, you should check to see that the EJB component names are not in conflict with any other web application project component names. The EJB project components may have global scope because the project is automatically added to the global class path of all web projects in the application. The web application project may mistakenly access a component with the same name in the EJB project rather than within its own project.

For example, if both an EJB project and a web-based project have a `test1.jspx` page, when the web-based project is run, it may try to run the EJB project `test1.jspx` page.

At runtime, JDeveloper detects if there are EJB projects and web application projects with data binding in the same application. If there are both types in the application, when the project is run and the server starts up, a warning message will appear in the Log window.

38.1.1.2 The Naming Process for the ADF Library JAR Deployment Profile

Before you package the project, you must create a deployment profile with the name and path for the JAR file. You should choose a name that follows you and your development team's naming convention and that is descriptive of the function of the component. You should realize that the consuming project may also include other JAR files from other software authors.

For example, if the component is a task flow for self-service paying, you might name it `mycompany.hcm.pay.selfservice.taskflows.jar`. Other examples are `oracle.apps.hcm.pay.model.jar` and `mycompany.hcm.pay.model.overrides.jar`.

When you create the deployment profile, JDeveloper will present a default name with the following format:

`adflib`*`identifierN`*

Where *identifier* is random number up to three-digits and *N* is a number that starts with 1 and increments by 1 for each iteration of the profile.

For example, the default name for a profile may be `adflib7491(`random number is `749` and suffix is `1)`. The default name for the another profile may be `adflib51` (random number is `5` and suffix is `1)`.

You can change this default name to a name of your choice.

38.1.1.3 Keeping the Relevant Project

When you are creating reusable components, you should eliminate any projects that are not relevant to the reusable component. For example, if you want to create a reusable application module, you would need a data model project, but you would not need a user interface project. In this instance, if you had created an application with both a data model and a view-controller project, you could delete the user interface project. Of course, you should rename the default name `Model` to something more relevant, such as `StoreFrontService`. Similarly, if you are creating a reusable task flow that is not databound, you can delete any data model project from the application.

38.1.1.4 Selecting the Relevant Feature

If you know the technology scope of your consuming projects, you can design your component with technologies that will be compatible. For example, if the consuming application uses only standard JSF Faces, then it may not be compatible with a declarative component that is built with ADF Faces.

When you create your application, you can define the features using the Create Application Wizard by selecting from the application template.

After the project has been created, you can define the technology using the Features page of the Project Properties dialog.

38.1.1.5 Selecting Paths and Folders

If you are using the file system to store your ADF Library JARs, you should select file system locations that can function as repositories. You may want to put groups of JARs into a common directory folder, for example, `C:\ADF11\jdev\DevTeamADFRepository`. If your team or organization plans to share ADF Library JARs, you should consider setting up network-accessible repository folders, directories, or services. The Resource Palette has provisions to connect to different repository sources and make multiple connections. For more information about accessing ADF Library JARs using the Resource Palette, see Section 38.2.2, "Using the Resource Palette."

38.1.1.6 Including Connections Within Reusable Components

If the project you are packaging into an ADF Library JAR includes a connection, that information can be included in the JAR and may be available to the consuming project. Oracle ADF uses connection architecture which defines a connection as two parts, the connection name and the connection details (or endpoint definition). JDeveloper will present the producer of the JAR the option to package the connection name only or to include connection details with the connection name.

If a connection is present in the project, the packaged ADF Library JAR will contain a `jar-connections.xml` file and a `jar-adf-config.xml` file. They will be added to the `META-INF` directory. The `jar-connections.xml` file contains the connection name and other relevant connection information. The `jar-adf-config.xml` file stores the information about the credentials used for the connections. If connection

credentials were also specified, then a `jar-credential-jazn-data.xml` will also be included for the credential store. You can select the individual connections or connection types that you want to be packaged with the ADF Library JAR when you create the ADF Library JAR deployment profile. You make connection selections in the Create or Edit ADF Library Deployment Profile Properties dialog Connections page, as described in Section 38.3.1, "How to Package a Component into an ADF Library JAR."

When an ADF Library JAR is being added to a project, JDeveloper checks for conflicts between the application's connections and the JAR's connections. A dialog will be presented to allow you to decide whether to proceed with adding the JAR to the project. Connections defined in the ADF Library JAR may be added to the consuming project, depending on the following conditions:

- If the connection defined in the JAR is fully configured and there are no connection name conflicts with the consuming project, a new connection will be added.
- If the connection defined in the JAR is partially configured, a new connection will be added to the consuming project but it must be configured before use. Connection dialogs may appear to allow the user to enter connection information. This partially configured connection may be indicated by an incomplete icon.
- If the connection defined in the JAR has the same name as the application's connection, it will not be added to the project. The application's existing connection will be used.
- If the connection defined in the JAR has the same name as the application's connection but is of different type, the JAR's connection will not be added to the project. For example, if a database connection in the JAR and a URL connection in the application have the same name, the database connection will not be added to the application.

For instructions on how to add an ADF Library to a project that includes connections, see Section 38.4.1, "How to Add an ADF Library JAR into a Project using the Resource Palette."

> **Note:** In the process of adding the ADF Library to a project, you will see one message displayed in the Log window for each connection defined in the JAR. The messages indicate the merge action taking place in the project.

38.1.2 Reusable ADF Components Use Cases and Examples

Packaging projects into ADF Library JARs allows components to be reused multiple times in the same or in different applications. Different development teams can create and consume projects and create libraries of reusable modules. ADF Library JARs support the M-V-C design pattern. You can create ADF Libraries and access them using drag and drop from the Resource Palette.

For instance, you can package up an application module into an ADF Library JAR and place it in a public folder for other team to use. Different implementation teams may access that JAR to create their applications. They may use the application module to create different pages for their use instead on having to create their own application modules. When there is a change in the model, the original developer can repackage the application module and republish the ADF Library JAR. The other implementation teams can update their application using the updated ADF Library.

38.1.3 Additional Functionality for Reusable ADF Components

You may find it helpful to understand other ADF features before you work with reusable ADF components. Following are links to other functionality that may be of interest.

- You can package business components for reuse. For more information on application modules, see Chapter 3, "Getting Started with ADF Business Components."
- You can package application modules for reuse. For more information on application modules, see Chapter 9, "Implementing Business Services with Application Modules."
- You can package task flows for reuse. For more information on task flows, see Chapter 18, "Getting Started with ADF Task Flows."
- You can package page templates for reuse. For more information on task flows, see Chapter 24, "Getting Started with Your Web Interface."
- You can package page templates for reuse. For more information, see the "Using Declarative Components" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

38.2 Common Functionality of Reusable ADF Components

When you package or consume an ADF Library JAR, you also include and consume the extension libraries that are associated with the project. When you consume an ADF Library, you can use the Resource Palette.

38.2.1 Using Extension Libraries

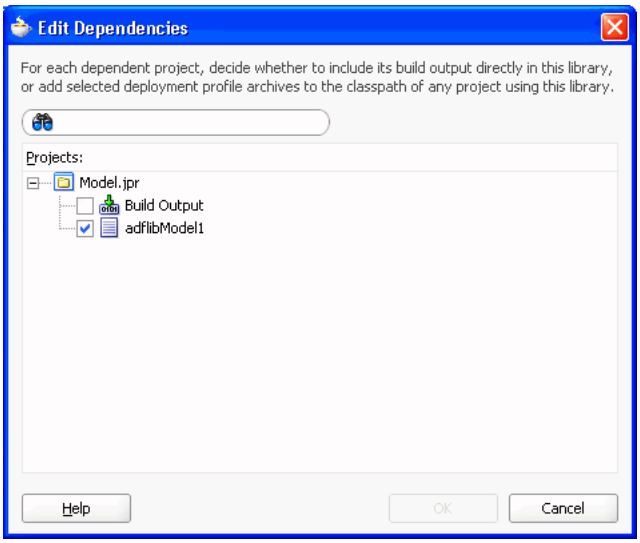
An ADF project usually includes a list of extension libraries that it needs to run. These libraries are loaded in the class path of the project. You can view a project's dependent libraries by selecting the **Libraries and Classpath** node of the Project Properties dialog. Some of the libraries that may appear are JSP Runtime, ADF Page Flow Runtime, Connection Manager, and Oracle JDBC.

When a project is packaged into an ADF Library JAR, its extension libraries are packaged with it. And when an ADF Library JAR is being consumed by another project, JDeveloper automatically resolves any extension library conflicts between them. During the consuming process, JDeveloper checks to see whether the consuming project already has the extension libraries of the ADF Library JAR in its class path and loads only those libraries that it does not have. For example, if JSP Runtime already exists in the consuming project, it will not be loaded again if the ADF Library JAR also includes it. The consuming project's extension libraries will be a union of its own libraries and the libraries in the ADF Library JAR.

If the project you want to package into an ADF Library has a dependent project, you can include the dependent project's extension libraries directly in the JAR or, if the dependent project has a deployment profile, you can add the dependent project's JAR to the ADF Library. For more information about setting up the deployment process, see Section 38.3.1, "How to Package a Component into an ADF Library JAR."

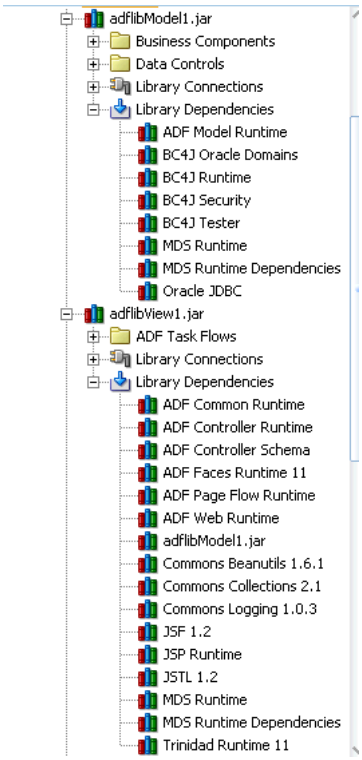
For example, suppose that project `View` is being packaged into ADF Library `adflibView1.jar` and that has a dependency on the `Model` project. For project `View`, the `Model` project is a dependent project with the deployment profile option (`adflibmodel1`) selected, as shown in Figure 38–1.

Figure 38–1 Edit Dependencies Dialog with Deployment Profile Option



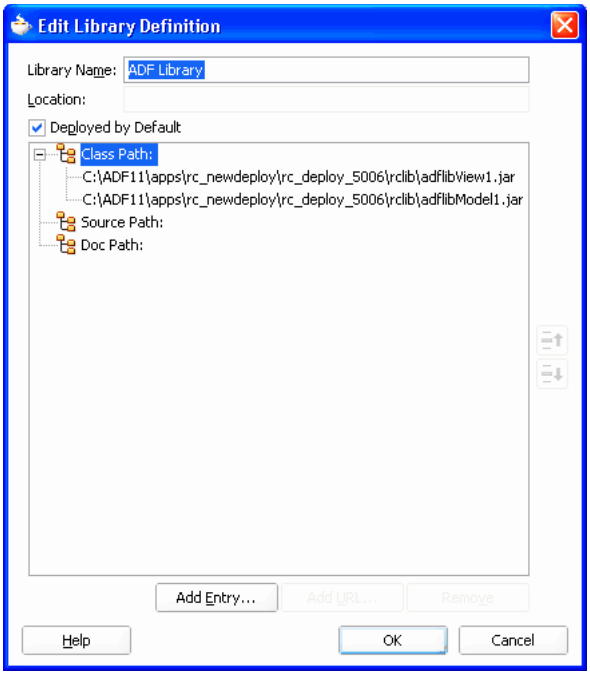
When the deployment profile is selected, the dependent project's JAR file will be added to the ADF Library JAR. As a result, the extension libraries of the dependent project will also be made available to any consuming project. In Figure 38–2, the ADF Library being packaged, `adflibView1.jar`, includes the dependent `adflibModel.jar` as listed under the Library Dependencies node.

Figure 38–2 Resource Palette Showing adflibView1.jar and adflibModel.jar Extension Libraries



When you select `adflibView1.jar` in the Resource Palette and choose **Add to Project** from the context menu, both `adflibView1.jar` and `adflibModel.jar` will be added to the consuming project's class path.

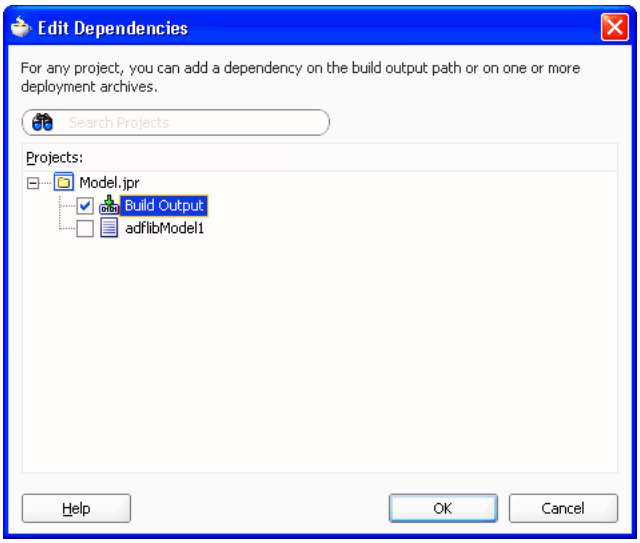
Figure 38–3 Class path of Consuming Project Showing ADF Library JAR and Dependent Project Library JAR



Alternately, you can include the dependent project's artifacts and extension libraries directly into the ADF Library JAR.

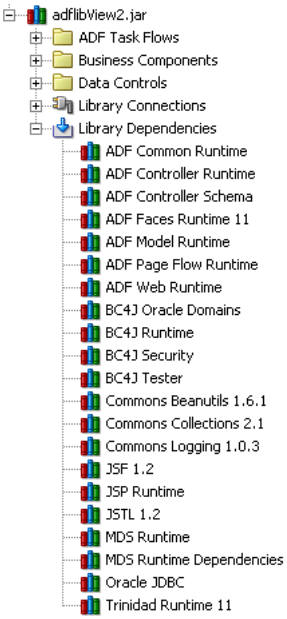
For example, project `View` can also be packaged into ADF Library `adflibView2.jar`. It also has a dependency on the `Model` project. But in this second deployment profile, the `Model` project is a dependent project with the **Build Output** option selected (as opposed to the deployment profile (`adflibmodel1`) being selected), as shown in Figure 38–4.

Figure 38–4 Edit Dependencies Dialog Used to Built adflibview2.jar



When **Build Output** is selected, the dependent project's classes and extension libraries will be added directly to the ADF Library JAR. Figure 38–5 shows the ADF Library `adflibView2.jar`, which includes artifacts of the `Model` project and its extension libraries. Note that the extension libraries under the `adflibView2.jar` Library Dependencies node are the same as the combined extension libraries under the `adflibView1.jar` and `adflibModel.jar` shown in Figure 38–2.

Figure 38–5 Resource Palette Showing adflibView2.jar Extension Libraries



How you decide to package the dependent project depends on how you intend the ADF Library to be used. Including the dependent project as a JAR may be advantageous when the dependent project is itself a reusable component, or when it is a dependent project for several projects that will be packaged into an ADF Library JAR. In the example, the dependent project `Model` may be a dependent project for several view projects. On the other hand, packaging the dependent project as **Build Output** is straightforward and eliminates the need for multiple JARs.

Note: If you are creating an ADF Library JAR that is included in a JDeveloper extension library, you should include the additional manifest `JDevLibrary: extension_library_name` entry in the JAR. When you use **Add to Project** from the Resource Palette to add a ADF Library JAR that has a `JDevLibrary` manifest entry, the JDeveloper extension library containing this JAR will be added instead of the ADF Library JAR itself.

38.2.2 Using the Resource Palette

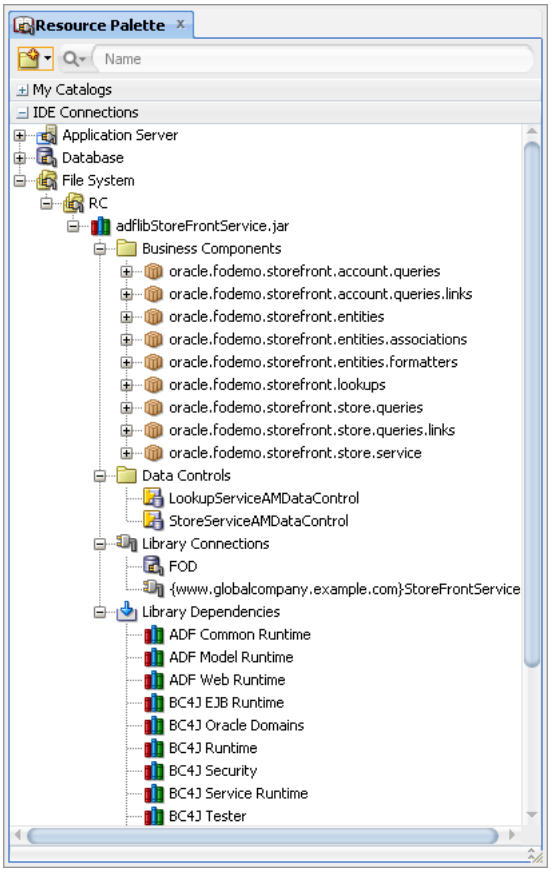
ADF Library JARs can be packaged, deployed, discovered, and consumed like any other Oracle Library component. Creating an ADF Library JAR is the action of packaging all the artifacts (and additional control files) of a project into a JAR. Consuming a reusable component from an ADF Library JAR is the action of loading that ADF Library JAR into the project's set of libraries.

However, the easiest way to manage and use ADF Library JAR components is by using JDeveloper's Resource Palette. With the Resource Palette, developers who want to consume reusable components can easily find and discover available components and add them to their projects. The Resource Palette provides search and browse functions across different data management systems to locate the component. It provides multiple connections to access different sources. It has a structure tree view for displaying different connections and the ADF Library JAR component types. Figure 38–6 shows the Resource Palette window with three file system connections, `RCconnect`, `rcconnect2`, and `StoreFront`. In this example, `StoreFront` contains the ADF Library component, `adflibStorFrontService.jar`.

The Resource Palette tree structure displays each JAR as subcategories. Separate nodes are created for each type of reusable component. For example, application modules are under the Data Controls node, and task flows are under the ADF Task Flows node.

The tree structure for the ADF Library JAR lists any connection information under a Library Connections node and lists all the producing project's extension libraries under the Library Dependencies node.

Figure 38–6 Resource Palette Showing ADF Library Structure



38.3 Packaging a Reusable ADF Component into an ADF Library

Once you have decided that a certain component or components can be reused, create an application and a project to develop that component. Follow the guidelines in Section 38.1.1, "Creating Reusable Components" to name your application, project, package, and other objects and files. A project corresponds to one ADF Library JAR. If you create multiple projects and want to reuse components from each of the projects, you may need to create an ADF Library JAR for each project. In other situations, you may be able to involve multiple components under one project to create a single ADF Library JAR. For example, you may be able to create application modules, business components (entity object, view objects, associations), task flows, and page templates all under one project and create one ADF Library JAR.

Creating an ADF Library JAR involves compiling the project and validating the components, creating a resource service file, control files, an `adflibREADME.txt`, and adding the relevant project files into a JAR. For more information about the ADF Library JAR, see Section 38.3.2, "What Happens When You Package a Project to an ADF Library JAR."

If you are packaging a component that itself uses another ADF Library component, the final consuming project must have both ADF Library JARs added to the project. For example, say you created a reusable task flow that contains tables dropped from a data control in another ADF Library JAR. When you add the task flow from an ADF Library JAR into a consuming project, that project will also require the data control ADF Library JAR.

If you are packaging a component that has dependent JARs, such as third-party JARs, you have two options:

- If the consuming environment has control over the placement of JARs, you can place the ADF Library JAR and its dependent JARs in a JDeveloper Extension Library. The advantage of using a JDeveloper Extension Library is that it does not clutter the consuming project with specific references to the dependent JARs.
- If using JDeveloper Extension Library is not possible, you can place the dependent JARs in the same location as the ADF Library JAR and include a manifest classpath entry for each dependent JAR.

38.3.1 How to Package a Component into an ADF Library JAR

To package up a reusable component, you first create a deployment profile that specifies the archive type, the name of the JAR file, and the directory path where the JAR will be created. Then you deploy the project using the deployment profile.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you package an ADF component. For more information, see Section 38.3, "Packaging a Reusable ADF Component into an ADF Library."

You may also find it helpful to understand functionality that can be used with packaging an ADF component. For more information, see Section 38.1.3, "Additional Functionality for Reusable ADF Components."

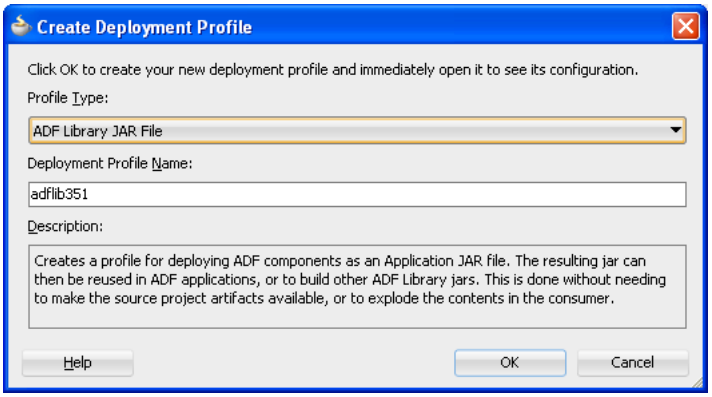
You will need to complete this task:

Create a project and determine whether it has dependent projects and JAR that need to be packaged. For more information, see Section 38.3, "Packaging a Reusable ADF Component into an ADF Library."

To package and deploy a project into the ADF Library JAR:

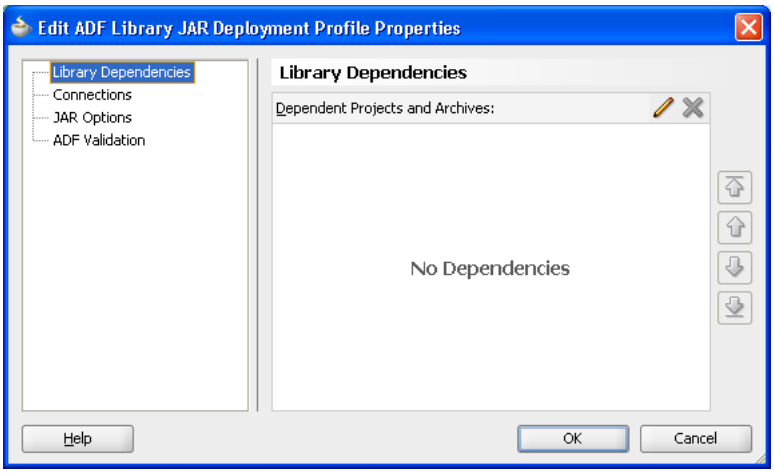
1. In the Application Navigator, double-click the project that contains the component you want to make reusable.
2. In the Project Properties dialog's left pane, select **Deployment** and then click **New**.
3. In the Create Deployment Profile dialog, select **ADF Library JAR file** for **Profile Type**, enter a name or accept the default name for **Deployment Profile Name**, and click **OK**.

Figure 38–7 Create Deployment Profile Dialog with Default Name



4. In the Project Properties dialog, select the deployment profile and click **Edit**.
5. In the Edit ADF Library JAR Deployment Profile Properties dialog, select the **Library Dependencies** node, as shown in Figure 38–8.

Figure 38–8 ADF Library JAR Deployment Profile Properties Dialog



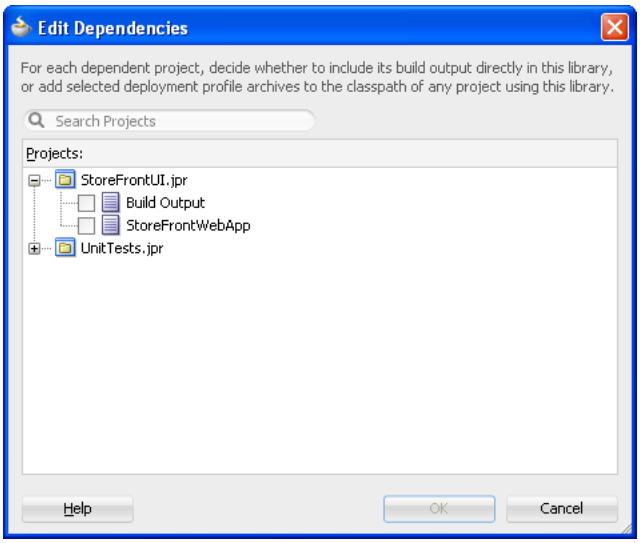
The Library Dependencies pane shows a list of dependent projects for the project being packaged. You can add the dependent project's build output directly into the packaging project's library, or you can add selected deployment profile archives to the class path.

a. To add dependent projects, click the **Edit** icon to bring up the Edit Dependencies dialog, as shown in Figure 38–9.

   If you select **Build Output**, the dependent project's extension libraries will be added directly to the ADF Library JAR. If you select deployment profile, the dependent project's JAR file (which includes its own extension libraries) will be added to the ADF Library JAR. For more information about library dependencies, see Section 38.2.1, "Using Extension Libraries."

   In this example, the `StoreFrontService` project can be set as a dependency only as **Build Output**. However, the `StoreFrontUI` project can be set as a dependency either as **Build Output**, or as a deployment profile (`StoreFrontWebApp`).

Figure 38–9 ADF Library Deployment Edit Dependencies Dialog



b. For each dependent project, select the **Build Output** node for the project or select the dependent profile and click **OK**.

6. In the Edit ADF Library JAR Deployment Profile Properties dialog, select the **Connections** node, as shown in Figure 38–10.

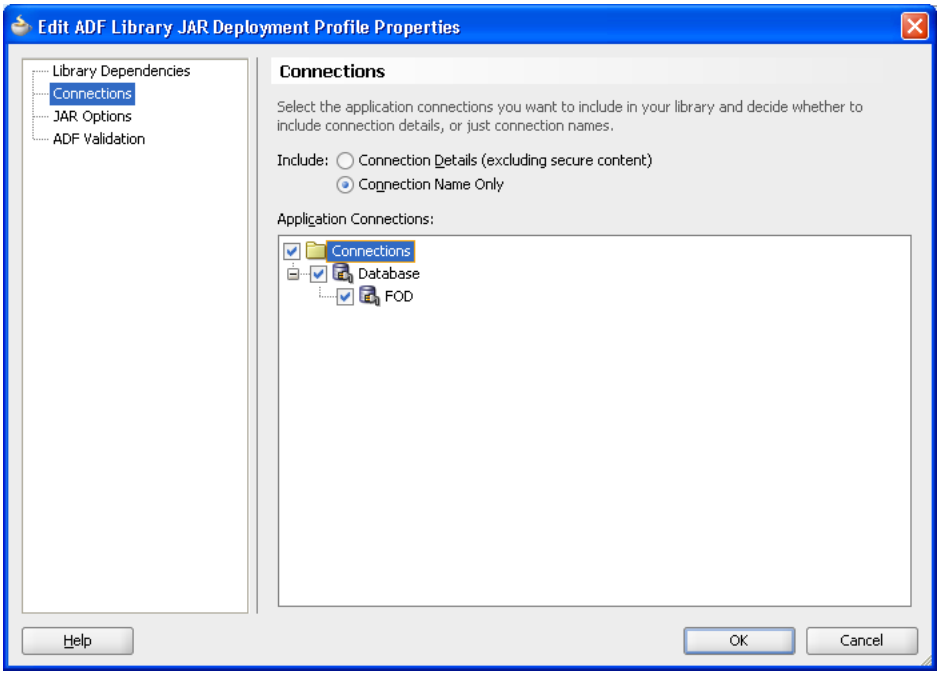
   You can select:

   - **Connection Details (excluding secure content)**: If the project has a connection, select this checkbox if you want to add any available connection details in addition to the connection name. For more information, see Section 38.1.1.6, "Including Connections Within Reusable Components."
   - **Connection Name Only**: Select this checkbox if you want to add the connection name without any connection details such as security.

   Note: The connection options available depends on the JDeveloper role. **Connection Details (excluding secure content)** and **Connection Name Only** are the selections available for the Studio Developer role. If your JDeveloper is set to a different role, you may have different options and defaults.

   In the Applications Connections tree structure, select the checkbox for the level of connection you want to include.

Figure 38–10 Connections Page of the Edit ADF Library JAR Deployment Profile Properties Dialog

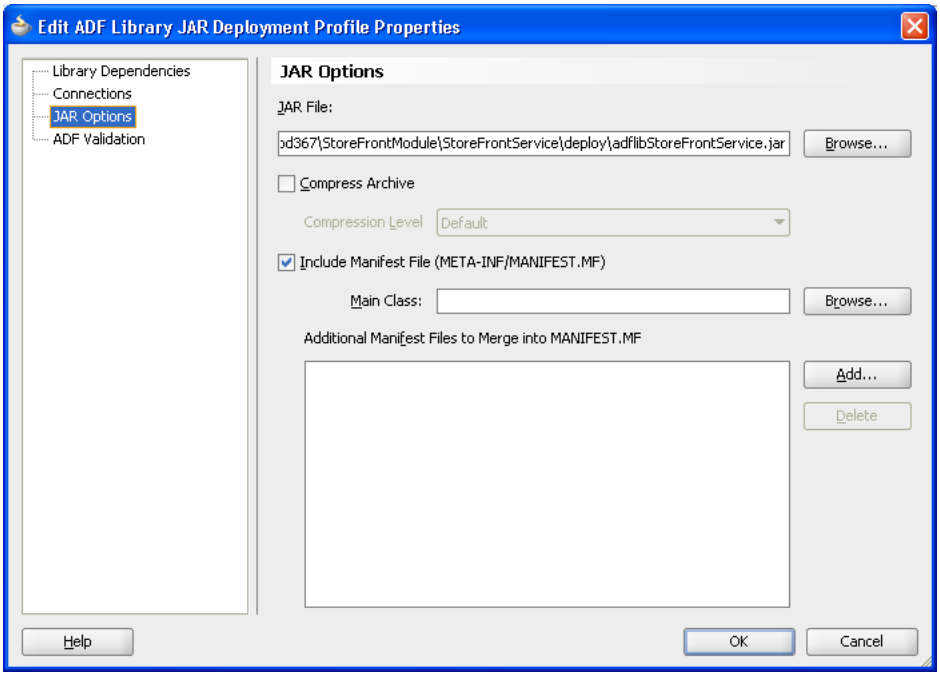


7. In the Edit ADF Library JAR Deployment Profile Properties dialog **JAR Options** node, as shown in Figure 38–11, verify the default directory path or enter a new path to store your ADF Library JAR file.

   If the ADF Library JAR is to be included in a JDeveloper extension library, create a text file with a `JDevLibrary: extension_library_name` entry and place the file in the project root directory. Click **Add** to locate and merge that file into the `Manifest.mf` file.

   When you create your manifest text file, make sure the `JDevLibrary` entry starts in column 1 with a space after the colon, and that there is a blank line at the end of the file.

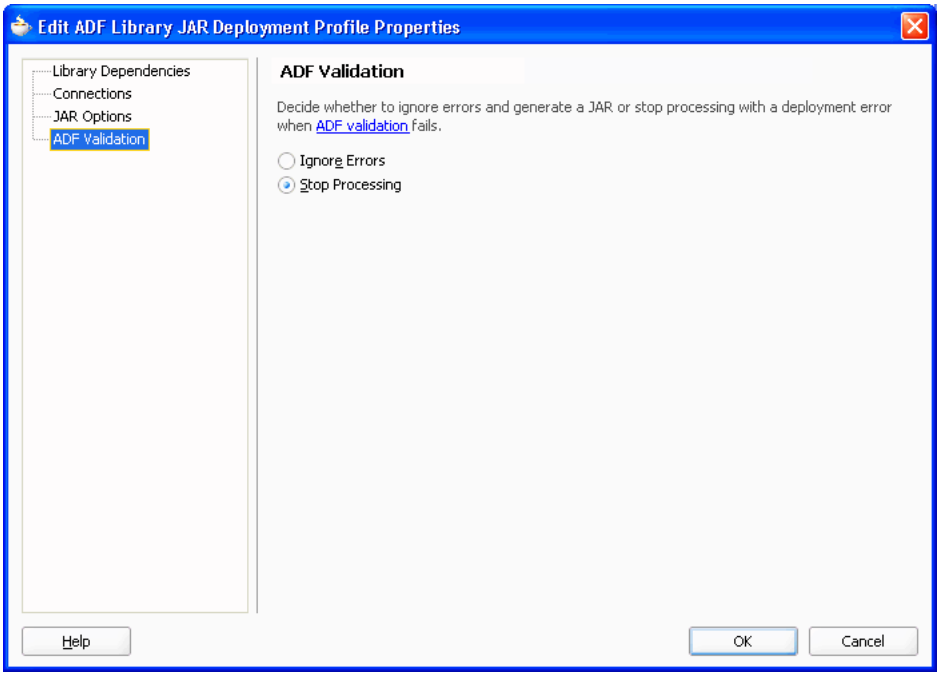
Figure 38–11 ADF Library JAR Deployment Profile Properties Dialog JAR Option



8. In the Edit ADF Library JAR Deployment Profile Properties dialog **ADF Validation** node, select:

   - **Ignore Errors**: Select this option to create the JAR file even when validation fails. This is the default option.
   - **Stop Processing**: Select this option to stop processing when validation fails.

Figure 38–12 ADF Library JAR Deployment Profile Properties Dialog ADF Validation



9. Click **OK** to finish setting up the deployment profile.

10. In the Application Navigator, right-click the project and choose **Deploy > *deployment***, where ***deployment*** is the name of the deployment profile.
11. In the Deploy dialog Deployment Action page, click **Next** and then click **Finish**.

    JDeveloper will create the ADF Library JAR in the directory specified in Step 7. You can check that directory to see whether the JAR was created.

38.3.2 What Happens When You Package a Project to an ADF Library JAR

When you deploy the library JAR, JDeveloper packages up all the necessary artifacts, adds the appropriate control files, generates the JAR file, and places it in the directory specified in the deployment profile. During deployment, you will see compilation messages in the Log window.

When you deploy a project into an ADF Library JAR, JDeveloper performs the following actions:

- Package the HTML root directory artifacts into the JAR. When the JAR is added to the consuming project, JDeveloper will make the reusable component's `public_html` resources available by adding it to the class path.
- Add the `adfm.xml` file to the JAR. If there are multiple `META-INF/adfm.xml` files in the workspace, only the `adfm.xml` in the project being deployed is added. JDeveloper modifies this file to include relevant content from any dependent project's `adfm.xml` file.
- Add a service resources file, `oracle.adf.common.services.ResourceService.sva`, into the `META-INF` directory of the JAR. The addition of this file differentiates an ADF Library JAR file from standard JAR files. This file defines the service strategies of the JAR and allows the Resource Palette to properly discover and display the contents of the JAR.
- Add a `Manifest.mf` file to the JAR. The `Manifest.mf` file is used to specify dependencies between JAR files, and to determine whether to copy and include the contents of a JAR file or to reference it. JDeveloper will create a default manifest file. For example:

  ```
  Manifest-Version: 1.0
  ```

- Adds a `jar-connections.xml` file to the JAR for components that require a connection and that use the connections architecture. Note that in the consuming application, connection information in configuration files that are defined in the class path and accessible at runtime may be merged together. If the same connection is named multiple times in the class path, the connection in the main application will be given priority.

Different types of reusable components have different artifact files and different entries in the service resource file.

38.3.2.1 Application Modules

For application modules, JDeveloper adds these control files to the JAR: `oracle.adf.common.services.ResourceService.sva`, `Manifest.mf`, `adfm.xml`, and the business components `.jpx` file. The service resource file for an application module includes entries for the business components associated with the application module, as well as an entry for the application module data control.

The `jar-connections.xml` file may appear for components that use the connection architecture and that contain connection information regarding the data source.

38.3.2.2 Data Controls

For data controls such as placeholder data controls, JDeveloper includes three control files in the JAR: `oracle.adf.common.services.ResourceService.sva`, `Manifest.mf`, and `DataControl.dcx` file. Data controls are used when the data source is not based on ADF Business Components, and so business components are not included in the JAR file, as is the case in an application module JAR file. The service resource file for a standard data control has an entry for the data control.

The JAR also includes the `Datacontrol.dcx` file from the project to describe the data control type.

38.3.2.3 Task Flows

For task flows, JDeveloper includes three control files in the JAR: `oracle.adf.common.services.ResourceService.sva`, `Manifest.mf`, and `task-flow-registry.xml`. The service resource file for a task flow includes an entry that indicates that one or more task flows are in the JAR.

38.3.2.4 Page Templates

For page templates, JDeveloper includes two control files in the JAR: `oracle.adf.common.services.ResourceService.sva` and `Manifest.mf`.

38.3.2.5 Declarative Components

For declarative components, JDeveloper includes two control files in the JAR: `oracle.adf.common.services.ResourceService.sva` and `Manifest.mf`.

38.3.3 How to Place and Access JDeveloper JAR Files

If you have JAR files that can be reused by other projects, such as third-part JAR files, you can use JDeveloper to place them in an accessible location and create a library file (`.library`) to designate them. The consumer can use JDeveloper to navigate to this location and add the JAR to the project.

An ADF Library JAR contains ADF components and does not and cannot contain other JARs.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you package an ADF component. For more information, see Section 38.3, "Packaging a Reusable ADF Component into an ADF Library."

You may also find it helpful to understand functionality that can be used with packaging an ADF component. For more information, see Section 38.1.3, "Additional Functionality for Reusable ADF Components."

You will need to complete this task:

- Place the JAR files in a accessible location for both the producer and the consumer of these JAR files. For example, the directory may be on a network drive where other shared files are located. For more information, see Section 38.3, "Packaging a Reusable ADF Component into an ADF Library."

To place and access a JDeveloper LIbrary JAR:

1. From the main menu, choose **Tools > Manage Libraries.**
2. In the Manage Libraries dialog, click **Load Dir**.

3. In the Load Directory dialog, select the directory where the secondary JAR files are located.
4. In the Manage Libraries dialog, click **New**.
5. In the Create Library dialog, enter a library name and click **Add Entry**.

   You have created a library file (with a `.library` extension). You should place library files in source control systems.
6. In the Select Path Entry dialog, select the JARs you want to add and click **Select**.
7. In the Create Library dialog, be sure that the **Deployed by Default** option is set correctly for your JAR, click **OK** and then click **OK** again.
8. In the consuming project.
   a. From the main menu, choose **Tools > Manage Libraries.**
   b. In the Manage Libraries dialog, click **Load Dir**.
   c. In the Load Directory dialog, select the directory where the secondary JAR files are located and click **OK**. This should be the same location specified in step 3.
   d. Right-click the project and select **Project Properties**.
   e. In the Project Properties window, select **Libraries and Classpath** and click **Add Library**.
   f. In the Add Library dialog, select the library, click **OK** and then click **OK** again.

38.4 Adding ADF Library Components into Projects

After ADF Library JARs are created, they must be distributed to the developers who will use these JARs. Distributing the ADF Library JARs may include putting the JARs into a network file system to be searched, browsed, and discovered. It may include using other forms of data store or services to access and retrieve these JARs. Since ADF Library JARs are simply binary files, they can distributed like any other file such as ftp and email.

Once you have access to the ADF Library JARs, you can use JDeveloper to access them and add them to your consuming projects. Using the JDeveloper Resource Palette is the easiest and most efficient way. You can also use JDeveloper to manually add the JARs into the project by entering them into the class path.

When a project is packaged into an ADF Library JAR, it captures the list of dependent JARs the project needs for deployment and run time. This list is based on the project's dependent profiles and the information in the project's Libraries and Classpath with **Deployed By Default** selected. When the ADF Library JAR is added to the consuming project, this list of dependent JARs is placed in a library called *ADF Library Dependencies*. This is a locked library in the consuming project and its content will not be shown in the Data Controls panel, Component Palette, or other places in JDeveloper. The library is maintained and updated as different ADF Library JARs are added and deleted from the project. If the dependencies have changed (for example, the producer project was rebuilt), then you can refresh the dependencies in the consuming project using a JDeveloper menu command.

Once reusable components have been added, how they are used depends on the type of component.

38.4.1 How to Add an ADF Library JAR into a Project using the Resource Palette

You can use the Resource Palette to search, discover, and add the ADF Library JAR to your project.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you package an ADF component. For more information, see Section 38.4, "Adding ADF Library Components into Projects."

You may also find it helpful to understand functionality that can be used with packaging an ADF component. For more information, see Section 38.1.3, "Additional Functionality for Reusable ADF Components."

You will need to complete this task:

Create an ADF Library JAR in a repository folder. If you do not already have a Resource Palette connection to this repository, you must know the location of this folder. For more information, see Section 38.4, "Adding ADF Library Components into Projects."

To add a component to the project using the Resource Palette:

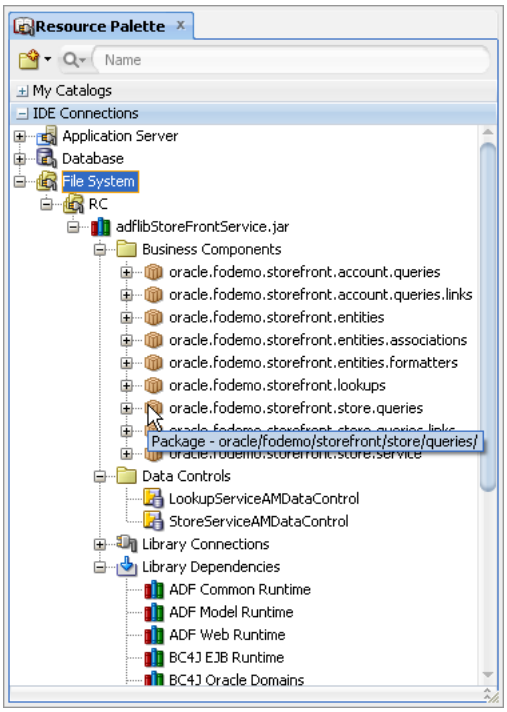
1. From the main menu, choose **View > Resource Palette**.
2. In the Resource Palette, click the **New** icon, and then choose **New Connection > File System**.
3. In the Create File System Connection dialog, enter a name and the path of the folder that contains the JAR.

   For file path guidelines, see Section 38.1.1.5, "Selecting Paths and Folders."
4. Click **OK**.

   The new ADF Library JAR appears under the connection name in the Resource Palette.
5. To examine each item in the JAR structure tree, use tooltips. The tooltip shows pertinent information such as the source of the selected item.

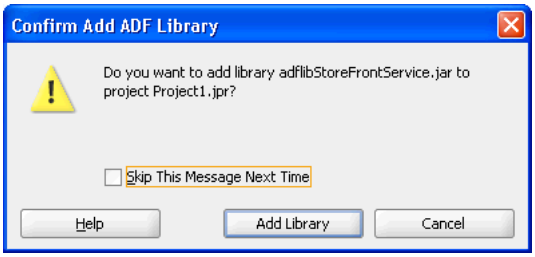
   Figure 38–13 shows a Resource Palette with a tooltip message that shows package information for a business component.

Figure 38–13 Tooltip Message for a Connection in the Resource Palette



6. To add the ADF Library JAR or one of its items to the project, right-click the item and choose **Add to Project**. In the confirmation dialog that appears, click **Add Library**, as shown in Figure 38–14.

Figure 38–14 Confirm Add ADF Library Dialog



If you had previously added that library JAR, you will get a Confirm Refresh ADF Library dialog asking you whether you want to refresh the library.

7. If the ADF Library JAR has changes to its dependencies, you can refresh the project with the new changes. From the Application Navigator, right-click the project and select **Refresh ADF Library Dependencies in *project*.jpr**.

For application modules and data controls, you have the option to drag and drop the application module or data control from the Resource Palette into the Data Controls panel.

Note: JDeveloper will load whichever ADF Library JAR extension libraries are not already in the consuming project. Extension libraries from the ADF Library's dependent JARs will also be checked and loaded if not already part of the consuming project. For more information, see Section 38.2.1, "Using Extension Libraries"

38.4.2 How to Add an ADF Library JAR into a Project Manually

You can add an ADF Library JAR in the same way as you would other library JARs.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you package an ADF component. For more information, see Section 38.4, "Adding ADF Library Components into Projects."

You may also find it helpful to understand functionality that can be used with packaging an ADF component. For more information, see Section 38.1.3, "Additional Functionality for Reusable ADF Components."

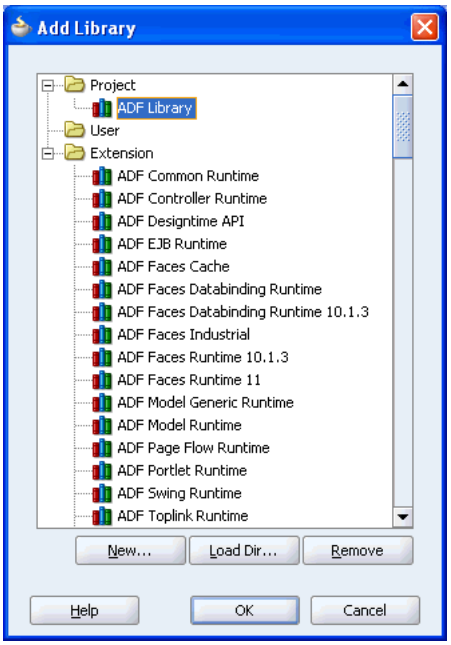
You will need to complete this task:

Create an ADF Library JAR. For more information, see Section 38.4, "Adding ADF Library Components into Projects."

To add a component to a project manually:

1. In the Application Navigator, double-click the project to which the component is to be added.
2. In the Project Properties dialog, select the **Libraries and Classpath** node and then click **Add Library**.
3. In the Add Library dialog, click **New**.
4. In the Create Library dialog, select **Project** in the **Location** dropdown list and enter a name for the ADF Library. The preferable name is "ADF Library". Select **Deploy by Default**, and click **Add Entry**.
5. In the Select Path Entry dialog, enter or browse to the ADF Library JAR file and click **Select**.
6. The Create Library dialog reappears with the path of the JAR file filled in under the **Class Path** node. Click **OK**.
7. The Add Library dialog reappears with the ADF Library entry filled in under the **Project** node. Click **OK**.

Figure 38–15 Add Library Dialog



8. The Project Properties dialog reappears with the JAR file added to the list of libraries. Click **OK**.

38.4.3 What Happens When You Add an ADF Library JAR to a Project

When you add an ADF Library JAR to a project, either by using the Resource Palette or by manually adding the JAR, an ADF Library definition is created in the project. The ADF Library JAR file will be added to the class path of the project. The project will access and consume the components in the JAR by reference.

By default, the **ADF Library Deployed by Default** option in the Create Library dialog is set to true. If this option is set, when the application or module is further archived or built into a WAR file, the contents of the ADF Library JAR will be copied to that archive or WAR file. If the Deploy by Default option is not set, then the JARs in the ADF Library must be loaded in some other way, such as by deploying them in a shared library.

Figure 38–16 shows the empty Data Controls panel for a consuming project before the ADF Library was added.

Figure 38–16 Data Controls Panel of the Consuming Project

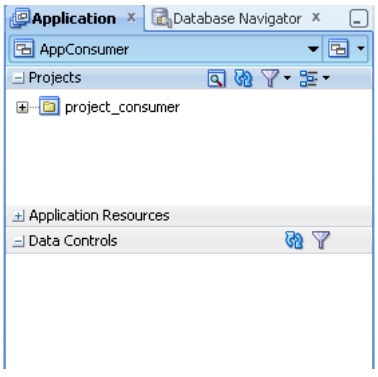


Figure 38–17 shows the `adflibStoreFrontService` ADF Library being added to the consuming project.

Figure 38–17 Adding the ADF Library JAR to the Project

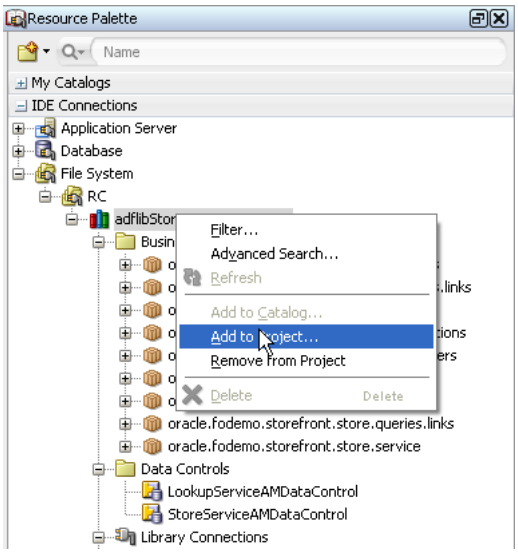
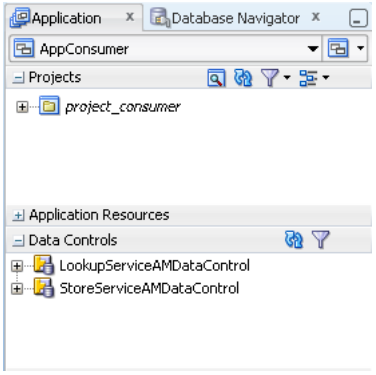


Figure 38–18 shows several data controls from the ADF Library added to the Data Controls panel in the consuming project.

Figure 38–18 Consuming Project Data Controls Panel with Added Application Modules



After adding the ADF Library JAR, you may notice some changes to some of the JDeveloper windows. These changes are different depending on the type of components being added. Table 38–3 lists the effects on several JDeveloper windows.

Table 38–3 JDeveloper Window After Adding an ADF Library

Added Component	Data Controls Panel	Creation Wizards	Component Palette
Data controls	Data control appears.		
Application module	Application module appears.		

Table 38–3 (Cont.) JDeveloper Window After Adding an ADF Library

Added Component	Data Controls Panel	Creation Wizards	Component Palette
Business components		Entity objects available in view object creation wizard. View objects in JAR also available for use.	
Task flows			Task flows appear in Component Palette.
Page template		Page template available during JSF creation wizard.	
Declarative components			Tag library appears in Component Palette. Declarative component appears in the list.

38.4.4 What You May Need to Know About Using ADF Library Components

Although the procedure to add an ADF Library JAR to a project is standardized, the component type determines where it appears in JDeveloper and how it can be reused.

38.4.4.1 Using Data Controls

When you add a data control to a project, the data control appears in the Data Controls panel. If you are using the Resource Palette, you have the option of dragging and dropping the data control from the Resource Palette onto the Data Controls panel, and then dragging and dropping from the Data Controls panel onto the page.

38.4.4.2 Using Application Modules

When you add an application module to a project, the application module appears in the Data Controls panel. If you are using the Resource Palette, you have the option of dragging and dropping the application module item from the Resource Palette onto the Data Controls panel, and then dragging and dropping from the Data Controls panel onto the page.

Application modules are associated with business components. When the reusable application module was packaged, the JAR includes the business components used to create the application module. These components will be available for reuse.

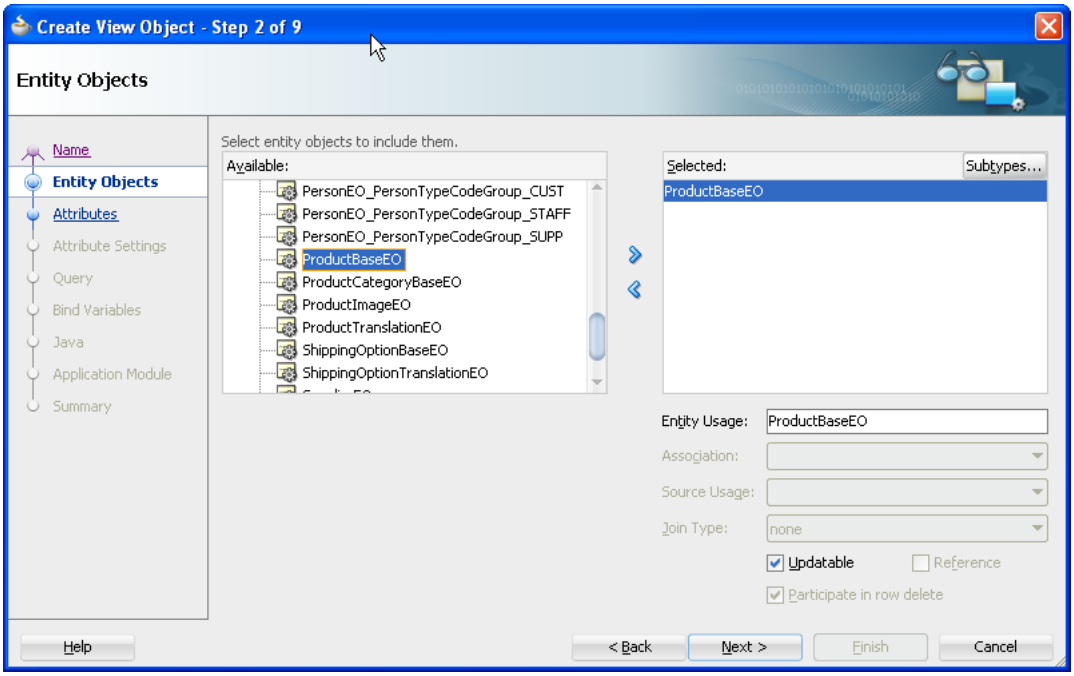
38.4.4.3 Using Business Components

Business components can also be packaged and reused. The entity objects, view objects, and associations can be packaged together and added to a consuming project. By default, packaged application modules will include the business components in the

JAR, but business components can be reused by themselves without the accompanying application module.

One way to reuse business components is to create new view objects using the entity objects from an ADF Library JAR. When you add view objects using the wizard, the entity objects will become available within the wizard to support view object generation. For instructions on creating view objects, see Section 5.2.1, "How to Create an Entity-Based View Object." When the wizard presents a screen for entity objects used to create view objects, the entity objects from the ADF Library will be available in the shuttle window, as shown in Figure 38–19.

Figure 38–19 Creating View Object Using Entity Objects from ADF Library



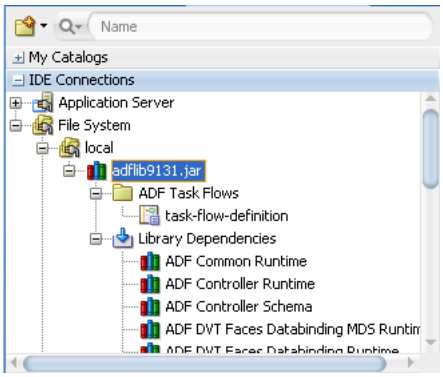
If the consuming project has the **Initialize Project for Business Component** option selected (in the Project Properties dialog Business Components page) before the ADF Library JAR is added, the business components within the JAR will automatically be made available to the project.

If you add the ADF Library JAR first, and then select **Initialize Project for Business Component**, JDeveloper will automatically load the business components.

38.4.4.4 Using Task Flows

Task flows added to a project will appear in the Component Palette when you are adding components to a JSF page. If you are using the Resource Palette, you can also drag and drop the task flow directly from the Resource Palette onto another task flow or page, as shown in Figure 38–20.

Figure 38–20 Using the Resource Palette to Drag and Drop Task Flows



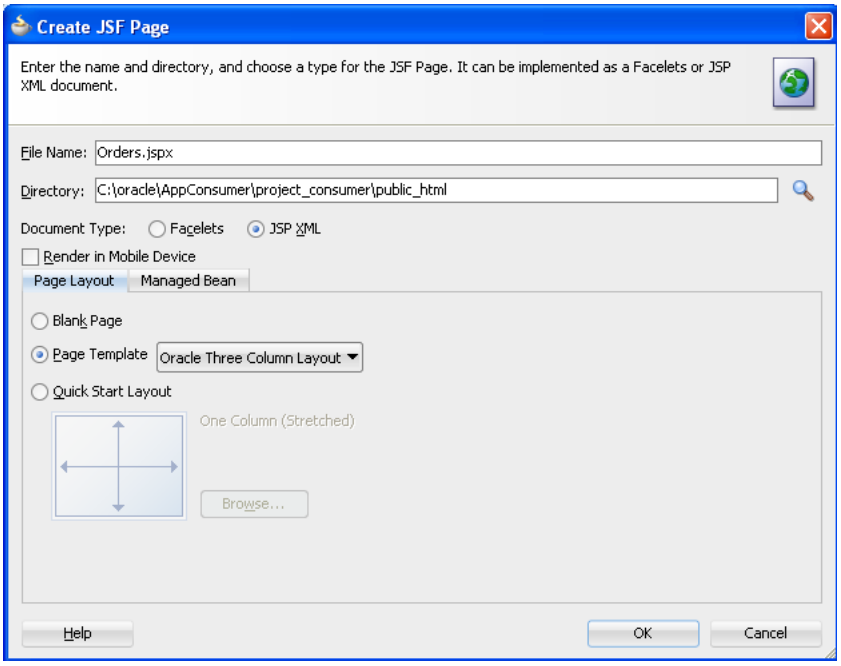
For more information about creating task flows, see Chapter 18, "Getting Started with ADF Task Flows."

38.4.4.5 Using Page Templates

When you add a page template to a project, the template will not be exposed in the Application Navigator. You will not have direct access to individual supporting files, such as image files. However, the template retains its access to its supporting files inside the JAR and is fully reusable within the project. When you apply the template, it will retain all the images that were loaded with the template.

The page template is exposed and accessible when you create a new JSF page using the wizard. When the wizard presents you with the option to use a page template, the ADF Library template will appear in the dropdown list. For example, if you loaded a page template called `StoreFrontTemplate` from an ADF Library, when you use the wizard to create a JSF page, `StoreFrontTemplate` will appear in the wizard, as shown in Figure 38–21. For information on how to use page templates and create a JSF page, see Section 24.2, "Using Page Templates" and the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Figure 38–21 Using a Page Template from ADF Library



38.4.4.6 Using Declarative Components

When you add a declarative component to a project, the JSP tag libraries that contain the component will be added to the project. The tag libraries will appear in the Component Palette, and the declarative components will be available for selection. For information about creating and using declarative components, see the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

38.4.5 What You May Need to Know About Differentiating ADF Library Components

If you mix components created during application development with components imported from the ADF Library, you may be able to differentiate between them by using the tooltips feature of JDeveloper.

Move the cursor over an application module or data control and you will see the full path of the source. If you see the ADF Library JAR file in the path, that means the component source is the ADF Library.

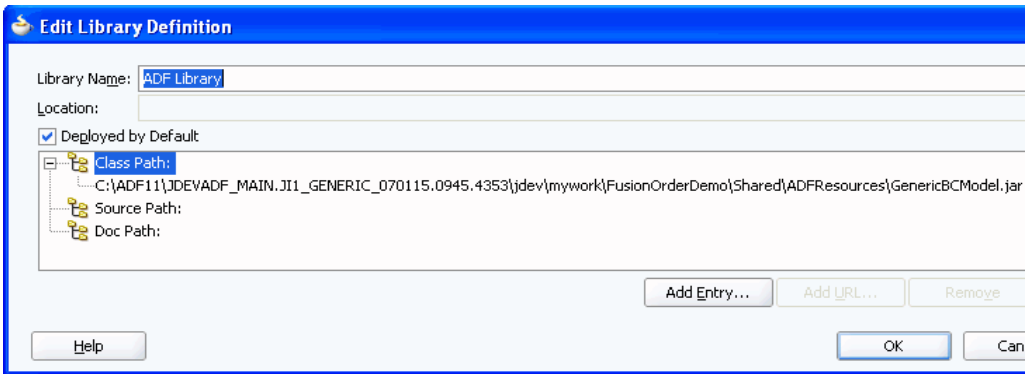
Note: The tooltip for items in the Data Controls panel varies depending on whether or not the node has ever been opened in this session. To ensure that you are seeing the most complete tooltip, you should expand the item.

38.4.6 What Happens at Runtime: Adding ADF Libraries

After an ADF Library JAR has been added to a project and to the class path, it behaves like any other library file. During runtime, any component that uses the component in the ADF Library JAR will reference that object. The process is transparent and there is no need to distinguish between components that were developed for the project and

those that are in ADF Library JARs. Figure 38–22 shows the ADF Library and the path to the JAR as defined for a project.

Figure 38–22 Edit Library Definition Dialog Showing the ADF Library in the Class Path



Note: Once an ADF Library JAR is created, any Java source used to create the component is not available in the JAR. If you want your consumers to have access to the source, such as for debugging, they must be delivered separately (usually in ZIP files). The consumer can then designate those ZIP files as the related component source for the **Source Path** entry in their ADF Library **Libraries and Classpath** entry in their own projects.

38.5 Removing an ADF Library JAR from a Project

You can use the Resource Palette to remove an ADF Library JAR from a project, or you can manually remove the JAR using the Project Properties dialog. You can remove an ADF Library JAR only if the components in the project do not have any dependencies on the components in the ADF Library JAR.

When you remove a JAR, it will no longer be in the project class path and its components will no longer be available for use.

38.5.1 How to Remove an ADF Library JAR from a Project Using the Resource Palette

The Resource Palette allows you to remove previously added ADF Library JARs from a project using a simple command.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you package an ADF component. For more information, see Section 38.5, "Removing an ADF Library JAR from a Project."

You may also find it helpful to understand functionality that can be used with packaging an ADF component. For more information, see Section 38.1.3, "Additional Functionality for Reusable ADF Components."

You will need to complete this task:

Add the ADF Library JAR using the Resource Palette. For more information, see Section 38.4, "Adding ADF Library Components into Projects."

To remove an ADF Library JAR from the project using the Resource Palette:

1. From the main menu, choose **View > Resource Palette**.
2. In the Application Navigator, select the project that has the ADF Library JAR you want to remove.
3. In the Resource Palette, locate the ADF Library JAR you want to remove from the current project.
4. Right-click the JAR and choose **Remove from Project**.

38.5.2 How to Remove an ADF Library JAR from a Project Manually

When you remove an ADF Library JAR manually, be sure to remove the correct library. Be aware of other libraries that are critical to the operation of the project.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you package an ADF component. For more information, see Section 38.5, "Removing an ADF Library JAR from a Project."

You may also find it helpful to understand functionality that can be used with packaging an ADF component. For more information, see Section 38.1.3, "Additional Functionality for Reusable ADF Components."

You will need to complete this task:

Add the ADF Library JAR manually. For more information, see Section 38.4, "Adding ADF Library Components into Projects."

To remove an ADF Library JAR from the project manually:

1. In the Application Navigator, double-click the project.
2. In the Project Properties dialog, select the **Libraries and Classpath** node.
3. In the **Classpath Entries** list, select **ADF Library** and click **Edit**.
4. In the Edit Library Definition dialog, select the ADF Library JAR you want to remove under the **Class Path** node, and click **Remove**.
5. Click **OK** to accept the deletion, and click **OK** again to exit the dialog.

39

Customizing Applications with MDS

The Oracle Metadata Services (MDS) framework allows you to create customizable applications. This chapter describes how to develop applications that can be customized and subsequently deployed by a customer. It also covers how to implement customizations on such applications.

This chapter includes the following sections:

- Section 39.1, "About Customization and MDS"
- Section 39.2, "Developing a Customizable Application"
- Section 39.3, "Customizing an Application"
- Section 39.4, "Extended Metadata Properties"
- Section 39.5, "Enabling Runtime Modification of Customization Configuration"

For information on how to deploy customized applications, see Section 41.4, "Deploying the Application."

39.1 About Customization and MDS

Using the customization features provided by MDS, you can create applications that fall into the following customization patterns:

- Seeded customization

  Seeded customization of an application is the process of taking a generalized application and making modifications to suit the needs of a particular group, such as a specific industry or site. Seeded customizations exist as part of the deployed application, and endure for the life of a given deployment. This chapter describes how to create a customizable application, and then customize the application using JDeveloper.

- User customization (change persistence)

  User customization allows an end user to change the content of the application at runtime to suit individual preferences (for example, which columns are visible in a table), and have those changes "remembered" the next time the user opens the application. For information about user customization, see Chapter 40, "Allowing User Customizations at Runtime."

For more information about the MDS architecture and metadata repositories (database- and file-based) and archives (EAR, MAR), refer to the section about managing the MDS repository in the *Oracle Fusion Middleware Administrator's Guide*.

39.1.1 Customization and Layers: Use Cases and Examples

A customized application contains a base application and one or more layers containing customizations. MDS stores the customizations in a metadata repository and retrieves them at runtime to merge the customizations with the base metadata to reveal the customized application. Since the customizations are saved separately from the base, the customizations are upgrade safe; a new patch to base can be applied without breaking customizations. When a customized application is launched, the customization content is applied over the base application.

For example, say you have a generalized payroll application with a validation rule that limits the salary field to 4000. Then you create a customization of that validation rule that limits the salary field to 3300. At runtime, the customization is applied to the base application and the validation rule for the salary field limits it to 3300.

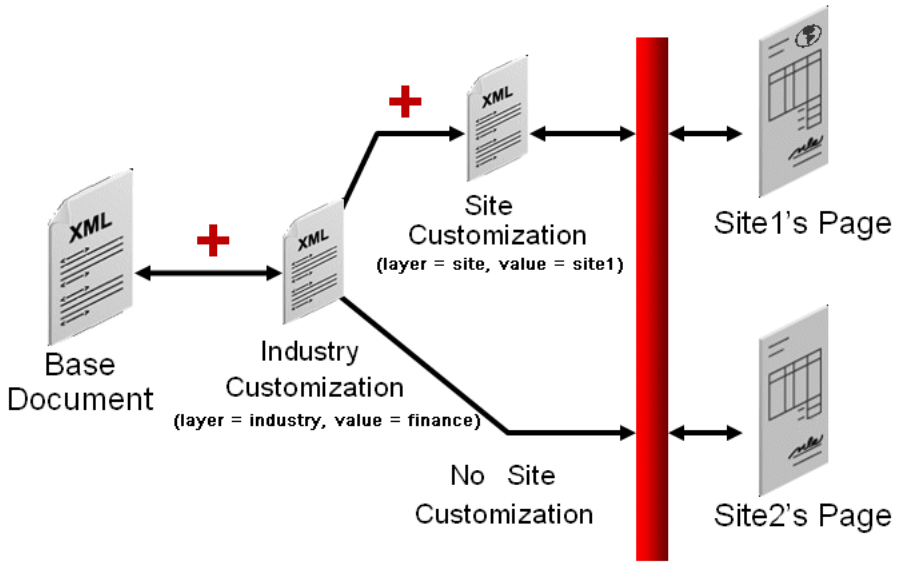
A customizable application can have multiple *customization layer*s. Examples of customization layers are `industry` and `site`. Each layer can have multiple *customization layer value*s, but typically only one such layer value from each layer is applied at runtime. For example, the `industry` layer for a customizable application can contain values for `healthcare` and `financial` industries; but in the deployed customized application, only one of the values from this layer is used at a time.

Layer values from multiple customization layers can be applied, in a specified order of precedence, on top of the base metadata. For example, a customized application can contain customizations in the `financial` layer value of the `industry` layer and the `Financial Company #1` layer value of the `site` layer. Each customization layer corresponds to a customization document that contains a set of instructions that change the underlying metadata.

The *customization context* of a customized application is defined by the set of customization layer values applied to it.

Figure 39–1 illustrates how layers are applied in a customized application.

Figure 39–1 Example of Layered Customization



To support this, you use JDeveloper to create customization classes, define layers and values, and specify the order of precedence. These processes are described in Section 39.2, "Developing a Customizable Application."

39.1.2 Static and Dynamic Customization Content

Customizations can be categorized as either static or dynamic. Static customizations have only one layer value in effect for all executions of the application, while dynamic customizations can have values that vary based on the execution context of the application. If a customization can vary for different users executing the application, then it is dynamic. If a customization has the same value for all users executing the application then it is static.

When you implement customizations in ADF Business Components objects, the customizations remain the same for entire runtime of the application. This is because these objects are loaded only once for an application and reused for the duration of the application. For example, you can have a customized validation rule in the Healthcare Company #1 value of the site layer that limits salaries for that site to 3300. This is static customization content.

However, you can also implement customizations at the controller or view level that allow the layer value to be determined at runtime, based on user roles (responsibilities) or other application-specific criteria. For example, you can design an application so that users from different organizations see different sets of fields on a given screen. This is dynamic customization content.

The determination of whether a customization is static or dynamic is made in the customization class. In the customization class, if the `getCacheHint()` method returns `ALL_USERS`, then the customization layer is static. For more information about `CacheHint` values, see Section 39.2.2, "What You May Need to Know About Customization Classes."

All objects could have a static customization layer, depending on how the customization classes are implemented. But for ADF Business Components objects, customizations can only be static.

39.1.3 Additional Functionality for Customization

You may find it helpful to understand other features before you start working with customization. Following are links to other functionality that may be of interest.

- For information about the MDS architecture and metadata repositories (database- and file-based) and archives (EAR, MAR), see to the section about managing the MDS repository in the *Oracle Fusion Middleware Administrator's Guide*.
- You can use the ADF Faces change persistence framework to create JSF pages that users can customize at runtime. For information, see Chapter 40, "Allowing User Customizations at Runtime."

39.2 Developing a Customizable Application

To create a customizable application, create the base application and perform the following procedures:

To prepare an application for customization:

1. Create the customization classes that will be used, as described in Section 39.2.1, "How to Create Customization Classes."
2. Enable seeded customization in the application, as described in Section 39.2.4, "How to Enable Seeded Customizations for View Projects."
3. Specify the customization classes in the `adf-config.xml` file, as described in Section 39.2.7, "How to Configure the adf-config.xml file."

4. You can optionally restrict runtime customizations on the application, as described in Section 39.4.1, "How to Edit Extended Metadata Properties."
5. After you have prepared the application for customization, you must prepare JDeveloper so you can use it to implement the customizations, as described in Section 39.3.4, "How to Configure Customization Layers."

39.2.1 How to Create Customization Classes

A customization class is the interface that MDS uses to define which customization applies to the base definition metadata. Each customization class defines a customization layer (for example, `industry` or `site`) and can contain multiple layer values. The customization classes that are used in the application must be available to JDeveloper when customizing the application, and included in the deployed application.

39.2.1.1 Customization Classes

A customization class evaluates the current context and returns a String result. This String result is used to locate the customization layer.

The customization class provides the following information:

- A name, that represents the name of the layer.
- An array of values, that represent the customization layer values. Typically, each layer returns a single value. If multiple values are returned, the customizations available in the MDS repository for those values are applied in the order in which they appear in the array. For more information, see Section 39.2.1.2, "Implementing the getValue() Method in Your Customization Class."
- An IDPrefix, for objects created in the layer. When new objects are created in a customization layer they need a unique ID. The IDPrefix is added to the autogenerated identifier for the object to create an ID for the newly added object. Each layer needs a unique IDPrefix so that objects created at different customization layers will have unique IDs.
- A cache hint, for the layer defined by the customization class. The cache hint defines whether a layer is static or dynamic. If the `getCacheHint()` method returns `ALL_USERS`, then the customization layer is static. For more information about dynamic customizations, see Section 39.1.2, "Static and Dynamic Customization Content." For more information about `CacheHint` values, see Section 39.2.2, "What You May Need to Know About Customization Classes."

Customizations can be used to tailor an application to suit a specific industry domain (verticalization). Each such domain denotes a customization layer and is depicted using a customization class.

To implement seeded customizations using your customization classes:

- The `cust-config` section (under `mds-config`) in the `adf-config.xml` must contain a reference to the customization classes (as shown in Example 39–6).

  The customization configuration (`cust-config`) section provides the customization classes and their precedence for a customized application. See Section 39.2.7, "How to Configure the adf-config.xml file."
- The customization classes must be included in the application to support seeded customizations.

For more information, see Section 39.2.3.1, "Making Customization Classes Available to JDeveloper at Design Time." At runtime, your customization classes must be available in the EAR-level application class loader.

- The layer values must be listed in the `CustomizationLayerValues.xml` file.

  This file is located in the *`jdev_install`*`\jdev` directory. The names of the layers in this file must be consistent with the customization classes. (For more information, see Section 39.3.4, "How to Configure Customization Layers.") JDeveloper uses this file to retrieve layer values at design time.

When JDeveloper is launched in the **Customization Developer** role, the Customization Context window displays the available customization layers and layer values. You can select the layer and value to which you want to apply customizations in the Customization Context window. For more information about working in the **Customization Developer** role, see Section 39.3.1, "Introducing the Customization Developer Role." The layer you choose to customize is called the *tip layer*. For more information, see Section 39.3.3, "Introducing the Tip Layer."

Example 39–1 shows a sample customization class. Note that all customization classes should have a single, no-argument constructor.

Example 39–1 Sample IndustryCC customization class in the mycompany package

```
package mycompany;

import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;
import oracle.mds.core.MetadataObject;
import oracle.mds.core.RestrictedSession;
import oracle.mds.cust.CacheHint;
import oracle.mds.cust.CustomizationClass;

public class IndustryCC extends CustomizationClass {
  private static final String DEFAULT_LAYER_NAME = "industry";
  private String mLayerName = DEFAULT_LAYER_NAME;

  public IndustryCC() {
  }
  public CacheHint getCacheHint() {
    return CacheHint.ALL_USERS;
  }
  public String getName() {
    return mLayerName;
  }

  public String generateIDPrefix(RestrictedSession sess, MetadataObject mo) {
      return new String ("I");
  }

  public String[] getValue(RestrictedSession sess, MetadataObject mo) {
    // This needs to return the appropriate value at runtime.
    return new String[] {"financial"};
  }
}
```

This example shows the four methods that define how the customization class will work: `getCacheHint()`, `getName()`, `generateIDPrefix()`, and `getValue()`.

The return value for `getCacheHint()` indicates to MDS how widely visible a metadata object with this customization is, and therefore its likely duration in memory. It also defines whether or not a layer is static or dynamic. In this example, the `getCacheHint()` method returns `ALL_USERS`, which means the customization layer is static. For more information about `CacheHint` values, see Section 39.2.2, "What You May Need to Know About Customization Classes."

The `getName()` method returns the name of the customization layer. A SiteCC customization class, for example, might return "site." In this example, the `getName()` method returns "industry."

The `generateIDPrefix()` method creates an IDPrefix. The IDPrefix is a unique, abbreviated string that identifies the name and value of the layer. It is used as the prefix of the ID for objects created in the customization layer. The default implementation (if no IDPrefix is specified) returns name of the layer concatenated with customization value. For performance reasons, the IDPrefix should be kept short (4 characters or less). Therefore, the default implementation should be overridden. The related `getIDPrefix()` method returns the IDPrefix of the customization layer. In this example, the `getIDPrefix()` method would return "I" (for "industry").

The `getValue()` method returns the customization value (or values) for the customization class. In this example, the `getValue()` method returns a single value, "financial," that defines the customization context when combined with the layer name. There are additional techniques for using the `getValue()` method described in Section 39.2.1.2, "Implementing the getValue() Method in Your Customization Class."

Note: The possible layer values corresponding to a layer name are retrieved by JDeveloper in the **Customization Developer** role from the `CustomizationLayerValues.xml` file. The precedence of layers is defined by the order of the customization classes specified in the `adf-config.xml` file. The names of the layers must be consistent in these files and in the customization classes.

39.2.1.2 Implementing the getValue() Method in Your Customization Class

The `getValue()` method is used to retrieve the layer value(s) of the customization class based on the application context and the metadata. For example, calling `getValue()` on a `SiteCC` customization class might return an array with single entry "`headquarters`." Typically, the `getValue()` method returns an array with a single value, as shown in Example 39–1.

You can also return multiple values from the `getValue()` method, as shown in Example 39–2.

Example 39–2 The getValue() Method Returning Multiple Values

```
public String[] getValue(RestrictedSession sess, MetadataObject mo) {
    return new String[]{"North America", "US", "CA"}
}
```

When multiple values are returned, customizations applicable to all values are applied. Customizations are applied in the order in which they appear in the array. In this example, `North America` customizations are applied over the base application, then `US` customizations are applied, and finally `CA`.

> **Note:** Returning multiple values for a customization layer is an advanced concept that is typically unnecessary.

The `getValue()` method can return a layer value based on the current execution context for the current user, pulled either from static or thread local state maintained by the client, or from properties set by the client on the MDS session and based on the metadata object name. Example 39–3 shows this type of implementation.

Example 39–3 The getValue() Method Returning a Layer Value Based on the Current Execution Context

```
public String[] getValue(RestrictedSession sess, MetadataObject mo) {
    if (mo.getName().equals("/sample/abc.jspx"))
    {
        return new String[]{"Headquarters"};
    }
     else
    {
        return new String[]{"RemoteSite"};
    }
}
```

In this example, the `getValue()` method uses the `getName()` method on the metadata object to determine if the name of the metadata document is "`/sample/abc.jspx`". If so, `getValue()` returns `Headquarters` to apply headquarters customizations. If not, it returns `RemoteSite` to apply customizations for remote sites.

> **Note:** Coding the `getValue()` method to return a value based on the metadata object is an advanced concept that is typically unnecessary. Customization context for dynamic layers is typically determined through facets of the application context, such as user name or responsibility.

An additional technique that can be useful during the development cycle is to use an external properties file to specify layer values. Example 39–4 references a properties file (`customization.properties`) that stores the layer values.

Example 39–4 The getValue() Method Using a Properties File to Specify Layer Values

```
public String[] getValue(RestrictedSession sess, MetadataObject mo) {
  Properties properties = new Properties();
    String configuredValue = null;
    Class clazz = IndustryCC.class;
    InputStream is = clazz.getResourceAsStream("/customization.properties");
    if (is != null){
      try {
        properties.load(is);
        String propValue = properties.getProperty(mLayerName);
        if (propValue != null){
          configuredValue = propValue;
        }
      } catch (IOException e) {
          e.printStackTrace();
      }
      finally {
```

```
        try {
          is.close();
        } catch (IOException e) {
        e.printStackTrace();
        }
      }
    }
  return new String[] {configuredValue};
}
```

When an application using this technique is run in JDeveloper, you can change the layer value in the properties file and refresh the browser to see new customizations being applied. This customization class and properties file combination allows you to maintain the layer value in a separate file so you don't need to modify and recompile Java code to change it for a particular deployment. If you use a `customization.properties` file, it must be packaged with the customization classes so that they are loaded from the same class loader.

Example 39–5 shows a sample `customization.properties` file. When the `IndustryCC` class is loaded with this properties file, the layer value `healthcare` is applied.

Example 39–5 Sample Contents of a customization.properties File

```
#Configured values for the default layer values
industry=healthcare
site=headquarters
```

39.2.1.3 Creating a Customization Class

When creating customization classes, put them in a separate project. This allows you to deploy them to a JAR and to import the JAR into your lowest level project (which is typically the data model project). This approach increases modularity, making it easier to include the customization classes in multiple applications across your company, and easier to patch them centrally. For more information about this approach, see Section 39.2.3, "How to Consume Customization Classes."

Alternatively, you can put the customization classes in the data model project if you are creating customization classes that will be used in a single application only, or if you have no need to use the **Customization Developer** role and want simply to run the application locally from JDeveloper or to deploy remotely with MDS configuration.

Make sure that there is only one copy of the customization classes in the application and that they are packaged in a JAR file so that they are loaded at the EAR-level application class loader. By default, adding project dependencies will add the customization classes to the WAR profile, which will not work correctly after the application is packaged and deployed. For more information about packaging your application for deployment, see Section 41.3.2, "How to Create Deployment Profiles."

Use the following procedure to create a customization class.

Before you begin:

It may be helpful to have an understanding of the features of customization classes. For more information, see Section 39.2.1.1, "Customization Classes."

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Section 39.1.3, "Additional Functionality for Customization."

You will need to launch JDeveloper using the **Studio Developer** role, and open (or create) the application that will hold the customization classes.

To create a customization class:

1. From the main menu, choose **File > New**.
2. In the New Gallery, expand **General**, select **Projects** and then **Java Project**, and click **OK**.

   This opens the Create Java Project dialog. For more information about the options in this dialog, see the online help.
3. Specify the appropriate settings for your project, and click **Finish**.

   Note: You should use a single project to hold all of your customization classes.
4. In the Application Navigator, right-click your project and choose **Project Properties**.
5. In the Project Properties dialog, click **Libraries and Classpath**, and then click **Add Library**.
6. In the Add Library dialog, select **MDS Runtime**, and click **OK**, then click **OK** to close the Project Properties dialog.
7. In the Application Navigator, right-click the project and choose **New**.
8. In the New Gallery, expand **General**, select **Java** and then **Class**, and click **OK**.
9. In the Create Java Class dialog, enter a name for the class and the appropriate package.

   Note: Customization classes are typically named for the layer name they return. For example, a customization class that returns the layer name `industry` would be named `IndustryCC`.
10. In the **Extends** field, enter `oracle.mds.cust.CustomizationClass`.

    For more information about this class, refer to the Javadoc.
11. Make sure that **Implement Abstract Methods** is checked and click **OK**.
12. Replace the code in the generated file with code like that shown in Example 39–1.
13. Save your changes and rebuild the project.

This creates the customization class. The sample code uses the package name `mycompany` and the class name `IndustryCC`. You will need to change these as appropriate for your application.

39.2.2 What You May Need to Know About Customization Classes

As described in Section 39.2.1.1, "Customization Classes," the customization class defines a `CacheHint` which specifies the visibility of metadata objects in a

customization layer, and therefore its likely duration in memory. This information is used by MDS to decide whether or not to cache a customization, and where to cache it. Any customization layers constructed using this customization class have this cache hint.

The following constants are supported values of `CacheHint`:

- `ALL_USERS` — The customization is applied globally (non-conditionally) for a given deployment. This constant is used for static customization layers.
- `MULTI_USER` — The customization is applied to multiple users.
- `REQUEST` — The customization is applied for the duration of the request only.
- `USER` — The customization is applied for a single user to documents which are accessed throughout the user's application session. (In web applications the application session is typically a servlet session.)

Note: Customization classes are likely to be executed frequently, once for each document being accessed to get the layer name and layer value, so take care to ensure their efficiency.

39.2.3 How to Consume Customization Classes

After you have created your customization classes, you can use them at design time in the **Customization Developer** role, and at runtime in the application. To be consumed in an application or in JDeveloper, the classes must be packaged appropriately.

39.2.3.1 Making Customization Classes Available to JDeveloper at Design Time

After you create the customization classes, you must make them available to JDeveloper so that you can use them when implementing customizations while working in the **Customization Developer** role.

Because the customization classes are reusable components, you can create a separate project to contain them, and package them into their own JAR file. You can then import the JAR into the consuming application, which makes the customization classes available to JDeveloper. For more information about creating the customization classes in a separate project, see Section 39.2.1.3, "Creating a Customization Class." For more information about packaging the customization classes into a JAR file, see Section 41.3.2.6, "Adding Customization Classes into a JAR."

Note: This procedure is not required if you created your customization classes in the data model project of the consuming application.

Use the following procedure to make the customization classes visible to the application, and then add the customization classes to the `cust-config` section of the `adf-config.xml` file, as described in Section 39.2.7, "How to Configure the adf-config.xml file."

Before you begin:

It may be helpful to have an understanding of the how customization classes are packaged. For more information, see Section 39.2.1.1, "Customization Classes."

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Section 39.1.3, "Additional Functionality for Customization."

You will also need to complete these tasks:

- Create your customization classes in an external project, as described in Section 39.2.1.3, "Creating a Customization Class."
- Create a JAR using the procedure described in Section 41.3.2.6, "Adding Customization Classes into a JAR."
- Launch JDeveloper using the **Studio Developer** role, and open the application that you want to customize.

To use customization classes from an external project:

1. In the Application Navigator, right-click the data model project and choose **Project Properties**.
2. In the Project Properties dialog, click **Libraries and Classpath**.
3. Click **Add JAR/Directory**.
4. In the Add Archive or Directory dialog, select the JAR file you created that contains the customization classes, and click **Select**.
5. Click **OK** to close the Project Properties dialog.

   Now the customization classes are available to JDeveloper for customization, and for running your project locally in JDeveloper.

39.2.3.2 Making Customization Classes Available to the Application at Run Time

When you package and deploy your customized application, the customization classes must be available at the application level on the application's class path.

When you define the deployment profiles for your application, you need to add the customization classes JAR file to the EAR assembly, and to avoid duplication, you need to make sure that the WAR profile does not include the customization classes JAR file. For more information, see Section 41.3.2.3, "Creating an Application-Level EAR Deployment Profile."

39.2.4 How to Enable Seeded Customizations for View Projects

Like all customizable components, the XML elements of a customizable metadata object must be uniquely identifiable by MDS, and therefore must have a unique, non-null identifier. The component's identifier is used to refer to the element in the customization instructions in the customization layer. The ID property is the identifier for each type of component in an ADF Faces `.jspx` or `.jsff` file.

To allow for customizations on your JSF and JSP pages, you must enable seeded customizations in the application's view project, which drives some defaults for your pages. This is not necessary for your model and controller projects.

Note: MDS requires that pages be XML-based to be customized. Therefore, customizations are not allowed on `.jsp` files; use `.jspx` files instead.

Additionally, facelets files must have a `.jsf` extension to be customizable. MDS uses this extension to recognize it as a Facelets file.

Before you begin:

It may be helpful to have an understanding of the how seeded customization works. For more information, see Section 39.1.1, "Customization and Layers: Use Cases and Examples."

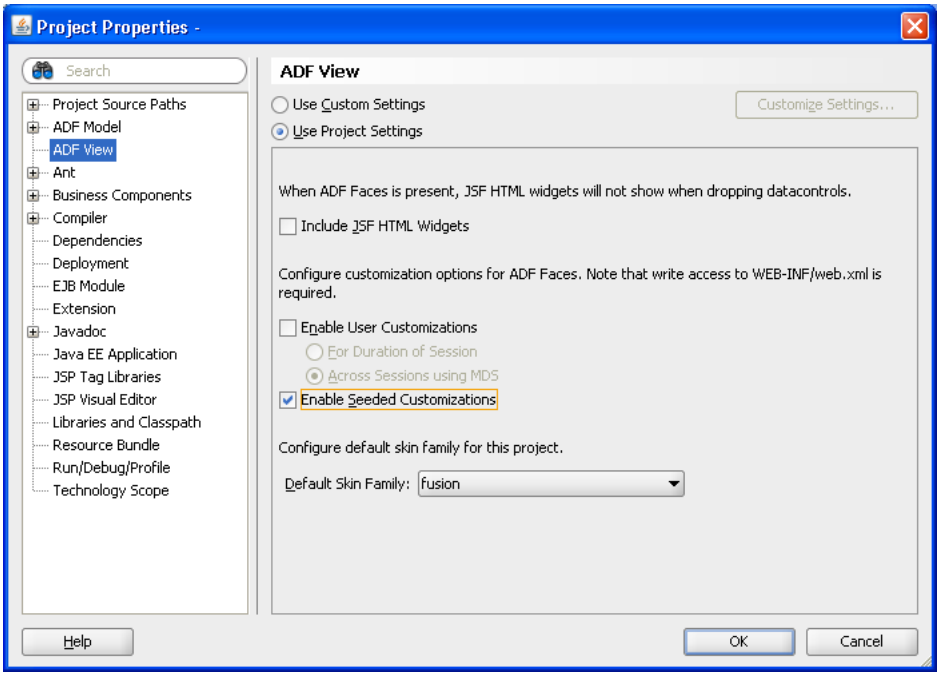
You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Section 39.1.3, "Additional Functionality for Customization."

You will also need to launch JDeveloper using the **Studio Developer** role, and open the application that you want to make customizable.

To enable seeded customizations in your view project:

1. In the Application Navigator, right-click the view project and choose **Project Properties**.
2. In the Project Properties dialog, click **ADF View**.
3. Select the **Enable Seeded Customizations** checkbox, as shown in Figure 39–2.
4. Click **OK**.
5. Save the changes to your project.

Figure 39–2 Project Properties - Enable Seeded Customizations



39.2.5 How to Enable Seeded Customizations in Existing Pages

If you have pages in your project that were created in an earlier version of JDeveloper, you must make sure that these preexisting pages are also enabled for seeded customizations. This is only necessary if you migrated the application from an earlier version of JDeveloper and did not generate IDs during migration.

Before you begin:

It may be helpful to have an understanding of the how seeded customization works. For more information, see Section 39.1.1, "Customization and Layers: Use Cases and Examples."

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Section 39.1.3, "Additional Functionality for Customization."

You will also need to launch JDeveloper using the **Studio Developer** role, and open the application that you want to make customizable.

To enable seeded customizations in an existing page:

1. Create an audit profile to implement ID tokens for all XML objects in your page.
   a. From the **Tools** menu, choose **Preferences**.
   b. In the Preferences dialog, select **Audit > Profiles**.
   c. In the **Rules** tab on the Profiles page, deselect all rules.
   d. Select the rule **ADF Faces > Component ID Rules > Check for ID When ADF Faces is Present**.
   e. From the **Default Fix** dropdown list, select **Generate a unique ID**.
   f. Click **Save As**, then enter an identifiable name for the profile (such as Generate Unique IDs). and click **Save**.
   g. Click **OK** to close the Preferences dialog.
2. In the Application Navigator, select the page for which you want to enable seeded customizations. Alternatively, you can select a project to run the audit on all the files it contains.
3. From the **Build** menu, choose **Audit** *filename*.
4. In the Audit dialog, select the profile you created to generate ids and click **Run**.
5. Use the Log window to review issues and apply fixes.
6. When the audit is complete, save your changes.

39.2.6 How to Enable Customizations in Resource Bundles

If you plan to create new resource keys when implementing customizations, you can specify the affected resource bundles using the Application Properties dialog.

Before you begin:

It may be helpful to have an understanding of the how seeded customization works. For more information, see Section 39.1.1, "Customization and Layers: Use Cases and Examples."

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Section 39.1.3, "Additional Functionality for Customization."

You will also need to launch JDeveloper using the **Studio Developer** role, and open the application that you want to make customizable.

To enable customizations in resource bundles:

1. From the application context menu, choose **Application Properties**.

2. In the Application Properties dialog, click **Resource Bundles**.
3. Click **Add**.
4. In the Select Resource Bundle dialog, navigate to and select the resource bundles for which you want to enable customization.
5. Click **Open**.
6. In the Application Properties dialog, select the checkbox in the **Overridden** column of the **Bundle** table.
7. Click **OK**.

39.2.7 How to Configure the adf-config.xml file

The application's `adf-config.xml` file must have an appropriate `cust-config` element in the `mds-config` section. The `cust-config` element allows clients to define an ordered and named list of customization classes. You use the overview editor for the `adf-config.xml` file to add customization classes (see Figure 39–3).

Before you begin:

It may be helpful to have an understanding of the how customization classes are created and packaged. For more information, see Section 39.2.1.1, "Customization Classes."

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Section 39.1.3, "Additional Functionality for Customization."

You will also need to complete these tasks:

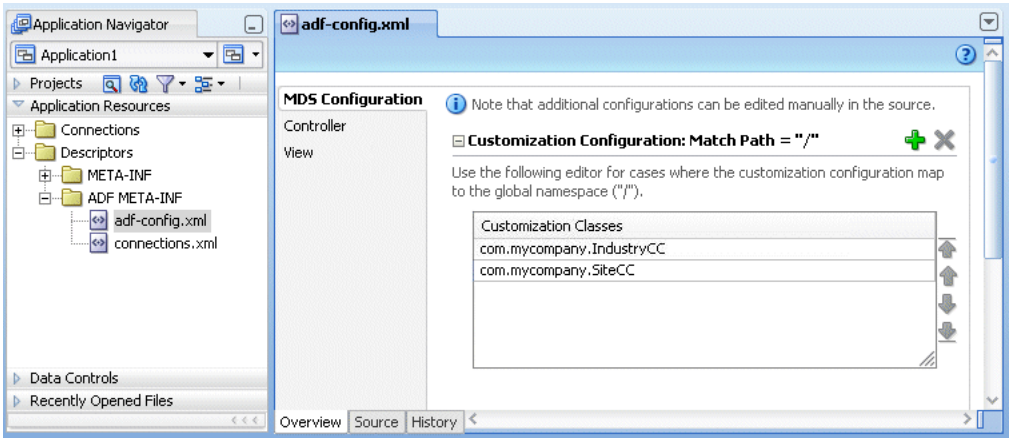
- Create your customization classes, as described in Section 39.2.1.3, "Creating a Customization Class."
- Make your classes available to JDeveloper, as described in Section 39.2.3.1, "Making Customization Classes Available to JDeveloper at Design Time."
- Create a JAR using the procedure described in Section 41.3.2.6, "Adding Customization Classes into a JAR."
- Launch JDeveloper using the **Studio Developer** role, and open the application that you want to customize.

To identify customization classes in the adf-config.xml file:

1. In the Application Navigator, open the Application Resources panel.
2. Expand **Descriptors**, then expand **ADF META-INF**.
3. Right-click `adf-config.xml`, and choose **Open**.
4. In the overview editor, click the **MDS Configuration** navigation tab, and then click the **Add** icon.
5. In the Edit Customization Classes dialog, search for or navigate to the customization classes you have already created.
6. Select the appropriate classes and click **OK**.
7. After you have added all of the customization classes, you can use the arrow icons to put them in the appropriate order.

Figure 39–3 shows the overview editor for the `adf-config.xml` file with two customization classes added.

Figure 39–3 adf-config.xml Overview Editor



The order of the customization-class elements defines the precedence of customization layers. For example, in the code shown in Example 39–6, the `IndustryCC` class is listed before the `SiteCC` class. This means that customizations at the industry layer are applied to the base application, and then customizations at the site layer are applied.

Example 39–6 Customization Class Order in the adf-config.xml File

```
<adf-config xmlns="http://xmlns.oracle.com/adf/config">
  <adf-mds-config xmlns="http://xmlns.oracle.com/adf/mds/config">
    <mds-config xmlns="http://xmlns.oracle.com/mds/config" version="11.1.1.000">
      <cust-config>
        <match path="/">
          <customization-class name="com.mycompany.IndustryCC"/>
          <customization-class name="com.mycompany.SiteCC"/>
        </match>
      </cust-config>
    </mds-config>
  </adf-mds-config>
</adf-config>
```

39.2.8 What Happens When You Create a Customizable Application

When you create a customizable application, you have a base application that includes the pieces necessary for you or someone else to use as the basis for a customized application.

To perform the customization, you must open the application in JDeveloper using the **Customization Developer** role, as described in Section 39.3, "Customizing an Application."

39.2.9 What You May Need to Know About Customizable Objects and Applications

Oracle ADF components (such as controller, model, and business components objects) must have a unique identifier so that they can be customized. ADF components generated by JDeveloper are created with identifiers by default, with the exception of JSP and JSF pages in your view controller projects. To cause JDeveloper to generate identifiers for components on pages in your view controller projects, you must explicitly specify this at the project level (as explained in Section 39.2.4, "How to Enable Seeded Customizations for View Projects").

Before you implement customizations in an application, make sure that all objects that you intend to customize have the necessary identifiers. In many cases, you can run an audit rule to catch and fix any omissions (as explained in Section 39.2.5, "How to Enable Seeded Customizations in Existing Pages").

Also, take care to ensure the efficiency of your customization classes, because they can be executed frequently, once for each document being accessed to get the layer name and layer value.

39.3 Customizing an Application

Using the **Customization Developer** role, you can create customizations in a customizable application.

39.3.1 Introducing the Customization Developer Role

The **Customization Developer** role is used to customize the metadata in a project. Customization features are available only in this role. When you are in the **Customization Developer** role, you can do the following:

- Create and update customizations
- Select and edit the tip layer of a customized application
- Remove existing customizations

When you are in the **Customization Developer** role, the source editor is read-only and the following JDeveloper features are disabled:

- Workspace migration
- Creation, deletion, and modification of application and IDE connections. You must configure connections in Default role before opening an application in Customization Developer role.

When working with an application in the **Customization Developer** role, new objects cannot be created, and noncustomizable objects cannot be modified. However, the following content types are customizable:

- Portal modules
- ADF modules, including ADF Faces, ADF Model, ADF Business Components, and ADF Controller

---

Note: ADF Business Components objects are customizable only if a static customization class is selected in the Customization Context window. Otherwise, business components objects are read-only.

When working in the **Customization Developer** role, you cannot edit noncustomizable files, such as Java classes, resource bundles, security policies, deployment descriptors, and configuration files. Noncustomizable files are indicated by a lock icon when you are working in the **Customization Developer** role.

Note: Facelets files must have a `.jsf` extension to be customizable. MDS uses this extension to recognize it as a Facelets file.

You are also restricted from modifying project settings and customizing certain ADF Business Components features, including service interfaces and business event definitions. Additionally, you cannot refactor, or make changes to customizable files that would, in turn, necessitate changes in noncustomizable files.

39.3.2 How to Switch to the Customization Developer Role in JDeveloper

The customization features of JDeveloper are available to you in the **Customization Developer** role. To work in this role, you can either choose it when you start JDeveloper or, if JDeveloper is already running, you can use the **Switch Roles** menu to switch to the **Customization Developer** role.

Before you begin:

It may be helpful to have an understanding of the features of the **Customization Developer** role. For more information, see Section 39.3.1, "Introducing the Customization Developer Role."

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Section 39.1.3, "Additional Functionality for Customization."

You will also need to launch JDeveloper.

To switch to the Customization Developer role in JDeveloper:

1. From the main menu, choose **Tools > Switch Roles > Customization Developer**.

> **Note:** You can optionally toggle the **Tools > Switch Roles > Always Prompt for Role Selection at Startup** menu item, to specify whether or not you want to choose the role when JDeveloper is launched. If deselected, JDeveloper launches in the role it was in when you last closed it.

39.3.3 Introducing the Tip Layer

When working in the **Customization Developer** role, the layer and layer value combination that is selected in the Customization Context window is called the *tip layer*. The changes you make while in the **Customization Developer** role are applied to this layer.

The metadata displayed in the JDeveloper editors is a combination of the base metadata and the customization layers up to and including the tip layer, according to the precedence set in `adf-config.xml`, with the values specified in the Customization Context window for each layer.

When working in the **Customization Developer** role, you can also see the noncustomized state of the application. When you select **View without Customizations** in the Customization Context window, there is no current tip layer. Therefore, what you see is the noncustomized state. While you are in this view, all customizable files show the lock icon (in the Application Navigator), indicating that these files are read-only.

When you make customizations in a tip layer, these customizations are indicated by an orange icon in the Property Inspector. A green icon indicates non-tip layer customizations. When you see an orange icon beside a property, you have the option of deleting that customization by choosing **Remove Customization** from the dropdown menu for that property.

39.3.4 How to Configure Customization Layers

To customize an application, you must specify the customization layers and their values in the `CustomizationLayerValues.xml` file so that they are recognized by JDeveloper.

When you open a customizable application in the **Customization Developer** role, JDeveloper reads the `adf-config.xml` file to determine the customization classes to use and their order of precedence. JDeveloper also reads the `CustomizationLayerValues.xml` file to determine the layer values to make available in the Customization Context window. If there are layer values defined in the `CustomizationLayerValues.xml` file that are not defined in the customization classes listed in the `adf-config.xml` file, they are not displayed in the Customization Context window.

Therefore, you can have a comprehensive list of layer values for all of your customization projects in the `CustomizationLayerValues.xml` file, and only those appropriate for the current application are available in the Customization Context window. Conversely, you could have a comprehensive list of customization classes for an application in the `adf-config.xml` file, and only the subset of layer values that you will work on in your `CustomizationLayerValues.xml` file.

> **Note:** At design time, JDeveloper retrieves customization layer values from the `CustomizationLayerValues.xml` file. But at runtime, the layer values are retrieved from the customization class.

The names of the layers and layer values that you enter in the `CustomizationLayerValues.xml` file must be consistent with those specified in your customization classes. Example 39–7 shows the contents of a sample `CustomizationLayerValues.xml` file.

Example 39–7 Layers and Layer Values Defined in CustomizationLayerValues.xml

```
<cust-layers  xmlns="http://xmlns.oracle.com/mds/dt">
  <cust-layer name="industry" id-prefix="i">
    <cust-layer-value value="financial" display-name="Financial" id-prefix="f"/>
    <cust-layer-value value="healthcare" display-name="Healthcare" id-prefix="h"/>
  </cust-layer>
  <cust-layer name="site" id-prefix="s">
    <cust-layer-value value="headquarters" display-name="HQ" id-prefix="hq"/>
    <cust-layer-value value="remoteoffices" display-name="Remote" id-prefix="rm"/>
  </cust-layer>
</cust-layers>
```

For each layer and layer value, you can add an `id-prefix` token. This helps to ensure the uniqueness of the `id`, so that customizations are applied accurately. When you add a new element (such as a command button) to a page during customization, JDeveloper adds the `id-prefix` of the layer and layer value (determined by the selected tip layer) to the autogenerated identifier for the element to create an `id` for the newly added element in the customization metadata file. For example, in Example 39–7, the `site` layer has an `id-prefix` of "`s`" and the `headquarters` layer value has an `id-prefix` of "`hq`". So, when you select `site/headquarters` as the tip layer and add a command button to a page, the command button will have an `id` of "`shqcb1`" in the metadata customization file.

For each layer value, you can also add a `display-name` token to provide a human-readable name for the layer value. When you are working in the

Customization Developer role, the value of the `display-name` token is shown in the Customization Context window for that layer value.

You can define the customization layer values either globally for JDeveloper or in an application-specific file. If you use an application-specific file, it takes precedence over the global file. For more information on configuring layer values globally for JDeveloper, see Section 39.3.4.1, "Configuring Layer Values Globally." For more information on configuring application-specific layer values, see Section 39.3.4.2, "Configuring Workspace-Level Layer Values from the Studio Developer Role."

39.3.4.1 Configuring Layer Values Globally

The following procedure describes how to configure the `CustomizationLayerValues.xml` file globally for JDeveloper.

Before you begin:

It may be helpful to have an understanding of layers and layer values. For more information, see Section 39.3.4, "How to Configure Customization Layers."

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Section 39.1.3, "Additional Functionality for Customization."

You will also need to complete these tasks:

- Create your customization classes, as described in Section 39.2.1.3, "Creating a Customization Class."
- Make your classes available to JDeveloper, as described in Section 39.2.3.1, "Making Customization Classes Available to JDeveloper at Design Time."

To configure design time customization layer values globally for JDeveloper:

1. Locate and open the `CustomizationLayerValues.xml` file.

   You can find this file in the `jdev` subdirectory of your JDeveloper installation directory (*`jdev_install`*`\jdev\CustomizationLayerValues.xml`).

2. For each layer, enter a `cust-layer` element, as shown in Example 39–7.
3. For each layer value, enter a `cust-layer-value` element, as shown in Example 39–7.
4. Save and close the `CustomizationLayerValues.xml` file.
5. After you have made changes to the global `CustomizationLayerValues.xml` file, you must restart JDeveloper.

39.3.4.2 Configuring Workspace-Level Layer Values from the Studio Developer Role

When configuring layer values for an application, you can use either the **Studio Developer** role or the **Customization Developer** role. Note that when you configure an application-specific `CustomizationLayerValues.xml` file, you can create and modify layer values, but you cannot create additional customization layers. It is not necessary to restart JDeveloper to pick up changes made to the application-specific layer values.

When you create an application-specific `CustomizationLayerValues.xml` file, JDeveloper stores it in an application-level directory (for example, *`workspace-directory`*`\.mds\dt\customizationLayerValues\Customizati`

onLayerValues.xml). You can access this file in the Application Resources panel of the Application Navigator, under the **MDS DT** node.

The following procedure describes how to configure the CustomizationLayerValues.xml file for a specific application from the **Studio Developer** role.

Before you begin:

It may be helpful to have an understanding of layers and layer values. For more information, see Section 39.3.4, "How to Configure Customization Layers."

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Section 39.1.3, "Additional Functionality for Customization."

You will also need to complete these tasks:

- Create your customization classes, as described in Section 39.2.1.3, "Creating a Customization Class."
- Make your classes available to JDeveloper, as described in Section 39.2.3.1, "Making Customization Classes Available to JDeveloper at Design Time."
- Launch JDeveloper using the **Studio Developer** role, and open the application that you want to customize.

To configure design time customization layer values at the workspace level from the Studio Developer role:

1. In the Application Navigator, double-click the adf-config.xml file.

   The adf-config.xml file is in the Application Resources panel of the Application Navigator, under **ADF META-INF** in the **Descriptors** node.

2. In the overview editor, click the **MDS Configuration** navigation tab.
3. On the MDS Configuration page, below the table of customization classes, click the **Configure Design Time Customization Layer Values** link to open the workspace-level CustomizationLayerValues.xml file.

   When you click the link, JDeveloper opens the file in the source editor. If the override file doesn't already exist, JDeveloper displays a confirmation dialog. Click **Yes** to create and open a copy of the global file.

4. Specify layer values as necessary, as described in Section 39.3.4, "How to Configure Customization Layers."
5. Save your changes.

39.3.4.3 Configuring Workspace-Level Layer Values from the Customization Developer Role

You can also configure layer values for an application from the **Customization Developer** role. Note that when you configure an application-specific CustomizationLayerValues.xml file, you can create and modify layer values, but you cannot create additional customization layers. It is not necessary to restart JDeveloper to pick up changes made to the application-specific layer values.

When you create an application-specific CustomizationLayerValues.xml file, JDeveloper stores it in an application-level directory (for example, *workspace-directory*\.mds\dt\customizationLayerValues\Customizati

onLayerValues.xml). You can access this file in the Application Resources panel of the Application Navigator, under the **MDS DT** node.

The following procedure describes how to configure the CustomizationLayerValues.xml file for a specific application from the **Customization Developer** role.

Before you begin:

It may be helpful to have an understanding of layers and layer values. For more information, see Section 39.3.4, "How to Configure Customization Layers."

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Section 39.1.3, "Additional Functionality for Customization."

You will also need to complete these tasks:

- Create your customization classes, as described in Section 39.2.1.3, "Creating a Customization Class."
- Make your classes available to JDeveloper, as described in Section 39.2.3.1, "Making Customization Classes Available to JDeveloper at Design Time."
- Launch JDeveloper using the **Customization Developer** role, and open the application that you want to customize.

To configure design time customization layer values at the workspace level from the Customization Developer role:

1. In the Customization Context window, click the **Override global layer values** link.

   When you click the link, JDeveloper opens the CustomizationLayerValues.xml file in the source editor. If the override file doesn't already exist, JDeveloper displays a confirmation dialog. Click **Yes** to create and open a copy of the global file.

2. Specify layer values as necessary, as described in Section 39.3.4, "How to Configure Customization Layers."

3. Save your changes.

   After you make changes to the application-specific CustomizationLayerValues.xml file while you are in the **Customization Developer** role, any tip layer you have selected in the Customization Context window is deselected. You can then select the desired tip layer.

39.3.5 How to Customize Metadata in JDeveloper

You use the same development procedures and techniques to customize metadata that you use when developing the base application. To implement customizations, however, you must be working in the **Customization Developer** role and specify the customization context by selecting a tip layer and layer value before editing the metadata. For an application to be customizable, customizations must be enabled in your project. For more information, see Section 39.2, "Developing a Customizable Application."

Before you begin:

It may be helpful to have an understanding of the how customization works. For more information, see Section 39.1.1, "Customization and Layers: Use Cases and Examples," and Section 39.3, "Customizing an Application."

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Section 39.1.3, "Additional Functionality for Customization."

You will also need to complete these tasks:

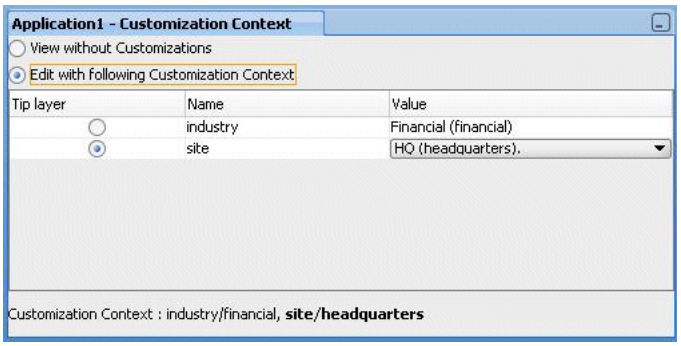
- Create your customization classes, as described in Section 39.2.1.3, "Creating a Customization Class."
- Make your classes available to JDeveloper, as described in Section 39.2.3.1, "Making Customization Classes Available to JDeveloper at Design Time."
- Create a JAR using the procedure described in Section 41.3.2.6, "Adding Customization Classes into a JAR."
- Enable seeded customizations in your application, as described in Section 39.2.4, "How to Enable Seeded Customizations for View Projects," and Section 39.2.5, "How to Enable Seeded Customizations in Existing Pages."
- Configure the applications customization layers, as described in Section 39.3.4, "How to Configure Customization Layers."
- Launch JDeveloper using the **Customization Developer** role, and open the application that you want to customize.

To customize metadata in JDeveloper:

1. In the Customization Context window, select the layer and value for which you want to implement customizations.

   The **Customization Context** (displayed at the bottom of the Customization Context window) changes to reflect your selection, as shown in Figure 39–4.

Figure 39–4 Customization Context Window with site/headquarters Selected as the Tip Layer



> **Note:** The selection you make in the Customization Context window indicates the context for the customizations that you will implement in JDeveloper. This selection does not directly impact the runtime for the application. At runtime, the customization context is returned from your customization classes. For more information, see Section 39.2.1, "How to Create Customization Classes."

2. Edit the metadata as you typically would during development. For example, right-click an entity object and choose **Open**. Then edit the object using the overview editor.

While you use the same techniques for editing metadata during customization that you would during development, certain restrictions apply. For example, some string properties, such as button labels, cannot be edited directly in the Property Inspector: they must be edited using the Select Text Resource dialog or the Expression Builder. For more information about restrictions to editing during customization, see Section 39.3.1, "Introducing the Customization Developer Role." For information about using the Expression Builder, see Section 13.8.1.1, "Opening the Expression Builder from the Property Inspector."

Even though you use the overview editor to implement customizations, you do not make changes to the base metadata file. Your changes are stored separately in a customization metadata file.

Note: To see the uncustomized base metadata, you can select **View without Customizations** in the Customization Context window.

---

3. You can optionally choose **Remove Customization** from the dropdown menu for a property (in the Property Inspector) to clear the existing customization.

Note: In the Property Inspector, tip layer customizations are indicated by an orange icon, while properties that are not customized in the current tip layer are indicated by a green icon. A customization can only be removed in the context in which it was added. So you can remove only those customizations that have an orange indicator in the Property Inspector.

---

4. Choose **Save** from the **File** menu to save your changes.

After you have completed your customizations, you can run and test the customized application.

39.3.6 What Happens When You Customize an Application

When you implement customizations in an application, JDeveloper creates a metadata file for the customizations and a subpackage to store them in.

The metadata file contains the customizations for the customized object, which are applied over the base metadata at runtime. The new metadata file is named the same as the base file for the object with an additional `.xml` extension. For example, if you implement customizations for the `browseOrders.jsff` page, the customization metadata file is named `browseOrders.jsff.xml`. Or if you implement customizations on the `OrderItems` entity object, the base metadata file is named `OrderItems.xml` and the customization metadata file is named `OrderItems.xml.xml`.

The customization metadata files are stored in a subpackage hierarchy that is created at the same level as the object you customize. The first-level package is named `mdssys`, and it contains a package named `cust`. The `cust` package contains a package for each customization layer for which you have implemented customizations.

For example, say you have a base application that has a package called `oracle.fod.model` containing your entity objects, and you have a customization layer named `site` with two layer values: `headquarters` and `remoteoffices`. Then you implement customizations for the `OrderItems` entity object at the

`headquarters` layer value. When you implement these customizations, JDeveloper creates the subpackage hierarchy `oracle.fod.model.mdssys.cust.site.headquarters` and stores the customization metadata files there.

Similarly, for pages in your view controller project, JDeveloper creates a directory structure to store the customization metadata files. For example, if you customize the `BrowseOrders.jsff` page in the Web Content folder of your view controller project, JDeveloper creates the directory structure `mdssys/cust/site/headquarters` under Web Content and stores the customization metadata file there.

39.3.7 How to Customize ADF Library Artifacts in JDeveloper

In the **Customization Developer** role, you can use JDeveloper to customize artifacts in an ADF library. This need can arise when, for example, one development team produces task flows as part of a framework service and makes them available to other teams as an ADF library. Then another development team uses one of the task flows in a consuming application, and needs to fine-tune it to fit the requirements of the application.

You can add an ADF library to your project in the **Customization Developer** role just as you would add it in the **Studio Developer** role. However, in the **Customization Developer** role, content from an ADF library appears as editable to allow you to implement customizations, whereas in the **Studio Developer** role it is read-only. For more information about working with ADF libraries, see Chapter 38, "Reusing Application Components."

Before you begin:

It may be helpful to have an understanding of the how customization works. For more information, see Section 39.1.1, "Customization and Layers: Use Cases and Examples," and Section 39.3, "Customizing an Application."

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Section 39.1.3, "Additional Functionality for Customization."

You will also need to complete these tasks:

- Create your customization classes, as described in Section 39.2.1.3, "Creating a Customization Class."
- Make your classes available to JDeveloper, as described in Section 39.2.3.1, "Making Customization Classes Available to JDeveloper at Design Time."
- Create a JAR using the procedure described in Section 41.3.2.6, "Adding Customization Classes into a JAR."
- Enable seeded customizations in your application, as described in Section 39.2.4, "How to Enable Seeded Customizations for View Projects," and Section 39.2.5, "How to Enable Seeded Customizations in Existing Pages."
- Configure the applications customization layers, as described in Section 39.3.4, "How to Configure Customization Layers."
- Launch JDeveloper using the **Customization Developer** role, and open the application that you want to customize.

To customize an ADF library artifact:

1. In the Application Navigator, click the **Navigator Display Options** icon and choose **Show Libraries**.

This displays the libraries in the Application Navigator, so that you can explore them and access their artifacts.

2. Add the desired library to your project if it is not already shown among the libraries in the Application Navigator.

   For information about how to do this, see Section 38.4, "Adding ADF Library Components into Projects."

3. Customize the artifacts just as you would customize other content in your project.

   For example, you can drag and drop taskflows from a library to `.jspx` or `.jsff` pages in a consuming project, drag and drop taskflows from a library to a page or fragment in another library, drag and drop library content or taskflows from the Resource catalog, drag and drop data controls from Data Controls panel to `.jspx` or `.jsff` pages in a library, edit business components, and drag and drop a data control from a library to the Data Controls panel and then drop to a page in another palette. All of these actions result in customizations of the library.

39.3.7.1 Specifying a Location for ADF Library Customizations

The location where ADF library customizations are stored is *`project-dir`*`\libraryCustomizations` by default. If your workspace contains multiple projects, you should change this to a workspace-level location for each project (for example, *`workspace-dir`*`\.mds\ADFLibraryCustomizations`).

You can change the location of ADF library customizations on the **Project Source Paths > ADF Library Customizations** page of the Project Properties dialog. If you change this location after you have implemented customizations on an ADF library, you must move the customization metadata files to the new location. To do this, use the file system to move the customization metadata (XML) files from the old directory to the new one.

If you have more than one project in the workspace with existing ADF library customizations that need to be moved to the common location, move the customizations to the new location one project at a time. For each project, change the location of ADF library customizations in the Project Properties dialog, and then move the customization metadata files from the old location to the new location. To mitigate conflicts where an ADF library artifact has customizations in more than one project, you have following options:

- From both projects, open the ADF Library artifact for which there are conflicting customizations, and decide upon which customizations you want. Preserve the customizations you want to keep and delete the others.
- If both customizations are important, open the ADF library artifact from the first project and implement the customizations that were previously done in second project. Then save the customizations in the first project and remove the customizations from the second project.

Do not open ADF library customization metadata files from multiple projects in a text editor and merge their contents. This can corrupt the customization metadata file.

Additionally, the location where ADF library customizations are stored is automatically included when you create a MAR deployment profile. If you change this location after you have created a MAR profile, you must also change the corresponding entry in the contributors list on the User Metadata file group page of the Edit MAR Deployment Profile Properties dialog before packaging. Alternatively, you can re-create the MAR profile to pick up this change. For more information about creating a MAR deployment profile, see Section 41.3.2, "How to Create Deployment Profiles."

39.3.8 How to View ADF Library Runtime Customizations from Exported JARs

When working in the JDeveloper Customization Developer role, you can view runtime customizations implemented on ADF library artifacts contained in an exported JAR.

Before you begin:

It may be helpful to have an understanding of the how customization works. For more information, see Section 39.1.1, "Customization and Layers: Use Cases and Examples," and Section 39.3, "Customizing an Application."

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Section 39.1.3, "Additional Functionality for Customization."

You will also need to complete these tasks:

- Create your customizable application, as described in Section 39.2, "Developing a Customizable Application."
- Deploy the application and implement runtime customizations on it, then export the runtime customizations to a JAR file. For more information, see the *Oracle WebCenter Framework Developer's Guide*.
- Launch JDeveloper using the **Customization Developer** role, and open the customizable application.

To make runtime customizations viewable from JDeveloper:

1. From the Application Navigator context menu, choose **Application Properties**.
2. In the Application Properties dialog, click **Customization Libraries**.
3. Click **Browse**, and navigate to the location of the exported JAR file.
4. Select the JAR file and click **Select**.
5. Click **OK** to close the Application Properties dialog.

Now the JAR is available so that JDeveloper can look up customizations on ADF library artifacts. When you open an object that contains ADF library artifacts, JDeveloper looks for customizations in this JAR file and displays them if appropriate. JDeveloper decides what to display for a given artifact in a given customization context as follows:

- If the artifact has *neither* runtime customizations *nor* seeded customizations associated with it, the artifact from the ADF library is displayed.
- If the artifact has only *either* runtime customizations *or* seeded customizations associated with it (but not both), the customized artifact is displayed.
- If the artifact has *both* runtime customizations *and* seeded customizations associated with it, the seeded customization takes precedence and is displayed.

Additionally, when you run the application locally from JDeveloper, the runtime customizations are displayed. However, the runtime customizations are not included in any packaging of the application for deployment.

39.3.9 What Happens When You Customize ADF Library Artifacts

During the development of enterprise applications, there might be artifacts (such as task flows) that can be reused in multiple applications. To facilitate reuse of these common artifacts they are usually packaged into an ADF library and distributed. This allows you to add the ADF library to the list of libraries on which the consuming

application depends on. Then when the application is packaged, the customizations from all such ADF libraries are included in the MAR, which is later deployed to the MDS repository.

> **Note:** The ADF library provider should take care to ensure that no name conflicts arise due to customizations in the library. In the event that name conflicts arise between customizations packaged in an ADF library and the customizations from the consuming project, the customizations from the ADF library are ignored.

When you implement customizations on objects from an ADF library, the customization metadata is stored by default in a subdirectory of the project called `libraryCustomizations`. And although you create ADF library customizations at the project level, they are merged together during packaging to be available at the application level at runtime. Essentially, ADF libraries are JARs that are added at the project level, which map to library customizations being created at the project level. However, although projects map to web applications at runtime, the MAR (which contains the library customizations) is at the EAR level, so the library customizations are seen from all web applications.

Therefore, you can customize an ADF library artifact in only one place in an application for a given customization context (customization layer and layer value). Customizing the same library content in different projects for the same customization context would result in duplication in MAR packaging. To avoid duplicates that would cause packaging to fail, implement customizations for a given library in only one project in your application.

For example, say the ADF library you are using contains a page fragment `text.jsff`. In the consuming application, customize this library page in only one project. By doing so, customizations are available for all projects in the application that consume this library at runtime.

You are also restricted from customizing an object from an ADF library when your project already contains an object with the same name. In case of duplication, you must fix the projects by deleting one of the duplicate documents or deleting one and manually merging the differences into the other.

Similarly, if the ADF library contains seeded customizations for an artifact within a given customization context (customization layer and layer value), you cannot implement customizations for that artifact within the same customization context. In this situation, the ADF library artifact is read-only. You can, however, implement customizations for the artifact within other customization contexts.

For example, say the ADF library you are using contains seeded customizations for the `Headquarters` layer value in the `Site` layer. When you select this as your tip layer in the Customization Context window, the customized objects in that ADF library are read-only. However, if you select `Site/Remote Site 1` as your tip layer, then the objects are customizable.

> **Note:** When the consuming application implements customizations on content from an ADF library, the customizations are written to the local project directories, but they are not automatically injected with the `web-app-root` during packaging. For more information, see Section 41.3.2.2, "Creating a MAR Deployment Profile."

39.3.10 How to Package and Deploy Customized Applications

After you customize the application, you will want to deploy it. Before you deploy the customized application, you must follow the configuration procedures for setting up your MDS repository, as described in the *Oracle Fusion Middleware Administrator's Guide*.

An enterprise application can contain model and view-controller projects, and both types of projects can contain customized metadata. The customized metadata is packaged into a MAR for deployment. By default, the customizations from both types of projects are added to a single MAR. For information about how to create a MAR profile, see Section 41.3.2.2, "Creating a MAR Deployment Profile."

39.3.10.1 Implicitly Creating a MAR Profile

When you use JDeveloper to package an ADF application, JDeveloper creates an auto-generated MAR profile that includes default metadata (such as customizations), when *either* of the following conditions are met.

- The **Enable User Customizations > Across Sessions using MDS** checkbox is selected on the ADF View settings page of the Project Properties dialog for the user interface project.
- The MDS configuration section of the `adf-config.xml` file contains a `<metadata-store-usage>` element that is marked as `deploy-target="true"`, as shown in Example 39–8.

Example 39–8 metadata-store-usage Element in adf-config.xml

```
< . . . >
  <persistence-config>
    <metadata-namespaces>
      <namespace path="/oracle/apps" metadata-store-usage="repos1"/>
    </metadata-namespaces>
    <metadata-store-usages>
      <metadata-store-usage id="repos1" deploy-target="true">
      . . .
      </metadata-store-usage>
    </metadata-store-usages>
  </persistence-config>
< . . . >
```

39.3.10.2 Explicitly Creating a MAR Profile

For customizations created in JDeveloper to take effect in the application when it is deployed, these customizations need to be made available to the application at runtime. There are two techniques you can use to accomplish this:

- Package the customizations along with the application using a MAR profile.

  Create a MAR profile that includes the customization metadata. The MAR profile should be included in the deployed EAR file to ensure that the customizations are available at runtime. Your customization classes must be packaged in the EAR file such that they are in the application-level class loader.

Note: If you have only seeded customizations, you do not need to create a MAR profile to import them into the MDS repository unless you also want to support runtime customizations. If you have seeded customizations and do not have cross-session persistence enabled, the seeded customizations will be packaged in the EAR file by default and loaded from the class path.

- Import the customizations to the runtime repository used by the application.

  You typically use this approach if customizations to library metadata need to be applied to an application that is deployed separately. Using this approach, you package the customizations into a JAR file and then use the `importMetadata` WLST command to import them to the MDS runtime repository. For more information about this and other WLST commands, see the *Oracle Fusion Middleware WebLogic Scripting Tool Command Reference*.

If your application has customizations on objects from an ADF library, the customization metadata is implicitly included when you create the MAR profile. If you change the location of ADF library customizations in the Project Properties dialog, you must re-create the MAR profile before packaging.

If you plan to use design time at runtime capabilities to add or edit resource keys in the override bundle at runtime, the override bundle must be packaged as part of the MAR profile. By default, the override bundle is packaged as part of the auto-generated MAR profile if the application contains seeded customizations. However, if the application doesn't contain seeded customizations, you must explicitly create the MAR deployment profile to package the override bundle. When the MAR profile is created explicitly, the override bundle is added and included by default as part of the user metadata.

When you package and deploy the completed customized application, you should do so from the **Studio Developer** role, rather than from the **Customization Developer** role. For information about how to create a MAR profile, see Section 41.3.2, "How to Create Deployment Profiles."

39.3.11 What Happens at Runtime in a Customized Application

At runtime, the application applies the customization metadata files over the base application in the order of precedence defined in the `cust-config` section of the `adf-config.xml` file.

The layer value is retrieved from the customization class at runtime and evaluated in the context the application is running, and the appropriate customizations for that layer value are applied.

39.3.12 What You May Need to Know About Customized Applications

When you are customizing an application, you might be using integrated source control or customizing resource strings. When you use these features, there is additional information you need to know.

39.3.12.1 Customization and Integrated Source Control

When working in the **Customization Developer** role, your source control integration complements the process of customization. If JDeveloper is configured to automatically check out and add new files to source control and you attempt to

customize a base document that is available from a source control system, JDeveloper behaves in the following way:

- If the corresponding customization file is not already available, then a new customization file is created in source control and the customizations are written to it.
- If the corresponding customization file exists, it is checked out and customizations are written to it.
- If the corresponding customization file exists and it is already checked out or not yet in version control, customizations are written to it without any further version control operation.

Since the base document is not modified in the **Customization Developer** role, the base document is not checked out.

If JDeveloper is not configured to automatically check out or add new files to source control, you must manually make the customization files editable and check in newly created customization files to source control. For more information about using source control in JDeveloper, see section Section 1.4.2, "Using a Source Control System."

39.3.12.2 Editing Resource Bundles in Customized Applications

During the course of customizing your application, you might want to customize the content to use different resource bundle keys or define and use new resource keys.

You can open a customizable application in the **Customization Developer** role and use the Property Inspector to customize the usages of resource bundle strings. You can change a document to use another already existing resource key in a resource bundle, or create a new resource. For more information about resource bundles, see Section 4.7, "Working with Resource Bundles."

New resource keys (created in the **Customization Developer** role) are saved to an application-level override bundle (in XLIFF format), and JDeveloper adds an entry to the `adf-config.xml` file like the one shown in Example 39–9 to configure the application-level override bundle.

You must also configure the `adf-config.xml` file to support the overriding of the base resource bundle. As shown in Example 39–9, you must tag the `bundleId` element with `override="true"` to make it overrideable. After it is marked as overriden, customizations of that bundle are stored in the application's override bundle.

Example 39–9 adf-resourcebundle-config Section in adf-config.xml

```
<adf-resourcebundle-config
xmlns="http://xmlns.oracle.com/adf/resourcebundle/config">
  <applicationBundleName>
    path-to-resource-bundle/bundle-name
  </applicationBundleName>
  <bundleList>
    <bundleId override="true">
      package.BundleID
    </bundleId>
  </bundleList>
</adf-resourcebundle-config>
```

Note: If an application is not configured for customization, you can open it in the **Customization Developer** role and define new resource keys by choosing **Edit Resource Bundles** from the **Application** menu. However, you cannot change a document to use the new resource keys if it is not configured for customization.

39.4 Extended Metadata Properties

Extended metadata is data that describes the metadata content. The extended metadata file contains additional information about the metadata file. One use of this extended information is to identify which parts of the metadata can be customized at runtime (design time at runtime customizations) and who can customize them. For more information about this use of extended metadata properties, see Section 39.4.2, "How to Enable Customization for Design Time at Runtime."

You can open a metadata file (such as a `.jspx` file) in JDeveloper and use the Property Inspector to view and edit its extended metadata properties. When you open a metadata file, its extended metadata properties are displayed in the Property Inspector. These properties can be edited to add metadata information at either of the following levels:

- File-level: These properties are displayed in the Property Inspector when the root element is selected in the Structure window.
- Element-level: These properties are displayed in the Property Inspector when an element is selected in the Structure window. The selected element should have a non-null identifier.

Extended metadata properties are supported for file types that support customizations and can be packaged in a MAR, such as `.jsff` and `.jspx` files.

Extended metadata for a metadata document is stored in an associated resource description framework (RDF) file. RDF is a W3C standard used to define an XML framework for defining metadata. The RDF file associated with the metadata document is created when the first property value is specified using the Property Inspector. Extended metadata properties are editable only when JDeveloper is in the **Studio Developer** role. RDF files are read-only in the **Customization Developer** role.

The RDF file is stored in the `mdssys` directory. For example, if the metadata being described is stored in the file system as `/myapp/data/page1.jspx`, the corresponding extended metadata document would be stored as `/myapp/data/mdssys/mdx/page1.jspx.rdf`. The extended metadata document must then be packaged with the corresponding metadata base file and added to the same deployment profile. For more information about creating a MAR deployment profile, see Section 41.3.2, "How to Create Deployment Profiles."

Note: Don't edit the extended metadata documents directly. Use the Property Inspector.

39.4.1 How to Edit Extended Metadata Properties

You can use extended metadata properties to provide additional metadata information that is not covered in the metadata file (such as a `.jspx` file). When you open the metadata file in JDeveloper, extended metadata properties are displayed in the Property Inspector, which you can use to edit these properties when you are using JDeveloper in the **Studio Developer** role.

For example, suppose you want to deliver a metadata file in some form to external customers. Along with metadata file, you need to provide additional information about the file, such as the creator, subject, description, format, and rights. You can accomplish this by creating an extended metadata property file for your metadata file.

To edit extended metadata properties:

1. Launch JDeveloper using the **Studio Developer** role.
2. In JDeveloper, open the appropriate application and project.
3. In the Application Navigator, select the object for which you want to edit extended metadata properties.
4. In the Structure window, select the appropriate element (typically the root element).
5. In the Property Inspector, expand the appropriate node to edit the properties.

   To display the Property Inspector with the values for the selected component, choose **Property Inspector** from the **View** menu.
6. Edit the value for the desired property and press Enter.
7. Choose **Save** from the **File** menu to save your work.

If you have edited extended metadata properties for a metadata file, you must package your extended metadata (or RDF) files with the metadata files when you deploy the application.

39.4.2 How to Enable Customization for Design Time at Runtime

You can also use extended metadata properties to provide information about which parts of the metadata can be customized at runtime and who can customize the metadata content.

By default, all components that you can add to a `.jspx` page are preconfigured to allow runtime customization to support implicit user personalization (such as changing the order of columns in a table). All pages and fragments that you create are also configured to allow customization by default. For more information about user customization, see Chapter 40, "Allowing User Customizations at Runtime."

Components that are preconfigured to allow customization need no further modification to enable design time at runtime customizations. If you use them in a `.jspx` page, then they are customizable by default.

Depending on the requirements of your application, you will need to modify customization properties in the following situations:

- For components that are configured to allow customization, you can override the default settings to disallow customization.
- For components that are not configured to allow customization and metadata objects (such as `.jspx` pages), you can override the settings to allow customization.
- For components that are configured to allow customization, you can optionally restrict who is allowed to perform customizations at runtime.

If you edit customization properties, you must package your extended metadata (RDF) files with the metadata files when you deploy the application.

Additionally, you can restrict customizations on a entire subtree of components by marking the root of the subtree to disallow customizations. You can then mark

individual components of that subtree as customizable to allow customizations for those specific components.

39.4.2.1 Editing Customization Properties in the Property Inspector

In the Customization group in the Property Inspector, there are two properties that allow you to specify whether customizations for an object are permitted at runtime and who is permitted to do them. The **CustomizationAllowed** property can be set on any element to specify whether or not it can be customized. The **CustomizationAllowedBy** property controls which users can customize the element. These settings are not enforced when you are implementing seeded customizations using JDeveloper, but are instead enforced when the application is customized at runtime (design time at runtime customizations).

For example, say you have a `.jspx` page with a form that contains two panels, and you need to allow runtime customization for content in Panel1, but not Panel2. You would set **CustomizationAllowed** to `true` for Panel1, and set it to `false` for Panel2. If you need to allow runtime customization on an entire page, you would set **CustomizationAllowed** to `true` for the `.jspx` page root.

To edit customization properties:

1. Launch JDeveloper using the **Studio Developer** role.
2. In JDeveloper, open the appropriate application and project.
3. In the Application Navigator, select the object for which you want to edit customization properties.
4. In the Structure window, select the appropriate element.
5. In the Property Inspector, expand the **Customization** node to edit the customization properties.

   To display the Property Inspector with the values for the selected component, choose **Property Inspector** from the **View** menu.
6. Edit the property value and press Enter.

   For example, to allow runtime customizations on a `.jspx` file, select the file and set the **CustomizationAllowed** property to `true`.
7. Choose **Save** from the **File** menu to save your work.

39.4.2.2 Using a Standalone Annotations File to Specify Type-Level Customization Properties

You can optionally use a standalone annotations file to specify whether or not customizations are allowed for a given type of component. This file allows you to set customization properties on element types, rather than on instances. You can override the setting for a given type by explicitly setting `true` or `false` on a specific instance. Example 39–10 shows the contents of a sample standalone annotations file.

Example 39–10 Contents of a Standalone Annotations file

```
<?xml version="1.0" encoding="UTF-8"?>
<grammarMetadata xmlns="http://xmlns.oracle.com/bali/xml/metadata"
                 xmlns:md="http://xmlns.oracle.com/bali/xml/metadata"
                 xmlns:mmd="http://xmlns.oracle.com/bali/xml/metadata/model"
                 xmlns:mds="http://xmlns.oracle.com/mds"
                 namespace="http://www.oracle.com/mds/restrictCustomizations">
  <elementMetadata elementName="orderDetail">
```

```
    <attributeMetadata attributeName="itemTypeRef">
      <mds:customizationAllowedBy>sales admin</mds:customizationAllowedBy>
    </attributeMetadata>
    <attributeMetadata attributeName="description">
      <mds:customizationAllowed>true</mds:customizationAllowed>
      <mds:customizationAllowedBy>hr</mds:customizationAllowedBy>
    </attributeMetadata>
    <attributeMetadata attributeName="title">
      <mds:customizationAllowed>true</mds:customizationAllowed>
    </attributeMetadata>
  </elementMetadata>

  <elementMetadata elementName="bankTransfer">
    <mds:customizationAllowed>false</mds:customizationAllowed>
  </elementMetadata>

  <elementMetadata elementName="order">
    <mds:customizationAllowed>true</mds:customizationAllowed>
    <attributeMetadata attributeName="paymentRef">
      <mds:customizationAllowedBy>sales</mds:customizationAllowedBy>
    </attributeMetadata>
    <elementMetadata elementName="itemDetail">
      <attributeMetadata attributeName="itemsRef">
        <mds:customizationAllowedBy>hr</mds:customizationAllowedBy>
      </attributeMetadata>
    </elementMetadata>
  </elementMetadata>

</grammarMetadata>
```

If you use a standalone annotations file, you must register it with MDS in the `adf-config.xml` file. Create a `mdsc:standalone-definitions` section in the `mdsc:type-config` section of the `adf-config.xml` file, as shown in Example 39–11.

Example 39–11 standalone-definitions Section in adf-config.xml

```
<?xml version="1.0"?>
<adf-config xmlns="http://xmlns.oracle.com/adf/config"
            xmlns:mdsc="http://xmlns.oracle.com/mds/config">
 <mdsdata>
  <mdsc:mds-config version="11.1.1.000" >
   <mdsc:type-config>
     <mdsc:type-definitions>
       <mdsc:url>jar:file:/c:/jdev/lib/oafwk.jar!/oracle/apps/schemas/oa.xsd</mdsc:url>
       <mdsc:mds>/oracle/mds/TypeSeven.xml</mdsc:mds>
       <mdsc:mds>/oracle/mds/TypeEight.xml</mdsc:mds>
       <mdsc:file>typefile7.xsd</mdsc:file>
       <mdsc:file>typefile8.xsd</mdsc:file>
      </mdsc:type-definitions>
     <mdsc:standalone-definitions>
       <mdsc:url>jar:file:/c:/jdev/lib/oafwk.jar!/oracle/apps/schemas/oa.xsd</mdsc:url>
       <mdsc:mds>/oracle/mds/standAloneSeven.xml</mdsc:mds>
       <mdsc:mds>/oracle/mds/standAloneEight.xml</mdsc:mds>
       <mdsc:file>standAloneFile7.xsd</mdsc:file>
       <mdsc:file>standAloneFile8.xsd</mdsc:file>
     </mdsc:standalone-definitions>
    </mdsc:type-config>
     ....
  </mdsc:mds-config>
```

```
 </mdsdata>
</adf-config>
```

39.5 Enabling Runtime Modification of Customization Configuration

You can prepare your customized application to accept overrides to the customization configuration (the `cust-config` section of the `adf-config.xml` file) at runtime on a per-session basis, thus allowing the user to change the way customizations are applied for a given session (or web request).

Consider a scenario where an application is configured with a `site` layer and a `user` layer and you want to make design time at runtime customizations to the `site` layer. If you use the application's customization configuration (defined in the `adf-config.xml` file), any customizations that you implement are applied to the `user` layer. So, you need to be able to adjust the customization configuration for a given session to allow your customizations to be applied to the `site` layer.

Or you have a requirement that an administrator wants to see the base metadata document with the `site` layer customizations alone. For cases like this, you need to specify a modified customization configuration, other than what was originally specified in application's `adf-config.xml` file.

For each web request, Oracle ADF creates an MDS session. For any MDS customization configuration modifications that apply to a session (web request), the user could programmatically provide modified MDS session options with a new customization configuration to Oracle ADF that would be applied on top of the original MDS configuration while creating a MDS session.

To implement this functionality, use the following ADF interfaces for `sessionOptionsFactory`:

- `oracle.adf.share.mds.SessionOptionsFactory` is the interface you use to specify modified MDS session options for a web request in an ADF application.

  ```
  SessionOptionsFactory :: oracle.mds.core.SessionOptions
  createSessionOptions(oracle.mds.core.SessionOptions defaultOptions)
  ```

  You implement this method to return the modified MDS `sessionOptions` to ADF.

- `oracle.adf.share.config.ConfigUtils` is the public class you can use to register your session options factory with ADF.

  ```
  ConfigUtils :: public static void setSessionOptionsFactory(ADFContext context,
  SessionOptionsFactory factory)
  ```

See your ADF API documentation for further detail on these interfaces.

You register your `sessionOptionsFactory` with ADF in a filter implementation so that ADF can get modified session options from your implementation before the MDS session is created in request lifecycle.

Example 39–12 shows how you can implement `sessionOptionsFactory`. This example sets a modified customization configuration for the session to use the `site` customization layer alone regardless of the customization configuration specified in the `adf-config.xml` file. See the Javadoc for `oracle.mds` for more information.

Example 39–12 Sample sessionOptionsFactory Class

```
package mycompany;
```

```
import oracle.adf.share.mds.SessionOptionsFactory;
import oracle.mds.config.CustClassListMapping;
import oracle.mds.config.CustConfig;
import oracle.mds.config.MDSConfigurationException;
import oracle.mds.core.SessionOptions;
import oracle.mds.cust.CustClassList;
import oracle.mds.cust.CustomizationClass;

public class MySessionOptionsFactory implements SessionOptionsFactory {
  public MySessionOptionsFactory() {
    super();
  }
  /**
   * Called to allow the application code to create a new SessionOptions object.
   * The application code should make sure to read the values from the
   * defaultOptions object as part of contruction of their new object and make
   * sure they only override the intended values.
   * @param defaultOptions
   * @return modified MDS session options
   */
  public SessionOptions createSessionOptions(SessionOptions defaultOptions) {
    // create new mds Customization configuration
    CustConfig custconfig = null;

    // create customization class array. Just put SiteCC implementation as we
    // wish to apply site customizations alone.
    CustomizationClass[] custclassarray = new CustomizationClass[] {new SiteCC()};

    CustClassList custclasslist = new CustClassList(custclassarray);

    // specify the base metdata package namespace mapping on which site
    // customizations would apply
    CustClassListMapping[] mappings =
        new CustClassListMapping[] {new CustClassListMapping("mycompany/package",
                                                  null, null, custclasslist)};
    // create new customization configuration
    try{
      custconfig = new CustConfig(mappings);
    }
    catch (Exception ex){
      //do nothing
    }

    // now return modified sessionOptions to ADF with new mds customization
    // configuration. Only use newly created customization configuration in here.
    // For rest of option, use whatever available in defaultOptions.
    return new SessionOptions(defaultOptions.getIsolationLevel(),
                              defaultOptions.getLocale(),
                              custconfig,
                              defaultOptions.getVersionContext(),
                              defaultOptions.getVersionCreatorName(),
                              defaultOptions.getCustomizationPolicy(),
                              defaultOptions.getServletContextAsObject());
  }
}
```

40

Allowing User Customizations at Runtime

This chapter describes how to use the ADF Faces change persistence framework to create JSF pages that users can customize at runtime.

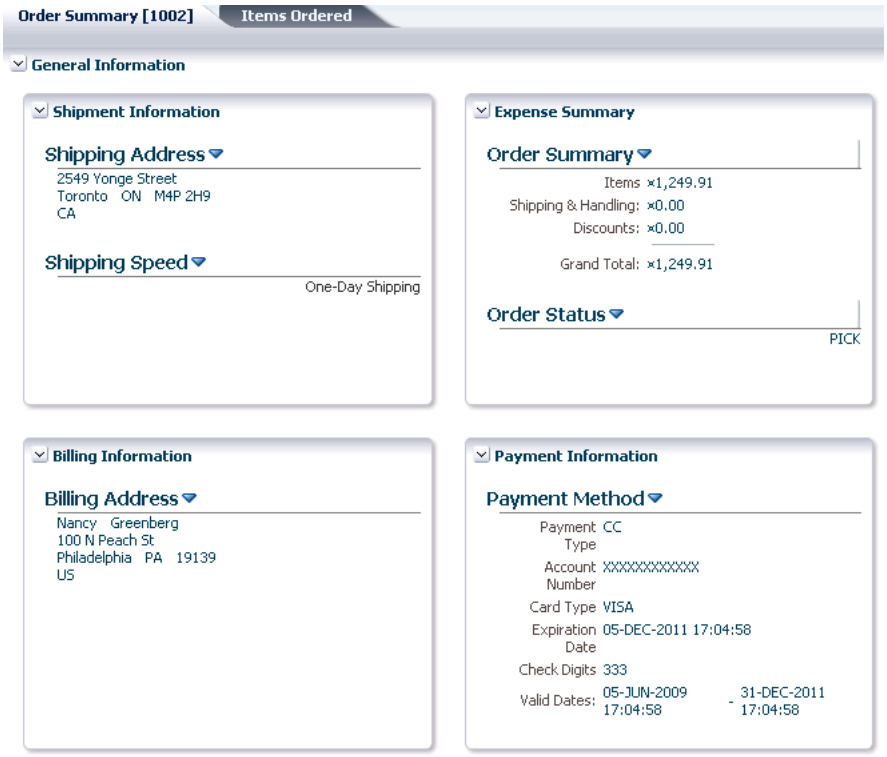
This chapter includes the following sections:

- Section 40.1, "About User Customizations"
- Section 40.2, "Enabling Runtime User Customizations for a Fusion Web Application"
- Section 40.3, "Configuring User Customizations"
- Section 40.4, "Controlling User Customizations in Individual JSF Pages"
- Section 40.5, "Implementing Custom User Customizations"
- Section 40.6, "Creating Implicit Change Persistence in Custom Components"

40.1 About User Customizations

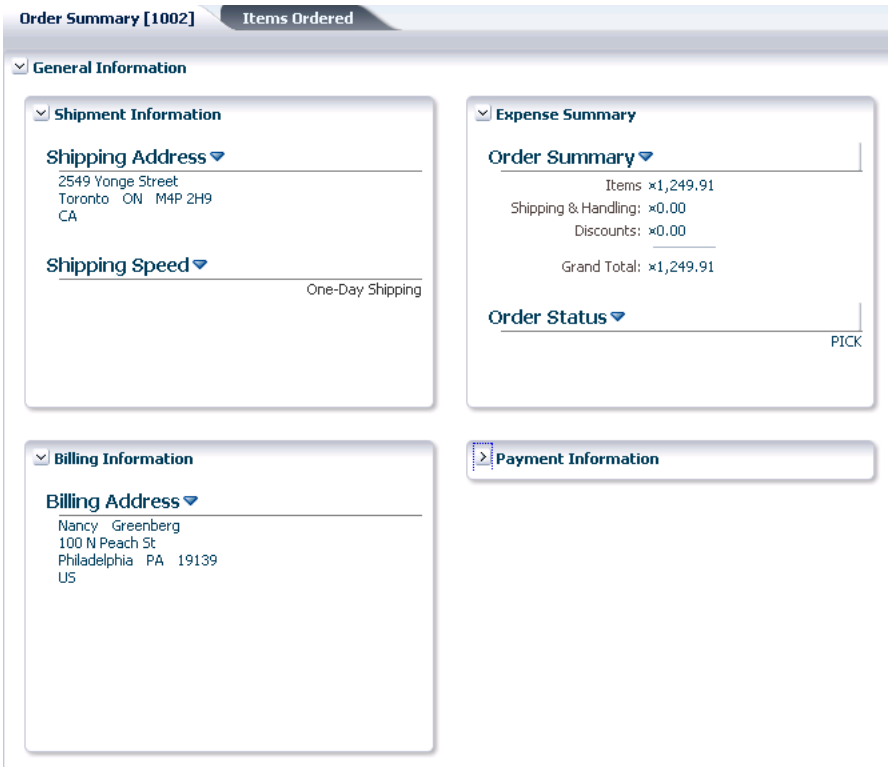
Certain ADF Faces components have attributes that can be saved for a specific user. For example, the value of the `disclosed` attribute on a `panelBox` component can be saved for a specific user during the current session. The `myOrders` page in the StoreFront module application contains four `panelBox` components that display order information. By default, they are expanded, as shown in Figure 40–1.

Figure 40–1 panelBox Components Are Expanded by Default



However, suppose a user decides to collapse one of the boxes, as shown in Figure 40–2.

Figure 40–2 panelBox Component Remains Collapsed



Because this application is configured to allow user customizations, then during the user's session, anytime that user returns to the page, the Payment Information box remains collapsed. You need only to enable user customizations for the project in order for these changes to be persisted to the user's session.

Table 40–1 shows the attribute value changes persisted by an ADF Faces application, after you configure the application to allow user customizations.

Table 40–1 Implicitly Persisted Attribute Values

Component	Attribute	Affect at Runtime

`panelBox`
`showDetail`
`showDetailHeader`
`showDetailItem`
	`disclosed`	Users can display or hide content using an icon in the header. Detail content will either display or be hidden, based on the last action of the user.

`showDetailItem` (used in a `panelAccordion` component)	`flex`	The heights of multiple `showDetailItem` components are determined by their relative value of the flex attribute. The `showDetailItem` components with larger flex values will be taller than those with smaller values. Users can change these proportions, and the new values will be persisted.
`showDetailItem` (used in a `panelAccordion` component)	`inflexibleHeight`	Users can change the size of a panel, and that size will remain.
`panelSplitter`	`collapsed`	Users can collapse either side of the splitter. The collapsed state will remain as last configured by the user.
`panelSplitter`	`splitterPosition`	The position of the splitter in the panel will remain where last moved by user.
`richTextEditor`	`editMode`	The editor will display using the mode (either WYSIWYG or source) last selected by the user.
`calendar`	`activeDay`	The day considered active in the current display will remain the active day.
`calendar`	`view`	The view (day, week, month, or list) that currently displays activities will be retained.
`panelWindow` `dialog`	`contentHeight`	Users can change the height of a `panelWindow` or `dialog` popup component, and that height will remain.
`panelWindow` `dialog`	`contentWidth`	Users can change the width of a `panelWindow` or `dialog` popup component, and that width will remain.

Table 40–1 (Cont.) Implicitly Persisted Attribute Values

Component	Attribute	Affect at Runtime
activeCommandToolbarButton commandButton commandImageLink commandLink commandMenuItem commandNavigationItem commandToolbarButton	windowHeight	When users change the contentHeight attribute value of a panelWindow or dialog component, any associated windowHeight value on a command component is also changed and will remain.
activeCommandToolbarButton commandButton commandImageLink commandLink commandMenuItem commandNavigationItem commandToolbarButton	windowWidth	When users change the contentWidth attribute value of a panelWindow or dialog component, any associated windowWidth value on a command component is also changed and will remain.
column	displayIndex	ADF Faces columns can be reordered by the user at runtime. The displayIndex attribute determines the order of the columns. (By default, the value is set to -1 for each column, which means the columns will display in the same order as the data source). When a user moves a column, the value on each column is changed to reflect the new order. These new values will be persisted.
column	frozen	ADF Faces columns can be frozen so that they will not scroll. When a column's frozen attribute is set to true, all columns before that column (based on the displayIndex value) will not scroll. When you use the table with a panelCollection component, you can configure the table so that a button appears that allows the user to freeze a column.
column	noWrap	The content of the column will either wrap or not. You need to create code that allows the user to change this attribute value. For example, you might create a context menu that allows a user to toggle the value from true to false.
column	selected	The selected column is based on the column last selected by the user.

Table 40–1 (Cont.) Implicitly Persisted Attribute Values

Component	Attribute	Affect at Runtime
`column`	`visible`	The column will either be visible or not, based on the last action of the user. You will need to write code that allows the user to change this attribute value. For example, you might create a context menu that allows a user to toggle the value from `true` to `false`.
`column`	`width`	The width of the column will remain the same size as the user last set it.
`table`	`filterVisible`	ADF Faces tables can contain a component that allows users to filter the table rows by an attribute value. For a table that is configured to use a filter, the filter will either be visible or not, based on the last action of the user. You will need to write code that allows the user to change this attribute value. For example, you might create a button that allows a user to toggle the value from `true` to `false`.

`dvt:areaGraph`
`dvt:barGraph`
`dvt:bubbleGraph`
`dvt:comboGraph`
`dvt:horizontal BarGraph`
`dvt:lineGraph`
`dvt:scatterGraph`
	`timeRangeMode`	The time range for the data displayed on a graph time axis can be specified for all data visualization graph components. By default, all data is displayed. The time range can also be set for a relative time range from the last or first data point, or an explicit time range. You will need to write code that allows the user to change this attribute value. For example, you might create a dropdown list to choose the time range for a graph.

`dvt:ganttLegend`	`visible`	The legend for data visualization project, resource utilization, and scheduling Gantt chart components will either be visible or not inside the information panel. You will need to write code that allows the user to change this attribute value, for example, a hide and show button to display the legend.
`dvt:hierarchyViewer`	`layout`	The data visualization hierarchy viewer component supports nine hierarchy layout options including a top-to-bottom vertical, tree, circle, radial, and so on. Users can change the layout in the map control panel and the last selected layout will be retained.

Table 40–1 (Cont.) Implicitly Persisted Attribute Values

Component	Attribute	Affect at Runtime
dvt:map	mapZoom	This data visualization geographic map component attribute specifies the beginning zoom level of the map. The zoom levels are defined in the map cache instance as part of the base map. You will need to write code that allows the user to change this attribute value.
dvt:map	srid	This data visualization geographic map component attribute specifies the srid (spatial reference id) of all the coordinates of the map, which includes the center of the map, defined by startingX and startingY, and all the points in the point theme. You will need to write code that allows the user to change this attribute value.
dvt:map	startingX, startingY	This data visualization geographic map component attribute specifies the X and Y coordinate of the center of the map. The srid for the coordinate is specified in the srid attribute. If the srid attribute is not specified, this attribute assumes that its value is the longitude of the center of the map. You will need to write code that allows the user to change this attribute value.
dvt:projectGantt dvt:resource UtilizationGantt dvt:schedulingGantt	splitterPosition	The position of the splitter in the panel will remain where last moved by user.
dvt:timeAxis	scale	Data visualization components for project, resource utilization, and scheduling Gantt charts use this facet to specify the major and minor time axes in the Gantt chart. The time scale (twoyears, year, halfyears, quarters, twomonths, months, weeks, twoweeks, days, sixhours, threehours, hours, halfhours, quarterhours) can be set by the user using the menu bar **View** menu and the selection will be retained. Note that a custom time scale can also be named for this component value.
dvt:timeSelector	explicitStart, explicitEnd	Data visualization area, bar, combo, line, scatter, and bubble graph components use this child tag attribute to specify the explicit start and end dates for the time selector. Only value-binding is supported for this attribute. You will need to write code that allows the user to change this attribute value.

40.1.1 Runtime User Customization Use Cases and Examples

You can configure an application so that the value of the attributes listed in Table 40–1 can be persisted across sessions using the MDS repository. For example, if the StoreFront module allowed persistence to the MDS repository, then a user could collapse the Payment Information box, as shown in Figure 40–2, and it would be collapsed the next time that user entered the application.

> **Note:** Before you can enable persistence to the repository, you must first follow all MDS configuration procedures as documented in the *Oracle Fusion Middleware Administrator's Guide*.

Along with the automatic persistence available through ADF Faces, you can create your own custom user customization capabilities for the following types of changes:

- Changing an attribute value
- Adding or removing a facet
- Adding or removing a child component
- Reordering child components
- Moving a child component to a different parent

If you want to create these types of custom user customizations, you need to add code (for example, in an event handler) that will call the APIs to handle the persistence.

Enabling an application to use the change persistence framework requires that you first enable your application to allow user customizations. Part of this process is determining where those changes should be persisted, either to the session or the MDS repository.

If you choose to persist changes to the session, by default all values as shown in Table 40–1 will be saved for the user's session. However if you choose to persist changes to a repository, you must explicitly configure which of these attribute values will be persisted to the repository. Instead of persisting all these attribute values, you can restrict changes so that only certain attribute value changes for a component are persisted, or so that only specific instances of the components persist changes.

> **Note:** You cannot persist changes to a component that is contained inside (anywhere in the subtree) of `af:forEach` or `af:iterator` tags using the `addComponentChange()` method. While such structure results in multiple copies of a component in the view tree, each component has only a single representation in the JSP document. However, the `addDocumentChange()` method does allow persistence for such components. For more information, see the discussion of the `DocumentChange` and `ComponentChange` classes in Section 40.5.1, "Change Persistence Framework API."

For any applications that persist changes to an MDS repository, when you deploy your application, you must create a metadata archive (MAR) profile in the application's EAR assembly. For more information, see Section 41.3.2, "How to Create Deployment Profiles."

40.1.2 Additional Functionality for Runtime User Customization

You may find it helpful to understand other features before you start working with user customizations. Following are links to other functionality that may be of interest.

- For information about the MDS architecture and metadata repositories (database- and file-based) and archives (EAR, MAR), see to the section about managing the MDS repository in the *Oracle Fusion Middleware Administrator's Guide*.
- The MDS framework allows you to create customizable applications that can be customized and subsequently deployed by a customer. For information, see Chapter 39, "Customizing Applications with MDS."

40.2 Enabling Runtime User Customizations for a Fusion Web Application

Enabling an application to allow user customizations (whether for the default changes that some ADF Faces components provide, or for custom capabilities that you create) requires that you configure your application to use the change persistence framework and that you also determine where those changes should be persisted (either the session or the MDS repository).

> **Note:** If you are planning on persisting changes to the MDS repository, before configuring an ADF Faces application to use change persistence, you must first follow all MDS configuration procedures as documented in the *Oracle Fusion Middleware Administrator's Guide*.

40.2.1 How to Enable User Customizations

You enable your application to use the change persistence framework by editing the `web.xml` and `adf-config.xml` files.

Before you begin:

It may be helpful to have an understanding of the features of runtime user customization. For more information, see Section 40.2, "Enabling Runtime User Customizations for a Fusion Web Application."

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Section 40.1.2, "Additional Functionality for Runtime User Customization."

You will need to launch JDeveloper using the **Studio Developer** role, and open (or create) the application in which you want to enable runtime user customization.

To enable user customizations:

1. Double-click the web project in your application.
2. In the Project Properties dialog, select the **ADF View** node.
3. On the ADF View page, select the **Enable User Customizations** checkbox. If you want the changes to be persisted to only the session, select the **For Duration of Session** radio button. If you want the changes to persist to the MDS repository, select **Across Sessions Using MDS**.
4. If you choose to persist to the repository, you now need to declare each component tag and associated attribute values that you want persisted to the repository (if you choose to persist only to the session, all values will be persisted).

For procedures to accomplish this task, see Section 40.3, "Configuring User Customizations." Once that configuration is complete, you can override those settings on a per component instance basis. For procedures, see Section 40.4, "Controlling User Customizations in Individual JSF Pages."

Note: If you have created custom user customization capabilities as documented in Section 40.5, "Implementing Custom User Customizations," then you also need to declare those attribute values or operations.

40.2.2 What Happens When You Enable User Customizations

When you elect to save changes only to the session, JDeveloper adds the `CHANGE_PERSISTENCE` context parameter to the `web.xml file`, and sets the value to `session`. This context parameter registers the `ChangeManager` class that will be used to handle persistence. If you instead elect to save the changes to the MDS repository, the value is set to `oracle.adf.view.rich.change.FilteredPersistenceChangeManager`, as shown in Example 40–1.

Example 40–1 Context Parameter in web.xml Used For Change Persistence

```
<context-param>
  <param-name>org.apache.myfaces.trinidad.CHANGE_PERSISTENCE</param-name>
  <param-value>
    oracle.adf.view.rich.change.FilteredPersistenceChangeManager
  </param-value>
</context-param>
```

Tip: If needed, you can manually set this value to `oracle.adf.view.rich.change.MDSDocumentChangeManager` if you do not want any customizations to be restricted based on configurations in the `adf-config.xml` file or on the individual JSF pages, and you always want the changes to be persisted to the MDS repository and not the session.

When you elect to persist to the repository, JDeveloper also does the following:

- Adds the following JARs to the class path if they don't exist:
  - `javatools-nodep.jar`
  - `facesconfigmodel.jar`
  - `taglib.jar`
- Adds another context parameter to `web.xml` to register the `MDSJSPProviderHelper` class to handle merging MDS customization documents with the base JSP document, as shown in Example 40–2

Example 40–2 Context Parameter in web.xml Used For Merging Changes

```
<context-param>
  <param-name>oracle.adf.jsp.provider.0</param-name>
  <param-value>oracle.mds.jsp.MDSJSPProviderHelper</param-value>
</context-param>
```

- Adds the ADF Faces Change Manager Runtime 11 library to the project.

- In the `adf-config.xml` descriptor file, sets the `persistent-change-manager` element to the `MDSDocumentChangeManager`, which is the class that will be used to persist the changes. Example 40–3 shows the configuration for persisting to the MDS repository.

Example 40–3 Registered ChangeManager Class for Restricted Change Persistence

```
<persistent-change-manager>
  <persistent-change-manager-class>
    oracle.adf.view.rich.change.MDSDocumentChangeManager
  </persistent-change-manager-class>
</persistent-change-manager>
```

- Creates JSF JSP pages as XML documents. For more information, see Section 40.4, "Controlling User Customizations in Individual JSF Pages."

Additionally, when you elect to save the changes to the MDS repository, and you include facelets in your application, JDeveloper adds the `faceletCache` context parameter to the `web.xml` file. The value of this parameter is set to `oracle.adfinternal.view.faces.facelets.rich.MDSFaceletCache`, as shown in Example 40–4.

Example 40–4 Context Parameter in web.xml Used For Facelet Caching

```
<context-param>
  <param-name>com.sun.faces.faceletCache</param-name>
  <param-value>oracle.adfinternal.view.faces.facelets.rich.MDSFaceletCache</param-value>
</context-param>
```

The facelets cache enhances the performance of your application and the cache capacity can be configured in MDS on the server. For more information, see the *Oracle Fusion Middleware Administrator's Guide*.

40.3 Configuring User Customizations

If you choose to persist changes to an MDS repository, you must decide which of the attribute values that are by default persisted to the session (as shown in Table 40–1) should also be persisted to the repository. Alternatively, you can configure which changes you do not want persisted.

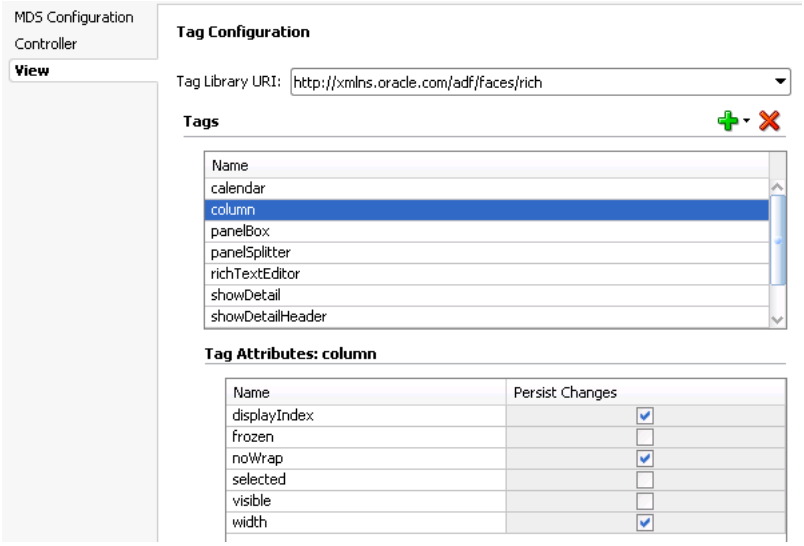
> **Tip:** Often, a system administrator is the one to set the configurations in the `adf-config.xml`. The `persist` and `dontPersist` attributes on a component allow page authors to override that setting as needed.

For example, suppose you decide that you don't want the value for the `width` attribute on columns to be persisted to the repository, but you do want all other default attribute changes for columns to be persisted. You must explicitly set the other default column values that you want to be persisted, and you also must explicitly configure the application to NOT persist the `width` attribute.

Note: If you have created custom user customization capabilities as documented in Section 40.5, "Implementing Custom User Customizations," then you must explicitly declare those attribute values or operations as well.

You set (and unset) these values using the overview editor for the `adf-config.xml` file. Figure 40–3 shows the overview editor where only certain attribute values for the `column` component will be persisted.

Figure 40–3 Overview Editor for the adf-config.xml File



Once set, you can override persistence for a specific component on a page. For example, suppose you want to disallow change on the `width` attribute on only one table's columns. You want the rest of the tables in the application to persist changes to that attribute. You would configure the columns to globally persist changes to the `width` attribute, but then for that one table, you would override the global configuration directly on the JSF page. For more information, see Section 40.4, "Controlling User Customizations in Individual JSF Pages."

Note: If you've enabled just session persistence, then all attribute values shown in Table 40–1 will be persisted to the session. There is no way to override this either globally or on an instance.

40.3.1 How to Configure Change Persistence

By default, when you configure your application to use any type of change persistence (that is, to either the session or a repository), the values of all attributes shown in Table 40–1 will always be persisted to the user's session. If you configured your changes to be persisted to a repository, then you must declare the attributes whose values should be persisted to that repository. If there are any values that you don't want persisted, then you need to configure those values as well.

Before you begin:

It may be helpful to have an understanding of how runtime user customization are persisted. For more information, see Section 40.3, "Configuring User Customizations."

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Section 40.1.2, "Additional Functionality for Runtime User Customization."

You will need to launch JDeveloper using the **Studio Developer** role, and open (or create) the application in which you want to enable runtime user customization.

To declare attribute value persistence to a repository:

1. In the Application Navigator, expand the **Application Resources** pane, expand the **Descriptors** and **ADF META-INF** nodes, and double-click **adf-config.xml**.
2. In the overview editor, click the **View** navigation tab.
3. In the **Tags** table, select the component whose changes you want to persist (or not persist) to the repository.

   If the component does not appear in the table, click the **Add** icon to select and add it.

   Note: The filter rules specified in the `adf-config.xml` file are applicable only when you have chosen to persist to the MDS repository (see Section 40.2.1, "How to Enable User Customizations"). These rules do not apply for persistence within session scope.

   If persistence fails for any reason (for example if one of the filter rules fails or there are MDS repository errors), then the values will be stored only within the session scope.

4. The **Tag Attributes** table displays all the attributes for the selected component whose values can be persisted. Select **Persist Changes** for all attributes whose values you want persisted. Deselect any if you do not want the values persisted.

   Note: If you are implementing custom user customizations (see Section 40.5, "Implementing Custom User Customizations"), then you will need to edit the `adf-config.xml` manually to add the configuration. See Example 40–5 for an example on how to configure user customizations.

40.3.2 What Happens When You Configure Change Persistence

When you select the component tags and attribute values to be persisted in the `adf-config.xml` file, JDeveloper enters tag library information for the components and attributes that are to be persisted. Example 40–5 shows the entry for persisting the value of the `disclosed` attribute on the `panelBox` component.

Example 40–5 Registration of Attribute Changes in adf-config.xml

```
<taglib-config>
  <taglib uri="http://xmlns.oracle.com/adf/faces/rich">
    <tag name="panelBox">
      <attribute name="disclosed">
        <persist-changes>
          true
        </persist-changes>
      </attribute>
...
    </tag>
  </taglib>
</taglib-config>
```

40.4 Controlling User Customizations in Individual JSF Pages

Once you have enabled your application to allow user customizations, you can control user customizations for specific components on the page.

By default, the framework persists changes for all component instances, based on the configuration in the `adf-config.xml` file. You can override this default behavior by explicitly setting what should be persisted and what should not be persisted on each component instance using the `persist` and `dontPersist` attributes.

> **Note:** The filter rules specified using the `persist` and `dontPersist` attributes are applicable only when you have chosen to persist to the MDS repository (see Section 40.2.1, "How to Enable User Customizations"). These rules do not apply for persistence within session scope.
>
> If persistence fails for any reason (for example if one of the filter rules fails or there are MDS repository errors), then the values will be stored only within the session scope.

The following components support the `persist` and `dontPersist` attributes:

- `panelBox`
- `showDetail`
- `showDetailHeader`
- `showDetailItem`
- `column`
- `tree`
- `treeTable`
- `panelSplitter`
- `calendar`
- `dvt:projectGantt`
- `dvt:resourceUtilizationGantt`
- `dvt:schedulingGantt`
- `dvt:ganttLegend`
- `dvt:hierarchyViewer`
- `dvt:timeAxis`

40.4.1 How to Control User Customizations on a JSF Page

You can override any globally set persistence configuration for a component using its `persist` and `dontPersist` attributes.

> **Tip:** Often, a system administrator is the one to set the configurations in the `adf-config.xml`. The `persist` and `dontPersist` attributes allow page authors to override that setting as needed.

Before you begin:

It may be helpful to have an understanding of how runtime user customizations can be restricted on an individual page. For more information, see Section 40.4, "Controlling User Customizations in Individual JSF Pages."

You may also find it helpful to understand additional customization functionality that can be added to your applications. For more information, see Section 40.1.2, "Additional Functionality for Runtime User Customization."

You will need to launch JDeveloper using the **Studio Developer** role, open the application, and open the page for which you want to modify user customization behavior.

To implement user customizations on a JSF Page:

1. Add components to the page, as needed, including components that will be persisting changes.
2. If you want to persist all persistable attributes for a component:
   a. In the Property Inspector, expand the **Advanced** section.
   b. Click the drop-down list for the **Persist** field and choose **All Available**.
3. If you do not want to persist any attributes, repeat Step 2 for the **Don'tPersist** field.
4. If more than one attribute can be persisted for the component, and you do not want to persist all of them:
   a. Click the drop-down menu to the right of the **Persist** field and choose **Edit** to open the Edit Property dialog.
   b. Shuttle any attributes to be persisted from **Available** to **Selected**.
5. If you do not want to persist an attribute value, repeat Step 4 for the **Don't Persist** field.

Note: The filter rules specified using the `persist` and `dontPersist` attributes take precedence over any `adf-config.xml` configuration set for global component-level restrictions.

Values specified for the `dontPersist` attribute take precedence over values specified for the `persist` attribute. For example, if for a `panelBox` component you set `disclosed` as the value for both the `persist` and `dontPersist` attributes, the value of the `disclosed` attribute will not be persisted.

If you set the value of the `persist` or `dontPersist` attribute to `All Available`, then any values entered as choices using the Edit dialog and the shuttle will be ignored and all available attribute values will be persisted or not persisted.

40.4.2 What Happens at Runtime

When an application is configured to persist changes to the session, any changes made during the session are recorded in a session variable in a data structure that is indexed according to the view ID and the component's ID attribute value. Every time the page is requested, in the subsequent create View or Restore View phase, all changes are applied in the same order as they were added. This means that the changes registered

through the session will be applied only during subsequent requests in the same session.

When an application is configured to persist changes to the MDS repository, any changes made during the session are recorded by mutating the Document Object Model that MDS maintains for the JSP document behind the view. A JSF phase listener registered by ADF controller triggers a commit on the MDS session during the appropriate lifecycle phase, resulting in the change document being persisted in the MDS store. Every time the page is requested, Oracle's JSP engine seeks the JSP document from an MDS JSP provider, which provides a flattened document after merging the stored changes to the base document. MDS records the change against the unique value of the component's ID attribute.

> **Tip:** If changes are applied in response to a partial submit of the page (for example, a `commandButton` with the `partialSubmit` attribute set to `true`), the component for which changes are applied must be set as the value for the `partialTarget` attribute.

Additionally, be aware that when you run the application from JDeveloper in the Integrated WebLogic Server, MDS creates a local file-based repository to persist metadata customizations. In contrast, when the application is deployed to a test or production environment, customizations are persisted to the configured MDS repository. For more information about MDS repository configuration, see the *Oracle Fusion Middleware Administrator's Guide*. For more information about deploying an application, see Section 41.4, "Deploying the Application."

40.4.3 What You May Need to Know About Using Change Persistence on Templates and Regions

How changes are persisted for components on templates or regions is handled differently, depending on whether the changes are persisted to the session or to the MDS repository. With session persistence, changes are recorded and restored on components against the `viewId` for the given session. As a result, when the change is applied on a component that belongs to a region or page template, that change is applicable only in scope of the page that uses the region or template. It does not span all pages that consume the region or template.

For example, suppose you have `pageOne.jspx` and `pageTwo.jspx`, and they both contain the region defined in `region.jsff`, which in turn contains a `showDetail` component. When `pageOne.jspx` is rendered and the `disclosed` attribute on the `showDetail` component changes, the implicit attribute change is recorded and will be applied only for `pageOne.jspx`. If the user navigates to `pageTwo.jspx`, no attribute change is applied.

When you persist changes to the MDS repository, MDS records and restores customizations for a document identified by a combination of the JSP page path and customization name/value configuration setting as set on the customization class (for more information, see Section 39.2.1.1, "Customization Classes"). As a result, for a given page that is rendered, when MDS applies a change on a component within a region or template, it is applicable for all pages that consume the region or template and that have the same customization name and value as the source page.

In the previous example, assume that the `showDetail` component uses the ID of `myShowDetail`. When `pageOne.jspx` is rendered and the `disclosed` attribute on the `showDetail` component changes, the attribute change is recorded for `region.jsff` (and not the page that consumes it). This change is applied when any page that contains the region is rendered, as long as the ID remains the same.

40.5 Implementing Custom User Customizations

In addition to the user customization capabilities built in to certain ADF Faces components, you can create your own custom user customization capabilities. The change persistence framework supports the following types of user customizations:

- Changing an attribute value
- Adding or removing a facet
- Adding or removing a child component
- Reordering child components
- Moving a child component to a different parent

To create custom user customizations, you must create a customization class for each type of user customization and then configure your application to use that class. You also need to set up the layers of customization for your application. For more information about both of these procedures, see Section 39.2, "Developing a Customizable Application."

Once those prerequisites are satisfied, you add logic that calls methods on the ADF Faces classes that handle persisting change either to the session or the MDS repository. To handle the change, you create code that uses the APIs from one of the ADF Faces specialized component change classes. For most cases, you add this code to the event handler method on a managed bean associated with the page the persisting component is on. If you want all instances of a component to persist the same change, you need to add this code for each page on which that component appears.

If you are creating a custom component, you can implement user customizations for the component by adding code directly to the custom component class. In that case, you will need to add the code only to the component class, and not once for each instance of the component. For more information, see Section 40.6, "Creating Implicit Change Persistence in Custom Components."

40.5.1 Change Persistence Framework API

To better understand what you need to do to create custom user customizations, it may help to have a deeper understanding of the change persistence and MDS frameworks. When you elect to persist changes to the MDS repository, the change persistence framework works in conjunction with the MDS framework. Where and how the customizations are saved are determined by how you set up your MDS repository, your customization layers, and your customization classes. Details about the MDS framework and the repository and how to use it are covered in Chapter 39, "Customizing Applications with MDS."

The change persistence framework uses the underlying change manager classes from Apache MyFaces Trinidad (in the `org.apache.myfaces.trinidad.change` package) along with a few ADF Faces-specific classes (in the `oracle.adf.view.rich.change` package). The instance of the registered `ChangeManager` class is accessible through the `RequestContext` object. It is responsible for gathering changes as they are created and added during a request, and then persisting them. The `SessionChangeManager` class is an implementation of `ChangeManager` which handles persistence within a session only, while the `MDSDocumentChangeManager` class is an implementation that persists to the MDS repository only. The `FilteredPersistenceChangeManager` class is an implementation of `ChangeManager` that stores the changes that pass the filter rules into the repository using the registered persistence change manager. Any change that

does not get persisted to the repository will be persisted to the session when `FilteredPersistenceChangeManager` is used.

Additional classes are used to describe the changes to a component. You use these APIs to handle persisting any changes to components other than the implicit value changes the ADF Faces framework provides (as shown in Table 40–1). `ComponentChange` is the base class for all classes used to implement specific changes that act on the JSF component hierarchy, such as adding or removing a facet or a child component. These changes are automatically applied during subsequent creation of the view, in the same order in which they were added. Classes that extend the `ComponentChange` class and that also implement the `DocumentChange` interface can directly persist changes to the MDS repository. Classes that do not implement the `DocumentChange` interface can persist changes only to the session.

Table 40–2 describes the specialized classes that handle specific customizations. If "yes" appears in the Repository column, then the class implements the `DocumentChange` interface and it can persist changes to the MDS repository.

Table 40–2 Classes Used to Handle Change Persistence

Class Name	Repository	Description
`AddChildDocumentChange`	Yes	Adds a child or complex child (`dvt:`) component using document mark up. While applying this change, the child or complex child component is created and added to the document.
`AddComplexChildDocumentChange`	Yes	
`AddComplexChildAttributeChange`	Yes	Adds a complex child (`dvt:`) component attribute.
`AttributeComponentChange`	No	Changes the value of an attribute.
`ComplexAttributeComponent Change`	Yes	
`AttributeDocumentChange`	Yes	Changes the value of a facet attribute.
`ComplexAttributeDocumentChange`	Yes	
`MoveChildComponentChange`	Yes	Moves a child from one container to another.
`RemoveChildComponentChange`	Yes	Removes a child component.
`RemoveComplexChildAttribute ComponentChange`	Yes	Removes a complex child (`dvt:`) component attribute.
`RemoveComplexChildAttribute DocumentChange`	Yes	
`SetFacetChildComponentChange`	No	Adds a child component to the facet using a document markup. While applying this change, the markup will be added to the document.
`SetFacetChildDocumentChange`	Yes	Adds a child component to a facet. While applying this change, the DOM element corresponding to the child component is added to the document. If the facet doesn't exist, it will be created. If the facet does exist, all of its content will be removed and the new content added.
`RemoveFacetComponentChange`	Yes	Removes a facet.
`ReorderChildrenComponentChange`	Yes	Reorders children of a component.

Aside from a ChangeManager class, you may also need to implement and register the `DocumentChangeFactory` interface with the `ChangeManager` class. If the `DocumentChangeFactory` implementation can provide an equivalent `DocumentChange` for a `ComponentChange`, the `ChangeManager` will use it to persist the `DocumentChange` to the repository.

40.5.2 How to Create Code for Custom User Customizations

You need to add code to handle any explicit changes you want to create, and to configure the components on the JSF page to handle customization. As with the default user customizations, you also must register the custom changes in the `adf-config.xml` file.

> **Note:** When the changes are expressible in more than one form, the change must be recorded in the form with highest precedence. For example:
>
> - Attribute change for a component: The attribute can be specified on the component tag or it can be expressed using the `<f:attribute>` tag. In a JSF JSP document, `<f:attribute>` takes lesser precedence over the attribute specified on the component tag. Therefore, the attribute change on the component tag will be recorded for customization.
> - Column header text in a column component: The header text for the column can be specified using either the `headerText` attribute or using header facet. In this case, the facet component will have precedence.

To create custom user customizations:

1. Create a managed bean for the page that contains the component.
2. Add code to the event handler method for the component that will be used to make the change. This code should obtain the component that contains the change. It should then use the component and the appropriate APIs to create, record, and persist the change.

   Example 40–6 shows the code on the action event handler for a command button for a change that will be persisted to the MDS repository. When a user clicks the button, that source graphic file changes. The event handler method accesses the component and changes the source attribute for the graphic. It then calls the private `addAttributeChange` method, which first uses the component API to record the change, and then uses the `AttributeComponentChange` class to set the new source attribute value.

Example 40–6 Persisting Change to the Repository from an Event Handler on a Managed Bean

```
public void modifyObjectImage(ActionEvent event) {
       UIComponent uic = event.getComponent().findComponent("oi1");
       String source = "/images/mediumAd.gif";
       uic.getAttributes().put("source", source);
       _addAttributeChange(uic, "source", source);
    }
.
.
.
```

```
private static void _addAttributeChange(UIComponent uic, String attribName,
                                        Object attribValue) {
    FacesContext fc = FacesContext.getCurrentInstance();
    ChangeManager cm =
        RequestContext.getCurrentInstance().getChangeManager();
    ComponentChange cc =
        new AttributeComponentChange(attribName, attribValue);
    cm.addComponentChange(fc, uic, cc);
}
```

Note: When you persist changes, in addition to explicitly recording a change on the component (which is done in Example 40–6 using `uic.getAttributes().put("source", source)` method), you must also directly apply the change using the component API, as was done using the private `_addAttributeChange(uic, "source", source)` method. Applying the change in this way allows the user to see the change in response to the same request. If the change is recorded on the component, then the change will not be seen until a subsequent request.

Additionally, if you know that the component will always persist to the repository regardless of any restricted change persistence settings, you can instead call the `AdfFacesContext.getCurrentInstance().getPersistentChangeManager()` method.

---

3. The `ChangeManager` class provides support for automatically converting an `AttributeComponentChange` into an `AttributeDocumentChange`, thereby allowing persistence to a repository. However, if you need to convert another type of change and you use a specialized change manager class that does not implement the `DocumentChange` class, you need to create a custom `DocumentFactory` implementation that converts the component change to a document change.

Note: Automatic conversion of `AttributeComponentChange` into an `AttributeDocumentChange` assumes that the component attribute is represented as an attribute of the same name on the associated element in the JSPX document.

Only those attribute values that are expressible in the JSPX document can be persisted using `AttributeDocumentChange`. In other words, `CharSequence`, `Number`, `Boolean` and `ValueExpression` are the only supported data types.

Only values that implement `java.io.Serializable` can be persisted using `AttributeComponentChange`.

---

4. If you create a custom `DocumentFactory` implementation, you need to register it with the appropriate change manager class using the following method in your bean:

```
public static void registerDocumentFactory(String targetClassName,
                                           String converterClassName)
```

Where `targetClassName` is the name of the `ComponentChange` class and `converterClassName` is the name of your `DocumentChangeFactory` extension that is capable of converting the target `ComponentChange` into a `DocumentChange`. The semantics of `name` for these classes is same as that of `getName()` in the `java.lang.Class` class.

5. If the class you use to create the component change adds a child that has a subtree of components, and you want to persist the changes to the repository, you must create a `DocumentFragment` to represent the change.

   Example 40–7 shows how to use the `AddComponentDocumentChange` specialized class to create a `DocumentChange` object and use a `DocumentFragment` to represent the change.

Example 40–7 Converting a ComponentChange Object to a DocumentChange Object

```
public void appendChildToDocument(ActionEvent event)
{
  UIComponent eventSource = event.getComponent();
  UIComponent uic = eventSource.findComponent("pg1");
  // only allow the image to be added once
  if (_findChildById(uic,"oi3") != null)
    return;
  FacesContext fc = FacesContext.getCurrentInstance();
  DocumentFragment imageFragment = _createDocumentFragment(_IMAGE_MARK_UP);
  DocumentChange change = new AddChildDocumentChange(imageFragment);
  ChangeManager apm = RequestContext.getCurrentInstance().getChangeManager();
  apm.addDocumentChange(fc, uic, change);
}
 private static final String _IMAGE_MARK_UP =
 "<af:objectImage id='oi3' height='100' width='120' " +
     "source='http://www.somewhere.com/someimage.jpg' " +
     "xmlns:af='http://xmlns.oracle.com/adf/faces'/>";

private static DocumentFragment _createDocumentFragment(
    String markUp)
{
 // prepend XML declaration
  markUp = "<?xml version = '1.0' encoding = 'ISO-8859-1'?>" + markUp;
  DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
  factory.setNamespaceAware(true);
  factory.setValidating(false);
  DocumentBuilder builder;
  try
  {
    builder = factory.newDocumentBuilder();
  }
  catch (ParserConfigurationException pce)
  {
    _LOG.log(Level.WARNING, "Unable to get XML Parser:", pce);
     return null;
  }
  try
  {
    // use a version explicitly with ISO-8859-1 instead
    byte[] markupBytes = markUp.getBytes();
    Document newDoc = builder.parse(new ByteArrayInputStream(markupBytes));
    DocumentFragment fragment = newDoc.createDocumentFragment();
    // add the document's root element to the fragment
    fragment.appendChild(newDoc.getDocumentElement());
```

```
    return fragment;
  }
  catch (SAXException se)
  {
    _LOG.log(Level.WARNING, "Unable to parse markup:" + markUp, se);
    return null;
  }
  catch (IOException ioe)
  {
    _LOG.log(Level.WARNING, "IO Problem with markup:" + markUp, ioe);
    return null;
  }
}
```

6. Register the user customizations in the `adf-config.xml` file, as documented in Section 40.3, "Configuring User Customizations." If the custom changes are of any type other than `AttributeDocumentChange`, you will need to manually edit the `adf-config.xml` file and indicate that all changes are allowed for the component, as shown in Example 40–8.

Example 40–8 Custom Registration in adf-config.xml

```
<tag name="inputText">
  <attribute name="label">
    <persist-changes>true</persist-changes>
  </attribute>
  <persist-operations>ALL</persist-operations>
</tag>
```

40.6 Creating Implicit Change Persistence in Custom Components

When you create a custom component, you may decide that you want certain attribute values on that component to be persisted whenever change persistence is enabled in an application. Setting implicit change on a custom component is similar to setting explicit change persistence on existing components. You add code that executes the actual persistence, but instead of you placing that code on a managed bean, that code can be handled directly by the component class. If your component's attribute values are synchronized with the server using events, then you can use the broadcast method to persist the changes. If the attribute value that you want to persist does not use events, then you need to add code in the renderer and component class.

40.6.1 How to Set Implicit Change Persistence For Attribute Values that Use Events

When an attribute value uses events, you need to add code to the component class.

To set implicit change persistence for attribute values that use events:

1. Open the custom component class java file.
2. Add code to the broadcast method that will use the specialized class to create a new `ComponentChange` object and then call the `ChangeManager` to add the change.

   Example 40–9 shows the code added to the `UIXShowDetail` class that persists a change to the `disclosed` attribute. In this case, the `AttributeComponentChange` class is used.

Example 40–9 Persisting Change from a Component Class

```
public class UIXShowDetail extends UIXComponentBase
{
  ...
  public void broadcast(FacesEvent event) throws AbortProcessingException
  {
    super.broadcast(event);
    ...
    if (event instanceof DisclosureEvent)
    {
      boolean isDisclosed = ((DisclosureEvent) event).isExpanded();
      setDisclosed(isDisclosed);
      //Record a Change for 'disclosed' attribute
      AttributeComponentChange aa =
       new AttributeComponentChange('disclosed', isDisclosed ? Boolean.TRUE :
Boolean.FALSE);
      AdfFacesContext adfContext = AdfFacesContext.getCurrentInstance();
      adfContext.getChangeManager().addComponentChange(getFacesContext(), this,
aa);
      ...
    }
  }
  ...
```

40.6.2 How to Set Implicit Change Persistence For Other Attribute Values

When an attribute does not use events, you need to place code in the component's renderer class.

To set implicit change persistence for other attribute values:

1. Open the custom component's render class java file.
2. Use the `findTypeConstants` method, which takes a `ClientMetadata` instance and use the `addPersistedProperty` method to mark certain properties as persisted. Example 40–10 shows a code snippet from the renderer class used for the ADF Faces `PanelSplitter` component, which implicitly persists the `splitterPosition` attribute value.

Example 40–10 Method in Component Renderer Class to Implicitly Persist Changes

```
// Code snippet from PanelSplitterRenderer.java
protected void findTypeConstants(
    FacesBean.Type type,
    ClientMetadata metadata)
  {
    super.findTypeConstants(type, metadata);
    metadata.addRequiredProperty(
      _orientationKey = type.findKey("orientation"));
    metadata.addRequiredProperty(
      _positionedFromEndKey = type.findKey("positionedFromEnd"));
    metadata.addRequiredProperty(
      _disabledKey = type.findKey("disabled"));
    metadata.addRequiredProperty(
      _splitterPositionKey = type.findKey("splitterPosition"));
    metadata.addPersistedProperty(_splitterPositionKey);
  }
```

3. In the JavaScript component peer class, define the attribute value to be persisted, using the `setProperty` function. This function needs to be invoked with the attribute name (as defined in the renderer in the previous step), the value, and "`true`", meaning the value of the attribute will be set. Example 40–11 shows a code snippet from the `panelSplitter` class that sets the splitter position.

Example 40–11 Method in Component Class to Implicitly Persist Changes

```
// Code snippet from AdfDhtmlPanelSplitterPeer.js file where we set the
   splitter position

  var component = this.getComponent();
 component.setProperty("splitterPosition",position, true);
// position is the value to be set
```

41

Deploying Fusion Web Applications

This chapter describes how to deploy Oracle ADF applications to a target application server. It describes how to create deployment profiles, how to create deployment descriptors, and how to load ADF runtime libraries.

This chapter includes the following sections:

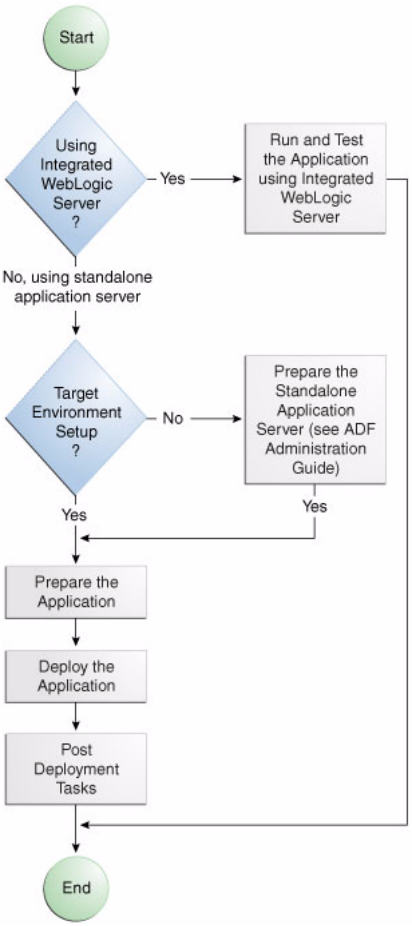
- Section 41.1, "About Deploying Fusion Web Applications"
- Section 41.2, "Running an ADF Application in Integrated WebLogic Server"
- Section 41.3, "Preparing the Application"
- Section 41.4, "Deploying the Application"
- Section 41.5, "Postdeployment Configuration"
- Section 41.6, "Testing the Application and Verifying Deployment"

41.1 About Deploying Fusion Web Applications

Deployment is the process of packaging application files as an archive file and transferring that file to a target application server. You can use JDeveloper to deploy Oracle ADF applications directly to the application server (such as Oracle WebLogic Server or IBM WebSphere), or indirectly to an archive file as the deployment target, and then install this archive file to the target server. For application development, you can also use JDeveloper to run an application in Integrated WebLogic Server. JDeveloper supports deploying to server clusters, but you cannot use JDeveloper to deploy to individual Managed Servers that are members of a cluster.

Figure 41–1 shows the flow diagram that describes the overall deployment process. Note that preparing the target application server for deployment by installing the ADF runtime is described in the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*

Figure 41–1 Deployment Overview Flow Diagram



Note: Normally, you use JDeveloper to deploy applications for development and testing purposes. If you are deploying Oracle ADF applications for production purposes, you can use Enterprise Manager or scripts to deploy to production-level application servers.

For more information about deployment to later-stage testing or production environments, see the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

ADF Java EE applications are based on standardized, modular components and can be deployed to the following application servers:

- Oracle WebLogic Server

  Oracle WebLogic Server provides a complete set of services for those modules and handles many details of application behavior automatically, without requiring programming. For information about which versions of Oracle WebLogic Server are compatible with JDeveloper, see the certification information website at `http://www.oracle.com/technetwork/developer-tools/jdev/documentation/index.html`.

- IBM WebSphere Application Server

For information about which versions of IBM WebSphere are compatible, see the *Oracle Fusion Middleware Third-Party Application Server Guide*.

Deploying a Fusion web application is slightly different from deploying a standard Java EE application. JSF applications that contain ADF Faces components have a few additional deployment requirements:

- ADF Faces requires Sun's JSF Reference Implementation 1.2 and MyFaces 1.0.8 (or later).

You can use JDeveloper to:

- Run applications in Integrated WebLogic Server

  You can run and debug applications using Integrated WebLogic Server and then deploy to standalone WebLogic Server or WebSphere Application Server.

- Deploy directly to the standalone application server

  You can deploy applications directly to the standalone application server by creating a connection to the server and choosing the name of that server as the deployment target.

- Deploy to an archive file

  You can deploy applications indirectly by choosing an EAR file as the deployment target. The archive file can subsequently be installed on a target application server.

The StoreFront module of the Fusion Order Demo application demonstrates the use of the Fusion web application technology stack to create transaction-based web applications. You can run the StoreFront module of the Fusion Order Demo application in JDeveloper using Integrated WebLogic Server. You cannot run the Fusion Order Demo module in JDeveloper. You must deploy the Fusion Order Demo application to a SOA-enabled Oracle WebLogic Server. For more information about the StoreFront module and the Fusion Order Demo application, see Section 2.3, "Running the Fusion Order Demo Application StoreFront Module."

41.1.1 Developing Applications with Integrated WebLogic Server

If you are developing an application in JDeveloper and you want to run the application in Integrated WebLogic Server, you do not need to perform the tasks required for deploying directly to Oracle WebLogic Server or to an archive file. JDeveloper has a default connection to Integrated WebLogic Server and does not require any deployment profiles or descriptors. Integrated WebLogic Server has a preconfigured domain that includes the ADF libraries, as well as the `-Djps.app.credential.overwrite.allowed=true` setting, that are required to run Oracle ADF applications. You can run an application by choosing **Run** from the JDeveloper main menu.

You debug the application using the features described in Chapter 36, "Testing and Debugging ADF Components."

41.1.2 Developing Applications to Deploy to Standalone Application Server

Typically, for deployment to standalone application servers, you test and develop your application by running it in Integrated WebLogic Server. You can then test the application further by deploying it to standalone Oracle WebLogic Server in development mode or to IBM WebSphere Application Server to more closely simulate the production environment.

In general, you use JDeveloper to prepare the application or project for deployment by:

- Creating a connection to the target application server
- Creating deployment profiles (if necessary)
- Creating deployment descriptors (if necessary, and that are specific to the target application server)
- Updating `application.xml` and `web.xml` to be compatible with the application server (if required)
- Enabling the application for Real User Experience Insight (RUEI) in `web.xml` (if desired)
- Migrating application-level security policy data to a domain-level security policy store
- Configuring the Oracle Single Sign-On (Oracle SSO) service and properties in the domain `jps-config.xml` file when you intend the web application to run using Oracle SSO

You must already have an installed application server. For Oracle WebLogic Server, you can use the Oracle 11*g* Installer or the Oracle Fusion Middleware 11*g* Application Developer Installer to install one. For other application servers, follow the instructions in the applications server documentation to obtain and install the server.

You must also prepare the application server for ADF application deployment. For more information, see the "Preparing the Standalone Application Server for Deployment" section of the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

- Installing the ADF runtime into the application server installation:
  - For WebLogic Server
    - If you installed Oracle WebLogic Server together with JDeveloper using the Oracle 11*g* Installer for JDeveloper, the ADF runtime should already be installed.
    - If the ADF runtime is not installed and you want to use Oracle Enterprise Manager to manage standalone ADF applications (which are applications without Oracle SOA Suite or Oracle WebCenter components), use the Oracle Fusion Middleware 11*g* Application Developer Installer. This installer will install the necessary Oracle Enterprise Manager components into the Oracle WebLogic installation.
    - If the ADF runtime is not installed and you do not need to install Enterprise Manager, use the Oracle 11*g* Installer for JDeveloper.
  - For WebSphere Application Server
    * Use the Oracle Fusion Middleware 11*g* Application Developer Installer to install the ADF runtime and the necessary Oracle Enterprise Manager components into the WebSphere installation. For information about installing WebSphere, see the *Oracle Fusion Middleware Third-Party Application Server Guide*.
- Extending Oracle WebLogic Server domains or WebSphere Application Server Cells to be ADF-compatible using the ADF runtime

- For WebLogic, setting the Oracle WebLogic Server credential store overwrite setting as required (`-Djps.app.credential.overwrite.allowed=true` setting).
- Creating a global JDBC data source for applications that require a connection to a data source

After the application and application server have been prepared, you can:

- Use JDeveloper to:
  - Directly deploy to the application server using the deployment profile and the application server connection.
  - Deploy to an EAR file using the deployment profile. For Oracle ADF applications, WAR and MAR files can be deployed only as part of an EAR file.
- Use Enterprise Manager, scripts, or the application server's administration tools to deploy the EAR file created in JDeveloper. For more information, see the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

41.2 Running an ADF Application in Integrated WebLogic Server

JDeveloper is installed with Integrated WebLogic Server which you can use to test and develop your application. For most development purposes, Integrated WebLogic Server will suffice. When your application is ready to be tested, you can select the run target and then choose the **Run** command from the main menu.

> **Note:** The first time you run an application in Integrated WebLogic Server, the Configure Default Domain dialog appears for you to define an administrative password for the new domain.

When you run the application target, JDeveloper detects the type of Java EE module to deploy based on artifacts in the projects and workspace. JDeveloper then creates an in-memory deployment profile for deploying the application to Integrated WebLogic Server. JDeveloper copies project and application workspace files to an "exploded EAR" directory structure. This file structure closely resembles the EAR file structure that you would have if you were to deploy the application to an EAR file. JDeveloper then follows the standard deployment procedures to register and deploy the "exploded EAR" files into Integrated WebLogic Server. The "exploded EAR" strategy reduces the performance overhead of packaging and unpackaging an actual EAR file.

In summary, when you select the run target and run the application in Integrated WebLogic Server, JDeveloper:

- Detects the type of Java EE module to deploy based on the artifacts in the project and application
- Creates a deployment profile in memory
- Copies project and application files into a working directory with a file structure that would simulate the "exploded EAR" file of the application.
- Performs the deployment tasks to register and deploy the simulated EAR into Integrated WebLogic Server
- Automatically migrates identities, credentials, and policies

Later on, if you plan to deploy the application to a standalone WebLogic Server instance, you will need to migrate this security information. For more information, see Section 41.3.4, "How to Deploy Applications with ADF Security Enabled."

Note: JDeveloper ignores the deployment profiles that were created for the application when you run the application in Integrated WebLogic Server.

The application will run in the base domain in Integrated WebLogic Server. This base domain has the same configuration as a base domain in a standalone WebLogic Server instance. In other words, this base domain will be the same as if you had used the Oracle Fusion Middleware Configuration Wizard to create a base domain with the default options in a standalone WebLogic Server instance.

JDeveloper will extend this base domain with the necessary domain extension templates, based on the JDeveloper technology extensions. For example, if you have installed JDeveloper Studio, JDeveloper will automatically configure the Integrated WebLogic Server environment with the ADF runtime template (JRF Fusion Middleware runtime domain extension template).

You can explicitly create a default domain for Integrated WebLogic Server. You can use the default domains to run and test your applications. Open the Application Server Navigator, right-click **IntegratedWebLogicServer** and choose **Create Default Domain**.

41.2.1 How to Run an Application in Integrated WebLogic Server

You can test an application by running it in Integrated WebLogic Server. You can also set breakpoints and then run the application within the ADF Declarative Debugger.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 41.2, "Running an ADF Application in Integrated WebLogic Server."

To run an application in Integrated WebLogic Server:

1. In the Application Navigator, select the project, unbounded task flow, JSF page, or file as the run target.
2. Right-click the run target and choose **Run** or **Debug**.

   The Configure Default Domain dialog displays the first time you run your application and start a new domain in Integrated WebLogic Server. Use the dialog to define an administrator password for the new domain. Passwords you enter can be eight characters or more and must have a numeric character.

41.2.2 How to Run an Application with Metadata in Integrated WebLogic Server

When an application is running in Integrated WebLogic Server, the MAR (Metadata Archive) profile itself will not be deployed to a repository, but a simulated MDS repository will be configured for the application that reflects the metadata information contained in the MAR. This metadata information is simulated, and the application runs based on this location in source control.

Any customizations or documents created by the application that are not configured to be stored in other MDS repositories are written to this simulated MDS repository

directory. For example, if you customize an object, the customization is written to the simulated MDS repository. If you execute code that creates a new metadata object, then this new metadata object is also written to the same location in the simulated MDS repository. You can keep the default location for this directory (`ORACLE_HOME\jdeveloper\systemXX.XX\o.mds.dt\adrs\Application\AutoGeneratedMar\mds_adrs_writedir`), or you can set it to a different directory. You also have the option to preserve this directory across different application runs, or to delete this directory before each application run.

If your workspace has different working sets, only the metadata from the projects defined in the working set and their dependent projects will be included in the MAR. You can view and change a project's dependencies by right-clicking the project in the Application Navigator, choosing **Project Properties**, and then selecting **Dependencies**. For instance, an application may have several projects but `workingsetA` is defined to be `viewcontroller2` and `viewcontroller5`; and `viewcontroller5` has a dependency on `modelproject1`. When you run or debug `workingsetA`, only the metadata for `viewcontroller2`, `viewcontroller5`, and `modelproject1` will be included in the MAR for deployment.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 41.2, "Running an ADF Application in Integrated WebLogic Server."

You will need to complete this task:

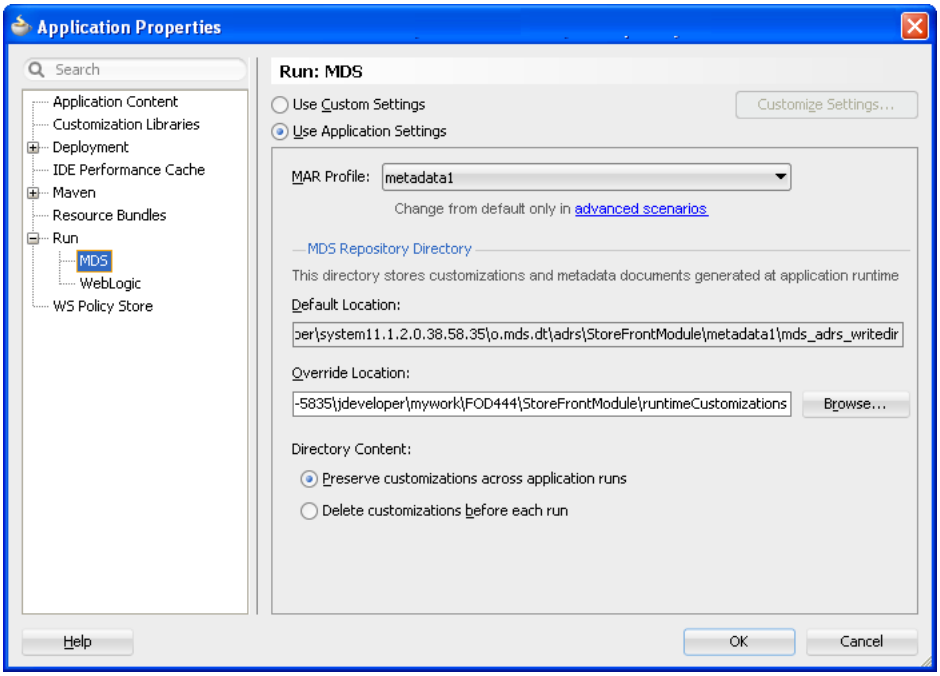
Created a MAR profile, either manually or automatically by JDeveloper.

To deploy the MAR profile to Integrated WebLogic Server:

1. In the Application Navigator, right-click the application and choose **Application Properties**.
2. In the Application Properties dialog, expand **Run** and choose **MDS**.
3. On the Run MDS page:
   - Select the MAR profile from the **MAR Profile** dropdown list
   - Enter a directory path in **Override Location** if you want to customize the location of the simulated MDS repository.
   - Select the **Directory Content** option. You can chose to preserve the customizations across application runs or delete customizations before each run.

   Select the MAR profile from the **MAR Profile** dropdown list. Figure 41–2 shows **Demometadata1** selected as the MAR profile.

Figure 41–2 Setting the Run MDS Options

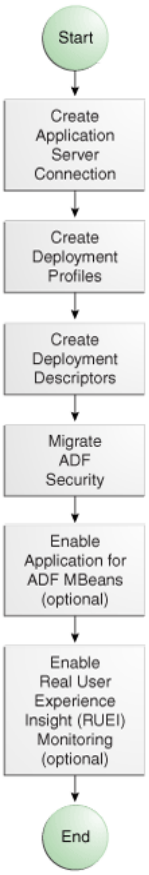


41.3 Preparing the Application

Before you deploy an ADF application to a standalone application server, you must perform prerequisite tasks within JDeveloper to prepare the application for deployment.

Figure 41–3 show the process flow to prepare the application for deployment. After the application has been prepared and the application server has been prepared as described in the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*, you can proceed to deploy the application as described in Section 41.4, "Deploying the Application."

Figure 41–3 Preparing the Application for Deployment Flow Diagram



41.3.1 How to Create a Connection to the Target Application Server

You can deploy applications to the application server via JDeveloper application server connections.

If your application involves customization using MDS, you should register your MDS repository with the application server:

- WebLogic: register the MDS into the WebLogic Domain

  For more information about registering MDS in WebLogic, see the *Oracle Fusion Middleware Administrator's Guide*.

- WebSphere: register the MDS into the WebSphere Cell

  For more information about registering MDS in WebSphere, see the *Oracle Fusion Middleware Third-Party Application Server Guide*.

For more information about registering MDS, see the *Oracle Fusion Middleware Administrator's Guide*.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 41.3, "Preparing the Application."

You will need to complete this task:

Installed an application server.

Before you begin:

Install the application server. For more information, see Section 41.3, "Preparing the Application."

To create a connection to an application server:

1. Launch the Application Server Connection wizard.

   You can:

   - In the Application Server Navigator, right-click **Application Servers** and choose **New Application Server Connection**.
   - In the New Gallery, expand **General**, select **Connections** and then **Application Server Connection**, and click **OK**.
   - In the Resource Palette, choose **New** > **New Connections > Application Server**.

2. In the Create AppServer Connection dialog, on the Usage page, select **Standalone Server**.
3. On the Name and Type page, enter a connection name.
4. In the **Connection Type** dropdown list, choose:
   - **WebLogic 10.3** to create a connection to Oracle WebLogic Server
   - **WebSphere Server 7.x** to create a connection to IBM WebSphere Server
5. Click **Next**.
6. On the Authentication page, enter a user name and password for the administrative user authorized to access the application server.
7. Click **Next**.
8. On the Configuration page, enter the information for your server:

   For WebLogic:

   - The Oracle WebLogic host name is the name of the WebLogic Server instance containing the TCP/IP DNS where your application (`.jar`, `.war`, `.ear`) will be deployed.
   - In the **Port** field, enter a port number for the WebLogic Server instance on which your application (`.jar`, `.war`, `.ear`) will be deployed.

     If you don't specify a port, the port number defaults to 7001.
   - In the **SSL Port** field, enter an SSL port number for the WebLogic Server instance on which your application (`.jar`, `.war`, `.ear`) will be deployed.

     Specifying an SSL port is optional. It is required only if you want to ensure a secure connection for deployment.

     If you don't specify an SSL port, the port number defaults to 7002.
   - Select **Always Use SSL** to connect to the WebLogic Server instance using the SSL port.
   - Optionally enter a **WebLogic Domain** only if WebLogic Server is configured to distinguish nonadministrative server nodes by name.

   For WebSphere:

- In the **Host Name** field, enter the name of the WebSphere server containing the TCP/IP DNS where your Java EE applications (`.jar`, `.war`, `.ear`) are deployed. If no name is entered, the name defaults to localhost.
- In the **SOAP Connector Port** field, enter the port number. The host name and port are used to connect to the server for deployment. The default SOAP connector port is 8879.
- In the **Server Name** field, enter the name assigned to the target application server for this connection.
- In the **Target Node** field, enter the name of the target node for this connection. A node is a grouping of Managed Servers. The default is `machineNode01`, where `machine` is the name of the machine the node resides on.
- In the **Target Cell** field, enter the name of the target cell for this connection. A cell is a group of processes that host runtime components. The default is `machineNode01Cell`, where *`machine`* is the name of the machine the node resides on.
- In the **Wsadmin script location** field, enter, or browse to, the location of the `wsadmin` script file to be used to define the system login configuration for your IBM WebSphere application server connection. Note that you should not use the `wsadmin` files from the `ORACLE_HOME/oracle_common/common/bin` directory, which are not the correct version. The default location is `websphere-home/bin/wsadmin.sh` for Unix/Linux and `websphere-home/bin/wsadmin.bat` for Windows.

9. Click **Next**.
10. If you have chosen WebSphere, the JMX page appears. On the JMX page, enter the JMX information (optional):

    Note: JMX configuration is optional and is not required for connecting to the WebSphere Application Server. JMX is only needed for deploying SOA applications.

    - Select **Enable JMX** for this connection to enable JMX.
    - In the **RMI Port** field, enter the port number of the WebSphere RMI connector port. The default is 2809.
    - In the **WebSphere Runtime Jars Location** field, enter or browse to the location of the WebSphere runtime JARs.
    - In the **WebSphere Properties Location (for secure MBEAN access)** field, enter or browse to the location of the file that contains the properties for the security configuration and the mbeans that are enabled. This field is optional.
11. Click **Next**.
12. If the SSl Signer Exchange Prompt dialog appears, click **Y**.
13. On the Test page, click **Test Connection** to test the connection.

    JDeveloper performs several types of connections tests. The JSR-88 test must pass for the application to be deployable. If the test fails, return to the previous pages of the wizard to fix the configuration.
14. Click **Finish**.

41.3.2 How to Create Deployment Profiles

A *deployment profile* defines the way the application is packaged into the archive that will be deployed to the target environment. The deployment profile:

- Specifies the format and contents of the archive file that will be created
- Lists the source files, deployment descriptors, and other auxiliary files that will be packaged
- Describes the type and name of the archive file to be created
- Highlights dependency information, platform-specific instructions, and other information

You need a WAR deployment profile for each web user interface project that you want to deploy in your application. If you want to package seeded customizations or place base metadata in the MDS repository, you need an application-level metadata archive (MAR) deployment profile as well. For more information about seeded customizations, see Chapter 39, "Customizing Applications with MDS." If the application has customization classes, you need a JAR file for those classes and you need to add that JAR when you create the EAR file. Finally, you need an application-level EAR deployment profile and you must select the projects (such as WAR and MAR profiles and customization classes JAR files) to include from a list. When the application is deployed, the EAR file will include all the projects that were selected in the deployment profile.

Note: If you create your project or application using the Fusion Web Application (ADF) template, JDeveloper automatically creates default WAR, EAR, MAR, and JAR deployment profiles. Typically, you would not need to edit or create deployment profiles manually.

For Oracle ADF applications, you can deploy the application only as an EAR file. The WAR and MAR files that are part of the application should be included in the EAR file when you create the deployment profile.

Note: If your ADF application has business services that you want to deploy, you will need to create a Business Component Service Interface deployment profile and deploy it. For more information about business services, see Section 11.2.20, "How to Deploy Web Services to Oracle WebLogic Server."

41.3.2.1 Creating a WAR Deployment Profile

You will need to create a WAR deployment profile for each web-based project you want to package into the application. Typically, the WAR profile will include the dependent data model projects it requires.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 41.3, "Preparing the Application."

You will need to complete this task:

Create web-based projects. If you used the Fusion Web Application (ADF) template, you should already have a default WAR deployment profile.

To create WAR deployment profiles for an application:

1. In the Application Navigator, right-click the web project that you want to deploy and choose **New**.

   You will create a WAR profile for each web project.

2. In the New Gallery, expand **General**, select **Deployment Profiles** and then **WAR File**, and click **OK**.

   If you don't see **Deployment Profiles** in the **Categories** tree, click the **All Features** tab.

3. In the Create Deployment Profile -- WAR File dialog, enter a name for the project deployment profile and click **OK**.

4. In the Edit WAR Deployment Profile Properties dialog, choose items in the left pane to open dialog pages in the right pane. Configure the profile by setting property values in the pages of the dialog.

   - If you have customization classes in your application, they must be loaded from the EAR-level application class loader and not from the WAR. You will later add these customization classes to the EAR.

     By default, customization classes are added to the data model project's WAR class path. So for each WAR, you must exclude the customization classes.

     If you created your customization classes in an extension project of the application, be sure to deselect any customization class archive on the Library Dependencies page of the WAR deployment profile for each user interface project.

     If you created your customization classes in the data model project of the application, deselect any customization classes in the Edit WAR Deployment Profiles Properties dialog Filters page for each user interface project. If you are using a `customization.properties` file, it should also be deselected.

   - You might also want to change the Java EE web context root setting. To do so, choose **General** in the left pane.

     By default, when **Use Project's Java EE Web Context Root** is selected, the associated value is set to the project name, for example, `Application1-Project1-context-root`. You need to change this if you want users to use a different name to access the application.

     If you are using custom JAAS LoginModule for authentication with JAZN, the context root name also defines the application name that is used to look up the JAAS LoginModule.

5. Click **OK** to exit the Deployment Profile Properties dialog.

6. Click **OK** again to exit the Project Properties dialog.

7. Repeat Steps 1 through 7 for all web projects that you want to deploy.

41.3.2.2 Creating a MAR Deployment Profile

If you have seeded customizations or base metadata that you want to place in the MDS repository, you need to create a MAR deployment profile.

The namespace configuration under `<mds-config>` for MAR content in the `adf-config.xml` file is generated based on your selections in the MAR Deployment Profile Properties dialog.

Although uncommon, an enterprise application (packaged in an EAR) can contain multiple web application projects (packaged in multiple WARs), but the metadata for all these web applications will be packaged into a single metadata archive (MAR). The metadata contributed by each of these individual web applications can be global (available for all the web applications) or local to that particular web application.

To avoid name conflicts for metadata with global scope, make sure that all metadata objects and elements have unique names across all the web application projects that forms part of the enterprise application.

To avoid name conflicts and to ensure that the metadata for a particular web application remains local to that application, you can define a `web-app-root` for that web application project.

The `web-app-root` is an element in the `adf-settings.xml` file for a web application project. The `adf-settings.xml` file should be kept in the `META-INF` directory under the `public_html` directory for the web project. Example 41–1 shows the contents of a sample `adf-settings.xml` file.

Example 41–1 web-app-root Element in the adf-settings.xml File

```
<?xml version="1.0" encoding="UTF-8" ?>
 <adf-settings xmlns="http://xmlns.oracle.com/adf/settings"
 xmlns:wap="http://xmlns.oracle.com/adf/share/http/config">
    <wap:adf-web-config xmlns="http://xmlns.oracle.com/adf/share/http/config">
        <web-app-root rootName="order"/>
    </wap:adf-web-config>
</adf-settings>
```

In this example, the `adf-settings.xml` file has a `web-app-root` element that defines the `rootName` as `order`.

If your enterprise application has only one web application project then there is no need to define a `web-app-root` element. If your enterprise application has multiple web application projects, then you should supply a `web-app-root` for all the web applications except one, without which the deployment would fail. For example, if you have `web-application1`, `web-application2`, and `web-application3`, two of these web application projects must define a `web-app-root` to preclude any name conflicts.

JDeveloper creates an auto-generated MAR when the **Enable User Customizations** and **Across Sessions using MDS** options are selected in the ADF View page of the Project Properties dialog or when you explicitly specify the deployment target directory in the `adf-config.xml` file.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 41.3, "Preparing the Application."

You will need to complete this task:

Create an MDS repository for your customization requirements to deploy metadata using the MAR deployment profile. If you used the Fusion Web Application (ADF) template, you should already have a default MAR deployment profile.

To create a MAR deployment profile:

1. In the Application Navigator, right-click the application and choose **New**.

You will create a MAR profile if you want to include customizations.

2. In the New Gallery, expand **General**, select **Deployment Profiles** and then **MAR File,** and click **OK**.

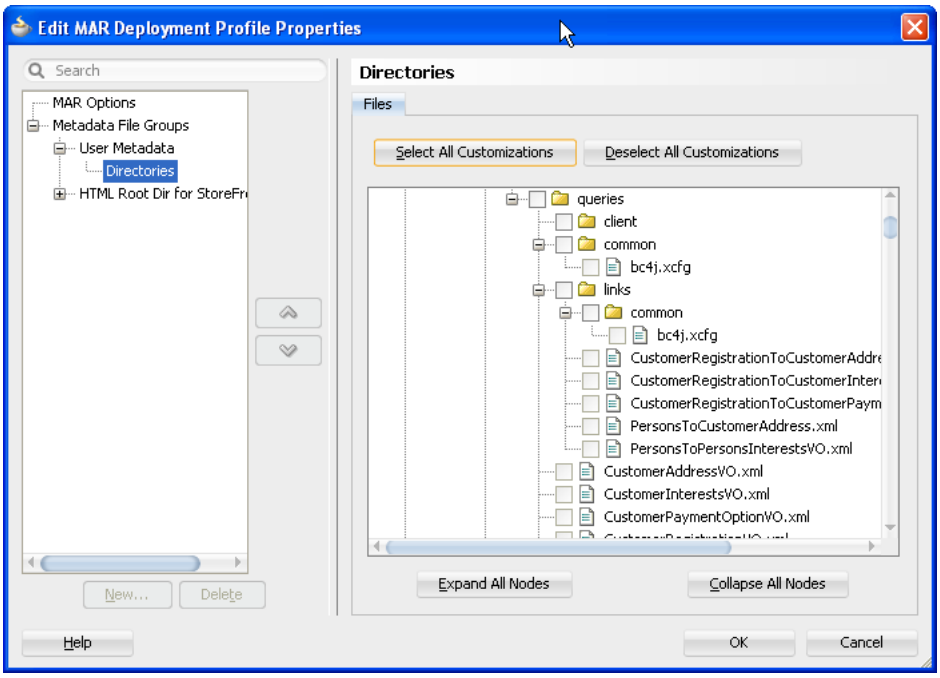
   If you don't see **Deployment Profiles** in the **Categories** tree, click the **All Features** tab.

3. In the Create Deployment Profile -- MAR File dialog, enter a name for the MAR deployment profile and click **OK**.

4. In the Edit MAR Deployment Profile Properties dialog, choose items in the left pane to open dialog pages in the right pane.

   Figure 41–4 shows a sample **User Metadata** directory tree.

Figure 41–4 Selecting Items for the MAR Deployment Profiles



Note the following important points:

- To include all customizations, you need only create a file group with the desired directories.
- ADF Model and ADF view directories are added by default. No further action is required to package the ADF Model and ADF view customizations into the MAR. ADF view content is added to **HTML Root dir**, while ADF Model and Business Components content is added to **User Metadata**.
- To include the base metadata in the MDS repository, you need to explicitly select these directories in the dialog.

  When you select the base document to be included in the MAR, you also select specific packages. When you select one package, all the documents (including subpackages) under that package will be used. When you select a package, you cannot deselect individual items under that package.

- To include files from other than ADF Model and ADF view, users should create a new file group under **User Metadata** with the desired directories and explicitly select the required content in the Directories page.
- If a dependent ADF library JAR for the project contains seeded customizations, they will automatically be added to the MAR during MAR packaging. They will not appear in the MAR profile.
- If ADF Library customizations were created in the context of the consuming project, those customizations would appear in the MAR profile dialog by default.

5. Click **OK** to exit the Deployment Profile Properties dialog.
6. Click **OK** again to exit the Application Properties dialog.

41.3.2.3 Creating an Application-Level EAR Deployment Profile

The EAR file contains all the necessary application artifacts for the application to run in the application server. If you used the Fusion Web Application (ADF) template, you should already have a default EAR deployment profile. For more information about the EAR file, see Section 41.4.5, "What You May Need to Know About EAR Files and Packaging."

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 41.3, "Preparing the Application."

You will need to complete this task:

1. Add classes into a JAR file, as described in Section 41.3.2.6, "Adding Customization Classes into a JAR."
2. Create the WAR deployment profiles, as described in Section 41.3.2.1, "Creating a WAR Deployment Profile."

To create an EAR deployment profile for an application:

1. In the Application Navigator, right-click the application and choose **New**.

   You will create an EAR profile for the application.

2. In the New Gallery, expand **General**, select **Deployment Profiles** and then **EAR File,** and click **OK**.

   If you don't see **Deployment Profiles** in the Categories tree, click the **All Features** tab.

3. In the Create Deployment Profile -- EAR File dialog, enter a name for the application deployment profile and click **OK**.
4. In the Edit EAR Deployment Profile Properties dialog, choose items in the left pane to open dialog pages in the right pane. Configure the profile by setting property values in the pages of the dialog.

   Be sure that you:

   - Select **Application Assembly** and then in the **Java EE Modules** list, select all the project profiles that you want to include in the deployment, including any **WAR** or **MAR** profiles.

- Select **Platform**, select the application server you are deploying to, and then select the target application connection from the **Target Connection** dropdown list.

  Note: If you are using a custom JAAS LoginModule for authentication with JAZN, the context root name also defines the application name that is used to look up the JAAS LoginModule.

5. If you have customization classes in your application, configure these classes so that they load from the EAR-level application class loader.
   a. In the Edit EAR Deployment Profile Properties dialog, select **Application Assembly**.
   b. Select the JAR deployment profile that contains the customization classes, and enter `lib` in the **Path in EAR** field at the bottom of the dialog.

      Note: You should have created this JAR as described in Section 41.3.2.6, "Adding Customization Classes into a JAR."

   The JAR file containing the customization classes is added to the EAR file's `lib` directory.

   Note: If you have customization classes in your application, you must also make sure they are not loaded from the WAR. By default, customization classes that are added to the data model project's libraries and class path are packaged to the WAR class path.

   To make sure customization classes from an extension project are not duplicated in the WAR, be sure to deselect any customization class archive on the Library Dependencies page for the WAR.

   If you created your customization classes in the data model project of the consuming application, deselect any customization classes in the Edit WAR Deployment Profile Properties dialog Filters page.

6. Click **OK** again to exit the Edit EAR Deployment Profile Properties dialog.
7. Click **OK** again to exit the Application Properties dialog.

   Note: To verify that your customization classes are put correctly in the EAR class path, you can deploy the EAR profile to file system. Then you can examine the EAR to make sure that the customization class JAR is available in the EAR class path (the `EAR/lib` directory) and not available in the WAR class path (the `WEB-INF/lib` and `WEB-INF/classes` directories).

41.3.2.4 Delivering Customization Classes as a Shared Library

As an alternative to adding your customization classes to the EAR, as described in Section 41.3.2.3, "Creating an Application-Level EAR Deployment Profile," you can also include the customization classes in the consuming application as a shared library.

Note: This procedure describes how to create and use a shared library if you are deploying to Oracle Weblogic Server.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 41.3, "Preparing the Application."

You will need to complete this task:

Add customization classes into a JAR using JDeveloper in Studio Developer role. Follow the procedure described in Section 41.3.2.6, "Adding Customization Classes into a JAR," and make sure that you select **Shared Library JAR File** as the type of archive to create.

To create and use a shared library for your customization classes:

1. In the Application Navigator, right-click the customization classes project, and choose **Deploy** > ***deployment-profile***.
2. In the Deploy wizard, select **Deploy to a Weblogic Application Server** and click **Next**.
3. Select the appropriate application server, and click **Finish**.

   This makes the shared library available on the application server. You must now add a reference to the shared library from the consuming application.
4. Open the application you want to customize in JDeveloper in the Studio Developer role.
5. In the Application Resources panel of the Application Navigator, double-click the **`weblogic-application.xml`** file to open it.
6. In the overview editor, click the Libraries tab.
7. In the **Shared Library References** section, click the **Add** icon.
8. In the **Library Name** field of the newly created row in the **Shared Library References** table, enter the name of the customization classes shared library you deployed, and save your changes.

41.3.2.5 Viewing and Changing Deployment Profile Properties

After you have created a deployment profile, you can view and change its properties.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 41.3, "Preparing the Application."

To view, edit, or delete a project's deployment profile:

1. In the Application Navigator, right-click the project and choose **Project Properties**.
2. In the Project Properties dialog, click **Deployment**.

   The **Deployment Profiles** list displays all profiles currently defined for the project.
3. In the list, select a deployment profile.
4. To edit or delete a deployment profile, click **Edit** or **Delete**.

41.3.2.6 Adding Customization Classes into a JAR

If your application has customization classes, create a JAR that contains only these customization classes. When you create your EAR, you can add the JAR to the EAR assembly. And when you create WAR profiles for your web projects, you must make sure they don't include the customization classes JAR.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 41.3, "Preparing the Application."

You will need to complete this task:

Make sure that your project has customization classes. You do not need to perform this procedure if the application does not have customization classes. For more information about customization classes, see Section 39.2.1, "How to Create Customization Classes."

To add customization classes into a JAR:

1. In the Application Navigator, right-click the data model project that contains the customization classes you want to create a JAR for, and choose **New**.
2. In the New Gallery, expand **General**, select **Deployment Profiles** and then **JAR File**, and click **OK**.

   Alternatively, if you want to create a shared library, select **Shared Library JAR File** from the list of profile types, and click **OK**.

   Note: If you don't see **Deployment Profiles** in the **Categories** tree, click the **All Features** tab.

3. In the Create Deployment Profile -- JAR File dialog, enter a name for the project deployment profile (for example, `CCArchive`) and click **OK**.
4. In the Edit JAR Deployment Profile Properties dialog, select **JAR Options**.
5. Enter the location for the JAR file.
6. Expand **Files Groups > Project Output > Filters**.
7. In Filters page Files tab, select the customization classes you want to add to the JAR file.

   If you are using a `customization.properties` file, it needs to be in the same class loader as the JAR file. You can select the `customization.properties` file to package it along with the customization classes in the same JAR.
8. Click **OK** to exit the Edit JAR Deployment Profile Properties dialog.
9. Click **OK** again to exit the Project Properties dialog.
10. In the Application Navigator, right-click the project containing the JAR deployment profile, and choose **Deploy > *deployment profile* > to JAR file**.

    Note: If this is the first time you deploy to a JAR from this deployment profile, you choose **Deploy > *deployment profile*** and select **Deploy to JAR** in the wizard.

41.3.3 How to Create and Edit Deployment Descriptors

Deployment descriptors are server configuration files that define the configuration of an application for deployment and that are deployed with the Java EE application as needed. The deployment descriptors that a project requires depend on the technologies the project uses and on the type of the target application server. Deployment descriptors are XML files that can be created and edited as source files, but for most descriptor types, JDeveloper provides dialogs or an overview editor that you can use to view and set properties. If you cannot edit these files declaratively, JDeveloper opens the XML file in the source editor for you to edit its contents.

In addition to the standard Java EE deployment descriptors (for example, `application.xml` and `web.xml`), you can also have deployment descriptors that are specific to your target application server. For example, if you are deploying to Oracle WebLogic Server, you can also have `weblogic.xml`, `weblogic-application.xml`, and `weblogic-ejb-jar.xml`.

For WebLogic Server, make sure that the application EAR file includes a `weblogic-application.xml` file that contains a reference to `adf.oracle.domain`, and that it includes an `ADFApplicationLifecycleListener` to clean up application resources between deployment and undeployment actions. Example 41–2 shows a sample `weblogic-application.xml` file.

Example 41–2 Sample weblogic-application.xml

```
<weblogic-application xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.bea.com/ns/weblogic/weblogic-application.xsd"
 xmlns="http://www.bea.com/ns/weblogic/weblogic-application">
  <listener>
    <listener-class>oracle.adf.share.weblogic.listeners.
          ADFApplicationLifecycleListener</listener-class>
  </listener>
  <listener>
    <listener-class>oracle.mds.lcm.weblogic.WLLifecycleListener</listener-class>
  </listener>
  <library-ref>
    <library-name>adf.oracle.domain</library-name>
  </library-ref>
</weblogic-application>
```

If you are deploying web services, you may need to modify your `weblogic-application.xml` and `web.xml` files as described in Section 11.2.20, "How to Deploy Web Services to Oracle WebLogic Server."

If you want to enable the application for Real User Experience Insight (RUEI) monitoring, you must add a parameter to the `web.xml` file, as described in Section 41.3.3.5, "Enabling the Application for Real User Experience Insight."

During deployment, the application's security properties are written to the `weblogic-application.xml` file to be deployed with the application in the EAR file. For more information, see Section 35.8.2, "What Happens When You Configure Security Deployment Options."

Because Oracle WebLogic Server runs on Java EE 1.5, you may need to modify the `application.xml` and `web.xml` files to be compatible with the application server.

For IBM WebSphere, the deployment descriptors are created at runtime and cannot be edited. Some of the relevant descriptors are shown in Table 41–1.

Table 41–1 IBM WebSphere Deployment Descriptors

WebSphere	Action
`ibm-application-bnd.xml`	References the security role just mapped in `application.xml` and maps it to the well-known name `AllAuthenticatedUsers`. Similar to `weblogic.xml` for WebLogic Server. Maps the `valid-users` JEE security role to the well-known name `Users`.
`application.xml`	A standard Java EE deployment description. Also used to populate a security mapping for the "valid-users" role (which is defined in `web.xml` when using ADF Security).
`<EAR_ROOT>/META-INF/manifest.mf`	References application shared libraries such as `adf.oracle.domain`
`<EAR_ROOT>/META-INF/deployment.xml`	References WAR shared libraries such as `adf.oracle.domain.webapp`

41.3.3.1 Creating Deployment Descriptors

JDeveloper automatically creates many of the required deployment descriptors for you. If they are not present, or if you need to create additional descriptors, you can use JDeveloper to create them.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 41.3, "Preparing the Application."

You will need to complete this task:

Check to see whether JDeveloper has already generated deployment descriptors.

To create a deployment descriptor:

1. In the Application Navigator, right-click the project for which you want to create a descriptor and choose **New**.
2. In the New Gallery, expand **General**, select **Deployment Descriptors** and then a descriptor type, and click **OK**.

   If you can't find the item you want, make sure that you chose the correct project, and then choose the **All Features** tab or use the **Search** field to find the descriptor. If the item is not enabled, check to make sure that the project does not already have a descriptor of that type. A project is allowed only one instance of a descriptor.

   JDeveloper starts the Create Deployment Descriptor wizard and then opens the file in the overview or source editor, depending on the type of deployment descriptor you choose.

Note: For EAR files, do not create more than one deployment descriptor file of the same type per application or workspace. These files can be assigned to projects, but have application workspace scope. If multiple projects in an application have the same deployment descriptor, the one belonging to the launched project will supersede the others. This restriction applies to `application.xml`, `weblogic-jdbc.xml`, `jazn-data.xml`, and `weblogic.xml`.

The best place to create an application-level descriptor is in the **Descriptors** node of the Application Resources panel in the Application Navigator. This ensures that the application is created with the correct descriptors.

Application-level descriptors created in the project will be ignored at runtime. Only the application resources descriptors or descriptors generated at the EAR level will be used by the runtime.

41.3.3.2 Viewing or Modifying Deployment Descriptor Properties

After you have created a deployment descriptor, you can change its properties by using JDeveloper dialogs or by editing the file in the source editor. The deployment descriptor is an XML file (for example, `application.xml`) typically located under the Application Sources node.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 41.3, "Preparing the Application."

To view or change deployment descriptor properties:

1. In the Application Navigator or in the Application Resources panel, double-click the deployment descriptor.
2. In the overview editor, select either the **Overview tab** or the **Source tab**, and configure the descriptor by setting property values.

   If the overview editor is not available, JDeveloper opens the file in the source editor.

41.3.3.3 Configuring the application.xml File for Application Server Compatibility

You may need to configure your `application.xml` file to be compliant with Java EE 1.5.

Note: Typically, your project has an `application.xml` file that is compatible and you would not need to perform this procedure.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 41.3, "Preparing the Application."

To configure the application.xml file:

1. In the Application Navigator, right-click the application and choose **New**.

2. In the New Gallery, expand **General**, select **Deployment Descriptors** and then **Java EE Deployment Descriptor Wizard**, and click **OK**.
3. In the Create Java EE Deployment Descriptor dialog, on the Select Descriptor page, select **application.xml** and click **Next**.
4. On the Select Version page, select **5.0** and click **Next**.
5. On the Summary page, click **Finish**.
6. Edit the `application.xml` file with the appropriate values.

41.3.3.4 Configuring the web.xml File for Application Server Compatibility

You may need to configure your `web.xml` file to be compliant with Java EE 1.5 (which corresponds to servlet 2.5 and JSP 1.2). For more information, see Section A.13, "web.xml."

Note: Typically, your project has a `web.xml` file that is compatible and you would not need to perform this procedure. JDeveloper creates a starter `web.xml` file when you create a project.

If the application uses ADF Security and will be deployed to WebSphere, you need to manually edit the `web.xml` file. For more information, see Section 41.3.4.3.3, "Editing the web.xml File to Protect the Application Root for WebSphere."

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 41.3, "Preparing the Application."

To configure the web.xml file:

1. In the Application Navigator, right-click the project and choose **New**.
2. In the New Gallery, expand **General**, select **Deployment Descriptors** and then **Java EE Deployment Descriptor**, and click **OK**.
3. In the Create Java EE Deployment Descriptor dialog, on the Select Descriptor page, select **web.xml** and click **Next**.
4. On the Select Version page, select **2.5** and click **Next**.
5. On the Summary page, click **Finish**.

41.3.3.5 Enabling the Application for Real User Experience Insight

Real User Experience Insight (RUEI) is a web-based utility to report on real-user traffic requested by, and generated from, your network. It measures the response times of pages and transactions at the most critical points in the network infrastructure. Session diagnostics allow you to perform root-cause analysis.

RUEI enables you to view server and network times based on the real-user experience, to monitor your Key Performance Indicators (KPIs) and Service Level Agreements (SLAs), and to trigger alert notifications on incidents that violate their defined targets. You can implement checks on page content, site errors, and the functional requirements of transactions. Using this information, you can verify your business and technical operations. You can also set custom alerts on the availability, throughput, and traffic of all items identified in RUEI.

For more information about RUEI, see the *Oracle Real User Experience Insight User's Guide* at http://download.oracle.com/docs/cd/E16339_01/doc.60/e16359/toc.htm.

You must enable an application for RUEI by adding the following `context-param` tag to the `web.xml` file, as shown in Example 41–3.

Example 41–3 Enabling RUEI Monitoring for an Application in web.xml

```
<context-param>
  <description>This parameter notifies ADF Faces that the
               ExecutionContextProvider service provider is enabled.
               When enabled, this will start monitoring and aggregating
               user activity information for the client initiated
               requests. By default this param is not set or is false.
  </description>
  <param-name>
         oracle.adf.view.faces.context.ENABLE_ADF_EXECUTION_CONTEXT_PROVIDER
  </param-name>
  <param-value>true</param-value>
</context-param>
```

41.3.4 How to Deploy Applications with ADF Security Enabled

If you are developing an application in JDeveloper using Integrated WebLogic Server, application security deployment properties are configured by default, which means that the application and security credentials and policies will be overwritten each time you redeploy for development purposes. However, the application security deployment properties are the same for Integrated WebLogic Server and the standalone WebLogic Server.

You can change the default behavior in the Application Properties dialog, as described in Section 35.8.1, "How to Configure, Deploy, and Run a Secure Application in JDeveloper."

41.3.4.1 Applications That Will Run Using Oracle Single Sign-On (SSO)

Before you can deploy and run the web application with ADF Security enabled on the application server, the administrator of the target server must configure the domain-level `jps-config.xml` file for the Oracle Access Manager (OAM) security provider. To assist with this configuration task, an Oracle WebLogic Scripting Tool (WLST) script has been provided with the JDeveloper install. For details about running this configuration script (with command `addOAMSSOProvider(loginuri, logouturi, autologinuri)`), see the procedure for configuring Oracle WebLogic Server for a web application using ADF Security, OAM SSO, and OPSS SSO in the *Oracle Fusion Middleware Application Security Guide*.

Running the configuration script ensures that the ADF Security framework defers to the OAM service provider to clear the `ObSSOCookie` token. OAM uses this token to save the identity of authenticated users and, unless it is cleared during logout, the user will be unable to log out.

After the system administrator runs the script on the target server, the domain `jps-config.xml` file will contain the following security provider definition that is specific for ADF Security:

```
<propertySet name="props.auth.uri">
    <property name="login.url.FORM" value="/${app.context}/adfAuthentication"/>
    <property name="logout.url" value=""/>
```

```
</propertySet>
```

Additionally, the authentication type required by SSO is `CLIENT-CERT`. The `web.xml` authentication configuration for the deployed application must specify the `<auth-method>` element as one of the following `CLIENT-CERT` types.

WebLogic supports two types of authentication methods:

- For `FORM`-type authentication method, specify the elements like this:

```
<login-config>
  <auth-method>CLIENT-CERT,FORM</auth-method>
  <realm-name>myrealm</realm-name>
  <form-login-config>
     <form-login-page>/login.html</form-login-page>
     <form-error-page>/error.html</form-error-page>
  </form-login-config>
</login-config>
```

- For `BASIC`-type authentication method, specify the elements like this:

```
<login-config>
   <auth-method>CLIENT-CERT,BASIC</auth-method>
   <realm-name>myrealm</realm-name>
</login-config>
```

WebSphere supports a single authentication method. Specify the elements like this:

```
<login-config>
  <auth-method>CLIENT-CERT</auth-method>
  <realm-name>myrealm</realm-name>
  <form-login-config>
     <form-login-page>/login.html</form-login-page>
     <form-error-page>/error.html</form-error-page>
  </form-login-config>
</login-config>
```

You can configure the `web.xml` file either before or after deploying the web application. For further details about setting up the authentication method for Single Sign-On, see the *Oracle Fusion Middleware Application Security Guide*.

41.3.4.2 Configuring Security for Weblogic Server

In a development environment, JDeveloper will automatically migrate application-level credentials, identities, and policies to the standalone WebLogic Server instance only if the server is set up to be in development mode. Integrated WebLogic Server is set up in development mode by default. You can set up a standalone WebLogic Server to be in development mode during Oracle WebLogic Server domain creation using the Oracle Fusion Middleware Configuration Wizard. For more information about configuring Oracle WebLogic Server domains, see *Oracle Fusion Middleware Creating Domains Using the Configuration Wizard*.

JDeveloper will not migrate application-level security credentials to WebLogic Server setup in production mode. Typically, in a production environment, administrators will use Enterprise Manager or WLST scripts to deploy an application, including its security requirements.

When you deploy an application to WebLogic Server, credentials (in the `cwallet.sso` and `jazn-data.xml` files) will either overwrite or merge with the WebLogic Server's domain-level credential store, depending on whether an option in `weblogic-application.xml` is set to `OVERWRITE` or `MERGE`. In production-mode

WebLogic Server, to avoid security risks, only MERGE is allowed. For development-mode WebLogic Server, you can set to OVERWRITE to test user names and passwords. You can also set the option by running setDomainEnv.cmd or setDomainEnv.sh with the following option added to the command (usually located in ORACLE_HOME/user_projects/domains/*MyDomain*/bin).

For setDomainEnv.cmd:

```
set EXTRA_JAVA_PROPERTIES=-Djps.app.credential.overwrite.allowed=true
    %EXTRA_JAVA_PROPERTIES%
```

For setDomainEnv.sh:

```
EXTRA_JAVA_PROPERTIES="-Djps.app.credential.overwrite.allowed=true
     ${EXTRA_JAVA_PROPERTIES}"
export EXTRA_JAVA_PROPERTIES
```

If the Administration Server is already running, you must restart it for this setting to take effect.

You can check to see whether WebLogic Server is in production mode by using the Oracle WebLogic Server Administration Console or by verifying the following line in WebLogic Server's config.xml file:

```
<production-mode-enabled>true</production-mode-enabled>
```

By default, JDeveloper sets the application's credential, identities, and policies to OVERWRITE mode. That is, the **Application Policies**, **Credentials**, and **Users and Groups** options are selected by default in the Application Properties dialog Deployment page. However, an application's credentials will be migrated only if the target WebLogic Server instance is set to development mode with -Djps.app.credential.overwrite.allowed=true

Policy migration only works in development-mode. Identity migration only works when using JDeveloper to directly deploy to WebLogic Server regardless of whether it is in development or production-mode.

When your application is ready for deployment to a production environment, you should remove the identities from the jazn-data.xml file or disable the migration of identities by deselecting **Users and Groups** from the Application Properties dialog. Application credentials must be manually migrated outside of JDeveloper.

> **Note:** Before you migrate the jazn-data.xml file to a production environment, check that the policy store does not contain duplicate permissions for a grant. If a duplicate permission (one that has the same name and class) appears in the file, the administrator migrating the policy store will receive an error and the migration of the policies will be halted. You should manually edit the jazn-data.xml file to remove any duplicate permissions from a grant definition.

For more information about migrating application credentials and other jazn-data user credentials, see the *Oracle Fusion Middleware Application Security Guide*.

41.3.4.2.1 Applications with JDBC URL for WebLogic

If your application has components that use JDBC URL connections, the connection user names and passwords are also stored in the application-level credential and policy stores. For the deployed application to be able to connect to the database using the JDBC URL, these credentials and policies must be migrated. That is, if WebLogic

Server is in production mode, system administrators must migrate this security information. If WebLogic Server is in development mode, it must have domain-level credential and policy stores set to `OVERWRITE` to allow the migration of security information.

41.3.4.2.2 Applications with JDBC Data Source for WebLogic

If your application uses application-level JDBC data sources with password indirection for database connections, you may need to create credential maps in WebLogic Server to enable the database connection. For more information, see the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

41.3.4.3 Configuring Security for Websphere Application Server

Applications with credentials and policies in the `jazn-data.xml` and `cwallet.sso` files can be migrated to WebSphere. You will need to perform additional tasks in WebSphere. Be aware that the `opss-application.xml` file is not included in the application EAR file if it is intended for WebSphere Application Server.

> **Note:** Before you migrate the `jazn-data.xml` file to a production environment, check that the policy store does not contain duplicate permissions for a grant. If a duplicate permission (one that has the same name and class) appears in the file, the administrator migrating the policy store will receive an error and the migration of the policies will be halted. You should manually edit the `jazn-data.xml` file to remove any duplicate permissions from a grant definition.

For more information about setting up WebSphere to accept credentials and policies, see the *Oracle Fusion Middleware Third-Party Application Server Guide*.

41.3.4.3.1 Applications with JDBC URL for WebSphere

If your application has components that use JDBC URL connections, the connection user names and passwords are also stored in the application-level credential and policy stores. For the deployed application to be able to connect to the database using the JDBC URL, OPSS migration must be enabled.

41.3.4.3.2 Applications with JDBC Data Source for WebSphere

If your application uses application-level JDBC data sources with password indirection for database connections, you will need to create a JDBC data source in WebSphere For more information, see IBM WebSphere documentation.

41.3.4.3.3 Editing the web.xml File to Protect the Application Root for WebSphere

When you enable ADF Security for your web application, the `web.xml` file includes the Java EE security constraint `allPages` to protect the Java EE application root. By default, to support deploying to Oracle WebLogic Server, JDeveloper specifies the URL pattern for the security constraint as `/` (backslash). If you intend to deploy the application to IBM WebSphere, the correct URL pattern is `/*` (backslash-asterisk). Before you deploy the application to WebSphere, manually edit the `web.xml` file for your application to change the `allPages` security constraint as follows:

```
<security-constraint>
   <web-resource-collection>
      <web-resource-name>allPages</web-resource-name>
```

```
        <url-pattern>/*</url-pattern>
    </web-resource-collection>
    . . .
</security-constraint>
```

41.3.5 How to Replicate Memory Scopes in a Clustered Environment

If you are deploying an application that is intended to run in a clustered environment, you need to ensure that all managed beans with a lifespan longer than one request are serializable, and that the ADF framework is aware of changes to managed beans stored in ADF scopes (view scope and page flow scope).

For more information, see Section 24.4.3, "How to Set Managed Bean Memory Scopes in a Server-Cluster Environment."

41.3.6 How to Enable the Application for ADF MBeans

An ADF application uses many XML files for setting configuration information. Three of these configuration files have ADF MBean counterparts that are deployed with the application. After the application has been deployed, you can change configuration properties by accessing the ADF MBeans using the Enterprise Manager Fusion Middleware Control MBean browser.

To enable ADF MBeans, register them in the `web.xml` file. Example 41–4 shows a `web.xml` file with listener entries for connections, configuration, and business components.

Example 41–4 Enabling ADF MBeans in the web.xml File

```
<listener>
   <listener-class>
        oracle.adf.mbean.share.connection.ADFConnectionLifeCycleCallBack
   </listener-class>
</listener>
<listener>
   <listener-class>
        oracle.adf.mbean.share.config.ADFConfigLifeCycleCallBack</listener-class>
</listener>
<listener>
   <listener-class>
        oracle.bc4j.mbean.BC4JConfigLifeCycleCallBack</listener-class>
</listener>
```

Additionally, ADF connection MBeans require the application to be configured with an MDS repository. ADF Business Components MBeans do not require MDS, but business components configuration is limited to updating the underlying `bc4j.xcfg` file in the exploded EAR location. MDS configuration entries in the `adf-config.xml` file for a file-based MDS are shown in Example 41–5. For more information about configuring MDS, see the *Oracle Fusion Middleware Administrator's Guide*.

Example 41–5 MDS Configuration Entries in the adf-config.xml File

```
<adf-mds-config xmlns="http://xmlns.oracle.com/adf/mds/config">
   <mds-config xmlns="http://xmlns.oracle.com/mds/config" version="11.1.1.000">
      <persistence-config>
         <metadata-store-usages>
            <metadata-store-usage
                default-cust-store="true" deploy-target="true" id="myStore">
            </metadata-store-usage>
```

```
        </metadata-store-usages>
      </persistence-config>
   </mds-config>
</adf-mds-config>
```

In a production environment, an MDS repository that uses a database is required. You can use JDeveloper, Enterprise Manager Fusion Middleware Control or scripts to switch from a file-based repository to a database MDS repository.

Additionally, if several applications are sharing the same MDS configuration, each application can achieve distinct customization layers by defining a `adf:adf-properties-child` property in the `adf-config.xml` file. JDeveloper automatically generates this entry when creating applications. If your `adf-config.xml` file does not have this entry, add it to the file with code similar to that of Example 41–6.

Example 41–6 Adding MDS Partition Code to the adf-config.xml File

```
<adf:adf-properties-child xmlns="http://xmlns.oracle.com/adf/config/properties">
     <adf-property name="adfAppUID" value="Application3-4434"/>
     <adf-property name="partition_customizations_by_application_id"
            value="true"/>
</adf:adf-properties-child>
```

The `value` attribute is either generated by JDeveloper or you can set it to any unique identifier within the server farm where the application is deployed. This value can be set to the `value` attribute of the `adfAppUID` property.

When `adf-property name` is set to `adfAppUid`, then the corresponding `value` property should be set to the name of the application. By default, JDeveloper generates the `value` property using the application's package name. If the package name is not specified, JDeveloper generates the `value` property by using the workspace name and a four digit random number.

For more information about configuring ADF applications using ADF MBeans, see the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

41.3.7 What You May Need to Know About JDBC Data Source for Oracle WebLogic Server

ADF applications can use either a JDBC data source or a JDBC URL for database connections. You use the Oracle WebLogic Server Administration Console to configure a JDBC data source. For more information about database access, see Section 9.3, "Configuring Your Application Module Database Connection."

> **Best Practice:** Fusion web applications are not compatible with data sources defined with the JDBC XA driver. When creating a data source on Oracle WebLogic Server, be sure to change the Fusion web application data source's JDBC driver from "Oracle's Driver (Thin XA)" to "Oracle's Driver (Thin)". Because XA data sources close all cursors upon commit, random JBO-27122 and closed statement errors may result when running the the Fusion web application with an XA data source.

The ADF application module in the data model project can be configured to use a JDBC URL connection type, a JDBC data source connection type, or a combination of

both types. By default, ADF application modules use a JDBC URL connection. A component that uses a JDBC URL will attempt to connect directly to the database using the JDBC URL, and it will ignore any JDBC data sources (global or application-level) that are available in WebLogic Server. For more information about migrating JDBC URL security information (user names and passwords) from the application to WebLogic Server, see Section 41.3.4, "How to Deploy Applications with ADF Security Enabled."

An ADF application can use a JDBC data source to connect to the database. A JDBC data source has three types: global, application level, and application level with password indirection. You generally set up a global JDBC data source in WebLogic Server. Any application that requires access to that database can use that JDBC data source. An application can also include application-level JDBC data sources. When the application is packaged for deployment, if the **Auto Generate and Synchronize weblogic-jdbc.xml Descriptor During Deployment** option is selected, JDeveloper creates a *connection_name*`-jdbc.xml` file for each connection that was defined. Each connection's information is written to the corresponding *connection_name*`-jdbc.xml` file (entries are also changed in `weblogic-application.xml` and `web.xml`). When the application is deployed to WebLogic Server, the server looks for application-level data source information before it looks for the global data source.

If the application is deployed with password indirection set to `true`, WebLogic Server will look for the *connection_name*`-jdbc.xml` file for user name information and it will then attempt to locate application-level credential maps for these user names to obtain the password. If you are using JDeveloper to directly deploy the application to WebLogic Server, JDeveloper automatically creates the credential map and populates the map to the server using an MBean call.

However, if you are deploying to an EAR file, JDeveloper will not be able to make the MBean call to WebLogic Server. You must set up the credential maps using the Oracle WebLogic Administration Console. Even if you have a global JDBC data source set up, if you do not also have credential mapping set up, WebLogic Server will not be able to map the credentials with passwords and the connection will fail.

Once the data source has been created in Oracle WebLogic Server, it can be used by an application module. For more information, see the "Preparing the Standalone Application Server for Deployment" section of the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

41.4 Deploying the Application

You can use JDeveloper to deploy ADF applications directly to the standalone application server or you can create an archive file and use other tools to deploy to the application server.

> **Note:** Before you begin to deploy applications that use Oracle ADF to the standalone application server, you need to prepare the application server environment by performing tasks such as installing the ADF runtime and creating and extending domains or cells. For more information, see the "Preparing the Standalone Application Server for Deployment" section of the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

Figure 41–5 show the process flow for deploying an application and also for deploying customizations to the target standalone application server.

Figure 41–5 Application Deployment Flow Diagram

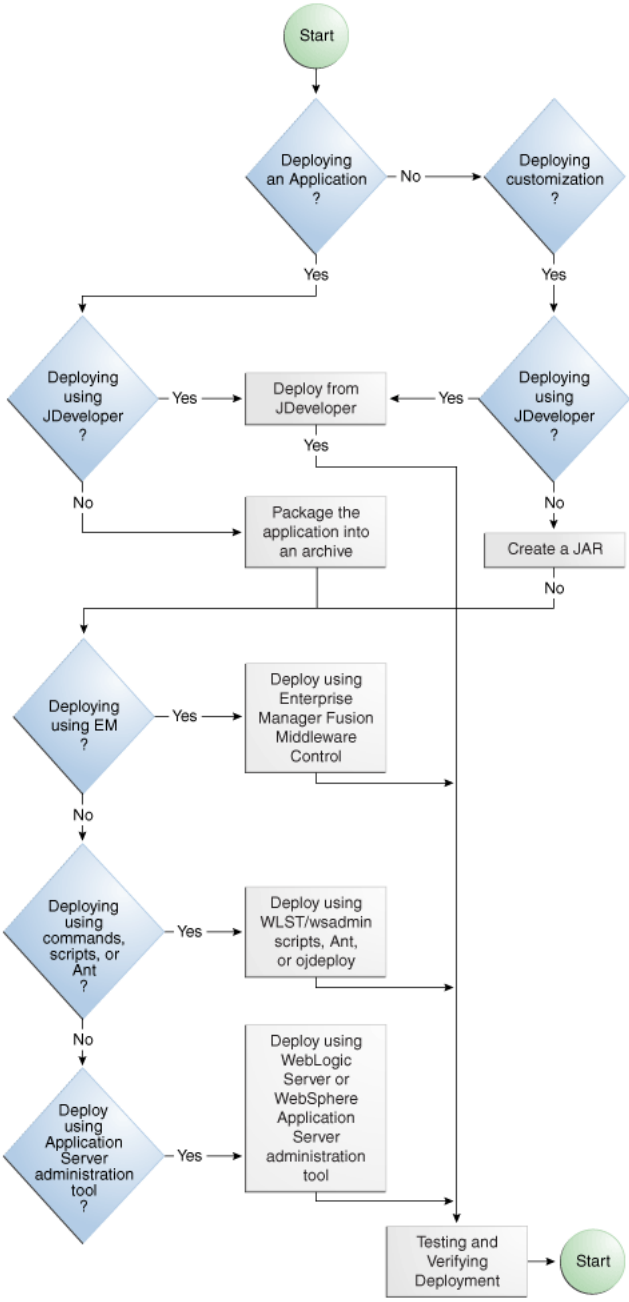


Table 41–2 describes some common deployment techniques that you can use during the application development and deployment cycle. The deployment techniques are listed in order from deploying on development environments to deploying on production environments. It is likely that in the production environment, the system administrators deploy applications by using Enterprise Manager Fusion Middleware Control or scripts.

Table 41–2 Deployment Techniques for Development or Production Environments

Deployment Technique	Environment	When to Use
Run directly from JDeveloper	Test or Development	When you are developing your application. You want deployment to be quick because you will be repeating the editing and deploying process many times. JDeveloper contains Integrated WebLogic Server, on which you can run and test your application.
Use JDeveloper to directly deploy to the target application server	Test or Development	When you are ready to deploy and test your application on an application server in a test environment. On the test server, you can test features (such as LDAP and Oracle Single Sign-On) that are not available on the development server. You can also use the test environment to develop your deployment scripts, for example, using Ant.
Use JDeveloper to deploy to an EAR file, then use the target application server's tools for deployment	Test or Development	When you are ready to deploy and test your application on an application server in a test environment. As an alternative to deploying directly from JDeveloper, you can deploy to an EAR file and then use other tools to deploy to the application server. On the test server, you can test features (such as LDAP and Oracle Single Sign-On) that are not available on the development server. You can also use the test environment to develop your deployment scripts, for example, using Ant.
Use Enterprise Manager or WLST script to deploy applications	Production	When your application is in a test and production environment. In production environments, system administrators usually use Enterprise Manager or run WLST scripts to deploy applications.

Any necessary MDS repositories must be registered with the application server. If the MDS repository is a database, the repository maps to a data source with MDS-specific requirements.

If you are deploying the application to Oracle WebLogic Server, make sure to target this data source to the WebLogic Administration Server and to all Managed Servers to which you are deploying the application. For more information about registering MDS, see the *Oracle Fusion Middleware Administrator's Guide*.

If you are using the application server's administrative console or scripts to deploy an application packaged as an EAR file that requires MDS repository configuration in `adf-config.xml`, you must run the `getMDSArchiveConfig` command to configure MDS before deploying the EAR file. MDS configuration is required if the EAR file contains a MAR file or if the application is enabled for DT@RT (Design Time At Run Time).

For more information about WLST commands, see the *Oracle Fusion Middleware WebLogic Scripting Tool Command Reference*. For more information about wsadmin commands, see the *Oracle Fusion Middleware Third-Party Application Server Guide* and the *Oracle Fusion Middleware Configuration Guide for IBM WebSphere*.

Note: For IBM WebSphere Application Server, after you have deployed the application, you need to perform additional tasks such as starting the server.

For more information about IBM WebSphere Application Server, wsadmin commands, and using the WebSphere Administrative Console, see the *Oracle Fusion Middleware Third-Party Application Server Guide*.

For more information about wsadmin commands specific to Oracle ADF and installing the ADF runtime into the IBM WebSphere Application Server, see the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

If you plan to configure ADF connection information, ADF Business Components information, or `adf-config.xml` using ADF MBeans after the application has been deployed, make sure that the application is configured with MDS and have the MBean listeners enabled in the `web.xml` file. For more information, see Section 41.3.6, "How to Enable the Application for ADF MBeans."

Note: If your ADF application has business services that you want to deploy to WebLogic Server, see Section 11.2.20, "How to Deploy Web Services to Oracle WebLogic Server."

41.4.1 How to Deploy to the Application Server from JDeveloper

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 41.4, "Deploying the Application."

You will need to complete this task:

Create an application-level deployment profile that deploys to an EAR file.

Note: When you are deploying to Oracle WebLogic Server from JDeveloper, ensure that the HTTP Tunneling property is enabled in the Oracle WebLogic Server Administration Console. This property is located under **Servers > *ServerName* > Protocols**. ***ServerName*** refers to the name of Oracle WebLogic Server.

Note: JDeveloper does not support deploying applications to individual Managed Servers that are members of a cluster. You may be able to target one or more Managed Servers within a cluster using the Oracle WebLogic Server Administration Console or other Oracle WebLogic tools; however, the cluster can be negatively affected. For more information about deploying to Oracle WebLogic Server clusters, see the *Oracle Fusion Middleware Administrator's Guide*.

To deploy to the target application server from JDeveloper:

1. In the Application Navigator, right-click the application and choose **Deploy >** ***deployment profile***.
2. In the Deploy wizard, on the Deployment Action page, select **Deploy to Application Server** and click **Next**.
3. On the Select Server page, select the application server connection, and click **Next**.
4. If you are deploying to a WebLogic Server instance, the WebLogic Options page appears. Select a deployment option and click **Next**.

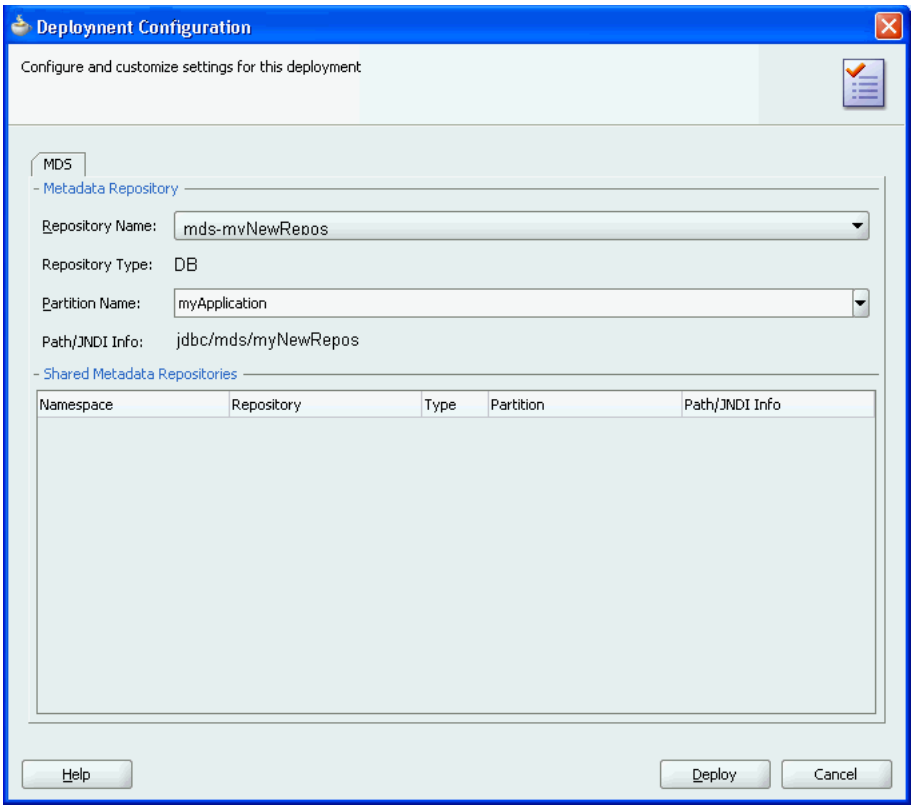
   Note: If you are deploying an ADF application, do not use the **Deploy to all instances in the domain** option.

5. Click **Finish**.

   During deployment, you can see the process steps displayed in the deployment Log window. You can inspect the contents of the modules (archives or exploded EAR) being created by clicking on the links that are provided in the log window. The archive or exploded EAR file will open in the appropriate editor or directory window for inspection.

   If the `adf-config.xml` file in the EAR file requires MDS repository configuration, the Deployment Configuration dialog appears for you to choose the target metadata repository or shared metadata repositories, as shown in Figure 41–6. The **Repository Name** dropdown list allows you to choose a target metadata repository from a list of metadata repositories registered with the Administration Server. The **Partition Name** dropdown list allows you to choose the metadata repository partition to which the application's metadata will be imported during deployment. You can use WLST/wsadmin commands, the Oracle WebLogic Server Administration Tool, or WebSphere Administrative Tool, respectively, to configure and register MDS. For more information about managing the MDS Repository, see the *Oracle Fusion Middleware Administrator's Guide*.

Figure 41–6 MDS Configuration and Customization for Deployment



Note: If you are deploying a Java EE application, click the application menu next to the Java EE application in the Application Navigator.

For more information on creating application server connections, see Section 41.3.1, "How to Create a Connection to the Target Application Server."

41.4.2 How to Create an EAR File for Deployment

You can also use the deployment profile to create an archive file (EAR file). You can then deploy the archive file using Enterprise Manager, WLST/wsadmin scripts, Oracle WebLogic Server Administration Console, or WebSphere Administrative Tool, respectively.

Although an ADF application is encapsulated in an EAR file (which usually includes WAR, MAR, and JAR components), it may have parts that are not deployed with the EAR. For instance, ADF Business Services can be deployed as a JAR. For more information about business services, see Section 11.2.20, "How to Deploy Web Services to Oracle WebLogic Server."

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 41.4, "Deploying the Application."

You will need to complete this task:

Create an application-level deployment profile that deploys to an EAR file.

To create an EAR archive file:

- In the Application Navigator, right-click the application containing the deployment profile, and choose **Deploy > *deployment profile* > to EAR file**.

  If an EAR file is deployed at the application level, and it has dependencies on a JAR file in the data model project and dependencies on a WAR file in the user interface project, then the files will be located in the following directories by default:

  - *ApplicationDirectory*/deploy/*EARdeploymentprofile*.EAR
  - *ApplicationDirectory*/*ModelProject*/deploy/*JARdeploymentprofile*.JAR
  - *ApplicationDirectory*/*ViewControllerProject*/deploy/*WARdeploymentprofile*.WAR

    Tip: Choose **View** >**Log** to see messages generated during creation of the archive file.

41.4.3 How to Deploy New Customizations Applied to ADF Library

If you have created new customizations for an ADF Library, you can use the MAR profile to deploy these customizations to any deployed application that consumes that ADF Library. For instance, suppose `applicationA`, which consumes `ADFLibraryB`, is deployed to a standalone application server. Later on, when new customizations are added to `ADFLibraryB`, you only need to deploy the updated customizations into `applicationA`. You do not need to repackage and redeploy the whole application nor do you need to manually patch the MDS repository.

Note: This procedure is for applying ADF Library customizations changes to an application that has already been deployed to a standalone application server. It is not for the initial packaging of customizations into a MAR that will eventually be a part of an EAR. For information about initial packaging of the customization using a MAR, see Section 41.3.2.2, "Creating a MAR Deployment Profile."

To deploy ADF Library customizations, create a new MAR profile and only include the customizations to be deployed and then use JDeveloper to:

- Deploy the customizations directly into the MDS repository in the standalone application server.
- Deploy the customizations to a JAR. And then import the JAR into the MDS repository using tools such as the Fusion Middleware Control.

41.4.3.1 Exporting Customization to a Deployed Application

You can export the customizations directly from JDeveloper into the MDS repository for the deployed application on the standalone application server.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 41.4, "Deploying the Application."

You will need to complete this task:

Create new customizations to the ADF Library.

To export the customizations directly into the application server:

1. In the Application Navigator, right-click the application and choose **Deploy > metadata**.
2. In the Deploy metadata dialog on the Deployment Action page, select **Export to a Deployed Application**, and click **Next**.

   If the MAR profile is included in the EAR profile of any application, **Export to a Deployed Application** will be dimmed and disabled.
3. On the Application Server page, select the application server connection and click **Next**.
4. For WebLogic Server, the Server Instance page appears. In this page, select the server instance where the deployed application is located and click **Next**.
5. On the Deployed Application page, select the application you want to apply the customizations to and click **Next**.
6. On the Sandbox Instance page, if you want to deploy to a sandbox, select **Deploy to an associated sandbox**, choose the sandbox instance, and click **Next**.
7. On the Summary page, verify the information and click **Finish**.

41.4.3.2 Deploying Customizations to a JAR

When you deploy the ADF Library customizations to a JAR, you are packaging the contents as defined by the MAR profile.

Before you begin:

It may be helpful to have an understanding of the options that are available to you when you create contextual events. For more information, see Section 41.4, "Deploying the Application."

You will need to complete this task:

Create new customizations to the ADF Library.

To deploy the customizations as a JAR

1. In the Application Navigator, right-click the application and choose **Deploy > metadata**.
2. In the Deploy Metadata dialog, on the Deployment Action page, select **Deploy to MAR**.
3. On the Summary page, click **Finish**.
4. Use Enterprise Manager Fusion Middleware Control or the application server's administration tool to import the JAR into the MDS repository.

41.4.4 What You May Need to Know About ADF Libraries

An ADF Library is a JAR file that contains JAR services registered for ADF components such as ADF task flows, pages, or application modules. If you want the ADF components in a project to be reusable, you create an ADF Library deployment profile for the project and then create an ADF Library JAR based on that profile.

An application or project can consume the ADF Library JAR when you add it using the Resource Palette or manually by adding it to the library classpath. When the ADF Library JAR is added to a project, it will be included in the project's WAR file if the **Deployed by Default** option is selected.

For more information, see Chapter 38, "Reusing Application Components."

41.4.5 What You May Need to Know About EAR Files and Packaging

When you package an ADF application into an EAR file, it can contain the following:

- WAR files: Each web-based view controller project should be packaged into a WAR file.
- MAR file: If the application has customizations that are deployed with the application, it should be packaged into a MAR.
- ADF Library JAR files: If the application consumes ADF Library JARs, these JAR files may be packaged within the EAR.
- Other JAR files: The application may have other dependent JAR files that are required. They can be packaged within the EAR.

41.4.6 How to Deploy the Application Using Scripts and Ant

You can deploy the application using commands and automate the process by putting those commands in scripts. The `ojdeploy` command can be used to deploy an application without JDeveloper. You can also use Ant scripts to deploy the application. JDeveloper has a feature to help you build Ant scripts. Depending on your requirements, you may be able to integrate regular scripts with Ant scripts.

For more information about commands, scripts, and Ant, see the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

41.4.7 What You May Need to Know About JDeveloper Runtime Libraries

When an application is deployed, it includes some of its required libraries with the application. The application may also require shared libraries that have already been loaded to WebLogic Server as JDeveloper runtime libraries. It may be useful to know which JDeveloper libraries are packaged within which WebLogic Server shared library. For a listing of the contents of the JDeveloper runtime libraries, see the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

41.5 Postdeployment Configuration

After you have deployed your application to WebLogic Server, you can perform configuration tasks.

41.5.1 How to Migrate an Application

If you want to migrate an ADF application from one WebLogic Server to another WebLogic Server, you may need to perform some of the same steps you did for a first time deployment.

In general, to migrate an application, you would:

- Load the ADF runtime (if it is not already installed) to the target application server.

  For more information, see the "Preparing the Standalone Application Server for Deployment" section of the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

- Configure the target application server with the correct database or URL connection information.

- Migrate security information from the source to the target.

  For instructions, see Section 41.3.4, "How to Deploy Applications with ADF Security Enabled."

- Deploy the application using Enterprise Manager, administration console, or scripts.

  For more information, see the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

41.5.2 How to Configure the Application Using ADF MBeans

If ADF MBeans were enabled and packaged with the deployed application, you can configure ADF properties using the Enterprise Manager MBean Browser. For instructions on how to enable an application for MBeans, see Section 41.3.6, "How to Enable the Application for ADF MBeans."

For information on how to configure ADF applications using ADF MBeans, see the *Oracle Fusion Middleware Administrator's Guide for Oracle Application Development Framework*.

41.6 Testing the Application and Verifying Deployment

After you deploy the application, you can test it from the application server. To test-run your ADF application, open a browser window and enter a URL:

- For non-Faces pages: `http://<host>:port/<context root>/<page>`
- For Faces pages: `http://<host>:port/<context root>/faces/<view_id>`

  where `<view_id>` is the view ID of the ADF task flow view activity.

  Tip: The context root for an application is specified in the user interface project settings by default as *`ApplicationName`*`/`*`ProjectName`*`/context-root`. You can shorten this name by specifying a name that is unique across the target application server. Right-click the user interface project, and choose **Project Properties**. In the Project Properties dialog, select **Java EE Application** and enter a unique name for the context root.

Note: `/faces` has to be in the URL for Faces pages. This is because JDeveloper configures your `web.xml` file to use the URL pattern of `/faces` in order to be associated with the Faces Servlet. The Faces Servlet does its per-request processing, strips out `/faces` part in the URL, then forwards the URL to the JSP. If you do not include the `/faces` in the URL, then the Faces Servlet is not engaged (since the URL pattern doesn't match). Your JSP is run without the necessary JSF per-request processing.

Part VII

Advanced Topics

Part VI contains the following chapters:

- Chapter 42, "Advanced View Object Techniques"
- Chapter 43, "Application State Management"
- Chapter 44, "Tuning Application Module Pools and Connection Pools"
- Chapter 45, "Using the Active Data Service"

42

Advanced View Object Techniques

This chapter describes advanced techniques you can use while designing and working with your view objects.

This chapter includes the following sections:

- Section 42.1, "Advanced View Object Concepts and Features"
- Section 42.2, "Tuning Your View Objects for Best Performance"
- Section 42.3, "Generating Custom Java Classes for a View Object"
- Section 42.4, "Working Programmatically with Multiple Named View Criteria"
- Section 42.5, "Performing In-Memory Sorting and Filtering of Row Sets"
- Section 42.6, "Using View Objects to Work with Multiple Row Types"
- Section 42.7, "Reading and Writing XML"
- Section 42.8, "Using Programmatic View Objects for Alternative Data Sources"
- Section 42.9, "Creating a View Object with Multiple Updatable Entities"
- Section 42.10, "Programmatically Creating View Definitions and View Objects"
- Section 42.11, "Declaratively Preventing Insert, Update, and Delete"

42.1 Advanced View Object Concepts and Features

This section describes a number of interesting view object concepts and features that have not been discussed in previous chapters.

> **Note:** To experiment with the examples in this chapter, use the `AdvancedViewObjectsExamples` workspace in the `StandaloneExamples` module of the Fusion Order Demo application, as described in Section 2.4.5, "Standalone Applications in the AdvancedViewObjectExamples Application Workspace." For information about how to obtain and install the Fusion Order Demo, see Section 2.2, "Setting Up the Fusion Order Demo Application."

42.1.1 Limiting the View Object Max Fetch Size to Fetch the First n Rows

The default maximum fetch size of a view object is minus one (-1), which indicates there should be no limit to the number of rows that can be fetched. Keep in mind that by default, rows are fetched as needed, so -1 does not imply a view object will

necessarily fetch all the rows. It simply means that if you attempt to iterate through all the rows in the query result, you will get them all.

However, you might want to put an upper bound on the maximum number of rows that a view object will retrieve. You can use the following settings:

- You can configure a global threshold for all view objects queries using the **Row Fetch Limit** property on the Business Components page of the overview editor for the `adf-config.xml` file. You can locate the file in the Application Resources pane by expanding the **Descriptors-ADF META-INF** folder.

  Note: Since **Row Fetch Limit** specifies a global threshold for all query operations in the application (including iterator binding property **RowCountThreshold** used to determine an estimated row count for the iterator result set), using this property means you can avoid changing settings for individual query operations where that operation's default behavior allows all rows to be fetched. If you do specify a fetch limit for individual view objects, the **Row Fetch Limit** setting will be ignored in those cases.

- You can configure a local threshold for specific view object queries using the **Max Fetch Size** with the **Only up to row number** field selected in the Tuning section of the General page of the overview editor for the view object.

  Tip: If you want to set the global threshold for query operations using **Row Fetch Limit** and you still need to allow specific view object queries to return all available rows, then you can set the **Max Fetch Size** with the **Only up to row number** field for those view objects to a very large number.

For example, if you write a query containing an `ORDER BY` clause and only want to return the first *n* rows to display the "Top-N" entries in a page, you can use the overview editor for the view object to specify a value for the **Only up to row number** field in the Tuning section of the General page. For example, to fetch only the first five rows, you would enter "5" in this field. This is equivalent to calling the `setMaxFetchSize()` method on your view object to set the maximum fetch size to `5`. The view object will stop fetching rows when it hits the maximum fetch size. Often you will combine this technique with specifying a **Query Optimizer Hint** of `FIRST_ROWS` also on the Tuning section of the General page of the overview editor. This gives a hint to the database that you want to retrieve the first rows as quickly as possible, rather than trying to optimize the retrieval of all rows.

42.1.2 Maintaining New Row Consistency in View Objects Based on the Same Entity

When multiple instances of entity-based view objects in an application module are based on the same underlying entity object, a new row created in one of them can be automatically added (without having to re-query) to the row sets of the others to keep your user interface consistent or simply to consistently reflect new rows in different application pages for a pending transaction. Consider the Fusion Order Demo application's `orderSummary.jspx` page that displays a customers list of orders. If the customer goes to create a new order, this task is performed through a different view object and handled by a custom application module method. Using this view object new row consistency feature, the newly created order automatically appears in the customer's list of open orders on the `orderSummary.jspx` page without having to re-query the database.

For historical reasons, this capability is known as the *view link consistency* feature because in prior releases of Oracle Application Development Framework (Oracle ADF) the addition of new rows to other relevant row sets only was supported for detail view

object instances in a view link based on an association. Now this view link consistency feature works for any view objects for which it is enabled, regardless of whether they are involved in a view link or not.

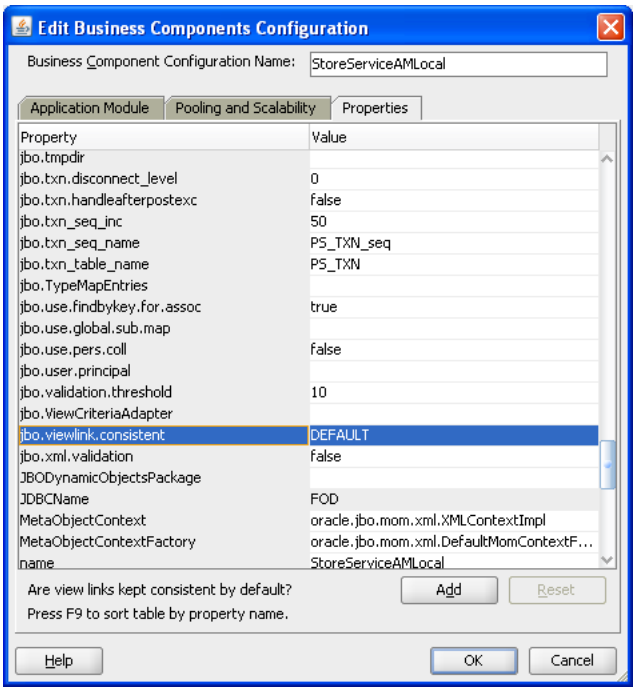
42.1.2.1 What Happens at Runtime When View Link Consistency is Enabled

Consider two entity-based view objects `OrdersViewSummary` and `OrdersView` both based on the same underlying `Orders` entity object. When a new row is created in a row set for one of these view objects (like `OrdersView`) and the row's primary key is set, any of the *other* row sets for view objects based on the same `Orders` entity object (like `OrdersViewSummary`) receive an event indicating a new row has been created. If their view link consistency flag is enabled, then a copy of the new row is inserted into their row set as well.

42.1.2.2 How to Change the Default View Link Consistency Setting

You can use the Edit Business Components Configuration dialog to control the default setting for the view link consistency feature using the `jbo.viewlink.consistent` configuration parameter, as shown in Figure 42–1.

Figure 42–1 jbo.viewlink.consistent Property Setting



To display the configuration editor, right-click the application module in the Application Navigator and choose **Configurations**. Then, in the Manage Configurations dialog, select the configuration and click **Edit**. In the Edit Business Components Configuration dialog, select the Properties tab. The default setting for this parameter is the word "`DEFAULT`" which has the following meaning. If your view object has:

- A single entity usage, view link consistency is enabled
- Multiple entity usages, and:
  - If all secondary entity usages are marked as contributing reference information, then view link consistency is enabled

- If any secondary entity usage marked as **not** being a reference view link consistency is disabled.

You can globally disable this feature by setting the `jbo.viewlink.consistent` to the value `false` in your configuration. Conversely, you could globally enable this feature by setting `jbo.viewlink.consistent` to the value `true`, but Oracle does **not** recommend doing this. Doing so would force view link consistency to be set on for view objects with secondary entity usages that are not marked as a reference which presently do not support the view link consistency feature well.

To set the feature programmatically, use the `setAssociationConsistent()` API on any `RowSet`. When you call this method on a view object, it affects its default row set.

42.1.2.3 How to Use a RowMatch to Qualify Which New, Unposted Rows Get Added to a Row Set

If a view object has view link consistency enabled, any new row created by another view object based on the same entity object is added to its row set. By default the mechanism adds new rows in an unqualified way. If your view object has a design-time `WHERE` clause that queries only a certain subset of rows, you can apply a `RowMatch` object to your view object to perform the same filtering in-memory. The filtering expression of the `RowMatch` object you specify prevents new rows from being added that wouldn't make sense to appear in that view object.

For example, an `OrdersByStatus` view object might include a design time `WHERE` clause like this:

```
WHERE /* ... */ AND STATUS LIKE NVL(:StatusCode,'%')
```

Its custom Java class overrides the `create()` method as shown in Example 42–1 to force view link consistency to be enabled. It also applies a `RowMatch` object whose filtering expression matches rows whose `Status` attribute matches the value of the `:StatusCode` named bind variable (or matches any row if `:StatusCode` = '`%`'). This `RowMatch` filter is used by the view link consistency mechanism to qualify the row that is a candidate to add to the row set. If the row qualifies by the `RowMatch`, it is added. Otherwise, it is not.

Example 42–1 Providing a Custom RowMatch to Control Which New Rows are Added

```
// In OrdersByStatusImpl.java
protected void create() {
  super.create();
  setAssociationConsistent(true);
  setRowMatch(new RowMatch("Status = :StatusCode or :StatusCode = '%'"));
}
```

See Section 42.5.4, "Performing In-Memory Filtering with RowMatch" for more information on creating and using a `RowMatch` object. For a list of supported SQL operators see Table 42–2. For a list of supported SQL function, see Table 42–3.

Note: If the `RowMatch` facility does not provide enough control, you can override the view object's `rowQualifies()` method to implement a custom filtering solution. Your code can determine whether a new row qualifies to be added by the view link consistency mechanism or not.

42.1.2.4 What You May Need to Know About the Dynamic WHERE Clause and View Link Consistency

If you call `setWhereClause()` on a view object to set a dynamic `WHERE` clause, the view link consistency feature is disabled on that view object. If you have provided an appropriate custom `RowMatch` object to qualify new rows for adding to the row set, you can call `setAssociationConsistent(true)` after `setWhereClause()` to reenable view link consistency.

If a row set has view link consistency enabled, then new rows added due to creation by other row sets are added to the bottom of the row set.

If a row set has view link consistency enabled, then when you call the `executeQuery()` method, any qualifying new, unposted rows are added to the top of the row set before the queried rows from the database are added.

42.1.3 Understanding View Link Accessors Versus Data Model View Link Instances

View objects support two different styles of master-detail coordination:

- View link *instances* for data model master-detail coordination, as described in Section 42.1.3.1, "Enabling a Dynamic Detail Row Set with Active Master-Detail Coordination."
- View link *accessor* attributes for programmatically accessing detail row sets on demand, as described in Section 42.1.3.2, "Accessing a Stable Detail Row Set Using View Link Accessor Attributes."
- You can combine both styles, as described in Section 42.1.3.3, "Accessor Attributes Create Distinct Row Sets Based on an Internal View Object."

42.1.3.1 Enabling a Dynamic Detail Row Set with Active Master-Detail Coordination

When you add a view link instance to your application module's data model, you connect two specific view object instances. The use of the view link instance indicates that you want active master-detail coordination between the two. At runtime the view link instance in the data model facilitates the eventing that enables this coordination. Whenever the current row is changed on the master view object instance, an event causes the detail view object to be refreshed by automatically invoking `executeQuery()` with a new set of bind parameters for the new current row in the master view object.

A key feature of this data model master-detail is that the master and detail view object instances are stable objects to which client user interfaces can establish bindings. When the current row changes in the master — instead of producing a *new* detail view object instance — the existing detail view object instance updates its default row set to contain the set of rows related to the new current master row. In addition, the user interface binding objects receive events that allow the display to update to show the detail view object's refreshed row set.

Another key feature that is exclusive to data model hierarchy is that a detail view object instance can have multiple master view object instances. For example, an `PaymentOptions` view object instance may be a detail of *both* a `Customers` and a `Orders` view object instance. Whenever the current row in *either* the `Customers` or `Orders` view object instance changes, the default row set of the detail `PaymentOptions` view object instance is refreshed to include the row of payment information for the current customer and the current order. See Section 42.1.6, "Setting Up a Data Model with Multiple Masters" for details on setting up a detail view object instance with multiple-masters.

42.1.3.2 Accessing a Stable Detail Row Set Using View Link Accessor Attributes

When you need to programmatically access the detail row set related to a view object row by virtue of a view link, you can use the view link accessor attribute. You specify the finder name of the view link accessor attribute from the overview editor for the view link. Click the **Edit** icon in the Accessors section of the Relationship page and in the Edit View Link Properties dialog, edit the name of the view link accessor attribute.

Example 42–2 shows the XML for the view link that defines the `_findername` value of the `<Attr>` element.

Example 42–2 View Link Accessor Attribute Name

```
<ViewLinkDefEnd
    Name="Orders"
    Cardinality="1"
    Owner="devguide.advanced.multiplemasters.Orders"
    Source="true">
    <AttrArray Name="Attributes">
      <Item Value="devguide.advanced.multiplemasters.Orders.PaymentOptionId"/>
    </AttrArray>
    <DesignTime>
      <Attr Name="_minCardinality" Value="1"/>
      <Attr Name="_isUpdateable" Value="true"/>
      <Attr Name="_finderName" Value="Orders"/>
    </DesignTime>
</ViewLinkDefEnd>
```

Assuming you've named your accessor attribute *`AccessorAttrName`*, you can access the detail row set using the generic `getAttribute()` API like:

```
RowSet detail = (RowSet)currentRow.getAttribute("AccessorAttrName");
```

If you've generated a custom view row class for the master view object and exposed the getter method for the view link accessor attribute on the client view row interface, you can write strongly-typed code to access the detail row set like this:

```
RowSet detail = (RowSet)currentRow.getAccessorAttrName();
```

Unlike the data model master-detail, programmatic access of view link accessor attributes does not require a detail view object instance in the application module's data model. Each time you invoke the view link accessor attribute, it returns a `RowSet` containing the set of detail rows related to the master row on which you invoke it.

Using the view link accessor attribute, the detail data rows are stable. As long as the attribute value(s) involved in the view link definition in the master row remain unchanged, the detail data rows will not change. Changing of the current row in the master does not affect the detail row set which is "attached" to a given master row. For this reason, in addition to being useful for general programmatic access of detail rows, view link accessor attributes are appropriate for UI objects like the tree control, where data for each master node in a tree needs to retain its distinct set of detail rows.

42.1.3.3 Accessor Attributes Create Distinct Row Sets Based on an Internal View Object

When you *combine* the use of data model master-detail with programmatic access of detail row sets using view link accessor, it is even more important to understand that they are distinct mechanisms. For example, imagine that you:

- Define `PersonsVO` and `OrdersVO` view objects

- Define a view link between them, naming the view link accessor `PersonsToOrders`
- Add instances of them to an application module's data model named `master` (of type `PersonsVO`) and `detail` (of type `OrdersVO`) coordinated actively by a view link instance.

If you find a person in the `master` view object instance, the `detail` view object instance updates as expected to show the corresponding orders. At this point, if you invoke a custom method that programmatically accesses the `PersonsToOrders` view link accessor attribute of the current `PersonsVO` row, you get a `RowSet` containing the set of `OrdersVO` rows. You might reasonably expect this programmatically accessed `RowSet` to have come from the `detail` view object instance in the data model, but this is not the case.

The `RowSet` returned by a view link accessor always originates from an *internally created* view object instance, not one you that added to the data model. This internal view object instance is created as needed and added with a system-defined name to the root application module.

The principal reason a distinct, internally-created view object instance is used is to guarantee that it remains unaffected by developer-related changes to their own view objects instances in the data model. For example, if the view row were to use the detail view object in the data model for view link accessor `RowSet`, the resulting row set could be inadvertently affected when the developer dynamically:

1. Adds a `WHERE` clause with new named bind parameters

   If such a view object instance were used for the view link accessor result, unexpected results or an error could ensue because the dynamically-added `WHERE` clause bind parameter values have not been supplied for the view link accessor's `RowSet`: they were only supplied for the default row set of the detail view object instance in the data model.

2. Adds an additional master view object instance for the detail view object instance in the data model.

   In this scenario, the semantics of the accessor would be changed. Instead of the accessor returning `OrdersVO` rows for the current `PersonsVO` row, it could all of a sudden start returning *only* the `OrdersVO` rows for the current `PersonsVO` that were created by a current logged in customer, for example.

3. Removes the detail view object instance or its containing application module instance.

   In this scenario, all rows in the programmatically-accessed detail `RowSet` would become invalid.

Furthermore, Oracle ADF needs to distinguish between the data model master-detail and view link accessor row sets for certain operations. For example, when you create a new row in a detail view object, the framework automatically populates the attributes involved in the view link with corresponding values of the master. In the data model master-detail case, it gets these values from the current row(s) of the possibly multiple master view object instances in the data model. In the case of creating a new row in a `RowSet` returned by a view link accessor, it populates these values from the master row on which the accessor was called.

42.1.4 Presenting and Scrolling Data a Page at a Time Using the Range

To present and scroll through data a page at a time, you can configure a view object to manage for you an appropriately-sized range of rows. The range facility allows a client

to easily display and update a subset of the rows in a row set, as well as easily scroll to subsequent pages, *n* rows as a time. You call `setRangeSize()` to define how many rows of data should appear on each page. The default range size is one (`1`) row. A range size of minus one (`-1`) indicates the range should include all rows in the row set.

> **Note:** When using the ADF Model layer's declarative data binding, the iterator binding in the page definition has a `RangeSize` attribute. At runtime, the iterator binding invokes the `setRangeSize()` method on its corresponding row set iterator, passing the value of this `RangeSize` attribute. The ADF design time by default sets this `RangeSize` attribute to `10` rows for most iterator bindings. An exception is the range size specified for a List binding to supply the set of valid values for a UI component like a dropdown list. In this case, the default range size is minus one (`-1`) to allow the range to include all rows in the row set.

When you set a range size greater than one, you control the row set paging behavior using the iterator mode. The two iterator mode flags you can pass to the `setIterMode()` method are:

- `RowIterator.ITER_MODE_LAST_PAGE_PARTIAL`

  In this mode, the last page of rows may contain fewer rows than the range size. For example, if you set the range size to 10 and your row set contains 23 rows, the third page of rows will contain only three rows. This is the style that works best for Fusion web applications.

- `RowIterator.ITER_MODE_LAST_PAGE_FULL`

  In this mode, the last page of rows is kept full, possibly including rows at the top of the page that had appeared at the bottom of the previous page. For example, if you set the range size to 10 and your row set contains 23 rows, the third page of rows will contain 10 rows, the first seven of which appeared as the last seven rows of page two. This is the style that works best for desktop-fidelity applications using Swing.

42.1.5 Efficiently Scrolling Through Large Result Sets Using Range Paging

As a general rule, for highest performance, Oracle recommends building your application in a way that avoids giving the end user the opportunity to scroll through very large query results. To enforce this recommendation, call the `getEstimatedRowCount()` method on a view object to determine how many rows would be returned by the user's query *before* actually executing the query and allowing the user to proceed. If the estimated row count is unreasonably large, your application can demand that the end-user provide additional search criteria.

However, when you *must* work with very large result sets, typically over 100 rows, you can use the view object's access mode called "range paging" to improve performance. The feature allows your applications to page back and forth through data, a range of rows at a time, in a way that is more efficient for large data sets than the default "scrollable" access mode.

The range paging access mode is typically used for paging through read-only row sets, and often is used with read-only view objects. You allow the user to find the row they are looking for by paging through a large row set with range paging access mode, then you use the `Key` of that row to find the selected row in a different view object for editing.

Range paging for view objects supports a standard access mode and a variation of the standard access mode that combines the benefits of range paging and result set scrolling with a minimum number of visits to the database. These modes for the view object range paging feature include:

- `RANGE_PAGING`, standard access mode fetches the number of rows specified by a range size. In this mode, the number of rows that may be scrolled without requerying the database is determined by a range size that you set. The default is to fetch a single row, but it is expected that you will set a range size equal to the number of rows you want to be able to display to the user before they scroll to the next result set. The application requeries the database each time a row outside of the range is accessed by the end user. Thus, scrolling backward and forward through the row set will requery the database. For clarification about this database-centric paging strategy, see Section 42.1.5.1, "Understanding How to Oracle Supports "TOP-N" Queries."
- `RANGE_PAGING_INCR`, incremental access mode gives the UI designer more flexibility for the number of rows to display at a time while keeping database queries to a minimum. In this mode, the UI incrementally displays the result set from the memory cache and thus supports scrolling within a single database query. The number of rows that the end user can scroll though in a single query is determined by the range size and a range paging cache factor that you set. For example, suppose that you set the range size to 4 and the cache factor to 5. Then, the maximum number of rows to cache in memory will be 4*5 = 20. For further explanation of the caching behavior, see Section 42.1.5.4, "What Happens When View Rows are Cached When Using Range Paging."

> **Caution:** Additionally, the view object supports a `RANGE_PAGING_AUTO_POST` access mode to accommodate the inserting and deleting of rows from the row set. This mode behaves like the `RANGE_PAGING` mode, except that it eagerly calls `postChanges()` on the database transaction whenever any changes are made to the row set. However, this mode is typically not appropriate for use in Fusion web applications unless you can guarantee that the transaction will definitely be committed or rolled-back during the same HTTP request. Failure to heed this advice can lead to strange results in an environment where both application modules and database connections can be pooled and shared serially by multiple different clients.

42.1.5.1 Understanding How to Oracle Supports "TOP-N" Queries

The Oracle database supports a feature called a "Top-N" query to efficiently return the first *n* ordered rows in a query. For example, if you have a query like:

```
SELECT EMPNO, ENAME,SAL FROM EMP ORDER BY SAL DESC
```

If you want to retrieve the top 5 employees by salary, you can write a query like:

```
SELECT * FROM (
   SELECT X.*,ROWNUM AS RN FROM (
      SELECT EMPNO,ENAME,SAL FROM EMP ORDER BY SAL DESC
   ) X
) WHERE RN <= 5
```

which gives you results like:

```
     EMPNO ENAME       SAL   RN
```

```
---------- -------- ------ ----
      7839 KING       5000    1
      7788 SCOTT      3000    2
      7902 FORD       3000    3
      7566 JONES      2975    4
      7698 BLAKE      2850    5
```

The feature is not only limited to retrieving the first *n* rows in order. By adjusting the criteria in the outermost `WHERE` clause you can efficiently retrieve any range of rows in the query's sorted order. For example, to retrieve rows `6` through `10` you could alter the query this way:

```
SELECT * FROM (
  SELECT X.*,ROWNUM AS RN FROM  (
     SELECT EMPNO,ENAME,SAL FROM EMP ORDER BY SAL DESC
  ) X
) WHERE RN BETWEEN 6 AND 10
```

Generalizing this idea, if you want to see page number *`P`* of the query results, where each page contains *`R`* rows, then you would write a query like:

```
SELECT * FROM (
  SELECT X.*,ROWNUM AS RN FROM (
    SELECT EMPNO,ENAME,SAL FROM EMP ORDER BY SAL DESC
  ) X
) WHERE RN BETWEEN ((:P - 1) * :R) + 1 AND (:P) * :R
```

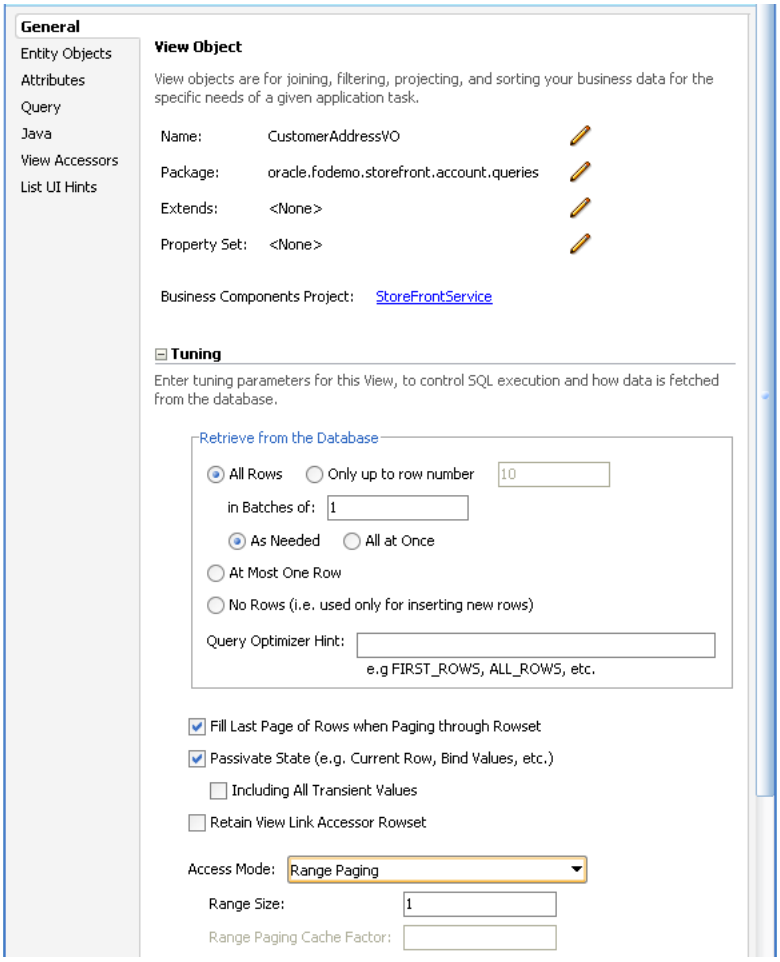
As the result set you consider grows larger and larger, it becomes more and more efficient to use this technique to page through the rows. Rather than retrieving hundreds or thousands of rows over the network from the database, only to display ten of them on the page, instead you can produce a clever query to retrieve only the `R` rows on page number `P` from the database. No more than a handful of rows at a time needs to be returned over the network when you adopt this strategy.

To implement this database-centric paging strategy in your application, you could handcraft the clever query yourself and write code to manage the appropriate values of the `:R` and `:P` bind variables. Alternatively, you can use the view object's range paging access mode, which implements it automatically for you.

42.1.5.2 How to Enable Range Paging for a View Object

You can use the Tuning panel of the overview editor for the view object to set the access mode to either standard range paging or incremental range paging. The **Range Paging Cache Factor** field is only editable when you select **Range Paging Incremental**. Figure 42–2 shows the view object's **Access Mode** set to **Range Paging** (standard mode) with the default range size of 1. To understand the row set caching behavior of both access modes, see Section 42.1.5.4, "What Happens When View Rows are Cached When Using Range Paging."

Figure 42–2 Access Mode in the Overview Editor for the View Object



To programmatically enable standard range paging for your view object, first call `setRangeSize()` to define the number of rows per page, then call the following method with the desired mode:

```
yourViewObject.setAccessMode(RowSet.RANGE_PAGING | RANGE_PAGING_INCR);
```

If you set `RANGE_PAGING_INCR`, then you must also call the following method to set the cache factor for your defined range size:

```
yourViewObject.setRangePagingCacheFactor(int f);
```

42.1.5.3 What Happens When You Enable Range Paging

When a view object's access mode is set to `RANGE_PAGING`, the view object takes its default query like:

```
SELECT EMPNO, ENAME, SAL FROM EMP ORDER BY SAL DESC
```

and automatically "wraps" it to produce a Top-N query.

For best performance, the statement uses a combination of greater than and less than conditions instead of the `BETWEEN` operator, but the logical outcome is the same as the Top-N wrapping query you saw above. The actual query produced to wrap a base query of:

```
SELECT EMPNO, ENAME, SAL FROM EMP ORDER BY SAL DESC
```

looks like this:

```
SELECT * FROM (
  SELECT /*+ FIRST_ROWS */ IQ.*, ROWNUM AS Z_R_N FROM (
    SELECT EMPNO, ENAME, SAL FROM EMP ORDER BY SAL DESC
  ) IQ  WHERE ROWNUM < :0)
WHERE Z_R_N > :1
```

The two bind variables are bound as follows:

- `:1` index of the first row in the current page
- `:0` is bound to the last row in the current page

42.1.5.4 What Happens When View Rows are Cached When Using Range Paging

When a view object operates in `RANGE_PAGING` access mode, it only keeps the current range (or "page") of rows in memory in the view row cache at a time. That is, if you are paging through results ten at a time, then on the first page, you'll have rows 1 through 10 in the view row cache. When you navigate to page two, you'll have rows 11 through 20 in the cache. This also can help make sure for large row sets that you don't end up with tons of rows cached just because you want to preserve the ability to scroll backwards and forwards.

When a view object operates in `RANGE_PAGING_INCR` access mode, the cache factor determines the number of rows to cache in memory for a specific range size. For example, suppose the range size is set to 4 and cache factor to 5. Then, the memory will keep at most 4*5 = 20 rows in its collection. In this example, when the range is refreshed for the first time, the memory will have just four rows even though the range paging query is bound to retrieve rows 0 to 19 (for a total of twenty rows). When the range is scrolled past the forth row, more rows will be read in from the current result set. This will continue until all twenty rows from the query result are read. If the user's action causes the next set of rows to be retrieve, the query will be re-executed with the new row number bind values. The exact row number bind values are determined by the new range-start and the number of rows that can be retained from the cache. For example, suppose all twenty rows have been filled up and the user asks to move the range-start to 18 (0-based). This means that memory can retain row 18 and row 19 and will need two more rows to fill the range. The query is re-executed for rows 20 and 21.

42.1.5.5 How to Scroll to a Given Page Number Using Range Paging

When a view object operates in `RANGE_PAGING` access mode, to scroll to page number *n* call its `scrollToRangePage()` method, passing *n* as the parameter value.

42.1.5.6 Estimating the Number of Pages in the Row Set Using Range Paging

When a view object operates in `RANGE_PAGING` access mode, you can access an estimate of the total number of pages the entire query result would produce using the `getEstimatedRangePageCount()` method.

42.1.5.7 Understanding the Tradeoffs of Using a Range Paging Mode

You might ask yourself, "Why wouldn't I *always* want to use `RANGE_PAGING` or `RANGE_PAGING_INCR` mode?" The answer is that using range paging potentially causes more overall queries to be executed as you are navigating forward and backward through your view object rows. You would want to avoid using `RANGE_PAGING` mode in these situations:

- You plan to read all the rows in the row set immediately (for example, to populate a dropdown list).

  In this case your range size would be set to `-1` and there really is only a single "page" of all rows, so range paging does not add value.

- You need to page back and forth through a small-sized row set.

  If you have 100 rows or fewer, and are paging through them 10 at a time, with `RANGE_PAGING` mode you will execute a query each time you go forward and backward to a new page. Otherwise, in the default scrollable mode, you will cache the view object rows as you read them in, and paging backwards through the previous pages will not re-execute queries to show those already-seen rows. Alternatively, you can use `RANGE_PAGING_INCR` mode to allow scrolling through in-memory results based on a row set cache factor that you determine.

In the case of a very large (or unpredictably large) row set, the trade off of potentially doing a few more queries — each of which only returns up to the `RangeSize` number of rows from the database — is more efficient then trying to cache all of the previously-viewed rows. This is especially true if you allow the user to jump to an arbitrary page in the list of results. Doing so in default, scrollable mode requires fetching and caching all of the rows between the current page and the page the users jumps to. In `RANGE_PAGING` mode, it will ask the database just for the rows on that page. Then, if the user jumps back to a page of rows that they have already visited, in `RANGE_PAGING` mode, those rows get re-queried again since only the current page of rows is held in memory in this mode. The incremental range paging access mode `RANGE_PAGING_INCR` combines aspects of both standard range paging and scrollable access mode since it allows the application to cache more rows in memory and permits the user to jump to any combination of those rows without needing to requery.

42.1.6 Setting Up a Data Model with Multiple Masters

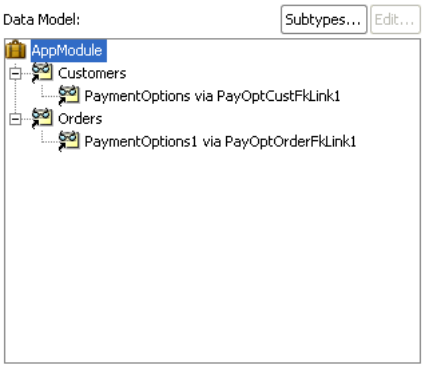
When useful, you can set up your data model to have multiple master view object instances for the same detail view object instance. Consider view objects named `Customers`, `Orders`, and `PaymentOptions` with view links defined between:

- `Customers` and `PaymentOptions`
- `Orders` and `PaymentOptions`

> **Note:** The examples in this section are not based on the Fusion Order Demo application. They currently refer to the `MultipleMasters` project in the `AdvancedViewObjectExamples` workspace, available as noted at the beginning of this chapter for download.

Figure 42–3 shows what the data model panel looks like when you've configured both `Customers` and `Orders` view object instances to be masters of the same `PaymentOptions` view object instance.

Figure 42–3 Multiple Master View Object Instances for the Same Detail



To set up the data model as shown in Figure 42–3 open the overview editor for the application module and follow these steps in the Data Model Components section of the Data Model page:

1. Add an instance of the `Customers` view object to the data model.

   Assume you name it `Customers`.

2. Add an instance of the `Orders` view object to the data model

   Assume you name it `Orders`.

3. Select the `Customers` view object instance in the **Data Model** list

4. In the **Available View Objects** list, select the `PaymentOptions` view object indented beneath the `Customers` view object and enter the view object instance name of `PaymentOptions` in the **New Instance Name** field. Click **>** to shuttle it into data model as a detail of the existing `Customers` view object instance.

5. Select the `Orders` view object instance in the **Data Model** list

6. In the **Available View Objects** list, select the `PaymentOptions` view object indented beneath the `Orders` view object and enter the view object instance name of `PaymentOptions` in the **New Instance Name** field. Click **>** to shuttle it into data model as a detail of the existing `Orders` view object instance.

   An alert will appear: **An instance of a View Object with the name PaymentOptions has already been used in the data model. Would you like to use the same instance?**

7. Click **Yes** to confirm you want the `PaymentOptions` view object instance to *also* be the detail of the `Orders` view object instance.

42.1.7 Understanding When You Can Use Partial Keys with findByKey()

View objects based on multiple entity usages support the ability to find view rows by specifying a partially populated key. A partial key is a multi-attribute `Key` object with some of its attributes set to `null`. However, there are strict rules about what kinds of partial keys can be used to perform the `findByKey()`.

If a view object is based on *n* entity usages, where $n > 1$, then the view row key is by default comprised of all of the primary key attributes from *all* of the participating entity usages. Only the ones from the *first* entity object are *required* to participate in the view row key, but by default all of them do.

If you allow the key attributes from some *secondary* entity usages to remain as key attributes at the view row level, then you should leave *all* of the attributes that form

the primary key for that entity object as part of the view row key. Assuming you have left the one or more key attributes in the view row for *m* of the *n* entity usages, where (*m* <= *n*), then you can use `findByKey()` to find rows based on any subset of these *m* entity usages. Each entity usage for which you provide values in the `Key` object, requires that you must provide non-null values for *all* of the attributes in that entity's primary key.

You have to follow this rule because when a view object is based on at least one or more entity usages, its `findByKey()` method finds rows by delegating to the `findByPrimaryKey()` method on the entity definition corresponding to the first entity usage whose attributes in the view row key are non-null. The entity definition's `findByPrimaryKey()` method requires all key attributes for any given entity object to be non-null in order to find the entity row in the cache.

As a concrete example, imagine that you have a `OrderInfoVO` view object with a `OrderEO` entity object as its primary entity usage, and an `AddressEO` entity as secondary reference entity usage. Furthermore, assume that you leave the **Key Attribute** property of *both* of the following view row attributes set to `true`:

- `OrderId` — primary key for the `OrderEO` entity
- `AddressId` — primary key for the `AddressEO` entity

The view row key will therefore be the (`OrderId`, `AddressId`) combination. When you do a `findByKey()`, you can provide a `Key` object that provides:

- A completely specified key for the underlying `OrderEO` entity

  ```
  Key k = new Key(new Object[]{new Number(200), null});
  ```

- A completely specified key for the underlying `AddressEO` entity

  ```
  Key k = new Key(new Object[]{null, new Number(118)});
  ```

- A completely specified key for both entities

  ```
  Key k = new Key(new Object[]{new Number(200), new Number(118)});
  ```

When a valid partial key is specified, the `findByKey()` method can return multiple rows as a result, treating the missing entity usage attributes in the Key object as a wildcard.

42.1.8 Handling View Object Queries with Primary Keys Defined by Transient Attributes

If your programmatic view object query relies on transient attributes for it primary key, it is possible for the user to receive a null pointer exception when they scroll the UI out of the view object's cached rows. In this case, since the view object query is not generated from a database table, your view object implementation must override the `ViewObjectImpl` classes' `retrieveByKey()` method to return the rows (or return an empty array when no rows are found).

Overriding this method will allow ADF Business Components to execute `findByKey()` to first find the requested rows from the cache. When that fails (because the primary key is a transient attribute), ADF Business Components will execute your `retrieveByKey()` override to perform the operations you defined to retrieve the rows that match the key coming in. The default implementation of this method tries to issue a database query to get the row(s) from the database:

```
protected Row[] retrieveByKey(ViewRowSetImpl rs, Key key, int maxNumOfRows,
boolean skipWhere)
```

The method has these arguments:

`maxNumOfRows` is the `maxNumOfRows` you passed into the call to `findByKey()`. It may be 1 .. *n* or -1. *n* means that it's looking for *n* many rows whose key matches the one that got passed in. -1 means match all rows. Note that it is possible for the view object to have more than one row that matches the key when the key is a partial key and the view object is based on multiple entity objects.

`skipWhere` controls whether `findByKey()` should apply the same `WHERE` clause as the base view object. If the base view object has a `WHERE` clause `DEPTNO = 10`, if `skipWhere` is `false`, you're supposed to apply the same `WHERE` clause when looking for the row(s) from the backing store. If `skipWhere` is `true`, then don't bother with the `WHERE` clause from the base view object.

42.1.9 Creating Dynamic Attributes to Store UI State

You can add one or more dynamic attributes to a view object at runtime using the `addDynamicAttribute()` method. Dynamic attributes can hold any serializable object as their value. Typically, you will consider using dynamic attributes when writing generic framework extension code that requires storing some additional per-row transient state to implement a feature you want to add to the framework in a global, generic way.

42.1.10 Working with Multiple Row Sets and Row Set Iterators

While you typically work with a view object's *default* row set, you can call the `createRowSet()` method on the `ViewObject` interface to create secondary, named row sets based on the same view object's query. One situation where this could make sense is when your view object's SQL query contains named bind variables. Since each `RowSet` object stores its own copy of bind variable values, you could use a single view object to produce and process multiple row sets based on different combinations of bind variable values. You can find a named row set you've created using the `findRowSet()` method. When you're done using a secondary row set, call its `closeRowSet()` method.

For any `RowSet`, while you typically work with its *default* row set iterator, you can call the `createRowSetIterator()` method of the `RowSet` interface to create secondary, named row set iterators. You can find a named row set iterator you've created using the `findRowSetIterator()` method. When you're done using a secondary row set iterator, call its `closeRowSetIterator()` method.

> **Performance Tip:** When you need to perform programmatic iteration over a result set, create a secondary iterator to avoid disturbing the current row of the *default* row set iterator. For example, through the ADF Model declarative data binding layer, user interface pages in your application work with the default row set iterator of the default row set of view objects in the application module's data model. In this scenario, if you did not create a secondary row set iterator for the business logic you write to iterate over a view object's default row set, you would consequently change the current row of the *default* row set iterator used by the user interface layer.

42.1.11 Optimizing View Link Accessor Access By Retaining the Row Set

Each time you retrieve a view link accessor row set, by default the view object creates a new `RowSet` object to allow you to work with the rows. This does *not* imply re-*executing* the query to produce the results each time, only creating a new instance of

a `RowSet` object with its default iterator reset to the "slot" before the first row. To force the row set to refresh its rows from the database, you can call its `executeQuery()` method.

You can enable caching of the view link accessor row set when you do not want the application to incur the small amount of overhead associated with creating new detail row sets. For example, because view accessor row sets remain stable as long as the master row view accessor attribute remains unchanged, it would not be necessary to recreate a new row set for UI components, like the tree control, where data for each master node in a tree needs to retain its distinct set of detail rows. The view link accessor's detail row set can also be accessed programmatically. In this case, if your application makes numerous calls to the same view link accessor attributes, you can consider caching the view link accessor row set. This style of managing master-detail coordination differs from creating view link instances in the data model, as explained in Section 42.1.3, "Understanding View Link Accessors Versus Data Model View Link Instances."

You can enable retention of the view link accessor row set using the overview editor for the view object that is the source for the view link accessor. Select **Retain View Link Accessor Row Set** in the Tuning section of the General page of the overview editor for the view object.

Alternatively, you can enable a custom Java class for your view object, override the `create()` method, and add a line after `super.create()` that calls the `setViewLinkAccessorRetained()` method passing `true` as the parameter. It affects all view link accessor attributes for that view object.

When this feature is enabled for a view object, since the view link accessor row set is not recreated each time, the current row of its default row set iterator is also retained as a side-effect. This means that your code will need to explicitly call the `reset()` method on the row set you retrieve from the view link accessor to reset the current row in its default row set iterator back to the "slot" before the first row.

Note, however, that with accessor retention enabled, your failure to call `reset()` each time before you iterate through the rows in the accessor row set can result in a subtle, hard-to-detect error in your application. For example, if you iterate over the rows in a view link accessor row set like this, for example to calculate some aggregate total:

```
RowSet rs = (RowSet)row.getAttribute("OrdersShippedToPurchaser");
while (rs.hasNext()) {
  Row r = rs.next();
  // Do something important with attributes in each row
}
```

The first time you work with the accessor row set the code will work. However, since the row set (and its default row set iterator) are retained, the second and subsequent times you access the row set the current row will already be at the end of the row set and the while loop will be skipped since `rs.hasNext()` will be `false`. Instead, with this feature enabled, write your accessor iteration code like this:

```
RowSet rs = (RowSet)row.getAttribute("OrdersShippedToPurchaser");
rs.reset(); // Reset default row set iterator to slot before first row!
while (rs.hasNext()) {
  Row r = rs.next();
  // Do something important with attributes in each row
}
```

Recall that if view link consistency is on, when the accessor is retained the new unposted rows will show up at the end of the row set. This is slightly different from when the accessor is not retained (the default), where new unposted rows will appear at the beginning of the accessor row set.

42.2 Tuning Your View Objects for Best Performance

You can use view objects to read rows of data, create and store rows of transient data, as well as automatically coordinate inserts, updates, and deletes made by end users with your underlying business objects. How you design and use your view objects can definitely affect their performance at runtime. This section provides guidance on configuring your view objects to get the best possible performance.

42.2.1 Use Bind Variables for Parameterized Queries

Whenever the `WHERE` clause of your query includes values that might change from execution to execution, you should use named bind variables. The Create View Criteria dialog that you display from the Query page of the view object overview editor makes this an easy task. Their use also protects your application against abuse through SQL injection attacks by malicious end-users. For information about defining view criteria with bind variables, see Section 5.11.1, "How to Create Named View Criteria Declaratively."

42.2.1.1 Use Bind Variables to Avoid Re-parsing of Queries

Bind variables are place holders in the SQL string whose value you can easily change at runtime without altering the text of the SQL string itself. Since the query text doesn't change from execution to execution, the database can efficiently reuse the same parsed statement each time. Avoiding re-parsing of your statement alleviates the database from having to continually re-determine its query optimization plan and eliminates contention by multiple end-users on other costly database resources used during this parsing operation. This savings leads to higher runtime performance of your application. See Section 5.10.1, "How to Add Bind Variables to a View Object Definition" for details on how to use named bind variables.

42.2.1.2 Use Bind Variables to Prevent SQL-Injection Attacks

Using bind variables for parameterized `WHERE` clause values is especially important if their values will be supplied by *end-users* of your application. Consider the example shown in Example 42–3. It adds a dynamic `WHERE` clause formed by concatenating a user-supplied parameter value into the statement.

Example 42–3 Using String Concatenation Instead of Bind Variables is Vulnerable to SQL-Injection Attacks

```
// EXAMPLE OF BAD PRACTICE, Do not follow this approach!
String userSuppliedValue = ... ;
yourViewObject.setWhereClause("BANK_ACCOUNT_ID = "+userSuppliedValue);
```

A user with malicious intentions — if able to learn any details about your application's underlying database schema — could supply a carefully-constructed "bank account number" as a field value or URL parameter like:

```
BANK_ACCOUNT_ID
```

When the code in Example 42–3 concatenates this value into the dynamically-applied where clause, what the database sees is a query predicate like this:

```
WHERE (BANK_ACCOUNT_ID = BANK_ACCOUNT_ID)
```

This `WHERE` clause retrieves *all* bank accounts instead of just the current user's, perhaps allowing the hacker to view private information of another person's account. This technique of short-circuiting an application's `WHERE` clause by trying to supply a maliciously-constructed parameter value into a SQL statement is called a SQL injection

attack. Using named bind variables instead for these situations as shown in Example 42–4 prevents the vulnerability.

Example 42–4 Use Named Bind Variables Instead of String Concatenation

```
// Best practice using named bind variables
String userSuppliedValue = ... ;
yourViewObject.setWhereClause("BANK_ACCOUNT_ID = :BankAcccountId");
yourViewObject.defineNamedWhereClauseParam("BankAcccountId", null, null);
yourViewObject.setNamedWhereClauseParam("BankAcccountId",userSuppliedValue);
```

If a malicious user supplies an illegal value in this case, they receive an error your application can handle instead of obtaining data they are not suppose to see.

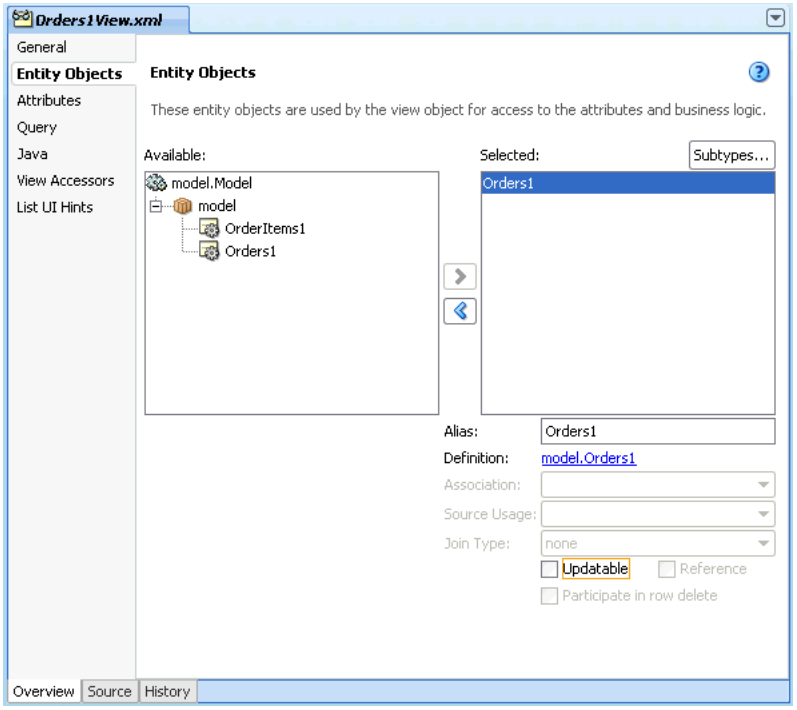
42.2.2 Consider Using Entity-Based View Objects for Read-Only Data

Typically view objects used for SQL-based validation purposes, as well as for displaying the list of valid selections in a dropdown list, can be read-only. You need to decide what kind of functionality your application requires and design the view object accordingly.

> **Best Practice:** When you need to create a read-only view object for data lookup, you should use the entity-based view object and deselect the **Updatable** option in the Entity Objects page of the view object overview editor. The approach benefits from the design time editors which aid in generating the SQL query. The alternative of creating an expert-mode view object requires writing a SQL query. Expert mode queries are still useful for cases where Unions and Group By queries cannot be expressed using entity objects.

View objects can either be related to underlying entity objects or not. When a view object is related to one or more underlying entity objects the default behavior supports creating new rows and modifying or removing queried rows. However, the update feature can be disabled by deselecting **Updatable** in the overview editor for the entity-based view object, as shown in Figure 42–4.

Figure 42–4 Deselecting the Updatable Option for an Entity-based View Object



The alternative is to create a read-only view object and define the SQL query using **Expert Mode** in the Edit Query dialog. For the Business Components developer not comfortable with constructing a complex SQL statement, it will always be more convenient to create a non-updatable view object based on an entity object since the editor simplifies the task of creating the query. Entity-based view objects that you set to non-updatable compare favorably to read-only, expert mode-based view objects:

- There is the ability to optimize the select list at runtime to include only those attributes that are required by the user interface
- There is no significant performance degradation incurred by using the entity object to create the local cache
- The data in the view object will reflect the state of the local cache rather than need to return to the database for each read operation
- The data in the local cache will stay consistent should another view object you define need to perform an update on the non-updatable view object's base entity object.

So, while there is a small amount of runtime overhead associated with the coordination between view object rows and entity object rows (estimates show less than 5% overhead), weigh this against the ability to keep the view object definition entirely declarative and maintain a customizable view object. Expert mode-based view objects are not customizable but they can be used to perform Unions and Group By queries that cannot be expressed in entity objects. Expert mode-based view objects are also useful in SQL-based validation queries used by the view object-based Key Exists validator.

When data is not read-only, the best (and only) choice is to create entity-based view objects. Entity-based view objects that are updatable (default behavior) are the only way to pickup entity-derived attribute default values, reflect pending changes made to relevant entity object attributes through other view objects in the same transaction,

and reflect updated reference information when foreign key attribute values are changed is to use an entity-based view object.

42.2.3 Use SQL Tracing to Identify Ill-Performing Queries

After deciding whether your view object should be mapped to entities or not, your attention should turn to the query itself. On the Query page of the view object overview editor, click the **Edit SQL Query** icon to display the Edit Query dialog. Click the **Explain Plan** button on the Query page of the Edit Query dialog to see the query plan that the database query optimizer will use. If you see that it is doing a full table scan, you should consider adding indexes or providing a value for the **Query Optimizer Hint** field on the Tuning section of the overview editor's General page. This will let you explicitly control which query plan will be used. These facilities provide some useful tools to the developer to evaluate the query plans for individual view object SQL statements. However, their use is not a substitute for tracing the SQL of the entire application to identify poorly performing queries in the presence of a production environment's amount of data and number of end users.

You can use the Oracle database's SQL Tracing facilities to produce a complete log of all SQL statements your application performs. The approach that works in all versions of the Oracle database is to issue the command:

```
ALTER SESSION SET SQL_TRACE TRUE
```

Specifically in version 10g of Oracle, the DBA would need to grant `ALTER SESSION` privilege in order to execute this command.

This command enables tracing of the current database session and logs all SQL statements to a server-side trace file until you either enter `ALTER SESSION SET SQL_TRACE FALSE` or close the connection. To simplify enabling this option to trace your Fusion web applications, override the `afterConnect()` method of your application module (or custom application module framework extension class) to conditionally perform the `ALTER SESSION` command to enable SQL tracing based on the presence of a Java system property as shown in Example 42–5.

Example 42–5 Conditionally Enabling SQL Tracing in an Application Module

```
// In YourCustomApplicationModuleImpl.java
protected void afterConnect() {
  super.afterConnect();
  if (System.getProperty("enableTrace") != null) {
    getDBTransaction().executeCommand("ALTER SESSION SET SQL_TRACE TRUE");
  }
}
```

After producing a trace file, you use the `TKPROF` utility supplied with the database to format the information and to better understand information about each query executed like:

- The number of times it was (re)parsed
- The number of times it was executed
- How many round-trips were made between application server and the database
- Various quantitative measurements of query execution time

Using these techniques, you can decide which additional indexes might be required to speed up particular queries your application performs, or which queries could be changed to improve their query optimization plan. For details about working with the

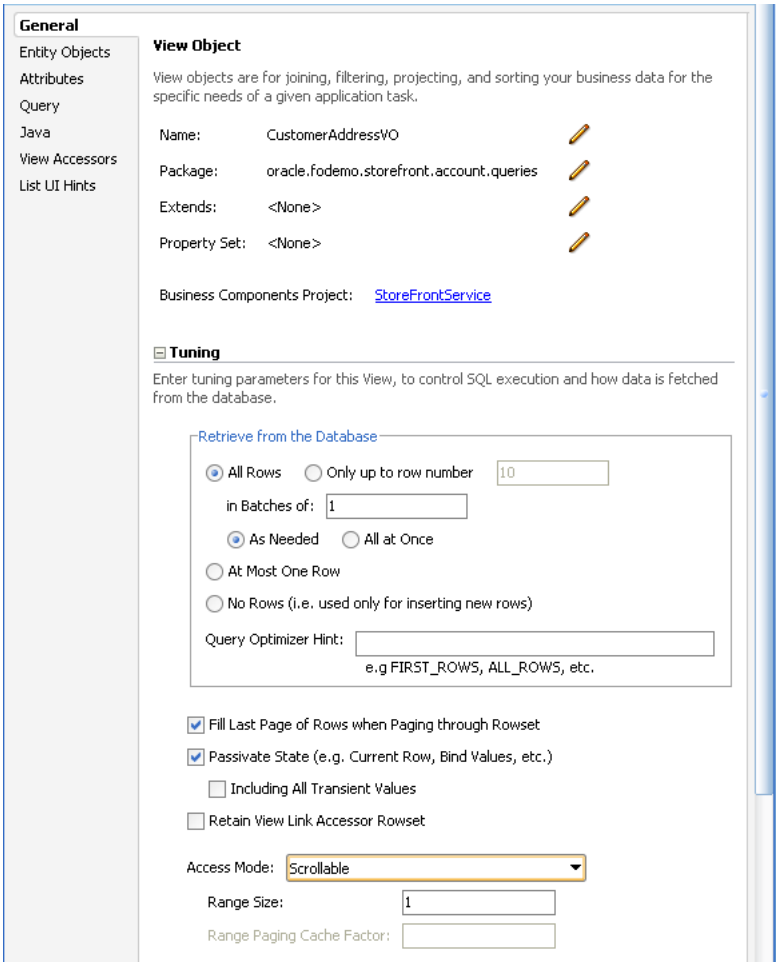
TKPROF utility, see sections "Understanding SQL Trace and TKPROF" and "Using the SQL Trace Facility and TKPROF" in the *Oracle Database Performance Tuning Guide*.

> **Note:** The Oracle database provides the DBMS_MONITOR package that further simplifies SQL tracing and integrates it with Oracle Enterprise Manager for visually monitoring the most frequently performed query statements your applications perform.

42.2.4 Consider the Appropriate Tuning Settings for Every View Object

The Tuning section on the General page of the view object overview editor lets you set various options that can dramatically effect your query's performance. Figure 42–5 shows the default options that the new view object defines.

Figure 42–5 View Object Default Tuning Options



42.2.4.1 Set the Database Retrieval Options Appropriately

The **Retrieve from the Database** group box, controls how the view object retrieves rows from the database server. The options for the fetch mode are **All Rows**, **Only Up To Row Number, At Most One Row**, and **No Rows**. Most view objects will stick with the default **All Rows** option, which will be retrieved **As Needed** (default) or **All at Once** depending on which option you choose.

> **Note:** The **All at Once** option does not enforce a single database round trip to fetch the rows specified by the view object query. The **As Needed** and **All at Once** options work in conjunction with the value of **in Batches of** (also known as fetch size) to determine the number of round trips. For best database access performance, you should consider changing the fetch size as described in Section 42.2.4.2, "Consider Whether Fetching One Row at a Time is Appropriate."

The **As Needed** option ensures that an `executeQuery()` operation on the view object initially retrieves only as many rows as necessary to fill the first page of a display, whose number of rows is set based on the view object's range size. If you use **As Needed**, then you will require only as many database round trips as necessary to deliver the number of rows specified by the initial range size. Whereas, if you use **All at Once**, then the application will perform as many round trips as necessary to deliver all the rows based on the value of **in Batches of** (fetch size) and the number of rows identified by the query.

For view objects whose `WHERE` clause expects to retrieve a *single* row, set the option to **At Most One Row** for best performance. This way, the view object knows you don't expect any more rows and will skip its normal test for that situation. Finally, if you use the view object only for creating new rows, set the option to **No Rows** so no query will ever be performed.

42.2.4.2 Consider Whether Fetching One Row at a Time is Appropriate

The fetch size controls how many rows will be returned in each round trip to the database. By default, the framework will fetch rows in batches of one row at a time. If you are fetching any more than one row, you will gain efficiency by setting this **in Batches of** value.

However the higher the number, the larger the client-side buffer required, so avoid setting this number arbitrarily high. If you are displaying results *n* rows at a time in the user interface, it's good to set the fetch size to at least *n*+1 so that each page of results can be retrieved in a single round trip to the database.

> **Caution:** Unless your query *really* fetches just one row, leaving the *default* fetch size of one (`1`) in the **in Batches of** field on the Tuning section of the General page of the view object overview editor is a recipe for bad performance due to many unnecessary round trips between the application server and the database. Oracle strongly recommends considering the appropriate value for each view object's fetch size.

42.2.4.3 Specify a Query Optimizer Hint if Necessary

The **Query Optimizer Hint** field allows you to specify an optional hint to the Oracle query optimizer to influence what execution plan it will use. You can set this hint in the Tuning page of the overview editor for the view object, as shown in Figure 42–5.

At runtime, the hint you provide is added immediately after the `SELECT` keyword in the query, wrapped by the special comment syntax `/*+ YOUR_HINT */`. Two common optimizer hints are:

- `FIRST_ROWS` — to hint that you want the first rows as quickly as possible
- `ALL_ROWS` — to hint that you want all rows as quickly as possible

There are many other optimizer hints that are beyond the scope of this manual to document. Reference the Oracle database reference manuals for more information on available hints.

42.2.5 Using Care When Creating View Objects at Runtime

It's important to understand the overhead associated with creating view objects at runtime. Avoid the temptation to do this without a compelling business requirement. For example, if your application issues a query against a table whose name you know at design time and if the list of columns to retrieve is also fixed, then create a view object at design time. When you do this, your SQL statements are neatly encapsulated, can be easily explained and tuned during development, and incur no runtime overhead to discover the structure and data types of the resulting rows.

In contrast, when you use the `createViewObjectFromQueryStmt()` API on the `ApplicationModule` interface at runtime, your query is buried in code, it's more complicated to proactively tune your SQL, and you pay a performance penalty each time the view object is created. Since the SQL query statement for a dynamically-created view object could theoretically be different each time a new instance is created using this API, an extra database round trip is required to discover the "shape" of the query results on-the-fly. Only create queries dynamically if you *cannot* know the name of the table to query until runtime. Most other needs can be addressed using a design-time created view object in combination with runtime API's to set bind variables in a fixed where clause, or to add an additional `WHERE` clause (with optional bind variables) at runtime.

42.2.6 Use Forward Only Mode to Avoid Caching View Rows

Often you will write code that programmatically iterates through the results of a view object. A typical situation will be custom validation code that must process multiple rows of query results to determine whether an attribute or an entity is valid or not. In these cases, if you intend to read each row in the row set a single time and never require scrolling backward or re-iterating the row set a subsequent time, then you can use "forward only" mode to avoid caching the retrieved rows. To enable forward only mode, call `setForwardOnly(true)` on the view object.

> **Note:** Using a read-only view object (with no entity usages) in forward-only mode with an appropriately tuned fetch size is the most efficient way to programmatically read data.

You can also use forward-only mode to avoid caching rows when inserting, updating, or deleting data as long as you never scroll backward through the row set and never call `reset()` to set the iterator back to the first row. Forward only mode only works with a range size of one (`1`).

42.3 Generating Custom Java Classes for a View Object

As you've seen, all of the basic querying functionality of a view object can be achieved without using custom Java code. Clients can retrieve and iterate through the data of any SQL query without resorting to any custom code on the view object developer's part. In short, for many read-only view objects, once you have defined the SQL statement, you're done. However, it's important to understand how to enable custom Java generation for a view object when your needs might require it. For example, reasons you might write code in a custom Java class include:

- To add validation methods (although Groovy Script expressions can provide this support without needing Java)
- To add custom logic
- To augment built-in behavior

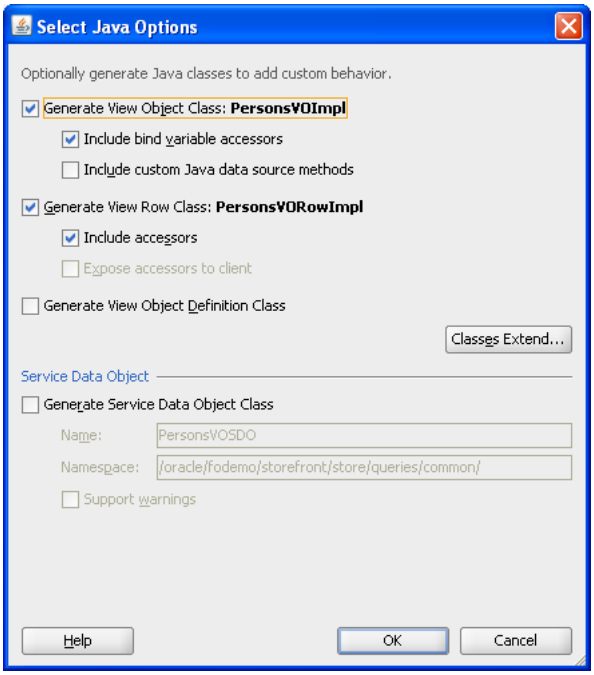
Appendix D, "Most Commonly Used ADF Business Components Methods" provides a quick reference to the most common code that you will typically write, use, and override in your custom view object and view row classes.

42.3.1 How To Generate Custom Classes

To enable the generation of custom Java classes for a view object, use the **Java** page of the view object overview editor. As shown in Figure 42–6, there are three optional Java classes that can be related to a view object. The first two in the list are the most commonly used:

- *View object class*, which represents the component that performs the query
- *View row class*, which represents each row in the query result

Figure 42–6 View Object Custom Java Generation Options



42.3.1.1 Generating Bind Variable Accessors

When you enable the generation of a custom view object class, if you also select the **Bind Variable Accessors** checkbox, then JDeveloper generates getter and setter methods in your view object class. Since the `Users` view object had three named bind variables (`TheName`, `LowUserId`, and `HighUserId`), the custom `PersonsImpl.java` view object class would have corresponding methods like this:

```
public String getTheName() {...}
public void setTheName(String value){...}
public Number getHighUserId(){...}
public void setHighUserId(Number value) {...}
public Number getLowUserId() {...}
```

```
public void setLowUserId(Number value) {...}
```

These methods allow you to set a bind variable with compile-time type-checking to ensure you are setting a value of the appropriate type. That is, instead of writing a line like this to set the value of the `LowUserId`:

```
vo.setNamedWhereClauseParam("LowUserId",new Number(150));
```

You can write the code like:

```
vo.setLowUserId(new Number(150));
```

You can see that with the latter approach, the Java compiler would catch a typographical error had you accidentally typed `setLowUserName` instead of `setLowUserId`:

```
// spelling name wrong gives compile error
vo.setLowUserName(new Number(150));
```

Or if you were to incorrectly pass a value of the wrong data type, like "ABC" instead of `Number` value:

```
// passing String where number expected gives compile error
vo.setLowUserId("ABC");
```

Without the generated bind variable accessors, an incorrect line of code like the following cannot be caught by the compiler:

```
// Both variable name and value wrong, but compiler cannot catch it
vo.setNamedWhereClauseParam("LowUserName","ABC");
```

It contains both an incorrectly spelled bind variable name, as well as a bind variable value of the wrong datatype. If you use the generic APIs on the `ViewObject` interface, errors of this sort will raise exceptions at runtime instead of being caught at compile time.

42.3.1.2 Generating View Row Attribute Accessors

When you enable the generation of a custom view row class, if you also select the **Accessors** checkbox, then JDeveloper generates getter and setter methods for each attribute in the view row. For example, for the `Persons` view object, the corresponding custom `PersonsRowImpl.java` class might have methods like this generated in it:

```
public Number getPersonId() {...}
public void setPersonId(Number value) {...}
public String getEmail() {...}
public void setEmail(String value) {...}
public String getFirstName() {...}
public void setFirstName(String value) {...}
public String getLastName() {...}
public void setLastName(String value) {...}
```

These methods allow you to work with the row data with compile-time checking of the correct datatype usage. That is, instead of writing a line like this one that gets the value of the `PersonId` attribute:

```
Number personId = (Number)row.getAttribute("PersonId");
```

you can write the code like:

```
Number personId = row.getPersonId();
```

You can see that with the latter approach, the Java compiler would catch a typographical error had you accidentally typed `PersonIdentifier` instead of `PersonId`:

```
// spelling name wrong gives compile error
Number personId = row.getPersonIdentifier();
```

Without the generated view row accessor methods, an incorrect line of code like the following cannot be caught by the compiler:

```
// Both attribute name and type cast are wrong, but compiler cannot catch it
String personId = (String)row.getAttribute("PersonIdentifier");
```

It contains both an incorrectly spelled attribute name, as well as an incorrectly-typed cast of the `getAttribute()` return value. Using the generic APIs on the `Row` interface, errors of this kind will raise exceptions at runtime instead of being caught at compile time.

42.3.1.3 Exposing View Row Accessors to Clients

When enabling the generation of a custom view row class, if you choose to generate the view row attribute accessor, you can also optionally select the **Expose Accessor to the Client** checkbox. This causes an additional custom row interface to be generated which application clients can use to access custom methods on the row without depending directly on the implementation class.

> **Best Practice:** When you create client code for business components, you should use business service interfaces rather than concrete classes. Using the interface instead of the implementation class, ensures that client code does not need to change when your server-side implementation does. For more details working with client code, see in Section 3.5.9, "Custom Interface Support for Client-Accessible Components."

For example, in the case of the `Persons` view object, exposing the accessors to the client will generate a custom row interface named `PersonsRow`. This interface is created in the `common` subpackage of the package in which the view object resides. Having the row interface allows clients to write code that accesses the attributes of query results in a strongly typed manner. Example 42–6 shows a `TestClient3` sample client program that casts the results of the `next()` method to the `PersonsRow` interface so that it can call accessors like `getPersonId()` and `getEmail()`.

Example 42–6 Simple Example of Using Client Row Interface with Accessors

```
package devguide.examples.readonlyvo.client;

import devguide.examples.readonlyvo.queries.common.PersonsRow;
import oracle.jbo.*;
import oracle.jbo.client.Configuration;
import oracle.jbo.domain.Number;

public class TestClient3 {
  public static void main(String[] args) {
    String amDef = "devguide.examples.PersonService";
    String config = "PersonServiceLocal";
    ApplicationModule am =
      Configuration.createRootApplicationModule(amDef, config);
    ViewObject vo = am.findViewObject("PersonList");
```

```
    vo.executeQuery();
    while (vo.hasNext()) {
     // Cast next() to a strongly-typed PersonsRow interface
      PersonsRow curPerson = (PersonsRow)vo.next();
      Number personId = curPerson.getPersonId();
      String email  = curPerson.getEmail();
      System.out.println(personId+ " " + email);
    }
    Configuration.releaseRootApplicationModule(am, true);
  }
}
```

42.3.1.4 Configuring Default Java Generation Preferences

You've seen how to generate custom Java classes for your view objects when you need to customize their runtime behavior, or if you simply prefer to have strongly typed access to bind variables or view row attributes.

To change the default settings that control how JDeveloper generates Java classes, choose **Tools | Preferences** and open the **Business Components** page. The settings you choose will apply to all future business components you create.

Oracle recommends that developers getting started with ADF Business Components set their preference to generate no custom Java classes by default. As you run into specific needs, you can enable just the bit of custom Java you need for that one component. Over time, you'll discover which set of defaults works best for you.

42.3.2 What Happens When You Generate Custom Classes

When you choose to generate one or more custom Java classes, JDeveloper creates the Java file(s) you've indicated.

For example, in the case of a view object named `devguide.examples.Persons`, the default names for its custom Java files will be `PersonsImpl.java` for the view object class and `PersonsRowImpl.java` for the view row class. Both files get created in the same `./devguide/examples` directory as the component's XML component definition file.

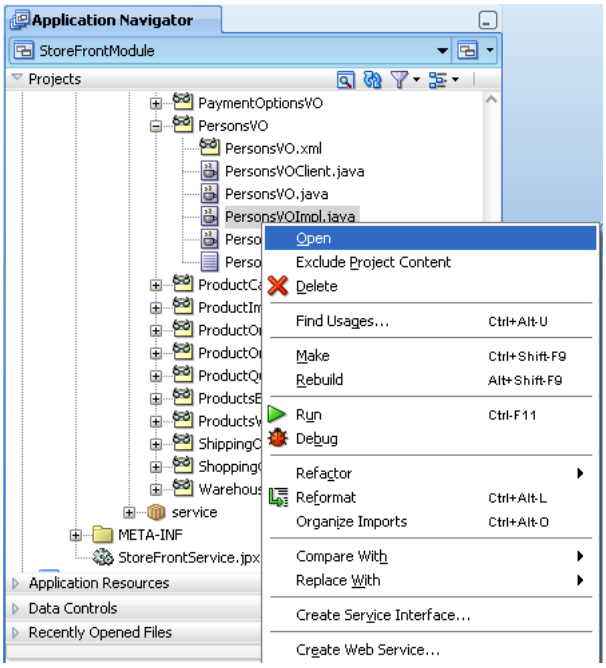
The Java generation options for the view object continue to be reflected on the Java page on subsequent visits to the view object overview editor. Just as with the XML definition file, JDeveloper keeps the generated code in your custom Java classes up to date with any changes you make in the editor. If later you decide you didn't require a custom Java file for any reason, unchecking the relevant options in the Java page causes the custom Java files to be removed.

42.3.2.1 Seeing and Navigating to Custom Java Files

As shown in Figure 42–7, when you've enabled generation of custom Java classes, they also appear under the node for the view object. When you need to see or work with the source code for a custom Java file, there are two ways to open the file in the source editor:

- Choose **Open** in the context menu as shown in Figure 42–7
- With the Java file node selected in the Application Navigator, double-click a node in the Structure window

Figure 42–7 Seeing and Navigating to Custom Java Classes for a View Object



42.3.3 What You May Need to Know About Custom Classes

This section provides additional information to help you use custom Java classes.

42.3.3.1 About the Framework Base Classes for a View Object

When you use an "XML-only" view object, at runtime its functionality is provided by the default ADF Business Components implementation classes. Each custom Java class that gets generated will automatically extend the appropriate ADF Business Components base class so that your code inherits the default behavior and can easily add or customize it. A view object class will extend `ViewObjectImpl`, while the view row class will extend `ViewRowImpl` (both in the `oracle.jbo.server` package).

42.3.3.2 You Can Safely Add Code to the Custom Component File

Based perhaps on previous negative experiences, some developers are hesitant to add their own code to generated Java source files. Each custom Java source code file that JDeveloper creates and maintains for you includes the following comment at the top of the file to clarify that it is safe to add your own custom code to this file:

```
// ---------------------------------------------------------------------
// ---    File generated by Oracle ADF Business Components Design Time.
// ---    Custom code may be added to this class.
// ---    Warning: Do not modify method signatures of generated methods.
// ---------------------------------------------------------------------
```

JDeveloper does not blindly regenerate the file when you click the **OK** or **Apply** button in the component dialogs. Instead, it performs a smart update to the methods that it needs to maintain, leaving your own custom code intact.

42.3.3.3 Attribute Indexes and InvokeAccessor Generated Code

The view object is designed to function either in an XML-only mode or using a combination of an XML component definition and a custom Java class. Since attribute

values are not stored in private member fields of a view row class, such a class is not present in the XML-only situation. Instead, attributes are defined as an `AttributesEnum` type, which specifies attribute names (and accessors for each attribute) based on the view object's XML component definition, in sequential order of the `<ViewAttribute>` tag, the association-related `<ViewLinkAccessor>` tag, and the `<ViewAccessor>` tag in that file. At runtime, the attribute values in an view row are stored in a structure that is managed by the base `ViewRowImpl` class, indexed by the attribute's numerical position in the view object's attribute list.

For the most part this private implementation detail is unimportant. However, when you enable a custom Java class for your view row, this implementation detail is related to some of the generated code that JDeveloper automatically maintains in your view row class, and you may want to understand what that code is used for. For example, in the custom Java class for the `Users` view row, Example 42–7 shows that each attribute, view link accessor attribute, or view accessor attribute has a corresponding generated `AttributesEnum` enum. JDeveloper defines enums instead of constants in order to prevent merge conflicts that could result when multiple developers add new attributes to the XML component definition.

Example 42–7 Attribute Constants Are Automatically Maintained in the Custom View Row Java Class

```
public class PersonsRowImpl extends ViewRowImpl implements PersonsRow {
/**
 * AttributesEnum: generated enum for identifying attributes and accessors. Do not
modify.
 */
public enum AttributesEnum {...}
  public static final int PERSONID = AttributesEnum.PersonId.index();
  public static final int EMAIL = AttributesEnum.Email.index();
  public static final int FIRSTNAME = AttributesEnum.FirstName.index();
  public static final int LASTNAME = AttributesEnum.LastName.index();
  public static final int PERSONTYPECODE = AttributesEnum.PersonTypeCode.index();
  public static final int PRIMARYADDRESSID = AttributesEnum.PrimaryAddressId.index
();
  // etc.
```

You'll also notice that the automatically maintained, strongly typed getter and setter methods in the view row class use these attribute constants like this:

```
public String getEmail() {
  return (String) getAttributeInternal(EMAIL); // <-- Attribute constant
}
public void setEmail(String value) {
  setAttributeInternal(EMAIL, value);// <-- Attribute constant
}
```

The last two aspects of the automatically maintained code related to view row attribute constants defined by the `AttributesEnum` type are the `getAttrInvokeAccessor()` and `setAttrInvokeAccessor()` methods. These methods optimize the performance of attribute access by numerical index, which is how generic code in the `ViewRowImpl` base class typically accesses attribute values. An example of the `getAttrInvokeAccessor()` method looks like the following from the `PersonsRowImpl.java` class. The companion `setAttrInvokeAccessor()` method looks similar.

```
protected Object getAttrInvokeAccessor(int index, AttributeDefImpl attrDef) throws Exception {
    if ((index >= AttributesEnum.firstIndex()) && (index < AttributesEnum.count())) {
        return AttributesEnum.staticValues()[index - AttributesEnum.firstIndex()].get(this);
    }
```

```
    return super.getAttrInvokeAccessor(index, attrDef);
}
```

The rules of thumb to remember about this generated attribute-related code are the following.

The Do's

- Add custom code if needed inside the strongly typed attribute getter and setter methods
- Use the view object overview editor to change the order or type of view object attributes

  JDeveloper will change the Java signature of getter and setter methods, as well as the related XML component definition for you.

The Don'ts

- Don't modify the list of enums in the generated `AttributesEnum` enum
- Don't modify the `getAttrInvokeAccessor()` and `setAttrInvokeAccessor()` methods

42.4 Working Programmatically with Multiple Named View Criteria

You can define *multiple* named view criteria in the overview editor for a view object and then selectively apply any combination of them to your view object at runtime as needed. For information about working with named view criteria at design time, see Section 5.11.1, "How to Create Named View Criteria Declaratively."

Note: The example in this section refers to the `MultipleViewCriteria` project in the `AdvancedViewObjectsExamples` application workspace in the `StandaloneExamples` module of the Fusion Order Demo application.

42.4.1 Applying One or More Named View Criteria

To apply one or more named view criteria, use the `setApplyViewCriteriaNames()` method. This method accepts a String array of the names of the criteria you want to apply. If you apply more than one named criteria, they are `AND`-ed together in the WHERE clause produced at runtime. New view criteria that you apply with the `setApplyViewCriteriaNames()` method will not overwrite view criteria that were previously applied.

When you need to apply more than one named view criteria, you can expose custom methods on the client interface of the view object to encapsulate applying combinations of the named view criteria. For example, Example 42–8 shows custom methods `showMaleCustomers()`, `showFemaleStaff()`, and `showFemaleCustomers()`, each of which uses the `setApplyViewCriteriaNames()` method to apply an appropriate combination of named view criteria. Once these methods are exposed on the view object's client interface, at runtime clients can invoke these methods as needed to change the information displayed by the view object.

Example 42–8 Exposing Client Methods to Enable Appropriate Named Criteria

```
// In PersonsViewImpl.java
```

```
public void showMaleCustomers() {
    ViewCriteriaManager vcm = getViewCriteriaManager();
    ViewCriteria vc_gender = vcm.getViewCriteria("GenderIsNotFCriteria");
    ViewCriteria vc_type = vcm.getViewCriteria("IsCustomerCriteria");
    VariableValueManager vvm_gender = vc_gender.ensureVariableManager();
    VariableValueManager vvm_type = vc_type.ensureVariableManager();
    vvm_gender.setVariableValue("bv_Gender","F");
    vvm_type.setVariableValue("bv_PersonTypeCode", "CUST");
    setApplyViewCriteriaNames(new String[]{"GenderIsNotFCriteria",
                                           "IsCustomerCriteria"});
}

public void showFemaleStaff() {
    ViewCriteriaManager vcm = getViewCriteriaManager();
    ViewCriteria vc_gender = vcm.getViewCriteria("GenderIsFCriteria");
    ViewCriteria vc_type = vcm.getViewCriteria("IsStaffSupplierCriteria");
    VariableValueManager vvm_gender = vc_gender.ensureVariableManager();
    VariableValueManager vvm_type = vc_type.ensureVariableManager();
    vvm_gender.setVariableValue("bv_Gender","F");
    vvm_type.setVariableValue("bv_PersonTypeCode", "CUST");
    setApplyViewCriteriaNames(new String[]{"GenderIsFCriteria",
                                           "IsStaffSupplierCriteria"});
    executeQuery();
}

public void showFemaleCustomers() {
    ViewCriteriaManager vcm = getViewCriteriaManager();
    ViewCriteria vc_gender = vcm.getViewCriteria("GenderIsFCriteria");
    ViewCriteria vc_type = vcm.getViewCriteria("IsCustomerCriteria");
    VariableValueManager vvm_gender = vc_gender.ensureVariableManager();
    VariableValueManager vvm_type = vc_type.ensureVariableManager();
    vvm_gender.setVariableValue("bv_Gender","F");
    vvm_type.setVariableValue("bv_PersonTypeCode", "CUST");
    setApplyViewCriteriaNames(new String[]{"GenderIsFCriteria",
                                           "IsCustomerCriteria"});
    executeQuery();
```

42.4.2 Removing All Applied Named View Criteria

To remove any currently applied named view criteria, use `setApplyViewCriteriaNames(null)`. For example, you could add the `showAll()` method in Example 42–9 to the `Users` view object and expose it on the client interface. This would allow clients to return to an unfiltered view of the data when needed.

Do not remove any design time view criteria because the row level bind variable values may already be applied on the row set. To help ensure this, named view criteria that get defined for a view accessor in the design time, will be applied as "required" view criteria on the view object instance so that it does not get removed by the view criteria's life cycle methods.

Example 42–9 Removing All Applied Named View Criteria

```
// In UsersImpl.java
public void showAll() {
  setApplyViewCriteriaNames(null);
  executeQuery();
}
```

Note: The `setApplyViewCriterias(null)` removes all applied view criteria, but allows you to later reapply any combination of them. In contrast, the `clearViewCriterias()` method *deletes* all named view criteria. After calling `clearViewCriterias()` you would have to use `putViewCriteria()` again to define new named criteria before you could apply them.

42.4.3 Using the Named Criteria at Runtime

At runtime, your application can invoke different client methods on a single view object interface to return different filtered sets of data. Example 42–10 shows the interesting lines of a `TestClient` class that works with the `Persons` view object described above. The `showResults()` method is a helper method that iterates over the rows in the view object to display some attributes.

Example 42–10 Test Client Code Working with Named View Criterias

```
// In TestClientMultipleViewCriterias.java
PersonsView vo = (PersonsView)am.findViewObject("PersonsView");
vo.showMaleCustomers();
showResults(vo,"After applying view criterias for male customers");
vo.applyViewCriteria(null);
vo.showFemaleStaff();
showResults(vo,"After applying view criterias for female staff");
vo.showFemaleCustomers();
showResults(vo,"After applying view criterias for female customers");
vo.showAll();
vo.showResults(vo,"After clearing all view criterias");
```

Running the `TestClient` program produces output as follows:

```
---After applying view criterias for male customers ---
Daniel Faviet [CUST, M]
John Chen [CUST, M]
Ismael Sciarra [CUST, M]
Jose Manuel Urman [CUST, M]
Luis Popp [CUST, M]
Den Raphaely [CUST, M]
Alexander Khoo [CUST, M]
Sigal Tobias [CUST, M]
Guy Himuro [CUST, M]
Matthew Weiss [CUST, M]
Adam Fripp [CUST, M]
Payam Kaufling [CUST, M]
Kevin Mourgos [CUST, M]
James Landry [CUST, M]
Steven Markle [CUST, M]
...
---After applying view criterias for female staff ---
Neena Kochhar [STAFF, F]
Valli Pataballa [STAFF, F]
Diana Lorentz [STAFF, F]
Terra Bralick [SUPP, F]
Rachel Berman [SUPP, F]
Claudia Benghiat [SUPP, F]
Sharon Hemant [SUPP, F]
Alison Chen [SUPP, F]
Alex Duckers [SUPP, F]
```

```
Katrina Han [SUPP, F]
---After applying view criterias for female customers ---
Nancy Greenberg [CUST, F]
Shelli Baida [CUST, F]
Karen Colmenares [CUST, F]
Shanta Vollman [CUST, F]
Julia Nayer [CUST, F]
Irene Mikkilineni [CUST, F]
Laura Bissot [CUST, F]
---After clearing all view criterias ---
Steven King [STAFF, M]
Neena Kochhar [STAFF, F]
Lex De Haan [STAFF, M]
Alexander Hunold [STAFF, M]
Bruce Ernst [STAFF, M]
David Austin [STAFF, M]
Valli Pataballa [STAFF, F]
Diana Lorentz [STAFF, F]
Nancy Greenberg [CUST, F]
...
```

42.5 Performing In-Memory Sorting and Filtering of Row Sets

By default a view object performs its query against the database to retrieve the rows in its resulting row set. However, you can also use view objects to perform in-memory searches and sorting to avoid unnecessary trips to the database.

Note: The example in this section refers to the `InMemoryOperations` project in the `AdvancedViewObjectsExamples` application workspace in the `StandaloneExamples` module of the Fusion Order Demo application.

42.5.1 Understanding the View Object's SQL Mode

The view object's SQL mode controls the source used to retrieve rows to populate its row set. The `setQueryMode()` allows you to control which mode, or combination of modes, are used:

- `ViewObject.QUERY_MODE_SCAN_DATABASE_TABLES`

  This is the default mode that retrieves results from the database.

- `ViewObject.QUERY_MODE_SCAN_VIEW_ROWS`

  This mode uses rows already in the row set as the source, allowing you to progressively refine the row set's contents through in-memory filtering.

- `ViewObject.QUERY_MODE_SCAN_ENTITY_ROWS`

  This mode, valid only for entity-based view objects, uses the entity rows presently in the entity cache as the source to produce results based on the contents of the cache.

You can use the modes individually, or combine them using Java's logical `OR` operator (*`X | Y`*). For example, to create a view object that queries the entity cache for unposted new entity rows, as well as the database for existing rows, you could write code like:

```
setQueryMode(ViewObject.QUERY_MODE_SCAN_DATABASE_TABLES |
             ViewObject.QUERY_MODE_SCAN_ENTITY_ROWS)
```

If you combine the SQL modes, the view object automatically handles skipping of duplicate rows. In addition, there is an implied order to the results that are found:

1. Scan view rows (if specified)
2. Scan entity cache (if specified)
3. Scan database tables (if specified) by issuing a SQL query

If you call the `setQueryMode()` method to change the SQL mode, your new setting takes effect the next time you call the `executeQuery()` method.

42.5.2 Sorting View Object Rows In Memory

To sort the rows in a view object at runtime, use the `setSortBy()` method. You pass a sort expression that looks like a SQL ORDER BY clause. However, instead of referencing the column names of the table, you use the view object's attribute names. For example, for a view object containing attributes named `Customer` and `DaysOpen`, you could sort the view object first by `Customer` descending, then by `DaysOpen` by calling:

```
setSortBy("Customer desc, DaysOpen");
```

Alternatively, you can use the *zero-based* attribute index position in the sorting clause like this:

```
setSortBy("3 desc, 2");
```

After calling the `setSortBy()` method, the rows will be sorted the next time you call the `executeQuery()` method. The view object translates this sorting clause into an appropriate format to use for ordering the rows depending on the SQL mode of the view object. If you use the default SQL mode, the `SortBy` clause is translated into an appropriate `ORDER BY` clause and used as part of the SQL statement sent to the database. If you use either of the in-memory SQL modes, then the `SortBy` by clause is translated into one or more `SortCriteria` objects for use in performing the in-memory sort.

Note: While SQL ORDER BY expressions treat column names in a case-insensitive way, the attribute names in a `SortBy` expression are case-*sensitive*.

42.5.2.1 Combining setSortBy and setQueryMode for In-Memory Sorting

You can perform an in-memory sort on the rows produced by a read-only view object using the `setSortBy()` and `setQueryMode()` methods. Example 42–11 shows the interesting lines of code from the `TestClientSetSortBy` class that uses `setSortBy()` and `setQueryMode()` to perform an in-memory sort on the rows produced by a read-only view object `ClosedOrders`.

Example 42–11 Combining setSortBy and setQueryMode for In-Memory Sorting

```
// In TestClientSetSortBy.java
am.getTransaction().executeCommand("ALTER SESSION SET SQL_TRACE TRUE");
ViewObject vo = am.findViewObject("ClosedOrders");
vo.executeQuery();
showRows(vo,"Initial database results");
```

```
vo.setSortBy("Customer desc");
vo.setQueryMode(ViewObject.QUERY_MODE_SCAN_VIEW_ROWS);
vo.executeQuery();
showRows(vo,"After in-memory sorting by Customer desc");
vo.setSortBy("Customer desc, DaysOpen");
vo.executeQuery();
showRows(vo,"After in-memory sorting by Customer desc, DaysOpen");
```

Running the example produces the results:

```
--- Initial database results ---
106,Ice machine not working,1,mhartste
103,Washing machine leaks,4,ngreenbe
105,Air in dryer not hot,4,jmurman
109,Freezer is not cold,4,jwhalen
:
--- After in-memory sorting by Customer desc ---
100,I have noticed that every time I do a...,9,dfaviet
101,Agitator does not work,8,sbaida
103,Washing machine leaks,4,ngreenbe
105,Air in dryer not hot,4,jmurman
:
--- After in-memory sorting by Customer desc, DaysOpen ---
100,I have noticed that every time I do a...,9,dfaviet
101,Agitator does not work,8,sbaida
105,Air in dryer not hot,4,jmurman
109,Freezer is not cold,4,jwhalen
:
```

The first line in Example 42–11 containing the `executeCommand()` call issues the `ALTER SESSION SET SQL TRACE` command to enable SQL tracing for the current database session. This causes the Oracle database to log every SQL statement performed to a server-side trace file. It records information about the text of each SQL statement, including how many times the database parsed the statement and how many round-trips the client made to fetch batches of rows while retrieving the query result.

> **Note:** You might need a DBA to grant permission to the `FOD` account to perform the `ALTER SESSION` command to do the tracing of SQL output.

Once you've produced a trace file, you can use the `TKPROF` utility that comes with the database to format the file:

```
tkprof xe_ora_3916.trc trace.prf
```

For details about working with the `TKPROF` utility, see sections "Understanding SQL Trace and TKPROF" and "Using the SQL Trace Facility and TKPROF" in the *Oracle Database Performance Tuning Guide*.

This will produces a `trace.prf` file containing the interesting information shown in Example 42–12 about the SQL statement performed by the `ClosedOrders` view object. You can see that after initially querying six rows of data in a single execute and fetch from the database, the two subsequent sorts of those results did not cause any further executions. Since the code set the SQL mode to `ViewObject.QUERY_MODE_SCAN_VIEW_ROWS` the `setSortBy()` followed by the `executeQuery()` performed the sort in memory.

Example 42–12 TKPROF Output of a Trace File Confirming Sort Was Done In Memory

```
********************************************************************
SELECT * FROM (select o.order_id,
       case
         when length(o.giftwrap_message) > 5 then
         rtrim(substr(o.giftwrap_message,1,5))||'...'
         else o.giftwrap_messagen
       end as giftwrap_message,
       ceil(
         (select trunc(max(creation_date))
            from order_histories
           where order_id = or.order_id)
         - trunc(o.order_date)
       ) as days_open,
       p.email as customer
from orders o, persons p
 where o.customer_id = p.person_id
   and order status code = 'COMPLETE')

call    count     cpu  elapsed disk  query  current     rows
------- -----  ------ -------- ---- ------ --------  -------
Parse       1    0.00     0.00    0      0        0        0
Execute     1    0.00     0.00    0      0        0        0
Fetch       1    0.00     0.00    0     22        0        6
------- -----  ------ -------- ---- ------ --------  -------
total       3    0.00     0.00    0     22        0        6
********************************************************************
```

42.5.2.2 Extensibility Points for In-Memory Sorting

Should you need to customize the way that rows are sorted in memory, you have the following two extensibility points:

1. You can override the method:

   ```
   public void sortRows(Row[] rows)
   ```

   This method performs the actual in-memory sorting of rows. By overriding this method you can plug in an alternative sorting approach if needed.

2. You can override the method:

   ```
   public Comparator getRowComparator()
   ```

   The default implementation of this method returns an `oracle.jbo.RowComparator`. `RowComparator` invokes the `compareTo()` method to compare two data values. These methods/objects can be overridden to provide custom compare routines.

42.5.3 Performing In-Memory Filtering with View Criteria

To filter the contents of a row set using `ViewCriteria`, you can call:

- `applyViewCriteria()` or `setApplyViewCriteriaNames()` followed by `executeQuery()` to produce a new, filtered row set.
- `findByViewCriteria()` to retrieve a new row set to process programmatically without changing the contents of the original row set.

Both of these approaches can be used against the database or to perform in-memory filtering, or both, depending on the view criteria mode. You set the criteria mode using

the `setCriteriaMode()` method on the `ViewCriteria` object, to which you can pass either of the following integer flags, or the logical `OR` of both:

- `ViewCriteria.CRITERIA_MODE_QUERY`
- `ViewCriteria.CRITERIA_MODE_CACHE`

When used for in-memory filtering with view criteria, the operators supported are shown in Table 42–1. You can group subexpressions with parenthesis and use the `AND` and `OR` operators between subexpressions.

Table 42–1 SQL Operators Supported By In-Memory Filtering with View Criteria

Operator	Operation
`=, >, <, <=, >=, <>, LIKE, BETWEEN`	Comparison
`NOT`	Logical negation
`AND`	Conjunction
`OR`	Disjunction

Example 42–13 shows the interesting lines from a `TestClientFindByViewCriteria` class that uses the two features described above both against the database and in-memory. It uses a `CustomerList` view object instance and performs the following basic steps:

1. Queries customers from the database with a last name starting with a 'C', producing the output:

```
--- Initial database results with applied view criteria ---
John Chen
Emerson Clabe
Karen Colmenares
```

2. Subsets the results from step 1 in memory to only those with a first name starting with 'J'. It does this by adding a second view criteria row to the view criteria and setting the conjunction to use "AND". This produces the output:

```
--- After augmenting view criteria and applying in-memory ---
John Chen
```

3. Sets the conjunction back to `OR` and re-applies the criteria to the database to query customers with last name like 'J%' or first name like 'C%'. This produces the output:

```
--- After changing view criteria and applying to database again ---
John Chen
Jose Manuel Urman
Emerson Clabe
Karen Colmenares
Jennifer Whalen
```

4. Defines a new criteria to find customers in-memory with first or last name that contain a letter 'o'
5. Uses `findByViewCriteria()` to produce new row set instead of subsetting, producing the output:

```
--- Rows returned from in-memory findByViewCriteria ---
John Chen
Jose Manuel Urman
Emerson Clabe
```

```
Karen Colmenares
```

6. Shows that original row set hasn't changed when `findByViewCriteria()` was used, producing the output:

```
--- Note findByViewCriteria didn't change rows in the view ---
John Chen
Jose Manuel Urman
Emerson Clabe
Karen Colmenares
Jennifer Whalen
```

Example 42–13 Performing Database and In-Memory Filtering with View Criteria

```
// In TestClientFindByViewCriteria.java
ViewObject vo = am.findViewObject("CustomerList");
// 1. Show customers with a last name starting with a 'M'
ViewCriteria vc = vo.createViewCriteria();
ViewCriteriaRow vcr1 = vc.createViewCriteriaRow();
vcr1.setAttribute("LastName","LIKE 'M%'");
vo.applyViewCriteria(vc);
vo.executeQuery();
vc.add(vcr1);
vo.executeQuery();
showRows(vo, "Initial database results with applied view criteria");
// 2. Subset results in memory to those with first name starting with 'S'
vo.setQueryMode(ViewObject.QUERY_MODE_SCAN_VIEW_ROWS);
ViewCriteriaRow vcr2 = vc.createViewCriteriaRow();
vcr2.setAttribute("FirstName","LIKE 'S%'");
vcr2.setConjunction(ViewCriteriaRow.VCROW_CONJ_AND);
vc.setCriteriaMode(ViewCriteria.CRITERIA_MODE_CACHE);
vc.add(vcr2);
vo.executeQuery();
showRows(vo,"After augmenting view criteria and applying in-memory");
// 3. Set conjuction back to OR and re-apply to database query to find
// customers with last name like 'H%' or first name like 'S%'
vc.setCriteriaMode(ViewCriteria.CRITERIA_MODE_QUERY);
vo.setQueryMode(ViewObject.QUERY_MODE_SCAN_DATABASE_TABLES);
vcr2.setConjunction(ViewCriteriaRow.VCROW_CONJ_OR);
vo.executeQuery();
showRows(vo,"After changing view criteria and applying to database again");
// 4. Define new critera to find customers with first or last name like '%o%'
ViewCriteria nameContainsO = vo.createViewCriteria();
ViewCriteriaRow lastContainsO = nameContainsO.createViewCriteriaRow();
lastContainsO.setAttribute("LastName","LIKE '%o%'");
ViewCriteriaRow firstContainsO = nameContainsO.createViewCriteriaRow();
firstContainsO.setAttribute("FirstName","LIKE '%o%'");
nameContainsO.add(firstContainsO);
nameContainsO.add(lastContainsO);
// 5. Use findByViewCriteria() to produce new rowset instead of subsetting
nameContainsO.setCriteriaMode(ViewCriteria.CRITERIA_MODE_CACHE);
RowSet rs = (RowSet)vo.findByViewCriteria(nameContainsO,
                        -1,ViewObject.QUERY_MODE_SCAN_VIEW_ROWS);
showRows(rs,"Rows returned from in-memory findByViewCriteria");
// 6. Show that original rowset hasn't changed
showRows(vo,"Note findByViewCriteria didn't change rows in the view");
```

42.5.4 Performing In-Memory Filtering with RowMatch

The `RowMatch` object provides an even more convenient way to express in-memory filtering conditions. You create a `RowMatch` object by passing a query predicate expression to the constructor like this:

```
RowMatch rm =
 new RowMatch("LastName = 'Popp' or (FirstName like 'A%' and LastName like 'K%')")
;
```

As you do with the `SortBy` clause, you phrase the `RowMatch` expression in terms of the view object attribute names, using the supported operators shown in Table 42–2. You can group subexpressions with parenthesis and use the `and` and `or` operators between subexpressions.

Table 42–2 SQL Operators Supported By In-Memory Filtering with RowMatch

Operator	Operation
`=, >, <, <=, >=, <>, LIKE, BETWEEN`	Comparison
`NOT`	Logical negation
`AND`	Conjunction
`OR`	Disjunction

You can also use a limited set of SQL functions in the `RowMatch` expression, as shown in Table 42–3.

Table 42–3 SQL Functions Supported By In-Memory Filtering with RowMatch

Operator	Operation
`UPPER`	Converts all letters in a string to uppercase.
`TO_CHAR`	Converts a number or date to a string.
`TO_DATE`	Converts a character string to a date format.
`TO_TIMESTAMP`	Converts a string to timestamp.

> **Note:** While SQL query predicates treat column names in a case-insensitive way, the attribute names in a `RowMatch` expression are case-*sensitive*.

42.5.4.1 Applying a RowMatch to a View Object

To apply a `RowMatch` to your view object, call the `setRowMatch()` method. In contrast to a `ViewCriteria`, the `RowMatch` is only used for *in-memory* filtering, so there is no "match mode" to set. You can use a `RowMatch` on view objects in any supported SQL mode, and you will see the results of applying it the next time you call the `executeQuery()` method.

When you apply a `RowMatch` to a view object, the `RowMatch` expression can reference the view object's named bind variables using the same `:VarName` notation that you would use in a SQL statement. For example, if a view object had a named bind variable named `StatusCode`, you could apply a RowMatch to it with an expression like:

```
Status = :StatusCode or :StatusCode = '%'
```

Example 42–14 shows the interesting lines of a `TestClientRowMatch` class that illustrate the `RowMatch` in action. The CustomerList view object used in the example has a transient `Boolean` attribute named `Selected`. The code performs the following basic steps:

1. Queries the full customer list, producing the output:

   ```
   --- Initial database results ---
   Neena Kochhar [null]
   Lex De Haan [null]
   Nancy Greenberg [null]
   :
   ```

2. Marks odd-numbered rows selected by setting the `Selected` attribute of odd rows to `Boolean.TRUE`, producing the output:

   ```
   --- After marking odd rows selected ---
   Neena Kochhar [null]
   Lex De Haan [true]
   Nancy Greenberg [null]
   Daniel Faviet [true]
   John Chen [null]
   Ismael Sciarra [true]
   :
   ```

3. Uses a `RowMatch` to subset the row set to contain only the select rows, that is, those with `Selected = true`. This produces the output:

   ```
   --- After in-memory filtering on only selected rows ---
   Lex De Haan [true]
   Daniel Faviet [true]
   Ismael Sciarra [true]
   Luis Popp [true]
   :
   ```

4. Further subsets the row set using a more complicated `RowMatch` expression, producing the output:

   ```
   --- After in-memory filtering with more complex expression ---
   Lex De Haan [true]
   Luis Popp [true]
   ```

Example 42–14 Performing In-Memory Filtering with RowMatch

```
// In TestClientRowMatch.java
// 1. Query the full customer list
ViewObject vo = am.findViewObject("CustomerList");
vo.executeQuery();
showRows(vo,"Initial database results");
// 2. Mark odd-numbered rows selected by setting Selected = Boolean.TRUE
markOddRowsAsSelected(vo);
showRows(vo,"After marking odd rows selected");
// 3. Use a RowMatch to subset row set to only those with Selected = true
RowMatch rm = new RowMatch("Selected = true");
vo.setRowMatch(rm);
// Note: Only need to set SQL mode when not defined at design time
vo.setQueryMode(ViewObject.QUERY_MODE_SCAN_VIEW_ROWS);
vo.executeQuery();
showRows(vo, "After in-memory filtering on only selected rows");
// 5. Further subset rowset using more complicated RowMatch expression
```

```
rm = new RowMatch("LastName = 'Popp' "+
                  "or (FirstName like 'A%' and LastName like 'K%')");
vo.setRowMatch(rm);
vo.executeQuery();
showRows(vo,"After in-memory filtering with more complex expression");
// 5. Remove RowMatch, set query mode back to database, requery to see full list
vo.setRowMatch(null);
vo.setQueryMode(ViewObject.QUERY_MODE_SCAN_DATABASE_TABLES);
vo.executeQuery();
showRows(vo,"After re-querying to see a full list again");
```

42.5.4.2 Using RowMatch to Test an Individual Row

In addition to using a `RowMatch` to filter a row set, you can also use its rowQualifies() method to test whether any individual row matches the criteria it encapsulates. For example:

```
RowMatch rowMatch = new RowMatch("CountryId = 'US'");
if (rowMatch.rowQualifies(row)) {
  System.out.println("Customer is from the United States ");
}
```

42.5.4.3 How a RowMatch Affects Rows Fetched from the Database

Once you apply a `RowMatch`, if the view object's SQL mode is set to retrieve rows from the database, when you call `executeQuery()` the `RowMatch` is applied to rows as they are fetched. If a fetched row does not qualify, it is not added to the rowset.

Unlike a SQL `WHERE` clause, a `RowMatch` can evaluate expressions involving transient view object attributes and not-yet-posted attribute values. This can be useful to filter queried rows based on `RowMatch` expressions involving transient view row attributes whose values are calculated in Java. This interesting aspect should be used with care, however, if your application needs to process a large rowset. Oracle recommends using database-level filtering to retrieve the smallest-possible rowset first, and then using RowMatch as appropriate to subset that list in memory.

42.6 Using View Objects to Work with Multiple Row Types

Sometimes you will create a view object to work with entity rows of a single type like `Supplier`, which perhaps includes `Supplier`-specific attributes. At other times you may want to query and update rows based on an entity object inheritance hierarchy in the same row set. For example, you might work, in the same row set, with attributes that are common to the inheritance hierarchy of `Persons`, `Supplier`, *and* `Staff` entity objects.

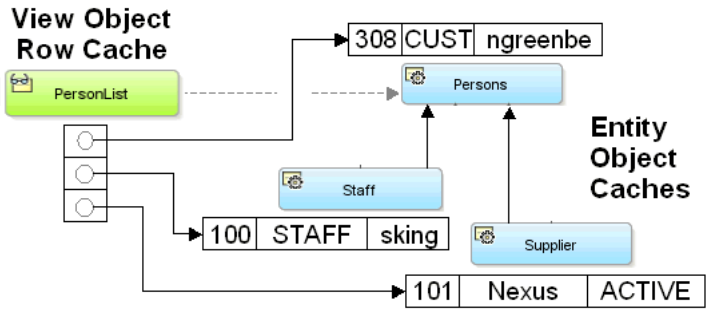
Note: To experiment with the example described in this section, use the same `InheritanceAndPolymorphicQueries` project in the `AdvancedEntityExamples` workspace used in Section 4.19, "Using Inheritance in Your Business Domain Layer."

42.6.1 Working with Polymorphic Entity Usages

A *polymorphic* entity usage is one that references a base entity object in an inheritance hierarchy and is configured to handle *subtypes* of that entity as well. Figure 42–8 shows the results of using a view object with a polymorphic entity usage. The entity-based `PersonList` view object has the `Person` entity object as its primary entity usage. The view object partitions each row retrieved from the database into an entity row part of

the appropriate entity object *subtype* of `Person`. It creates the appropriate entity row subtype based on consulting the value of the discriminator attribute. For example, if the `PersonList` query retrieves one row for person `ngreenbe`, one row for staff `sking`, and one row for supplier `ahunold`, the underlying entity row parts would be as shown in the figure.

Figure 42–8 View Object with a Polymorphic Entity Usage Handles Entity Subtypes



42.6.2 How to Create a View Object with a Polymorphic Entity Usage

The view object that you create with a polymorphic entity usage may inherit one or more of the attributes of the base entity object and the subtype entities. Attributes that you select from the entity objects will be overridden by the view object attribute definitions. When an entity-based view object references an entity object with a discriminator attribute, then JDeveloper enforces that the discriminator attribute is included in the query (in addition to the primary key attribute).

Before you begin:

Create the base entity object from which the polymorphic entity usages will inherit, as described in Section 4.2.2, "How to Create Single Entity Objects Using the Create Entity Wizard."

Extend the base entity object to create the polymorphic entity usages and specify the discriminator attribute upon which the entity row subtype will be based, as described in Section 4.10.14, "How to Set the Discriminator Attribute for Entity Object Inheritance Hierarchies."

Create an entity-based view object from the base entity object and select the attributes that are common to the view object you will create for the polymorphic entity usage, as described in Section 5.2.1, "How to Create an Entity-Based View Object."

To create a view object with a polymorphic entity usage:

1. In the Application Navigator, under the user interface project, right-click your application and choose **New**.
2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **View Object**, and click **OK**.
3. In the Create View Object wizard, name the view object and click **Browse**.

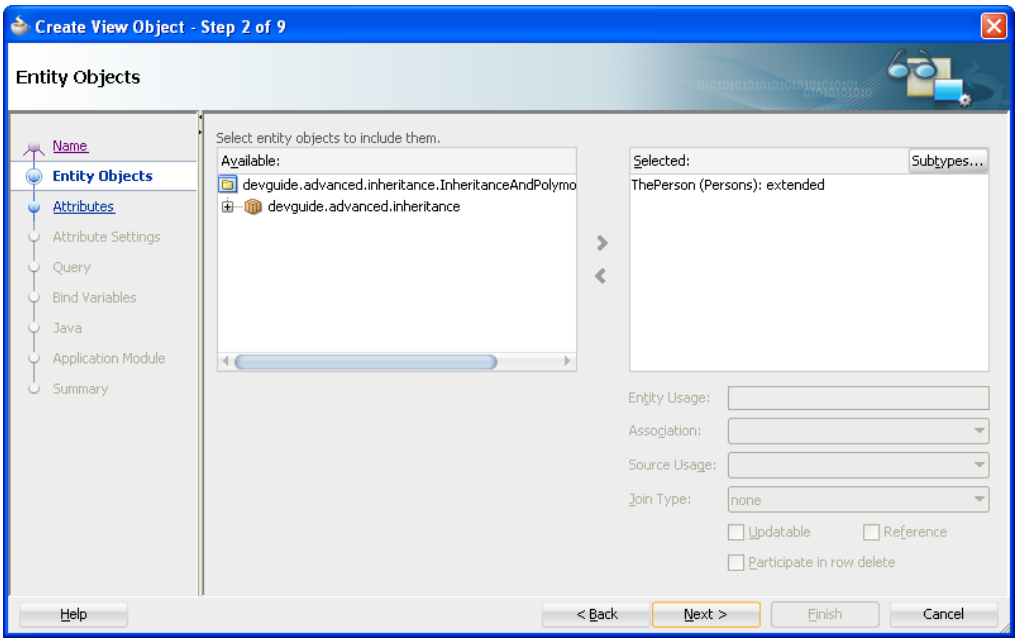
   For example, the data model project might define a base `Persons` entity object in order to support the creation of view objects with polymorphic entity usages for the `Supplier` and `Staff` subtype entity objects. When you create the view object for these polymorphic entity usages, you might create view objects named `SupplierList` or `StaffList`.

4. In the Select Parent dialog, select the entity-based view object that you created from the based entity object and click **OK**.

   For example, when you create the view object `PersonList` for the base entity object `Persons`, you would select `PersonList` as the view object to extend.

5. In the Create View Object wizard, click **Next** and note that the base entity object already appears in the **Selected** list and is labeled `Extended`, as shown in Figure 42–9.

Figure 42–9 View Object with a Base Entity Selection

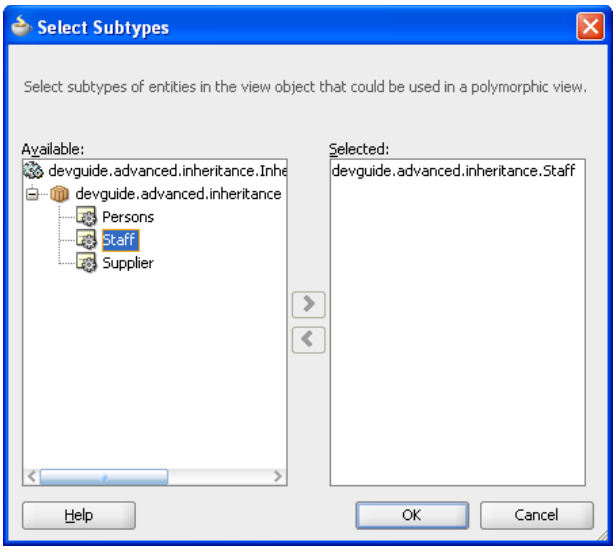


6. On the Entity Objects page of the Create View Object wizard, select the base entity object in the **Selected** list and click **Subtypes**.

7. In the **Select Subtypes** dialog, shuttle the desired entity subtypes you want to allow from the **Available** to the **Selected** list, and click **OK**.

   For example, for the `StaffList` view object you would select the entity subtype `Staff`, as shown in Figure 42–10.

Figure 42–10 View Object with a Entity Subtype Selection



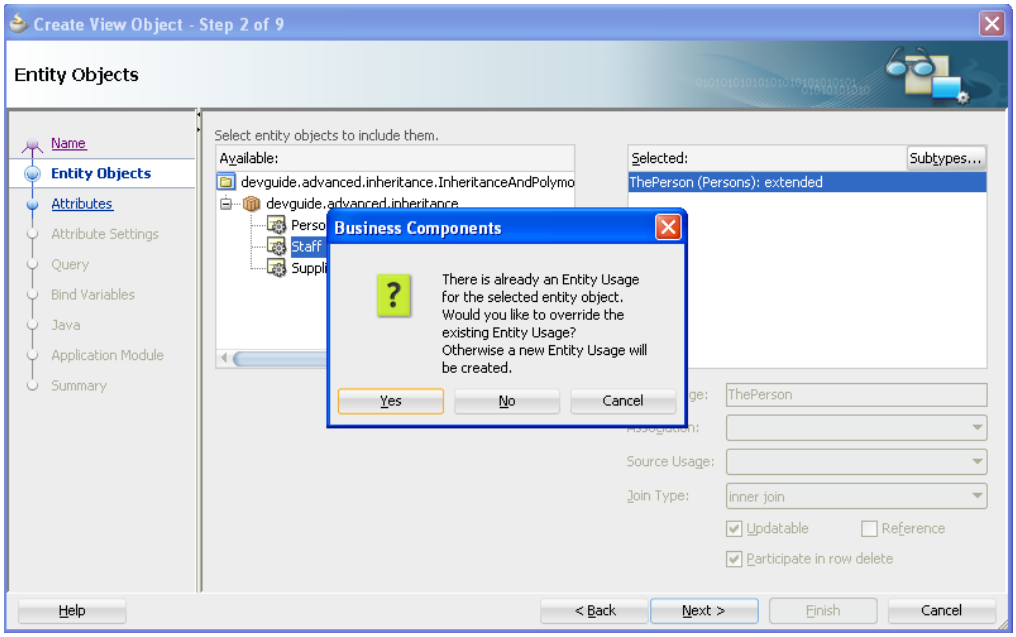
8. On the Entity Objects page of the Create View Object wizard, from the **Available** list, identify the entity object that represents the entity subtype that you want this view object to work with and add it to the **Selected** list.

   For example, in order to support the creation of `StaffList` view object with a polymorphic entity usage, select the `Staff` entity subtype.

9. In the Business Components dialog, click **OK** to override the entity usage for your view object.

   The Business Components dialog warns you that you will override the attributes of the base entity usage with the entity subtype, as shown in Figure 42–11.

Figure 42–11 View Object with a Polymorphic Entity Subtype Selection

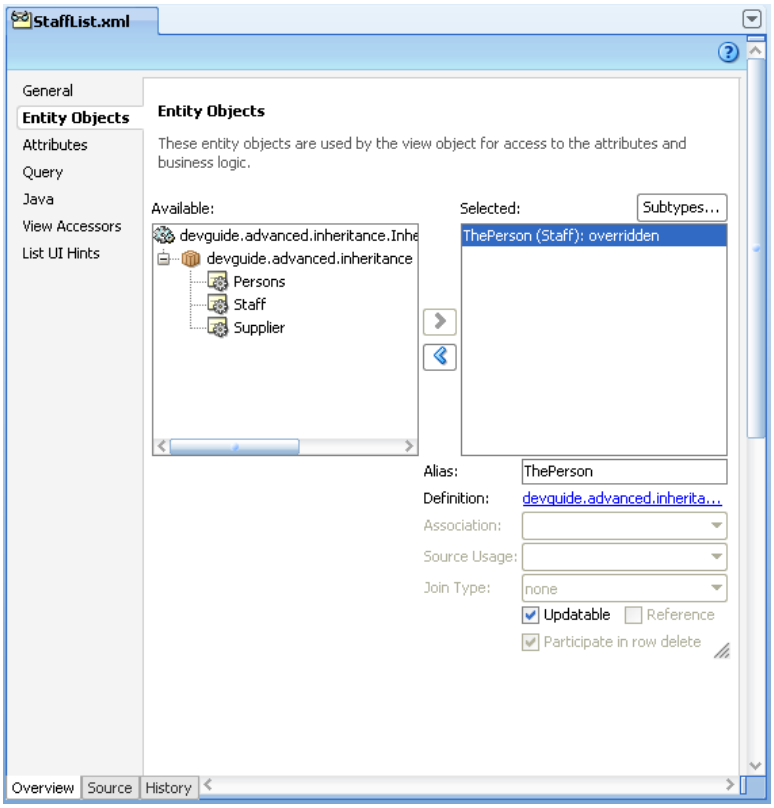


10. On the Entity Objects page of the Create View Object wizard, click **Next** and on the Attibutes page of the wizard shuttle desired attributes from the base entity object to the **Selected** list.

11. Complete the wizard and click **Finish**.

    The Entity Objects page of the overview editor identifies the selected entity object with the entity subtype override. For example, the overview editor for the `StaffList` view object identifies the overridden entity object `ThePerson (Staff): overridden` with the subtype in parenthesis, as shown in Figure 42–12.

Figure 42–12 View Object Editor Shows Entity Subtype is Overridden



12. Repeat this procedure to create view objects for additional polymorphic entity usages that you created for the base entity object.

42.6.3 What Happens When You Create a View Object with a Polymorphic Entity Usage

When you create an entity-based view object with a polymorphic entity usage, JDeveloper adds information about the allowed entity subtypes to the view object's XML component definition. For example, when creating the `PersonList` view object above, the names of the allowed subtype entity objects are recorded in an <AttrArray> tag like this:

```
<ViewObject Name="PersonList" ... >
   <EntityUsage Name="ThePerson"
                Entity="devguide.advanced.inheritance.Persons" >
   </EntityUsage>
...
   <AttrArray Name="EntityImports">
```

```
      <Item Value="devguide.advanced.inheritance.Staff" />
      <Item Value="devguide.advanced.inheritance.Supplier" />
   </AttrArray>
   <!-- etc. -->
</ViewObject>
```

42.6.4 What You May Need to Know About Entity Usages

This section provides additional information to help you work with polymorphic entity usages.

42.6.4.1 Your Query Must Limit Rows to Expected Entity Subtypes

If your view object expects to work with only a *subset* of the available entity subtypes in a hierarchy, you need to include an appropriate `WHERE` clause that limits the query to only return rows whose discriminator column matches the expected entity types.

42.6.4.2 Exposing Selected Entity Methods in View Rows Using Delegation

By design, clients do not work directly with entity objects. Instead, they work indirectly with entity objects through the view rows of an appropriate view object that presents a relevant set of information related to the task as hand. Just as a view object can expose a particular set of the underlying *attributes* of one or more entity objects related to the task at hand, it can also expose a selected set of *methods* from those entities. You accomplish this by enabling a custom view row Java class and writing a method in the view row class that:

- Accesses the appropriate underlying entity row using the generated entity accessor in the view row, and
- Invokes a method on it

For example, assume that the `Persons` entity object contains a `performPersonFeature()` method in its `PersonsImpl` class. To expose this method to clients on the `PersonsList` view row, you can enable a custom view row Java class and write the method shown in Example 42–15. JDeveloper generates an entity accessor method in the view row class for each participating entity usage based on the entity usage *alias* name. Since the alias for the `Persons` entity in the `PersonsList` view object is "`ThePerson`", it generates a `getThePerson()` method to return the entity row part related to that entity usage.

Example 42–15 Exposing Selected Entity Object Methods on View Rows Through Delegation

```
// In PersonListRowImpl.java
public void performPersonFeature() {
  getThePerson().performPersonFeature();
}
```

The code in the view row's `performPersonFeature()` method uses this `getThePerson()` method to access the underlying `PersonImpl` entity row class and then invokes *its* `performPersonFeature()` method. This style of coding is known as *delegation*, where a view row method delegates the implementation of one of *its* methods to a corresponding method on an underlying entity object. When delegation is used in a view row with a polymorphic entity usage, the delegated method call is handled by appropriate underlying entity row subtype. This means that if the `PersonsImpl`, `StaffImpl`, and `SupplierImpl` classes implement the `performPersonFeature()` method in a different way, the appropriate implementation is used depending on the entity subtype for the current row.

After exposing this method on the client row interface, client programs can use the custom row interface to invoke custom business functionality on a particular view row. Example 42–16 shows the interesting lines of code from a `TestEntityPolymorphism` class. It iterates over all the rows in the `PersonList` view object instance, casts each one to the custom `PersonListRow` interface, and invokes the `performPersonFeature()` method.

Example 42–16 Invoking a View Row Method That Delegates to an Entity Object

```
PersonList personlist = (PersonList)am.findViewObject("PersonList");
personlist.executeQuery();
while (personlist.hasNext()) {
  PersonListRow person = (PersonListRow)personlist.next();
  System.out.print(person.getEmail()+"->");
  person.performPersonFeature();
}
```

Running the client code in Example 42–16 produces the following output:

```
austin->## performPersonFeature as Supplier
hbaer->## performPersonFeature as Person
:
sking->## performPersonFeature as Staff
:
```

Rows related to `Persons` entities display a message confirming that the `performPersonFeature()` method in the `PersonsImpl` class was used. Rows related to `Supplier` and `Staff` entities display a different message, highlighting the different implementations that the respective `SupplierImpl` and StaffImpl classes have for the inherited `performPersonFeature()` method.

42.6.4.3 Creating New Rows With the Desired Entity Subtype

In a view object with a polymorphic entity usage, when you create a new view row it contains a new entity row part whose type matches the base entity usage. To create a new view row with one of the entity *subtypes* instead, use the `createAndInitRow()` method. Example 42–17 shows two custom methods in the `PersonList` view object's Java class that use `createAndInitRow()` to allow a client to create new rows having entity rows either of `Staff` or `Supplier` subtypes. To use the `createAndInitRow()`, as shown in the example, create an instance of the `NameValuePairs` object and set it to have an appropriate value for the discriminator attribute. Then, pass that `NameValuePairs` to the `createAndInitRow()` method to create a new view row with the appropriate entity row subtype, based on the value of the discriminator attribute you passed in.

Example 42–17 Exposing Custom Methods to Create New Rows with Entity Subtypes

```
// In PersonListImpl.java
public PersonListRow createStaffRow() {
  NameValuePairs nvp = new NameValuePairs();
  nvp.setAttribute("PersonTypeCode","STAFF");
  return (PersonListRow)createAndInitRow(nvp);
}
public PersonListRow createSupplierRow() {
  NameValuePairs nvp = new NameValuePairs();
  nvp.setAttribute("PersonTypeCode","SUPP");
  return (PersonListRow)createAndInitRow(nvp);
}
```

If you expose methods like this on the view object's custom interface, then at runtime, a client can call them to create new view rows with appropriate entity subtypes. Example 42–18 shows the interesting lines relevant to this functionality from a `TestEntityPolymorphism` class. First, it uses the `createRow()`, `createStaffRow()`, and `createSupplierRow()` methods to create three new view rows. Then, it invokes the `performPersonFeature()` method from the `PersonListRow` custom interface on each of the new rows.

As expected, each row handles the method in a way that is specific to the subtype of entity row related to it, producing the results:

```
## performPersonFeature as Person
## performPersonFeature as Staff
## performPersonFeature as Supplier
```

Example 42–18 Creating New View Rows with Different Entity Subtypes

```
// In TestEntityPolymorphism.java
PersonListRow newPerson = (PersonListRow)Personlist.createRow();
PersonListRow newStaff  = Personlist.createStaffRow();
PersonListRow newSupplier = Personlist.createSupplierRow();
newPerson.performPersonFeature();
newStaff.performPersonFeature();
newSupplier.performPersonFeature();
```

42.6.5 Working with Polymorphic View Rows

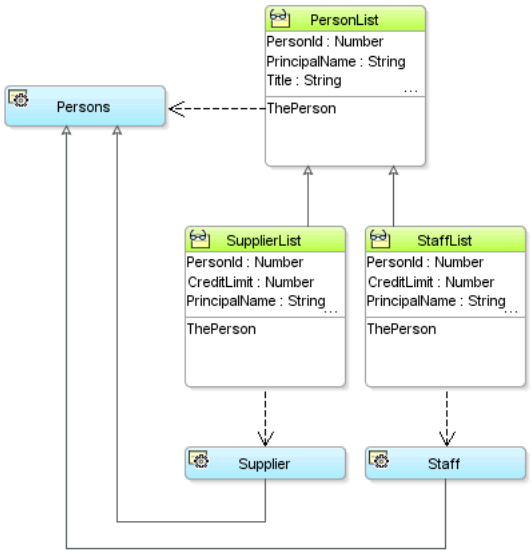
In the example shown in Section 42.6, "Using View Objects to Work with Multiple Row Types," the polymorphism occurs "behind the scenes" at the entity object level. Since the client code works with all view rows using the same `PersonListRow` interface, it cannot distinguish between rows based on a `Staff` entity object from those based on a `Persons` entity object. The code works with all view rows using the same set of view row attributes and methods common to all types of underlying entity subtypes.

If you configure a view object to support polymorphic view rows, then the client can work with different types of view rows using a view row interface specific to the type of row it is. By doing this, the client can access view attributes or invoke view row methods that are specific to a given subtype as needed. Figure 42–13 illustrates the hierarchy of view objects that enables this feature for the `PersonList` example considered above. `SupplierList` and `StaffList` are view objects that extend the base `PersonList` view object. Notice that each one includes an additional attribute specific to the subtype of `Person` they have as their entity usage. `SupplierList` includes an additional `ContractExpires` attribute, while `StaffList` includes the additional `DiscountEligible` attribute. When configured for view row polymorphism as described in the next section, a client can work with the results of the `PersonList` view object using:

- `PersonListRow` interface for view rows related to persons
- `SupplierListRow` interface for view rows related to suppliers
- `StaffListRow` interface for view rows related to staff

As you'll see, this allows the client to access the additional attributes and view row methods that are specific to a given subtype of view row.

Figure 42–13 Hierarchy of View Object Subtypes Enables View Row Polymorphism



42.6.6 How to Create a View Object with Polymorphic View Rows

To create a view object with polymorphic view rows, follow these steps:

1. In the Application Navigator, double-click the view object that you want to be the base.

   In the example above, the `PersonList` view object is the base.

2. In the overview editor, click the **Attributes** navigation tab and select a discriminator attribute for the view row, and then click the **Details** tab.

3. In the Details section, give the discriminator attribute a default value and check the **Polymorphic Discriminator** checkbox to mark the attribute as the one that distinguishes which view row interface to use.

   You must supply a value for the **Subtype Value** field that matches the attribute value for which you expect the base view object's view row interface to be used. For example, in the `PersonList` view object, you would mark the `PersonTypeCode` attribute as the discriminator attribute and supply a default subtype value of "`person`".

4. Enable a custom view row class for the base view object, and expose at least one method on the client row interface. This can be one or all of the view row attribute accessor methods, as well as any custom view row methods.

5. Create a new view object that *extends* the base view object

   In the example above, `SupplierList` extends the base `PersonList` view object.

6. Enable a custom view row class for the extended view object.

   If appropriate, add additional custom view row methods or override custom view row methods inherited from the parent view object's row class.

7. Supply a *distinct* value for the discriminator attribute in the extended view object.

   The `SupplierList` view object provides the value of "`SUPP`" for the `PersonTypeCode` discriminator attribute.

8. Repeat steps 4-6 to add additional extended view objects as needed.

For example, the `StaffList` view object is a second one that extends `PersonList`. It supplies the value "`STAFF`" for the `PersonTypeCode` discriminator attribute.

After setting up the view object hierarchy, you need to define the list of view object subtypes that participate in the view row polymorphism. To accomplish this, do the following:

1. Add an instance of each type of view object in the hierarchy to the data model of an application module.

   For example, the `PersonModule` application module in the example has instances of `PersonList`, `SupplierList`, and `StaffList` view objects.

2. In the overview editor for the application module, click the **Data Model** navigation tab and click the **Subtypes** button.

3. In the **Subtypes** dialog that appears, shuttle the desired view object subtypes that you want to participate in view row polymorphism from the **Available** to the **Selected** list, and click **OK**

42.6.7 What You May Need to Know About Polymorphic View Rows

This section provides additional information to help you work with polymorphic view rows.

42.6.7.1 Selecting Subtype-Specific Attributes in Extended View Objects

When you create an extended view object, it inherits the entity usage of its parent. If the parent view object's entity usage is based on an entity object with subtypes in your domain layer, you may want your extended view object to work with one of these subtypes instead of the inherited parent entity usage type. Two reasons you might want to do this are:

1. To select attributes that are *specific* to the entity subtype

2. To be able to write view row methods that delegate to methods *specific* to the entity subtype

In order to do this, you need to override the inherited entity usage to refer to the desired entity subtype. To do this, perform these steps in the overview editor for your extended view object:

1. In the Application Navigator, double-click the view object.

2. In the overview editor, click the **Entity Objects** navigation tab and verify that you are working with an extended entity usage.

   For example, when creating the `SupplierList` view object that extends the `PersonList` view object, the entity usage with the alias `ThePerson` will initially display in the **Selected** list as: **ThePerson(Person): extended**. The type of the entity usage is in parenthesis, and the "extended" label confirms that the entity usage is currently inherited from its parent.

3. Select the desired entity *subtype* in the **Available** list that you want to override the inherited one. It must be a subtype entity of the existing entity usage's type.

   For example, you would select the `Supplier` entity object in the **Available** list to overridden the inherited entity usage based on the `Persons` entity type.

4. Click **>** to shuttle it to the **Selected** list

5. Acknowledge the alert that appears, confirming that you want to override the existing, inherited entity usage.

When you have performed these steps, the **Selected** list updates to reflect the overridden entity usage. For example, for the `SupplierList` view object, after overriding the `Persons`-based entity usage with the `Supplier` entity subtype, it updates to show: **ThePerson (Supplier): overridden**.

After overriding the entity usage to be related to an entity subtype, you can then use the **Attributes** tab of the editor to select additional attributes that are specific to the subtype. For example, the `SupplierList` view object includes the additional attribute named `ContractExpires` that is specific to the `Supplier` entity object.

42.6.7.2 Delegating to Subtype-Specific Methods After Overriding the Entity Usage

After overriding the entity usage in an extended view object to reference a subtype entity, you can write view row methods that delegate to methods specific to the subtype entity class. Example 42–19 shows the code for a `performSupplierFeature()` method in the custom view row class for the `SupplierList` view object. It casts the return value from the `getThePerson()` entity row accessor to the subtype `SupplierImpl`, and then invokes the `performSupplierFeature()` method that is specific to `Supplier` entity objects.

Example 42–19 View Row Method Delegating to Method in Subtype Entity

```
// In SupplierListRowImpl.java
public void performSupplierFeature() {
   SupplierImpl supplier = (SupplierImpl)getThePerson();
   supplier.performSupplierFeature();
}
```

> **Note:** You need to perform the explicit cast to the entity subtype here because JDeveloper does not yet take advantage of the JDK feature called covariant return types that would allow a subclass like `SupplierListRowImpl` to override a method like `getThePerson()` and change its return type.

42.6.7.3 Working with Different View Row Interface Types in Client Code

Example 42–20 shows the interesting lines of code from a `TestViewRowPolymorphism` class that performs the following steps:

1. Iterates over the rows in the `PersonList` view object.

   For each row in the loop, it uses Java's `instanceof` operator to test whether the current row is an instance of the `StaffListRow` or the `SupplierListRow`.

2. If the row is a `StaffListRow`, then cast it to this more specific type and:
   - Call the `performStaffFeature()` method specific to the `StaffListRow` interface, and
   - Access the value of the `DiscountEligible` attribute that is specific to the `StaffList` view object.

3. If the row is a `SupplierListRow`, then cast it to this more specific type and:
   - Call the `performSupplierFeature()` method specific to the `SupplierListRow` interface, and

- Access the value of the `ContractExpires` attribute that is specific to the `SupplierList` view object.

4. Otherwise, just call a method on the `PersonListRow`

Example 42–20 Using View Row Polymorphism in Client Code

```
// In TestViewRowPolymorphism.java
ViewObject vo = am.findViewObject("PersonList");
vo.executeQuery();
// 1. Iterate over the rows in the PersonList view object
while (vo.hasNext()) {
  PersonListRow Person = (PersonListRow)vo.next();
  System.out.print(Person.getEmail()+"->");
  if (Person instanceof StaffListRow) {
    // 2. If the row is a StaffListRow, cast it
    StaffListRow mgr = (StaffListRow)Person;
    mgr.performStaffFeature();
    System.out.println("Discount Status: "+staff.getDiscountEligible());
  }
  else if (Person instanceof SupplieristRow) {
    // 3. If the row is a StaffListRow, cast it
    SupplierListRow tech = (SupplierListRow)Person;
    supplier.performSupplierFeature();
    System.out.println("Contract expires: "+tech.getContractExpires());
  }
  else {
    // 4. Otherwise, just call a method on the PersonListRow
    Person.performPersonFeature();
  }
}
```

Running the code in Example 42–20 produces the following output:

```
daustin->## performSupplierFeature called
Contract expires: 2006-05-09
hbaer->## performPersonFeature as Person
:
sking->## performStaffFeature called
Discount Status: Y
:
```

This illustrates that by using the view row polymorphism feature the client was able to distinguish between view rows of different types and access methods and attributes specific to each subtype of view row.

42.6.7.4 View Row Polymorphism and Polymorphic Entity Usage are Orthogonal

While often even more useful when used *together*, the view row polymorphism and the polymorphic entity usage features are distinct and can be used separately. In particular, the view row polymorphism feature can be used for read-only view objects, as well as for entity-based view objects. When you combine both mechanisms, you can have both the entity row part being polymorphic, as well as the view row type.

Note to use view row polymorphism with either view objects or entity objects, you must configure the discriminator attribute property separately for each. This is necessary because read-only view objects contain no related entity usages from which to infer the discriminator information.

In summary, to use view row polymorphism:

1. Configure an attribute to be the discriminator at the view object level in the root view object in an inheritance hierarchy.
2. Have a hierarchy of inherited view objects each of which provides a distinct value for the Subtype Value property of that view object level discriminator attribute (identified as `DefaultValue` for the attribute in the view object definition file).
3. List the subclassed view objects in this hierarchy in the application module's list of Subtypes.

Whereas, to create a view object with a polymorphic entity usage:

1. Configure an attribute to be the discriminator at the entity object level in the root entity object in an inheritance hierarchy.
2. Have a hierarchy of inherited entity objects, each of which overrides and provides a distinct value for the Subtype Value property of that entity object level discriminator attribute.
3. List the subclassed entity objects in a view object's list of Subtypes.

42.7 Reading and Writing XML

The Extensible Markup Language (XML) standard from the Worldwide Web Consortium (W3C) defines a language-neutral approach for electronic data exchange. Its rigorous set of rules enables the structure inherent in data to be easily encoded and unambiguously interpreted using human-readable text documents.

View objects support the ability to write these XML documents based on their queried data. View objects also support the ability to read XML documents in order to apply changes to data including inserts, updates, and deletes. When you've introduced view links, this XML capability supports reading and writing multi-level nested information for master-detail hierarchies of any complexity. While the XML produced and consumed by view objects follows a canonical format, you can combine the view object's XML features with XML Stylesheet Language Transformations (XSLT) to easily convert between this canonical XML format and any format you need to work with.

Note: The example in this section refers to the `ReadingAndWritingXML` project in the `AdvancedViewObjectsExamples` application workspace in the `StandaloneExamples` module of the Fusion Order Demo application.

42.7.1 How to Produce XML for Queried Data

To produce XML from a view object, use the `writeXML()` method. If offers two ways to control the XML produced:

1. For precise control over the XML produced, you can specify a view object attribute map indicating which attributes should appear, including which view link accessor attributes should be accessed for nested, detail information:

   ```
   Node writeXML(long options, HashMap voAttrMap)
   ```

2. To producing XML that includes all attributes, you can simply specify a depth-level that indicates how many levels of view link accessor attributes should be traversed to produce the result:

   ```
   Node writeXML(int depthCount, long options)
   ```

The `options` parameter is a integer flag field that can be set to one of the following bit flags:

- `XMLInterface.XML_OPT_ALL_ROWS`

  Includes all rows in the view object's row set in the XML.

- `XMLInterface.XML_OPT_LIMIT_RANGE`

  Includes only the rows in the current range in the XML.

Using the logical OR operation, you can combine either of the above flags with the `XMLInterface.XML_OPT_ASSOC_CONSISTENT` flag when you want to include new, unposted rows in the current transaction in the XML output.

Both versions of the `writeXML()` method accept an optional third argument which is an XSLT stylesheet that, if supplied, is used to transform the XML output before returning it.

42.7.2 What Happens When You Produce XML

When you produce XML using `writeXML()`, the view object begins by creating a wrapping XML element whose default name matches the name of the view object definition. For example, for a `Persons` view object in the `devguide.advanced.xml.queries` package, the XML produces will be wrapped in an outermost <Persons> tag.

Then, it converts the attribute data for the appropriate rows into XML elements. By default, each row's data is wrapped in an row element whose name is the name of the view object with the `Row` suffix. For example, each row of data from a view object named `Persons` is wrapped in an <PersonsRow> element. The elements representing the attribute data for each row appear as nested children inside this row element.

If any of the attributes is a view link accessor attribute, and if the parameters passed to `writeXML()` enable it, the view object will include the data for the detail rowset returned by the view link accessor. This nested data is wrapped by an element whose name is determined by the name of the view link accessor attribute. The return value of the `writeXML()` method is an object that implements the standard W3C `Node` interface, representing the root element of the generated XML.

> **Note:** The `writeXML()` method uses view link accessor attributes to programmatically access detail collections. It does not require adding view link instances in the data model.

For example, to produce an XML element for all rows of a `Persons` view object instance, and following view link accessors as many levels deep as exists, Example 42–21 shows the code required.

Example 42–21 Generating XML for All Rows of a View Object to All View Link Levels

```
ViewObject vo = am.findViewObject("PersonsView");
printXML(vo.writeXML(-1,XMLInterface.XML_OPT_ALL_ROWS));
```

The `Persons` view object is linked to a `Orders` view object showing the orders created by that person. In turn, the `Orders` view object is linked to a `OrderItems` view object providing details on the items ordered by customers. Running the code in Example 42–21 produces the XML shown in Example 42–22, reflecting the nested structure defined by the view links.

Example 42–22 XML from a Persons View Object with Two Levels of View Linked Details

```
...
<PersonsViewRow>
     <PersonId>111</PersonId>
     <PrincipalName>ISCIARRA</PrincipalName>
     <FirstName>Ismael</FirstName>
     <LastName>Sciarra</LastName>
     <PersonTypeCode>CUST</PersonTypeCode>
     <ProvisionedFlag>N</ProvisionedFlag>
     <PrimaryAddressId>42</PrimaryAddressId>
     <MembershipId>2</MembershipId>
     <Email>ISCIARRA</Email>
     <ConfirmedEmail>ISCIARRA</ConfirmedEmail>
     <PhoneNumber>228.555.0126</PhoneNumber>
     <DateOfBirth>1971-09-30</DateOfBirth>
     <MaritalStatusCode>SING</MaritalStatusCode>
     <Gender>M</Gender>
     <ContactableFlag>Y</ContactableFlag>
     <ContactByAffilliatesFlag>Y</ContactByAffilliatesFlag>
     <CreatedBy>SEED_DATA</CreatedBy>
     <CreationDate>2008-08-15 11:26:36.0</CreationDate>
     <LastUpdatedBy>SEED_DATA</LastUpdatedBy>
     <LastUpdateDate>2008-08-15 11:26:36.0</LastUpdateDate>
     <ObjectVersionId>1</ObjectVersionId>
     <OrdersView>
        <OrdersViewRow>
           <OrderId>1017</OrderId>
           <OrderDate>2008-08-06 11:28:26.0</OrderDate>
           <OrderStatusCode>STOCK</OrderStatusCode>
           <OrderTotal>1649.92</OrderTotal>
           <CustomerId>111</CustomerId>
           <ShipToAddressId>8</ShipToAddressId>
           <ShippingOptionId>2</ShippingOptionId>
           <PaymentOptionId>1006</PaymentOptionId>
           <DiscountId>3</DiscountId>
           <FreeShippingFlag>Y</FreeShippingFlag>
           <CustomerCollectFlag>Y</CustomerCollectFlag>
           <CollectionWarehouseId>102</CollectionWarehouseId>
           <GiftwrapFlag>N</GiftwrapFlag>
           <CreatedBy>0</CreatedBy>
           <CreationDate>2008-08-15 11:28:26.0</CreationDate>
           <LastUpdatedBy>0</LastUpdatedBy>
           <LastUpdateDate>2008-08-15 11:28:26.0</LastUpdateDate>
           <ObjectVersionId>0</ObjectVersionId>
           <OrderItemsView>
              <OrderItemsViewRow>
                 <OrderId>1017</OrderId>
                 <LineItemId>1</LineItemId>
                 <ProductId>22</ProductId>
                 <Quantity>1</Quantity>
                 <UnitPrice>199.95</UnitPrice>
                 <CreatedBy>0</CreatedBy>
                 <CreationDate>2008-08-15 11:32:26.0</CreationDate>
                 <LastUpdatedBy>0</LastUpdatedBy>
                 <LastUpdateDate>2008-08-15 11:32:26.0</LastUpdateDate>
                 <ObjectVersionId>0</ObjectVersionId>
              </OrderItemsViewRow>
              <OrderItemsViewRow>
                 <OrderId>1017</OrderId>
                 <LineItemId>2</LineItemId>
```

```
                  <ProductId>9</ProductId>
                  <Quantity>1</Quantity>
                  <UnitPrice>129.99</UnitPrice>
                  <CreatedBy>0</CreatedBy>
                  <CreationDate>2008-08-15 11:32:27.0</CreationDate>
                  <LastUpdatedBy>0</LastUpdatedBy>
                  <LastUpdateDate>2008-08-15 11:32:27.0</LastUpdateDate>
                  <ObjectVersionId>0</ObjectVersionId>
               </OrderItemsViewRow>
               <OrderItemsViewRow>
                  <OrderId>1017</OrderId>
                  <LineItemId>3</LineItemId>
                  <ProductId>36</ProductId>
                  <Quantity>2</Quantity>
                  <UnitPrice>659.99</UnitPrice>
                  <CreatedBy>0</CreatedBy>
                  <CreationDate>2008-08-15 11:32:27.0</CreationDate>
                  <LastUpdatedBy>0</LastUpdatedBy>
                  <LastUpdateDate>2008-08-15 11:32:27.0</LastUpdateDate>
                  <ObjectVersionId>0</ObjectVersionId>
               </OrderItemsViewRow>
            </OrderItemsView>
         </OrdersViewRow>
      </OrdersView>
   </PersonsViewRow>
...
```

42.7.3 What You May Need to Know About Reading and Writing XML

This section provides additional information to help you work with XML.

42.7.3.1 Controlling XML Element Names

You can use the Property Inspector to change the default XML element names used in the view object's canonical XML format by setting several properties. To accomplish this, open the overview editor for the view object, then:

- Select the attribute on the Attributes page and in the Property Inspector, select the Custom Properties navigation tab and set the custom *attribute*-level property named **Xml Element** to a value `SomeOtherName` to change the XML element name used for that attribute to `<SomeOtherName>`

  For example, the `Email` attribute in the `Persons` view object defines this property to change the XML element you see in Example 42–22 to be `<EmailAddress>` instead of `<Email>`.

- Select the General navigation tab in the Property Inspector and set the custom *view object*-level property named **Xml Row Element** to a value `SomeOtherRowName` to change the XML element name used for that view object to `<SomeOtherRowName>`.

  For example, the `Persons` view object defines this property to change the XML element name for the rows you see in Example 42–22 to be `<Person>` instead of `<PersonsRow>`.

- To change the name of the element names that wrapper nested row set data from view link attribute accessors, use the View Link Properties dialog. To open the dialog, in the view link overview editor, click the **Edit accessors** icon on the

Accessors section of Relationship page. Enter the desired name of the view link accessor attribute in the **Accessor Name** field.

42.7.3.2 Controlling Element Suppression for Null-Valued Attributes

By default, if a view row attribute is `null`, then its corresponding element is omitted from the generated XML. Select the attribute on the Attributes page of the overview editor and in the Property Inspector, select the Custom Properties navigation tab and set the custom *attribute*-level property named **Xml Explicit Null** to any value (e.g. "`true`" or "`yes`") to cause an element to be included for the attribute if its value is null. For example, if an attribute named `AssignedDate` has this property set, then a row containing a `null` assigned date will contain a corresponding <AssignedDate null="true"/> element. If you want this behavior for all attributes of a view object, you can define the **Xml Explicit Null** custom property at the view object level as a shortcut for defining it on each attribute.

42.7.3.3 Printing or Searching the Generated XML Using XPath

Two of the most common things you might want to do with the XML `Node` object returned from `writeXML()` are:

1. Printing the node to its serialized text representation — to send across the network or save in a file, for example
2. Searching the generated XML using W3C XPath expressions

Unfortunately, the standard W3C Document Object Model (DOM) API does not include methods for doing *either* of these useful operations. But there is hope. Since ADF Business Components uses the Oracle XML parser's implementation of the DOM, you can cast the `Node` return value from `writeXML()` to the Oracle specific classes `XMLNode` or `XMLElement` (in the `oracle.xml.parser.v2` package) to access additional useful functionality like:

- Printing the XML element to its serialized text format using the `print()` method
- Searching the XML element in memory with XPath expressions using the `selectNodes()` method
- Finding the value of an XPath expression related to the XML element using the `valueOf()` method.

Example 42–23 shows the `printXML()` method in the `TestClientWriteXML`. It casts the `Node` parameter to an `XMLNode` and calls the `print()` method to dump the XML to the console.

Example 42–23 Using the XMLNode's print() Method to Serialize XML

```
// In TestClientWriteXML.java
private static void printXML(Node n) throws IOException {
  ((XMLNode)n).print(System.out);
}
```

42.7.3.4 Using the Attribute Map For Fine Control Over Generated XML

When you need fine control over which attributes appear in the generated XML, use the version of the `writeXML()` method that accepts a `HashMap`. Example 42–24 shows the interesting lines from a `TestClientWriteXML` class that use this technique. After creating the `HashMap`, you put `String[]`-valued entries into it containing the names of the attributes you want to include in the XML, keyed by the fully-qualified name of the view definition those attributes belong to. The example includes the `PersonId`, `Email`, `PersonTypeCode`, and `OrdersView` attributes from

the `Persons` view object, and the `OrderId`, `OrderStatusCode`, and `OrderTotal` attributes from the `OrdersView` view object.

> **Note:** For upward compatibility reasons with earlier versions of ADF Business Components the `HashMap` expected by the `writeXML()` method is the one in the `com.sun.java.util.collections` package.

While processing the view rows for a given view object instance:

- If an entry exists in the attribute map with a key matching the fully-qualified view definition name for that view object, then only the attributes named in the corresponding `String` array are included in the XML.

  Furthermore, if the string array includes the name of a view link accessor attribute, then the nested contents of its detail row set are included in the XML. If a view link accessor attribute name does not appear in the string array, then the contents of its detail row set are not included.

- If no such entry exists in the map, then *all* attributes for that row are included in the XML.

Example 42–24 Using a View Definition Attribute Map for Fine Control Over Generated XML

```
HashMap viewDefMap = new HashMap();
viewDefMap.put("devguide.advanced.xml.queries.PersonsView",
        new String[]{"PersonId","Email", "PersonTypeCode",
                     "OrdersView" /* View link accessor attribute */
                     });
viewDefMap.put("devguide.advanced.xml.queries.OrdersView",
        new String[]{"OrderId","OrderStatusCode","OrderTotal"});
printXML(vo.writeXML(XMLInterface.XML_OPT_ALL_ROWS,viewDefMap));
```

Running the example produces the XML shown in Example 42–25, including only the exact attributes and view link accessors indicated by the supplied attribute map.

Example 42–25 XML from a Users View Object Produced Using an Attribute Map

```
<OrdersViewRow>
            <OrderId>1033</OrderId>
            <OrderDate>2009-01-30 13:59:39.0</OrderDate>
            <OrderShippedDate>2009-02-02 13:59:39.0</OrderShippedDate>
            <OrderStatusCode>COMPLETE</OrderStatusCode>
            <OrderTotal>2677.96</OrderTotal>
            <CustomerId>108</CustomerId>
            <ShipToAddressId>20</ShipToAddressId>
            <ShippingOptionId>1</ShippingOptionId>
            <PaymentOptionId>1016</PaymentOptionId>
            <DiscountId>3</DiscountId>
            <FreeShippingFlag>Y</FreeShippingFlag>
            <CustomerCollectFlag>Y</CustomerCollectFlag>
            <CollectionWarehouseId>101</CollectionWarehouseId>
            <GiftwrapFlag>N</GiftwrapFlag>
            <CreatedBy>0</CreatedBy>
            <CreationDate>2009-02-23 13:59:39.0</CreationDate>
            <LastUpdatedBy>0</LastUpdatedBy>
            <LastUpdateDate>2009-02-23 13:59:39.0</LastUpdateDate>
            <ObjectVersionId>0</ObjectVersionId>
```

```
            <OrderItemsView>
               <OrderItemsViewRow>
                  <OrderId>1033</OrderId>
                  <LineItemId>1</LineItemId>
                  <ProductId>10</ProductId>
                  <Quantity>3</Quantity>
                  <UnitPrice>225.99</UnitPrice>
                  <CreatedBy>0</CreatedBy>
                  <CreationDate>2009-02-23 13:59:40.0</CreationDate>
                  <LastUpdatedBy>0</LastUpdatedBy>
                  <LastUpdateDate>2009-02-23 13:59:40.0</LastUpdateDate>
                  <ObjectVersionId>0</ObjectVersionId>
               </OrderItemsViewRow>
               <OrderItemsViewRow>
                  <OrderId>1033</OrderId>
                  <LineItemId>2</LineItemId>
                  <ProductId>1</ProductId>
                  <Quantity>1</Quantity>
                  <UnitPrice>1999.99</UnitPrice>
                  <CreatedBy>0</CreatedBy>
                  <CreationDate>2009-02-23 13:59:40.0</CreationDate>
                  <LastUpdatedBy>0</LastUpdatedBy>
                  <LastUpdateDate>2009-02-23 13:59:40.0</LastUpdateDate>
                  <ObjectVersionId>0</ObjectVersionId>
               </OrderItemsViewRow>
            </OrderItemsView>
</OrdersViewRow>
...
```

42.7.3.5 Use the Attribute Map Approach with Bi-Directional View Links

If your view objects are related through a view link that you have configured to be bi-directional, then you must use the `writeXML()` approach that uses the attribute map. If you were to use the `writeXML()` approach in the presence of bi-directional view links and were to supply a maximum depth of `-1` to include all levels of view links that exist, the `writeXML()` method will go into an infinite loop as it follows the bi-directional view links back and forth, generating deeply nested XML containing duplicate data until it runs out of memory. Use `writeXML()` with an attribute map instead in this situation. Only by using this approach can you control which view link accessors are included in the XML and which are not to avoid infinite recursion while generating the XML.

42.7.3.6 Transforming Generated XML Using an XSLT Stylesheet

When the canonical XML format produced by `writeXML()` does not meet your needs, you can supply an XSLT stylesheet as an optional argument. It will produce the XML as it would normally, but then transform that result using the supplied stylesheet before returning the final XML to the caller.

Consider the XSLT stylesheet shown in Example 42–26. It is a simple transformation with a single template that matches the root element of the generated XML from Example 42–25 to create a new <CustomerEmailAddresses> element in the result. The template uses the <xsl:for-each> instruction to process all <PersonsView> elements that contain more than one <OrdersViewRow> child element inside a nested <OrdersViews> element. For each <PersonsView> element that qualifies, it creates a <Customer> element in the result whose `Contact` attribute is populated from the value of the <Email> child element of the <PersonsView>.

Example 42–26 XSLT Stylesheet to Transform Generated XML Into Another Format

```
<?xml version="1.0" encoding="windows-1252" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <CustomerEmailAddresses>
      <xsl:for-each
           select="/PersonsView/PersonsViewRow[count(OrdersView/OrdersViewRow) >
                                                1]">
        <xsl:sort select="Email"/>
        <Customer Contact="{Email}"/>
      </xsl:for-each>
    </CustomerEmailAddresses>
  </xsl:template>
</xsl:stylesheet>
```

Example 42–27 shows the interesting lines from a `TestClientWriteXML` class that put this XSLT stylesheet into action when calling `writeXML()`.

Example 42–27 Passing an XSLT Stylesheet to writeXML() to Transform the Resulting XML

```
// In TestClientWriteXML.java
XSLStylesheet xsl = getXSLStylesheet();
printXML(vo.writeXML(XMLInterface.XML_OPT_ALL_ROWS,viewDefMap,xsl));
```

Running the code in Example 42–27 produces the transformed XML shown here:

```
<CustomerEmailAddresses>
   <Customer Contact="dfaviet"/>
   <Customer Contact="jchen"/>
   <Customer Contact="ngreenbe"/>
</CustomerEmailAddresses>
```

The `getXSLStylesheet()` helper method shown in Example 42–28 is also interesting to study since it illustrates how to read a resource like an XSLT stylesheet from the classpath at runtime. The code expects the `Example.xsl` stylesheet to be in the same directory as the `TestClientWriteXML` class. By referencing the `Class` object for the `TestClientWriteXML` class using the `.class` operator, the code uses the `getResource()` method to get a `URL` to the resource. Then, it passes the URL to the `newXSLStylesheet()` method of the `XSLProcessor` class to create a new `XSLStylesheet` object to return. That object represents the compiled version of the XSLT stylesheet read in from the `*.xsl`file.

Example 42–28 Reading an XSLT Stylesheet as a Resource from the Classpath

```
private static XSLStylesheet getXSLStylesheet()
        throws XMLParseException, SAXException,IOException,XSLException {
  String xslurl = "Example.xsl";
  URL xslURL = TestClientWriteXML.class.getResource(xslurl);
  XSLProcessor xslProc = new XSLProcessor();
  return xslProc.newXSLStylesheet(xslURL);
}
```

> **Note:** When working with resources like XSLT stylesheets that you want to be included in the output directory along with your compiled Java classes and XML metadata, you can use the **Compiler** page of the **Project Properties** dialog to update the **Copy File Types to Output Directory** field to include `.xsl` in the semicolon-separated list.

42.7.3.7 Generating XML for a Single Row

In addition to calling `writeXML()` on a view object, you can call the same method with the same parameters and options on any `Row` as well. If the `Row` object on which you call `writeXML()` is a entity row, you can bitwise-OR the additional `XMLInterface.XML_OPT_CHANGES_ONLY` flag if you only want the changed entity attributes to appear in the XML.

42.7.4 How to Consume XML Documents to Apply Changes

To have a view object consume an XML document to process inserts, updates, and deletes, use the `readXML()` method:

```
void readXML(Element elem, int depthcount)
```

The canonical format expected by `readXML()` is the same as what would be produced by a call to the `writeXML()` method on the same view object. If the XML document to process does not correspond to this canonical format, you can supply an XSLT stylesheet as an optional third argument to `readXML()` to transform the incoming XML document *into* the canonical format before it is read for processing.

42.7.5 What Happens When You Consume XML Documents

When a view object consumes an XML document in canonical format, it processes the document to recognize row elements, their attribute element children, and any nested elements representing view link accessor attributes. It processes the document recursively to a maximum level indicated by the `depthcount` parameter. Passing `-1` for the `depthcount` to request that it process all levels of the XML document.

42.7.5.1 How ViewObject.readXML() Processes an XML Document

For each row element it recognizes, the `readXML()` method does the following:

- Identifies the related view object to process the row.
- Reads the children attribute elements to get the values of the primary key attributes for the row.
- Performs a `findByKey()` using the primary key attributes to detect whether the row already exists or not.
- If the row exists:
  - If the row element contains the marker attribute `bc4j-action="remove"`, then the existing row is deleted.
  - Otherwise, the row's attributes are updated using the values in any attribute element children of the current row element in the XML
- If the row does not exist, then a new row is created, inserted into the view object's rowset. Its attributes are populated using the values in any attribute element children of the current row element in the XML.

42.7.5.2 Using readXML() to Processes XML for a Single Row

The same `readXML()` method is also supported on any `Row` object. The canonical XML format it expects is the same format produced by a call to `writeXML()` on the same row. You can invoke `readXML()` method on a row to:

- Update its attribute values from XML

- Remove the row, if the `bc4j-action="remove"` marker attribute is present on the corresponding row element.
- Insert, update, or delete any nested rows via view link accessors

Consider the XML document shown in Example 42–29. It is in the canonical format expected by a single row in the `PersonsView` view object. Nested inside the root <PersonsViewRow> element, the <ConfirmedEmail> attribute represents the customer's email address. The nested <OrdersView> element corresponds to the `Orders` view link accessor attribute and contains three <OrdersViewRow> elements. Each of these includes <OrderId> elements representing the primary key of a `OrdersView` row.

Example 42–29 XML Document in Canonical Format to Insert, Update, and Delete Rows

```
<PersonsViewRow>
   <!-- This will update Person's ConfirmedEmail attribute -->
   <PersonId>110</PersonId>
   <ConfirmedEmail>NewConfirmed</ConfirmedEmail>
   <OrdersView>
      <!-- This will be an update since it does exist -->
      <OrdersViewRow>
         <OrderId>1011</OrderId>
         <OrderStatusCode>SHIP</OrderStatusCode>
      </OrdersViewRow>
      <!-- This will be an insert since it doesn't exist -->
      <OrdersViewRow>
         <OrderId>1070</OrderId>
         <OrderStatusCode>PENDING</OrderStatusCode>
      </OrdersViewRow>
      <!-- This will be deleted -->
      <OrdersViewRow bc4j-action="remove">
         <OrderId>1026</OrderId>
      </OrdersViewRow>
   </OrdersView>
</PersonsViewRow>
```

Example 42–30 shows the interesting lines of code from a `TestClientReadXML` class that applies this XML datagram to a particular row in the `PersonsView` view object. `TestClientReadXML` class performs the following basic steps:

1. Finds a target row by key (e.g. for customer "jchen").
2. Shows the XML produced for the row before changes are applied.
3. Obtains the parsed XML document with changes to apply using a helper method.
4. Reads the XML document to apply changes to the row.
5. Shows the XML with the pending changes applied.

   `TestClientReadXML` class is using the `XMLInterface.XML_OPT_ASSOC_CONSISTENT` flag described in Section 42.7.1, "How to Produce XML for Queried Data" to ensure that new, unposted rows are included in the XML.

Example 42–30 Applying Changes to an Existing Row with readXML()

```
ViewObject vo = am.findViewObject("CustomersView");
Key k = new Key(new Object[] { 110 });
// 1. Find a target row by key (e.g. for customer "jchen")
Row jchen = vo.findByKey(k, 1)[0];
// 2. Show the XML produced for the row before changes are applied
```

```
printXML(jchen.writeXML(-1, XMLInterface.XML_OPT_ALL_ROWS));
// 3. Obtain parsed XML document with changes to apply using helper method
Element xmlToRead = getInsertUpdateDeleteXMLGram();
printXML(xmlToRead);
// 4. Read the XML document to apply changes to the row
jchen.readXML(getInsertUpdateDeleteXMLGram(), -1);
// 5. Show the XML with the pending changes applied
printXML(jchen.writeXML(-1, XMLInterface.XML_OPT_ALL_ROWS |
                            XMLInterface.XML_OPT_ASSOC_CONSISTENT));
```

Running the code in Example 42–30 initially displays the "before" version of John Chen's information. Notice that:

- The `ConfirmedEmail` attribute has the value "JCHEN"
- The status code for order `1011` has a level of "CANCEL"
- There is an orders row for order `1026`, and
- There is no orders row related to order `1070`.

```
<PersonsViewRow>
   <PersonId>110</PersonId>
   <PrincipalName>JCHEN</PrincipalName>
   <FirstName>John</FirstName>
   <LastName>Chen</LastName>
   <PersonTypeCode>CUST</PersonTypeCode>
   <ProvisionedFlag>N</ProvisionedFlag>
   <PrimaryAddressId>37</PrimaryAddressId>
   <MembershipId>1</MembershipId>
   <Email>JCHEN</Email>
   <ConfirmedEmail>JCHEN</ConfirmedEmail>
   <PhoneNumber>706.555.0103</PhoneNumber>
   <DateOfBirth>1967-09-28</DateOfBirth>
   <MaritalStatusCode>MARR</MaritalStatusCode>
   <Gender>M</Gender>
   <ContactableFlag>Y</ContactableFlag>
   <ContactByAffilliatesFlag>Y</ContactByAffilliatesFlag>
   <CreatedBy>SEED_DATA</CreatedBy>
   <CreationDate>2009-02-23 13:59:38.0</CreationDate>
   <LastUpdatedBy>SEED_DATA</LastUpdatedBy>
   <LastUpdateDate>2009-02-23 13:59:38.0</LastUpdateDate>
   <ObjectVersionId>1</ObjectVersionId>
   <OrdersView>
      <OrdersViewRow>
         <OrderId>1011</OrderId>
         <OrderDate>2009-02-17 13:59:38.0</OrderDate>
         <OrderStatusCode>CANCEL</OrderStatusCode>
         <OrderTotal>99.99</OrderTotal>
         <CustomerId>110</CustomerId>
         <ShipToAddressId>9</ShipToAddressId>
         <ShippingOptionId>2</ShippingOptionId>
         <PaymentOptionId>1005</PaymentOptionId>
         <DiscountId>5</DiscountId>
         <FreeShippingFlag>N</FreeShippingFlag>
         <CustomerCollectFlag>N</CustomerCollectFlag>
         <GiftwrapFlag>N</GiftwrapFlag>
         <CreatedBy>0</CreatedBy>
         <CreationDate>2009-02-23 13:59:38.0</CreationDate>
         <LastUpdatedBy>anonymous</LastUpdatedBy>
         <LastUpdateDate>2009-02-23 13:59:38.0</LastUpdateDate>
         <ObjectVersionId>8</ObjectVersionId>
```

```
         <OrderItemsView>
            <OrderItemsViewRow>
               <OrderId>1011</OrderId>
               <LineItemId>1</LineItemId>
               <ProductId>18</ProductId>
               <Quantity>1</Quantity>
               <UnitPrice>99.99</UnitPrice>
               <CreatedBy>0</CreatedBy>
               <CreationDate>2009-02-23 13:59:39.0</CreationDate>
               <LastUpdatedBy>0</LastUpdatedBy>
               <LastUpdateDate>2009-02-23 13:59:39.0</LastUpdateDate>
               <ObjectVersionId>0</ObjectVersionId>
            </OrderItemsViewRow>
         </OrderItemsView>
      </OrdersViewRow>
...
```

After applying the changes from the XML document using `readXML()` to the row and printing its XML again using `writeXML()` you see that:

- The `ConfirmedEmail` is now "NewConfirmed"
- A new orders row for order `1070` got created.
- The status code for order `1011` has a level of "SHIP", and
- The orders row for order `1026` is removed

```
<PersonsViewRow>
   <PersonId>110</PersonId>
   <PrincipalName>JCHEN</PrincipalName>
   <FirstName>John</FirstName>
   <LastName>Chen</LastName>
   <PersonTypeCode>CUST</PersonTypeCode>
   <ProvisionedFlag>N</ProvisionedFlag>
   <PrimaryAddressId>37</PrimaryAddressId>
   <MembershipId>1</MembershipId>
   <Email>JCHEN</Email>
   <ConfirmedEmail>NewConfirmed</ConfirmedEmail>
   <PhoneNumber>706.555.0103</PhoneNumber>
   <DateOfBirth>1967-09-28</DateOfBirth>
   <MaritalStatusCode>MARR</MaritalStatusCode>
   <Gender>M</Gender>
   <ContactableFlag>Y</ContactableFlag>
   <ContactByAffilliatesFlag>Y</ContactByAffilliatesFlag>
   <CreatedBy>0</CreatedBy>
   <CreationDate>2009-02-23 13:59:38.0</CreationDate>
   <LastUpdatedBy>0</LastUpdatedBy>
   <LastUpdateDate>2009-02-23 13:59:38.0</LastUpdateDate>
   <ObjectVersionId>1</ObjectVersionId>
   <OrdersView>
      <OrdersViewRow>
         <OrderId>1070</OrderId>
         <OrderDate>2009-06-22</OrderDate>
         <OrderStatusCode>PENDING</OrderStatusCode>
         ...
      </OrdersViewRow>
      <OrdersViewRow>
         <OrderId>1011</OrderId>
         <OrderDate>2009-02-17 13:59:38.0</OrderDate>
         <OrderStatusCode>SHIP</OrderStatusCode>
         <OrderTotal>99.99</OrderTotal>
```

```
            <CustomerId>110</CustomerId>
            <ShipToAddressId>9</ShipToAddressId>
            <ShippingOptionId>2</ShippingOptionId>
            <PaymentOptionId>1005</PaymentOptionId>
            <DiscountId>5</DiscountId>
            <FreeShippingFlag>N</FreeShippingFlag>
            <CustomerCollectFlag>N</CustomerCollectFlag>
            <GiftwrapFlag>N</GiftwrapFlag>
            <CreatedBy>0</CreatedBy>
            <CreationDate>2009-02-23 13:59:38.0</CreationDate>
            <LastUpdatedBy>0</LastUpdatedBy>
            <LastUpdateDate>2009-02-23 13:59:38.0</LastUpdateDate>
            <ObjectVersionId>0</ObjectVersionId>
            <OrderItemsView>
               <OrderItemsViewRow>
                  <OrderId>1011</OrderId>
                  <LineItemId>1</LineItemId>
                  <ProductId>18</ProductId>
                  <Quantity>1</Quantity>
                  <UnitPrice>99.99</UnitPrice>
                  <CreatedBy>0</CreatedBy>
                  <CreationDate>2009-02-01 13:59:39.0</CreationDate>
                  <LastUpdatedBy>0</LastUpdatedBy>
                  <LastUpdateDate>2009-02-01 13:59:39.0</LastUpdateDate>
                  <ObjectVersionId>0</ObjectVersionId>
               </OrderItemsViewRow>
            </OrderItemsView>
         </OrdersViewRow>
...
```

Note: The example illustrated using `readXML()` to apply changes to a single row. If the XML document contained a wrapping <PersonsView> row, including the primary key attribute in each of its one or more nested <PersonsViewRow> elements, then that document could be processed using the `readXML()` method on the `PersonsView` view object for handling operations for multiple `PersonsView` rows.

42.8 Using Programmatic View Objects for Alternative Data Sources

By default view objects read their data from the database and automate the task of working with the Java Database Connectivity (JDBC) layer to process the database result sets. However, by overriding appropriate methods in its custom Java class, you can create a view object that programmatically retrieves data from alterative data sources like a `REF CURSOR`, an in-memory array, or a Java `*.properties` file, to name a few.

42.8.1 How to Create a Read-Only Programmatic View Object

To create a read-only programmatic view object, you use the Create View Object wizard.

To create the read-only programmatic view object:

1. In the Application Navigator, right-click the project in which you want to create the view object and choose **New**.

2. In the New Gallery, expand **Business Tier**, select **ADF Business Components** and then **View Object**, and click **OK**.
3. In the Create View Object wizard, in the Name page, provide a name and package for the view object. For the data source, select **Rows populated programmatically, not based on a query**.
4. In the Attributes page, click **New** one or more times to define the view object attributes your programmatic view object requires.
5. In the Attribute Settings page, adjust any setting you may need to for the attributes you defined.
6. In the Java page, select **Generate View Object Class** to enable a custom view object class (`ViewObjImpl`) to contain your code.
7. Click **Finish** to create the view object.

In your view object's custom Java class, override the methods described in Section 42.8.3, "Key Framework Methods to Override for Programmatic View Objects" to implement your custom data retrieval strategy.

42.8.2 How to Create an Entity-Based Programmatic View Object

To create a entity-based view object with programmatic data retrieval, create the view object in the normal way, enable a custom Java class for it, and override the methods described in the next section to implement your custom data retrieval strategy.

42.8.3 Key Framework Methods to Override for Programmatic View Objects

A programmatic view object typically overrides all of the following methods of the base `ViewObjectImpl` class to implement its custom strategy for retrieving data:

- `create()`

  This method is called when the view object instance is created and can be used to initialize any state required by the programmatic view object. At a minimum, this overridden method will contain the following lines to ensure the programmatic view object has no trace of a SQL query related to it:

  ```
  // Wipe out all traces of a query for this VO
  getViewDef().setQuery(null);
  getViewDef().setSelectClause(null);
  setQuery(null);
  ```

- `executeQueryForCollection()`

  This method is called whenever the view object's query needs to be executed (or re-executed).

- `hasNextForCollection()`

  This method is called to support the `hasNext()` method on the row set iterator for a row set created from this view object. Your implementation returns `true` if you have not yet exhausted the rows to retrieve from your programmatic data source.

- `createRowFromResultSet()`

  This method is called to populate each row of "fetched" data. Your implementation will call `createNewRowForCollection()` to create a new blank row and then `populateAttributeForRow()` to populate each attribute of data for the row.

- `getQueryHitCount()`

  This method is called to support the `getEstimatedRowCount()` method. Your implementation returns a count, or estimated count, of the number of rows that will be retrieved by the programmatic view object's query.

- `protected void releaseUserDataForCollection()`

  Your code can store and retrieve a user data context object with each row set. This method is called to allow you to release any resources that may be associated with a row set that is being closed.

Since the view object component can be related to several active row sets at runtime, many of the above framework methods receive an `Object` parameter named `qc` in which the framework will pass the collection of rows in question that your code is supposed to be filling, as well as the array of bind variable values that might affect which rows get populated into the specific collection.

You can store a user-data object with each collection of rows so your custom datasource implementation can associate any needed datasource context information. The framework provides the `setUserDataForCollection()` and `getUserDataForCollection()` methods to get and set this per-collection context information. Each time one of the overridden framework methods is called, you can use the `getUserDataForCollection()` method to retrieve the correct `ResultSet` object associated with the collection of rows the framework wants you to populate.

The examples in the following sections each override these methods to implement different kinds of programmatic view objects.

42.8.4 How to Create a View Object on a REF CURSOR

Sometimes your application might need to work with the results of a query that is encapsulated within a stored procedure. PL/SQL allows you to open a cursor to iterate through the results of a query, and then return a reference to this cursor to the client. This so-called `REF CURSOR` is a handle with which the client can then iterate the results of the query. This is possible even though the client never actually issued the original SQL `SELECT` statement.

> **Note:** The example in this section refers to the `ViewObjectOnRefCursor` project in the `AdvancedViewObjectsExamples` application workspace in the `StandaloneExamples` module of the Fusion Order Demo application.

Declaring a PL/SQL package with a function that returns a `REF CURSOR` is straightforward. For example, your package might look like this:

```
CREATE OR REPLACE PACKAGE RefCursorExample IS
  TYPE ref_cursor IS REF CURSOR;
  FUNCTION get_orders_for_customer(p_email VARCHAR2) RETURN ref_cursor;
  FUNCTION count_orders_for_customer(p_email VARCHAR2) RETURN NUMBER;
END RefCursorExample;
```

After defining an entity-based `OrdersForCustomer` view object with an entity usage for a `Order` entity object, go to its custom Java class `OrdersForCustomerImpl.java`. At the top of the view object class, define some constant Strings to hold the anonymous blocks of PL/SQL that you'll execute using JDBC `CallableStatement` objects to invoke the stored functions:

```
/*
 * Execute this block to retrieve the REF CURSOR
 */
 private static final String SQL =
           "begin ? := RefCursorSample.get_orders_for_customer(?);end;";
/*
 * Execute this block to retrieve the count of orders that
 * would be returned if you executed the statement above.
 */
private static final String COUNTSQL =
           "begin ? := RefCursorSample.count_orders_for_customer(?);end;";
```

Then, override the methods of the view object as described in the following sections.

42.8.4.1 The Overridden create() Method

The `create()` method removes all traces of a SQL query for this view object.

```
protected void create() {
  getViewDef().setQuery(null);
  getViewDef().setSelectClause(null);
  setQuery(null);
}
```

42.8.4.2 The Overridden executeQueryForCollection() Method

The `executeQueryForCollection()` method is executed when the framework needs to issue the database query for the query collection based on this view object. One view object can produce many related result sets, each potentially the result of different bind variable values. If the row set in query is involved in a framework-coordinated master/detail view link, then the `params` array will contain one or more framework-supplied name-value pairs of bind parameters from the source view object. If there are any user-supplied bind parameter values, they will *precede* the framework-supplied bind variable values in the `params` array, and the number of user parameters will be indicated by the value of the `numUserParams` argument.

The method calls a helper method `retrieveRefCursor()` to execute the stored function and return the `REF CURSOR` return value, cast as a JDBC `ResultSet`.

```
protected void executeQueryForCollection(Object qc,Object[] params,
                                         int numUserParams) {
  storeNewResultSet(qc,retrieveRefCursor(qc,params));
  super.executeQueryForCollection(qc, params, numUserParams);
}
```

Then, it calls the helper method `storeNewResultSet()` that uses the `setUserDataForCollection()` method to store this `ResultSet` with the collection of rows for which the framework is asking to execute the query.

```
private void storeNewResultSet(Object qc, ResultSet rs) {
  ResultSet existingRs = getResultSet(qc);
  // If this query collection is getting reused, close out any previous rowset
  if (existingRs != null) {
     try {existingRs.close();} catch (SQLException s) {}
  }
  setUserDataForCollection(qc,rs);
  hasNextForCollection(qc); // Prime the pump with the first row.
}
```

The `retrieveRefCursor()` method uses the helper method described in Section 12.6, "Invoking Stored Procedures and Functions" to invoke the stored function and return the `REF CURSOR`:

```
private ResultSet retrieveRefCursor(Object qc, Object[] params) {
  ResultSet rs = (ResultSet)callStoredFunction(OracleTypes.CURSOR,
                  "RefCursorExample.get_requests_for_customer(?)",
                  new Object[]{getNamedBindParamValue("CustEmail",params)});
  return rs ;
}
```

42.8.4.3 The Overridden createRowFromResultSet() Method

For each row that the framework needs fetched from the data source, it will invoke your overridden `createRowFromResultSet()` method. The implementation retrieves the collection-specific `ResultSet` object from the user-data context. It uses the `getResultSet()` method to retrieve the result set wrapper from the query-collection user data, and the `createNewRowForCollection()` method to create a new blank row in the collection, and then uses the `populateAttributeForRow()` method to populate the attribute values for each attribute defined at design time in the view object overview editor.

```
protected ViewRowImpl createRowFromResultSet(Object qc, ResultSet rs) {
  /*
   * We ignore the JDBC ResultSet passed by the framework (null anyway) and
   * use the resultset that we've stored in the query-collection-private
   * user data storage
   */
  rs = getResultSet(qc);

  /*
   * Create a new row to populate
   */
  ViewRowImpl r = createNewRowForCollection(qc);
  try {
    /*
     * Populate new row by attribute slot number for current row in Result Set
     */
    populateAttributeForRow(r,0, rs.getLong(1));
    populateAttributeForRow(r,1, rs.getString(2));
    populateAttributeForRow(r,2, rs.getString(3));
  }
  catch (SQLException s) {
    throw new JboException(s);
  }
  return r;
}
```

42.8.4.4 The Overridden hasNextForCollectionMethod()

The overridden implementation of the framework method `hasNextForCollection()` has the responsibility to return `true` or `false` based on whether there are more rows to fetch. When you've hit the end, you call the `setFetchCompleteForCollection()` to tell view object that this collection is done being populated.

```
protected boolean hasNextForCollection(Object qc) {
  ResultSet rs = getResultSet(qc);
  boolean nextOne = false;
  try {
    nextOne = rs.next();
```

```
    /*
     * When were at the end of the result set, mark the query collection
     * as "FetchComplete".
     */
    if (!nextOne) {
      setFetchCompleteForCollection(qc, true);
      /*
       * Close the result set, we're done with it
       */
      rs.close();
    }
  }
  catch (SQLException s) {
   throw new JboException(s);
  }
  return nextOne;
}
```

42.8.4.5 The Overridden releaseUserDataForCollection() Method

Once the collection is done with its fetch-processing, the overridden `releaseUserDataForCollection()` method gets invoked and closes the `ResultSet` cleanly so no database cursors are left open.

```
  protected void releaseUserDataForCollection(Object qc, Object rs) {
     ResultSet userDataRS = getResultSet(qc);
     if (userDataRS != null) {
      try {
        userDataRS.close();
      }
      catch (SQLException s) {
        /* Ignore */
      }
    }
    super.releaseUserDataForCollection(qc, rs);
  }
```

42.8.4.6 The Overridden getQueryHitCount() Method

Lastly, in order to properly support the view object's `getEstimatedRowCount()` method, the overridden `getQueryHitCount()` method returns a count of the rows that would be retrieved if all rows were fetched from the row set. Here the code uses a `CallableStatement` to get the job done. Since the query is completely encapsulated behind the stored function API, the code also relies on the PL/SQL package to provide an implementation of the count logic as well to support this functionality.

```
public long getQueryHitCount(ViewRowSetImpl viewRowSet) {
  Object[] params = viewRowSet.getParameters(true);
  BigDecimal id = (BigDecimal)params[0];
  CallableStatement st = null;
  try {
    st = getDBTransaction().createCallableStatement(COUNTSQL,
                            DBTransaction.DEFAULT);
    /*
     * Register the first bind parameter as our return value of type CURSOR
     */
    st.registerOutParameter(1,Types.NUMERIC);
    /*
     * Set the value of the 2nd bind variable to pass id as argument
     */
    if (id == null) st.setNull(2,Types.NUMERIC);
```

```
    else            st.setBigDecimal(2,id);
    st.execute();
    return st.getLong(1);
  }
  catch (SQLException s) {
    throw new JboException(s);
  }
  finally {try {st.close();} catch (SQLException s) {}}
}
```

42.9 Creating a View Object with Multiple Updatable Entities

By default, when you create a view object with multiple entity usages, each secondary entity usage that you add to a view object in the overview editor is configured with these settings:

- The **Updatable** checkbox is deselected
- The **Reference** checkbox is selected

You can change the default behavior to enable a secondary entity usage to be updatable by selecting the usage in the **Selected** list of the **Entity Objects** page of the view object overview editor and selecting the **Updatable** checkbox.

Additionally, for each entity usage, you can decide whether to leave **Reference** select to control whether or not to refresh the attributes of the secondary entity when the entity lookup information changes. By default, **Reference** is selected to ensure attributes of each secondary entity objects will be refreshed. For details about this setting when you allow row inserts with multiple entity usages, see Section 42.9.2, "What Happens at Runtime: View Row Creation."

Table 42–4 summarizes the combinations you can select when you define multiple entity usages for a view object.

Table 42–4 View Object Properties to Control View Row Creation Behavior

Updatable	Reference	View Row Behavior
`true`	`true`	This combination allows the entity usage's attributes to be updated and keeps its attributes synchronized with the value of the primary key. Since this combination works fine with the view link consistency feature, you can use it to make sure your view object only has one entity object usage that will participate in inserts.

Table 42–4 (Cont.) View Object Properties to Control View Row Creation Behavior

Updatable	Reference	View Row Behavior
`true`	`false`	This combination allows the entity usage's attributes to be updated but prevents its attributes from being changed by the a primary key lookup. This is a rather rare combination, and works best in situations where you only plan to use the view object to update or delete existing data. With this combination, the user can update attributes related to any of the non-reference, updatable entity usages and the view row will delegate the changes to the appropriate underlying entity rows. Note: The combination of the view link consistency feature with a view object having some of its secondary entity usages set as Updatable=true, Reference=false can end up creating unwanted extra new entities in your application.
`false`	`true`	This is the default behavior, described in Section 5.5.1, "How to Create Joins for Entity-Based View Objects." This combination assumes you do not want the entity usage to be updatable.

If you need a view object with multiple updatable entities to support creating new rows (`Updatable=true`, `Reference=false`) and the association between the entity objects is not a composition, then you need to write a bit of code, as described in Section 42.9.1, "How to Programmatically Create New Rows With Multiple Updatable Entity Usages."

42.9.1 How to Programmatically Create New Rows With Multiple Updatable Entity Usages

If you need a view object with multiple updatable entities to support creating new rows (`Updatable=true`, `Reference=false`) and the association between the entity objects is not a composition, then you need to override the `create()` method of the view object's custom view row class to enable that to work correctly.

> **Note:** You only need to write code to handle creating new rows when the association between the updatable entities is not a composition. If the association is a composition, then ADF Business Components handles this automatically.

When you call `createRow()` on a view object with multiple update entities, it creates new entity row parts for each updatable entity usage. Since the multiple entities in this scenario are related by an association, there are three pieces of code you might need to implement to ensure the new, associated entity rows can be saved without errors:

1. You may need to override the `postChanges()` method on entity objects involved to control the correct posting order.

2. If the primary key of the associated entity is populated by a database sequence using `DBSequence`, and if the multiple entity objects are associated but not composed, then you need to override the `postChanges()` and `refreshFKInNewContainees()` method to handle cascading the refreshed primary key value to the associated rows that were referencing the temporary value.
3. You need to override the `create()` method of the view object's custom view row class to modify the default row creation behavior to pass the context of the parent entity object to the newly-created child entity.

To understand the code for steps 1 and 2, see the example with associated `Suppliers` and `Products` entity objects described in Section 4.14.7, "How to Control Entity Posting Order to Prevent Constraint Violations." The last thing you need to understand is how to override `create()` method on the view row. Consider a `ProductAndSupplier` view object with a primary entity usage of `Product` and secondary entity usage of `Supplier`. Assume the `Product` entity usage is marked as updatable and non-reference, while the `Supplier` entity usage is a reference entity usage.

Example 42–31 shows the commented code required to correctly sequence the creation of the multiple, updatable entity row parts during a view row create operation.

Example 42–31 Overriding View Row create() Method for Multiple Updatable Entities

```
/**
 * By default, the framework will automatically create the new
 * underlying entity object instances that are related to this
 * view object row being created.
 *
 * We override this default view object row creation to explicitly
 * pre-populate the new (detail) ProductsImpl instance using
 * the new (master) SuppliersImpl instance. Since all entity objects
 * implement the AttributeList interface, we can directly pass the
 * new SuppliersImpl instance to the ProductsImpl create()
 * method that accepts an AttributeList.
 */
protected void create(AttributeList attributeList) {
  // The view row will already have created "blank" entity instances
  SuppliersImpl newSupplier = getSupplier();
  ProductsImpl newProduct = getProduct();
   try {
      // Let product "blank" entity instance to do programmatic defaulting
      newSupplier.create(attributeList);
      // Let product "blank" entity instance to do programmatic
      // defaulting passing in new SuppliersImpl instance so its attributes
      // are available to the EmployeeImpl's create method.
      newProduct.create(newSupplier);
   }
   catch (JboException ex) {
     newSupplier.revert();
     newProduct.revert();
     throw ex;
   }
   catch (Exception otherEx) {
     newSupplier.revert();
     newProduct.revert();
     throw new RowCreateException(true      /* EO Row? */,
                                  "Product" /* EO Name */,
                                  otherEx   /* Details */);
```

```
    }
  }
```

In order for this `ProductAndSupplier` view object's view row class (`ProductAndSupplierRowImpl` class) to be able to invoke the protected `create()` method on the `Suppliers` and `Products` entity objects, the entity object classes need to override their `create()` methods:

```
/**
 * Overridding this method in this class allows friendly access
 * to the create() method by other classes in this same package, like the
 * ProductsAndSuppliers view object implementation class, whose overridden
 * create() method needs to call this.
 * @param nameValuePair
 */
  protected void create(AttributeList nameValuePair) {
    super.create(nameValuePair);
  }
```

When overriding the `create()` method, the declaration of the method will depend on the following conditions:

- If the view object and entity objects are in the same package, the overridden `create()` method can have protected access and the `ProductAndSupplierRowImpl` class will have access to them.
- If either entity object is in a different package, then `SupplierImpl.create()` and `ProductImpl.create()` (whichever is in a different package) have to be declared `public` in order for the `SupplierAndProductViewRowImpl` class to be able to invoke them.

42.9.2 What Happens at Runtime: View Row Creation

If you need a view object with multiple updatable entities to support creating new rows, you will want to understand that the **Reference** flag controls behavior related to view row creation, as well as automatic association-driven lookup of information. If you disable the **Reference** flag for a given entity usage, then:

- Each time a new view row is created, a new entity instance will be created for that entity usage.
- The entity row to which the view row points for its storage of view row attributes related to that entity usage is never changed automatically by the framework.

Conversely, if you leave the **Reference** flag enabled (default) for an entity usage then:

- No new entity instance will be created for that entity usage when a new view row is created.
- The entity row to which the view row points for storage of view row attributes related to that entity usage will automatically be kept in sync with changes made to attributes in the view row that participate in an association with said entity.

Consider an `EmployeeView` view object that joins information from the DEPT and EMP tables to show `Empno`, `Ename`, `EmpDeptno` (a name chosen in the view object editor to identify this as an attribute of `EmployeeView`), `DeptDeptno` (a name chosen in the view object editor to identify this as an attribute of `DepartmentView`) and `Dname` attributes.

Now, consider what happens at runtime for the default case where you setup the secondary entity object marked as both updatable and reference:

- The Employee EO is the primary entity usage.
- The Department EO is a secondary entity usage and is marked as a Reference.

When the user creates a new row in the `EmployeeView`, ADF Business Components only creates a new Employee EO instance (since the Reference flag for the Department EO usage is enabled). If the user changes the employee's `DeptDeptno` attribute to `10`, then ADF Business Components will automatically look up the Department entity with primary key `10` in the entity cache (reading it in from the database if not already in the cache) and make this new view row's Department entity usage point to this department `10` entity. That has the result of automatically reflecting the right `Dname` value for department `10`.

In the default scenario, the reference lookup occurs both because the entity usage for Department is marked as a reference, as well as the fact that an association exists between the Employee entity and the Department entity. Through the association definition, ADF Business Components knows which attributes are involved on each side of this association. When any of the attributes on the Employee side of the `EmpToDept` association are modified, if the Department entity usage is marked as a Reference, ADF Business Components will perform that automatic reference lookup. If the user sets the `Dname` to `NewDept` in this new view row, after committing the change, the database will have a new employee in department `10` and have updated the name of department `10` to `NewDept`.

Now, consider what happens at runtime where you setup the secondary entity object marked as updatable and reference is disabled:

- The Employee entity object is the primary entity usage.
- The Department entity object is a secondary entity usage, but this usage is not marked as a Reference.

In this scenario, when the user creates a new row in the `EmployeeView`, ADF Business Components will create both a new Employee EO instance and a new Department EO instance (since the Reference flag for the Department EO usage is disabled). If the user changes the employee's `Deptno` attribute to `10`, it will have no effect on the value of the `Dname` attribute being displayed in the row. Additionally, if the user sets `DeptDeptno` to `99` and `Dname` to `NewDept` in this new view row, after commiting the changes, the database will have both a new employee in department `10` and a new department number `99`.

42.10 Programmatically Creating View Definitions and View Objects

The `oracle.jbo.server.ViewDefImpl` class lets you dynamically define the view definition meta-object for view object instances. The view definition describes the view object's structure.

Typically, the application creates the view definition object by loading an XML file that you create using JDeveloper overview editors. When the application needs to create a view object instance, it queries the `MetaObjectManager` for the view definition with the same name, it then finds the XML file, opens it, parses it, and builds a view definition.

Alternatively, you can create the view definition programmatically using methods of the `ViewDefImpl` class. The `ViewDefImpl` methods simplify creating view objects dynamically without incurring runtime overhead that occurs if you were to call `createViewObjectFromQueryStmt()` API on the `ApplicationModule` interface at runtime.

When you create a programmatic view definition, your application code must begin with:

```
ViewDefImpl viewDef = new ViewDefImpl("MyViewDef");
```

Where `MyViewDef` is the name by which the view definition object will identified by `MetaObjectManager`. This instance name is the name that the application should pass when an API requires a view definition name. For example, your application might request the `defName` parameter when invoking `ApplicationModule.createViewObject(String, String)`.

Example 42–32 illustrates instantiating the view definition and defining the view object SQL statement (either in one statement or in parts). The view definition code must resolve the definition object through `resolveDefObject()` and register it with `MetaObjectManager` through `RowDef.registerDefObject()`.

To create a view definition and then create a view object instance based on that definition, follow these basic steps (as illustrated in Example 42–32):

1. Create the view definition object.
2. Define the view object SQL statement.
3. Resolve and register the view definition.
4. With the view definition, construct instance of view objects based on it.

Example 42–32 Creating a View Definition Using the ViewDefImpl API

```
/*
 * 1. Create the view definition object.
 */
ViewDefImpl v = new ViewDefImpl("some.unique.DefNameForTheObject");

/*
* 2. Then, define the view object's SQL statement by either using a fully-
* specified "expert-mode" SQL query.
*/
v.setQuery("select e.empno,e.ename,e.sal,e.deptno,d.dname,d.loc,"+
              "d.deptno,trunc(sysdate)+1 tomorrow_date, "+
              "e.sal + nvl(e.comm,0) total_compensation, "+
              "to_char(e.hiredate,'dd-mon-yyyy') formated_hiredate"+
              "  from emp e, dept d "+
              " where e.deptno = d.deptno (+)"+
              " order by e.ename");
v.setFullSql(true);

/*
* Or, you can construct the SQL statement in parts like this.
*/
v.setSelectClause("e.empno,e.ename,e.sal,e.deptno,d.dname,d.loc,"+
                      "d.deptno,trunc(sysdate)+1 tomorrows_date,"+
                       "e.sal + nvl(e.comm,0) total_compensation, "+
                       "to_char(e.hiredate,'dd-mon-yyyy') formated_hiredate");
v.setFromClause("emp e, dept d");
v.setWhereClause("e.deptno = d.deptno (+)");
v.setOrderByClause("e.ename");

/*
* 3. Then resolve and register the view definition.
*/
v.resolveDefObject();
```

```
v.registerDefObject();

/*
* 4. Finally, use the dynamically-created view definition to construct
* instances of view objects based on it.
*/
ViewObject vo = createViewObject("SomeInstanceName",v);
```

42.11 Declaratively Preventing Insert, Update, and Delete

Some 4GL tools like Oracle Forms provide declarative properties that control whether a given data collection allows inserts, updates, or deletes. While the view object does not yet support this as a built-in feature in the current release, it's easy to add this facility using a framework extension class that exploits custom metadata properties as the developer-supplied flags to control insert, update, or delete on a view object.

> **Note:** The example in this section refers to the `DeclarativeBlockOperations` project in the `AdvancedViewObjectsExamples` application workspace in the `StandaloneExamples` module of the Fusion Order Demo application.

To allow developers to have control over individual view object instances, you could adopt the convention of using application module custom properties by the same name as the view object instance. For example, if an application module has view object instances named `ProductsInsertOnly`, `ProductsUpdateOnly`, `ProductsNoDelete`, and `Products`, your generic code might look for application module custom properties by these same names. If the property value contains `Insert`, then insert is enabled for that view object instance. If the property contains `Update`, then update allowed. And, similarly, if the property value contains `Delete`, then delete is allowed. You could use helper methods like this to test for these application module properties and determine whether insert, update, and delete are allowed for a given view object:

```
private boolean isInsertAllowed() {
  return isStringInAppModulePropertyNamedAfterVOInstance("Insert");
}
private boolean isUpdateAllowed() {
  return isStringInAppModulePropertyNamedAfterVOInstance("Update");
}
private boolean isDeleteAllowed() {
  return isStringInAppModulePropertyNamedAfterVOInstance("Delete");
}
private boolean isStringInAppModulePropertyNamedAfterVOInstance(String s) {
  String voInstName = getViewObject().getName();
  String propVal = (String)getApplicationModule().getProperty(voInstName);
  return propVal != null ? propVal.indexOf(s) >= 0 : true;
}
```

Example 42–33 shows the other code required in a custom framework extension class for view rows to complete the implementation. It overrides the following methods:

- `isAttributeUpdateable()`

  To enable the user interface to disable fields in a new row if insert is not allowed or to disable fields in an existing row if update is not allowed.

- `setAttributeInternal()`

  To prevent setting attribute values in a new row if insert is not allowed or to prevent setting attributes in an existing row if update is not allowed.

- `remove()`

  To prevent remove if delete is not allowed.

- `create()`

  To prevent create if insert is not allowed.

Example 42–33 Preventing Insert, Update, or Delete Based on Custom Properties

```
public class CustomViewRowImpl extends ViewRowImpl {
  public boolean isAttributeUpdateable(int index) {
    if (hasEntities() &&
        ((isNewOrInitialized() && !isInsertAllowed()) ||
         (isModifiedOrUnmodified() && !isUpdateAllowed()))) {
      return false;
    }
    return super.isAttributeUpdateable(index);
  }
  protected void setAttributeInternal(int index, Object val) {
    if (hasEntities()) {
      if (isNewOrInitialized() && !isInsertAllowed())
        throw new JboException("No inserts allowed in this view");
      else if (isModifiedOrUnmodified() && !isUpdateAllowed())
        throw new JboException("No updates allowed in this view");
    }
    super.setAttributeInternal(index, val);
  }
  public void remove() {
    if (!hasEntities() || isDeleteAllowed() || isNewOrInitialized())
      super.remove();
    else
        throw new JboException("Delete not allowed in this view");
  }
  protected void create(AttributeList nvp) {
    if (isInsertAllowed()) {
      super.create(nvp);
    } else {
      throw new JboException("Insert not allowed in this view");
    }
  }
  // private helper methods omitted from this example
}
```

43

Application State Management

This chapter describes the Fusion web application state management facilities and how to use them.

This chapter includes the following sections:

- Section 43.1, "Understanding Why State Management is Necessary"
- Section 43.2, "Introduction to Fusion Web Application State Management"
- Section 43.3, "Using Save For Later"
- Section 43.4, "Setting the Application Module Release Level at Runtime"
- Section 43.5, "What Model State Is Saved and When It Is Cleaned Up"
- Section 43.6, "Timing Out the HttpSession"
- Section 43.7, "Managing Custom User-Specific Information"
- Section 43.8, "Managing the State of View Objects"
- Section 43.9, "Using State Management for Middle-Tier Savepoints"
- Section 43.10, "Testing to Ensure Your Application Module is Activation-Safe"
- Section 43.11, "Keeping Pending Changes in the Middle Tier"

43.1 Understanding Why State Management is Necessary

Most real-world business applications need to support multi-step user tasks. Modern sites tend to use a step-by-step style user interface to guide the end user through a logical sequence of pages to complete these tasks. When the task is done, the user can save or cancel everything as a unit. However, the HTTP protocol is built on the concept of individual, stateless requests, so the responsibility of grouping a sequence of user actions into a logical, stateful unit of work falls to the application developer.

43.1.1 Examples of Multi-Step Tasks

In a typical search-then-edit scenario, the end user searches to find an appropriate row to update, then may open several different pages of related master/detail information to make edits before deciding to save or cancel his work. Consider another scenario where the end user wants to book a vacation online. The process may involve the end user's entering details about:

- One or more flight segments that comprise the journey
- One or more passengers taking the trip
- Seat selections and meal preferences

- One or more hotel rooms in different cities
- Car they will rent

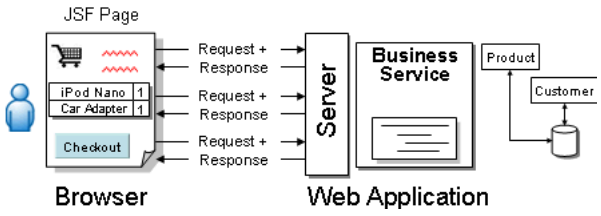
Along the way, the user might decide to complete the transaction, save the reservation for finishing later, or abandon the whole thing.

It's clear these scenarios involve a logical unit of work that spans multiple web pages. You've seen in previous chapters how to use JDeveloper's JSF page navigation diagram to design the page flow for these use cases, but that is only part of the puzzle. The pending changes the end user makes to business domain objects along the way — `Trip`, `Flight`, `Passenger`, `Seat`, `HotelRoom`, `Auto`, etc. — represent the in-progress state of the application for each end user. Along with this, other types of "bookkeeping" information about selections made in previous steps comprise the complete picture of the application state.

43.1.2 Stateless HTTP Protocol Complicates Stateful Applications

While it may be easy to imagine these multi-step scenarios, *implementing* them in web applications is complicated by the stateless nature of HTTP, the hypertext transfer protocol. Figure 43–1 illustrates how an end user's visit to a site comprises a series of HTTP request/response pairs. However, HTTP affords a web server no way to distinguish one user's request from another user's, or to differentiate between a single user's first request and any subsequent requests he makes while interacting with the site. The server gets each request from any user always as if it were the first (and only) one they make.

Figure 43–1 Web Applications Use the Stateless HTTP Protocol



43.1.3 How Cookies Are Used to Track a User Session

As shown in Figure 43–2, the technique used to recognize an ongoing sequence of requests from the same end user over the stateless HTTP protocol involves a unique identifier called a *cookie*. A cookie is a name/value pair that is sent in the header information of each HTTP request the user makes to a site. On the initial request made by a user, the cookie is not part of the request. The server uses the *absence* of the cookie to detect the start of a user's session of interactions with the site, and it returns a unique identifier to the browser that represents this session for this user. In practice, the cookie value is a long string of letters and numbers, but for the simplicity of the illustration, assume that the unique identifier is a letter like "A" or "Z" that corresponds to different users using the site.

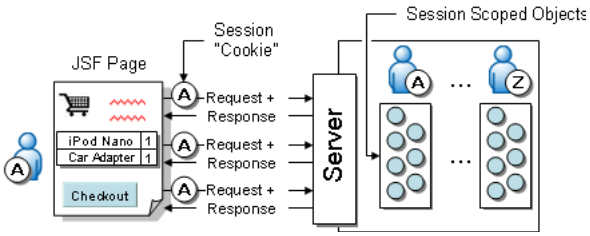
Web browsers support a standard way of recognizing the cookie returned by the server that allows the browser to identify the following:

- the site that sent the cookie
- how long it should remember the cookie value

On each subsequent request made by that user, until the cookie expires, the browser sends the cookie along in the header of the request. The server uses the value of the cookie to distinguish between requests made by different users.

Cookies can be set to live beyond a single browser session so that they might expire in a week, a month, or a year from when they were first created, while a session cookie expires when you close your browser.

Figure 43–2 Tracking State Using a Session Cookies and Server-Side Session



Java EE-compliant web servers provide a standard server-side facility called the `HttpSession` that allows a web application to store Java objects related to a particular user's session as named attribute/value pairs. An object placed in this session `Map` on one request can be retrieved by the application while handling a subsequent request during the same session.

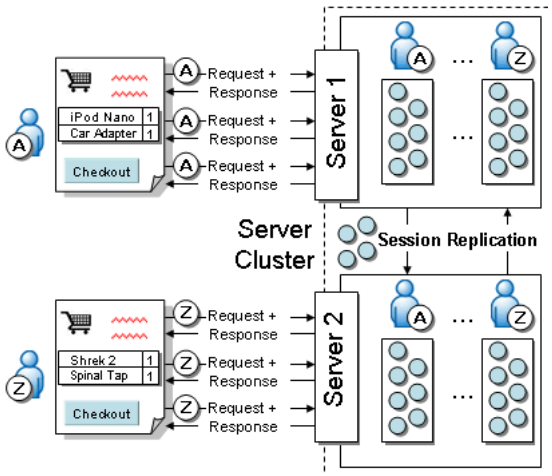
The session remains active while the user continues to send new requests within the timeframe configured by the `<session-timeout>` element in the `web.xml` file. The default session length is 35 minutes.

43.1.4 Performance and Reliability Impact of Using HttpSession

The `HttpSession` facility is an ingredient in most application state management strategies, but it can present performance and reliability problems if not used judiciously. First, because the session-scope Java objects you create are held in the memory of the Java EE web server, the objects in the HTTP session are lost if the server should fail.

As shown in Figure 43–3, one way to improve the reliability is to configure multiple Java EE servers in a cluster. By doing this, the Java EE application server replicates the objects in the HTTP session for each user across multiple servers in the cluster so that if one server goes down, the objects exist in the memory of the other servers in the cluster that can continue to handle the users requests. Since the cluster comprises separate servers, replicating the HTTP session contents among them involves broadcasting the changes made to HTTP session objects over the network.

Figure 43–3 Session Replication in a Server Cluster



You can begin to see some of the performance implications of overusing the HTTP session:

- The more active users, the more HTTP sessions will be created on the server.
- The more objects stored in each HTTP session, the more memory you will need. Note that the memory is not reclaimed when the user becomes inactive; this only happens with a session timeout or an explicit session invalidation. Session invalidations don't always happen because users don't always logout.
- In a cluster, the more objects in each HTTP session that *change*, the more network traffic will be generated to replicate the changed objects to other servers in the cluster.

At the outset, it would seem that keeping the number of objects stored in the session to a minimum addresses the problem. However, this implies leveraging an alternative mechanism for temporary storage for each user's pending application state. The most popular alternatives involve saving the application state to the database between requests or to a file of some kind on a shared file system.

Of course, this is easier said than done. A possible approach involves eagerly saving the pending changes to your underlying database tables and committing the transaction at the end of each HTTP request. But this idea has two key drawbacks:

- *Your database constraints might fail.*

  At any given step of the multi-step process, the information may only be partially complete, and this could cause errors at the database level when trying to save the changes.

- *You complicate rolling back the changes.*

  Cancelling the logical of unit of work would involve carefully deleting all of the eagerly-committed rows in possible multiple tables.

These limitations have led developers in the past to invent solutions involving a "shadow" set of database tables with no constraints and with all of the column types defined as character-based. Using such a solution becomes very complex very quickly. Ultimately, you will conclude that you need some kind of generic application state management facility to address these issues in a more generic and workable way. The solution comes in the form of ADF Business Components, which implements this for you out of the box.

43.2 Introduction to Fusion Web Application State Management

State management enables you to easily create web applications that support multi-step use cases without falling prey to the memory, reliability, or implementation complexity problems described in Section 43.1, "Understanding Why State Management is Necessary."

Application state management is provided at two levels, by the Save For Later feature in a task flow, and application module state management in the model layer.

Save For Later is activated at the controller layer and automatically saves a "snapshot" of the current UI and controller states, and delegates to the model layer to *passivate* (save) its state as well.

If you are not using ADF data controls, you can still use application module state management alone, but since this will save only the model state, this is an outside case for most applications.

43.2.1 Basic Architecture of the Save for Later Facility

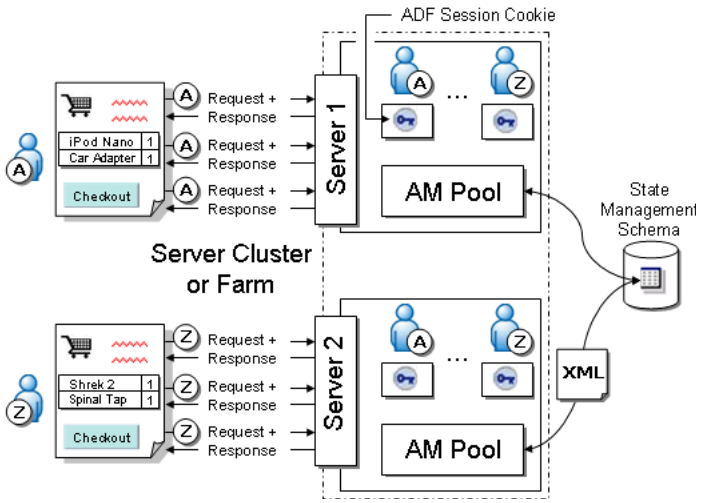
Save for Later saves an incomplete transaction without enforcing validation rules or submitting the data. The end user can resume working on the same transaction later with the same data that was originally saved when the application was exited.

43.2.2 Basic Architecture of the Application Module State Management Facility

Your ADF Business Components-based application automatically manages the application state of each user session. This provides the simplicity of a stateful programming model that you are accustomed to in previous 4GL tools, yet implemented in a way that delivers scalability nearing that of a purely stateless application. Understanding what happens behind the scenes is essential to make the most efficient use of this important feature.

You can use application module components to implement completely stateless applications or to support a unit of work that spans multiple browser pages. Figure 43–4 illustrates the basic architecture of the state management facility to support these multi-step scenarios. An application module supports *passivating* (storing) its pending transaction state to an XML document, which is stored in the database in a single, generic table, keyed by a unique passivation snapshot ID. It also supports the reverse operation of *activating* pending transaction state from one of these saved XML snapshots. This passivation and activation is performed automatically by the application module pool when needed.

Figure 43–4 ADF Provides Generic, Database-Backed State Management



The ADF binding context is the one object that lives in the `HttpSession` for each end user. It holds references to lightweight application module data control objects that manage acquiring an application module instance from the pool during the request (when the data control is accessed) and releasing it to the pool at the end of each request. The data control holds a reference to the ADF session cookie that identifies the user session. In particular, business domain objects created or modified in the pending transaction are *not* saved in the `HttpSession` using this approach. This minimizes both the session memory required per user and eliminates the network traffic related to session replication if the servers are configured in a cluster.

For improved reliability, serialize your session objects. Objects stored in distributed sessions need to implement the `java.io.Serializable` interface. Implementing this interface ensures the data can be transported over-the-wire to each server instance in the cluster. Use a custom method like the `addObjectToSession(String key, Serializable value)` method, instead of the default `HttpSession.setAttribute (String key, Object value)` method when adding session data. The distinction is, if you were to call the `addObjectToSession()` method with a non-serializable object, you would see a compile-time error. If you were to try to replicate a session object that had non-serializable objects placed into session with the `put()` method, you would see a runtime error and potentially, a broken user experience.

Additionally, if you have multiple application servers and you enable the optional ADF Business Components failover support (explained in Section 43.2.2.2, "How Passivation Changes When Optional Failover Mode is Enabled"), then subsequent end-user requests can be handled by any server in your server farm or cluster. The session cookie can reactivate the pending application state from the database-backed XML snapshot if required, regardless of which server handles the request.

43.2.2.1 Understanding When Passivation and Activation Occurs

To better understand when the automatic passivation and activation of application module state occurs, consider the following simple case:

1. At the beginning of an HTTP request, the application module data control handles the `beginRequest` event by checking out an application module instance from the pool.

The application module pool returns an *unreferenced* instance. An unreferenced application module is one that is not currently managing the pending state for any other user session.

2. At the end of the request, the application module data control handles the `endRequest` event by checking the application module instance back into the pool in "managed state" mode.

   That application module instance is now *referenced* by the data control that just used it. And the application module instance is an object that still contains pending transaction state made by the data control (that is, entity object and view object caches; updates made but not committed; and cursor states), stored in memory. As you'll see below, it's not *dedicated* to this data control, just referenced by it.

3. On a subsequent request, the same data control — identified by its `SessionCookie` — checks out an application module instance again.

   Due to the "stateless with user affinity" algorithm the pool uses, you might assume that the pool returns the exact same application module instance, with the state still there in memory. (To understand this algorithm, read Section 44.1, "About Application Module Pooling" and the discussion of Referenced Pool Size in Section 44.2.7.2, "Pool Sizing Parameters.")

Sometimes due to a high number of users simultaneously accessing the site, application module instances must be sequentially reused by different user sessions. In this case, the application pool must *recycle* a currently referenced application module instance for use by another session, as follows:

1. The application module data control for User A's session checks an application module instance into the application pool at the end of a request. Assume this instance is named `AM1`.

2. The application module data control for User Z's new session requests an application module instance from the pool for the first time, but there are no unreferenced instances available. The application module pool then:

   - Passivates the state of instance `AM1` to the database.
   - Resets the state of `AM1` in preparation to be used by another session.
   - Returns the `AM1` instance to User Z's data control.

3. On a subsequent request, the application module data control for User A's session requests an application module instance from the pool. The application module pool then:

   - Obtains an unreference instance.

     This could be instance `AM1`, obtained by following the same steps as in (2) above, or another `AM2` instance if it had become unreferenced in the meantime.

   - Activates the appropriate pending state for User A from the database.
   - Returns the application module instance to User A's data control.

The process of passivation, activation, and recycling allows the state referenced by the data control to be preserved across requests without requiring a dedicated application module instance for each data control. Both browser users in the above scenario are carrying on an application transaction that spans multiple HTTP requests, but the end users are unaware whether the passivation and activation is occurring in the background. They just continue to see the pending changes. In the process, the

pending changes never need to be saved into the underlying application database tables until the end user is ready to commit the logical unit of work.

Note that this keeps the session memory footprint low because the only business component objects that are directly referenced by the session (and are replicable) are the data control and the session cookie.

The application module pool makes a best effort to keep an application module instance "sticky" to the current data control whose pending state it is managing. This is known as maintaining user session affinity. The best performance is achieved if a data control continues to use exactly the same application module instance on each request, since this avoids any overhead involved in reactivating the pending state from a persisted snapshot.

43.2.2.2 How Passivation Changes When Optional Failover Mode is Enabled

The `jbo.dofailover` parameter controls when and how often passivation occurs. You can set this parameter in your application module configuration on the **Pooling and Scalability** tab of the Business Components Configuration dialog. When the failover feature is disabled, which it is by default, then application module pending state will only be passivated on demand when it must be. This occurs just before the pool determines it must hand out a currently-referenced application module instance to a *different* data control.

The passivation activity (and the corresponding wait) occurs only if failover is disabled *and* session replication is disabled. If either failover or session replication is enabled, then the recycling thread will reuse the snapshot that was already acquired to support the failover.

Note: Passivation can also occur when an application module is timed out. For more information about application pool removal algorithms (such as `jbo.ampool.timetolive`), see Section 44.2.7.3, "Pool Cleanup Parameters."

In contrast, with the failover feature turned on, the application module's pending state is passivated every time it is checked back into application module pool. This provides the most pessimistic protection against application server failure. The application module instances' state is always saved and may be activated by any application module instance at any time. Of course, this capability comes at expense of the additional overhead of eager passivation on each request.

When failover is turned on, a failure can occur when Oracle WebLogic Server is configured to forcibly release connections back into the pool. A failure of this type produces a `SQLException` (Connection has already been closed.) that is saved to the server log. The exception is not reported through the user interface. To ensure that state passivation occurs and users' changes are saved, the server administrator should set an appropriate value for the `weblogic-application.xml` deployment descriptor parameter `inactive-connection-timeout-seconds` on the `<connection-check-params>` pool params element. Setting the deployment descriptor parameter to several minutes, in most cases, should avoid forcing the inactive connection timeout and the resulting passivation failure. Adjust the setting as needed for your environment.

Best Practice: By default, the failover feature is disabled (`jbo.dofailover=false`) as a performance optimization when there is only one web server instance configured, by reducing the need for passivation and activation. This allows for application module affinity to a specific user session.

For high availability, enable the failover feature (`jbo.dofailover=true`) to ensure that more application modules are readily available, thereby increasing scalability. In this mode, passivation occurs at the end of every request.

To understand application module pooling and the "stateless with affinity" algorithm, read Section 44.1, "About Application Module Pooling" and the discussion of Referenced Pool Size in Section 44.2.7.2, "Pool Sizing Parameters."

Note: When running or debugging an application that uses failover support within the JDeveloper environment, you are frequently starting and stopping the application server. The ADF failover mechanism has no way of knowing whether you stopped the server to simulate an application server failure, or whether you stopped it because you want to retest something from scratch in a fresh server instance. If you intend to do the latter, exit out of your browser before restarting the application on the server. This eliminates the chance that you will be confused by the correct functioning of the failover mechanism when you didn't intend to be testing that aspect of your application.

43.2.2.3 About State Management Release Levels

When a data control handles the `endRequest` notification indicating the processing for the current HTTP request has completed, it releases the application module instance by checking it back into the application module pool. The application module pool manages instances and performs state management tasks (or not) based on the *release level* you use when returning the instance to the pool.

There are three release levels used for returning an instance of an application module to a pool:

- Managed - This is the default level, where the application module pool prefers to keep the same application module instance for the same data control, but may release an instance if necessary.
- Unmanaged - No state needs to be preserved beyond the current request.
- Reserved - A one-to-one relationship is preserved between an application module instance and a data control.

Caution: In general, it is strongly recommended never to use Reserved release level. You would normally avoid using this mode because the data control to application module correlation becomes one to one, the scalability of the application reduces very sharply, and so does reliability of the application.

43.2.2.3.1 About Managed Release Level This is the default release level and implies that application module's state is relevant and has to be preserved for this data control to

span over several HTTP requests. Managed level does not guarantee that for the next request this data control will receive the same physical application module instance, but it does guarantees that an application module with identical state will be provided so it is logically the same application module instance each time. It is important to note that the framework makes the best effort it can to provide the same instance of application module for the same data control if it is available at the moment. This is done for better performance since the same application module does not need to activate the previous state which it still has intact after servicing the same data control during previous request. However, the data control is not guaranteed to receive the same instance for all its requests and if the application module that serviced that data control during previous is busy or unavailable, then a different application module will activate this data control's state. For this reason, it is not valid to cache references to application module objects, view objects, or view rows across HTTP requests in controller-layer code.

This mode was called the "Stateful Release Mode" in previous releases of JDeveloper.

Note: If the `jbo.ampool.doampooling` configuration property is `false` — corresponding to your unchecking the **Enable Application Module Pooling** option in the **Pooling and Scalability** tab of the Business Components Configuration dialog — then there is effectively no pool. In this case, when the application module instance is released at the end of a request it is immediately removed. On subsequent requests made by the same user session, a new application module instance must be created to handle each user request, and pending state must be reactivated from the passivation store. Setting this property to `false` is useful to discover problems in your application logic that might occur when reactivation does occur due to unpredictable load on your system. However, the property `jbo.ampool.doampooling` set to `false` is not a supported configuration for production applications and must be set to `true` before you deploy your application. For further details, see Section 43.10, "Testing to Ensure Your Application Module is Activation-Safe."

43.2.2.3.2 About Unmanaged Release Level This mode implies that no state associated with this data control has to be preserved to survive beyond the current HTTP request. This level is the most efficient in performance because there is no overhead related to state management. However, you should limit its use to applications that require no state management, or to cases when state no longer needs to be preserved at this point. Usually, you can programmatically release the application module with the unmanaged level when you want to signal that the user has ended a logical unit of work.

Performance Tip: .The default release level is Managed, which implies that the application module's state is relevant and has to be preserved to allow the data control to span over several HTTP requests. Set release level to Unmanaged programmatically at runtime for particular pages to eliminate passivation and achieve better performance. A typical example is releasing the application module after servicing the HTTP request from a logout page.

This mode was called the "Stateless Release Mode" in previous releases of JDeveloper.

43.2.2.3.3 About Reserved Release Level This level guarantees that each data control will be assigned its own application module during its first request and for all subsequent requests coming from the `HttpSession` associated with this data control. This data control will always receive the same physical instance of application module. This mode exists for legacy compatibility reasons and for very rare special use cases.

An example of using Reserved level occurs when there is a pending database state across a request resulting from the `postChanges()` method or a PL/SQL stored procedure but not issuing a `commit()` or `rollback()` at the end of the request. In this case, if any other release level is used instead of Reserved, when the application module instance is recycled, a rollback is issued on the database connection associated with this application module instance and all uncommitted changes would be lost.

> **Performance Tip:** If you must use Reserved level, call `setReleaseLevel()` on the data control to keep its period as short as possible. For details about changing the release level programmatically, see Section 43.4, "Setting the Application Module Release Level at Runtime."

Consequences of Reserved mode can be adverse. Reliability suffers because if for whatever reason the application module is lost, the data control will not be able to receive any other application module in its place from the pool, and so `HttpSession` gets lost as well, which is not the case for managed level.

The failover option is ignored for an application module released with Reserved release level since its use implies your application absolutely requires working with the same application module instance on each request.

43.2.2.4 State Management and Subclassed Entity Objects

If your application employs subclassed entity objects, the key attribute of new entities must be prepopulated. If the key is not prepopulated, passivation and activation will fail. You can prepopulate the key attribute by overriding the `create()` method or by using the `DBSequence` type that will assign a temporary negative value to the key before the real value is fetched from the database after commit. For more information, see Section 4.10.10, "How to Get Trigger-Assigned Primary Key Values from a Database Sequence."

43.3 Using Save For Later

To enable Save For Later, you must first add Save Points to the application at points where you would like application state and data to be preserved if the end user leaves the application. You can use it to save data and state information about a region, view port, or portlet. Later, you use the Save Point Restore activity to restore application state and data associated with a Save Point.

For more information on how create and restore Save Points, see Section 22.6, "Using Save Points in Task Flows."

Save For Later can also perform implicit saves. These occur when data is saved automatically without the end user performing an explicit Save action when the user session times out or closes the browser window, for example.

For more information on how to perform an implicit save, see Section 22.6, "Using Save Points in Task Flows."

43.4 Setting the Application Module Release Level at Runtime

If you do not want to use the default "Managed State" release level for application modules, you can set your desired level programmatically.

43.4.1 How to Set Unmanaged Level

To set a data control to release its application module using the unmanaged level, call the `resetState()` method on the `DCDataControl` class (in the `oracle.adf.model.binding` package).

You can call this method any time during the request. This will cause application module not to passivate its state at all when it is released to the pool at the end of the request. Note that this method only affects the current application module instance in the current request. After this, the application module is released in unmanaged level to the pool, it becomes unreferenced and gets reset. The next time the application module is used by a client, it will be used in the managed level again by default.

Note: You can programmatically release the application module with the unmanaged level when you want to signal that the user has ended a logical unit of work. This will happen automatically when the `HTTPSession` times out, as described below.

43.4.2 How to Set Reserved Level

To set a data control to release its application module using the reserved level, call the `setReleaseLevel()` method of the `DCJboDataControl` class (in the `oracle.adf.model.bc4j` package), and pass the integer constant `ApplicationModule.RELEASE_LEVEL_RESERVED`.

When the release level for an application module has been changed to "Reserved" it will stay so for all subsequent requests until explicitly changed.

43.4.3 How to Set Managed Level

If you have set an application module to use reserved level, you can later set it back to use managed level by calling the `setReleaseLevel()` method of the `DCJboDataControl` class, and passing the integer constant `ApplicationModule.RELEASE_LEVEL_MANAGED`.

43.4.4 How to Set Release Level in a JSF Backing Bean

Example 43–1 shows calling the `resetState()` method on a data control named `UserModuleDataControl` from the action method of a JSF backing bean.

Example 43–1 Calling resetState() on Data Control in a JSF Backing Bean Action Method

```
package devguide.advanced.releasestateless.controller.backing;
import devguide.advanced.releasestateless.controller.JSFUtils;
import oracle.adf.model.BindingContext;
import oracle.adf.model.binding.DCDataControl;
/**
 * JSF Backing bean for the "Example.jspx" page
 */
public class Example {
  /**
   * In an action method, call resetState() on the data control to cause
```

```
   * it to release to the pool with the "unmanaged" release level.
   * In other words, as a stateless application module.
   */
  public String commandButton_action() {
    // Add event code here...
    getDataControl("UserModuleDataControl").resetState();
    return null;
  }
  private DCDataControl getDataControl(String name) {
    BindingContext bc =
      (BindingContext)JSFUtils.resolveExpression("#{data}");
    return bc.findDataControl(name);
  }
}
```

43.4.5 How to Set Release Level in an ADF PagePhaseListener

Example 43–2 shows calling the `resetState()` method on a data control named `UserModuleDataControl` from the after-prepareRender phase of the ADF lifecycle using a custom ADF page phase-listener class. You would associate this custom class to a particular page by setting the `ControllerClass` attribute on the page's page definition to the fully-qualified name of this class.

Example 43–2 Calling resetState() on Data Control in a Custom PagePhaseListener

```
package devguide.advanced.releasestateless.controller;
import oracle.adf.controller.v2.lifecycle.Lifecycle;
import oracle.adf.controller.v2.lifecycle.PagePhaseEvent;
import oracle.adf.controller.v2.lifecycle.PagePhaseListener;
import oracle.adf.model.binding.DCDataControl;
public class ReleaseStatelessPagePhaseListener
       implements PagePhaseListener {
  /**
   * In the "after" phase of the final "prepareRender" ADF Lifecycle
   * phase, call resetState() on the data control to cause it to release
   * to the pool with the "unmanaged" release level. In other words,
   * as a stateless application module.
   *
   * @param event ADF page phase event
   */
  public void afterPhase(PagePhaseEvent event) {
    if (event.getPhaseId() == Lifecycle.PREPARE_RENDER_ID) {
      getDataControl("UserModuleDataControl", event).resetState();
    }
  }
  // Required to implement the PagePhaseListener interface
  public void beforePhase(PagePhaseEvent event) {}
  private DCDataControl getDataControl(String name,
                                       PagePhaseEvent event) {
    return event.getLifecycleContext()
                .getBindingContext()
                .findDataControl(name);
  }
}
```

43.4.6 How to Set Release Level in an ADF PageController

Example 43–3 shows calling the `resetState()` method on a data control named `UserModuleDataControl` from an overridden `prepareRender()` method of a

custom ADF page controller class. You would associate this custom class to a particular page by setting the `ControllerClass` attribute on the page's page definition to the fully-qualified name of this class.

> **Note:** You can accomplish basically the same kinds of page-specific lifecycle customization tasks using a custom `PagePhaseListener` or a custom `PageController` class. The key difference is that the `PagePhaseListener` interface can be implemented on any class, while a custom `PageController` must extend the PageController class in the `oracle.adf.controller.v2.lifecycle` package.

Example 43–3 Calling resetState() on Data Control in a Custom ADF PageController

```
package devguide.advanced.releasestateless.controller;
import oracle.adf.controller.v2.context.LifecycleContext;
import oracle.adf.controller.v2.lifecycle.PageController;
import oracle.adf.controller.v2.lifecycle.PagePhaseEvent;
import oracle.adf.model.binding.DCDataControl;
public class ReleaseStatelessPageController extends PageController {
  /**
   * After calling the super in the final prepareRender() phase
   * of the ADF Lifecycle, call resetState() on the data control
   * to cause it to release to the pool with the "unmanaged"
   * release level. In other words, as a stateless application module.
   *
   * @param lcCtx ADF lifecycle context
   */
  public void prepareRender(LifecycleContext lcCtx) {
    super.prepareRender(lcCtx);
    getDataControl("UserModuleDataControl", lcCtx).resetState();
  }
  private DCDataControl getDataControl(String name,
                                       LifecycleContext lcCtx) {
    return lcCtx.getBindingContext().findDataControl(name);
  }
}
```

43.4.7 How to Set Release Level in a Custom ADF PageLifecycle

If you wanted to build a Fusion web application where every request was handled in a completely stateless way, use a global custom `PageLifecycle` class as shown in Example 43–4. For details on how to configure your application to use your custom lifecycle see Section 25.2, "About the JSF and ADF Page Lifecycles."

Example 43–4 Calling resetState() on Data Control in a Custom ADF PageLifecycle

```
package devguide.advanced.releasestateless.controller;
import oracle.adf.controller.faces.lifecycle.FacesPageLifecycle;
import oracle.adf.controller.v2.context.LifecycleContext;
import oracle.adf.model.binding.DCDataControl;
public class ReleaseStatelessPageLifecycle extends FacesPageLifecycle {
  /**
   * After calling the super in the final prepareRender() phase
   * of the ADF Lifecycle, call resetState() on the data control
   * to cause it to release to the pool with the "unmanaged"
   * release level. In other words, as a stateless application module.
   *
   * @param lcCtx ADF lifecycle context
```

```
   */
  public void prepareRender(LifecycleContext lcCtx) {
    super.prepareRender(lcCtx);
    getDataControl("UserModuleDataControl", lcCtx).resetState();
  }
  private DCDataControl getDataControl(String name,
                                       LifecycleContext lcCtx) {
    return lcCtx.getBindingContext().findDataControl(name);
  }
}
```

43.5 What Model State Is Saved and When It Is Cleaned Up

The information saved by application model passivation is divided in two parts: transactional and non-transactional state. Transactional state is the set of updates made to entity object data – performed either directly on entity objects or on entities through view object rows – that are intended to be saved into the database. Non-transactional state comprises view object runtime settings, such as the current row index, `WHERE` clause, and `ORDER BY` clause.

43.5.1 State Information Saved During Passivation

The information saved as part of the application module passivation "snapshot" includes the following.

Transactional State

- New, modified, and deleted entities in the entity caches of the root application module for this user session's (including old/new values for modified ones).

Non-Transactional State

- For each active view object (both statically and dynamically created):
  - Current row indicator for each row set (typically one)
  - New rows and their positions. (New rows are treated differently then updated ones. Their index in the view object is traced as well.)
  - ViewCriteria and all related parameters such as view criteria row, etc.
  - Flag indicating whether or not a row set has been executed
  - Range start and Range size
  - Access mode
  - Fetch mode and fetch size
  - Any view object-level custom data

  Note: Transient view object attributes can be saved if they are selected for passivation at design time. However, use this feature judiciously because this results in a snapshot that will grow in size with the number of rows that have been retrieved.

  - `SELECT`, `FROM`, `WHERE`, and `ORDER BY` clause if created dynamically or changed from the View definition

Note: If you enable ADF Business Components runtime diagnostics, the contents of each XML state snapshot are also saved. See Section 6.3.8, "How to Enable ADF Business Components Debug Diagnostics" for information on how to enable diagnostics.

43.5.2 Where the Model State Is Saved

By default, passivation snapshots are saved in the database, but you can configure it to use the file system as an alternative.

43.5.2.1 How Database-Backed Passivation Works

The passivated XML snapshot is written to a `BLOB` column in a table named `PS_TXN`, using a connection specified by the `jbo.server.internal_connection` property. Each time a passivation record is saved, it is assigned a unique passivation snapshot ID based on the sequence number taken from the `PS_TXN_SEQ` sequence. The ADF session cookie held by the application module data control in the ADF binding context remembers the latest passivation snapshot ID that was created on its behalf and remembers the previous ID that was used.

43.5.2.2 Controlling the Schema Where the State Management Table Resides

The ADF runtime recognizes a configuration property named `jbo.server.internal_connection` that controls which database connection and schema should be used for the creation of the `PS_TXN` table and the `PS_TXN_SEQ` sequence. If you don't set the value of this configuration parameter explicitly, then the state management facility creates the temporary tables using the credentials of the current application database connection.

To keep the temporary information separate, the state management facility uses a different connection *instance* from the database connection pool, but the database credentials are the same as the current user. Since the framework creates temporary tables, and possibly a sequence if they don't already exist, the implication of not setting a value for the `jbo.server.internal_connection` is that the current database user must have `CREATE TABLE`, `CREATE INDEX` and `CREATE SEQUENCE` privileges. Since this is often not desirable, Oracle recommends always supplying an appropriate value for the `jbo.server.internal_connection` property, providing the credentials for a state management schema where table and schema be created. Valid values for the `jbo.server.internal_connection` property in your configuration are:

- A fully-qualified JDBC connection URL like:

  `jdbc:oracle:thin:`*`username/password@host:port:SID`*

- A JDBC datasource name like:

  `java:/comp/env/jdbc/YourJavaEEDataSourceName`

  Performance Tip: When creating the `PS_TXN` table, use securefiles to store LOB data (the content column), and create a primary column index on the `PS_TXN` table as global, partitioned reverse key index. The securefile configuration delivers superior performance over the basicfile configuration when working with LOB data. The reverse key index helps by reducing contention that can happen when the rate of inserts is high.

43.5.2.3 Configuring the Type of Passivation Store

Passivated information can be stored in several places. You can control it programmatically or by configuring an option in the application module configuration. The choices are database or a file stored on local file system:

- **File**

  This choice may be the fastest available, because access to the file is faster then access to the database. This choice is good if the entire middle tier (one or multiple Oracle Application Server installation(s) and all their server instances) is either installed on the same machine or has access to a commonly shared file system, so passivated information is accessible to all. Usually, this choice may be good for a small middle tier where one Oracle Application Server is used. In other words this is a very suitable choice for small middle tier such as one Oracle Application Server with all its components installed on one physical machine. The location and name of the persistent snapshot files are determined by `jbo.tmpdir` property if specified. It follows usual rules of ADF property precedence for a configuration property. If nothing else is specified, then the location is determined by `user.dir` if specified. This is a default property and the property is OS specific.

- **Database**

  This is the *default* choice. While it may be a little slower than passivating to file, it is by far the most reliable choice. With passivation to file, the common problem might be that it is not accessible to Oracle Application Server instances that are remotely installed. In this case, in a cluster environment, if one node goes down the other may not be able to access passivated information and then failover will not work. Another possible problem is that even if file is accessible to the remote node, the access time for the local and remote node may be very different and performance will be inconsistent. With database access, time should be about the same for all nodes.

To set the value of your choice in design time, set the property `jbo.passivationstore` to `database` or `file`. The value `null` will indicate that a connection-type-specific default should be used. This will use database passivation for Oracle or DB2, and file serialization for any others.

To set the storage programmatically use the method `setStoreForPassiveState()` of interface `oracle.jbo.ApplicationModule`. The parameter values that you can pass are:

- `PASSIVATE_TO_DATABASE`
- `PASSIVATE_TO_FILE`

43.5.3 Cleaning Up the Model State

Under normal circumstances, the ADF state management facility provides automatic cleanup of the passivation snapshot records.

43.5.3.1 Previous Snapshot Removed When Next One Taken

When a passivation record is saved to the database on behalf of a session cookie, as described above, this passivation record gets a new, unique snapshot ID. The passivation record with the previous snapshot ID used by that same session cookie is deleted as part of the same transaction. In this way, assuming no server failures, there will only ever be a single passivation snapshot record per active end-user session.

43.5.3.2 Passivation Snapshot Removed on Unmanaged Release

The passivation snapshot record related to a session cookie is removed when the application module is checked into the pool with the unmanaged state level. This can occur when:

- Your code specifically calls `resetState()` on the application module data control.
- Your code explicitly invalidates the `HttpSession`, for example, as part of implementing an explicit "Logout" functionality.
- The `HttpSession` times out due to exceeding the session timeout threshold for idle time and failover mode is disabled (which is the default).

In each of these cases, the application module pool also resets the application module referenced by the session cookie to be "unreferenced" again. Since no changes were ever saved into the underlying database tables, once the pending session state snapshots are removed, there remains no trace of the unfinished work the user session had completed up to that point.

43.5.3.3 Passivation Snapshot Retained in Failover Mode

When the failover mode is enabled, if the `HttpSession` times out due to session inactivity, then the passivation snapshot is retained so that the end user can resume work upon returning to the browser.

After a break in the action, when the end user returns to his browser and continues to use the application, it continues working as if nothing had changed. The session cookie is used to reactivate any available application module instance with the user's last pending state snapshot before handling the request. So, even though the users next request will be processed in the context of a new `HttpSession` (perhaps even in a different application server instance), the user is unaware that this has occurred.

Note: If an application module was released with reserved level then the `HttpSession` times out, the user will have to go through authentication process, and all unsaved changes are lost.

43.5.4 Cleaning Up Temporary Storage Tables

JDeveloper supplies the `adfbc_purge_statesnapshots.sql` script to help with periodically cleaning up the application module state management table. You can find this file in the `oracle_common` subdirectory of your Oracle Middleware installation directory (for example, *`ORACLE_HOME`*`\oracle_common\common\sql`).

Persistent snapshot records can accumulate over time if the server has been shutdown in an abnormal way, such as might occur during development or due to a server failure. Running the script in SQL*Plus will create the `BC4J_CLEANUP` PL/SQL package. The two relevant procedures in this package are:

- `PROCEDURE Session_State(olderThan DATE)`

  This procedure cleans-up application module session state storage for sessions older than a given date.

- `PROCEDURE Session_State(olderThan_minutes INTEGER)`

  This procedures cleans-up application module session state storage for sessions older than a given number of minutes.

You can schedule periodic cleanup of your ADF temporary persistence storage by submitting an invocation of the appropriate procedure in this package as a database job.

You can use an anonymous PL/SQL block like the one shown in Example 43–5 to schedule the execution of `bc4j_cleanup.session_state()` to run starting tomorrow at 2:00 am and each day thereafter to cleanup sessions whose state is over 1 day (1440 minutes) old.

Example 43–5 Scheduling Periodic Cleanup of the State Management Table

```
SET SERVEROUTPUT ON
DECLARE
  jobId    BINARY_INTEGER;
  firstRun DATE;
BEGIN
  -- Start the job tomorrow at 2am
  firstRun := TO_DATE(TO_CHAR(SYSDATE+1,'DD-MON-YYYY')||' 02:00',
              'DD-MON-YYYY HH24:MI');
   -- Submit the job, indicating it should repeat once a day
  dbms_job.submit(job       => jobId,
                  -- Run the ADF Purge Script for Session State
                  -- to cleanup sessions older than 1 day (1440 minutes)
                  what      => 'bc4j_cleanup.session_state(1440);',
                  next_date => firstRun,
                  -- When completed, automatically reschedule
                  -- for 1 day later
                  interval  => 'SYSDATE + 1'
                 );
  dbms_output.put_line('Successfully submitted job. Job Id is '||jobId);
END;
.
/
```

43.6 Timing Out the HttpSession

Since HTTP is a stateless protocol, the server receives no implicit notice that a client has closed his browser or gone away for the weekend. Therefore any Java EE-compliant server provides a standard, configurable session timeout mechanism to allow resources tied to the HTTP session to be freed when the user has stopped performing requests. You can also programmatically force a timeout.

43.6.1 How to Configure the Implicit Timeout Due to User Inactivity

You configure the session timeout threshold using the <session-timeout> tag in the `web.xml` file. The default value is 35 minutes. When the `HttpSession` times out the `BindingContext` goes out of scope, and along with it, any data controls that might have referenced application modules released to the pool in the managed state level. The application module pool resets any of these referenced application modules and marks the instances unreferenced again.

43.6.2 How to Code an Explicit HttpSession Timeout

To end a user's session before the session timeout expires, you can call the `invalidate()` method on the `HttpSession` object from a backing bean in response to the user's click on a Logout button or link. This cleans up the `HttpSession` in the same way as if the session time had expired. Using JSF and ADF, after invalidating the session, you *must* perform a redirect to the next page you want to display, rather than

just doing a forward. Example 43–6 shows sample code to perform this task from a Logout button.

Example 43–6 Programatically Terminating a Session

```
public String logoutButton_action() throws IOException{
  ExternalContext ectx = FacesContext.getCurrentInstance().getExternalContext();
  HttpServletResponse response = (HttpServletResponse)ectx.getResponse();
  HttpSession session = (HttpSession)ectx.getSession(false);
  session.invalidate();
  response.sendRedirect("Welcome.jspx");
  return null;
}
```

As with the implicit timeouts, when the HTTP session is cleaned up this way, it ends up causing any referenced application modules to be marked unreferenced.

43.7 Managing Custom User-Specific Information

It is fairly common practice to add custom user-defined information in the application module in the form of member variables or some custom information stored in `oracle.jbo.Session` user data hashtable. The ADF state management facility provides a mechanism to save this custom information to the passivation snapshot as well, by overriding the `passivateState()` method and either the `activateState()` method or the `prepareForActivation()` method in the `ApplicationModuleImpl` class.

Note: Similar methods are available on the `ViewObjectImpl` class and the `EntityObjectImpl` class to save custom state for those objects to the passivation snapshot as well.

43.7.1 How to Passivate Custom User-Specific Information

You can override `passivateState()` and `activateState()` to ensure that custom application module state information is included in the passivation/activation cycle. Example 43–7 shows how this is done.

Note: The `activateState()` method is called at the end of the activation process after the view object have been activated. Most of the time this is where you want to place the application module state activation logic. However, if your application module activation logic needs to setup custom state information before the ADF statement management facility activates the view objects (for example, you might need to write custom code to allow the view objects to internally reference custom values at execution time), then the `prepareForActivation()` method in the `ApplicationModuleImpl` class would be the right place since it fires at the beginning of the activation process.

In the example, `jbo.counter` contains custom values you want to preserve across passivation and activation of the application module state. Each application module has an `oracle.jbo.Session` object associated with it that stores application module-specific session-level state. The session contains a user data hashtable where you can store transient information. For the user-specific data to "survive" across

application module passivation and reactivation, you need to write code to save and restore this custom value into the application module state passivation snapshot.

Example 43–7 Passivating and Activating Custom Information in the State Snapshot XML Document

```
/**
 * Overridden framework method to passivate custom XML elements
 * into the pending state snapshot document
 */
public void passivateState(Document doc, Element parent) {
  // 1. Retrieve the value of the value to save
  int counterValue = getCounterValue();
  // 2. Create an XML element to contain the value
  Node node = doc.createElement(COUNTER);
  // 3. Create an XML text node to represent the value
  Node cNode = doc.createTextNode(Integer.toString(counterValue));
  // 4. Append the text node as a child of the element
  node.appendChild(cNode);
  // 5. Append the element to the parent element passed in
  parent.appendChild(node);
}
/**
 * Overridden framework method to activate  custom XML elements
 * into the pending state snapshot document
 */
public void activateState(Element elem) {
  super.activateState(elem);
  if (elem != null) {
    // 1. Search the element for any <jbo.counter> elements
    NodeList nl = elem.getElementsByTagName(COUNTER);
    if (nl != null) {
      // 2. If any found, loop over the nodes found
      for (int i=0, length = nl.getLength(); i < length; i++) {
        // 3. Get first child node of the <jbo.counter> element
        Node child = nl.item(i).getFirstChild();
        if (child != null) {
          // 4. Set the counter value to the activated value
          setCounterValue(new Integer(child.getNodeValue()).intValue()+1);
          break;
        }
      }
    }
  }
}
/*
 * Helper Methods
 */
private int getCounterValue() {
  String counterValue = (String)getSession().getUserData().get(COUNTER);
  return counterValue == null ? 0 : Integer.parseInt(counterValue);
}
private void setCounterValue(int i) {
  getSession().getUserData().put(COUNTER,Integer.toString(i));
}
private static final String COUNTER = "jbo.counter";
```

43.7.1.1 What Happens When You Passivate Custom Information

In Example 43–7, when `activateState()` is overridden, the following steps are performed:

1. Search the element for any `jbo.counter` elements.
2. If any are found, loop over the nodes found in the node list.
3. Get first child node of the `jbo.counter` element.

   It should be a DOM Text node whose value is the string you saved when your `passivateState()` method above got called, representing the value of the `jbo.counter` attribute.
4. Set the counter value to the activated value from the snapshot.

When `passivateState()` is overridden, it performs the reverse job by doing the following:

1. Retrieve the value of the value to save.
2. Create an XML element to contain the value.
3. Create an XML text node to represent the value.
4. Append the text node as a child of the element.
5. Append the element to the parent element passed in.

Note: The API's used to manipulate nodes in an XML document are provided by the Document Object Model (DOM) interfaces in the `org.w3c.dom` package. These are part of the Java API for XML Processing (JAXP). See the Javadoc for the `Node`, `Element`, `Text`, `Document`, and `NodeList` interfaces in this package for more details.

43.8 Managing the State of View Objects

By default, all view objects are marked as passivation-enabled, so their state will be saved. However, view objects that have transient attributes do not have those attributes passivated by default. You can change how a view object is passivated, and even which attributes are passivated, using the Tuning page of the view object overview editor.

43.8.1 How to Manage the State of View Objects

Each view object can be declaratively configured to be passivation-enabled or not. If a view object is not passivation enabled, then no information about it gets written in the application module passivation snapshot.

> **Performance Tip:** There is no need to passivate read-only view objects since they are not intended to be updated and are easily recreated from the XML definition. This eliminates the performance overhead associated with passivation and activation and reduces the CPU usage needed to maintain the application module pool.

To set the passivation state of a view object:

1. In the Application Navigator, double-click a view object to open it in the overview editor.
2. On the General page, expand the **Tuning** section.

3. Select **Passivate State** to make sure the view object data is saved.

   Optionally, you can select **Including All Transient Attributes** to passivate all transient attributes at this time, but see Section 43.8.4, "What You May Need to Know About Passivating Transient View Objects" for additional information.

43.8.2 What You May Need to Know About Passivating View Objects

The activation mechanism is designed to return your view object to the state it was in when the last passivation occurred. To ensure that, Oracle ADF stores in the state snapshot the values of any bind variables that were used for the last query execution. These bind variables are in addition to those that are set on the row set at the time of passivation. This approach works for an application that does not dynamically reset the WHERE clause and bind variables on each request. In this case, the set of bind variable values used for the last `executeQuery()` and the set of bind variables current on the row set at passivation time are the same. The passivated state also stores the user-supplied WHERE clause on the view object related to the row set at the time of passivation.

However, when your application needs to dynamically change WHERE clauses and corresponding bind variables during the span of a single HTTP request, you need to ensure that the user-defined WHERE clause on the view object at the time of passivation matches the set of bind variable values used the last time the view object was executed before passivation. In this case, this is the correct sequence to follow:

1. (Request begins and AM is acquired.)
2. Call `setWhereClause()` on a view object instance that references *n* bind variables.
3. Call `setWhereClauseParam()` to set the *n* values for those *n* bind variables.
4. Call `executeQuery()`.
5. Call `setWhereClause(null)` to clear WHERE clause.
6. Call `setWhereClauseParams(null)` to clear the WHERE clause bind variables.
7. (AM is released.)

If you do not adhere to this strategy of changing runtime view object settings before using them, your application will fail with an SQL exception during application module pooling state activation:

```
JBO-27122: SQLStmtException: <... SQL Statement ...>
```

Because many of the view object's instance settings are saved in the passivation state snapshot and restored on activation (as described in Section 43.5.1, "State Information Saved During Passivation"), it is not advisable to change any of these settings just after executing the view object if you won't be re-executing the view object again during the same block of code (and so, during the same HTTP request). Instead, change the view object instance settings the next time you need them to be different before executing the query.

If you are dynamically adding named `WHERE` clause parameters to your view object instances, you might find it useful to add the following helper method to your `ViewObjectImpl` framework extension class. This method removes named bind variables that have been added to the view instance at runtime, without removing the ones that have been declaratively defined on the view definition at design time.

```
protected void clearWhereState() {
   ViewDefImpl viewDef = getViewDef();
```

```
   Variable[] viewInstanceVars = null;
   VariableManager viewInstanceVarMgr = ensureVariableManager();
   if (viewInstanceVarMgr != null) {
      viewInstanceVars = viewInstanceVarMgr.getVariablesOfKind(Variable.VAR_KIND_
WHERE_CLAUSE_PARAM);
      if (viewInstanceVars != null) {
         for (Variable v: viewInstanceVars) {
            // only remove the variable if its not on the view def.
            if (!hasViewDefVariableNamed(v.getName())) {
              removeNamedWhereClauseParam(v.getName());
            }
         }
      }
   }
   getDefaultRowSet().setExecuteParameters(null, null, true);
   setWhereClause(null);
   getDefaultRowSet().setWhereClauseParams(null);
}

private boolean hasViewDefVariableNamed(String name) {
   boolean ret = false;
   VariableManager viewDefVarMgr = getViewDef().ensureVariableManager();
   if (viewDefVarMgr != null) {
      try {
         ret = viewDefVarMgr.findVariable(name) != null;
      }
      catch (NoDefException ex) {
         // ignore
      }
   }
   return ret;
}
```

43.8.3 How to Manage the State of Transient View Objects and Attributes

Because view objects are marked as passivated by default, a transient view object — one that contains only transient attributes — is marked to be passivation enabled, but only passivates its information related to the current row and other non-transactional state.

> **Performance Tip:** Transient view object attributes are not passivated by default. Due to their nature, they are usually intended to be read-only and are easily recreated. So, it often doesn't make sense to passivate their values as part of the XML snapshot. This also avoids the performance overhead associated with passivation and activation and reduces the CPU usage needed to maintain the application module pool.

To individually set the passivation state for transient view object attributes:

1. In the Application Navigator, double-click a view object to open it in the overview editor.
2. On the Attributes page, select the transient attribute you want to passivate and then click the **Details** tab.
3. In the Details section, select the **Passivate** checkbox.

43.8.4 What You May Need to Know About Passivating Transient View Objects

Passivating transient view object attributes is more costly resource-wise and performance- wise, because transactional functionality is usually managed on the entity object level. Since transient view objects are not based on an entity object, this means that all updates are managed in the view object row cache and not in the entity cache. Therefore, passivating transient view objects or transient view object attributes requires special runtime handling.

Usually passivation only saves the values that have been changed, but with transient view objects passivation has to save entire row. The row will include only the view object attributes marked for passivation.

43.8.5 How to Use Transient View Objects to Store Session-level Global Variables

Using passivation, you can use a view object to store one or more global variables, each on a different transient attribute. When you mark a transient attribute as passivated, the ADF Business Components framework will remember the transient values across passivation and activation in high-throughput and failover scenarios. Therefore, it is an easy way to implement a session-level global value that is backed up by the state management mechanism, instead of the less-efficient HTTP Session replication. This also makes it easy to bind to controls in the UI if necessary.

There are two basic approaches to store values between invocations of different screens, one is controller-centric, and the other is model-centric.

Implementation of the task in the ADF controller

The controller-centric approach involves storing and referencing values using attributes in the pageFlow scope. This approach might be appropriate if the global values do not need to be referenced internally by any implementations of ADF Business Components.

For more information about pageFlow scope, see Section 18.2.4, "What You May Need to Know About Memory Scope for Task Flows."

Implementation of the task in the ADF model

The model-centric approach involves creating a transient view object, which is conceptually equivalent to a non-database block in Oracle Forms.

1. Create a new view object using the View Object Wizard, as described in Section 5.2.1, "How to Create an Entity-Based View Object."
   - On step 1 of the wizard, select the option for **Rows populated programmatically, not based on a query**.
   - On step 2, click **New** to define the transient attribute names and types the view object should contain. Make sure to set the **Updateable** option to **Always**.
   - Click **Finish** and the newly-created view object appears in the overview editor.
2. Disable any queries from being performed in the view object.
   - On the General page of the overview editor, expand the **Tuning** section, and in the **Retrieve from Database** group box, select the **No Rows** option.
3. Make sure data in the view object is not cleared out during a rollback operation. To implement this, you enable a custom Java class for the view object and override two rollback methods.

- On the Java page of the overview editor, click the **Edit** icon in the **Java Classes** section to open the Java dialog.
- In the Java dialog, select **Generate View Object Class** and click **OK**.
- In the overview editor, click on the hyperlink next to the View Object Class in the **Java Classes** section to open the source editor.
- From the **Source** menu, choose **Override Methods**.
- In the Override Methods dialog, select the `beforeRollback()` and `afterRollback()` methods to override, and click then **OK**.
- In both the `beforeRollback()` and `afterRollback()` methods, comment out the call to `super` in the Java code.

4. Add an instance of the transient view object to your application module's data model, as described in Section 9.2.3.2, "Adding Master-Detail View Object Instances to an Application Module."
5. Create an empty row in the view object when a new user begins using the application module.
   - Enable a Java class for your application module if you don't have one yet.
   - Override the `prepareSession()` method of the application module, as described in Section 9.11.1, "How to Override a Built-in Framework Method."
   - After the call to `super.prepareSession()`, add code to create a new row in the transient view object and insert it into the view object.

Now you can bind read-only and updateable UI elements to the "global" view object attributes just as with any other view object using the Data Controls panel.

43.9 Using State Management for Middle-Tier Savepoints

In the database server you are likely familiar with the savepoint feature that allows a developer to rollback to a certain point within a transaction instead of rolling back the entire transaction. An application module offers the same feature but implemented in the middle tier.

Best Practice: Oracle ADF provides a declarative approach to working with savepoints, described in Section 22.6, "Using Save Points in Task Flows." Use the programmatic approach described in Section 43.9.1, "How to Use State Management for Savepoints" only if the declarative approach doesn't meet your needs.

43.9.1 How to Use State Management for Savepoints

To use state management for implementing middle-tier savepoints, you override three methods in the `oracle.jbo.ApplicationModule` interface

```
public String passivateStateForUndo(String id,byte[] clientData,int flags)
public byte[] activateStateForUndo(String id,int flags)
public boolean isValidIdForUndo(String id)
```

You can use these methods to create a stack of named snapshots and restore the pending transaction state from them by name. Keep in mind that those snapshots do not survive past duration of transaction (for example, events of commit or rollback). This feature could be used to develop complex capabilities of the application, such as

the ability to undo and redo changes. Another ambitious goal that could exploit this functionality would be functionality to make the browser back and forward buttons behave in an application-specific way. Otherwise, simple uses of these methods can come quite in handy.

43.10 Testing to Ensure Your Application Module is Activation-Safe

If you have not *explicitly* tested that your application module functions when its pending state gets activated from a passivation snapshot, then you may encounter an unpleasant surprise in your production environment when heavy system load tests this aspect of your system for the first time.

43.10.1 Understanding the jbo.ampool.doampooling Configuration Parameter

The `jbo.ampool.doampooling` configuration property corresponds to the **Enable Application Module Pooling** option in the **Pooling and Scalability** tab of the Business Components Configuration dialog. By default, this checkbox is checked so that application module pooling is enabled. Whenever you deploy your application in a production environment the default setting of `jbo.ampool.doampooling` to `true` is the way you will run your application. But, as long as you run your application in a test environment, setting the property to `false` can play an important role in your testing. When this property is false, there is effectively no application pool. When the application module instance is released at the end of a request it is immediately removed. On subsequent requests made by the same user session, a new application module instance must be created to handle it and the pending state of the application module must be reactivated from the passivation store.

43.10.2 Disabling Application Module Pooling to Test Activation

As part of your overall testing plan, you should adopt the practice of testing your application modules with the `jbo.ampool.doampooling` configuration parameter set to `false`. This setting completely disables application module pooling and forces the system to activate your application module's pending state from a passivation snapshot on *each* page request. It is an excellent way to detect problems that might occur in your production environment due to assumptions made in your custom application code.

> **Caution:** It is important to reenable application module pooling after you conclude testing and are ready to deploy the application to a production environment. The configuration property `jbo.ampool.doampooling` set to `false` is not a supported configuration for production applications and must be set to `true` before deploying the application.

For example, if you have transient view object attributes you believe *should* be getting passivated, this technique allows you to test that they are working as you expect. In addition, consider situations where you might have introduced:

- Private member fields in application modules, view objects, or entity objects
- Custom user session state in the `Session` user data hashtable

Your custom code likely assumes that this custom state will be maintained across HTTP requests. As long as you test with a single user on the JDeveloper Integrated WebLogic Server, or test with a small number of users, things will appear to work fine.

This is due to the "stateless with affinity" optimization of the ADF application module pool. If system load allows, the pool will continue to return the same application module instance to a user on subsequent requests. However, under heavier load, during real-world use, it may not be able to achieve this optimization and will need to resort to grabbing any available application module instance and reactivating its pending state from a passivation snapshot. If you have not correctly overridden `passivateState()` and `activateState()` (as described in Section 43.7, "Managing Custom User-Specific Information") to save and reload your custom component state to the passivation snapshot, then your custom state will be missing (i.e. `null` or back to your default values) after this reactivation step. Testing with `jbo.ampool.doampooling` set to false allows you to quickly isolate these kinds of situations in your code.

43.11 Keeping Pending Changes in the Middle Tier

The ADF state management mechanism relies on passivation and activation to manage the state of an application module instance. Implementing this feature in a robust way is only possible if all pending changes are managed by the application module transaction in the middle tier. The most scalable strategy is to keep pending changes in middle-tier objects and not perform operations that cause pending database state to exist across HTTP requests. This allows the highest leverage of the performance optimizations offered by the application module pool and the most robust runtime behavior for your application.

When the `jbo.doconnectionpooling` configuration parameter is set to `true` — typically in order to share a common pool of database connections across multiple application module pools — upon releasing your application module to the application module pool, its JDBC connection is released back to the database connection pool and a `ROLLBACK` will be issued on that connection. This implies that all changes which were posted but *not commited* will be lost. On the next request, when the application module is used, it will receive a JDBC connection from the pool, which may be a different JDBC connection instance from the one it used previously. Those changes that were posted to the database but not commited during the previous request are lost.

Caution: When the `jbo.doconnectionpooling` configuration parameter is set to `true` — typically in order to share a common pool of database connections across multiple application module pools — upon releasing your application module to the application module pool, its JDBC connection is released back to the database connection pool and a `ROLLBACK` will be issued on that connection. This implies that all changes which were posted but *not commited* will be lost. On the next request, when the application module is used, it will receive a JDBC connection from the pool, which may be a different JDBC connection instance from the one it used previously. Those changes that were posted to the database but not commited during the previous request are lost.

The `jbo.doconnectionpooling` configuration parameter is set by checking the **Disconnect Application Module Upon Release** property on the **Pooling and Scalability** tab of the Business Components Configuration dialog.

43.11.1 How to Confirm That Applications Use Optimistic Locking

Oracle recommends using optimistic locking, the default mode for web applications. Pessimistic locking should not be used for web applications as it creates pending transactional state in the database in the form of row-level locks. If pessimistic locking is set, state management will work, but the locking mode will not perform as expected. Behind the scenes, every time an application module is recycled, a rollback is issued in the JDBC connection. This releases all the locks that pessimistic locking had created.

> **Performance Tip:** Always use the default mode optimistic locking for web applications. Only optimistic locking is compatible with the application module unmanaged release level mode, which allows the application module instance to be immediately released when a web page terminates. This provides the best level of performance for web applications that expect many users to access the application simultaneously.

To ensure your configuration uses optimistic locking, open the **Properties** tab of the Business Components Configuration dialog and confirm that the value of the `jbo.locking.mode` property is set to `optimistic` or `optupdate`.

> **Note:** To open the Business Components Configuration dialog, right-click the application module in the Application Navigator, and choose **Configurations** from the context menu. Then, in the Manage Configurations dialog, select the configuration you want to edit and click **Edit**.

Optimistic locking (`optimistic`) issues a `SELECT FOR UPDATE` statement to lock the row, then detects whether the row has been changed by another user by comparing the change indicator attribute — or, if no change indicator is specified, the values of all the persistent attributes of the current entity as they existed when the entity object was fetched into the cache.

Optimistic update locking (`optupdate`) does not perform any locking. The `UPDATE` statement determines whether the row was updated by another user by including a `WHERE` clause that will match the existing row to update only if the attribute values are unchanged since the current entity object was fetched.

43.11.2 How to Avoid Clashes Using the postChanges() Method

The transaction-level `postChanges()` method exists to force the transaction to post unvalidated changes without committing them. This method is not recommended for use in web applications unless you can guarantee that the transaction will definitely be committed or rolled-back during the same HTTP request. Failure to heed this advice can lead to strange results in an environment where both application modules and database connections can be pooled and shared serially by multiple different clients.

43.11.3 How to Use the Reserved Level For Pending Database States

If for some reason you need to create a transactional state in the database in some request by invoking `postChanges()` method or by calling PL/SQL stored procedure, but you cannot issue a commit or rollback by the end of that same request, then you *must* release the application module instance with the **reserved** level from that request until a subsequent request when you either commit or rollback.

> **Performance Tip:** Use as short a period of time as possible between creation of transactional state in the database and performing the concluding commit or rollback. This ensures that reserved level doesn't have to be used for a long time, as it has adverse effects on application's scalability and reliability.

Once an application module has been released with reserved level, it remains at that release level for all subsequent requests until release level is explicitly changed back to managed or unmanaged level. So, it is your responsibility to set release level back to managed level once commit or rollback has been issued.

For more information, see Section 43.4, "Setting the Application Module Release Level at Runtime."

44

Tuning Application Module Pools and Connection Pools

This chapter describes how ADF Business Components application module pools work and how you can tune both application module pools and database connection pools to optimize application performance.

This chapter includes the following sections:

- Section 44.1, "About Application Module Pooling"
- Section 44.2, "Setting Pool Configuration Parameters"
- Section 44.3, "Initializing Database State and Pooling Considerations"

44.1 About Application Module Pooling

An application module pool is a collection application module instances of the same type. For example, the Fusion Order Demo application has one or more instances of the application module in it, based on the number of users that are visiting the site. This pool of application module instances is shared by multiple browser clients whose typical "think time" between submitting web pages allows optimizing the number of application module components to be effectively smaller than the total number of active users working on the system. That is, twenty users visiting the web site from their browser might be able to be serviced by 5 or 10 application module instances instead of having as many application module instances as you have browser users.

Application module components can be used to support Fusion web application scenarios that are completely stateless, or they can be used to support a unit of work that spans multiple browser pages. As a performance optimization, when an instance of an application module is returned to the pool in "managed state" mode, the pool tracks session references to the application module. The application module instance is still in the pool and available for use, but it would *prefer* to be used by the same session that was using it the last time because maintaining this so-called "session affinity" improves performance.

So, at any one moment in time, the instances of application modules in the pool are logically partitioned into three groups, reflecting their state:

- Unconditionally *available* for use
- Available for use, but *referenced* for session affinity reuse by an active user session
- *Unavailable*, inasmuch as it's currently in use (at that very moment) by some thread in the web container.

Section 44.2.5, "What You May Need to Know About Configuration Property Scopes" describes the application module pool configuration parameters and how they affect the behavior of the pool.

44.1.1 Types of Pools Created When Running the Fusion Web Application

There are two kinds of pools in use when running a typical Fusion web application, Application Module pools and database connection pools. It's important to understand how many of each kind of pool your application will create.

44.1.1.1 Application Module Pools

Application Module components can be used at runtime in two ways:

- As an application module the client accesses directly
- As a reusable component aggregated (or "nested") inside of another application module instance

When a client accesses it directly, an application module is called a *root application module*. Clients access nested application modules *indirectly* as a part of their containing application module instance. It's possible, but not common, to use the same application module at runtime in both ways. The important point is that ADF only creates an application module pool for a *root* application module.

The basic rule is that one application module pool is created for each *root* application module used by a Fusion web application in *each* Java VM where a root application module of that type is used by the ADF controller layer.

44.1.1.2 Database Connection Pools

The type of database connection pool the Fusion web application uses depends on the connection type that you configure for your application modules:

- JDBC URL (e.g. `jdbc:oracle:thin:@penguin:1521:ORCL`)
- JNDI name for a data source (e.g. `java:comp/env/jdbc/YourConnectionDS`)

If you supply a JDBC URL connection while configuring your application module — which happens when you select a JDeveloper named connection which encapsulates the JDBC URL and username information — then the ADF database connection pool will be used for managing the connection pool.

If you supply the JNDI name of a JDBC data source then the ADF database connection pool will *not* be used and the configuration parameters described below relating to the ADF database connection pool are not relevant.

> **Note:** To configure the database connection pool for JDBC data sources looked-up by JNDI from your Java EE web and/or EJB container, consult the documentation for your Java EE container to understand the pooling configuration options and how to set them.

When using ADF database connection pooling, you have the following basic rule: One database connection pool is created for each unique `<JDBCURL,Username>` pair, in *each* Java VM where a `<JDBCURL,Username>` connection is requested by a root application used by the ADF controller layer.

44.1.2 Understanding Application Module and Connection Pools

The number of pools and the type of pools that your application will utilize will depend upon how the target platform is configured. For example, will there be more than one Java Virtual Machine (JVM) available to service the web requests coming from your application users and will there be more than one Oracle Application Server instance? To understand how many pools of which kinds are created for an application in both a single-JVM scenario and a multiple-JVM runtime scenario, review the following assumptions:

- Your Fusion web application makes use of two application modules `HRModule` and `PayablesModule`.
- You have a `CommonLOVModule` containing a set of commonly used view objects to support list of values in your application, and that both `HRModule` and `PayablesModule` aggregate a nested instance of `CommonLOVModule` to access the common LOV view objects it contains.
- You have configured both `HRModule` and `PayablesModule` to use the same JDeveloper connection definition named `appuser`.
- In both `HRModule` and `PayablesModule` you have configured `jbo.passivationstore=database` (the default) and configured the ADF "internal connection" (`jbo.server.internal_connection`) used for state management persistence to have the value of a fully-qualified JDBC URL that points to a *different* username than the `appuser` connection does.

44.1.2.1 Single Oracle Application Server Instance, Single Oracle WebLogic Server Instance, Single JVM

If you deploy this application to a single Oracle Application Server instance, configured with a single Oracle WebLogic Server instance having a single Java VM, there is only a single Java VM available to service the web requests coming from your application users.

Assuming that all the users are making use of web pages that access both the `HRModule` and the `PayablesModule`, this will give:

- One application module pool for the `HRModule` root application module
- One application module pool for the `PayablesModule` root application module
- One DB connection pool for the `appuser` connection
- One DB connection pool for the JDBC URL supplied for the internal connection for state management.

This gives a total of two application module pools and two database pools in this single Java VM.

> **Note:** There is no separate application module pool for the nested instances of the reusable `CommonLOVModule`. Instances of `CommonLOVModule` are wrapped by instances of `HRModule` and `PayablesModule` in their respective application module pools.

44.1.2.2 Multiple Oracle Application Server Instances, Single Oracle WebLogic Server Instance, Multiple JVMs

Next consider a deployment environment involving multiple Java VMs. Assume that you have installed Oracle Application Server onto two different physical machines,

with a hardware load-balancer in front of it. On each of these two machines, imagine that the Oracle Application Server instance is configured to have one Oracle WebLogic Server instance with two JVMs. As users of your application access the application, their requests are shared across these two Oracle Application Server instances, and within each Oracle Application Server instance, across the two JVMs that its Oracle WebLogic Server instance has available.

Again assuming that all the users are making use of web pages that access both the `HRModule` and the `PayablesModule`, this will give:

- Four application module pools for `HRModule`, one in each of four JVMs.

  (1 `HRModule` root application module) x (2 Oracle Application Server Instances) x (2 Oracle WebLogic Server JVMs each)

- Four application module pools for `PayablesModule`, one in each of four JVMs.

  (1 `PayablesModule` root application module) x (2 Oracle Application Server Instances) x (2 Oracle WebLogic Server JVMs each)

- Four DB connection pools for `appuser`, one in each of four JVMs.

  (1 `appuser` DB connection pool) x (2 Oracle Application Server Instances) x (2 Oracle WebLogic Server JVMs each)

- Four DB connection pools for the internal connection JDBC URL, one in each of four JVMs.

  (1 internal connection JDBC URL DB connection pool) x (2 Oracle Application Server Instances) x (2 Oracle WebLogic Server JVMs each)

This gives a total of eight application module pools and eight DB connection pools spread across four JVMs.

As you begin to explore the configuration parameters for the application module pools in Section 44.2.7, "What You May Need to Know About Application Module Pool Parameters," keep in mind that the parameters apply to a given application module pool for a given application module in a single JVM.

As the load balancing spreads user requests across the multiple JVMs where ADF is running, each individual application module pool in each JVM will have to support one n^{th} of the user load — where n is the number of JVMs available to service those user requests. The appropriate values of the application module and DB connection pools need to be set with the number of Java VMs in mind. The basic approach is to base sizing parameters on load testing and the results of the application module pooling statistics, then divide that total number by the n number of pools you will have based on your use of multiple application servers and multiple Oracle WebLogic Server instances. For example, if you decide to set the minimum number of application modules in the pool to ten and you end up with five pools due to having five Oracle WebLogic Server instances servicing this application, then you would want to configure the parameter to 2 (ten divided by five), not 10 (which would only serve a given application module in a single JVM).

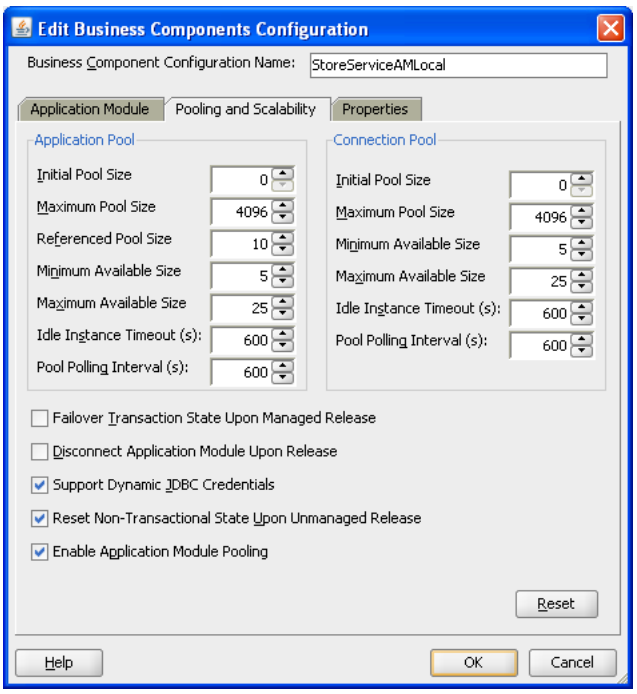
44.2 Setting Pool Configuration Parameters

You control the runtime behavior of an application module pool by setting appropriate configuration parameters. You can set these declaratively in an application module configuration, supply them as Java System parameters, or set them programmatically at runtime.

44.2.1 How to Set Configuration Properties Declaratively

The **Pooling and Scalability** tab of the Edit Business Components Configuration dialog shown in Figure 44–1 is used for viewing and setting parameters.

Figure 44–1 Pooling and Scalability Tab of the Configuration Manager



To edit your application module's pooling configuration:

1. In the Application Navigator, double-click the application module.
2. In the overview editor, click the **Configurations** navigation tab.
3. In the Configurations page, double-click the configuration you want to edit.
4. In the Edit Business Components Configuration dialog, click the **Pooling and Scalability** tab and edit the desired runtime properties and click **OK** to save the changes for your configuration.

44.2.2 What Happens When You Set Configuration Properties Declaratively

The values that you supply through the Configuration Manager are saved in an XML file named `bc4j.xcfg` in the `./common` subdirectory relative to the application module's XML component definition. All of the configurations for all of the application modules in a single Java package are saved in that same file.

For example, if you look at the `bc4j.xcfg` file in the `.src/oracle/fodemo/storefront/store/service/common` directory of the Fusion Order Demo application's `StoreFront` project, you will see the three named configurations for its `StoreServiceAM` application module, as shown in Example 44–1. In this case, The `StoreServiceAMLocal` and the `StoreServiceAMLocalWeb` configurations specify JDBC URL connections for use by the Oracle ADF Model Tester. The connection details for the JDBC connections appear in the `connections.xml` file located in the `./.adf/META-INF` subdirectory relative to the project directory. The third configuration `StoreFrontService` specifies a JDBC data source name and is used by the Fusion web application. This

type of configuration is generated by default when you expose a service interface for the application module in the Service Interface page of the overview editor.

Example 44–1 Configuration Settings for the StoreService Application Module

```
<BC4JConfig version="11.1" xmlns="http://xmlns.oracle.com/bc4j/configuration">
   <AppModuleConfigBag ApplicationName="oracle.fodemo.storefront.store.service.StoreServiceAM">
      <AppModuleConfig
            DeployPlatform="LOCAL"
            JDBCName="FOD"
            jbo.project="StoreFrontService"
            name="StoreServiceAMLocal"
            ApplicationName="oracle.fodemo.storefront.store.service.StoreServiceAM">
         <Database jbo.locking.mode="optimistic"/>
         <Security AppModuleJndiName="oracle.fodemo.storefront.store.service.StoreServiceAM"/>
      </AppModuleConfig>
      <AppModuleConfig
               DeployPlatform="LOCAL"
               JDBCName="FOD"
               jbo.project="StoreFrontService"
               name="StoreServiceAMLocalWeb"
               ApplicationName="oracle.fodemo.storefront.store.service.StoreServiceAM">
         <AM-Pooling jbo.ampool.initpoolsize="1"/>
         <Database jbo.locking.mode="optimistic"/>
         <Security AppModuleJndiName="oracle.fodemo.storefront.store.service.StoreServiceAM"/>
         <Custom fod.application.issoaenabled="true"/>
      </AppModuleConfig>
      <AppModuleConfig
               name="StoreFrontService"
               ApplicationName="oracle.fodemo.storefront.store.service.StoreServiceAM"
               jbo.project="StoreFrontService"
               DeployPlatform="SI">
         <AM-Pooling jbo.ampool.resetnontransactionalstate="true"/>
         <Database jbo.SQLBuilder="ORACLE" jbo.locking.mode="optimistic"
                   jbo.TypeMapEntries="Java"/>
         <Security AppModuleJndiName="oracle.fodemo.storefront.store.service.StoreServiceAM"/>
         <Custom JDBCDataSource="java:comp/env/jdbc/FODDS"/>
      </AppModuleConfig>
   </AppModuleConfigBag>
</BC4JConfig>
```

Note that attributes of child elements of the `<AppModuleConfig>` tag have names beginning with `jbo` that match the names of their ADF Business Components properties (for example, the `<Database>` tag defines the attribute `jbo.locking.mode` that corresponds to the property `jbo.locking.mode`). It's also important to understand that if a property is currently set to its runtime *default* value in the Edit Business Components Configuration dialog, then JDeveloper does *not* write the entry to the `bc4j.xcfg` file.

44.2.3 How to Set Configuration Properties as System Parameters

As an alternative to specifying configuration properties in the `bc4j.xcfg` file, you can also set Java VM system parameters with the same property names. These system parameters will be used *only if* a corresponding property *does not exist* in the relevant `bc4j.xcfg` file for the application module in question. In other words, configuration parameters that appear in the application module configuration take precedence over parameters of the same name supplied as Java system parameters.

You typically set Java system parameters using the `-D` command line flag to the Java VM like this:

```
java -Dproperty=value -jar yourserver.jar
```

Alternatively, your Java EE container probably has a section in its own configuration files where Java system parameters can be specified for use at Java EE container startup time.

If you adopt the technique of specifying site-specific *default* values for Oracle Application Development Framework (Oracle ADF) configuration parameters as Java system parameters, you should make sure that your application's `bc4j.xcfg` files do *not* include references to these parameters unless you want to define an application-module-specific exception to these global default values.

Caution: The values of **Idle Instance Timeout**, **Pool Polling Interval** settings for both the **Application Pool** and the database **Connection Pool** are displayed and edited in this dialog as a number of *seconds*, but are saved to the configuration file in milliseconds. If you provide a value for any of these four parameters as a Java System parameter — or if you hand-edit the `bc4j.xcfg` file — make sure to provide these time interval values in milliseconds!

44.2.4 How to Programmatically Set Configuration Properties

You can set configuration properties programmatically by creating a Java class that implements the `EnvInfoProvider` interface in the `oracle.jbo.common.ampool` package. In your class, you override the `getInfo()` method and call `put()` to put values into the environment Hashtable passed in as shown in Example 44–2.

Example 44–2 Setting Environment Properties with a Custom EnvInfoProvider

```
package devguide.advanced.customenv.view;
import java.util.Hashtable;
import oracle.jbo.common.ampool.EnvInfoProvider;
/**
 * Custom EnvInfoProvider implementation to set
 * environment properties programmatically
 */
public class CustomEnvInfoProvider implements EnvInfoProvider {
  /**
   * Overridden framework method to set custom values in the
   * environment hashtable.
   *
   * @param string - ignore
   * @param environment Hashtable of config parameters
   * @return null - not used
   */
  public Object getInfo(String string, Object environment) {
    Hashtable envHashtable = (Hashtable)environment;
    envHashtable.put("some.property.name","some value");
    return null;
  }
  /* Required to implement EnvInfoProvider */
  public void modifyInitialContext(Object object) {}
  /* Required to implement EnvInfoProvider */
  public int getNumOfRetries() {return 0;}
```

```
}
```

When creating an application module for a stateless or command-line-client, with the `createRootApplicationModule()` method of the `Configuration` class, you can pass the custom EnvInfoProvider as the optional second argument. In order to use a custom EnvInfoProvider in an ADF web-based application, you need to implement a custom session cookie factory class as shown in Example 44–3. To use your custom session cookie factory, set the `jbo.ampool.sessioncookiefactoryclass` configuration property to the fully-qualified name of your custom session cookie factory class.

Example 44–3 Custom SessionCookieFactory to Install a Custom EnvInfoProvider

```
package devguide.advanced.customenv.view;
import java.util.Properties;
import oracle.jbo.common.ampool.ApplicationPool;
import oracle.jbo.common.ampool.EnvInfoProvider;
import oracle.jbo.common.ampool.SessionCookie;
import oracle.jbo.http.HttpSessionCookieFactory;
/**
 * Example of custom http session cookie factory
 * to install a custom EnvInfoProvider implementation
 * for an ADF web-based application.
 */
public class CustomHttpSessionCookieFactory
       extends HttpSessionCookieFactory {
  public SessionCookie createSessionCookie(String appId,
                                           String sessionId,
                                           ApplicationPool pool,
                                           Properties props) {
    SessionCookie cookie =
      super.createSessionCookie(appId, sessionId,pool, props);
    EnvInfoProvider envInfoProv = new CustomEnvInfoProvider();
    cookie.setEnvInfoProvider(envInfoProv);
    return cookie;
  }
}
```

44.2.5 What You May Need to Know About Configuration Property Scopes

Each runtime configuration property used by ADF Business Components has a scope. The scope of each property indicates at what level the property's value is evaluated and whether its value is effectively shared (i.e. static) in a single Java VM, or not. The ADF Business Components `PropertyManager` class is the registry of all supported properties. It defines the property names, their default values, and their scope. This class contains a `main()` method so that you can run the class from the command line to see a list of all the configuration property information.

Assuming JDEVHOME is the JDeveloper installation directory, to see this list of settings for reference, do the following:

```
$ java -cp JDEVHOME/BC4J/lib/bc4jmt.jar oracle.jbo.common.PropertyManager
```

Issuing this command will send all of the ADF Business Components configuration properties to the console. It also lists a handy reference about the different levels at which you can set configuration property values and remind you of the precedence order these levels have:

```
------------------------------------------------------------
Properties loaded from following sources, in order:
```

```
1. Client environment [Provided programmatically
                       or declaratively in bc4j.xcfg]
2. Applet tags
3. -D flags (appear in System.properties)
4. bc4j.properties file (in current directory)
5. /oracle/jbo/BC4J.properties resource
6. /oracle/jbo/commom.jboserver.properties resource
7. /oracle/jbo/common.Diagnostic.properties resource
8. System defined default
---------------------------------------------------------------
```

You'll see each property is listed with one of the following scopes:

- MetaObjectManager

  Properties at this scope are initialized once per Java VM when the ADF PropertyManager is first initialized.

- SessionImpl

  Properties at this scope are initialized once per invocation of `ApplicationModule.prepareSession()`.

- Configuration

  Properties at this scope are initialized when the ApplicationModule pool is first created and the application module's configuration is read the first time.

- Diagnostic

  Properties at this scope are specific to the built-in ADF Business Components diagnostic facility.

At each of these scopes, the layered value resolution described above is performed when the properties are initialized. Whenever property values are initialized, if you have specified them in the *Client Environment* (level 1 in the resolution order) the values will take precedence over values specified as System parameters (level 3 in the resolution order).

The Client Environment is a hashtable of name/value pairs that you can either programatically populate, or which will be automatically populated for you by the `Configuration` object when loaded, with the name/value pairs it contains in its entry in the `bc4j.xcfg` file. The implication of this is that for any properties scoped at MetaObjectManager level, the most reliable way to ensure that all of your application modules use the same default value for those properties is to do both of the following:

1. Make sure the property value does not appear in any of your application module's `bc4j.xcfg` file configuration name/value pair entries.
2. Set the property value using a Java system property in your runtime environment.

If, instead, you leave any MetaObjectManager-scoped properties in your `bc4j.xcfg` files, you will have the undesirable behavior that they will take on the value specified in the configuration of the *first* application module whose pool gets created after the Java VM starts up.

44.2.6 What You May Need to Know About How Database and Application Module Pools Cooperate

How ADF application module pools use the database connection pool depends on the setting of the `jbo.doconnectionpooling` application module configuration

parameter. In the Configuration Manager panel shown in Figure 44–1, you set this parameter using the checkbox labelled **Disconnect Application Module Upon Release**.

> **Note:** The notion of *disconnecting* the application module upon release to the pool better captures what the actual feature is doing than the related configuration parameter name (`jbo.doconnectionpooling`) does. The setting of `jbo.doconnectionpooling=false` does not mean that there is no database connection pooling happening. What it means is that the application module is not disconnected from its JDBC connection upon check in back to the application module pool.

If the default setting of `jbo.doconnectionpooling=false` is used, then when an application module instance is created in any pool it acquires a JDBC connection from the appropriate connection pool (based on the JDBC URL in the ADF case, or from the underlying JDBC data source implementation's pool in the case of a JNDI data source name). That application module instance holds onto the JDBC connection object that it acquired from the pool until the application module instance is removed from the application module pool. During its lifetime, that application module instance may service many different users, and ADF worries about issuing rollbacks on the database connection so that different users don't end up getting pending database state confused. By holding onto the JDBC connection, it allows each application module instance to keep its JDBC `PreparedStatement` objects open and usable across subsequent accesses by clients, thereby providing the best performance.

If `jbo.doconnectionpooling=true`, then each time a user session finishes using an application module (typically at the end of each HTTP request), the application module instance disassociates itself with the JDBC connection it was using on that request and it returns it to the JDBC connection pool. The next time that application module instance is used by a user session, it will reacquire a JDBC connection from the JDBC connection pool and use it for the span of time that application module is checked out of the application module pool (again, typically the span of one HTTP request). Since the application module instance "unplugs" itself from the JDBC connection object used to create the PreparedStatements it might have used during the servicing of the current HTTP request, those PreparedStatements are no longer usable on the next HTTP request because they are only valid in the context of the Connection object in which they were created. So, when using the connection pooling mode turned on like this, the trade-off is slightly more JDBC overhead setup each time, in return for using a smaller number of overall database connections.

The key difference is seen when many application module pools are all using the same underlying database user for their application connection.

- If 50 different application module pools each have even just a single application module instance in them, with `jbo.doconnectionpooling=false` there will be 50 JDBC application connections in use. If the application module pooling parameters are set such that the application module pools are allowed to shrink to `0` instances after an appropriate instance idle timeout by setting `jbo.ampool.minavailablesize=0`, then when the application module is removed from its pool, it will put back the connection its holding onto.
- In contrast, if 50 different application module pools each have a single application module instance and `jbo.doconnectionpooling=true`, then the amount of JDBC connections in use will depend on how many of those application modules are *simultaneously* being used by different clients. If an application module instance

is in the pool and is not currently being used by a user session, then with `jbo.doconnectionpooling=true` it will have released its JDBC connection back to the connection pool and while the application module instance is sitting there waiting for either another user to need it again, or to eventually be cleaned up by the application module pool monitor, it will not be "hanging on" to a JDBC connection.

> **Performance Tip:** Leave the `jbo.doconnectionpooling` configuration parameter set to `false` for best performance without sacrificing scalability and reliability. Database connection pooling is still achieved through application module pooling. The only exception is when multiple application module pools (and therefore a large number of application module instances) share the same database, making the total available database connections the highest priority.

Highest performance is achieved by not disconnecting the application module instance from its database connection on each check in to the application module pool. Accordingly, the default setting of the `jbo.doconnectionpooling` configuration parameter is `false`. The pooling of application module instances is already an effective way to optimize resource usage, and the Oracle ADF runtime is more efficient when you do not have to disconnect application module instances from their associated JDBC connection after each release to the pool. Effectively, by pooling the application modules which are related one-to-one with a JDBC connection, you are already achieving a pooling of database connections that is optimal for most Fusion web applications.

However, when minimizing the total overall number of database sessions is a priority, one situation in which it might be appropriate to use database connection pooling is when you have a large number of application module pools all needing to use database connections from the same underlying application user at the database level. In this case, the many application module pools can economize on the total overall database sessions by sharing a single, underlying database connection pool of JDBC connections, albeit at a loss of efficiency of each one. This choice would be favored only if total overall database sessions is of maximum priority. In this scenario, if a user scrolls through some, but not all rows of a view object's row set, then with `jbo.doconnectionpooling=true`, Oracle ADF will automatically passivate the pending application module state (including current row information) so that the next time the application module is used, the queried view object can be put back into the same current row with the same initial rows fetched. This passivation behavior may reduce performance.

44.2.7 What You May Need to Know About Application Module Pool Parameters

The application module pool configuration parameters fall into three logical categories relating to pool behavior, pool sizing, and pool cleanup behavior.

44.2.7.1 Pool Behavior Parameters

Table 44–1 lists the application module configuration parameters that affect the behavior of the application module pool.

Table 44–1 Application Module Pool Behavior Configuration Parameters

Pool Configuration Parameter	Description
Failover Transaction State Upon Managed Release (`jbo.dofailover`)	Enables eager passivation of pending transaction state each time an application module is released to the pool in "Managed State" mode. See Section 43.2.2.2, "How Passivation Changes When Optional Failover Mode is Enabled" for more information. Fusion web applications should set enable failover (`true`) to allow any other application module to activate the state at any time. By default, the failover feature is disabled (`jbo.dofailover=false`) as a performance optimization when there is only one web server instance configured, by reducing the need for passivation and activation. This allows for application module affinity to a specific user session. For high availability, enable the failover feature (`jbo.dofailover=true`) to ensure that more application modules are readily available, thereby increasing scalability. In this mode, passivation occurs at the end of every request. When failover is enabled, a failure can occur when Oracle WebLogic Server is configured to forcibly release connections back into the pool. A failure of this type produces a `SQLException` (Connection has already been closed.) that is saved to the server log. To ensure that state passivation occurs and users' changes are saved, the server administrator should set an appropriate value for the `weblogic-application.xml` deployment descriptor parameter `inactive-connection-timeout-seconds` on the `<connection-check-params>` pool params element. Setting the deployment descriptor parameter to several minutes, in most cases, should avoid forcing the inactive connection timeout and the resulting passivation failure. Adjust the setting as needed for your environment.
Row-Level Locking Behavior Upon Release (`jbo.locking.mode`)	Forces the application module pool not to create a pending transaction state on the database with row-level locks each time the application module is released to the pool. See Section 43.11.1, "How to Confirm That Applications Use Optimistic Locking" for more information. Fusion web applications should leave the locking mode set to the default value `optimistic` to avoid creating the row-level locks. This feature is disabled when the property is set to `pessimistic`.

Table 44–1 (Cont.) Application Module Pool Behavior Configuration Parameters

Pool Configuration Parameter	Description
Disconnect Application Module Upon Release (`jbo.doconnectionpooling`)	Forces the application module pool to release the JDBC connection used each time the application module is released to the pool. See Section 44.2.6, "What You May Need to Know About How Database and Application Module Pools Cooperate" for more information. This feature is disabled by default (`false`).
Transaction Memory State Upon Release (`jbo.txn.disconnect_level`)	By default, after the application module is passivated, the view objects and its row sets are closed and removed, to be recreated and reverted to their original state upon activation. This behavior corresponds to the default `jbo.txn.disconnect_level=0`. When setting `jbo.txn.disconnect_level=1` the application module, view objects and row sets all remain in memory and stay valid but their corresponding references to JDBC objects are dropped. Upon activation, the framework reexecutes and synchronizes the cursor positions. Setting `jbo.txn.disconnect_level=1` can improve performance when enabling application modules to disconnect from their JDBC connection (when used in conjunction with `jbo.doconnectionpooling=true`) and reduces the memory overhead associated with this situation.
Enable Application Module Pooling (`jbo.ampool.doampooling`)	Enables application module pooling by default. Whenever you deploy your application in a production environment the default setting of `jbo.ampool.doampooling` set to `true` is the way you will run your application. But, as long as you run your application in a test environment, setting the property to `false` can play an important role in your testing. When this property is `false`, there is effectively no application pool. See Section 43.10, "Testing to Ensure Your Application Module is Activation-Safe" for more information. This feature is enabled by default (`true`).
Support Dynamic JDBC Credentials (`jbo.ampool.dynamicjdbccredentials`)	Enables additional pooling lifecycle events to allow developer-written code to change the database credentials (username/password) each time a new user session begins to use the application module. This feature is enabled by default (`true`), however this setting is a necessary but not sufficient condition to implement the feature. The complete implementation requires additional developer-written code.

Table 44–1 (Cont.) Application Module Pool Behavior Configuration Parameters

Pool Configuration Parameter	Description
Reset Non-Transactional State Upon Unmanaged Release (`jbo.ampool.resetnontransactionalstate`))	Forces the application module to reset any non-transactional state like view object runtime settings, JDBC prepared statements, bind variable values, etc. when the application module is released to the pool in unmanaged or "stateless" mode. This feature is enabled by default (`true`). Disabling this feature can improve performance, however since it does not clear bind variable values, your application needs to ensure that it systemically sets bind variable values correctly. Failure to do so with this feature disabled can mean one user might see data with another users bind variable values.)

44.2.7.2 Pool Sizing Parameters

Table 44–2 lists the application module configuration parameters that affect the sizing of the application module pool.

Table 44–2 Application Module Pool Sizing Configuration Parameters

Pool Configuration Parameter	Description
Initial Pool Size (`jbo.ampool.initpoolsize`)	The number of application module instances to created when the pool is initialized. The default is `0` (zero) instances. A general guideline is to configure this to 10% more than the anticipated number of concurrent application module instances required to service all users. Creating application module instances during initialization takes the CPU processing costs of creating application module instances during the initialization instead of on-demand when additional application module instances are required.
Maximum Pool Size (`jbo.ampool.maxpoolsize`)	The maximum number of application module instances that the pool can allocate. The pool will never create more application module instances than this limit imposes. The default is `4096` instances. A general guideline is to configure this to 20% more than the initial pool size to allow for some additional growth. If this is set too low, then some users may see an error accessing the application if no application module instances are available.

Table 44–2 (Cont.) Application Module Pool Sizing Configuration Parameters

Pool Configuration Parameter	Description
Referenced Pool Size (`jbo.recyclethreshold`)	The maximum number of application module instances in the pool that attempt to preserve session affinity for the next request made by the session which used them last before releasing them to the pool in managed-state mode. The referenced pool size should always be less than or equal to the maximum pool size. The default is to allow `10` available instances to try and remain "loyal" to the affinity they have with the most recent session that released them in managed state mode. Configure this value to maintain the application module instance's affinity to a user's session. A general guideline is to configure this to the expected number of concurrent users that perform multiple operations with short think times. If there are no users expected to use the application with short think times, then this can be configured to `0` zero to eliminate affinity. Maintaining this affinity as much as possible will save the CPU processing cost of needing to switch an application module instance from one user session to another.
Maximum Instance Time to Live (`jbo.ampool.timetolive`)	The number of milliseconds after which to consider an application module instance in the pool as a candidate for removal during the next resource cleanup regardless of whether it would bring the number of instances in the pool below minavailablesize. The default is 3600000 milliseconds of total time to live (which is 3600 seconds, or one hour). The default value is sufficient for most applications.

44.2.7.3 Pool Cleanup Parameters

A single "application module pool monitor" per Java VM runs in a background thread and wakes up every so often to do resource reclamation. Table 44–3 lists the parameters that affect how resources are reclaimed when the pool monitor does one of its resource cleanup passes.

> **Best Practice:** When you specify the length of time between application module pool cleanup passes, set all application modules to use the same **Pool Polling Interval** value. Since there is only a single application monitor pool monitor per Java VM, the value that will effectively be used for the application module pool monitor polling interval will be the value found in the application module configuration read by the *first* application module pool that gets created. Setting all application modules to use the same value ensures that this value is set in a predictable way.

Table 44–3 Application Module Resource Management Configuration Parameters

Pool Configuration Parameter	Description
Pool Polling Interval (`jbo.ampool.monitorsleepinterval`)	The length of time in milliseconds between pool resource cleanup. While the number of application module instances in the pool will never exceed the maximum pool size, available instances which are candidates for getting removed from the pool do not get "cleaned up" until the next time the application module pool monitor wakes up to do its job. The default is to have the application module pool monitor wake up every `600000` milliseconds (which is 600 seconds, or ten minutes). Configuring a lower interval results in inactive application module instances being removed more frequently to save memory. Configuring a higher interval results in less frequent resource cleanups.
Maximum Available Size (`jbo.ampool.maxavailablesize`)	The ideal maximum number of available application module instances in the pool when not under abnormal load. When the pool monitor wakes up to do resource cleanup, it will try to remove available application module instances to bring the total number of available instances down to this ideal maximum. Instances that have been not been used for a period longer than the idle instance time-out will always get cleaned up at this time, then additional available instances will be removed if necessary to bring the number of available instances down to this size. The default maximum available size is `25` instances. Configure this to leave the maximum number of available instances desired after a resource cleanup. A lower value generally results in more application module instances being removed from the pool on a cleanup.
Minimum Available Size (`jbo.ampool.minavailablesize`)	The minimum number of available application module instances that the pool monitor should leave in the pool during a resource cleanup operation. Set to `0` (zero) if you want the pool to shrink to contain no instances when all instances have been idle for longer than the idle time-out after a resource cleanup. The default is `5` instances.

Table 44–3 (Cont.) Application Module Resource Management Configuration Parameters

Pool Configuration Parameter	Description
Idle Instance Timeout (`jbo.ampool.maxinactiveage`)	The number of milliseconds after which to consider an inactive application module instance in the pool as a candidate for removal during the next resource cleanup. The default is 600000 milliseconds of idle time (which is 600 seconds, or ten minutes). A lower value results in more application module instances being marked as a candidate for removal at the next resource cleanup. A higher value results in fewer application module instances being marked as a candidate for removal at the next resource cleanup.
Maximum Instance Time to Live (`jbo.ampool.timetolive`)	The number of milliseconds after which to consider an connection instance in the pool as a candidate for removal during the next resource cleanup regardless of whether it would bring the number of instances in the pool below minavailablesize. The default is 3600000 milliseconds of total time to live (which is 3600 seconds, or one hour). A lower value reduces the time an application module instance can exist before it must be removed at the next resource cleanup. The default value is sufficient for most applications. A higher value increases the time an application module instance can exist before it must be removed at the next cleanup.

44.2.8 What You May Need to Know About Data Source Configuration

When you specify a JDBC data source as your application module's connection type, any configuration parameters that you have configured for the database connection pool will be ignored. To configure the connection pool for your data source, you must use the means provided by your Java EE container. In the case of Oracle WebLogic Server, you configure the data source using the Oracle WebLogic Server Administration Console.

The main steps for configuring JDBC data sources are:

1. Create a data source for each data base that you want to connect to. When you create the data source, specify the configuration options to match the ones for ADF Business Components database connection pools described in Table 44–4. The configuration settings that you specify will depend on your database and the capacity planning that you need anticipate for your application.

   For details about configuring JDBC data sources and connection pool capacity planning, see *Oracle Fusion Middleware Configuring and Managing JDBC for Oracle WebLogic Server*.

2. Optionally, configure transaction options for the data source.
3. Optionally, configure connection testing options for the data source.
4. Optionally, target the data source to additional servers and clusters.

For detailed procedures for each of these steps, see the topic "Configure JDBC data sources" in the Administration Console Online Help.

Table 44–4 Equivalent Oracle WebLogic Server Data Source Parameters

ADF Business Components Parameter	\Oracle WebLogic Server Parameter
Initial Pool Size (`jbo.initpoolsize`)	Initial Capacity
Maximum Pool Size (`jbo.maxpoolsize`)	Maximum Capacity
Pool Polling Interval (`jbo.poolmonitorsleepinterval`)	No equivalent for Oracle WebLogic Server.
Maximum Available Size (`jbo.poolmaxavailablesize`)	Maximum Capacity
Minimum Available Size (`jbo.poolminavailablesize`)	Maximum Capacity
Idle Instance Timeout (`jbo.poolmaxinactiveage`)	Shrink Frequency Seconds

44.2.9 What You May Need to Know About Database Connection Pool Parameters

If you are using a JDBC URL for your connection information so that the ADF database connection pool is used, then configuration parameters listed in Table 44–5 can be used to tune the behavior of the database connection pool. A single "database connection pool monitor" per Java VM runs in a background thread and wakes up every so often to do resource reclamation. The parameters in Table 44–3 include the ones that affect how resources are reclaimed when the pool monitor does one of its resource cleanup passes.

Note: The configuration parameters for database connection pooling have MetaObjectManager scope (described in Section 44.2.5, "What You May Need to Know About Configuration Property Scopes" earlier). This means their settings are global and will be set once when the first application module pool in your application is created. To insure the most predictable behavior, leave the values of these parameters in the **Connection Pool** section of the **Pooling and Scalability** tab at their *default* values — so that no entry for them is written into the `bc4j.xcfg` file — and to instead set the desired values for the database connection pooling tuning parameters as Java System Parameters in your Java EE container.

Table 44–5 Database Connection Pool Parameters

Pool Configuration Parameter	Description
Initial Pool Size (`jbo.initpoolsize`)	The number of JDBC connection instances to created when the pool is initialized The default is an initial size of `0` instances.
Maximum Pool Size (`jbo.maxpoolsize`)	The maximum number of JDBC connection instances that the pool can allocate. The pool will never create more JDBC connections than this imposes. The default is `4096` instances.

Table 44–5 (Cont.) Database Connection Pool Parameters

Pool Configuration Parameter	Description
Pool Polling Interval (`jbo.poolmonitorsleepinterval`)	The length of time in milliseconds between pool resource cleanup. While the number of JDBC connection instances in the pool will never exceed the maximum pool size, available instances which are candidates for getting removed from the pool do not get "cleaned up" until the next time the JDBC connection pool monitor wakes up to do its job. The default is `600000` milliseconds of idle time (which is 600 seconds, or ten minutes). Configuring a lower interval results in inactive connection instances being removed more frequently to save memory. Configuring a higher interval results in less frequent resource cleanups.
Maximum Available Size (`jbo.poolmaxavailablesize`)	The ideal maximum number of JDBC connection instances in the pool when not under abnormal load. When the pool monitor wakes up to do resource cleanup, it will try to remove available JDBC connection instances to bring the total number of available instances down to this ideal maximum. Instances that have been not been used for a period longer than the idle instance time-out will always get cleaned up at this time, then additional available instances will be removed if necessary to bring the number of available instances down to this size. The default is an ideal maximum of `25` instances (when not under load).
Minimum Available Size (`jbo.poolminavailablesize`)	The minimum number of available JDBC connection instances that the pool monitor should leave in the pool during a resource cleanup operation. Set to zero (`0`) if you want the pool to shrink to contain no instances when all instances have been idle for longer than the idle time-out. The default is to not let the minimum available size drop below `5` instances.
Idle Instance Timeout (`jbo.poolmaxinactiveage`)	The number of seconds after which to consider an inactive JDBC connection instance in the pool as a candidate for removal during the next resource cleanup. The default is `600000` milliseconds of idle time (which is 600 seconds, or ten minutes).

Notice that since the database connection pool does not implement the heuristic of session affinity, there is *no* configuration parameter for the database connection pool which controls the referenced pool size.

44.3 Initializing Database State and Pooling Considerations

Sometimes you may need to invoke stored procedures to initialize database state related to the current user's session. The correct place to perform this initialization is in an overridden `prepareSession()` method of your application module.

44.3.1 How to Set Database State Per User

The Fusion web application can set database state on a per-user basis. You typically create a database `CONTEXT` namespace, associate a PL/SQL procedure with it, and then use the `SYS_CONTEXT()` SQL function to reference values from the context.

For example, you can use the PL/SQL package to set and get a package-level variable that holds the name of the currently authenticated Fusion web application user as shown in Example 44–4.

Example 44–4 CONTEXT_PKG PL/SQL Package

```
create or replace package context_pkg as
  procedure set_app_user_name(username varchar2);
  function app_user_name return varchar2;
end context_pkg;
```

Then your application can define the `WHERE` clause of a view object to reference the `context_pkg.app_user_name` function and query the per-user state.

To set the database state, the application module framework extension class (*`AppModuleImpl`*`.java`) defines a `callStoredProcedure()` helper method similar to the ones in Section 12.6.2, "How to Invoke Stored Procedure with Only IN Arguments." The custom application module class then extends this framework extension class and defines the `setCurrentUserInPLSQLPackage()` helper method shown in Example 44–5. The helper method uses the `callStoredProcedure()` method to invoke the `context_pkg.set_app_user_name()` stored procedure, passing the value of the currently authenticated user as a parameter value.

Example 44–5 Method to Call Context_Pkg.Set_App_User_Name Stored Procedure

```
// In CustomAppModuleImpl.java
public void setCurrentUserInPLSQLPackage() {
  String user = getUserPrincipalName();
  callStoredProcedure("context_pkg.set_app_user_name(?)",new Object[]{user});
}
```

With this helper method in place, the custom application module class then overrides the `prepareSession()` method as shown in Example 44–6.

Example 44–6 Overridden afterConnect() and prepareSession() to Set Database State

```
// In CustomAppModuleImpl.java
  protected void prepareSession(Session session) {
    super.prepareSession(session);
    getLoggedInUser().retrieveUserInfoForAuthenticatedUser();
    setUserIdIntoUserDataHashtable();
    setCurrentUserInPLSQLPackage();
  }
```

44.3.2 What You May Need to Know About Database User State and jbo.doconnectionpooling = true

The default setting for `jbo.doconnectionpooling` is `false`. This means the application module instance hangs onto its JDBC connection while it's in the application module pool. This is the most efficient setting because the application module can keep its JDBC prepared statements open across application module checkouts/checkins.The application module instance will trigger its `prepareSession()` method each time a new user session begins using it.

If you set `jbo.doconnectionpooling` to `true`, then on each checkout of an application module from the pool, that application module pool will acquire a JDBC connection from the database connection pool and use it during the span of the current request. At the end of the request when the application module is released back to the application module pool, that application module pool releases the JDBC connection it was using back to the database connection pool.

It follows that with `jbo.doconnectionpooling` set to `true` the application module instance in the pool may have a completely different JDBC connection each time you check it out of the pool. In this situation, the `prepareSession()` method will fire each time the application module is checked out of the pool to give you a chance to reinitialize the database state.

Alternatively, you can set `jbo.txn.disconnect_level=1` (default is `0`) to ensure that all application modules, view objects and row sets remain in memory and stay valid after their corresponding references to JDBC connections are dropped. Upon activation, the framework reexecutes and synchronizes the cursor positions. When used in conjunction with `jbo.doconnectionpooling=true`, setting `jbo.txn.disconnect_level=1` reduces the memory overhead associated with this situation.

45 Using the Active Data Service

This chapter provides information for using the ADF Model layer and ADF Faces components with an active data service.

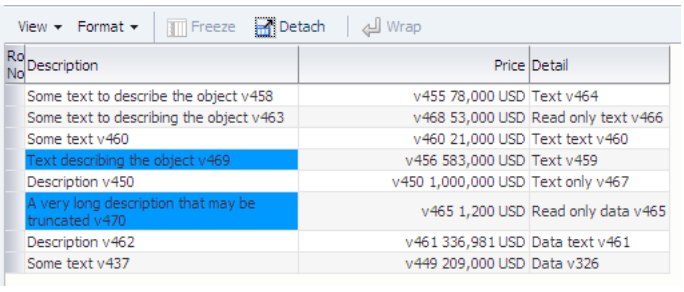
This chapter includes the following sections:

- Section 45.1, "About the Active Data Service"
- Section 45.2, "Configuring the Active Data Service"
- Section 45.3, "Configuring Components to Use the Active Data Service"
- Section 45.4, "Using the Active Data Proxy"
- Section 45.5, "Using the Active Data with a Scalar Model"

45.1 About the Active Data Service

The Fusion technology stack includes Active Data Service (ADS), which is a server-side push framework that allows you to provide real-time data updates for ADF Faces components. You bind ADF Faces components to a data source and ADS pushes the data updates to the browser client without requiring the browser client to explicitly request it. For example, you may have a table bound to attributes of an ADF data control whose values change on the server periodically, and you want the updated values to display in the table. You can configure your application and the component so that whenever the data changes on the server, the ADF Model layer notifies the component and the component rerenders the changed data with the new value highlighted, as shown in Figure 45–1.

Figure 45–1 Table Displays Updated Data as Highlighted



45.1.1 Active Data Service Use Cases and Examples

Using ADS is an alternative to using automatic partial page rendering (PPR) to rerender data that changes on the backend as a result of business logic associated with

the ADF data control bound to the ADF Faces component. Whereas automatic PPR requires sending a request to the server (typically initiated by the user), ADS enables changed data to be pushed from the data store as the data arrives on the server. Also, in contrast to PPR, ADS makes it possible for the component to rerender only the changed data instead of the entire component. This makes ADS ideal for situations where the application needs to react to data that changes periodically.

To use this functionality, you must configure the application to use ADS. If your application services do not support ADS, then you also need to create a proxy of the service so that the components can display the data as it updates in the source.

Any ADF Faces page can use ADS. However, you can configure only the following ADF Faces components to work with active data:

- `activeCommandToolbarButton`
- `activeImage`
- `activeOutputText`
- `table`

  Note: Do not use filtering on a table that will be using active data. Once a table is filtered at runtime, active data cannot be displayed. Currently, ADS supports `table` components with the `outputText` component contained within a column; other components are not supported inside the table column

- `tree`
- `treeTable`
- DVT graph, gauge, and geographical map components

45.1.2 Limitations of the Active Data Service Framework

The framework for ADS has the following limitations.

- ADS does not support active data for ADF Faces components added dynamically, using PPR.
- ADS does not support active data for ADF Faces components that appear in a web page or region that gets invoked from a task flow call activity. Do not use task flow call activities with bounded task flows when active data is required.
- ADS does not support active data in web browsers that have popup blockers enabled.

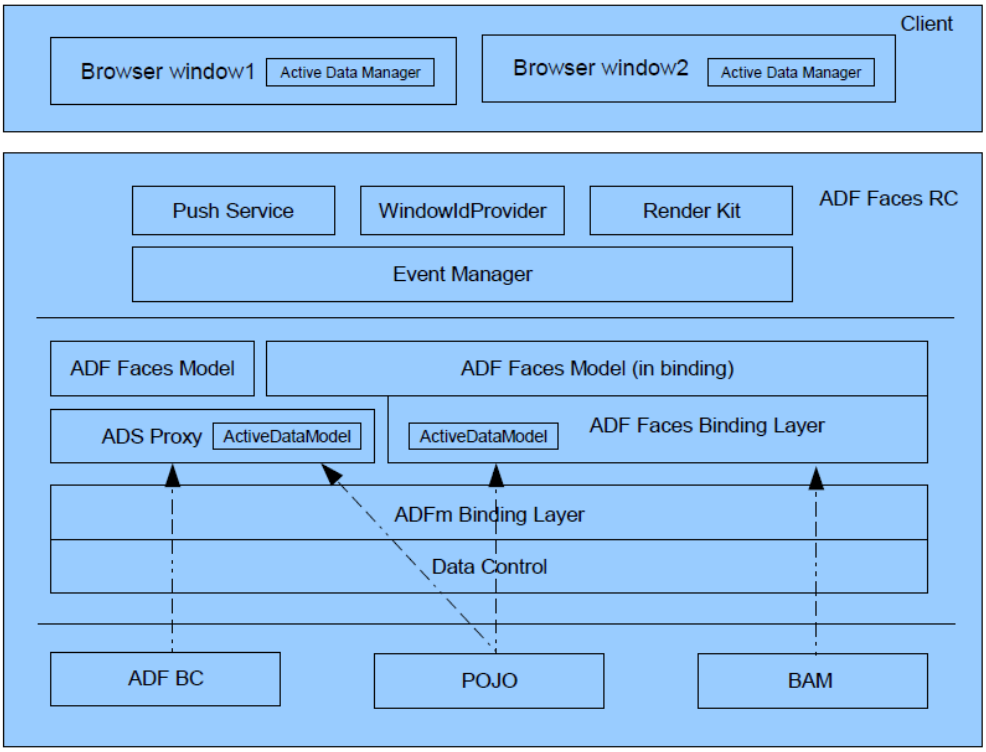
  Before users can run the ADF Faces page with ADS configured for the application, they must disable the popup blocker for their web browser.
- ADS does not support active data on ADF Faces table components with filtering enabled. Once a table is filtered at runtime, active data cannot be displayed.

45.1.3 Active Data Service Framework

The framework for ADS contains a number of components that work together to send the active data from the source to the UI component. When a data event occurs, if the associated ADF Model layer binding is configured for active data, the Active Data model delivers the data to the Event Manager. The Event Manager then retrieves the data and invokes the Push Service, which delivers the data to the correct component,

based on how the service is configured (for more information, see Section 45.1.4, "Data Transport Modes"). The component then applies the new data pushed from the server. Figure 45–2 shows the ADS framework.

Figure 45–2 Active Data Service Framework



In order to use the Active Data Service, you need to have a data store that publishes events when data is changed, and you need to create business services that react to those events. By default, ADF Business Components does not react to data events. The Active Data Proxy framework allows all types of data sources, including ADF Business Components, to work with ADS. It combines the `ActiveDataModel` with the JSF model, so that you need to override functionality only on this proxy rather than on both the `ActiveDataModel` and the JSF model.

The following comprise the ADS framework:

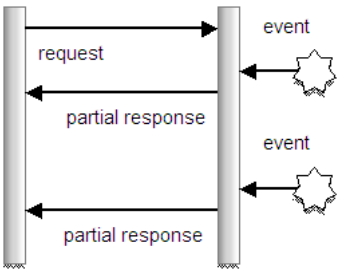
- `ActiveDataModel` interface: Abstraction of the active data model. Its responsibilities include:
  - Starting and stopping active data
  - Keeping track of the current active data event ID
  - Letting the renderer know whether the model needs active data or not.
- Event Manager: A server-side component that works with the ADF Model layer. It is responsible for the following:
  - Listening to binding events
  - Retrieving active data
  - Managing active data encoding
  - Invoking the Push service to send the encoded active data

- Push service: A delivery channel that interacts with the Event Manager on the server side and with the Active Data Manager on the client side. It provides the following:
  - Establishing and maintaining the connection between the server and the client
  - Transmitting the active data over this connection from the server to the client
  - Ensuring that active data gets delivered within desired parameters and forcing component update if not
- Active Data Manager: A client-side component that distributes the active data to the correct component. Specifically, it is responsible for the following;
  - Delivering events from the server side that are coming through the channel, using an event delivery service
  - Handling multiple browser windows through a shared channel
  - Dispatching active data events to rich client components, so that the components can render the change accordingly
- Active Data Proxy: A proxy that allows all types of data sources to enable push functionality. Specifically, the proxy is responsible for the following:
  - Implementing and delegating `ActiveDataModel` functionality
  - Delegating to JSF models
  - Listening to data change events from the data source
  - Generating active data events based on the data change events

45.1.4 Data Transport Modes

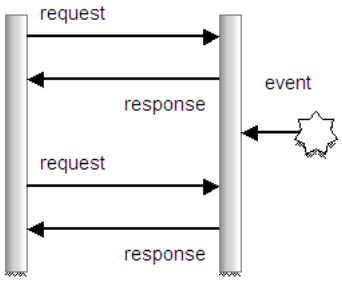
Active data is sent to the client using data streaming (push) or one of two types of data polling. With *data streaming*, there is only one request, which stays open. When a data change event occurs, a partial response is sent (the response is not closed), the client is notified, and the associated component is updated to show the new data, as shown in Figure 45–3.

Figure 45–3 Streaming Mode



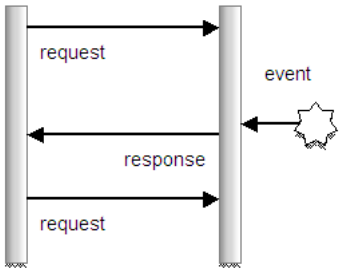
With *data polling*, the application is configured to poll the data source at specified intervals, as shown in Figure 45–4. With each request, a response is sent and closed, whether or not a data change event has occurred. If the data has changed, then the client is notified and the component is updated.

Figure 45–4 Poll Mode



Long polling is similar to streaming. When the page is rendered, a request is sent to the active channel. However, a response is not returned until there is a data change event. At that point, the connection is closed. As soon as the connection is closed, a new connection is initiated, which results in the connection being active most of the time: there are no specific intervals. Long polling results in the majority of data change events being received when they occur, because the connection is already established and ready to send a response to the client, as shown in Figure 45–5.

Figure 45–5 Long Polling Mode



For more information, see Section 45.2.2, "What You May Need to Know About Transport Modes."

To use ADS, you need to configure your application to determine the method of data transport, as well as other performance options.You also need to configure the bindings for your components so that they can use ADS. If you are using ADF Business Components, you need to modify your model to implement the `ActiveModel` interface to register itself as the listener for active data events using the Active Data Proxy.

> **Tip:** Oracle offers Oracle Business Activity Monitoring (BAM), which is a complete solution that provides the ability to create an active data store. For more information, see the *Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite*.

45.2 Configuring the Active Data Service

You need to configure ADS to determine the data transport mode, as well as to set other configurations, such as a latency threshold and reconnect information.

Note: If you enable ADS but do not configure it, the ADS framework will use the default values shown in Table 45–1.

45.2.1 How to Configure the Active Data Service

Configuration for running the Fusion web application with ADS in JDeveloper and Integrated WebLogic Server is done in the `adf-config.xml` file. For more information about the `adf-config.xml` file, including how to create one if you need to, see the "Configuration in adf-config.xml" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Note: To support automatic replication and failover for web applications within a clustered environment, Oracle WebLogic Server supports mechanisms for replicating HTTP session state across clusters. You can configure Oracle Application Development Framework (Oracle ADF) to ensure the Fusion web application's state can be restored from any server in the cluster. To support failover for pages configured to display active data, it is necessary to enable failover for the ADF Business Components application module and to enable ADF Controller to track changes to the ADF memory scopes. With these ADF settings configured, when failover occurs, the ADF server will detect the failover and request all pages configured to display active data to refresh themselves, after that ADS restarts and data is pushed to the client. For details about how to enable failover for the Fusion web application in a high availability environment, see the *Oracle Fusion Middleware High Availability Guide*.

To configure the Active Data Service:

1. In the Application Resources panel, double-click the **adf-config.xml** file.
2. Click the **Source** tab to open the file in the source editor, and create an entry for each of the elements shown in Table 45–1.

Table 45–1 ADS Configuration Elements in adf-config.xml

Element	Description	Default Value (in milliseconds)	Minimum Value (in milliseconds)
`<transport>`	The method by which data will be delivered to the client. Value values are: ▪ `streaming` (default) ▪ `polling` ▪ `long-polling` For more information, see Section 45.2.2, "What You May Need to Know About Transport Modes."		
`<latency threshold>`	Latency threshold in milliseconds. Active data messages with network delays greater than this threshold will be treated as late.	10000	1000

Table 45–1 (Cont.) ADS Configuration Elements in adf-config.xml

Element	Description	Default Value (in milliseconds)	Minimum Value (in milliseconds)
`<keep-alive-interval>`	Frequency in milliseconds for sending keep-alive messages when no events are generated.	10000	5000
`<polling-interval>`	When `<transport>` set to `polling`, frequency in milliseconds of the poll request.	5000	1000
`<max-reconnect-attempt-time>`	Maximum period of time in milliseconds a client will attempt to reconnect the push channel to the server upon getting disconnected.	1800000 (30 minutes)	0
`<reconnect-wait-time>`	Time interval in milliseconds to wait between reconnect attempts.	10000	1000

Performance Tip: Keep the following in mind when configuring the ADS.

- Set the latency threshold to more than 1000 to avoid frequent component refreshing.
- Set the keep-alive interval and reconnect wait time to be less than the browser's keep-alive timeout.
- Set the max reconnect time to be less than your web server's session timeout.

Example 45–1 shows a sample configuration that has content pushed to the client.

Example 45–1 Sample Configuration for ADS in adf-config.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<adf-config xmlns="http://xmlns.oracle.com/adf/config"
            xmlns:ads="http://xmlns.oracle.com/adf/activedata/config">
  <ads:adf-activedata-config xmlns=
                          "http://xmlns.oracle.com/adf/activedata/config">
    <transport>streaming</transport>
    <latency-threshold>5000</latency-threshold>
    <keep-alive-interval>10000</keep-alive-interval>
    <max-reconnect-attempt-time>90000</max-reconnect-attempt-time>
    <reconnect-wait-time>8000</reconnect-wait-time>
  </ads:adf-activedata-config>
</adf-config>
```

3. Synchronize the clocks on the data server and on the application server. If these are not synchronized, then events may appear to ADS to have occurred in the future.

45.2.2 What You May Need to Know About Transport Modes

ADS can use one of three transport modes to deliver active data to the component: streaming, polling, or long polling.

When you configure ADS to use the streaming mode, data is pushed to the client whenever a change event is raised by the data. On the client side, after the push channel is established, if there is no activity within the time of the value for the `latency-threshold` element plus the `keep-alive-interval`, an `establish-channel-request` will be sent out repeatedly based on the value of the `reconnect-wait-time` element, until the amount of time passed reaches the value of the `max-reconnect-attempt-time` element. After that, the connection will be considered disconnected. For example, given the values in Example 45–1, if there is no activity in 15,000 milliseconds, an establish channel request will be sent out every 8,000 milliseconds for up to 90,000 milliseconds. After that, the connection will be considered disconnected.

On the server side, the server disconnects the push channel and starts a timer to clean up with a `cleanup-delay-time` when there is an empty active data message or when it fails to send out an active data message. The `cleanup-delay-time` is calculated as `max-reconnect-attempt-time` + 30 * 1000 ms. Its default value is 30 minutes.

When you configure ADS to use the polling mode, on the client side the polling request is scheduled to be sent out repeatedly after the value of the `polling-interval` element has been reached. If no response is received after the value of the `max-reconnect-attempt-time` has elapsed, the connection is treated as disconnected and no more requests will be sent. After receiving a polling response, if the time the response has taken is greater than the `polling-interval` element, the service sends the next request out right away. If it is less, the next request will be sent as scheduled.

For the server side, the session ends after the polling response is returned. At that point, a timer with a `cleanup-delay-time` is set up to trigger cleanup. If a new request comes in before the timer fires, the old timer is canceled, and new timer is created.

When you configure ADS to use the long polling mode, requests are made as they are in streaming mode; however, as soon as the connection is treated as disconnected, a new connection is initiated. The result is a significant reduction in latency.

Table 45–2 compares the three different modes.

Table 45–2 Comparison of Streaming, Polling, and Long-Polling Modes

	Streaming	Polling	Long-Polling
Latency	Very good. Directly after an event occurs on the server side, a partial response is sent to the client. If there is another event, immediately, it is also sent as a partial response. There is almost no latency with this approach.	Poor, depending on the polling interval. If the polling interval is short (for example, 0.5 seconds), it will slow down the network because the connections are repeatedly opened. It is also expensive on the client- and server-side resources.	Good. However, when there is a new event immediately after a response has been closed, there is some latency until the new data appears on the client side. On average, this is not a problem.

Table 45–2 (Cont.) Comparison of Streaming, Polling, and Long-Polling Modes

	Streaming	Polling	Long-Polling
HTTP Proxy	Poor. For some older servers, because the response is never released, when a proxy is sitting between client and server, it is possible that the proxy will buffer responses. This is an unfortunate optimization that prevents real-time data from flowing into the browser. Long polling should be used if a proxy is used.	Good.	Good.
Number of live connections	Poor. Many concurrent connections, as the stream is always live.	Good. Connections are live only during the actual poll. Note however that if there is a high polling rate then the number of concurrent connections will also be high.	Poor. Many concurrent connections, as the stream is almost always live.
Communication channel	`HTTP GET` This can result in the display of a "busy" cursor or the animation of a browser's "throbber" icon.	`XMLHttpRequest` (XHR) `HTTP GET`	`XMLHttpRequest` (XHR) `HTTP GET`

45.3 Configuring Components to Use the Active Data Service

How you configure components to use ADS depends on whether or not you must use the Active Data proxy. If your application uses a data store that publishes events when data is changed, and your business services react to those events (for example, if your application uses BAM), then you do not need to use the Active Data proxy.

If your business services do not react to those events (for example, if your application uses ADF Business Components), then you must use the Active Data proxy and follow different procedures for configuring your components.

Note: If your business service requires the use of the Active Data proxy, then you can only use the following components with active data:

- table
- tree
- treeTable
- DVT graph, gauge, and geographical map components

45.3.1 How to Configure Components to Use the Active Data Service Without the Active Data Proxy

To use ADS without the proxy, you need to set a value on the binding element in the corresponding page definition file.

To configure a component to display active data without the Active Data proxy:

1. Drop a component onto a JSF page.
2. If you are using an ADF bound tree or tree table, you need to ensure the following:
   - The binding represents homogeneous data (that is, only one rule), although an accessor can still access a like accessor.
   - The binding rule contains a single attribute.
   - The table does not use filtering.
   - The tree component's `nodeStamp` facet contains a single `outputText` tag and contains no other tags, as described in Section 45.3.3, "What You May Need to Know About Displaying Active Data in ADF Trees."
3. Open the page's associated page definition file.
4. In the Structure window for the page definition file, select the node that represents the attribute binding for the component. In the Property Inspector, set the **ChangeEventPolicy** attribute to **Push**.

   Tip: You can use the `statusIndicator` component to indicate the server state. For more information, see the "Displaying Application Status Using Icons" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

45.3.2 How to Configure Components to Use the Active Data Service with the Active Data Proxy

To use ADS with the proxy, you need bind the value of your component to decorator class that will use the proxy. For more information about this class, see Section 45.4, "Using the Active Data Proxy." to set a value on the binding element in the corresponding page definition file.

To configure a component to display active data with the Active Data proxy:

1. Drop a component onto a JSF page.
2. Change the value attribute to be bound to a decorator class that you will create for use with the proxy. For more information, see Section 45.4, "Using the Active Data Proxy."

If you are using an ADF bound tree or tree table, you need to ensure the following:

- The binding represents homogeneous data (that is, only one rule), although an accessor can still access a like accessor.
- The binding rule contains a single attribute.
- The table does not use filtering.
- The tree component's `nodeStamp` facet contains a single `outputText` tag and contains no other tags, as described in Section 45.3.3, "What You May Need to Know About Displaying Active Data in ADF Trees."

45.3.3 What You May Need to Know About Displaying Active Data in ADF Trees

When you create an ADF Faces tree (or tree table) component, you configure a `nodeStamp` facet, which is a holder for the component used to display the data for each node of the tree. Each node is rendered (stamped) once, repeatedly for all nodes.

Because of this stamping behavior, only certain types of components are supported as children inside an ADF Faces tree component. When the tree component is not bound to an active data source, all components that have no behavior are supported. However, when you configure the tree to use ADS, only the `outputText` component is supported inside of the `nodeStamp` facet, as shown in Example 45–2. The `nodeStamp` facet must not contain any other tags. So, for example, active data will not work if you add `panelGroupLayout` tags to the `nodeStamp` facets of a tree configured to use ADS.

Example 45–2 Tree Configured to Use ADS Must Contain outputText Component

```
<f:facet name="nodeStamp">
  <af:outputText value="#{row.str2}"/>
</f:facet>
```

45.3.4 What Happens at Runtime: How Components Render When Bound to Active Data

After you configure your application and the component for ADS, whenever the data changes on the server, the component is notified and rerenders with only the changed data. In contrast, a component not configured for either active data or automatic PPR will need to be explicitly refreshed after a data change occurs. The explicit refresh will refetch all data that is visible on the client, including data that has not changed. Consequently, this will force the entire component to rerender. Additionally, when the component is bound to an active data source (with active data policy "Push"), the component rerenders with the new value highlighted.

45.3.5 What You May Need to Know About ADS and Google Chrome

When the Fusion web application runs in the Google Chrome web browser and a user presses Ctrl+N (or Ctrl+T) to open a new window (or tab) and then copies the URL from the original window into the new window, active data in the original window will stop streaming. According to the Google Chrome process model, the new browser window will be created in a separate process and both windows will share the same HTTP session. However, because browser windows that are created in two separate processes cannot communicate with each other, ADS will become out of sync between the client and server and will stop streaming. As a workaround, to allow active data in multiple Google Chrome windows, before copying the URL from the original window into the new window users must press Ctrl-Shift-N to open the browser window in

incognito mode (private browsing). Because Ctrl-Shift-N opens the window in a separate process and a separate HTTP session, ADS will not attempt to synchronize between the windows and streaming will be unaffected.

45.4 Using the Active Data Proxy

You use the active data proxy when your business services do not react to data events. If you want your components to update based on events passed into ADF Business Components, then you need to use the Active Data Proxy.

> **Tip:** If your application uses BAM for the business service, then you do not need to use the Active Data Proxy.

> **Note:** If your business service requires the use of the Active Data Proxy, then you can only use the following components with active data:
>
> - `table`
> - `tree`
> - `treeTable`
> - DVT graph, geographical map, and gauge components

You need to create a Java class that subclasses one of the following decorator classes:

- `ActiveCollectionModelDecorator` class
- `ActiveDataModelDecorator` class (for use with graphs)
- `ActiveGeoMapDataModelDecorator` class
- `ActiveGaugeDataModelDecorator` class

These classes are wrapper classes that delegate the active data functionality to a default implementation of `ActiveDataModel`. The `ActiveDataModel` class listens for data change events and interacts with the Event Manager. Specifically, it does the following:

- Starts and stops the active data and the `ActiveDataModel` object, and registers and unregisters listeners to the data source.
- Wraps the JSF model interface. For example, the `ActiveCollectionModelDecorator` class wraps the `CollectionModel` class.
- Generates active data events based on data change events from the data source.
- Manages listeners from the Event Manager and pushes active data events to the Event Manager.

You need to implement methods on this Java class that registers itself as the listener of the active data source and gets the model to which the data is being sent.

Note: The Active Data framework does not support complicated business logic or transformations that require the ADF runtime context, such as a user profile or security. For example, the framework cannot convert an ADF context locale-dependent value and return a locale-specific value.

As an example of complicated business logic, say you have logic that allows a user to move an order from `open` status to `pending`. This change results in an `update` event, which should cause the order to be removed from a data object called "Open Orders." The framework cannot handle this event type transformation based on business logic. Instead, you need to have your data source handle this before publishing the data change event.

Before you begin:

Implement the logic to fire the active data events asynchronously from the data source. For example, this logic might be a business process that updates the database, or a JMS client that gets notified from JMS.

To use the active data service:

1. Create a Java class that extends the decorator class appropriate for your component. Example 45–3 shows a class created for a table.

Example 45–3 Extend the Decorator Class

```
package view;

import oracle.adf.view.rich.model.ActiveCollectionModelDecorator;
import oracle.adf.view.rich.activedata.ActiveDataEventUtil;
import oracle.adf.view.rich.activedata.JboActiveDataEventUtil;

/**
 * This code wraps the existing collection model in the page and implements the
   ActiveDataModel interface to enable ADS for the existing page.
 */
public class DeptModel
  extends ActiveCollectionModelDecorator
{
}
```

2. Implement the method that returns the model. Example 45–4 shows an implementation of the `getCollectionModel()` method that relies on expression language (EL) to avoid needing to typecast to an internal class. The method returns the `DepartmentsView1` collection from the binding container.

Example 45–4 Return the Model

```
public CollectionModel getCollectionModel()
{
   if (_model == null)
   {
       FacesContext fc = FacesContext.getCurrentInstance();
       Application app = fc.getApplication();
       ExpressionFactory elFactory = app.getExpressionFactory();
       ELContext el = fc.getELContext();

       // This is EL to avoid typecasting to an Internal class.
```

```
        ValueExpression ve = elFactory.createValueExpression(el,
                    "#{bindings.DepartmentsView1.collectionModel}", Object.class);

        // Now GET the collectionModel
        _model = (CollectionModel)ve.getValue(el);
    }
    return _model;
}
```

3. Create an inner class that is your own implementation of an `ActiveDataModel` class, which the decorator can use to start and stop the active data and connect and disconnect from the data source. This class should also use the `changeCount` API to maintain data read consistency, as shown in Example 45–5. For more information, see Section 45.4.1, "What You May Need to Know About Read Consistency."

Example 45–5 Connect to the Data Source

```
public class MyActiveDataModel extends BaseActiveDataModel
{
    protected void startActiveData(Collection<Object> rowKeys,
                                   int startChangeCount)
    {
      _listenerCount.incrementAndGet();
      if (_listenerCount.get() == 1)
      {
        System.out.println("start up");

        Runnable dataChanger = new Runnable()
        {
          public void run()
          {
            System.out.println("MyThread starting.");
            try
            {
              Thread.sleep(2000);
              System.out.println("thread running");
              triggerDataChange(MyActiveDataModel.this);
            }
            catch (Exception exc)
            {
              System.out.println("MyThread exceptioned out.");
            }
            System.out.println("MyThread terminating.");
          }
        };
        Thread newThrd = new Thread(dataChanger);
        newThrd.start();
      }
    }

    protected void stopActiveData(Collection<Object> rowKeys)
    {
      _listenerCount.decrementAndGet();
      if (_listenerCount.get() == 0)
      {
        System.out.println("tear down");
      }
    }
```

```
    public int getCurrentChangeCount()
    {
      return _currEventId.get();
    }

    public void bumpChangeCount()
    {
      _currEventId.incrementAndGet();
    }

    public void dataChanged(ActiveDataUpdateEvent event)
    {
      fireActiveDataUpdate(event);
    }

    private final AtomicInteger _listenerCount = new AtomicInteger(0);
    private final AtomicInteger _currEventId = new AtomicInteger();
}
```

4. Implement the method that will return the `ActiveDataModel` class, as shown in Example 45–6.

Example 45–6 Return the ActiveDataModel

```
public ActiveDataModel getActiveDataModel()
  {
    return _activeDataModel;
  }
```

5. Create a method that creates application-specific events that can be used to insert or update data on the active model.

   Example 45–7 shows the `triggerDataChange()` method, which uses the active model (an instance of `MyActiveDataModel`) to create `ActiveDataUpdateEvent` objects to insert and update data. You will need to import `oracle.adf.view.rich.activedata.ActiveDataEventUtil` and `oracle.adf.view.rich.activedata.JboActiveDataEventUtil` to implement this method.

Example 45–7 Create Event Objects to Update Model

```
public void triggerDataChange(MyActiveDataModel l)
  throws Exception
  {


    // do an update on dept 1
      l.bumpChangeCount();
      ActiveDataUpdateEvent event =
        ActiveDataEventUtil.buildActiveDataUpdateEvent(ActiveDataEntry.ChangeType.UPDATE,
                                    l.getCurrentChangeCount(),
                                    JboActiveDataEventUtil.convertKeyPath(new Key
                                           (new Object[]{ new Long(1), new Integer(0) })),
                                    null,
                                    new String[] { "name" },
                                    new Object[] { "Name Pushed" });

      l.dataChanged(event);
```

```
try
{
  Thread.sleep(2000);
}
catch (InterruptedException ie)
{
  ie.printStackTrace();
}
// insert dept 99

  l.bumpChangeCount();
   event =
    ActiveDataEventUtil.buildActiveDataUpdateEvent(ActiveDataEntry.ChangeType.INSERT_AFTER,
                                l.getCurrentChangeCount(),
                                JboActiveDataEventUtil.convertKeyPath(new Key
                                      (new Object[]{ new Long(99), new Integer(0) })),
                                JboActiveDataEventUtil.convertKeyPath(null),
                                new String[]{ "addr1", "name" },
                                new Object[]{ "Addr Inserted", "Name Inserted" });

  l.dataChanged(event);


try
{
  Thread.sleep(2000);
}
catch (InterruptedException ie)
{
  ie.printStackTrace();
}

// delete dept 10

  l.bumpChangeCount();
   event =
    ActiveDataEventUtil.buildActiveDataUpdateEvent(ActiveDataEntry.ChangeType.REMOVE,
                                l.getCurrentChangeCount(),
                                JboActiveDataEventUtil.convertKeyPath(new Key
                                      (new Object[]{ new Long(9), new Integer(0) })),
                                null, null, null);

  l.dataChanged(event);


try
{
  Thread.sleep(2000);
}
catch (InterruptedException ie)
{
  ie.printStackTrace();
}

// refresh the entire table

  l.bumpChangeCount();
   event =
    ActiveDataEventUtil.buildActiveDataUpdateEvent(ActiveDataEntry.ChangeType.REFRESH,
                                l.getCurrentChangeCount(),
```

```
                                    null, null, null, null);

      l.dataChanged(event);
    }


  private MyActiveDataModel _activeDataModel = new MyActiveDataModel();

  private CollectionModel _model = null;

}
```

45.4.1 What You May Need to Know About Read Consistency

Using active data means that your component has two sources of data: the active data feed and the standard data fetch. Because of this, you must make sure your application maintains read consistency.

For example, say your page contains a table and that table has active data enabled. The table has two methods of delivery from which it updates its data: normal table data fetch and active data push. Say the back end data changes from `foo` to `bar` to `fred`. For each of these changes, an active data event is fired. If the table is refreshed *before* those events hit the browser, the table will display `fred` because standard data fetch will always get the latest data. But then, because the active data event might take longer, some time *after* the refresh the data change event would cause `foo` to arrive at the browser, and so the table would update to display `foo` instead of `fred` for a period of time. Therefore, you must implement a way to maintain the read consistency.

To achieve read consistency, the `ActiveDataModel` has the concept of a *change count*, which effectively timestamps the data. Both data fetch and active data push need to maintain this `changeCount` object by monotonically increasing the count, so that if any data returned has a lower `changeCount`, the active data event can throw it away. Example 45–7 shows how you can use your implementation of the `ActiveDataModel` class to maintain read consistency.

45.5 Using the Active Data with a Scalar Model

ADF components that display collection-based data can be configured to work with ADS and require no extra setup in the view layer. However, imagine that your JSPX page uses an `activeOutputText` component to display new counts based on a Java timer. In this case, you will replace the model layer with scalar or "flat" data that you display from a Java Bean.

To implement the scalar model, follow these basic steps (as illustrated in the Example 45–8):

1. Use the `ActiveModelContext` API to register the bean with ADS so the bean (a scalar model) imitates an actual model.
2. Implement a custom mechanism to push the data to the view layer.

Example 45–8 Implement the Scalar Model in the Java Bean

```
package oracle.adfdemo.view.feature.rich;

import java.util.Collection;
import java.util.Timer;
import java.util.TimerTask;
```

```
import java.util.concurrent.atomic.AtomicInteger;

import oracle.adf.view.rich.activedata.ActiveModelContext;
import oracle.adf.view.rich.activedata.BaseActiveDataModel;
import oracle.adf.view.rich.activedata.ActiveDataEventUtil;
import oracle.adf.view.rich.event.ActiveDataEntry;
import oracle.adf.view.rich.event.ActiveDataUpdateEvent;


public class CounterBean extends BaseActiveDataModel
// Example using a Java timer to create new counts
{
  public CounterBean()
  // 1. Use the ActiveModelContext API to register the scalar model key path for
  //    the "state" attribute.
  {
    ActiveModelContext context = ActiveModelContext.getActiveModelContext();
    Object[] keyPath = new String[0];
    context.addActiveModelInfo(this, keyPath, "state");

    timer.schedule(new UpdateTask(), 2000, 2000);
  }

  public String getState()
  {
    return String.valueOf(counter);
  }

  // Not needed. We do not need to connect to a (real) active data scource.
  protected void startActiveData(Collection<Object> rowKeys, int
          startChangeCount) {}

  // Not needed. We do not need to connect to a (real) active data scource.

  protected void stopActiveData(Collection<Object> rowKeys) {}

  public int getCurrentChangeCount()
  {
    return counter.get();
  }

  protected class UpdateTask extends TimerTask
  {
    public void run()
    {
      counter.incrementAndGet();

      // 2. Use ActiveDataEventUtil to create an event object to update the model.
      ActiveDataUpdateEvent event =
        ActiveDataEventUtil.buildActiveDataUpdateEvent(
          ActiveDataEntry.ChangeType.UPDATE,
          counter.get(),
          new String[0],
          null,
          new String[] { "state" },
          new Object[] { counter.get() });
      fireActiveDataUpdate(event);
    }
  }
```

```
  private static final Timer timer = new Timer();
  private final AtomicInteger counter = new AtomicInteger(0);
}
```

After you create the bean, register the bean as a managed bean in the `faces-config.xml` file (as illustrated for `counterBean` in Example 45–9):

Example 45–9 Register the Managed Bean

```
...
<managed-bean>
  <managed-bean-name>counterBean</managed-bean-name>
  <managed-bean-class>
      oracle.adfdemo.view.feature.rich.CounterBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Once the bean is registered, you can use ADS to stream the data to the view layer. Your ADF Faces component use expression language to receive the pushed data (as illustrated by the `activeOutputText` component `value` attribute in Example 45–10):

Example 45–10 Display the Active Data

```
...
<f:view>
   <af:document title="Active Data Visual Design Demo"
                binding="#{templateBindings.documentComponent}"
               smallIconSource="#{aboutBean.smallIconSource}"
               largeIconSource="#{aboutBean.largeIconSource}" theme="dark"
                id="d1">
      <af:pageTemplate id="dmoTpl" viewId="#{templates.componentTemplate}">
        <f:attribute name="tagName" value="Active Data Visual Design"/>
        <f:attribute name="demoKind" value="visualDesign"/>
        <f:attribute name="customEditorPresent" value="true"/>
        <f:facet name="center">
          <af:panelGroupLayout layout="scroll">
             <af:activeOutputText
                value="#{counterBean.state}"
                inlineStyle=
                   "color:brown;
                    font-size:100px;
                    font-weight:bold;
                    text-align:center;"
              />
          </af:panelGroupLayout>
        </f:facet>
      </af:pageTemplate>
   </af:document>
</f:view>
```

Part VIII

Appendices

Part VII contains the following appendices:

- Appendix A, "Oracle ADF XML Files"
- Appendix B, "Oracle ADF Binding Properties"
- Appendix C, "ADF Security Permission Grants"
- Appendix D, "Most Commonly Used ADF Business Components Methods"
- Appendix E, "ADF Business Components Java EE Design Pattern Catalog"
- Appendix F, "ADF Equivalents of Common Oracle Forms Triggers"
- Appendix G, "Performing Common Oracle Forms Tasks in Oracle ADF"
- Appendix H, "Data Controls in Fusion Web Applications"

A

Oracle ADF XML Files

This appendix provides reference for the Oracle ADF metadata files that you create in your data model and user interface projects. You may use this information when you want to edit the contents of the metadata these files define.

This appendix includes the following sections:

- Section A.1, "Introduction to the ADF Metadata Files"
- Section A.2, "ADF File Overview Diagram"
- Section A.3, "ADF File Syntax Diagram"
- Section A.4, "adfm.xml"
- Section A.5, "modelProjectName.jpx"
- Section A.6, "bc4j.xcfg"
- Section A.7, "DataBindings.cpx"
- Section A.8, "pageNamePageDef.xml"
- Section A.9, "adfc-config.xml"
- Section A.10, "task-flow-definition.xml"
- Section A.11, "adf-config.xml"
- Section A.12, "adf-settings.xml"
- Section A.13, "web.xml"
- Section A.14, "logging.xml"

A.1 Introduction to the ADF Metadata Files

Metadata files in the Oracle Fusion web application are structured XML files used by the application to:

- Specify the parameters, methods, and return values available to your application's Oracle ADF data control usages
- Create objects in the Oracle ADF binding context and define the runtime behavior of those objects
- Define configuration information about the UI components in JSF and Oracle ADF Faces
- Define application configuration information for the Java EE application server

In the case of ADF bindings, you can use the binding-specific editors to customize the runtime properties of the binding objects. You can open a binding editor when you display the Structure window for a page definition file and choose **Properties** from the context menu.

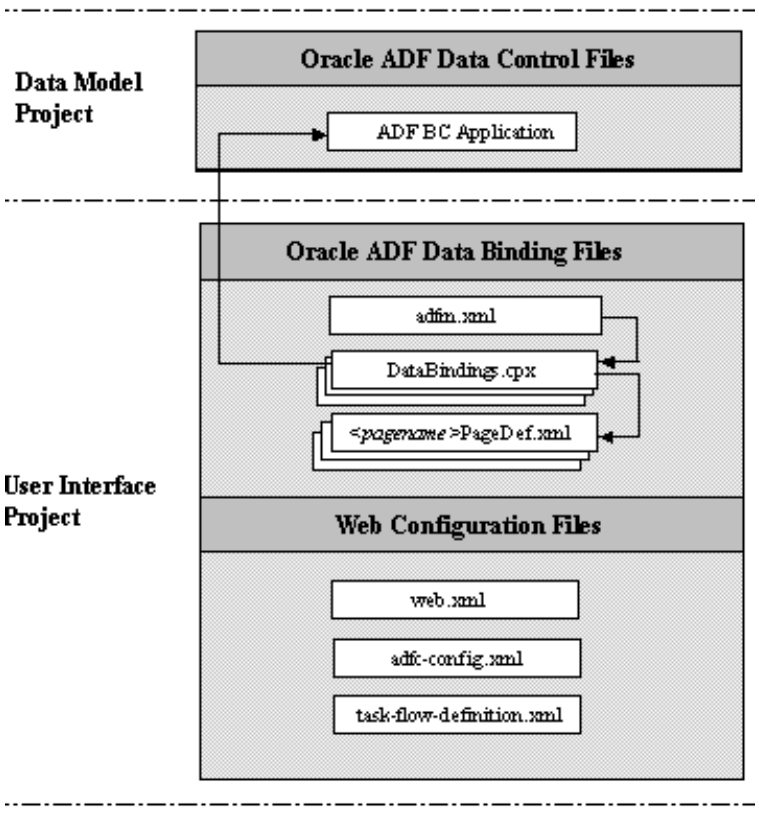
Additionally, you can view and edit the contents of any metadata file in JDeveloper's XML editor. The easiest way to work with these file is through the Structure window and Property Inspector. In the Structure window, you can select an element and in the Property Inspector, you can define attribute values for the element, often by choosing among dropdown menu choices. Use this reference to learn the choices you can select in the case of the Oracle ADF-specific elements.

A.2 ADF File Overview Diagram

The relationship between the Oracle ADF metadata files defines dependencies between the model data and the user interface projects. The dependencies are defined as file references within XML elements of the files.

Figure A–1 illustrates the hierarchical relationship of the XML metadata files that you might work with in a Fusion web application that uses an ADF Business Components application module as a service interface to JSF web pages.

Figure A–1 Oracle ADF File Hierarchy Overview for the Fusion Web Application



A.2.1 Oracle ADF Data Control Files

In an ADF Business Components application, the data control implementation files are contained within the application. The application module and view object XML component descriptor files provide the references for the data control. These files, in

conjunction with the `bc4j.xcfg` file, provide the necessary information for the data control.

An application that uses ADF Business Components in one project and a non-ADF Business Components data control in another project may have a `DataControls.dcx` file, as well as supporting *<sessionbeanname>*`.xml` and *<beanname>*`.xml` files.

A.2.2 Oracle ADF Data Binding Files

These standard XML configuration files for a Fusion web application appear in your user interface project:

- `adfm.xml`: This file lists the `DataBindings.cpx` file that is available in the current project.

  See Section A.4, "adfm.xml" for more information.

- `DataBindings.cpx` : This file contains the page map, page definitions references, and data control references. The file is created the first time you create a data binding for a UI component (either from the Structure window or from the Data Controls Panel). The `DataBindings.cpx` file defines the Oracle ADF binding context for the entire application. The binding context provides access to the bindings and data controls across the entire application. The `DataBindings.cpx` file also contains references to the *<pagename>*`PageDef.xml` files that define the metadata for the Oracle ADF bindings in each web page.

  For more information, see Section A.7, "DataBindings.cpx".

- *<pagename>*`PageDef.xml`: This is the page definition XML file. It associates web page UI components with data, or data controls. JDeveloper creates this file each time you design a new web page using the Data Controls Panel or Structure window. These XML files contain the metadata used to create the bindings that populate the data in the web page's UI components. For every web page that refers to an ADF binding, there must be a corresponding page definition file with binding definitions.

  For more information, see Section A.8, "pageNamePageDef.xml".

A.2.3 Web Configuration Files

These standard XML configuration files required for a JSF application appear in your user interface project:

- `web.xml`: Part of the application's configuration is determined by the contents of its Java EE application deployment descriptor, `web.xml`. The `web.xml` file defines everything about your application that a server needs to know. The file plays a role in configuring the Oracle ADF data binding by setting up the `ADFBindingFilter`. Additional runtime settings include servlet runtime and initialization parameters, custom tag library location, and security settings.

  For more information about ADF data binding and JSF configuration options, see Section A.13, "web.xml".

  An ADF Faces application typically uses its own set of configuration files in addition to `web.xml`. For more information, see the "Configuration in trinidad-config.xml" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

- `adfc-config.xml`: The configuration file for an ADF unbounded task flow. The configuration file contains metadata about the activities and control flows contained in the unbounded task flow. The default name for this file is `adfc-config.xml`, but an end user can change the name.

  For more information, see Section A.9, "adfc-config.xml".

- `task-flow-definition.xml`: The configuration file for an ADF bounded task flow. The configuration file contains metadata about the activities and control flows contained in the bounded task flow. The default name for this file can be `task-flow-defintion.xml` or whatever an end user specifies in the **Create ADF Task Flow** dialog. The same application can contain multiple task flow definition files.

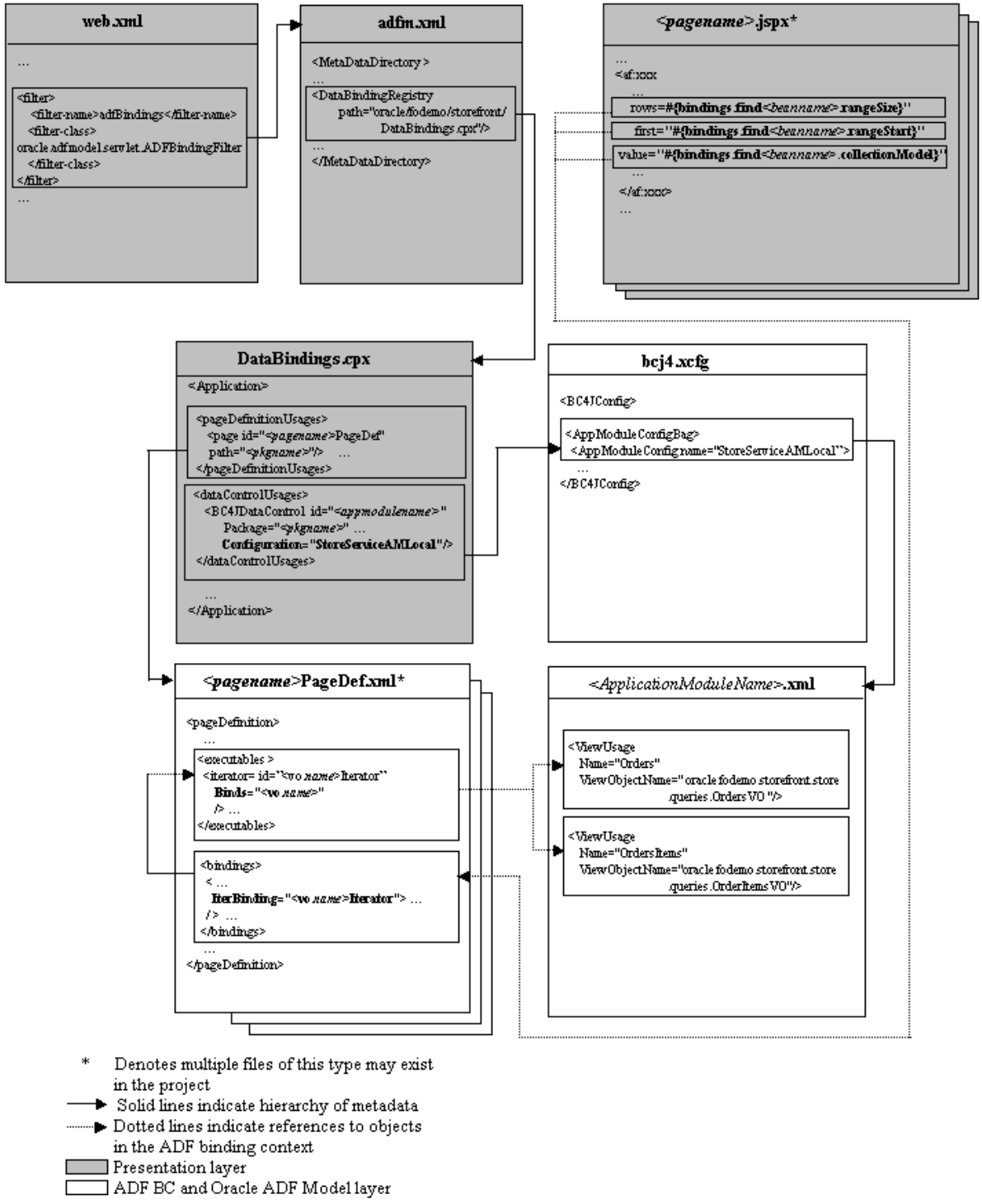
  For more information, see Section A.10, "task-flow-definition.xml".

A.3 ADF File Syntax Diagram

Figure A–2 illustrates the hierarchical relationship of the XML metadata files that you might work with in a web application that uses an ADF application module as a service interface to ADF Business Components. At runtime, the objects created from these files interact in this sequence:

1. When the first request for an ADF databound web page occurs, the servlet registers the Oracle ADF servlet filter `ADFBindingFilter` named in the `web.xml` file.
2. The binding filter creates an empty binding context.
3. When a page is rendered, the binding filter asks the binding context to load a corresponding `PageDef.xml` for the page.
4. The binding context creates the binding container by loading the `<page>` file as referenced by the `<pagemap>` element in the `DataBindings.cpx` file.
5. The `adfm.xml` file loads the `DataBindings.cpx` contents and finds the right `PageDef.xml` based on the `<pagemap>` element reference to the `<pageDefinitionUsage>` element.
6. The binding container's `prepareModel` phase prepares and refreshes all relevant executables (most are marked `deferred` by default).
7. An iterator binding gets executed by referencing the named method on the data control found through the data control factory named in the case of ADF Business Components in the `bc4j.xcfg` file.
8. The binding container also creates the bindings defined in the `<bindings>` section of the *pagename*`PageDef.xml` file for the mapped web page.
9. The web page references to ADF bindings through EL using the expression `#{bindings}` are resolved by accessing the binding container of the page.
10. The page pulls the available data from the bindings in the binding container.

Figure A–2 Oracle ADF File Hierarchy and Syntax Diagram for an ADF BC-based Web Application



A.4 adfm.xml

The `adfm.xml` file contains the classpath-relative paths for the `.cpx`, `.dcx`, `.jpx`, and `.xcfg` files in each design time project that is included in the runtime deployed application. The `adfm.xml` file operates as a dynamically maintained "Registry of

Registries" that is used to quickly find all `.cpx`, `.dcx`, `.jpx`, and `.xcfg` files (which are themselves registries of metadata).

The file registry is used extensively by the ADF Library resource catalog browsing implementations, by the ADF model layer design time, and at runtime during merge and discovery.

When a developer creates a binding on a page, JDeveloper adds metadata files (for example, page definitions) in the project source tree. The `adfm.xml` file then notes the location of each.

When a project is built, the `adfm.xml` file is put in *project-root*`/adfmsrc/META-INF/adfm.xml`. The project-level archive deployment profiles locate the file at `META-INF/adfm.xml`.

At runtime, the application classpath is scanned to build the list of `.cpx` files that comprise the application. The application then loads each `.cpx` as needed to create the binding context. For details about the ADF model layer usage, see Section 13.4.2, "What Happens When You Use the Data Controls Panel".

Four types of sub registries are recorded by the `adfm.xml` file:

- `DataBindingRegistry` (`.cpx`)
- `DataControlRegistry` (`.dcx`)
- `BusinessComponentServiceRegistry` (`.xcfg`)
- `BusinessComponentProjectRegistry` (`.jpx`)

A.5 *modelProjectName*.jpx

The `.jpx` file contains configuration information that JDeveloper uses in the design time to allow you to create the data model project with ADF Business Components. It also contains metadata that defines how a shared application module is used at runtime. Because the shared application module can be accessed by any data model project in the same Fusion web application, JDeveloper maintains the scope of the shared application module in the ADF Business Components project configuration file.

This file is saved in the `src` directory of the project. For example, if you look at the `StoreFrontService.jpx` file in the `./src/model` subdirectory of the Fusion Order Demo application's `StoreFrontService` project, you will see the `SharedLookupService` application module's usage definition. For details about the shared application module usage, see Section 10.2.2, "What Happens When You Define a Shared Application Module".

Example A–2 displays a sample default `.jpx` file.

Example A–1 Sample .jpx File

```
<JboProject
   xmlns="http://xmlns.oracle.com/bc4j"
   Name="StoreFrontService"
   Version="11.1.1.49.73"
   SeparateXMLFiles="true"
   PackageName="">
   <DesignTime>
      <Attr Name="_appModuleNames0"
         Value="oracle.fodemo.storefront.lookups.LookupServiceAM"/>
      <Attr Name="_domainNames0"
         Value="oracle.fodemo.storefront.entities.formatters.UppercaseOracleStyleDate"/>
      <Attr Name="_jprName" Value="../StoreFrontService.jpr"/>
```

```
   <Attr Name="_appModuleNames1"
      Value="oracle.fodemo.storefront.store.service.StoreServiceAM"/>
   <Attr Name="_NamedConnection" Value="FOD"/>
</DesignTime>
<Containee
   Name="links"
   FullName="oracle.fodemo.storefront.account.queries.links.links"
   ObjectType="JboPackage">
   <DesignTime>
      <Attr Name="_VO" Value="true"/>
      <Attr Name="_VL" Value="true"/>
</DesignTime>
</Containee>
<Containee
   Name="queries"
   FullName="oracle.fodemo.storefront.account.queries.queries"
   ObjectType="JboPackage">
   <DesignTime>
      <Attr Name="_VO" Value="true"/>
   </DesignTime>
</Containee>
<Containee
   Name="associations"
   FullName="oracle.fodemo.storefront.entities.associations.associations"
   ObjectType="JboPackage">
   <DesignTime>
      <Attr Name="_AS" Value="true"/>
   </DesignTime>
</Containee>
<Containee
   Name="entities"
   FullName="oracle.fodemo.storefront.entities.entities"
   ObjectType="JboPackage">
   <DesignTime>
      <Attr Name="_EO" Value="true"/>
   </DesignTime>
</Containee>
<Containee
   Name="formatters"
   FullName="oracle.fodemo.storefront.entities.formatters.formatters"
   ObjectType="JboPackage">
   <DesignTime>
      <Attr Name="_DO" Value="true"/>
   </DesignTime>
</Containee>
<Containee
   Name="lookups"
   FullName="oracle.fodemo.storefront.lookups.lookups"
   ObjectType="JboPackage">
   <DesignTime>
      <Attr Name="_VO" Value="true"/>
      <Attr Name="_AM" Value="true"/>
   </DesignTime>
</Containee>
<Containee
   Name="links"
   FullName="oracle.fodemo.storefront.store.queries.links.links"
   ObjectType="JboPackage">
   <DesignTime>
      <Attr Name="_VL" Value="true"/>
```

```
      </DesignTime>
   </Containee>
   <Containee
      Name="queries"
      FullName="oracle.fodemo.storefront.store.queries.queries"
      ObjectType="JboPackage">
      <DesignTime>
         <Attr Name="_VO" Value="true"/>
      </DesignTime>
   </Containee>
   <Containee
      Name="service"
      FullName="oracle.fodemo.storefront.store.service.service"
      ObjectType="JboPackage">
      <DesignTime>
         <Attr Name="_AM" Value="true"/>
      </DesignTime>
   </Containee>
   <AppModuleUsage
      Name="SharedLookupService"
      FullName="oracle.fodemo.storefront.lookups.LookupServiceAM"
      ConfigurationName="oracle.fodemo.storefront.lookups.null"
      SharedScope="1"/>
</JboProject>
```

A.6 bc4j.xcfg

The `bc4j.xcfg` file contains metadata information about application module names, the database connection used by the application module, and the runtime parameters the user has configured for the application module.

The `bc4j.xcfg` file is located in the `./common` subdirectory relative to the application module's XML component definition. All of the configurations for all of the application modules in a single Java package are saved in that same file. For example, if you look at the `bc4j.xcfg` file in the `./classes/oracle/fodemo/storefront/store/service/common` directory of the Fusion Order Demo application's `StoreFront` project, you will see the three named configurations for its `StoreServiceAM` application module. For details about editing the configurations, see Section 9.3.4, "How to Change Your Application Module's Runtime Configuration" and Section 44.2, "Setting Pool Configuration Parameters".

Example A–2 displays a sample `bc4j.xcfg` file from the Fusion Order Demo application.

Example A–2 Sample bc4j.xcfg File

```
<BC4JConfig version="11.1" xmlns="http://xmlns.oracle.com/bc4j/configuration">
   <AppModuleConfigBag ApplicationName="oracle.fodemo.storefront.store.service.StoreServiceAM">
      <AppModuleConfig
            DeployPlatform="LOCAL"
            JDBCName="FOD"
            jbo.project="StoreFrontService"
            name="StoreServiceAMLocal"
            ApplicationName="oracle.fodemo.storefront.store.service.StoreServiceAM">
         <Database jbo.locking.mode="optimistic"/>
         <Security AppModuleJndiName="oracle.fodemo.storefront.store.service.StoreServiceAM"/>
      </AppModuleConfig>
```

```
      <AppModuleConfig
               DeployPlatform="LOCAL"
               JDBCName="FOD"
               jbo.project="StoreFrontService"
               name="StoreServiceAMLocalWeb"
               ApplicationName="oracle.fodemo.storefront.store.service.StoreServiceAM">
         <AM-Pooling jbo.ampool.initpoolsize="1"/>
         <Database jbo.locking.mode="optimistic"/>
         <Security AppModuleJndiName="oracle.fodemo.storefront.store.service.StoreServiceAM"/>
         <Custom fod.application.issoaenabled="true"/>
      </AppModuleConfig>
      <AppModuleConfig
               name="StoreFrontService"
               ApplicationName="oracle.fodemo.storefront.store.service.StoreServiceAM"
               jbo.project="StoreFrontService"
               DeployPlatform="SI">
         <AM-Pooling jbo.ampool.resetnontransactionalstate="true"/>
         <Database jbo.SQLBuilder="ORACLE" jbo.locking.mode="optimistic"
                   jbo.TypeMapEntries="Java"/>
         <Security AppModuleJndiName="oracle.fodemo.storefront.store.service.StoreServiceAM"/>
         <Custom JDBCDataSource="java:comp/env/jdbc/FODDS"/>
      </AppModuleConfig>
   </AppModuleConfigBag>
</BC4JConfig>
```

A.7 DataBindings.cpx

The `DataBindings.cpx` file is created in the user interface project the first time you drop a data control usage onto a web page in the visual editor. The `DataBindings.cpx` file defines the Oracle ADF binding context for the entire application and provides the metadata from which the Oracle ADF binding objects are created at runtime. It is used extensively by the ADF Library Resource Palette browsing implementations, and also by the `.cpx` and `.dcx` design time and runtime merge and discovery. When you insert a databound UI component into your document, the page will contain binding expressions that access the Oracle ADF binding objects at runtime.

If you are familiar with building Fusion web applications in earlier releases of JDeveloper, you'll notice that the `.cpx` file no longer contains all the information copied from the `DataControls.dcx` file, but only a reference to it. If you need to make changes to the `.cpx` file, you must edit the `DataControls.dcx` file.

The `DataBindings.cpx` file appears in the `/src` directory of the user interface project. When you double-click the file node, the binding context description appears in the XML source editor. (To edit the binding context parameters, use the Property Inspector and select the desired parameter in the Structure window.)

A.7.1 DataBindings.cpx Syntax

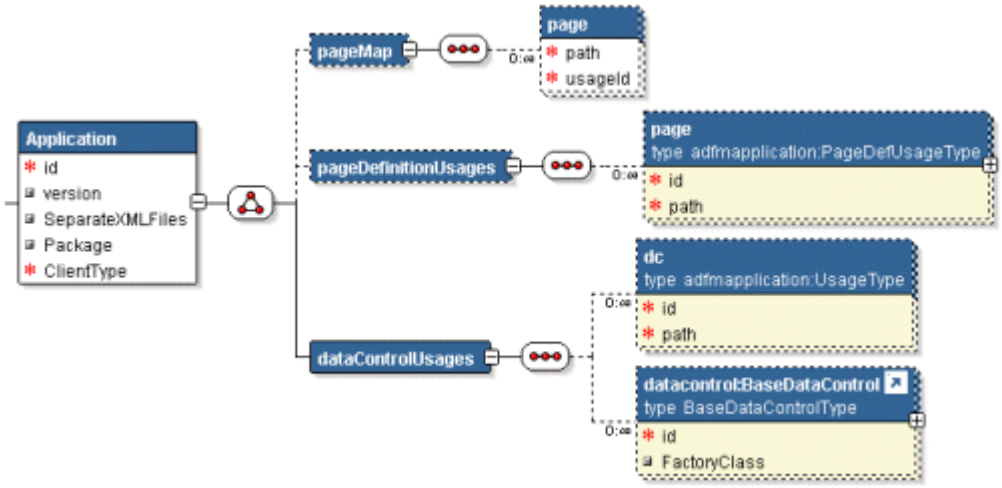
The top level element of the `DataBindings.cpx` file is `<DataControlConfigs>`:

```
<?xml version = '1.0' encoding = 'UTF-8'?>
<BC4JConfig version="11.0" xmlns="http://xmlns.oracle.com/bc4j/configuration">
```

where the XML namespace attribute (`xmlns`) specifies the URI to which the data controls bind at runtime. Only the package name is editable; all other attributes should have the values shown.

Figure A–3 displays the child element hierarchy of the <DataControlConfigs> element. Note that each business service for which you have created a data control will have its own <dataControlUsages> definition.

Figure A–3 Schema for the Structure Definition of the DataBindings.cpx File



The child elements have the following usages:

- <definitionFactories> registers a factory class to create the ADF binding objects associated with a particular namespace at runtime. The factory class is specific to the namespace associated with the type of ADF binding (for instance, a task flow binding).
- <pageMap> maps all user interface URLs and the corresponding page definition usage name. This map is used at runtime to map a URL to its page definition.
- <pageDefinitionUsages> maps a page definition usage (BindingContainer instance) name to the corresponding page definition. The id attribute represents the usage ID. The path attribute represents the full path to the page definition.
- <dataControlUsages> declares a list of data control usages (shortnames) and corresponding path to the data control definition entries in the .dcx or .xcfg file.

Table A–1 describes the attributes of the DataBindings.cpx elements.

Table A–1 Attributes of the DataBindings.cpx File Elements

Element Syntax	Attributes	Attribute Description
<definitionFactories> <factory/> </definitionFactories>	nameSpace	A URI. Identifies the location of the executable elements in the page definition usage.
	className	The fully qualified class name. Identifies the location of the factory class that creates the page definition usage objects.
<pageMap> <page /> </pageMap>	path	The full directory path. Identifies the location of the user interface page.

Table A–1 (Cont.) Attributes of the DataBindings.cpx File Elements

Element Syntax	Attributes	Attribute Description
	`usageId`	A unique qualifier. Names the page definition ID that appears in the ADF page definition file. The ADF binding servlet looks at the incoming URL requests and checks that the `bindings` variable is pointing to the ADF page definition associated with the URL of the incoming HTTP request.

`<pageDefinitionUsages>`
`  <page/>`
`</pageDefinitionUsages>`
	`id`	A unique qualifier. References the page definition ID that appears in the ADF page definition file.

	`path`	The fully qualified package name. Identifies the location of the user interface page's ADF page definition file.

`<dataControlUsages>`
`  <dc.../>`
`</dataControlUsages>`
	`id`	A unique qualifier. Identifies the data control usage as is defined in the `DataControls.dcx` file.

	`path`	The fully qualified package name. Identifies the location of the data control.

A.7.2 DataBindings.cpx Sample

Example A–3 shows the syntax for the `DataBindings.cpx` file in the Fusion Order Demo application.

The ADF executable definition factory (`factory` element) is named by a `className` attribute and is associated with a `namespace`. At runtime, the factory class creates the executable definition objects that leads to the creation of the binding objects for the ADF binding container associated with a particular page definition. The factory locates the page definition through two `DataBindings.cpx` file elements: the `pageMap` element that maps the page URL to the page definition ID (`usageId` attribute) assigned at design time and the `pageDefinitionUsages` element that maps the ID to the location of the page definition from the project or project classpath.

Additionally, the ADF Business Components data control (`BC4JDataControl` element) is named by the `id` attribute. The combination of the `Package` attribute and the `Configuration` attribute is used to locate the `bc4j.xcfg` file in the `./common` subdirectory of the indicated package. The configuration contains the information of the application module name and all the runtime parameters the user has configured.

Example A–3 Sample DataBindings.cpx File

```
<Application xmlns="http://xmlns.oracle.com/adfm/application"
             version="11.1.1.44.61" id="DataBindings" SeparateXMLFiles="false"
             Package="oracle.fodemo.storefront" ClientType="Generic"
             ErrorHandlerClass="oracle.fodemo.frmwkext.FODCustomErrorHandler">
  <definitionFactories>
    <factory nameSpace="http://xmlns.oracle.com/adf/controller/binding"
             className="oracle.adf.controller.internal.binding.
                                        TaskFlowBindingDefFactoryImpl"/>
    <factory nameSpace="http://xmlns.oracle.com/adfm/dvt"
             className="oracle.adfinternal.view.faces.dvt.model.binding.
                                        FacesBindingFactory"/>
  </definitionFactories>
  <pageMap>
    <page path="/home.jspx" usageId="homePageDef"/>
```

```
    ...
  </pageMap>
  <pageDefinitionUsages>
    <page id="homePageDef"
          path="oracle.fodemo.storefront.pageDefs.homePageDef"/>
    ...
  </pageDefinitionUsages>
  <dataControlUsages>
    <BC4JDataControl id="StoreServiceAMDataControl"
                     Package="oracle.fodemo.storefront.store.service"
                     FactoryClass="oracle.adf.model.bc4j.DataControlFactoryImpl"
                     SupportsTransactions="true" SupportsFindMode="true"
                     SupportsRangesize="true" SupportsResetState="true"
                     SupportsSortCollection="true"
                     Configuration="StoreServiceAMLocalWeb" syncMode="Immediate"
                     xmlns="http://xmlns.oracle.com/adfm/datacontrol"/>
    ...
  </dataControlUsages>
</Application>
```

A.8 *pageName*PageDef.xml

The *pageName*`PageDef.xml` files are created each time you insert a databound component into a web page using the Data Controls Palette or Structure window. These XML files define the Oracle ADF binding container for each web page in the application. The binding container provides access to the bindings within the page. You will have one XML file for each databound web page.

> **Caution:** The `DataBindings.cpx` file maps JSF pages to their corresponding page definition files. If you change the name of a page definition file or a JSF page, JDeveloper does *not* automatically refactor the `DataBindings.cpx` file. You must manually update the page mapping in the `DataBindings.cpx` file.

The `PageDef.xml` file appears in the `/src/view` directory of the user interface project. The Application Navigator displays the file in the view package of the Application Sources node. When you double-click the file node, the page description appears in the XML source editor. To edit the page description parameters, use the Property Inspector and select the desired parameter in the Structure window.

For more information, see Chapter 13.7, "Working with Page Definition Files".

There are important differences in how the page definitions are generated for methods that return a single value and a collection.

A.8.1 PageDef.xml Syntax

The top-level element of the `PageDef.xml` file is `<pageDefinition>`:

```
<pageDefinition xmlns="http://xmlns.oracle.com/adfm/uimodel"
                version="10.1.3.35.83" id="<pagename>PageDef"
                Package="oracle.fod.view.pageDefs">
```

where the XML namespace attribute (`xmlns`) specifies the URI to which the ADF binding container binds at runtime. Only the package name is editable; all other attributes should have the values shown.

Example A–4 displays the child element hierarchy of the <pageDefinition> element. Note that each business service for which you have created a data control will have its own <AdapterDataControl> definition.

Example A–4 PageDef.xml Element Hierarchy

```
<?xml version="1.0" encoding="UTF-8" ?>
<pageDefinition>
    <parameters>
        ...
    </parameters>
    <executables>
        ...
    </executables>
    <bindings>
        ...
    </bindings>
</pageDefinition>
```

The child elements have the following usages:

- <parameters> defines page-level parameters that are EL accessible. These parameters store information local to the web page request and may be accessed in the binding expressions.
- <executables> defines the list of items (methods, view objects, and accessors) to execute during the prepareModel phase of the ADF page lifecycle. Methods to be executed are defined by <methodIterator>. The lifecycle performs the execute in the sequence listed in the <executables> section. Whether or not the method or operation is executed depends on its refresh or refreshCondition attribute value. Built-in operations on the data control are defined by:

  - <page> - definition for a nested page definition (binding container)

  - <iterator> - definition to a named collection in DataControls

  - <accessorIterator> - definition to get an accessor in a data control hierarchy

  - <methodIterator> - definition to get to an iterator returned by an invoked method defined by a methodAction in the same file

  - <variableIterator> - internal iterator that contains variables declared for the binding container

  - <invokeAction> - definition of which method to invoke as an executable
- <bindings> refers to an entry in <executables> to get to the collection from which bindings extract/submit attribute level data.

Table A–2 describes the attributes of the top-level <pageDefinition> element.

Table A–2 Attributes of the PageDef.xml File <pageDefinition> Element

Element Syntax	Attributes	Attribute Description
`<pageDefinition>`	`ControllerClass`	Fully qualified class name to create when controller requests a `PageController` object for this `bindingContainer`.
	`EnableTokenValidation`	Enables currency validation for this bindingContainer when a postback occurs. This is to confirm that the web tier state matches the state that particular page was rendered with.
	`FindMode`	FindMode is for legacy (10.1.2) use only and indicates whether this bindingContainer should start out in findMode when initially prepared.
	`MsgBundleClass`	Fully qualified package name. Identifies the class which contains translation strings for any bindings.
	`SkipValidation`	Determines if data validation occurs. The supported values are: ▪ `true`: skips data validation. Note that client-side binding level attribute validation still occurs. For example, validates non-null and type conversion errors. ▪ `false`: validates all rows for all data controls referenced in the current page. This is the default value. ▪ `skipDataControls`: validates the current rows of iterator bindings modified in the current page. ▪ `custom`: set to `custom` if your application implements an instance of the `oracle.binding.BindingContainerValidator` interface and references it through an EL expression entry named `CustomValidator` in the binding container. Setting a value for this attribute can be useful if you want to skip data validation on, for example, a train component. For more information see the "How to Create the Train Model" section of the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*
	`Viewable`	An EL expression that should resolve at runtime to whether this binding and the associated component should be rendered or not.

Table A–3 describes the attributes of the child element of `<parameters>`.

Table A–3 Attributes of the PageDef.xml File <parameters> Element

Element Syntax	Attributes	Attribute Description
`<parameter>`	`evaluate`	Specifies when the parameter should be evaluated: `eachUse`, `firstUse`, or `inPrepareModel`.
	`id`	Unique identifier. May be referenced by ADF bindings.
	`option`	Indicates the usage of the variable within the binding container: ▪ `Final` indicates that this parameter cannot be passed in by a usage of this binding container. It must use the default value in the definition. ▪ `Optional` indicates that the variable value need not be provided. ▪ `Mandatory` indicates that the variable value must be provided or a binding container exception will be thrown.
	`readonly`	Indicates whether the parameter value may be modified or not. Set to `true` when you do not want the application to modify the parameter value.
	`value`	A default value, which can be an EL expression.

Table A–4 describes the attributes of the `PageDef.xml` `<executables>` elements.

Table A–4 Attributes of the PageDef.xml File <executables> Element

Element Syntax	Attributes	Attribute Description
`<accessorIterator>`	`Accessor`	Specifies any other accessor defined by this binding.
	`Binds`	Specifies the view or action to which the iterator is bound.
	`BeanClass`	Identifies the Java type of beans in the associated iterator or collection.
	`CacheResults`	If `true`, manages the data collection between requests.
	`ChangeEventRate`	Specifies the rate of events when a component is wired to data via this iterator and is in polling event mode.
	`DataControl`	Interprets and returns the collection referred to by this iterator binding.
	`id`	Unique identifier. May be referenced by any ADF value binding.
	`MasterBinding`	Reference to the `methodIterator` (or iterator) that binds the data collection that serves as the master to the accessor iterator's detail collection.
	`ObjectType`	Used for ADF Business Components only. A boolean value determines whether the collection is an object type or not.
	`RangeSize`	Specifies the number of data objects in a range to fetch from the bound collection. The range defines a window you can use to access a subset of the data objects in the collection. By default, the range size is set to a range that fetches just ten data objects. Use `RangeSize` when you want to work with an entire set or when you want to limit the number of data objects to display in the page. Note that the values -1 and 0 have specific meaning: the value -1 returns all available objects from the collection, while the value 0 will return the same number of objects as the collection uses to retrieve from its data source.

Table A–4 (Cont.) Attributes of the PageDef.xml File <executables> Element

Element Syntax	Attributes	Attribute Description
	`Refresh`	Determines when and whether the executable should be invoked. Set one of the following properties as required: ■ `always` - causes the executable to be invoked each time the binding container is prepared. This will occur when the page is displayed and when the user submits changes, or when the application posts back to the page. ■ `deferred` (default for ADF faces applications) - refresh occurs when another binding requires or refers to this executable. Since refreshing an executable may be a performance concern, you can set the refresh to occur only if `deferred` is used in a binding that is being rendered. ■ `<default>` - Always set to Deferred by default. ■ `ifNeeded` (default for all view technologies other than ADF Faces) - whenever the framework needs to refresh the executable because it has not been refreshed to this point. For example, when you have an accessor hierarchy such that a detail is listed first in the page definition, the master could be refreshed twice (once for the detail and again for the master's iterator). Using ifNeeded gives the mean to avoid duplicate refreshes. This is the default behavior for executables. ■ `never` - when the application itself will call refresh on the executable during one of the controller phases and does not want the framework to refresh it at all. ■ `prepareModel` - causes the executable to be invoked each time the page's binding container is prepared. ■ `prepareModelIfNeeded` - causes the executable to be invoked during the prepareModel phase if this executable has not been refreshed to this point. See also ifNeeded above. ■ `renderModel` - causes the executable to be invoked each time the page is rendered. ■ `renderModelIfNeeded` - causes the executable to be invoked during the page's renderModel phase on the condition that it is needed. See also ifNeeded above. ■ `refreshAfter` - Use to handle dependencies between executables. For example, you can set the condition so that this executable refreshes after another executable.
	`RefreshCondition`	An EL expression that when resolved, determines when and whether the executable should be invoked. For example, `${!bindings.findAllServiceRequestIter.findModel}` resolves the value of the `findMode` on the iterator in the ADF binding context `AllServiceRequest`. Hint: Use the Property Inspector to create expressions from the available objects of the binding context (bindings namespace) or binding context (data namespace), JSF managed beans, and JSP objects.
	`RefreshAfter`	Specifies the condition after which the page should be refreshed.

Table A–4 (Cont.) Attributes of the PageDef.xml File <executables> Element

Element Syntax	Attributes	Attribute Description
	RowCountThreshold	Specify a value to determine if a result set returns the number of rows you specify as a value. If you set RowCountThreshold to 0, the iterator returns the estimated row count in the result set by executing the count query. If you set RowCountThreshold to less than 0, the iterator does not execute the count query. Set RowCountThreshold to a value greater than 0 if you want the iterator to execute the count query with the maximum value equal to the value you specify for RowCountThreshold. If the estimated row count is less than the value of RowCountThreshold, return the number of rows in the estimated row count. If the estimated row count is greater than the value of RowCountThreshold, return -1.
	Sortable	Specifies whether the iterator is sortable or not.
<invokeAction>	Binds	Determines the action to invoke. This may be on any actionBinding. Additionally, in the case, of the EJB session facade data control, you may bind to the finder method exposed by the data control. Built-in actions supported by the EJB session facade data control include: ■ Execute executes the bound action defined by the data collection. ■ Find retrieves a data object from a collection. ■ First navigates to the first data object in the data collection range. ■ Last navigates to the first data object in the data collection range. ■ Next navigates to the first data object in the data collection range. If the current range position is already on the last data object, then no action is performed. ■ Previous navigates to the first data object in the data collection range. If the current position is already on the first data object, then no action is performed. ■ setCurrentRowWithKey passes the row key as a String converted from the value specified by the input field. The row key is used to set the currency of the data object in the bound data collection. When passing the key, the URL for the form will not display the row key value. You may use this operation when the data collection defines a multipart attribute key. ■ setCurrentRowWithKeyValue is used as above, but when you want to use a primary key value instead of the stringified key.
	id	Unique identifier. May be referenced by any ADF action binding.
	Refresh	See Refresh for <accessorIterator>.
	RefreshCondition	See RefreshCondition for <accessorIterator>.
<iterator> and <methodIterator>	BeanClass	Identifies the Java type of beans in the associated iterator or collection.

Table A–4 (Cont.) Attributes of the PageDef.xml File <executables> Element

Element Syntax	Attributes	Attribute Description
	BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. Not used in current JDeveloper release.
	Binds	See Binds for <invokeAction>.
	CacheResults	See CacheResults for <accessorIterator>.
	ChangeEventRate	Specifies the rate of events when a component is wired to data via this iterator and is in polling event mode.
	DataControl	Name of the DataControl usage in the bindingContext (.cpx) which this iterator is associated with.
	DefClass	Used internally by ADF.
	id	Unique identifier. May be referenced by any ADF value binding.
	ObjectType	Not used by EJB session facade data control (used by ADF Business Components only).
	RangeSize	See RangeSize for <accessorIterator>.
	Refresh	See Refresh for <accessorIterator>.
	RefreshAfter	Specifies the condition after which the page should be refreshed.
	RefreshCondition	See RefreshCondition for <accessorIterator>.
	RowCountThreshold	See RowCountThreshold for <accessorIterator>.
<page> and <variableIterator>	id	Unique identifier. In the case of <page>, refers to nested page or region that is included in this page. In the case of the <variableIterator> executable, the identifier may be referenced by any ADF value binding.
	ChangeEventRate	Specifies the rate of events when a component is wired to data via this iterator and is in polling event mode.
	path	Used by <page> executable only. Advanced, a fully qualified path that may reference another page's binding container.
	Refresh	See Refresh for <accessorIterator>.
	RefreshAfter	Specifies the condition after which the page should be refreshed.
	RefreshCondition	See RefreshCondition for <accessorIterator>.

Table A–5 describes the attributes of the PageDef.xml <bindings> element.

Table A–5 Attributes of the PageDef.xml File <bindings> Element

Element Syntax	Attributes	Attribute Description
<action>	Action	Fully qualified package name. Identifies the class for which the data control is created. In the case of the EJB session facade, this is the session bean.
	BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. This is used by earlier versions of JDeveloper.
	DataControl	Name of the DataControl usage in the bindingContext (.cpx) which this iteratorBinding or actionBinding is associated with.
	Execute	Used by default when you drop an operation from the Data Controls Panel in the automatically configured ActionListener property. It results in executing the action binding's operation at runtime.
	InstanceName	Specifies the instance name for the action.
	IterBinding	Specifies the iteratorBinding instance in this bindingContainer to which this binding is associated.
	Outcome	Use if you want to use the result of a method action binding (once converted to a String) as a JSF navigation outcome name.
<attributeValues>	ApplyValidation	Set to true by default. When true, controlBinding executes validators defined on the binding. You can set to false in the case of ADF Business Components, when running in local mode and the same validators are already defined on the corresponding attribute.
	BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. This is used by earlier versions of JDeveloper.
	ChangeEventPolicy	Specifies the event strategy for the component when run with ADS (Active Data Services). Can be specified as push poll ppr none
	ControlClass	Used internally by ADF.
	CustomInputHandler	This is the class name for a oracle.jbo.uicli.binding.JUCtrlValueHandler implementation that is used to process the inputValue for a given value binding.
	DefClass	Used internally by ADF.
	id	Unique identifier. May be referenced by any ADF action binding.
	IterBinding	Refers to the iteratorBinding instance in this bindingContainer to which this binding is associated.
	NullValueId	Refers to the entry in the message bundle for this bindingContainer that contains the String to indicate the null value in a list display.

Table A–5 (Cont.) Attributes of the PageDef.xml File <bindings> Element

Element Syntax	Attributes	Attribute Description
<button>	ApplyValidation	Set to true by default. When true, controlBinding executes validators defined on the binding. You can set to false in the case of ADF Business Components, when running in local mode and when the same validators are already defined on the corresponding attribute.
	BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. This is used by earlier versions of JDeveloper.
	BoolVal	Identifies whether the value at the zero index in the static value list in this boolean list binding represents true or false.
	ControlClass	Used internally by ADF.
	CustomInputHandler	This is the class name for a oracle.jbo.uicli.binding.JUCtrlValueHandler implementation that is used to process the inputValue for a given value binding.
	DefClass	Used internally by ADF.
	id	Unique identifier. May be referenced by any ADF action binding.
	IterBinding	Refers to the iteratorBinding instance in this bindingContainer to which this binding is associated.
	ListIter	Refers to the iteratorBinding that is associated with the source list of this listBinding.
	ListOperMode	Determines whether this list binding is for navigation, contains a static list of values or is an LOV type list.
	NullValueFlag	Describes whether this list binding has a null value and, if so, whether it should be displayed at the beginning or the end of the list.
	NullValueId	Refers to the entry in the message bundle for this bindingContainer that contains the String to indicate the null value in a list display.
<ganttDataMap>		Maps the data binding XML for an ADF Faces gantt component.
<gaugeDataMap>		Maps the data binding XML for an ADF Faces gauge component.
<graph>	ApplyValidation	Set to true by default. When true, controlBinding executes validators defined on the binding. You can set to false in the case of ADF Business Components, when running in local mode and when the same validators are already defined on the corresponding attribute.
	BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. This is used by earlier versions of JDeveloper.
	BoolVal	Identifies whether the value at the zero index in the static value list in this boolean list binding represents true or false.
	ChildAccessorName	The name of the accessor to invoke to get the next level of nodes for a given hierarchical node type in a tree.

Table A–5 (Cont.) Attributes of the PageDef.xml File <bindings> Element

Element Syntax	Attributes	Attribute Description
	ControlClass	Used internally by ADF.
	CustomInputHandler	This is the class name for a oracle.jbo.uicli.binding.JUCtrlValueHandler implementation that is used to process the inputValue for a given value binding.
	DefClass	Used internally by ADF.
	GraphPropertiesFile Name	An XML file that specifies the type of graph to use, for example, pie chart or bar graph. This XML file can be used to customize the visual properties of the graph. It contains graph attributes such as title, subtitle, footnote, graph type, legend area, and plot area. The default filename is BIGraphDef.xml.
	GroupLabel	For master-detail forms, specifies the attribute that will be used to group data.
	id	Unique identifier. May be referenced by any ADF action binding.
	IterBinding	Refers to the iteratorBinding instance in this bindingContainer to which this binding is associated.
	NullValueId	Refers to the entry in the message bundle for this bindingContainer that contains the String to indicate the null value in a list display.
	SeriesLabel	Defines the attribute, based on which data will be clubbed.
	SeriesType	Determines whether graph is for Single View(SINGLE_SERIES), or for MASTER_DETAIL.
<graphDataMap>		Wraps the data binding XML for an ADF Faces graph component.
<list>	ApplyValidation	Set to true by default. When true, controlBinding executes validators defined on the binding. You can set to false in the case of ADF Business Components, when running in local mode and when the same validators are already defined on the corresponding attribute.
	BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. This is used by earlier versions of JDeveloper.
	ControlClass	Used internally by ADF.
	CustomInputHandler	This is the class name for a oracle.jbo.uicli.binding.JUCtrlValueHandler implementation that is used to process the inputValue for a given value binding.
	DefClass	Used internally by ADF.
	id	Unique identifier. May be referenced by any ADF action binding.
	IterBinding	Refers to the iteratorBinding instance in this bindingContainer to which this binding is associated.
	ListIter	Refers to the iteratorBinding that is associated with the source list of this listBinding.

Table A–5 (Cont.) Attributes of the PageDef.xml File <bindings> Element

Element Syntax	Attributes	Attribute Description
	ListOperMode	Determines whether this list binding is for navigation, contains a static list of values, or is an LOV type list.
	Mode	The value of this attribute determines if the list binding passes the actual value (Object) or the location of the value in an index (Index). For more information about setting this attribute, see Section 30.3.6, "What You May Need to Know About Values in a Selection List."
	MRUCount	Specifies the number of items to display in a choice list when you want to provide a shortcut for the end-user to display their most recent selections. For example, a form might display a choice list of supplier ID values to drive a purchase order form. In this case, you can allow users to select from a list of their most recently view suppliers, where the number of supplier choices is determined by the count you enter. The default for the choice list is to display all values for the attribute and is specified by the count 0 (zero)."
	MRUId	Specifies the String that will be the discriminator line for the MRU list.
	NullValueFlag	Describes whether this list binding has a null value and, if so, whether it should be displayed at the beginning of the list or the end.
	NullValueId	Refers to the entry in the message bundle for this bindingContainer that contains the String to indicate the null value in a list display.
	StaticList	Defines a static list of values that will be rendered in the bound list component.
<mapThemeDataMap>		Wraps the data binding XML for an ADF Data Visualization geographic map component.
<methodAction>	Action	Fully qualified package name. Identifies the class for which the data control is created. In the case of the EJB session facade, this is the session bean.
	BindingClass	This is for backward compatibility to indicate which class implements the runtime for this binding definition. This is used by earlier versions of JDeveloper.
	ClassName	This is the class to which the method being invoked belongs.
	DataControl	Name of the DataControl usage in the bindingContext (.cpx) which this iteratorBinding or actionBinding is associated with.
	DefClass	Used internally by ADF.
	id	Unique identifier. May be referenced by any ADF action binding.
	InstanceName	A dot-separated EL path to a Java object instance on which the associated method is to be invoked.
	IsLocalObjectReference	Set to true if the instanceName contains an EL path relative to this bindingContainer.
	IsViewObjectMethod	Set to true if the instanceName contains an instance path relative to the associated data control's application module.

Table A–5 (Cont.) Attributes of the PageDef.xml File <bindings> Element

Element Syntax	Attributes	Attribute Description
	`MethodName`	Indicates the name of the operation on the given instance or class that needs to be invoked for this `methodActionBinding`.
	`RequiresUpdateModel`	Whether this action requires that the model be updated before the action is to be invoked.
	`ReturnName`	The EL path of the result returned by the associated method.
`<pivotTableDataMap>`		Wraps the data binding XML for an ADF Faces pivot table component.
`<table>` and `<tree>`	`ApplyValidation`	Set to `true` by default. When `true`, `controlBinding` executes validators defined on the binding. You can set to `false` in the case of ADF Business Components, when running in local mode and when the same validators are already defined on the corresponding attribute.
	`BindingClass`	This is for backward compatibility to indicate which class implements the runtime for this binding definition.This is used by earlier versions of JDeveloper.
	`CollectionModel`	Accesses the `CollectionModel` object, the data model that is used by ADF table components. A table's value is bound to the `CollectionModel` attribute. The table wraps the result set from the iterator binding in a `CollectionModel` object. The `CollectionModel` attribute allows each item in the collection to be available within the table component using the `var` attribute.
	`ControlClass`	Used internally for testing purposes.
	`CustomInputHandler`	This is the class name for a `oracle.jbo.uicli.binding.JUCtrlValueHandler` implementation that is used to process the `inputValue` for a given value binding.
	`DefClass`	Used internally by ADF.
	`DiscrValue`	Indicates the discriminator value for a hierarchical type binding (type definition for a tree node). This value is used to determine whether a given row in a collection being rendered in a polymorphic tree binding should be rendered using the containing hierarchical type binding.
	`id`	Unique identifier. May be referenced by any ADF action binding.
	`IterBinding`	Refers to the `iteratorBinding` instance in this `bindingContainer` to which this binding is associated.
	`TreeModel`	The data model used by ADF Tree components. `TreeModel` extends `CollectionModel` to add support for container rows. Rows in the `TreeModel` may (recursively) contain other rows.

A.9 adfc-config.xml

The `adfc-config.xml` file is the source file for the ADF unbounded task flow that JDeveloper creates by default when you create an application using the Fusion Web Application (ADF) template. By default, JDeveloper stores the `adfc-config.xml` file in the following directory:

application_root\ViewController\Web Content\WEB-INF

If you create additional unbounded task flows, JDeveloper proposes the following file name for the source files of the additional unbounded task flows:

adfc-config*N*.xml

where *N* is a number that increments each time you create a new unbounded task flow. Alternatively, you choose the file name you want for an unbounded task flow's source file.

Each source file for an unbounded task flow contains the metadata for activities, control flow rules, and managed beans that you added to an unbounded task flow so that end users can interact with the Fusion web application to complete a task.

The XML schema definition file (XSD) that defines valid metadata entries for the adfc-config.xml file is adfc-config_1_0.xsd. The adfc-config_1_0.xsd file is stored in the adf-controller-schema.jar file in the following directory of your JDeveloper installation:

jdev_install\oracle_common\modules\oracle.adf.model_11.1.1

Use JDeveloper's XSD Visual Editor to view the adfc-config_1_0.xsd file. Using this editor, you can identify the valid metadata elements and attributes for the adfc-config.xml file.

Example A–5 shows the source file for an unbounded task flow where view activities, control flow rules, and so on have yet to be added.

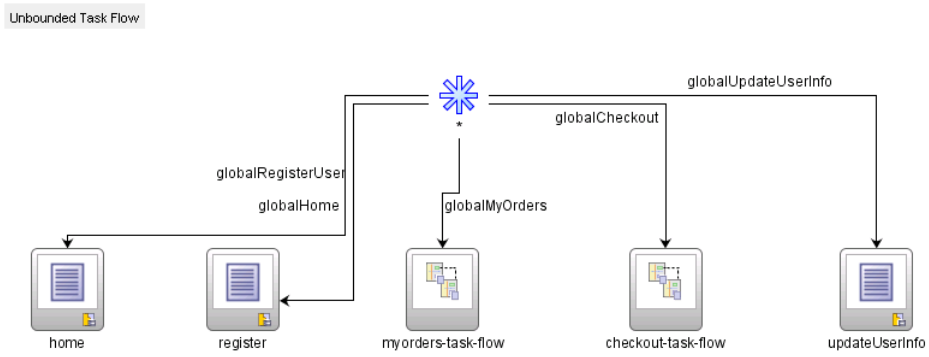
Example A–5 Sample adfc-config.xml file with no entries

```
<?xml version="1.0" encoding="windows-1252" ?>
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2">
</adfc-config>
```

Figure A–4 shows the Diagram view of the Fusion Order Demo application's adfc-config.xml file. Example A–6 shows the corresponding source view of the unbounded task flow that appears in Figure A–4. For brevity, many metadata entries have been removed from Example A–6. However, instances of metadata that identify view activities (<view id>), a task flow call activity (<task-flow-call>), control flow rules (<control-flow-rule>), and managed beans (<managed-bean>) remain.

For more information about unbounded task flows, Chapter 18, "Getting Started with ADF Task Flows."

Figure A–4 Diagram View of Fusion Order Demo Application's adfc-config.xml File



Example A–6 Source View of Fusion Order Demo Application's adfc-config.xml file

```
<?xml version="1.0" encoding="windows-1252" ?>
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller">
  <metadata-resource>/WEB-INF/adfc-config.xml</metadata-resource>
  <view id="home">
    <page>/home.jspx</page>
  </view>
  <task-flow-call id="checkout-task-flow">
    <task-flow-reference>
      <document>/WEB-INF/checkout-task-flow.xml</document>
      <id>checkout-task-flow</id>
    </task-flow-reference>
  </task-flow-call>
  <view id="updateUserInfo">
    <page>/account/updateUserInfo.jspx</page>
  </view>
  ...
  <task-flow-call id="myorders-task-flow">
    <task-flow-reference>
      <document>/WEB-INF/myorders-task-flow.xml</document>
      <id>myorders-task-flow</id>
    </task-flow-reference>
  </task-flow-call>
  <control-flow-rule>
    <from-activity-id>*</from-activity-id>
    <control-flow-case>
      <from-outcome>globalHome</from-outcome>
      <to-activity-id>home</to-activity-id>
    </control-flow-case>
   <!-- removed control flow cases for brevity -->
  </control-flow-rule>
  <managed-bean>
    <managed-bean-name>homePageBean</managed-bean-name>
    <managed-bean-class>oracle.fodemo.storefront.store.view.managed.HomeBean</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
  </managed-bean>
 <!-- removed managed beans for brevity -->
</adfc-config>
```

A.10 task-flow-definition.xml

The default file name that JDeveloper proposes for the source file of the first ADF bounded task flow that you create is `task-flow-definition.xml`. This file stores the metadata for a bounded task flow. It contains entries for each view activity, method call activity, task flow call, and so on that you add to the bounded task flow.

JDeveloper proposes the following file name for subsequent bounded task flows that you create:

`task-flow-definition`*N*`.xml`

where *N* is a number that increments each time you create a new bounded task flow. Alternatively, you choose the file name you want for a bounded task flow's source file. By default, JDeveloper stores the `task-flow-definition.xml` file in the following directory:

application_root`\ViewController\Web Content\WEB-INF`

The XML schema definition files (XSD) that define the valid metadata entries for the source file of a bounded task flow are stored in the `adf-controller-schema.jar` file in the following directory of your JDeveloper installation:

jdev_install`\oracle_common\modules\oracle.adf.model_11.1.1`

Use JDeveloper's XSD Visual Editor to view these XSD files and identify the valid metadata elements and attributes for the source file of a bounded task flow.

Example A–7 shows the content of a source file when you create a new bounded task flow using the Create Task Flow dialog. The `<task-flow-definition>` element identifies this file as a bounded task flow. JDeveloper generates the `<task-flow-definition>` element when you select the **Create as Bounded Task Flow** check box in the Create Task Flow dialog. The value of the `<task-flow-definition>` element's `id` attribute (`task-flow-definition`) corresponds to the value that you enter in the **Task Flow ID** field of the Create Task Flow dialog. The `<use-page-fragments/>` element indicates that all view activities in this bounded task flow must be associated with page fragments. JDeveloper generates this entry when you select the **Create with Page Fragments** check box in the Create Task Flow dialog.

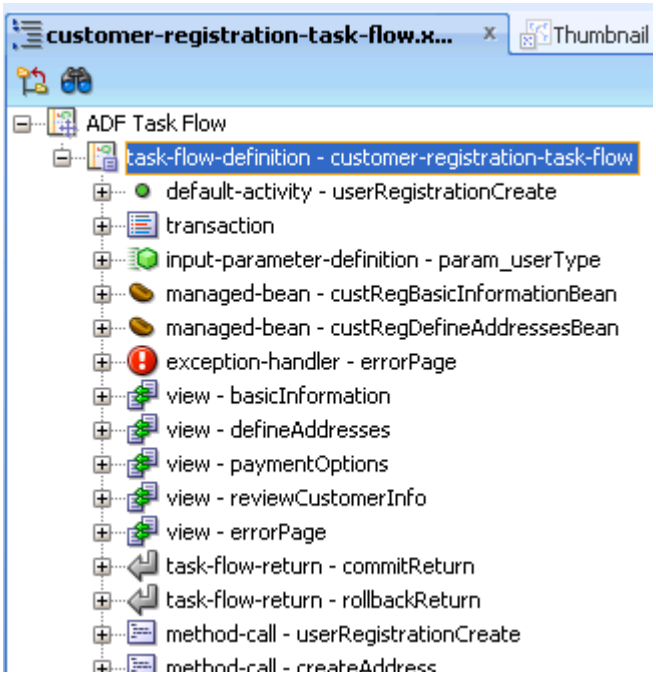
Example A–7 Sample task-flow-definition.xml with no entries

```
<?xml version="1.0" encoding="windows-1252" ?>
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2">
  <task-flow-definition id="task-flow-definition">
    <use-page-fragments/>
  </task-flow-definition>
</adfc-config>
```

Example A–8 shows an abbreviated version of the Fusion Order Demo application's `customer-registration-task-flow.xml` file. Many metadata entries have been removed for brevity. However, you can view entries for the activities shown in the Structure window in Figure A–5.

For more information about bounded task flows, see Chapter 18, "Getting Started with ADF Task Flows".

Figure A–5 Abbreviated Structure View of the customer-registration-task-flow.xml File



Example A–8 Abbreviated Source View of the customer-registration-task-flow.xml File

```
<?xml version="1.0" encoding="windows-1252" ?>
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.0">
  <task-flow-definition id="customer-registration-task-flow">
    <default-activity>userRegistrationCreate</default-activity>
    <transaction>
      <requires-transaction/>
    </transaction>
    <input-parameter-definition>
      <name>param_userType</name>
      <value>#{pageFlowScope.userType}</value>
      <class>java.lang.String</class>
      <required/>
    </input-parameter-definition>
    <managed-bean>
      <managed-bean-name>custRegBasicInformationBean</managed-bean-name>
      <managed-bean-class>....CustRegBasicInformationBean</managed-bean-class>
      <managed-bean-scope>view</managed-bean-scope>
      <managed-property>
        <property-name>allItemsIteratorName</property-name>
        <value>AvailableCategoriesShuttleListIterator</value>
      </managed-property>
    </managed-bean>
    <exception-handler>errorPage</exception-handler>
    <view id="reviewCustomerInfo">
      <display-name>Review</display-name>
      <page>/account/reviewCustomerInfo.jsff</page>
      <train-stop>
        <display-name>Review</display-name>
      </train-stop>
    </view>
    <task-flow-return id="rollbackReturn">
      <outcome>
        <name>cancelCreateCust</name>
```

```
        <rollback/>
      </outcome>
    </task-flow-return>
    <method-call id="createAddress">
      <method>#{bindings.CreateInsert.execute}</method>
      <outcome>
        <fixed-outcome>editAddress</fixed-outcome>
      </outcome>
    </method-call>
    <view id="paymentOptionDetails">
      <page>/account/paymentOptionDetails.jsff</page>
    </view>
    <parent-action id="Exit">
      <parent-outcome>globalHome</parent-outcome>
    </parent-action>
    <train/>
    <control-flow-rule>
      <from-activity-id>*</from-activity-id>
      <control-flow-case>
        <from-outcome>exitRegistration</from-outcome>
        <to-activity-id>Exit</to-activity-id>
      </control-flow-case>
      <control-flow-case>
        <from-outcome>cancelRegistration</from-outcome>
        <to-activity-id>rollbackReturn</to-activity-id>
      </control-flow-case>
    </control-flow-rule>
    <control-flow-rule>
      <from-activity-id>userRegistrationCreate</from-activity-id>
      <control-flow-case>
        <from-outcome>editBasicInfo</from-outcome>
        <to-activity-id>basicInformation</to-activity-id>
      </control-flow-case>
    </control-flow-rule>
    <use-page-fragments/>
  </task-flow-definition>
</adfc-config>
```

A.11 adf-config.xml

JDeveloper generates the `adf-config.xml` file when you create an application using the Fusion Web Application (ADF) template. It stores the file in the following directory:

application_root`\.adf\META-INF`

The `adf-config.xml` file specifies application-level settings that are usually determined at deployment and are often changed at runtime. You can use a deployment profile to specify settings that are used at application deploy time. You can change some of the settings at runtime using Oracle Enterprise Manager.

Examples of tasks that you can accomplish by changing settings in the `adf-config.xml` file include the following:

- Enable or disable the validation of ADF Controller metadata
- Replicate memory scope if you deploy your Fusion web application in a clustered environment

- Configure properties to manage the caching of resource bundles where your application uses EL expressions to retrieve strings at runtime from resource bundles

  The properties that you can configure are:

  - `initial-size`

    Specifies the initial number of resource bundles that your application can cache. The default value is `100`.

  - `max-size`

    Specifies the maximum number of resource bundles that your application can cache. The default value is `100`.

  - `load-factor`

    The default value is `0.75`.

  - `expire-time`

    The default value is `43200` seconds (12 hours).

    You specify these properties as attribute values of the `<resource-bundle-cache>` element in the `adf-config.xml` file. Example A–9 demonstrates how you might configure these values for your application in the `adf-config.xml` file.

  As an alternative to configuring the caching of resource bundles in the `adf-config.xml` file, you can specify the resource bundle caching properties as parameters for the Java Virtual Machine (JVM). If you specify the properties as parameters for the JVM, the changes apply to all applications managed by the JVM. For this reason, we recommend that you configure the resource bundle caching properties in the `adf-config.xml` file for you application. Use the following property names if you decide to specify the resource bundle caching properties as parameters for the JVM:

  - `resource-bundle-cache-initial-size`
  - `resource-bundle-cache-max-size`
  - `resource-bundle-cache-load-factor`
  - `resource-bundle-cache-expire-time`

At runtime, the Fusion web application loads the `adf-config.xml` file from the `META-INF` directory. If the Fusion web application finds more than one `adf-config.xml` file, it stops loading the file and logs a warning.

The following tasks modify or require you to modify the `adf-config.xml` file:

- Creating task flows

  For more information, see Chapter 18, "Getting Started with ADF Task Flows".

- Enabling implicit save points

  For more information, see Section 22.6.7, "How to Enable Implicit Save Points".

- Disable automatic partial page rendering as the default behavior for an application. This requires you to change the value for the `changeEventPolicy` attribute from the default value of `ppr`:

  ```
  <defaults changeEventPolicy="ppr"/>
  ```

For more information, see Section 25.2.1, "What You May Need to Know About Partial Page Rendering and Iterator Bindings."

- Persisting saved searches in MDS

  For more information, see Section 31.2.4, "How to Persist Saved Searches into MDS".

- Configuring ADF Business Components global settings

  For more information, see Section 42.1.1, "Limiting the View Object Max Fetch Size to Fetch the First n Rows" and Section 3.3.1, "Choosing a Connection, SQL Platform, and Data Type Map."

- Enabling Oracle ADF Security

  For more information, see Section 35.3.2, "What Happens When You Enable ADF Security".

- Enabling seeded customizations

  For more information, Section 39.2.4, "How to Enable Seeded Customizations for View Projects" and Section 39.2.5, "How to Enable Seeded Customizations in Existing Pages".

- Configuring change persistence

  For more information, see Section 40.2.1, "How to Enable User Customizations".

- Enabling user customizations

  For more information, see Section 40.2.2, "What Happens When You Enable User Customizations".

Example A–9 shows example entries for an application's `adf-config.xml` file.

Example A–9 Sample adf-config.xml File

```
<?xml version="1.0" encoding="windows-1252" ?>
<adf-config xmlns="http://xmlns.oracle.com/adf/config"
            xmlns:sec="http://xmlns.oracle.com/adf/security/config">
  <sec:adf-security-child xmlns="http://xmlns.oracle.com/adf/security/config">
    <JaasSecurityContext
initialContextFactoryClass="oracle.adf.share.security.JAASInitialContextFactory"

jaasProviderClass="oracle.adf.share.security.providers.jps.JpsSecurityContext"
                        authorizationEnforce="true"
                        authenticationRequire="true"/>
    <CredentialStoreContext
credentialStoreClass="oracle.adf.share.security.providers.jps.CSFCredentialStore"
                           credentialStoreLocation="../../src/META-INF/jps-config.xml"/>
  </sec:adf-security-child>
  <adf-controller-config xmlns="http://xmlns.oracle.com/adf/controller/config">
    <savepoint-datasource>java:comp/env/jdbc/FODDS</savepoint-datasource>
    <enable-implicit-savepoints>true</enable-implicit-savepoints>
  </adf-controller-config>
  <adf-faces-config xmlns="http://xmlns.oracle.com/adf/faces/config">
    <persistent-change-manager>

<persistent-change-manager-class>oracle.adf.view.rich.change.MDSDocumentChangeManager</persistent-c
hange-manager-class>
    </persistent-change-manager>
    <taglib-config>
      <taglib uri="http://xmlns.oracle.com/adf/faces/rich">
```

```
        <tag name="calendar">
          <attribute name="activeDay">
            <persist-changes>true</persist-changes>
          </attribute>
        </tag>
        <!-- Additional tags omitted to make this example concise -->
        <tag name="table">
          <attribute name="filterVisible">
            <persist-changes>true</persist-changes>
          </attribute>
        </tag>
      </taglib>
    </taglib-config>
  </adf-faces-config>
  <adf-mds-config xmlns="http://xmlns.oracle.com/adf/mds/config">
    <mds-config xmlns="http://xmlns.oracle.com/mds/config" version="11.1.1.000">
      <cust-config>
        <match path="/">
          <customization-class name="oracle.adf.share.config.UserCC"/>
        </match>
      </cust-config>
    </mds-config>
  </adf-mds-config>
  <adf-adfm-config xmlns="http://xmlns.oracle.com/adfm/config">
    <defaults rowLimit="100"/>
    <startup>
      <amconfig-overrides>
        <config:Database jbo.SQLBuilder="Oracle" jbo.locking.mode="optimistic"/>
      </amconfig-overrides>
    </startup>
  </adf-adfm-config>
<!-- Properties to manage the caching of a resource bundle in your application -->
<adf-resourcebundle-config xmlns="http://xmlns.oracle.com/adf/resourcebundle/config">
    <applicationBundleName>
       path-to-resource-bundle/bundle-name
    </applicationBundleName>
    <resource-bundle-cache initial-size="20" max-size="100" expire-time="30000" load-factor=".75"/>
    <bundleList>
      <bundleId override="true">
         package.BundleID
      </bundleId>
    </bundleList>
  </adf-resourcebundle-config>
</adf-config>
```

A.12 adf-settings.xml

The `adf-settings.xml` file holds project-level and library-level settings such as ADF Faces help providers and ADF Controller phase listeners.

The configuration settings for `adf-settings.xml` are fixed and cannot be changed during or after application deployment. There can be multiple `adf-settings.xml` files in an application. The users of `adf-settings.xml` files are responsible for merging the contents of their configuration.

JDeveloper creates an `adf-settings.xml` file when you:

- Create an application using the Fusion Web Application (ADF) template
- Add ADF Page Flow (ADF Controller) to the technology scope of an existing project

By default, JDeveloper stores the `adf-settings.xml` file in the following directory:

`application_root\ViewController\src\META-INF`

The following tasks modify or require you to modify the `adf-settings.xml` file:

- Registering a phase listener

  For more information, see Section 25.4.2, "How to Register a Listener Globally".

- Creating help for ADF Faces components

  For more information, see the "Displaying Help for Components" section in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

Example A–10 shows a sample `adf-setting.xml` file with settings configured for a phase listener and a help provider.

Example A–10 Sample adf-settings.xml File

```
<?xml version="1.0" encoding="windows-1252" ?>
<adf-settings xmlns="http://xmlns.oracle.com/adf/config">
  <adfc-controller-config xmlns="http://xmlns.oracle.com/adf/controller/config">
    <lifecycle>
      <phase-listener>
        <listener-id>FODPhaseListener</listener-id>
        <class>oracle.fodemo.storefront.listeners.FODPhaseListener</class>
      </phase-listener>
    </lifecycle>
  </adfc-controller-config>
  <adf-faces-config>
    <help-provider prefix="MYAPP">
      <help-provider-class>oracle.fodemo.storefront.MyHelpProvider</help-provider-class>
      <property>
        <property-name>myCustomProperty</property-name>
        <value>someValue</value>
      </property>
    </help-provider>
  </adf-faces-config>
</adf-settings>
```

A.13 web.xml

Oracle ADF has specific configuration settings for the standard `web.xml` deployment descriptor file.

When you create a project in JDeveloper that uses JSF technology, a starter `web.xml` file with default settings is created for you in the `/WEB-INF` folder. To edit the file, double-click **web.xml** in the Application Navigator to open it in the XML editor.

The following must be configured in `web.xml` for all applications that use JSF and ADF Faces:

- ADF Library resource servlet and mapping: Serves up web application resources (images, style sheets, JavaScript libraries) from ADF Library JAR files on the application class path.

  By default, static web application resources (including resources in ADF Libraries), such as images, have a staleness period setting of 364 days. This staleness period setting requests that the client not make a request to the server to validate the static web application resources until one of the following events occur:

- Staleness period expires
- Browser cache no longer contains the static resource
- User executes a page refresh

If the client does make a request to the server and the resources have not changed, the server returns a HTTP 304 response (Not Modified) with no body.

You can override the staleness period setting of 364 days for all static web application resources in ADF Libraries by adding initialization parameters to the `web.xml` file. Example A–11 demonstrates how to set these initialization parameters using a number of examples.

Example A–11 Staleness Settings for Static Resources in web.xml

```
<!-- Expires all static resources in 30 days -->
 <init-param>
   <param-name>expires</param-name>
   <param-value>60*60*24*30</param-value>
 </init-param>

<!-- Expires static resources with the .js file extension in 1 day -->
 <init-param>
   <param-name>expires.js</param-name>
   <param-value>60*60*24</param-value>
 </init-param>

<!-- Turns off staleness setting. Use for testing -->
 <init-param>
   <param-name>expires</param-name>
   <param-value>OFF</param-value>
 </init-param>
```

Example A–12 shows the default file extensions. You can add additional file extensions by configuring the `extensions` parameter value to include the file extensions that you want. Note that you must prepend and append "`.`" to the file extension. Example A–12 also shows how you specify visibility for cache control.

Example A–12 Specifying Additional File Extensions and Cache Control

```
<!-- Specify file extensions for a number of different file formats -->
 <init-param>
   <param-name>extensions</param-name>
   <param-value>.png.jpg.jpeg.gif.js.css.</param-value>
 </init-param>

<!-- Specify visibility for cache control, for example -->
 <init-param>
   <param-name>visibility</param-name>
   <param-value>Public</param-value>
 </init-param>
```

- JSF servlet and mapping: The servlet `javax.faces.webapp.FacesServlet` that manages the request-processing lifecycle for web applications utilizing JSF to construct the user interface.
- ADF Faces filter and mapping: A servlet filter to ensure that ADF Faces is properly initialized by establishing a `AdfFacesContext` object. This filter also processes file uploads.

The JSF servlet and mapping configuration settings are automatically added to the starter `web.xml` file when you first create a JSF project. When you insert an ADF Faces component into a JSF page for the first time, JDeveloper automatically inserts the configuration settings for ADF Faces filter and mapping, and resource servlet and mapping.

For more information, see the "ADF Faces Configuration" appendix in the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework*.

A.14 logging.xml

ADF Logger is a diagnostic tool that you can use in JDeveloper to capture runtime traces messages when you debug an application. You configure the use of this tool by editing the `logging.xml` file.

For more information about the `logging.xml` file and using the ADF Logger, see Section 36.6, "Using the ADF Logger".

B

Oracle ADF Binding Properties

This appendix provides a reference for the properties of the ADF bindings.

Table B–1 shows the properties that you can use in EL expressions to access values of the ADF binding objects at runtime. The properties appear in alphabetical order.

Table B–1 EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
`actionEnabled`	Use `operationEnabled` instead.	n/a	yes	n/a	n/a	n/a	n/a	n/a
`allRowsInRange`	Returns an array of the current set of rows from the associated collection. Calls `getAllRowsInRange()` on the `RowSetIterator` object.	yes	n/a	n/a	n/a	n/a	n/a	n/a
`attributeDef`	Returns the attribute definition for the first attribute with which the binding is associated.	n/a	n/a	yes	yes	yes	n/a	n/a
`attributeDefs`	Returns the attribute definitions for all the attributes to which the binding is associated.	n/a	n/a	yes	yes	yes	n/a	n/a
`attributeValue`	Returns an unformatted and typed (appropriate Java type) value in the current row, for the attribute to which the control binding is bound.	n/a	n/a	yes	yes	yes	n/a	n/a
`attributeValues`	Returns the value of all the attributes to which the binding is associated in an ordered array. Returns an array of unformatted and typed (appropriate Java type) values in the current row for all the attributes to which the control binding is bound.	n/a	n/a	yes	yes	yes	n/a	n/a
`bindings`	Returns a new binding for each cell or attribute exposed under the rows of a tree node binding.	no	no	no	no	no	no	yes
`children`	Returns the child nodes of a tree node binding.	n/a	n/a	n/a	n/a	n/a	n/a	yes

Table B–1 (Cont.) EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
`currentRow`	Returns the current row on an action binding bound to an iterator (for example, built-in navigation actions).	n/a	yes	n/a	n/a	n/a	n/a	n/a
`dataControl`	Returns the iterator's associated data provider.	yes	n/a	n/a	n/a	n/a	n/a	n/a
`displayData`	Returns a list of map elements. Each map entry contains the following elements: ▪ `selected`: A boolean `true` if the current entry should be selected. ▪ `index`: The index value of the current entry. ▪ `prompt`: A string value that may be used to render the entry in the UI. ▪ `displayValues`: An ordered list of display attribute values for all display attributes in the list binding.	n/a	n/a	n/a	n/a	yes	n/a	n/a
`displayHint`	Returns the display hint for the first attribute to which the binding is associated. The hint identifies whether the attribute should be displayed or not. For more information, see `oracle.jbo.AttributeHints.displayHint`.	n/a	n/a	n/a	n/a	yes	n/a	n/a

Table B–1 (Cont.) EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
`displayHints`	Returns a list of name-value pairs for UI hints for all display attributes to which the binding is associated. The map contains the following elements: ■ `label`: The label to display for the current attribute. ■ `tooltip`: The tooltip to display for the current attribute. ■ `displayHint`: The display hint for the current attribute. ■ `displayHeight`: The height in lines for the current attribute. ■ `displayWidth`: The width in characters for the current attribute. ■ `controlType`: The control type hint for the current attribute. ■ `format`: The format to be used for the current attribute.	n/a	n/a	n/a	yes	yes	n/a	n/a
`enabled`	Use the `operationEnabled` property.	n/a	n/a	n/a	n/a	n/a	n/a	n/a
`enabledString`	Returns `disabled` if the action binding is not ready to be invoked. Otherwise, returns an empty string (`""`).	n/a	yes	n/a	n/a	n/a	n/a	n/a
`error`	Returns any exception that was cached while updating the associated attribute value for a value binding or when invoking an operation bound by an operation binding.	yes	yes	yes	yes	yes	yes	yes
`estimatedRowCount`	Returns the maximum row count of the rows in the collection with which this iterator binding is associated	yes	n/a	n/a	n/a	n/a	yes	yes
`findMode`	Returns `true` if the iterator is currently operating in find mode. Otherwise, returns `false`.	yes	n/a	n/a	n/a	n/a	n/a	n/a
`fullName`	Returns the fully qualified name of the binding object in the Oracle ADF binding context.	yes	yes	yes	yes	yes	yes	yes

Table B–1 (Cont.) EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
`hints`	Returns the value of the UI hint indicated for the binding. See `displayHints` for the list of UI hint keywords.	yes	yes	yes	yes	yes	yes	yes
`inputValue`	Returns the value of the first attribute to which the binding is associated. If the binding was used to set the value on the attribute and the set operation failed, this method returns the invalid value that was being set.	n/a	n/a	yes	yes	yes	yes	yes
`iteratorBinding`	Returns the iterator binding that provides access to the data collection.	n/a	yes	yes	yes	yes	yes	yes
`label`	Returns the label (if supplied by control hints) for the first attribute of the binding.	n/a	n/a	yes	yes	yes	n/a	n/a
`labels`	Returns a map of labels (if supplied by control hints) keyed by attribute name for all attributes to which the binding is associated.	n/a	n/a	yes	yes	yes	yes	n/a
`labelSet`	Returns an ordered set of labels for all the attributes to which the binding is associated.	n/a	n/a	yes	yes	yes	yes	n/a
`mandatory`	Returns whether the first attribute to which the binding is associated is required.	n/a	n/a	yes	yes	yes	n/a	n/a
`name`	Returns the name of the binding object in the context of the binding container to which it is registered. Note this property is not visible in the EL expression builder dialog.	yes	yes	yes	yes	yes	yes	yes
`operationEnabled`	Returns `true` or `false`, depending on the state of the action binding. For example, the action binding may be enabled (`true`) or disabled (`false`) based on the currency (as determined, for example, when the user clicks the First, Next, Previous, Last navigation buttons).	n/a	yes	n/a	n/a	n/a	n/a	n/a

Table B–1 (Cont.) EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
`rangeSet`	Returns a list of map elements over the range of rows from the associated iterator binding. The elements in this list are wrapper objects over the indexed row in the range that restricts access to the attributes to which the binding is bound. The properties returned on the reference object are: ■ `index`: The range index of the row this reference is pointing to. ■ `key`: The key of the row this reference is pointing to. ■ `keyStr`: The string format of the key of the row this reference is pointing to. ■ `currencyString`: The current indexed row as a string. Returns "*" if the current entry belongs to the current row; otherwise, returns " ". This property is useful in JSP applications to display the current row. ■ `attributeValues`: The array of applicable attribute values from the row. You may also access an attribute value by name on a range set like `rangeSet.dname` if `dname` is a bound attribute in the range binding.	n/a	n/a	n/a	n/a	n/a	yes	yes
`rangeSize`	Returns the range size of the ADF iterator binding's row set. This allows you to determine the number of data objects to bind from the data source.	yes	n/a	n/a	n/a	n/a	yes	yes
`rangeStart`	Returns the absolute index in a collection of the first row in range. See the javadoc for `oracle.jbo.RowSetIterator.getRangeStart()`.	yes	n/a	n/a	n/a	n/a	yes	yes
`result`	Returns the result of a method that is bound and invoked by a method action binding.	n/a	yes	n/a	n/a	n/a	n/a	n/a
`rootNodeBinding`	Returns the root node of a tree binding.	n/a	n/a	n/a	n/a	n/a	n/a	yes

Table B–1 (Cont.) EL Properties of Oracle ADF Bindings

Runtime Property	Description	Iterator	Action	Attribute	Button	List	Table	Tree
`selectedValue`	Returns the value corresponding to the current selected index in the list or button binding.	n/a	n/a	n/a	yes	yes	n/a	n/a
`tooltip`	Returns the tooltip hint for the first attribute to which the binding is associated.	n/a	n/a	yes	yes	yes	n/a	n/a
`updateable`	Returns `true` if the first attribute to which the binding is associated is updateable. Otherwise, returns `false`.	n/a	n/a	yes	yes	yes	n/a	n/a

C

ADF Security Permission Grants

This appendix lists the security-aware components of Oracle Application Development Framework (Oracle ADF) and the actions that their `Permission` implementation classes define.

Table C–1 shows the ADF components and their permission grants that you can define to create ADF security policies. You add grants to the policy store using the overview editor for ADF security policies. A permission grant specifies the fully qualified permission class name, the fully qualified resource name, the action that can be performed against the resource, and the application role target of the grant. When you enable ADF security to enforce permission checking, the operations supported by ADF components will be inaccessible to users who do not possess sufficient access rights as defined by grants to their application role.

For complete details about defining ADF security policies in Fusion web applications, see Chapter 35, "Enabling ADF Security in a Fusion Web Application."

Table C–1 ADF Security Permission Grants

ADF Component	Grantable Action	Corresponding Implementation
ADF bounded task flow	`View`	The view action controls who can read and execute a bounded task flow. Pages that the user accesses within the process of executing a bounded task flow will not be individually security checked and will run under the permission of the task flow.
	`Customize`	Reserved for future use. This action is not checked at runtime.
	`Grant`	Reserved for future use. This action is not checked at runtime.
	`Personalize`	Reserved for future use. This action is not checked at runtime.
ADF page definition	`View`	The view action controls who can view the page. Page-level security is checked for pages that have an associated page definition binding file *only* if the page is accessed in the process of an unbounded task flow. There is a one-to-one relationship between the page definition file and the web page it secures.
ADF Business Components entity objects	`read`	The read action controls who can view a row of the bound collection.
	`update`	The update action controls who can update any attribute of the bound collection.
	`removeCurrentRow /delete`	The delete action controls who can delete a row from the bound collection.
ADF Business Components attributes of entity objects	`update`	The update action controls who can update a specific attribute of the bound collection.

D

Most Commonly Used ADF Business Components Methods

This appendix highlights the most commonly used methods in the interfaces and classes of ADF Business Components.

This appendix contains the following sections:

- Section D.1, "Methods for Creating Your Own Layer of Framework Base Classes"
- Section D.2, "Methods Used in the Client Tier"
- Section D.3, "Methods Used in the Business Service Tier"

D.1 Methods for Creating Your Own Layer of Framework Base Classes

Before you begin to develop application-specific business components, you can create a layer of classes that extend all of the ADF Business Components framework base implementation classes described in this appendix. An example of a customized framework base class for application module components might look like this:

```
package com.yourcompany.adfextensions;
import oracle.jbo.server.ApplicationModuleImpl;
public class CustomApplicationModuleImpl extends ApplicationModuleImpl {
  /*
   * We might not yet have any custom code to put here yet, but
   * the first time we need to add a generic feature that all of
   * our company's application modules need, we will be very happy
   * that we thought ahead to leave ourselves a convenient place
   * in our class hierarchy to add it so that all of the application
   * modules we have created will instantly benefit by that new feature,
   * behavior change, or even perhaps, bug workaround.
   */
}
```

A common set of customized framework base classes in a package name of your own choosing like `com.yourcompany.adfextensions`, each importing the `oracle.jbo.server.*` package, would consist of the following classes:

- `public class CustomEntityImpl extends EntityImpl`
- `public class CustomEntityDefImpl extends EntityDefImpl`
- `public class CustomViewObjectImpl extends ViewObjectImpl`
- `public class CustomViewRowImpl extends ViewRowImpl`
- `public class CustomApplicationModuleImpl extends ApplicationModuleImpl`

- `public class CustomDBTransactionImpl extends DBTransactionImpl2`
- `public class CustomDatabaseTransactionFactory extends DatabaseTransactionFactory`

For completeness, you may also want to create customized framework classes for the following classes as well:

- `public class CustomViewDefImpl extends ViewDefImpl`
- `public class CustomEntityCache extends EntityCache`
- `public class CustomApplicationModuleDefImpl extends ApplicationModuleDefImpl`

Overriding anything in these classes would be a fairly rare requirement.

D.2 Methods Used in the Client Tier

All of the interfaces described in this section are designed for use by client-layer code and are part of the `oracle.jbo.*` package.

This section provides a summary of the most frequently called, written, and overridden methods for the key ADF Business Components interfaces.

Note: The corresponding implementation classes for these `oracle.jbo.*` interfaces are intentionally designed to *not* be directly accessed by client code. Section D.3, "Methods Used in the Business Service Tier" shows that the implementation classes reside in the `oracle.jbo.server.*` package and generally have the suffix `Impl` in their name to help remind you not to use them in your client-layer code.

D.2.1 ApplicationModule Interface

An application module is a business service component that acts as a transactional container for other ADF components and coordinates *with* them to implement a number of Java EE design patterns important to business application developers. These design pattern implementations enable your client code to work easily with updatable collections of value objects, based on fast-lane reader SQL queries that retrieve only the data needed by the client, in the way the client wants to view it. Changes made to these value objects are automatically coordinated with your persistent business domain objects in the business service tier to enforce business rules consistently and save changes back to the database. Table D–1 describes the operations that you can perform on an application module using the `ApplicationModule` interface.

Note: For the complete list of design patterns that ADF Business Components implements, see Appendix E, "ADF Business Components Java EE Design Pattern Catalog."

Table D–1 ApplicationModule Interface

If you want to...	Call this ApplicationModule interface method
Access an existing view object instance using the assigned instance name (for example, `MyVOInstanceName`)	`findViewObject()`
Create a new view object instance from an existing definition	`createViewObject()`
Create a new view object instance from a SQL Statement	`createViewObjectFromQueryStmt()` **Notes:** This incurs runtime overhead to describe the "shape" of the dynamic query's `SELECT` list. Use this method only when you cannot know the `SELECT` list for the query at design time. Furthermore, if you are creating the dynamic query based on some kind of custom runtime repository, you can follow the steps to create (both read-only and updatable) dynamic view objects without the runtime-describe overhead, as described in Section 42.9, "Creating a View Object with Multiple Updatable Entities." If only the `WHERE` needs to be dynamic, create the view object at design time, then set the `WHERE` clause dynamically as needed using `ViewObject` APIs.
Access a nested application module instance by name	`findApplicationModule()`
Create a new nested application module instance from an existing definition	`createApplicationModule()`
Find a view object instance in a nested application module using a dot-notated name (for example, `MyNestedAMInstanceName.OneOfItsVONames`)	`findViewObject()` **Notes:** You can use this method to find an instance of a view object belonging to a nested application module in a single method call. This way you do not need to first call `findApplicationModule()` to find the nested application module, before calling `findViewObject()` on that nested application module.
Access the current transaction object	`getTransaction()`

In addition to generic application module access, JDeveloper can generate a custom *`YourApplicationModuleName`* interface containing service-level custom methods that you've chosen to expose to the client. You use the **Client Interface** page of the Edit Application Module dialog to select the methods that you want to appear in your client interface.

D.2.2 Transaction Interface

The `Transaction` interface exposes methods allowing the client to manage pending changes in the current transaction. Table D–2 describes the operations that you can perform on the transaction using the `Transaction` interface.

Table D–2 Transaction Interface

If you want to...	Call this Transaction interface method
Commit pending changes	`commit()`
Roll back pending changes	`rollback()`
Execute a one-time database command or block of PL/SQL	`executeCommand()` **Notes:** Do not use this command with methods that require retrieving `OUT` parameters and that will be executed more than once, or that could benefit from using bind variables. Instead, expose a custom method on your application module.
Validate all pending invalid changes in the transaction	`validate()`
Change the default locking mode	`setLockingMode()` **Notes:** You can set the locking mode in your configuration by setting the property `jbo.locking.mode` to one of the four supported values: `none`, `optimistic`, `pessimistic`, `optupdate`. If you don't explicitly set the locking mode, it will default to `optimistic`. For Fusion web applications, use `optimistic` or `optupdate` modes.
Decide whether to use bundled exception reporting mode or not	`setBundledExceptionMode()` **Notes:** ADF Controller layer support sets this parameter to `true` automatically for Fusion web applications.
Decide whether entity caches will be cleared upon a successful commit of the transaction	`setClearCacheOnCommit()` **Notes:** Default is `false`.
Decide whether entity caches will be cleared upon a rollback of the transaction	`setClearCacheOnRollback()` **Notes:** Default is `true`.
Clear the entity cache for a specific entity object	`clearEntityCache()`

D.2.3 ViewObject Interface

A view object is a component that encapsulates a database query and simplifies working with the row set of results it produces. You use view objects to project, filter, join, or sort business data using SQL from one or more tables to cast the data into exactly the format that the user should see on the page or panel. You can create "master-detail" hierarchies of any depth or complexity by connecting view objects together using view links. View objects can produce read-only query results, or when associated with one or more entity objects at design time, can be fully updatable. Updatable view objects can support insertion, modification, and deletion of rows in the result collection, with automatic delegation to the correct business domain objects.

Every view object contains a "default row set" for simplifying the 90 percent of use cases where you work with a single row set of results for the view object's query. A view object implements all the methods on the `RowSet` interface by delegating them to this default `RowSet`. That means you can invoke any `RowSet` methods on any view object as well.

Every view object implements the `StructureDef` interface to provide information about the number and types of attributes in a row of its row sets. So you can call `StructureDef` methods directly on any view object.

Table D–3 describes the operations that you can perform on a view object using the `ViewObject` interface

Table D–3 ViewObject Interface

If you want to...	Call this ViewObject interface method
Set an additional runtime `WHERE` clause on the row set	`setWhereClause()` **Notes:** This `WHERE` clause augments any `WHERE` clause specified at design time in the base view object. It does not replace it.
Set a dynamic `ORDER BY` clause	`setOrderByClause()`
Create a Query-by-Example criteria collection	`createViewCriteria()` **Notes:** You then create one or more `ViewCriteriaRow` objects using the `createViewCriteriaRow()` method on the `ViewCriteria` object you created. Then you `add()` these view criteria rows to the view criteria collection and apply the criteria using the `applyViewCriteria()` method.
Apply a Query-by-Example criteria collection	`applyViewCriteria()`
Set a query optimizer hint	`setQueryOptimizerHint()`
Access the attribute definitions for the key attributes in the view object	`getKeyAttributeDefs()`
Add a dynamic attribute to rows in this view object's row sets	`addDynamicAttribute()`
Clear all row sets produced by a view object	`clearCache()`
Remove a view object instance and its resources	`remove()`
Set an upper limit on the number of rows that the view object will attempt to fetch from the database	`setMaxFetchSize()` **Notes:** Default is -1, which means to impose no limit on how many rows would be retrieved from the database if you iterated through them all. By default, as you iterate through them, they are fetched lazily.

In addition to generic `ViewObject` access, JDeveloper can generate you a custom *`YourViewObjectName`* interface containing view object-level custom methods that you've chosen to expose to the client. You use the **Client Interface** page of the Edit View Object dialog to select the methods that you want to appear in your client interface.

D.2.4 RowSet Interface

A row set is an object that contains a set of rows, typically produced by executing a view object's query.

Every `RowSet` aggregates a "default row set iterator" for simplifying the 90 percent of use cases where you need only a single iterator over the row set. A `RowSet` object implements all the methods on the `RowSetIterator` interface by delegating them to this default `RowSetIterator`. This means you can invoke any `RowSetIterator` method on any `RowSet` object (or view object, since it implements `RowSet`, as well for its default `RowSet`).

Table D–4 describes the operations that you can perform on a row set using the `RowSet` interface.

Table D–4 RowSet Interface

If you want to...	Call this RowSet interface method
Set a `WHERE` clause bind variable value	`setWhereClauseParam()` **Notes:** Bind variable ordinal positions are zero-based.
Avoid view object row caching if data is being read only once	`setForwardOnly()`
Force a row set's query to be (re)executed (in the case of exclusive view objects) or potentially executed (in the case of shared view object instances)	`executeQuery()` **Notes:** The behavior of this method differs depending on whether the view object belongs to a shared application module or not. When reexecuting the query for an exclusive view object (not an instance of a shared module), a new query collection is created. Before executing the query for a shared view object instance, a check is performed to determine whether the results already exist. Already cached results will be reused for the shared view object instance instead of reexecuting the query. If you want to ensure that the results for a shared view object instance are refreshed, you can invoke the `forceExecuteQueryOfSharedVO()` method. However, if at the time of invoking force execute a user is iterating over the collection of a shared view object instance, then the behavior is undefined and exceptions may result.
Estimate the number of rows in a view object's query result	`getEstimatedRowCount()`
Produce an XML document for rows in a view object row set	`writeXML()`
Process all rows from an incoming XML document	`readXML()`
Set whether a row set will automatically see new rows based on the same entity object created through other row sets	`setAssociationConsistent()`

Table D–4 (Cont.) RowSet Interface

If you want to...	Call this RowSet interface method
Create a secondary iterator to use for programmatic iteration	`createRowSetIterator()` **Notes:** If you plan to find and use the secondary iterator by name later, then pass in a string name as the argument; otherwise, pass `null` for the name and make sure to close the iterator when done iterating by calling its `closeRowSetIterator()` method.

D.2.5 RowSetIterator Interface

A row set iterator is an iterator over the rows in a row set. By default it allows you to iterate both forward and backward through the rows. Table D–5 describes the operations that you can perform on a row set using the `RowSetIterator` interface.

Table D–5 RowSetIterator Interface

If you want to...	Call this RowSetIterator interface method
Get the first row of the iterator's row set	`first()`
Test whether there are more rows to iterate	`hasNext()`
Get the next row of an iterator's row set	`next()`
Find a row in this iterator's row set with a given key value	`findByKey()` **Notes:** It's important that the `Key` object that you pass to `findByKey` be created using the *exact* same data types as the attributes that comprise the key of the rows in the view object you're working with.
Create a new row to populate for insertion	`createRow()` **Notes:** The new row will already have default values set for attributes which either have a static default value supplied at the entity object or view object level, or if the values have been populated in an overridden `create()` method of the underlying entity object(s).
Create a view row with an initial set of foreign key and/or discriminator attribute values	`createAndInitRow()` **Notes:** You use this method when working with view objects that can return one of a "family" of entity object subtypes. By passing in the correct discriminator attribute value in the call to create the row, the framework can create you the correct matching entity object subtype underneath.

Table D–5 (Cont.) RowSetIterator Interface

If you want to...	Call this RowSetIterator interface method
Insert a new row into the iterator's row set	`insertRow()` **Notes:** It's a good habit to always immediately insert a newly created row into the rowset. That way you will avoid a common gotcha of creating the row but forgetting to insert it into the rowset.
Get the last row of the iterator's row set	`last()`
Get the previous row of the iterator's row set	`previous()`
Reset the current row pointer to the slot before the first row	`reset()`
Close an iterator when done iterating	`closeRowSetIterator()`
Set a given row to be the current row	`setCurrentRow()`
Remove the current row	`removeCurrentRow()`
Remove the current row to later insert it at a different location in the same iterator	`removeCurrentRowAndRetain()`
Remove the current row from the current collection but do not remove it from the transaction.	`removeCurrentRowFromCollection()`
Set/change the number of rows in the range (a "page" of rows the user can see)	`setRangeSize()`
Scroll to view the *n*th page of rows (1-based)	`scrollToRangePage()`
Scroll to view the range of rows starting with row number *n*	`scrollRangeTo()`
Set row number *n* in the range to be the current row	`setCurrentRowAtRangeIndex()`
Get all rows in the range as a row array	`getAllRowsInRange()`

D.2.6 Row Interface

A row is generic value object. It contains attributes appropriate in name and Java type for the view object that it is related to. Table D–6 describes the operations that you can perform on a view object row using the `Row` interface.

Table D–6 Row Interface

If you want to...	Call this Row interface method
Get the value of an attribute by name	`getAttribute()`
Set the value of an attribute by name	`setAttribute()`
Produce an XML document for a single row	`writeXML()`
Eagerly validate a row	`validate()`
Read row attribute values from XML	`readXML()`
Remove the row	`remove()`
Flag a newly created row as temporary (until updated again)	`setNewRowState(Row.STATUS_INITIALIZED)`

Table D–6 (Cont.) Row Interface

If you want to...	Call this Row interface method
Retrieve the attribute structure definition information for a row	`getStructureDef()`
Get the `Key` object for a row	`getKey()`

In addition to generic `Row` access, JDeveloper can generate a custom *`YourViewObjectName`*`Row` interface containing your type-safe attribute getter and setter methods, as well as any desired row-level custom methods that you've chosen to expose to the client. You use the **Client Row Interface** page of the Edit View Object dialog to select the methods that you want to appear in your client interface.

D.2.7 StructureDef Interface

The `StructureDef` interface provides access to runtime metadata about the structure of a `Row` object.

In addition, for convenience every view object implements the `StructureDef` interface as well, providing access to metadata about the attributes in the resulting view rows that its query will produce.

Table D–7 describes the operations that you can perform on a view object row using the `StructureDef` interface.

Table D–7 StructureDef Interface

If you want to...	Call this StructureDef interface method
Access attribute definitions for all attributes in the view object row	`getAttributeDefs()`
Find an attribute definition by name	`findAttributeDef()`
Get attribute definition by index	`getAttributeDef()`
Get number of attributes in a row	`getAttributeCount()`

D.2.8 AttributeDef Interface

The `AttributeDef` interface provides attribute definition information for any attribute of a view object row or entity object instance like attribute name, Java type, and SQL type. It also provides access to custom attribute-specific metadata properties that can be inspected by generic code you write, as well as UI hints that can assist in rendering an appropriate user interface display for the attribute and its value. Table D–8 describes the operations that you can perform on an attribute using the `AttributeDef` interface.

Table D–8 AttributeDef Interface

If you want to...	Call this AttributeDef interface method
Get the Java type of the attribute	`getJavaType()`
Get the SQL type of the attribute	`getSQLType()` **Notes:** The `int` value corresponds to constants in the JDBC class `java.sql.Types`.

Table D–8 (Cont.) AttributeDef Interface

If you want to...	Call this AttributeDef interface method
Determine the kind of attribute	`getAttributeKind()` **Notes:** A simple attribute is one that returns one of the constants `ATTR_PERSISTENT`, `ATTR_SQL_DERIVED`, `ATTR_TRANSIENT`, `ATTR_DYNAMIC`, `ATTR_ENTITY_DERIVED`. If the attribute is a 1-to-1 or many-to-1 association/viewlink accessor, it returns `ATTR_ASSOCIATED_ROW`. If the attribute is a 1-to-many or many-to-many association/viewlink accessor, it returns `ATTR_ASSOCIATED_ROWITERATOR`
Get the Java type of elements contained in an `Array`-valued attribute	`getElemJavaType()`
Get the SQL type of elements contained in an `Array`-valued attribute	`getElemSQLType()`
Get the name of the attribute	`getName()`
Get the index position of the attribute	`getIndex()`
Get the precision of a numeric attribute or the maximum length of a string attribute	`getPrecision()`
Get the scale of a numeric attribute	`getScale()`
Get the underlying column name corresponding to the attribute	`getColumnNameForQuery()`
Get attribute-specific custom property values	`getProperty()`,`getProperties()`
Get the UI `AttributeHints` object for the attribute	`getUIHelper()`
Test whether the attribute is mandatory	`isMandatory()`
Test whether the attribute is queriable	`isQueriable()`
Test whether the attribute is part of the primary key for the row	`isPrimaryKey()`

D.2.9 AttributeHints Interface

The `AttributeHints` interface exposes UI hint information that you can use to render an appropriate user interface display for the attribute and its value. Table D–9 describes the operations that you can perform on an attribute using the `AttributeHints` interface.

Table D–9 AttributeHints Interface

If you want to...	Call this AttributeHints interface method
Get the UI label for the attribute	`getLabel()`
Get the tooltip for the attribute	`getTooltip()`
Get the formatted value of the attribute, using any format mask supplied	`getFormattedAttribute()`

Table D–9 (Cont.) AttributeHints Interface

If you want to...	Call this AttributeHints interface method
Get the display hint for the attribute	`getDisplayHint()` **Notes:** The display hint will have a string value of either `Display` or `Hide`.
Get the preferred control type for the attribute	`getControlType()`
Parse a formatted string value using any format mask supplied for the attribute	`parseFormattedAttribute()`

D.3 Methods Used in the Business Service Tier

The implementation classes corresponding to the `oracle.jbo.*` interfaces, as described in Section D.2, "Methods Used in the Client Tier," are intentionally designed to *not* be directly accessed by client code. They reside in a different package named `oracle.jbo.server.*` and have the `Impl` suffix in their name to help remind you not to use them in your client-layer code.

In your business service tier implementation code, you can use any of the same methods that are available to clients, but in addition you can also:

- Safely cast any `oracle.jbo.*` interface to its `oracle.jbo.server.*` package implementation class and use any methods on that `Impl` class as well.
- Override any of the `public` or `protected` methods for the base framework implementation classes and write custom code in your component subclass before or after calling `super.`*`methodName`*`()` to augment or change the default functionality.

This section provides a summary of the most frequently called, written, and overridden methods for the key ADF Business Components classes.

D.3.1 Controlling Custom Java Files for Your Components

Before examining the specifics of individual classes, it's important to understand how you can control which custom Java files each of your components will use. When you don't need a customized subclass for a given component, you can just let the base framework class handle the implementation at runtime.

Each business component you create comprises a single XML component descriptor, and zero or more related custom Java implementation files. Each component that supports Java customization has a Java page in its component overview editor in the JDeveloper IDE. By selecting or deselecting the different Java classes, you control which ones will be created for your component. If none of the classes is specified, then your component will be an XML-only component, which simply uses the base framework class as its Java implementation. Otherwise, tick the checkbox of the related Java classes for the current component that you need to customize. JDeveloper will create a custom *subclass* of the framework base class in which you can add your code.

> **Note:** You can set up global IDE preferences for the Java classes to be generated by default for each ADF business component type by choosing **Tools** > **Preferences** > **Business Components** and ticking the checkboxes to indicate what you want your defaults to be.

A best practice is to *always* generate entity object and view row classes, even if you don't require any custom code in them other than the automatically generated getter and setter methods. These getter and setter methods offer you compile-time type checking that prevents errors surfacing at runtime in response to an attribute having been set to an incorrect kind of value.

D.3.2 ApplicationModuleImpl Class

The `ApplicationModuleImpl` class is the base class for application module components. Since the application module is the ADF component used to implement a business service, think of the application module class as the place where you can write your service-level application logic. The application module coordinates with view object instances to support updatable collections of value objects that are automatically "wired" to business domain objects. The business domain objects are implemented as ADF entity objects.

D.3.2.1 Methods You Typically Call on ApplicationModuleImpl

Table D–10 describes the operations that you can perform on an application module using the `ApplicationModuleImpl` class.

Table D–10 Methods You Typically Call on ApplicationModuleImpl

If you want to...	Call this method of the ApplicationModuleImpl class
Perform any of the common application module operations from inside your class, which can also be done from the client	For a list of these methods, see Section D.2.1, "ApplicationModule Interface."
Access a view object instance that you added to the application module's data model at design time	`get`*`ViewObjectInstanceName`*`()` **Notes:** JDeveloper generates this type-safe view object instance getter method for you to reflect each view object instance in the application module's design time data model.
Access the current `DBTransaction` object	`getDBTransaction()`
Access a nested application module instance that you added to the application module at design time	`get`*`AppModuleInstanceName`*`()` **Notes:** JDeveloper generates this type-safe application module instance getter method for you to reflect each nested application module instance added to the current application module at design time.

D.3.2.2 Methods You Typically Write in Your Custom ApplicationModuleImpl Subclass

Table D–11 describes the operations that you can perform on an application module using your custom `ApplicationModuleImpl` class.

Table D–11 Methods You Typically Write in Your Custom ApplicationModuleImpl Subclass

If you want to...	Write a method like this in your custom ApplicationModuleImpl class
Invoke a database stored procedure	`someCustomMethod()` **Notes:** Use the appropriate method on the `DBTransaction` interface to create a JDBC `PreparedStatement`. If the stored procedure has `OUT` parameters, then create a `CallableStatement` instead. For sample code that demonstrates encapsulating a call to a PL/SQL stored procedure inside your application module, see Section 12.6, "Invoking Stored Procedures and Functions."
Expose custom business service methods on your application module	`someCustomMethod()` **Notes:** Select the method name on the **Client Interface** page of the Edit Application Module dialog to expose it for client access if required.

JDeveloper can generate a custom `YourApplicationModuleName` interface containing service-level custom methods that you've chosen to expose to the client. You can use the **Client Interface** page of the Edit Application Module dialog to select the methods that you want to appear in your client interface.

D.3.2.3 Methods You Typically Override in Your Custom ApplicationModuleImpl Subclass

Table D–12 describes the operations that you can override on an application module using your custom `ApplicationModuleImpl` class.

Table D–12 Methods You Typically Override in Your Custom ApplicationModuleImpl Subclass

If you want to...	Override this method in your custom ApplicationModuleImpl class
Perform custom setup code the first time an application module is created and each subsequent time it gets used by a different client session.	`prepareSession()` **Notes:** This is the method you'd use to set up per-client context info for the current user in order to use Oracle's Virtual Private Database (VPD) features. It can also be used to set other kinds of PL/SQL package global variables, whose values might be client-specific, on which other stored procedures might rely. This method is also useful to perform setup code that is specific to a given view object *instance* in the application module. If instead of the view object setup code being instance-specific, you want it to be initialized for every instance ever created of that view object component, then put the setup logic in an overridden `create()` method in your `ViewObjectImpl` subclass instead.

Table D–12 (Cont.) Methods You Typically Override in Your Custom ApplicationModuleImpl Subclass

If you want to...	Override this method in your custom ApplicationModuleImpl class
Perform custom setup code after the application module's transaction is associated with a database connection from the connection pool	`afterConnect()` **Notes:** Can be a useful place to write a line of code that uses `getDBTransaction().executeCommand( )` to perform an `ALTER SESSION SET SQL TRACE TRUE` to enable database SQL trace logging for the current application connection. These logs can then be processed with the TKPROF utility to study the SQL statements being performed and the query optimizer plans that are getting used. For details about the TKPROF utility, see the "Understanding SQL Trace and TKPROF" section in the *Oracle Database Performance Tuning Guide*.
Perform custom setup code before the application module's transaction releases its database connection back to the database connection pool	`beforeDisconnect()` **Notes:** If you have set `jbo.doconnectionpooling` to `true`, then the connection is released to the database connection pool each time the application module is returned to the application module pool.
Write custom application module state to the state management XML snapshot	`passivateState()`
Read and restore custom application module state from the state management XML snapshot	`activateState()`

D.3.3 DBTransactionImpl2 Class

The `DBTransactionImpl2` class — which extends the base `DBTransactionImpl` class, and is constructed by the `DatabaseTransactionFactory` class — is the base class that implements the `DBTransaction` interface, representing the unit of pending work in the current transaction.

D.3.3.1 Methods You Typically Call on DBTransaction

Table D–13 describes the operations that you can perform on a transaction using the `DBTransaction` class.

Table D–13 Methods You Typically Call on DBTransaction

If you want to...	Call this method on the DBTransaction class
Commit the transaction	`commit()`
Roll back the transaction	`rollback()`
Eagerly validate any pending invalid changes in the transaction	`validate()`
Create a JDBC `PreparedStatement` using the transaction's `Connection` object	`createPreparedStatement()`

Table D–13 (Cont.) Methods You Typically Call on DBTransaction

If you want to...	Call this method on the DBTransaction class
Create a JDBC `CallableStatement` using the transaction's `Connection` object	`createCallableStatement()`
Create a JDBC `Statement` using the transaction's `Connection` object	`createStatement()`
Add a warning to the transaction's warning list	`addWarning()`

D.3.3.2 Methods You Typically Override in Your Custom DBTransactionImpl2 Subclass

Table D–14 describes the operations that you can perform on a transaction using your custom `DBTransactionImpl2` subclass.

Table D–14 Methods You Typically Override in Your Custom DBTransactionImpl2 Subclass

If you want to...	Override this method in your custom DBTransactionImpl2 class
Perform custom code before or after the transaction commit operation	`commit()`
Perform custom code before or after the transaction rollback operation	`rollback()`

In order for your custom `DBTransactionImpl2` subclass to be used at runtime, there are you must follow these steps:

1. Create a custom subclass of `DatabaseTransactionFactory` that overrides the create method to return an instance of your custom `DBTransactionImpl2` subclass like this:

```
package com.yourcompany.adfextensions;
import oracle.jbo.server.DBTransactionImpl2;
import oracle.jbo.server.DatabaseTransactionFactory;
import com.yourcompany.adfextensions.CustomDBTransactionImpl;
public class CustomDatabaseTransactionFactory
       extends DatabaseTransactionFactory {
  /**
   * Return an instance of our custom CustomDBTransactionImpl class
   * instead of the default implementation.
   *
   * @return An instance of our custom DBTransactionImpl2 implementation.
   */
  public DBTransactionImpl2 create() {
    return new CustomDBTransactionImpl();
  }
}
```

2. Tell the framework to use your custom transaction factory class by setting the value of the `TransactionFactory` configuration property to the fully qualified class name of your custom transaction factory. As with other configuration properties, if not supplied in the configuration XML file, it can be provided alternatively as a Java system parameter of the same name.

D.3.4 EntityImpl Class

The `EntityImpl` class is the base class for entity objects, which encapsulate the data, validation rules, and business behavior for your business domain objects.

D.3.4.1 Methods You Typically Call on EntityImpl

Table D–15 describes the operations that you can perform on an entity object using the `EntityImpl` class.

Table D–15 Methods You Typically Call on EntityImpl

If you want to...	Call this method in the EntityImpl class
Get the value of an attribute	`get`*`AttributeName`*`()` **Notes:** This code-generated getter method calls `getAttributeInternal()`, but provides compile-time type checking.
Set the value of an attribute	`set`*`AttributeName`*`()` **Notes:** This code-generated setter method calls `setAttributeInternal()`, but provides compile-time type checking.
Get the value of an attribute by name	`getAttributeInternal()`
Set the value of an attribute by name	`setAttributeInternal()`
Eagerly perform entity object validation	`validate()`
Refresh the entity from the database	`refresh()`
Populate the value of an attribute without *marking* it as being changed, but sending *notification* of its being changed so that the UI refreshes the value on the screen/page	`populateAttributeAsChanged()`
Access the `Definition` object for an entity	`getDefinitionObject()`
Get the `Key` object for an entity	`getKey()`
Determine the state of the entity instance, irrespective of whether it has already been posted (but not yet committed) in the current transaction	`getEntityState()` **Notes:** This method will return one of the constants `STATUS_UNMODIFIED`, `STATUS_INITIALIZED`, `STATUS_NEW`, `STATUS_MODIFIED`, `STATUS_DELETED`, or `STATUS_DEAD`, indicating the status of the entity instance in the current transaction.
Determine the state of the entity instance	`getPostState()` **Notes:** This method is typically relevant only if you are programmatically using the `postChanges()` method to post but not yet commit, entity changes to the database and need to detect the state of an entity with regard to its posting state.
Get the value originally read from the database for a given attribute	`getPostedAttribute()`

Table D–15 (Cont.) Methods You Typically Call on EntityImpl

If you want to...	Call this method in the EntityImpl class
Eagerly lock the database row for an entity instance	`lock()`

D.3.4.2 Methods You Typically Write in Your Custom EntityImpl Subclass

Table D–16 describes the operations that you can perform on an entity object using your custom `EntityImpl` subclass.

Table D–16 Methods You Typically Write in Your Custom EntityImpl Subclass

If you want to...	Write a method like this in your custom EntityImpl subclass
Perform attribute-specific validation	`public boolean validate`*`Something`*`(`*`AttrType`*`value)` **Notes:** Register the attribute validator method by adding a MethodValidator rule on the correct attribute in the **Validation** page of the Edit Entity Object dialog.
Perform entity-level validation	`public boolean validate`*`Something`*`()` **Notes:** Register the entity-level validator method by adding a MethodValidator rule on the entity in the **Validation** panel of the Edit Entity Object dialog.
Calculate the value of a transient attribute	Add your calculation code to the generated `get`*`AttributeName`*`()` method.

D.3.4.3 Methods You Typically Override in Your Custom EntityImpl Subclass

Table D–17 describes the operations that you can override on an entity object using your custom `EntityImpl` subclass.

Table D–17 Methods You Typically Override in Your Custom EntityImpl Subclass

If you want to...	Override this method in your EntityImpl subclass
Set calculated default attribute values, including programmatically populating the primary key attribute value of a new entity instance	`create()` **Notes:** After calling `super.create()`, call the appropriate `set`*`AttrName`*`()` method(s) to set the default values for the attributes.
Modify attribute values before changes are posted to the database	`prepareForDML()`
Augment or change the standard `INSERT`, `UPDATE`, or `DELETE` DML operation that the framework will perform on your entity object's behalf to the database	`doDML()` **Notes:** This method checks the value of the operation flag to the constants `DML_INSERT`, `DML_UPDATE`, or `DML_DELETE` to test what DML operation is being performed.

Table D–17 (Cont.) Methods You Typically Override in Your Custom EntityImpl Subclass

If you want to...	Override this method in your EntityImpl subclass
Perform complex, SQL-based validation after all entity instances have been posted to the database but before those changes are committed	`beforeCommit()`
Insure that a related, newly-created, parent entity gets posted to the database *before* the current child entity on which it depends	`postChanges()` **Notes:** If the parent entity is related to this child entity via a composition association, then the framework already handles posting the changes automatically. If they are only associated (but not composed), then you need to override `postChanges()` to force a newly created parent entity to post before the current, dependent child entity. For an example of the code you typically write in your overridden `postChanges()` method to accomplish this, see Section 4.14.7.3, "Overriding postChanges() to Control Post Order."

Note: It is possible to write attribute-level validation code directly inside the appropriate `setAttributeName` method of your `EntityImpl` class; however, adopting the `MethodValidator` approach suggested in Table D–16 conveniently places all the validations in effect on the **Validation Rules** page of the overview editor for the attributes of the entity object.

WARNING: It is also possible to override the `validateEntity()` method to write entity-level validation code; however, if you want to maintain the benefits of the ADF bundled exception mode — where the framework collects and reports a *maximal* set of validation errors back to the client user interface — use the `MethodValidator` approach suggested in Table D–16. This allows the framework to automatically collect all of your exceptions that your validation methods throw without your having to understand the bundled exception implementation mechanism. Overriding the `validateEntity()` method directly shifts the responsibility onto your *own* code to correctly catch and bundle the exceptions that Oracle ADF would have caught by default, which is non-trivial and a chore to remember and hand-code each time.

D.3.5 EntityDefImpl Class

The `EntityDefImpl` class is a singleton, shared metadata object for all entity objects of a given type in a single Java VM. For instance, if the Java VM contains two instances of the `EntityDefImpl` class, one with the customers XML data and one with the orders, there would only ever be one instance of each. This class defines the structure of the entity instances and provides methods to create new entity instances and find existing instances by their primary key.

D.3.5.1 Methods You Typically Call on EntityDefImpl

Table D–18 describes the operations that you can perform on an entity object using the `EntityDefImpl` class.

Table D–18 Methods You Typically Call on EntityDefImpl

If you want to...	Call this method in the EntityDefImpl class
Find an entity object of a given type by its primary key	`findByPrimaryKey()` **Notes:** For a tip about getting `findByPrimaryKey()` to find entity instances of subtype entities as well, see Section 4.19.4.2, "Subtype Entity Objects and the findByPrimaryKey() Method."
Access the current `DBTransaction` object	`getDBTransaction()`
Find any `EntityDefImpl` object by its fully qualified name	`findDefObject()` (static method)
Retrieve the value of an entity object's custom property	`getProperty()`,`getProperties()`
Set the value of an entity object's custom property	`setProperty()`
Create a new instance of an entity object	`createInstance2()` **Notes:** Alternatively, you can expose custom `create`*`XXX`*`()` methods with your own expected signatures in that same custom `EntityDefImpl` subclass. See Section D.3.5.2, "Methods You Typically Write in Your Custom EntityDefImpl Class" for details.
Iterate over the entity instances in the cache of this entity type	`getAllEntityInstancesIterator()`
Access an array list of entity definition objects for entities that extend the current one.	`getExtendedDefObjects()`

D.3.5.2 Methods You Typically Write in Your Custom EntityDefImpl Class

Table D–19 describes the operations that you can perform on an entity object using your custom `EntityDefImpl` class.

Table D–19 Methods You Typically Write on EntityDefImpl

If you want to...	Write a method like this in your custom EntityDefImpl class
Allow other classes to create an entity instance with an initial type-safe set of attribute values or setup information	`create`*`XXX`*`(`*`Type1arg1`*`,...,`*`TypeNargN`*`)` **Notes:** Internally, using this method would create and populate an instance of a `NameValuePairs` object (which implements `AttributeList`) and call the protected method `createInstance()`, passing that `NameValuePairs` object. Make sure that the method is `public` if other classes need to be able to call it.

D.3.5.3 Methods You Typically Override in Your Custom EntityDefImpl

Table D–20 describes the operations that you can perform on an entity object using the `EntityDefImpl` class.

Table D–20 Methods You Typically Override on EntityDefImpl

If you want to...	Override this method in your custom EntityDefImpl class
Perform custom metadata initialization when this singleton metaobject is loaded	`createDef()`
Avoid using the `RETURNING INTO` clause to support refresh-on-insert or refresh-on-update attributes	`isUseReturningClause()` **Notes:** Set this method to return `false` to disable the use of `RETURNING INTO`, necessary sometimes when your entity object is based on a view with `INSTEAD OF` triggers that don't support `RETURNING INTO` at the database level.
Control whether the `UPDATE` statements issued for this entity update only changed columns or for all columns	`isUpdateChangedColumns()` **Notes:** Defaults to `true`.
Find any `EntityDefImpl` object by its fully qualified name	`findDefObject()` **Notes:** Static method.
Set the value of an entity object's custom property	`setProperty()`
Allow other classes to create a new instance of an entity object without doing so implicitly via a view object	`createInstance()` **Notes:** If you don't write a custom create method as noted in Section D.3.5.2, "Methods You Typically Write in Your Custom EntityDefImpl Class", you'll need to override this method and widen the visibility from `protected` to `public` to allow other classes to construct an entity instance.

D.3.6 ViewObjectImpl Class

The `ViewObjectImpl` class is the base class for view objects.

D.3.6.1 Methods You Typically Call on ViewObjectImpl

Table D–21 describes the operations that you can perform on a view object using the `ViewObjectImpl` class.

Table D–21 Methods You Typically Call on ViewObjectImpl

If you want to...	Call this method in the ViewObjectImpl class
Perform any of the common view object, row set, or row set iterator operations from inside your class, which can also be done from the client	For more information about operations at the view object, row set, or row set iterator-level, see Section D.2.3, "ViewObject Interface," Section D.2.4, "RowSet Interface," and Section D.2.5, "RowSetIterator Interface."

Table D–21 (Cont.) Methods You Typically Call on ViewObjectImpl

If you want to...	Call this method in the ViewObjectImpl class
Set an additional runtime `WHERE` clause on the default row set	`setWhereClause()`
Define a named bind parameter	`defineNamedWhereClauseParam()`
Remove a named bind parameter	`removeNamedWhereClauseParam()`
Set bind variable values on the default row set by name	`setNamedWhereClauseParam()` **Notes:** Only works when you have formally defined named bind variables on your view object.
Set bind variable values on the default row set	`setWhereClauseParam()` **Notes:** Use this method for view objects with binding style of "Oracle Positional" or "JDBC Positional" when you have not formally defined named bind variables.
Retrieve a subset of rows in a view object's row set based on evaluating an in-memory filter expression	`getFilteredRows()`
Retrieve a subset of rows in the current range of a view object's row set based on evaluating an in-memory filter expression	`getFilteredRowsInRange()`
Override the runtime display of criteria items that appear inside an ADF query component. Subclasses may override to return custom `AttributeHints` implementation for the given criteria item. Returns null by default.	`getCriteriaItemAttributeHints()`
Set the number of rows that will be fetched from the database per roundtrip for this view object	`setFetchSize()` **Notes:** The default fetch size is a single row at a time. This is definitely not optimal if your view object intends to retrieve many rows, so you should either set the fetch size higher at design time on the **Tuning** page of the Edit View Object dialog, or set it at runtime using this method.
Force a row set's query to be (re)executed specifically on a lookup view object instance in a shared application module	`forceExecuteQueryOfSharedVO()` **Notes:** Reexecuting the query forces a new query collection and will prevent the application module cache from being used. You should only use this method when you are sure that you are accessing the shared application module during setup and not during runtime. This method when used during normal runtime may have unintended side-effects that disrupt the navigation of users accessing the collection concurrently. If you want to refresh the collection from the cache without creating a new query collection, call `executeQuery()` instead.

D.3.6.2 Methods You Typically Write in Your Custom ViewObjectImpl Subclass

Table D–22 describes the operations that you can perform on a view object using your custom `ViewObjectImpl` subclass.

Table D–22 Methods You Typically Write in Your Custom ViewObjectImpl Subclass

If you want to...	Write a method like this in your custom ViewObjectImpl subclass
Provide clients with type-safe methods to set bind variable values without exposing positional details of the bind variables themselves	*`someMethodName`*`(`*`Type1arg1`*`,...,`*`TypeNargN`*`)` **Notes:** Internally, this method would call the `setWhereClauseParam()` method to set the correct bind variables with the values provided in the type-safe method arguments.

JDeveloper can generate a custom *`YourViewObjectName`* interface containing view object custom methods that you've chosen to expose to the client. You can use the **Client Interface** page of the Edit View Object to select the methods that you want to appear in your client interface.

D.3.6.3 Methods You Typically Override in Your Custom ViewObjectImpl Subclass

Table D–23 describes the operations that you can perform on a view object using your custom `ViewObjectImpl` subclass.

Table D–23 Methods You Typically Override in Your Custom ViewObjectImpl Subclass

If you want to...	Override this method in your custom ViewObjectImpl subclass
Initialize custom view object class members (not row attributes) when the view object instance is created for the first time	`create()` **Notes:** This method is useful to perform set up logic that is applicable to every instance of a view object that will ever get created, in the context of any application module. If instead of *generic* view object setup logic, you need to perform logic specific to a given view object *instance* in an application module, then override the `prepareSession()` method of your application module's `ApplicationModuleImpl` subclass and perform the logic there after calling `findViewObject()` to find the view object instance whose properties you want to set.
Write custom view object instance state to the state management XML snapshot	`passivateState()`
Read and restore custom view object instance state from the state management XML snapshot	`activateState()`

Table D–23 (Cont.) Methods You Typically Override in Your Custom ViewObjectImpl

If you want to...	Override this method in your custom ViewObjectImpl subclass
Customize the execution of the view object query to utilize an alternative data source	`executeQueryForCollection()` **Notes:** By default view objects read their data from the database and automate the task of working with the JDBC layer to process the database result sets. However, by overriding appropriate methods in its custom Java class, you can create a view object that programmatically retrieves data from alterative data sources, as described in Section 42.8, "Using Programmatic View Objects for Alternative Data Sources."
Customize the programmatic view object to utilize an alternative data source and determine whether the query collection has more rows to fetch from the query execution	`hasNextForCollection()`
Customize the programmatic view object to utilize an alternative data source and populate each row of the retrieved data	`createRowFromResultSet()`
Customize the programmatic view object to utilize an alternative data source and return a count of the number of rows that will be retrieved	`getQueryHitCount()`
Customize the programmatic view object to utilize an alternative data source and release any resources that may be associated with a row set that is being closed	`releaseUserDataForCollection()`
Change or augment the way that the `ViewCriteria` collection of `ViewCriteriaRow`s is converted into a Query-by-Example `WHERE` clause	`getViewCriteriaClause(boolean)`

D.3.7 ViewRowImpl Class

The `ViewRowImpl` class is the base class for view row objects.

D.3.7.1 Methods You Typically Call on ViewRowImpl

Table D–24 describes the operations that you can perform on a view object row using your custom `ViewRowImpl` class.

Table D–24 Methods You Typically Call on ViewRowImpl

If you want to...	Call this method in your custom ViewRowImpl class
Perform any of the common view row operations from inside your class, which can also be done from the client	For more information about the row-level operations, see Section D.2.6, "Row Interface."
Get the value of an attribute	`get`*`AttrName`*`()`
Set the value of an attribute	`set`*`AttrName`*`()`

Table D–24 (Cont.) Methods You Typically Call on ViewRowImpl

If you want to...	Call this method in your custom ViewRowImpl class
Access the underlying entity instance to which this view row is delegating attribute storage	`get`*`EntityUsageAliasName`*`()` **Notes:** You can change the name of the entity usage alias name on the **Entity Objects** page of the Edit View Object dialog.

D.3.7.2 Methods You Typically Write in Your Custom ViewRowImpl Class

Table D–25 describes the operations that you can perform on a view object row using your custom `ViewRowImpl` class.

Table D–25 Methods You Typically Write on ViewRowImpl

If you want to...	Write a method like this in your custom ViewRowImpl class
Calculate the value of a view object-level transient attribute	`get`*`AttrName`*`()` **Notes:** JDeveloper generates the skeleton of the method for you, but you need to write the custom calculation logic inside the method body.
Perform custom processing of the setting of a view row attribute	`set`*`AttrName`*`()` **Notes:** JDeveloper generates the skeleton of the method for you, but you need to write the custom logic inside the method body if required.
Determine the updateability of an attribute in a conditional way	`isAttributeUpdateable()`
Expose logical operations on the current row, optionally callable by clients	`doSomething()` **Notes:** Often these view-row-level custom methods simply turn around and delegate to a method call on the underlying entity object related to the current row.

JDeveloper can generate a custom *`YourViewObjectName`*`Row` interface containing view row custom methods that you've chosen to expose to the client. You can use the **Client Row Interface** page of the Edit View Object dialog to select the methods that you want to appear in your client interface.

D.3.7.3 Methods You Typically Override in Your Custom ViewRowImpl Subclass

Table D–26 describes the operations that you can perform on a view object row using your custom `ViewRowImpl` subclass.

Table D–26 Methods You Typically Override in Your Custom ViewRowImpl Subclass

If you want to...	Write a method like this in your ViewRowImpl subclass
Determine the updateability of an attribute in a conditional way	`isAttributeUpdateable()`

E

ADF Business Components Java EE Design Pattern Catalog

This appendix provides a brief summary of the Java Platform, Enterprise Edition (Java EE) design patterns that the ADF Business Components layer implements for you.

By using the Oracle Application Development Framework's business components building-blocks and related design time extensions to JDeveloper, you get a prescriptive architecture for building richly functional and cleanly layered Java EE business services with great performance.

Table E–1 provides a brief overview of the numerous design patterns that the ADF Business Components layer implements for you. Some are the familiar patterns from Sun's Java EE BluePrints and some are design patterns that ADF Business Components adds to the list. For details about Java EE BluePrints, see the BluePrints page at the Oracle Technology Network web site at `http://www.oracle.com/technetwork/java/index-jsp-136701.html`.

Table E–1 Java EE Design Patterns Implemented by ADF Business Components

Pattern Name and Description	How ADF Business Components Implements It
Model/View/Controller Cleanly separates the roles of data and presentation, allowing multiple types of client displays to work with the same business information.	The ADF application module provides a generic implementation of a Model/View/Controller "application object" that simplifies exposing the application data model for any application or service, and facilitates declaratively specifying the boundaries of a logical unit of work. Additional UI-centric frameworks and tag libraries provided in JDeveloper help you implement the view and controller layers.
Interface / Implementation Separation Cleanly separates the API or Interface for components from their implementation class.	ADF Business Components enforces a logical separation of client-tier accessible functionality (via interfaces) and its business tier implementation. JDeveloper handles the creation of custom interfaces and client proxy classes automatically.
Service Locator Abstracts the technical details of locating a service so that the client can use it more easily.	ADF application modules are looked up using a simple configuration object which hides the low-level details of finding the service instance behind the scenes. For Fusion web applications, it also hides the implementation of the application module pool usage, a lightweight pool of service components that improves application scalability.

Table E–1 (Cont.) Java EE Design Patterns Implemented by ADF Business Components

Pattern Name and Description	How ADF Business Components Implements It
Inversion of Control A containing component orchestrates the lifecycle of the components it contains, invoking specific methods that you can override at the appropriate times, so as to be able to focus more on what the code should do, instead of when it should be executed.	ADF components contain a number of easy-to-override methods that the framework invokes as needed during the course of application processing.
Dependency Injection Simplifies application code, and increases configuration flexibility by deferring component configuration and assembly to the container.	ADF Business Components configures all its components from externalized XML metadata definition files. At runtime, the framework automatically injects dependent objects like view object instances into your application module service component and entity objects into your view rows, implementing lazy loading. It supports runtime factory substitution of components by any customized subclass of that component to simplify onsite application customization scenarios. Much of the ADF Business Components functionality is implemented via dynamic injection of validator and listener subscriptions that coordinate the framework interactions depending on what declarative features have been configured for each component in their XML metadata.
Active Record Avoids the complexity of "anything to anything" object/relational mapping, by providing an object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.	ADF entity objects handle the database mapping functionality you use most frequently, including inheritance, association, and composition support, so you don't have to focus on object/relational mapping. They also provide a place to encapsulate both declarative business rules and one-off programmatic business domain.
Data Access Objects Prevents unnecessary marshalling overhead by implementing dependent objects as lightweight, persistent classes instead of each as an individual enterprise bean. Isolates persistence details into a single, easy-to-maintain class.	ADF view objects automate the implementation of data access for reading data using SQL statements. ADF entity objects automate persistent storage of lightweight business entities. ADF view objects and entity objects cooperate to provide a sophisticated, performant data access objects layer, where any data queried through a view object can optionally be made fully updatable without requiring that you write any "application plumbing" code.
Session Facade Prevents inefficient client access of entity beans and inadvertent exposure of sensitive business information by wrapping entity beans with a session bean.	ADF application modules are designed to implement a coarse-grained "service facade" architecture in any of their supported deployment modes. When deployed as EJB session beans or as a service interface, they provide an implementation of the Session Facade pattern automatically.

Table E–1 (Cont.) Java EE Design Patterns Implemented by ADF Business Components

Pattern Name and Description	How ADF Business Components Implements It
Value Object Prevents unnecessary network roundtrips by creating one-off "transport" objects to group a set of related attributes needed by a client program.	ADF Business Components provides an implementation of a generic `Row` object, which is a metadata-driven container of any number and kind of attributes that need to be accessed by a client. The developer can work with the generic `Row` interface and do late-bound `getAttribute("Price")` and `setAttribute("Quantity")` calls, or optionally generate early-bound row interfaces like `OverdueOrdersRow`, to enable type-safe method calls like `getPrice()` and `setQuantity()`. Smarter than just a simple "bag 'o attributes", the ADF `Row` object can be introspected at runtime to describe the number, names, and types of the attributes in the row, enabling sophisticated, generic solutions to be implemented.
Page-by-Page Iterator Prevents sending unnecessary data to the client by breaking a large collection into page-sized "chunks" for display.	ADF Business Components provides an implementation of a generic `RowSet` interface which manages result sets produced by executing view object SQL queries. The `RowSet` interface allows you to set a desired page size, for example 10 rows, and page up and down through the query results in these page-sized chunks. Since data is retrieved lazily, only data the user actually visits will ever be retrieved from the database on the backend, and in the client tier the number of rows in the page can be returned over the network in a single roundtrip.
Fast-Lane Reader Prevents unnecessary overhead for read-only data by accessing JDBC APIs directly. This allows an application to retrieve only the attributes that need to be displayed, instead of finding all of the attributes by primary key when only a few are required by the client. Typically, implementations of this pattern sacrifice data consistency for performance, since queries performed at the raw JDBC level do not "see" pending changes made to business information represented by enterprise beans.	ADF view objects read data directly from the database for best performance; however, they give you a choice regarding data consistency. If updateability and/or consistency with pending changes is desired, you need only associate your view object with the appropriate entity objects whose business data is being presented. If consistency is not a concern, view objects can simply perform the query with no additional overhead. In either case, you never have to write JDBC data access code. You need only provide appropriate SQL statements in XML descriptors.
(Bean) Factory Allows runtime instantiation and configuration of an appropriate subclass of a given interface or superclass based on externally configurable information.	All ADF component instantiation is done based on XML configuration metadata through factory classes allowing runtime substitution of specialized components to facilitate application customization.
Entity Facade Provides a restricted view of data and behavior of one or more business entities.	ADF view objects can surface any set of attributes and methods from any combination of one or more underlying entity objects to furnish the client with a single, logical value object to work with.

Table E–1 (Cont.) Java EE Design Patterns Implemented by ADF Business Components

Pattern Name and Description	How ADF Business Components Implements It
Value Messenger Keeps client value object attributes in sync with the middle-tier business entity information that they represent in a bidirectional fashion.	The ADF Business Components value object implementation coordinates with a client-side value object cache to batch attribute changes to the EJB tier and receive batch attribute updates which occur as a result of middle-tier business logic. The ADF Value Messenger implementation is designed to not require any kind of asynchronous messaging to achieve this effect.
Continuations Gives you the simplicity and productivity of a stateful programming model with the scalability of a stateless web solution.	ADF Business Components application module pooling and state management functionality combine to deliver this value-add. Application module pooling eliminates the need to dedicate application server tier resources to individual users and supports a "stateless with user affinity" optimization that you can tune.

F

ADF Equivalents of Common Oracle Forms Triggers

This appendix provides a quick summary of how basic tasks performed with the most common Oracle Forms triggers are accomplished using Oracle ADF.

This appendix includes the following sections:

- Section F.1, "Validation and Defaulting (Business Logic)"
- Section F.2, "Query Processing"
- Section F.3, "Database Connection"
- Section F.4, "Transaction "Post" Processing (Record Cache)"
- Section F.5, "Error Handling"

F.1 Validation and Defaulting (Business Logic)

Table F–1 ADF Equivalents for Oracle Forms Validation and Defaulting Triggers

Forms Trigger	ADF Equivalent
`WHEN-VALIDATE-RECORD` Execute validation code at the record level	In the custom `EntityImpl` class for your entity object, write a public method returning a `boolean` type with a method name like `validate`*`XXXX`*`()` and have it return `true` if the validation succeeds or `false` if the validation fails. Then, add a Method validator for this validation method to your entity object at the entity level. When doing that, you can associate a validation failure message with the rule.
`WHEN-VALIDATE-ITEM` Execute validation code at the field level	In the custom `EntityImpl` class for your entity object, write a public method returning a `boolean` type and accepting a single argument of the same data type as your attribute, having a method name like `validate`*`XXXX`*`()`. Have it return `true` if the validation succeeds or `false` if the validation fails. Then, add a Method validator for this validation method to the entity object at the attribute level for the appropriate attribute. When doing that, you can associate a validation failure message with the rule.
`WHEN-DATABASE-RECORD` Execute code when a row in the data block is marked for insert or update	Override the `addToTransactionManager()` method of your entity object. Write code after calling the super.

Table F–1 (Cont.) ADF Equivalents for Oracle Forms Validation and Defaulting Triggers

Forms Trigger	ADF Equivalent
`WHEN-CREATE-RECORD` Execute code to populate complex default values when a new record in the data block is created, without changing the modification status of the record	Override the `create()` method of your entity object and after calling the super, use appropriate `set`*`AttrName`*`()` methods to set default values for attributes as necessary. To immediately set a primary key attribute to the value of a sequence, construct an instance of the `SequenceImpl` helper class and call its `getSequenceNumber()` method to get the next sequence number. Assign this value to your primary key attribute. If you want to wait to assign the sequence number until the new record is saved, but still without using a database trigger, you can use this technique in an overridden `prepareForDML()` method in your entity object. If instead you want to assign the primary key from a sequence using your own `BEFOREINSERTFOREACHROW` database trigger, then use the special data type called `DBSequence` for your primary key attribute instead of the regular `Number` type.
`WHEN-REMOVE-RECORD` Execute code whenever a row is removed from the data block	Override the `remove()` method of your entity object and write code either before or after calling the super.

F.2 Query Processing

Table F–2 ADF Equivalents for Oracle Forms Query Processing Triggers

Forms Trigger	ADF Equivalent
`PRE-QUERY` Execute logic before executing a query in a data block, typically to set up values for Query-by-Example criteria in the "example record"	Override the `executeQueryForCollection()` method on your view object class and write code before calling the super.
`ON-COUNT` Override default behavior to count the query hits for a data block	Override the `getQueryHitCount()` method in your view object and do something instead of calling the super.
`POST-QUERY` Execute logic after retrieving each row from the data source for a data block.	Generally instead of using a `POST-QUERY` style technique to fetch descriptions from other tables based on foreign key values in the current row, in ADF it's more efficient to build a view object that has multiple participating entity objects, joining in all the information you need in the query from the main table, as well as any auxiliary or lookup-value tables. This way, in a single roundtrip to the database you get all the information you need. If you still need a per-fetched-row trigger like `POST-QUERY`, override the `createInstanceFromResultSet()` method in your view object class.
`ON-LOCK` Override default behavior to attempt to acquire a lock on the current row in the data block	Override the `lock()` method in your entity object class and do something instead of calling the super.

F.3 Database Connection

Table F–3 ADF Equivalents for Oracle Forms Database Connection Triggers

Forms Trigger	ADF Equivalent
`POST-LOGON` Execute logic after logging into the database	Override the `afterConnect()` method on your custom application module. Since application module instances can stay connected while serving different logical client sessions, you can override the `prepareSession()` method, which is fired after initial login, as well as after any time the application module is accessed by a user that was different from the one that accessed it last time.
`PRE-LOGOUT` Execute logic before logging out of the database	Override the `beforeDisconnect()` method on your custom application module class.

F.4 Transaction "Post" Processing (Record Cache)

Table F–4 ADF Equivalents for Oracle Forms Transactional Triggers

Forms Trigger	ADF Equivalent
`PRE-COMMIT` Execute code before commencing processing of the changed rows in all data blocks in the transaction	Override the `commit()` method in a custom `DBTransactionImpl` class and write code before calling the super. **Note:** For an overview of creating and using a custom `DBTransaction` implementation, see Section 12.8.5.1, "Creating a Custom Database Transaction Framework Extension Class."
`PRE-INSERT` Execute code before a new row in the data block is inserted into the database during "post" processing	Override the `doDML()` method in your entity class, and if the `operation` equals `DML_INSERT`, then write code *before* calling the super.
`ON-INSERT` Override default processing for inserting a new row into the database during "post" processing	Override the `doDML()` method in your entity class, and if the `operation` equals `DML_INSERT`, then write code *instead of* calling the super.
`POST-INSERT` Execute code after new row in the data block is inserted into the database during "post" processing	Override the `doDML()` method in your entity class, and if the `operation` equals `DML_INSERT`, then write code *after* calling the super.
`PRE-DELETE` Execute code before a row removed from the data block is deleted from the database during "post" processing	Override the `doDML()` method in your entity class, and if the `operation` equals `DML_DELETE`, then write code *before* calling the super.
`ON-DELETE` Override default processing for deleting a row removed from the data block from the database during "post" processing	Override the `doDML()` method in your entity class, and if the `operation` equals `DML_DELETE`, then write code *instead of* calling the super.

Table F–4 (Cont.) ADF Equivalents for Oracle Forms Transactional Triggers

Forms Trigger	ADF Equivalent
`POST-DELETE` Execute code after a row removed from the data block is deleted from the database during "post" processing	Override the `doDML()` method in your entity class, and if the `operation` equals `DML_DELETE`, then write code *after* calling the super.
`PRE-UPDATE` Execute code before a row changed in the data block is updated in the database during "post" processing	Override the `doDML()` method in your entity class, and if the `operation` equals `DML_UPDATE`, then write code *before* calling the super.
`ON-UPDATE` Override default processing for updating a row changed in the data block from the database during "post" processing	Override the `doDML()` method in your entity class, and if the `operation` equals `DML_UPDATE`, then write code *instead of* calling the super.
`POST-UPDATE` Execute code after a row changed in the data block is updated in the database during "post" processing	Override the `doDML()` method in your entity class, and if the `operation` equals `DML_UPDATE`, then write code *after* calling the super.
`POST-FORMS-COMMIT` Execute code after Forms has "posted" all necessary rows to the database, but before issuing the data commit to end the transaction	If you want a single block of code for the whole transaction, you can override the `doCommit()` method in a custom `DBTransactionImpl` object and write code before calling the super. To execute entity-specific code before commit for each affected entity in the transaction, override the `beforeCommit()` method on your entity object, and write code there.
`POST-DATABASE-COMMIT` Execute code after database transaction has been committed	Override the `commit()` method in a custom `DBTransactionImpl` class, and write code *after* calling the super.

F.5 Error Handling

Table F–5 ADF Equivalents for Oracle Forms Error Handling Triggers

Forms Trigger	ADF Equivalent
`ON-ERROR` Override default behavior for handling an error	Install a custom error handler (`DCErrorHandler`) on the ADF `BindingContext`.

G

Performing Common Oracle Forms Tasks in Oracle ADF

This appendix describes how some common Oracle Forms tasks are implemented in Oracle ADF. In Oracle Forms, you do some tasks in the data block, and others in the UI. For this reason, the appendix is divided into two sections: tasks that relate to data, and tasks that relate to the UI.

This appendix includes the following sections:

- Section G.1, "Performing Tasks Related to Data"
- Section G.2, "Performing Tasks Related to the User Interface"

G.1 Performing Tasks Related to Data

In Oracle Forms, tasks that relate solely to data are performed in the data block. In Oracle ADF, these tasks are done on the business components that persist data (entity objects) and on the objects that query data (view objects).

G.1.1 How to Retrieve Lookup Display Values for Foreign Keys

In Oracle Forms, an editable table often has foreign key lookup columns to other tables. The user-friendly display values corresponding to the foreign key column values exist in related tables. You often need to present these related display values to the user.

In Oracle Forms, this was a complicated task that required adding nondatabase items to the data block, adding a block-level `POST-QUERY` trigger to the data block, and writing a SQL select statement for each foreign key attribute. Additionally, if the user changed the data, you needed to sync the foreign key values with an item-level `WHEN-VALIDATE-ITEM` trigger. This process is much easier in Oracle ADF.

Implementation of the task in Oracle ADF

1. Create a view object that includes the following:
   - The main, editable entity object as the primary entity usage
   - Secondary "reference" entity usages for the one or more associated entities whose underlying tables contain the display text

   For more information, see Section 5.5.1, "How to Create Joins for Entity-Based View Objects."

2. Select the desired attributes (at least the display text) from the secondary entity usages as described in Section 5.5.2, "How to Select Additional Attributes from Reference Entity Usages."

   At runtime, the data for the main entity and all related lookup display fields is retrieved from the database in a single join.

   If the user can change the data, no additional steps are required. If the user changes the value of a foreign key attribute, the reference information is automatically retrieved for the new, related row in the associated table.

G.1.2 How to Get the Sysdate from the Database

In Oracle Forms, when you wanted to get the current date and time, you retrieved the `sysdate` from the database. In Oracle ADF, you also have the option of getting the system date using a Java method or a Groovy expression.

Implementation of the task in Oracle ADF

To get the system date from the database, you can use the following Groovy expression at the entity level:

```
DBTransaction.currentDbTime
```

Note: The `DBTransaction` reference is for entity-level Groovy expressions only.

If you want to assign a default value to an attribute using this Groovy expression, see Section 4.10.7, "How to Define a Default Value Using a Groovy Expression."

To get the system date from Java, you call the `getCurrentDate()` method. For more information, see Section 8.10, "Accessing the Current Date and Time."

G.1.3 How to Implement an Isolation Mode That Is Not Read Consistent

In Oracle Forms, you might have been concerned with *read consistency*, that is, the ability of the database to deliver the state of the data at the time the SQL statement was issued.

Implementation of the task in Oracle ADF

If you use an entity-based view object, the query sees the changes currently in progress by the current user's session in the pending transaction. This is the default behavior, and the most accurate.

If instead, you want a snapshot of the data on the database without considering the pending changes made by the current user, you can use a read-only view object and reexecute the query to see the latest committed database values. For more information on read-only view objects, see Section 5.2.3, "How to Create an Expert Mode, Read-Only View Object."

G.1.4 How to Implement Calculated Fields

Calculated fields are often used to show the sum of two values, but they could also be used for the concatenated value of two or more fields, or the result of a method call.

Implementation of the task in Oracle ADF

Calculated attributes are usually not stored in the database, as their values can easily be obtained programmatically. Attributes that are used in the middle tier, but that are not stored in the database are called *transient attributes*. Transient attributes can be defined at the entity object level or the view object level.

If a transient attribute will be used by more than one view object that might be based on an entity object, then define the attribute at the entity object level. Otherwise, define the transient attribute at the view object level for a particular view object.

To define transient attributes at the entity object level, see Section 4.11, "Adding Transient and Calculated Attributes to an Entity Object." To define transient attributes at the view object level, see Section 5.14, "Adding Calculated and Transient Attributes to a View Object."

G.1.5 How to Implement Mirrored Items

In Oracle Forms, you may be used to using mirrored items to show two or more fields that share identical values.

Implementation of the task in Oracle ADF

There is no need to have mirrored items in Oracle ADF, because the UI and data are separated. The same view object can appear on any number of pages, so you don't need to create mirrored items that have the same value. Likewise, a form could have the same field represented in more than one place and it would not have to be mirrored.

G.1.6 How to Use Database Columns of Type CLOB or BLOB

If you are used to working with standard database types, you may be wondering how to use the `CLOB` and `BLOB` types in Oracle ADF.

Implementation of the task in Oracle ADF

In Oracle ADF, use the built-in data types `ClobDomain` or `BlobDomain`. These are automatically created when you reverse-engineer entity objects or view objects from existing tables with these column types. ADF Business Components also supports data types for Intermedia column types: `OrdImage`, `OrdAudio`, `OrdDoc`, and `OrdVideo`. For more information, see Section 4.10.1, "How to Set Database and Java Data Types for an Entity Object Attribute."

G.2 Performing Tasks Related to the User Interface

In Oracle ADF, common UI-related tasks (such as master-detail screens, popup list of values, and page layout) are handled quite differently than they were in Oracle Forms. This section describes how to perform some common Oracle Forms tasks that relate to the UI with Oracle ADF.

G.2.1 How to Lay Out a Page

Oracle Forms is based on an absolute pixel or point-based layout, as compared to the container-based approach of JSF, and the Layout Manager approach in ADF Swing.

Implementation of the task in Oracle ADF

See the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework* for information on how to lay out a page in Oracle ADF.

G.2.2 How to Stack Canvases

In Oracle Forms, stacked canvases were often used to hide and display areas of the screen.

Implementation of the task in Oracle ADF

The analog of stacked canvases in Oracle ADF is panels (layout containers) with the rendered property set to `true` or `false`. See the *Oracle Fusion Middleware Web User Interface Developer's Guide for Oracle Application Development Framework* for more information.

G.2.3 How to Implement a Master-Detail Screen

Master-detail relationships in Oracle ADF are coordinated through a view link. A view link is conceptually similar to a Oracle Forms *relation*.

Implementation of the task in Oracle ADF

For information on how create view links, see Section 5.6, "Working with Multiple Tables in a Master-Detail Hierarchy." Once you have established a relationship between two view objects with a view link, see Section 5.6.4, "How to Enable Active Master-Detail Coordination in the Data Model."

G.2.4 How to Implement an Enter Query Screen

In Oracle Forms, another common task was creating an *enter query screen*. That is, a screen that starts in Find mode.

Implementation of the task in Oracle ADF

In Oracle ADF, this accomplished with a search form. Complete information on how to create a search form is covered in Chapter 31, "Creating ADF Databound Search Forms." In particular, you may want to look at Section 31.3.1, "How to Set Search Form Properties on the View Criteria."

G.2.5 How to Implement an Updatable Multi-Record Table

In Oracle Forms, you may be used to creating tables where you can edit and insert many records at the same time. This can be slightly more complicated when using a JSF page in Oracle ADF, because the operations to edit an existing record and to create a new record are not the same.

Implementation of the task in Oracle ADF

In Oracle ADF, this is done using an input table. To create an input table, see Section 27.4, "Creating an Input Table."

G.2.6 How to Create a Popup List of Values

In Oracle Forms, it was simple to create a list of values (LOV) object and then associate that object with a field in a declarative manner. This LOV would display a popup window and provide the following capabilities:

- Selection of modal values
- Query area at the top of the LOV dialog
- Display of multiple columns
- Automatic reduction of LOV contents, possibly based on the contents of the field that launched the LOV
- Automatic selection of the list value when only one value matches the value in the field when the LOV function is invoked
- Validation of the field value based on the values cached by the LOV
- Automatic popup of the LOV if the field contents are not valid

Implementation of the task in Oracle ADF

To implement a popup list in Oracle ADF, you configure one of the view object's attributes to be of LOV type, and select **Input Text with List of Values** as the style for its UI hint. For a description of how to do this, see Section 5.12, "Working with List of Values (LOV) in View Object Attributes."

G.2.7 How to Implement a Dropdown List as a List of Values

In Oracle Forms, you could create a list of values (LOV) object and then associate that object with a field in a declarative manner. In Oracle ADF, you can implement an LOV (lookup-value) screen with a search item, usable for a lookup field with many possible values.

Implementation of the task in Oracle ADF

To implement a dropdown list in Oracle ADF, you configure one of the view object's attributes to be of LOV type, and select **Input Text with List of Values** as the style for its UI hint. For a description of how to do this, see Section 5.12, "Working with List of Values (LOV) in View Object Attributes."

G.2.8 How to Implement a Dropdown List with Values from Another Table

In Oracle Forms, you could create a list of values (LOV) object and then associate that object with a field in a declarative manner. In Oracle ADF, you can implement a dropdown list with string values from a different table. These string values populate the field with an id code that is valid input in the table that the screen is based on.

Implementation of the task in Oracle ADF

To implement a dropdown list of this type in Oracle ADF, you configure one of the view object's attributes to be of LOV type, and select **Choice List** as the style for its UI hint. For a description of how to do this, see Section 5.12, "Working with List of Values (LOV) in View Object Attributes."

G.2.9 How to Implement Immediate Locking

In Oracle ADF, you can lock a record in the database at the first moment it is obvious that the user is going to change a specific record.

Implementation of the task in Oracle ADF

Immediate row locking can be configured in ADF Business Components, although it is not the default and is typically not used in web application scenarios. For web applications, use the default configuration setting

`jbo.locking.mode=optimistic`. For more information, see Section 43.11.1, "How to Confirm That Applications Use Optimistic Locking."

G.2.10 How to Throw an Error When a Record Is Locked

When a record has been locked by a user, it's helpful to throw an error to let other users know that the record is not currently updatable.

Implementation of the task in Oracle ADF

Locking rows and throwing an exception if the row is already locked is built-in ADF Business Components functionality. There are a couple of different ways that you can handle the error message, depending on whether you want a static error message or a custom message with information about the current row.

- To throw a static message, register a custom message bundle in your data model project to substitute the default `RowAlreadyLockedException`'s error message with something more meaningful or user-friendly.
- To throw a message that contains information about the row, override the `lock()` method on the entity object, using a try/catch block to catch the `RowAlreadyLocked` exception. After you catch the exception, you can throw an error message that might contain more specific information about the current row.

H

Data Controls in Fusion Web Applications

This appendix provides a brief description of the various types of data controls available for Fusion web applications built with Oracle ADF. It also presents a brief comparison of how data access features are implemented for each type of data control.

This appendix contains the following sections:

- Section H.1, "Introduction to Data Controls"
- Section H.2, "Data Control Feature Implementation Comparison"
- Section H.3, "Data Control Objects"

H.1 Introduction to Data Controls

A data control is essentially a bridge that makes data from a source available to the user interface in a Fusion web application. You can use the objects in the data control to create databound user interface components.

The most commonly used types of data controls include the following:

- ADF Business Components Data Control

  This type of data control is generated by JDeveloper when you create an application module in your ADF Business Components application.

  For more information, see Section 13.4, "Using the Data Controls Panel."

- JavaBean Data Control

  This type of data control obtains the structure of the data from POJOs (plain, old Java objects).

  To create a JavaBean data control, right-click a Java class file (in the Application Navigator), and choose Create Data Control.

  For information about the data control objects available in a JavaBean data control, see Section H.3, "Data Control Objects."

- EJB Data Control

  The EJB data control is essentially the same as the JavaBean data control, except that it uses features inherent in the EJB architecture to obtain the structure of the data.

  You can create an EJB data control from the New Gallery. Expand the **Business Tier** node, select **Data Controls**, choose **EJB Data Control**, and click **OK**.

For information about the data control objects available in an EJB data control, see the *Oracle Fusion Middleware Java EE Developer's Guide for Oracle Application Development Framework*.

- URL Service Data Control

  A URL service data control lets you access and consume the data stream from a specified URL. This type of data control is not updateable.

  You can create a URL Service data control from the New Gallery. Expand the **Business Tier** node, select **Data Controls**, choose **URL Service Data Control**, and click **OK**.

  For information about the data control objects available in a URL Service data control, see Section 15.2, "Exposing URL Services with ADF Data Controls."

- Web Service Data Control

  A Web Service data control obtains the structure of the data from the WSDL for a web service.

  You can create a Web Service data control from the New Gallery. Expand the **Business Tier** node, select **Data Controls**, choose **Web Service Data Control**, and click **OK**.

  For more information, see Section 14.2, "Creating Web Service Data Controls."

- JMX Data Control

  A JMX data control obtains the structure of the JMX MBeans from an MBean Server.

  You can create a JMX data control from the New Gallery. Expand the **Business Tier** node, select **Data Controls**, choose **JMX Data Control**, and click **OK**. For more information about creating a JMX data control, see the online help for the Create JMX Data Control wizard.

  Before you can create a JMX data control, you must first have a JMX connection. For more information about JMX connections, see the online help for the Create JMX Connection dialog.

  For information about the data control objects available in a JMX data control, see Section H.3, "Data Control Objects."

- Placeholder Data Control

  A placeholder data control is a special type of data control that doesn't require a traditional data structure. As the name implies, it is a placeholder that can be used during UI development, and then replaced with the real data control when it becomes available.

  You can create a Placeholder data control from the New Gallery. Expand the **Business Tier** node, select **Data Controls**, choose **Placeholder Data Control**, and click **OK**.

  For more information, see Chapter 17, "Designing a Page Using Placeholder Data Controls."

H.2 Data Control Feature Implementation Comparison

The type of data control that you choose to use will impact how you implement data access features. Table H–1 provides a comparison of how you implement some commonly used data access features for each type of data control.

Table H–1 Comparison of Feature Implementation in Data Controls

	ADF Business Components Data Control	JavaBean Data Control	EJB Data Control	Web Services Data Control	URL Service Data Control	JMX Data Control	Placeholder Data Control
af:Query	Declarative	Declarative	Declarative	Implemented programmatically	Not available	Not available	Not available
af:quickQuery	Declarative	Declarative	Declarative	Implemented programmatically	Not available	Not available	Not available
af:inputComboListOfValues	Declarative	Declarative	Declarative	Implemented programmatically	Not available	Not available	Declarative
af:Calendar	Declarative	Implemented programmatically	Implemented programmatically	Implemented programmatically	Not available	Not available	Not available
af:Media	Declarative	Implemented programmatically	Implemented programmatically	Implemented programmatically	Not available	Not available	Not available

The features that are listed in the table as "implemented programmatically" can be implemented using the necessary Java classes required to implement a business model that can be used by the specific data-entry component. For more information, refer to the Javadoc for the appropriate classes.

H.3 Data Control Objects

In the Data Controls panel, each data control object is represented by an icon. Table H–2 describes what each icon represents, where it appears in the Data Controls panel hierarchy, and what components it can be used to create.

You can design a databound user interface by dragging an item from the Data Controls panel and dropping it on a page as a specific UI component.

The objects described in Table H–2 are applicable to the JavaBean data control, the EJB data control, and the JMX data control, unless otherwise noted. For information about the data control objects available in the other types of data controls, refer to the documentation for the desired type of data control, as listed in Section H.1, "Introduction to Data Controls."

Table H–2 Data Controls Panel Icons and Object Hierarchy for JavaBeans and EJBs

Icon	Name	Description	Used to Create...
	Data Control	Represents a data control. You cannot use the data control itself to create UI components, but you can use the child objects listed under the data control. There may be more than one data control, each representing a logical grouping of data functions. Typically, there is one data control for a given source (bean or EJB). However, you may have additional data controls that were created for other types of objects (for example, application modules or web services).	Serves as a container for the other objects. Not used to create anything.
	Create Method	Represents a built-in method that creates a new instance of an object in a data collection using the new Java constructor call. Create method icons are located in a node named after the data collection to which they belong. These data collection nodes are located in the **Constructors** node under the data control. The **Attributes** node, which appears as a child under a create method, contains all the attributes of the data collection. If the collection contains an attribute from another collection (called a *foreign key* in relational databases), that attribute is represented by an accessor return icon. In this case, the accessor returns a single object. This object is not available in the JMX data control.	Creation forms.
	Method	Represents a custom method on the data control that may accept parameters, perform some action or business logic, and return data or data collections. If the method is a `get` method of a `map` and returns a data collection, a method return icon appears as a child under it. If a method requires a parameter, a folder appears under the method, which lists the required parameters.	UI actions such as buttons or links.
	Method Return	Represents a data collection that is returned by a custom method. A method return appears as a child under the method that returns it. The objects that appear as children under a method return may be attributes of the collection, accessor returns that represent collections related to the parent collection, other methods that perform actions related to the parent collection, and operations that can be performed on the parent collection.	Forms, tables, trees, and range navigation components.

Table H–2 (Cont.) Data Controls Panel Icons and Object Hierarchy for JavaBeans and EJBs

Icon	Name	Description	Used to Create...
	Accessor Return	Represents an object returned by a bean-style accessor method on the business service. An accessor method is used when the objects returned are JavaBeans. Accessor returns appear as children under method returns, other accessor returns, or in the **Attributes** node under built-in create methods. Accessor returns are objects that are related to the current object in the parent collection. This relationship is usually based on a common unique attribute in both objects. For example, if a method returns a collection of users, an accessor return that is a child of that collection might be a collection of service requests that are assigned to a particular user. In ADF, the relationship between parent and child collections is called a master-detail relationship. For more information about master-detail objects, see Chapter 29, "Displaying Master-Detail Data.". Accessor returns can be either collections or single objects. For example, if a method returns a collection of service requests, one accessor return under that method might be a collection of service history details for the current service request, while another accessor return might be a single user assigned to the current service request. By default, when data controls are created from session beans over POJOs, the names of accessors that return collections end in `Collection` (for example, `OrderCollection`). The UI components available from the Data Controls panel context menu differ depending on whether the accessor return is a collection or a single object. The children under an accessor return may be attributes of the collection or object, other accessor returns, custom methods that return a value from the collection or object, and operations that can be performed on the collection or object. The accessor returns under a built-in create method are always a single object and never have any children.	For collections: Forms, tables, trees, range navigation components, and master-detail widgets. For single objects: Forms, master-detail widgets, and selection lists. For single objects under a constructor: selection lists only.
	Attribute	Represents a discrete data element in an object. Attributes appear as children under method returns or accessor returns.	Label, text field, and selection list components.
	Operation	Represents a built-in data control operation that performs actions on the parent object. If an operation requires a parameter, a folder appears under the method, which lists the required parameters. Data control operations are located in an **Operations** node under method returns or accessor returns and under the root data control node. The operations that are children of a particular method or accessor return operate on that return object only, while operations under the data control node operate on all the objects represented by the data control.	UI actions such as buttons or links.
	Parameter	Represents a parameter value that is declared by the method or operation under which it appears. Parameters appear in a folder under a method or operation.	Label, text, and selection list components.

Glossary

action binding

A binding for command components, such as buttons or links, to built-in or custom methods on the **data control**, or to built-in collection-level operations (such as, Create, Delete, Next, or Previous). An action binding object encapsulates the details of how to invoke a method and what parameters (if any) the method is expecting.

activity

A piece of work that is performed when an ADF Controller **task flow** runs, for example, a method call or view.

ADF

See **Oracle ADF**.

ADF binding filter

A servlet filter that ADF web applications use to preprocess any HTTP requests that may require access to the binding context.

ADF Business Components

A framework that simplifies the development, delivery, and customization of business applications for the Java 2 Platform. You use ADF Business Components to define associations between **entity object**s, **view object**s, and **application module**s to reflect the foreign keys present in the underlying tables.

ADF Controller layer

A mechanism that provides an enhanced navigation and state management model on top of JSF. This mechanism declaratively defines navigation using control flow rules.

ADF Faces Rich Client (RC)

A set of standard JSF components that include built-in AJAX functionality.

ADF Model layer

A service abstraction layer called the **data control**.

application module

A transactional component that UI clients use to work with application data. It defines an updatable data model and top-level procedures and functions (called *service methods*) related to a logical unit of work. This unit of work is related to an end-user task.

binding context

A container object that holds a list of available **data control**s and data binding objects. The `DataBindings.cpx` files define the binding context for the application.See also data control.

bounded task flow

A specialized form of ADF Controller task flow that has a single entry point and zero or more exit points. It contains its own set of private, **control flow** rules, activities, and managed beans. A bounded task flow allows reuse, parameters, transaction management, and reentry. When dropped on a page, it becomes a **region**.

control flow

An ADF Controller activity that enables navigation between other activities in an ADF task flow. The control flow links one **activity** to another in a **task flow**.

data control

XML configuration files that describe a service. At design time, visual tools like JDeveloper can leverage that metadata to UI component to be declaratively bound to an operation or data collection.

Data Controls panel

A panel in JDeveloper that lists all the **data control**s that have been created for the application's business services and exposes all the collections (row sets of data objects), methods, and built-in operations that are available for binding to UI components.

entity object

An object that represents a row in a database table and that simplifies modifying its data by handling all data manipulation language operations for you. Entity objects are **ADF Business Components** that provide the mapping to underlying data structures.

invoke action

An action that binds to a method that invokes the operations or methods defined in an action or a method action binding during any phase of the page lifecycle. See also **action binding**.

iterator binding

A binding to an iterator that iterates over view object collections. There is one iterator binding for each collection used on the page. All of the value bindings on the page must refer to an iterator binding in order for the component values to be populated with data at runtime.

Contrast with **variable iterator** and **method iterator**.

list of values (LOV)

See **LOV**.

LOV

Input components that allow a user to enter values by picking from a list that is generated by a query

MDS

An application server and Oracle relational database that keep metadata in these areas: a file-based repository data, dictionary tables (accessed by built-in functions) and a

metadata registry. One of the primary uses of MDS is to store customizations and persisted personalization for Oracle applications

method iterator

A binding to an iterator that iterates over the collections returned by custom methods in the data control. An iterator binding that is always related to a method **action binding** object. The method action binding encapsulates the details about how to invoke the method and what parameters the method is expecting. The method action binding is itself bound to the method iterator, which provides the data.

Oracle ADF

An end-to-end application framework that builds on Java Platform, Enterprise Edition standards and open-source technologies to simplify and accelerate implementing service-oriented applications.

Oracle Application Development Framework (Oracle ADF)

See **Oracle ADF**.

Oracle Metadata Services (MDS)

See **MDS**.

page definition file

A file that defines the binding objects that populate the data in UI components at runtime. For every page that has ADF bindings, there must be a corresponding page definition file that defines the binding objects used by that page.

region

An ADF Controller UI component whose content is based on a **task flow** definition. When first rendered, the region's content is that of the first view activity in a task flow. See also **activity**.

task flow

A set of ADF Controller activities, **control flow** rules and managed beans that interact to allow a user to complete a task.

unbounded task flow

A set of activities, ADF **control flow** rules, and managed beans that interact to allow a user to complete a task. An unbounded task flow has a single point of entry.

value binding

A binding used by ADF view UI components that display data. Value bindings range from the most basic variety that work with a simple text field to more sophisticated list and tree bindings that support the additional needs of list, table, and tree UI controls.

value iterator

An ADF Model iterator pointing to a collection that contains only one data object whose attributes are the binding container variable. Contrast with **method iterator** and **value iterator** iterator.

variable iterator

A binding to an iterator that exposes all the variables in the binding container to the other bindings. While there is an iterator binding for each collection, there is only one

variable iterator binding for all variables used on the page. Contrast with **method iterator** and **value iterator** iterator.

view accessor

An ADF Business Components object that points from an entity object attribute (or **view object**) to a destination view object or shared view instance in the same application workspace. The view accessor returns a row set that by default contains all the rows from the destination view object.

view object

An ADF Business Components object that represents a SQL query and simplifies working with its results. The SQL language is used to join, project, filter, sort, and aggregate data into the shape required by the end-user task being represented in the user interface.