



BEA WebLogic Integration™

Developing Adapters

Copyright

Copyright © 2005 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, subparagraph (d) of the Commercial Computer Software--Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA Systems DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks or Service Marks

BEA, BEA JRockit, BEA Liquid Data for WebLogic, BEA WebLogic Server, Built on BEA, Jolt, JoltBeans, SteelThread, Top End, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA AquaLogic, BEA AquaLogic Data Services Platform, BEA AquaLogic Enterprise Security, BEA AquaLogic Service Bus, BEA AquaLogic Service Registry, BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Manager, BEA MessageQ, BEA WebLogic Commerce Server, BEA WebLogic Enterprise, BEA WebLogic Enterprise Platform, BEA WebLogic Enterprise Security, BEA WebLogic Express, BEA WebLogic Integration, BEA WebLogic Java Adapter for Mainframe, BEA WebLogic JDriver, BEA WebLogic JRockit, BEA WebLogic Log Central, BEA WebLogic Personalization Server, BEA WebLogic Platform, BEA WebLogic Portal, BEA WebLogic Server Process Edition, BEA WebLogic WorkGroup Edition, BEA WebLogic Workshop, and Liquid Computing are trademarks of BEA Systems, Inc. BEA Mission Critical Support is a service mark of BEA Systems, Inc. All other company and product names may be the subject of intellectual property rights reserved by third parties.

All other trademarks are the property of their respective companies.

Contents

1. Introduction to the ADK

Section Objectives	1-1
What Is the ADK?	1-2
Requirements for Adapter Development	1-2
What the ADK Provides	1-3
What Are Adapters?	1-3
ResourceAdapter Interface	1-4
Service Connections	1-4
Event Connections	1-5
J2EE-Compliant Adapters Not Exclusive to WebLogic Integration	1-6
Design-Time GUI	1-6
Application Views	1-6
Packaging Framework	1-7
Before You Begin	1-7

2. Basic Development Concepts

Run Time Versus Design Time	2-1
Run-Time Framework	2-2
Design-Time Framework	2-2
Events and Services	2-3
What Are Events?	2-3
What Are Services?	2-3

How Adapters Use Logging	2-4
Logging Toolkit	2-4
Logging Framework.	2-4
Internationalization and Localization.	2-5
Adapter Logical Name	2-5
Where the Adapter Logical Name Is Used	2-6
Use of Adapter Logical Name in Adapter Deployment	2-6
Adapter Logical Name Used as an Organizing Principle	2-7
Adapter Logical Name Used as the Return Value for <code>getAdapterLogicalName</code>	2-8
Enterprise Archive (EAR) Files	2-8

3. Development Tools

Sample Adapter.	3-1
Why Use the Sample Adapter?	3-1
What Is In the Sample Adapter?	3-2
GenerateAdapterTemplate Utility.	3-3
ADK Javadoc	3-3
Ant-Based Build Process.	3-3
Why Use Ant?	3-4
XML Tools	3-4

4. Creating a Custom Development Environment

Adapter Setup Worksheet	4-1
Using GenerateAdapterTemplate	4-2
Step 1. Execute GenerateAdapterTemplate	4-2
Step 2. Rebuild the Tree.	4-5
Step 3. Test the Adapter	4-6
Step 4. Deploy the Adapter to WebLogic Integration	4-6

5. Using the Logging Toolkit

Logging Toolkit	5-2
Logging Configuration File	5-2
Logging Concepts	5-2
Message Categories	5-3
Message Priority	5-4
Assigning a Priority to a Category	5-5
Message Appenders	5-5
Message Layout	5-6
Putting the Components Together	5-7
How to Set Up Logging	5-7
Logging Framework Classes	5-9
com.bea.logging.ILogger	5-9
com.bea.logging.LogContext	5-10
com.bea.logging.LogManager	5-10
Internationalization and Localization of Log Messages	5-13
Saving Contextual Information in a Multithreaded Component	5-13

6. Developing a Service Adapter

J2EE-Compliant Adapters Not Specific to WebLogic Integration	6-2
Service Connections in a Run-Time Environment	6-2
Flow of Events	6-5
Step 1: Research Your Environment Requirements	6-6
Step 2: Configure the Development Environment	6-7
Step 2a: Set Up the Directory Structure	6-7
The web.xml and weblogic.xml Descriptor Files	6-9
Creating A Development Tree Within the Directory Structure	6-10
Step 2b: Assign the Adapter Logical Name	6-11

Step 2c: Set Up the Build Process	6-11
Manifest File	6-11
build.xml Components	6-13
Step 2d: Create the Message Bundle	6-23
Step 3: Implement the SPI	6-23
Basic SPI Implementation	6-24
ManagedConnectionFactory	6-24
Transaction Demarcation	6-24
ADK Implementations	6-25
AbstractManagedConnectionFactory Properties Required at Deployment . . .	6-31
ManagedConnection	6-32
ADK Implementation	6-33
ManagedConnectionMetaData	6-33
ADK Implementation	6-33
ConnectionEventListener	6-34
ADK Implementation	6-34
ConnectionManager	6-34
ADK Implementation	6-35
ConnectionRequestInfo	6-35
ADK Implementation	6-35
LocalTransaction	6-35
ADK Implementation	6-35
Step 4: Implement the CCI	6-36
How to Use This Section	6-36
Basic CCI Implementation	6-36
Connection	6-37
ADK Implementation	6-37
Interaction	6-38

ADK Implementation	6-38
Using XCCI to Implement the CCI	6-40
Services	6-40
DocumentRecord.	6-41
IDocument.	6-42
Proper Use of Namespaces in IDocument Instances.	6-44
ADK-Supplied XCCI Classes	6-44
XCCI Design Pattern.	6-46
Using NonXML J2EE-Compliant Adapters.	6-46
ConnectionFactory	6-47
ADK Implementation	6-47
ConnectionMetaData	6-48
ADK Implementation	6-48
ConnectionSpec	6-48
ADK Implementation	6-48
InteractionSpec.	6-49
ADK Implementation	6-50
LocalTransaction	6-50
Record.	6-51
ADK Implementation	6-52
ResourceAdapterMetaData.	6-52
ADK Implementation	6-52
Step 5: Enable Environment Variable Support (Optional)	6-52
Implementing ClientDataInteractionSpec	6-53
Extending DocumentInteractionSpecImpl	6-53
Step 6: Test the Adapter	6-54
Using the Test Harness.	6-54
Test Case Extensions Provided by the ADK	6-55

sample.spi.NonManagedScenarioTestCase	6-55
sample.event.OfflineEventGeneratorTestCase	6-55
sample.client.ApplicationViewClient	6-56
Step 7: Deploy the Adapter	6-56

7. Developing an Event Adapter

Introduction to Event Connections	7-1
Event Adapters in a Run-Time Environment	7-2
Flow of Events	7-4
Step 1: Define the Adapter	7-5
Step 2: Configure the Development Environment	7-6
Step 2a: Set Up the File Structure	7-7
Step 2b: Assign a Logical Name to the Adapter	7-7
Step 2c: Set Up the Build Process	7-7
Step 2d: Create the Message Bundle	7-8
Step 2e: Configure Logging	7-8
Create an Event Generation Logging Category	7-8
Step 3: Implement the Adapter	7-9
Step 3a: Create an Event Generator	7-9
How the Data Extraction Mechanism Is Implemented	7-10
How the Event Generator Is Implemented	7-12
Step 3b: Implement the Data Transformation Method	7-18
Step 3c: Implement Suspend/Resume Support	7-21
Step 3d: Implement Event Generator and EIS Status Reporting	7-22
Step 3e: Implement Event Generator Instance Support	7-25
Example: DBMS Sample Adapter	7-25
Why Implement Event Generator Instance Support?	7-27

Detecting and Responding to Changes in Event Generator Instance Specifications

7-27

Step 3f: Implement Environment Variable Support 7-29

Step 4: Test the Adapter 7-32

Step 5. Deploy the Adapter 7-32

8. Developing a Resource Adapter

Introduction to Resource Adapters 8-1

Resource Adapters in a Run-Time Environment. 8-2

Step 1: Define the Adapter 8-3

Step 2: Configure the Development Environment. 8-4

Step 2a: Set Up the File Structure 8-4

Step 2b: Assign a Logical Name to the Adapter 8-4

Step 2c: Set Up the Build Process 8-5

Step 2d: Create the Message Bundle 8-5

Step 2e: Configure Logging 8-6

 Create an Event Generation Logging Category 8-6

Step 3: Implement the Adapter 8-6

Step 3a: Create a Resource Adapter 8-7

Step 3b: Implement the Resource Adapter Properties 8-10

Step 4: Implement the Event and Service Adapters 8-11

Step 5. Deploy the Adapter 8-11

9. Developing a Design-Time GUI

Introduction to Design-Time Form Processing 9-2

Form Processing Classes 9-3

 RequestHandler 9-3

 ControllerServlet 9-4

 ActionResult 9-4

Word and Its Descendants	9-4
AbstractInputTagSupport and Its Descendants	9-5
Form Processing Sequence	9-5
Prerequisites	9-6
Steps in the Sequence	9-6
Design-Time Features	9-8
Java Server Pages	9-8
JSP Templates	9-9
ADK Library of JSP Tags	9-10
JSP Tag Attributes	9-11
The Application View	9-13
File Structure	9-13
Flow of Events	9-14
Step 1: Defining the Design-Time GUI Requirements	9-16
Step 2: Defining the Page Flow	9-17
Page 1: Logging In	9-17
Page 2: Managing Application Views	9-17
Page 3: Defining the New Application View	9-17
Page 4: Configuring the Connection	9-18
Page 5: Administering the Application View	9-18
Page 6: Adding an Event	9-19
Page 7: Adding a Service	9-19
Page 8: Testing an Application View	9-20
Publishing an Application View	9-20
Saving an Application View	9-21
Page 9: Summarizing an Application View	9-21
Step 3: Configuring the Development Environment	9-22
Step 3a: Create the Message Bundle	9-22

Step 3b: Configure the Environment to Update JSPs Without Restarting WebLogic Server	9-22
Step 4: Implement the Design-Time GUI	9-26
Extend AbstractDesignTimeRequestHandler.	9-26
Methods to Include	9-26
Step 4a. Supply the ManagedConnectionFactory Class.	9-27
Step 4b. Implement initServiceDescriptor().	9-27
Step 4c. Implement initEventDescriptor()	9-28
Step 5: Write the HTML Forms.	9-28
Step 5a: Create the confconn.jsp Form.	9-29
Including the ADK Tag Library	9-30
Posting the ControllerServlet	9-30
Displaying the Label for the Form Field.	9-31
Displaying the Text Field Size	9-31
Displaying a Submit Button on the Form	9-31
Implementing confconn()	9-32
Step 5b: Create the addevent.jsp form	9-32
Including the ADK Tag Library	9-33
Posting the ControllerServlet	9-33
Displaying the Label for the Form Field.	9-33
Displaying the Text Field Size	9-33
Displaying a Submit Button on the Form	9-33
Adding Additional Fields	9-34
Step 5c: Create the addservc.jsp form	9-34
Including the ADK Tag Library	9-34
Posting the ControllerServlet	9-34
Displaying the Label for the Form Field.	9-35
Displaying the Text Field Size	9-35

Displaying a Submit Button on the Form	9-35
Adding Additional Fields	9-35
Step 5d: Implement Editing Capability for Events and Services (optional)	9-36
Update the Adapter Properties File	9-36
Create edtservc.jsp and addservc.jsp	9-37
Implement Methods	9-38
Step 5e: Write the Web Application Deployment Descriptors.	9-39
Step 6. Implement the Look and Feel	9-40
Step 7. Implement Environment Variables	9-41
Step 7a – Displaying/Editing the Variable Set	9-42
Step 7b – Using the Variable Set	9-43
Step 8. Test the Sample Adapter Design-Time Interface	9-44
Files and Classes	9-44
Run the Tests	9-45

10. Deploying Adapters

Using Enterprise Archive (EAR) Files	10-1
Using Shared JAR Files in an EAR File	10-3
EAR File Deployment Descriptor	10-3
Deploying Adapters Using the WebLogic Server Administration Console	10-4
Adapter Auto-registration	10-5
Using a Naming Convention	10-5
Using a Text File	10-6
Editing Web Application Deployment Descriptors	10-6
Deployment Parameters	10-6
Editing the Deployment Descriptors	10-7
Deploying Adapters in a WebLogic Integration Cluster.	10-9
Redeploying Adapter Instances.	10-9

Creating an Adapter Not Specific to WebLogic Integration

Using This Section	A-1
Building the Adapter	A-2
Updating the Build Process	A-2

XML Toolkit

Toolkit Packages	B-1
IDocument	B-2
Schema Object Model (SOM)	B-3
How SOM Works	B-3
Creating the Schema	B-5
Resulting Schema	B-8
Validating an XML Document	B-11
How the Document Is Validated	B-12
Implementing isValid()	B-12
isValid() Sample Implementation	B-13

Adapter Setup Worksheet

Adapter Setup Worksheet	C-2
-----------------------------------	-----

Upgrading Adapters to WebLogic Integration 8.1

Learning to Develop Adapters Using the DBMS Sample Adapters

Introduction to the DBMS Sample Adapters	E-1
Connection Parameters for DBMS Sample Adapters	E-2
How the DBMS Sample Adapters Work	E-3
Before You Begin	E-3
Accessing the DBMS Sample Adapter	E-4

Tour of the DBMS Sample Adapter	E-4
How the DBMS Sample Adapters Were Developed	E-26
Step 1: Learn About the DBMS Sample Adapters.	E-26
Step 2: Define Your Environment	E-26
Step 3: Implement the Server Provider Interface Package.	E-29
ManagedConnectionFactoryImpl	E-29
ManagedConnectionImpl	E-30
ConnectionMetaDataImpl	E-31
LocalTransactionImpl	E-32
Step 4: Implement the Common Client Interface Package	E-33
ConnectionImpl	E-34
InteractionImpl	E-35
InteractionSpecImpl	E-36
Step 5: Implement the Event Package	E-37
EventGenerator	E-37
Step 6: Deploy the DBMS Sample Adapter.	E-38
Step 6a: Set Up Your Environment.	E-39
Step 6b: Update the ra.xml File	E-39
Step 6c: Create the RAR File	E-40
Step 6d: Build the JAR and EAR Files.	E-40
Step 6e: Create and Deploy the EAR File	E-40
Database-Specific Error Messages	E-42
How the DBMS Sample Adapter Design-Time GUI Was Developed	E-43
Step 1: Identify Requirements	E-43
Step 2: Identify Required Java Server Pages	E-44
Step 3: Create the Message Bundle	E-45
Step 4: Implement the Design-Time GUI	E-45
Step 5: Write Java Server Pages.	E-47

Use Custom JSP Tags	E-47
Save an Object's State	E-47
Write the WEB-INF/web.xml Deployment Descriptor.	E-47
Run-Time Considerations	E-48
Changing Event Connections	E-48
Administering a DBMS Sample Adapter Instance Used by Multiple Application Views E-49	
Multiple Event Generators Using the Same DBMS Instance	E-49

Index

Introduction to the ADK

This guide provides instructions for using the WebLogic Integration Adapter Development Kit (ADK). It shows you how to develop, test, and deploy event and service connections and the design-time user interface.

This section provides information about the following subjects:

- [What Is the ADK?](#)
- [What Are Adapters?](#)
- [Design-Time GUI](#)
- [Before You Begin](#)

Section Objectives

This section serves as an overview to using the ADK to develop event and service connections and a design-time GUI. You will learn:

- What adapters are and how they are used
- Prerequisites you must meet before beginning adapter development
- Terminology associated with adapter development

What Is the ADK?

The ADK is a set of tools for implementing the event and service protocols supported by BEA WebLogic Integration. These tools are organized in a collection of frameworks that support the development, testing, packaging, and distribution of resource adapters for WebLogic Integration. Specifically, the ADK includes frameworks for four purposes:

- Design-time operation
- Run-time operation
- Logging
- Packaging

Requirements for Adapter Development

The ADK addresses three requirements for adapter development:

- **Development environment structure:** The organization of a development project is important in any integrated development and debugging environment (IDDE). With a well structured development environment, you can begin coding an adapter immediately. The ADK provides an organized development environment, build process, intuitive class names and class hierarchy, and test methodology. By using the ADK, you avoid having to spend time designing and organizing a build process.

Because the ADK encompasses so many advanced technologies, an incremental development process (code a little, test a little) is the key to success. The ADK test process allows a developer to make a simple change and test it immediately.

- **Minimal exposure to peripheral implementation details:** Peripheral implementation details are sections of code that are needed to support the framework in which a robust software program runs.

For example, the J2EE Connector Architecture specification requires that the `javax.resource.cci.InteractionSpec` implementation class provide getter and setter methods that follow the JavaBeans design pattern. To support the JavaBeans design pattern, you must, in turn, support `PropertyChangeListeners` and `VetoableChangeListeners` in your implementation class. You do not want to have to study the JavaBeans specification to learn how to do this. Rather, you want to focus on implementing the enterprise information system (EIS)-specific details of the adapter. The ADK provides a base implementation for a majority of the peripheral implementation details of an adapter.

- A clear roadmap to success: Exit criteria enable you to answer the question: “How do I know my implementation is complete?” The ADK provides a clear methodology for developing an adapter. The methodology helps you organize your thoughts around a few key concepts: events, services, design-time operation, and run-time operation. Using this methodology, you can establish exit criteria that form a roadmap to implementation completion.

What the ADK Provides

The ADK provides:

- Run-time support for events and services
- An API for integrating an adapter’s user interface with the Application Integration Design Console

The ADK adds value by making it possible to make adapters an integral part of a single graphical console application that can be used by business users to construct integration solutions.

What Are Adapters?

Resource adapters—referred to in this document as *adapters*—are software components used to connect applications that were not originally designed to communicate with each other. For example, an adapter might be needed to enable an order entry system built by one company to communicate with a customer information system built by another.

By using the ADK, you can create two types of adapters:

- Service adapters, which accept requests and return responses
- Event adapters, which generate messages

You can also create an adapter that supports both services and events. All these types of adapters include an implementation of a top-level adapter interface

- `ResourceAdapter`—A top-level container for managing service and event adapters and connections.

You can also use the ADK to create J2EE-compliant adapters that are not specific to WebLogic Integration but that comply with the J2EE Connector Architecture Specification.

ResourceAdapter Interface

The `ResourceAdapter` interface is new in WebLogic Integration 8.1. It serves as a single object to unify both event and service handling. In prior WebLogic Integration releases, the event adapter and service adapter were treated as separate adapter components. Each was deployed and configured separately.

In this release, the event and service adapters are unified under the `ResourceAdapter` interface, and the adapter's implementation of that interface. It provides access to 0 or 1 event connection (created by the event adapter), and 0 or more service connections (created by the service adapter). The service connections are sometimes referred to as connection factories.

Configuration of event and service connections are now done under the umbrella of the `ResourceAdapter` interface. You will see the term `Resource Adapter` used in the Application Integration Design Console to represent a container of event and service connections.

Each adapter must implement `ResourceAdapter` in a concrete class and package that class into the adapter's EAR module in order to operate within WebLogic Integration. The ADK provides an abstract base implementation of the `ResourceAdapter` class in `com.bea.adapter.spi.AbstractWLIResourceAdapter`.

To learn how to develop a `ResourceAdapter` implementation, see [Chapter 8, “Developing a Resource Adapter”](#).

Service Connections

Service connections receive XML request documents from clients and invoke specific functions in the underlying enterprise information system (EIS). They are consumers of messages; they may or may not provide a response.

A service may be invoked in either of two ways: asynchronously or synchronously. When a service is invoked asynchronously, the client application issues a service request and then proceeds with processing without waiting for the response. When a service is invoked synchronously, the client waits for the response before proceeding with processing. BEA WebLogic Integration supports both types of service connection invocations, so you are not required to provide this functionality.

Service connections perform the following four functions:

- Receive service requests from an external client.

- Transform the XML format of a request document into an EIS-specific format. The request document conforms to the request XML schema for the service. The request XML schema is based on metadata in the EIS.
- Invoke the underlying function in the EIS and wait for its response.
- Transform the response from the EIS-specific data format to an XML document that conforms to the response XML schema for the service. The response XML schema is based on metadata in the EIS.

As with events, the ADK implements the aspects of these four functions that are common to all service connections.

To learn how to develop a service connection, see [Chapter 6, “Developing a Service Adapter.”](#)

Event Connections

Event connections are designed to propagate information from an EIS to WebLogic Server; they can be described as publishers of information.

There are two basic types of event connections: *in-process* and *out-of-process*. In-process event connections execute within the same process as the EIS. Out-of-process adapters execute in a separate process. In-process and out-of-process event connections differ only in terms of how they accomplish the data extraction process.

Event connections running in a WebLogic Integration environment perform the following three functions:

- Respond to events deemed to be of interest to some external party that occur inside the running EIS and extract data about such events from the EIS.
- Transform event data from an EIS-specific format to an XML document that conforms to the XML schema for the event. The XML schema is based on metadata in the EIS.
- Propagate each event to an event message endpoint provided by the application server.

The ADK implements the aspects of these three functions that are common to all event connections. Consequently, you can focus on the EIS-specific aspects of your adapter. This concept is the same as the concept behind Enterprise Java Beans (EJB): the container provides system-level services for EJB developers so they can focus on implementing business application logic.

To learn how to develop an event connection, see [Chapter 7, “Developing an Event Adapter.”](#)

J2EE-Compliant Adapters Not Exclusive to WebLogic Integration

These adapters are not designed for WebLogic Integration exclusively; they can be plugged into any application server that supports the J2EE Connector Architecture specification. These adapters can be developed by making minor modifications to the procedures given for developing a service connection. To learn how to develop an adapter that is not specific to WebLogic Integration, see [Appendix A, “Creating an Adapter Not Specific to WebLogic Integration.”](#)

Design-Time GUI

Along with event and service connections, the ADK’s design-time framework provides tools you can use to build the Web-based GUI that adapter users need to define, deploy, and test application views (see [“Application Views”](#)). Although each adapter has EIS-specific functionality, all adapters require a GUI for deploying application views. The design-time framework minimizes the effort required to create and deploy these interfaces, primarily by using two components:

- A Web application component and a web request handler base class (`AbstractDesignTimeRequestHandler`) that allows you to build an HTML-based GUI by using Java Server Pages (JSP). This component is augmented by tools such as the JSP templates, the JSP tag library, and the JavaScript library.
- A deployment helper component, called `DesignTimeHelper`, that provides a simple API for deploying, undeploying, and editing application views on WebLogic Server.

To learn how to develop a design-time GUI, see [Chapter 9, “Developing a Design-Time GUI.”](#)

Application Views

While an adapter represents a system-level interface to *all the functionality* in an application, an *application view* represents a business-level interface to a *particular set of functions* in the application.

An application view is configured for a single business purpose and contains only services related to that purpose. These services require only business-relevant data to be specified in the request document; they return only business-relevant data in the response document. Without user intervention, the application view combines this business-relevant data with stored metadata necessary for the adapter. The adapter takes both the business-relevant data and the stored metadata and executes a system-level function on the application.

The application view also represents both the events and services that support the specified business purpose. As a result, the business user can perform all communication with an application through the application view. Such bidirectional communication is supported by two adapter components: the event connection and the service connection. The application view abstracts this fact from users and presents them with a unified business interface to the application.

For more information about application views, see [“Introduction to Using Application Integration”](#) in *Using Application Integration*.

Packaging Framework

The ADK packaging framework is a tool set for packaging an adapter for delivery to a customer. Ideally, all adapters are installed, configured, and uninstalled in the same way on WebLogic Server. All service connections must be J2EE compliant. The packaging framework simplifies the creation of a J2EE adapter archive (RAR) file, a Web application archive (WAR) file, an enterprise archive (EAR) file, and a WebLogic Integration design environment archive.

Before You Begin

Before beginning your development work, make sure WebLogic Integration is installed on your computer. For more information, see [Installing BEA WebLogic Platform](#) and the [BEA WebLogic Integration Release Notes](#).

Introduction to the ADK

Basic Development Concepts

This section describes some basic concepts with which you should become familiar before attempting to develop an adapter or design-time graphical user interface (GUI). Specifically, it provides information about the following subjects:

- [Run Time Versus Design Time](#)
- [Events and Services](#)
- [How Adapters Use Logging](#)
- [Adapter Logical Name](#)
- [Enterprise Archive \(EAR\) Files](#)

Run Time Versus Design Time

The term *adapter activity* encompasses both run-time and design-time activity. Run-time activity is the execution of an adapter's processes. Design-time activity, performed by an adapter user, includes the creation, deployment, and testing of an application view.

Run-time and design-time activity are supported by ADK run-time and design-time frameworks, respectively. The run-time framework comprises tools for developing adapters, while the design-time framework includes tools for designing Web-based user interfaces. Both types of activity are discussed in greater detail in the following sections.

Run-Time Framework

The run-time framework is a set of tools you can use to develop event and service connections. To support event connection development, the run-time framework provides a basic, extensible event generator. For service connection development, the run-time framework provides a complete J2EE-compliant adapter.

The classes supplied by the run-time framework provide the following benefits:

- They allow you to focus on EIS details rather than J2EE details.
- They minimize the effort needed to use the ADK logging framework.
- They simplify the J2EE Connector Architecture.
- They minimize redundant code used in multiple adapters.

In addition, the run-time framework provides abstract base classes to help you implement an event generator that can leverage the event support provided by the ADK environment.

A key component of the run-time framework is the run-time engine, which hosts the adapter component responsible for handling service invocations and manages the following WebLogic Server features:

- Physical connections to the EIS
- Login authentication
- Transaction management

All three features comply with the J2EE Connector Architecture standard.

Design-Time Framework

The design-time framework provides tools for building the Web-based GUI that adapter users need to define, deploy, and test their application views. Although each adapter has EIS-specific functionality, all adapters require a GUI for deploying application views. This framework provides two tools that minimize the effort required to create and deploy such a GUI:

- A Web application component that allows you to build an HTML-based GUI by using JSPs. This component is augmented by tools such as the JSP templates, the tag library, and the JavaScript library.
- A deployment helper component that provides a simple API for deploying, undeploying, and editing application views on WebLogic Server.

The design-time interface for each adapter is a J2EE Web application that is bundled as a WAR file. A Web application is a bundle of `.jsp` files, `.html` files, image files, and so on. The Web application descriptor is `web.xml`. The descriptor provides the J2EE Web container with instructions for deploying and initializing the Web application.

Every Web application has a context that is specified during deployment. The context identifies resources associated with the Web application under the Web container's document root.

Events and Services

With the ADK you can create both event connections and service connections. Within the ADK architecture, services and events are defined as self-describing objects (for which a name indicates a business function) that use XML schema to define input and output.

What Are Events?

An event is an XML document published by an application view when an occurrence of interest takes place within an EIS. Clients that want to be notified of events request such notification by registering with an application view. The application view then acts as a broker between the target application and the client. When a client has subscribed to events published by an application view, the application view notifies the client whenever an event of interest occurs in the target application. When an event subscriber is notified that an event of interest has occurred, it is passed an XML document that describes the event. Application views that publish events can also provide clients with the XML schema for publishable events.

Note: An application view represents a business-level interface to a specific function in an application. For more information about this feature, see [Introducing Application Integration](#).

What Are Services?

A service is a business operation in an application that is exposed by an application view. It serves as a request/response mechanism: when an application receives a request to invoke a business service, the application view invokes the service in the target application and then returns (or, responds with) an XML document that describes the results.

To define a service, you must define input requirements, output expectations, and an interaction specification.

A service request and response consists of:

- An interaction specification, containing static *metadata* about the request. An example of this static metadata would be an SQL statement for a DBMS adapter. The end user of the service never sees the metadata.
- User input, which represents business information the user has obtained and provided as the request document for this service. This request document can contain variable information that relates to the metadata in the interaction specification. For example, in a DBMS adapter based service, the SQL statement is provided in the interaction specification, and the value of any variables in that SQL statement are provided in the request document.
- The response from the service conforms to the output expectations described at design-time for the service.

How Adapters Use Logging

Logging is an essential feature of an adapter. Most adapters are used to integrate different applications and do not interact with end-users while processing data. Unlike a front-end component, when an adapter encounters an error or warning condition, it cannot stop processing and wait for an end-user to respond.

Moreover, many business applications connected by adapters are mission-critical. For example, an adapter might be required to keep an audit report of every transaction with an EIS. Consequently, adapter components should provide both accurate logging and auditing information. The ADK's logging framework is designed to accommodate both logging and auditing.

Logging Toolkit

The ADK provides a toolkit that allows you to log localized messages to multiple output destinations. The logging toolkit leverages the work of the Apache Log4j open source project.

The logging toolkit wraps the critical classes in Log4j to provide added functionality when you are building J2EE-compliant adapters. The toolkit is provided in the `logtoolkit.jar` file.

For information about using the logging toolkit, see [Chapter 5, “Using the Logging Toolkit.”](#)

Logging Framework

With the ADK, logging of adapter activity is accomplished by implementing the logging framework. This framework gives you the ability to log internationalized and localized messages

to multiple output destinations. It provides a range of configuration parameters you can use to tailor message category, priority, format, and destination.

The logging framework uses a categorical hierarchy to allow inheritance of logging configuration by all packages and classes within an adapter. The framework allows parameters to be modified easily during run time.

Internationalization and Localization

The logging framework allows you to internationalize log messages. Internationalized applications are easy to tailor to the idioms and languages of end-users around the world without rewriting the code. Localization is the process of adapting software for a specific region or language by adding locale-specific components and text. The logging framework uses the internationalization and localization facilities provided by the Java platform.

Adapter Logical Name

Every adapter must have an *adapter logical name*: a unique identifier that represents an individual adapter and serves as the organizing principle for all adapters. An adapter logical name is the means by which both an individual adapter and the following related items are identified:

- Message bundle
- Logging configuration
- Log categories

An adapter logical name is formed by combining the vendor name, the type of EIS connected to the adapter, and the version number of the EIS. By convention, this information is expressed as *vendor_EIS-type_EIS-version*. For example, in the adapter logical name

`BEA_WLS_SAMPLE_ADK:`

- `BEA_WLS` is the vendor and product
- `SAMPLE` is the EIS type
- `ADK` is the EIS version

You may use another format for this information, if you prefer, as long as you include the required data.

Where the Adapter Logical Name Is Used

The adapter logical name is used with adapters in the following ways:

- It is used during adapter deployment as part of the WAR, RAR, JAR, and EAR filenames.
- It is used as an organizing principle, as described in [“Adapter Logical Name Used as an Organizing Principle” on page 2-7](#).
- It is used as a return value to the abstract method `getAdapterLogicalName()` in `com.bea.adapter.web`, as described in [“Adapter Logical Name Used as the Return Value for `getAdapterLogicalName`” on page 2-8](#).

Use of Adapter Logical Name in Adapter Deployment

To assign an adapter logical name, specify it as the value of the `Name` attribute of the `<Application>` element that contains the `<ConnectorComponent>` element. This value is the key used by WebLogic Integration to associate an application view with a deployed resource adapter, as shown for a sample adapter in [Listing 2-1](#).

Listing 2-1 Name Attribute of the ConnectorComponent Element

```
<Application Deployed="true" Name="BEA_WLS_DBMS_ADK"  
  Path="<WLI_HOME>/adapters/dbms/lib/BEA_WLS_DBMS_ADK.ear"  
  TwoPhase="true">  
  <ConnectorComponent Name="BEA_WLS_DBMS_ADK" Targets="myserver"  
    URI="BEA_WLS_DBMS_ADK.rar" />  
  <WebAppComponent Name="DbmsEventRouter" Targets="myserver"  
    URI="BEA_WLS_DBMS_ADK_EventRouter.war" />  
  <WebAppComponent Name="BEA_WLS_DBMS_ADK_Web" Targets="myserver"  
    URI="BEA_WLS_DBMS_ADK_Web.war" />  
</Application>
```

Note: The use of the adapter logical name as the name of the RAR file is an optional convention; such naming is not required in the URI attribute.

When an application view is deployed, it is associated with a J2EE Connector Architecture CCI connection factory deployment. For example, if a user deploys the `abc.xyz` application view,

WebLogic Integration deploys a new `ConnectionFactory` and binds it to the following JNDI location:

```
com.bea.wlai.connectionFactories.abc.xyz.connectionFactoryInstance
```

Adapter Logical Name Used as an Organizing Principle

[Table 2-1](#) lists the types of functionality that use the adapter logical name as an organizing principle.

Table 2-1 How an Adapter Logical Name Is Used as an Organizing Principle

In this area of functionality . . .	Adapter logical names are used as follows . . .
Logging	<p>The adapter logical name is used as the base log category name for all log messages that are unique to the adapter. Consequently, the adapter logical name is passed as the value for the <code>RootLogContext</code> parameters in the following XML documents:</p> <ul style="list-style-type: none"> • <code>WLI_HOME/adapters/ADAPTER/src/rar/META-INF/ra.xml</code> • <code>WLI_HOME/adapters/ADAPTER/src/rar/META-INF/weblogic-ra.xml</code> • <code>WLI_HOME/adapters/ADAPTER/src/war/WEB-INF/web.xml</code> <p>In these pathnames, <code>ADAPTER</code> represents the name of your adapter. For example:</p> <pre>WLI_HOME/adapters/dbms/src/war/WEB-INF/web.xml</pre> <p>In addition, the adapter logical name is used as the base for the name of the Log4J configuration file for the adapter; the name is completed by the addition of the <code>.xml</code> suffix.</p> <p><code>.xml</code> is appended to the name. For example, the Log4J configuration file for the sample adapter is <code>BEA_WLS_SAMPLE_ADK.xml</code>.</p>
Localization	<p>The logical name of the adapter is used as the base name for message bundles for the adapter. For example, the default message bundle for the sample adapter is <code>BEA_WLS_SAMPLE_ADK.properties</code>. Consequently, the adapter logical name is passed as the value for the <code>MessageBundleBase</code> parameters in the following XML documents:</p> <ul style="list-style-type: none"> • <code>WLI_HOME/adapters/ADAPTER/src/rar/META-INF/ra.xml</code> • <code>WLI_HOME/adapters/ADAPTER/src/rar/META-INF/weblogic-ra.xml</code> • <code>WLI_HOME/adapters/ADAPTER/src/war/WEB-INF/web.xml</code> <p>In these pathnames, the value of <code>ADAPTER</code> is the name of your adapter. For example:</p> <pre>WLI_HOME/adapters/dbms/src/war/WEB-INF/web.xml</pre>

Adapter Logical Name Used as the Return Value for `getAdapterLogicalName`

Lastly, the adapter logical name is used as the return value to the abstract method `getAdapterLogicalName()` on the `com.bea.adapter.web`.

`AbstractDesignTimeRequestHandler`. This return value is used during the deployment of application views as the value of the `RootLogContext` for a connection factory.

Enterprise Archive (EAR) Files

The ADK uses Enterprise Archive files, or EAR files, for deploying adapters. A single `.ear` file contains the WAR and RAR files necessary to deploy an adapter. An example of an EAR file is shown in [Listing 2-2](#).

Listing 2-2 EAR File Structure

```
adapter.ear
  META-INF
    application.xml
  sharedJar.jar
  adapter.jar
  adapter.rar
    META-INF
      ra.xml
      weblogic-ra.xml
      MANIFEST.MF
  designtime.war
    WEB-INF
      web.xml
    META-INF
      MANIFEST.MF
```

The EAR file for the sample adapter is shown in [Listing 2-3](#).

Listing 2-3 Sample Adapter EAR File

```
sample.ear
  META-INF
    application.xml
  shared.jar (shared .jar between .war and .rar)
  BEA_WLS_SAMPLE_ADK.war (Web application with
    META-INF/MANIFEST.MF entry Class-Path: shared.jar
  BEA_WLS_SAMPLE_ADK.rar (Resource Adapter
    META-INF/MANIFEST.MF entry Class-Path: shared.jar
```

Notice that neither the RAR nor WAR files include any shared JAR files; rather, both refer to the shared JAR files located in the root directory of the EAR file.

For more information about using EAR files to deploy adapters, see [Chapter 10, “Deploying Adapters.”](#)

Basic Development Concepts

Development Tools

The ADK provides a set of robust tools to assist you in developing adapters and the design-time GUI. This section describes these tools. Specifically, it includes information about the following subjects:

- [Sample Adapter](#)
- [GenerateAdapterTemplate Utility](#)
- [ADK Javadoc](#)
- [Ant-Based Build Process](#)
- [XML Tools](#)

Sample Adapter

To help you start building an adapter, the ADK provides a sample adapter with code examples that are not specific to EIS. Do not confuse this sample adapter with the DBMS sample adapters that are also provided by WebLogic Integration; the DBMS sample adapters are documented in [Appendix E, “Learning to Develop Adapters Using the DBMS Sample Adapters.”](#) You can find the DBMS sample adapters in `WLI_HOME/adapters/dbms`.

Why Use the Sample Adapter?

The purpose of the sample adapter is to free you from much of the coding necessary to build an adapter. It provides concrete implementations of key abstract classes that require customization only to meet the requirements of the EIS you are using. In addition, the ADK provides

`GenerateAdapterTemplate`, a utility with which you can quickly clone the sample adapter development tree for use by the adapter you are developing. See [“GenerateAdapterTemplate Utility” on page 3-3](#).

What Is In the Sample Adapter?

The sample adapter contains:

`sample.cci.ConnectionImpl`

A concrete implementation of the `Connection` interface that represents an application-level handle used by a client to access the underlying physical connection.

`sample.cci.InteractionImpl`

A class that demonstrates how to implement a design pattern using the `DesignTimeInteractionSpecImpl` class.

`sample.cci.InteractionSpecImpl`

An interface that provides a base implementation that you can extend by using getter and setter methods for the standard interaction properties.

`sample.client.ApplicationViewClient`

A class that demonstrates how to invoke a service and listen for an event on an application view.

`sample.eis.EIS`

`sample.eis.EISConnection`

`sample.eis.EISEvent`

`sample.eis.EISListener`

Classes that represent, for demonstration purposes, a simple EIS.

`sample.event.EventGenerator`

A concrete extension to `AbstractPullEventGenerator` that shows how to extend the ADK base class to construct an event generator.

`sample.event.OfflineEventGeneratorTestCase`

A class you can use to test the inner workings of your event generator outside WebLogic Server.

`sample.spi.ManagedConnectionFactoryImpl`

A concrete extension to `AbstractManagedConnectionFactory` that you can customize for a specific EIS.

`sample.spi.ManagedConnectionImpl`

A concrete extension to `AbstractManagedConnection` that you can customize for a specific EIS.

`sample.spi.ConnectionMetaDataImpl`

A concrete extension to `AbstractConnectionMetaData` that you can customize for a specific EIS.

`sample.spi.NonManagedScenarioTestCase`

A class you can use to test your SPI and CCI classes in an unmanaged scenario.

`sample.spi.ResourceAdapterImpl`

A concrete extension to `AbstractWLIResourceAdapter` that you can customize for a specific EIS.

`sample.web.DesignTimeRequestHandler`

A concrete extension to `AbstractDesignTimeRequestHandler` that shows how to add an event or service at design time.

Note: For details about the classes extended by those in the sample adapter, see the ADK Javadocs at the following URL:

<http://e-docs.bea.com/wli/docs81/javadoc/adk/>

GenerateAdapterTemplate Utility

To facilitate use of the sample adapter, the ADK provides `GenerateAdapterTemplate`, a command-line utility you can use to create a new adapter development tree by cloning the sample tree. For complete instructions on using this tool, see [Chapter 4, “Creating a Custom Development Environment.”](#)

ADK Javadoc

ADK classes, interfaces, methods, and constructors are defined in the development kit’s Javadoc. The ADK Javadoc is located at the following URL:

<http://e-docs.bea.com/wli/docs81/javadoc/adk/>

The Javadoc for the ADK is not specific to a single adapter. For additional information on specific sample adapter, refer to comments in the adapter source code provided in `WLI_HOME/adapters/ADAPTER/src` where `ADAPTER` is the specific adapter directory.

Ant-Based Build Process

The ADK employs a build process based on Ant, a 100% pure Java-based build tool. For the ADK, Ant does the following:

- Creates a Java archive (JAR) file for the adapter.

- Creates a WAR file for an adapter’s Web application.
- Creates a RAR file for a J2EE-compliant adapter.
- Bundles the other components in this list into an EAR file for deployment.

Why Use Ant?

Traditionally, build tools are shell-based. Like shell commands, they evaluate a set of dependencies and then execute various tasks. While the advantage of such tools is that it is simple to extend them by using or writing any program for your operating system (OS), the disadvantage is that you are limited to that OS.

Ant is preferable to shell-based make tools for the following reasons:

- It is extended with Java classes instead of shell-based commands.
- The configuration files are based on XML instead of shell commands: they invoke a target tree in which various tasks get executed. Each task is run by an object that implements a particular task interface. While this arrangement removes some of the expressive power inherent in the ability to construct a shell command, it makes your application portable across platforms.
- Ant allows you to execute various OS-specific shell commands.

For complete instructions for setting up Ant, see [“Step 2c: Set Up the Build Process” on page 6-11.](#)

XML Tools

The ADK includes the XML Toolkit, a set of two XML development tools that are considered part of the metadata support layer for the design-time framework:

- XML Schema API—Based on the Schema Object Model (SOM), this API is used to build XML schemas programmatically. The SOM is a set of tools that enables you to extract many common details, such as the syntactical complexities of XML schema operations, so you can focus on the more fundamental aspects of a schema.
- XML Document API—Based on `IDocument`, this API provides the x-path interface to a document object model (DOM) document.

For instructions on using these tools, see [Appendix B, “XML Toolkit.”](#)

WebLogic Integration provides Javadoc for both APIs:

- For SOM Javadoc, go to the WebLogic Integration Javadoc at <http://e-docs.bea.com/wli/docs81/javadoc/> and select the `com.bea.schema` package
- For IDocument Javadoc, go to the WebLogic Integration Javadoc at <http://e-docs.bea.com/wli/docs81/javadoc/> and select the `com.bea.document` package

Development Tools

Creating a Custom Development Environment

Warning: We strongly recommend that you *do not* alter the sample adapter directly. Instead, use the `GenerateAdapterTemplate` utility described in this chapter to make a copy of the adapter, and then make any changes you want to your copy. Modifying the sample adapter itself (or trying to create a copy of it without using `GenerateAdapterTemplate`) might result in unexpected and unsupported behavior.

To facilitate the use of the sample adapter (see “[Sample Adapter](#)” on page 3-1), the ADK provides `GenerateAdapterTemplate`, a command-line utility you can use to create a new adapter development tree by cloning the sample tree.

This section provides information about the following subjects:

- [Adapter Setup Worksheet](#)
- [Using GenerateAdapterTemplate](#)

Adapter Setup Worksheet

The adapter setup worksheet is a questionnaire designed to help you identify and collect critical information about the adapter you are developing. You can find this questionnaire in [Appendix C](#), “[Adapter Setup Worksheet](#).”

This worksheet is a set of 20 questions that can help you identify critical adapter information, such as EIS type, vendor, and version, locale and national language of the deployment, the adapter logical name, and whether or not the adapter supports services. When you run

GenerateAdapterTemplate, you are prompted to enter this information. When the information is processed, a custom development tree for your adapter is created.

Using GenerateAdapterTemplate

This section explains how to use GenerateAdapterTemplate. You must perform the following steps:

- [Step 1. Execute GenerateAdapterTemplate](#)
- [Step 2. Rebuild the Tree](#)
- [Step 3. Test the Adapter](#)
- [Step 4. Deploy the Adapter to WebLogic Integration](#)

Step 1. Execute GenerateAdapterTemplate

To use this tool, do the following:

1. Open a command line from the `WLI_HOME/adapters/utils` directory and execute one the following commands:

- For Windows NT: `GenerateAdapterTemplate.cmd`
- For UNIX: `GenerateAdapterTemplate.sh`

The system responds:

```
*****  
Welcome! This program helps you generate a new adapter development tree  
by cloning the ADK's sample adapter development tree.  
Do you wish to continue? (yes or no); default='yes':
```

For Windows systems, proceed to step 2. After completing step 2, the system displays the prompt shown in step 3.

For non-Windows systems, proceed to step 3.

2. Select `yes` by pressing `Enter`. For Windows systems only, select your console's codepage value from the following codepage list:

```
Cp437 - United States  
Cp850 - Multilingual (Latin I)  
Cp852 - Slavic (Latin II)  
Cp855 - Cyrillic (Russian)
```

```

Cp857 - Turkish
Cp860 - Portuguese
Cp861 - Icelandic
Cp863 - Canadian-French
Cp865 - Nordic
Cp866 - Russian
Cp869 - Modern Greek
MS932 - Japanese
Enter your console's codepage; default='Cp437':

```

If you do not know your codepage, launch a new DOS console prompt and enter `chcp` at your console prompt. Depending on the Windows version, this command displays your console's codepage value.

3. Select yes by pressing Enter.

The system responds:

```

Please choose a name for the root directory of your adapter development
tree (NOTE: this directory will be created under WLI_HOME/adapters):

```

4. Enter a unique, easy-to-remember directory name (*dir_name*) and press Enter.

The system responds:

```

created directory WLI_HOME/adapters/dir_name

```

```

Enter the EIS type for your adapter:

```

In the pathname specified in the system output, *dir_name* is the name of the new directory.

Note: If you enter the name of an existing directory, the system responds:

```

WLI_HOME/adapters/dir_name already exists, please choose
a new directory that does not already exist!

```

```

Please choose a name for the root directory of your adapter
development tree:

```

5. Enter an identifier for the EIS type to which your adapter will connect. Press Enter.

The system responds:

```

Enter a short description for your adapter:

```

6. Enter a short, meaningful description of the adapter you are about to develop and press Enter.

The system responds:

Creating a Custom Development Environment

Enter the major version number for your adapter; default='1':

7. Either press Enter to accept the default, or enter the appropriate version number and then press Enter.

The system responds:

Enter the minor version number for your adapter; default='0':

8. Either press Enter to accept the default, or type the appropriate minor version number and then press Enter.

The system responds:

Enter the vendor name for your adapter:

9. Enter the vendor's name and press Enter.

The system responds:

Enter an adapter logical name; default='default_name':

10. Either press Enter to accept the default or type the adapter logical name you want to use. Press Enter. The default adapter logical name ('default_name') is based on the format recommended for WebLogic Integration:

vendor_name_EIS-type_version-number.

The system responds:

Enter the Java package base name for your adapter
(e.g. sample adapter's is sample): *Java package base name*

11. Enter the base name of the Java package, in package format, and press Enter. A name in package format consists of the following strings, separated by dots:

- The extension used in the URL for your organization's Web site (such as .com, .org, or .edu)
- The name of your company
- Additional adapter identifiers. For example: *com.your_co.adapter.EIS.*

The system responds:

The following information will be used to generate your new adapter development environment:

EIS Type = 'SAP R/3'

Description = '*description*'

Major Version = '1'

Minor Version = '0'

Vendor = '*vendor_name*'

```
Adapter Logical Name = 'adapter_logical_name'
Java Package Base = 'com.java.package.base'
Are you satisfied with these values? (enter yes or no or q to quit);
default='yes':
```

12. To confirm the information, press Enter.

The system responds by displaying the appropriate build information.

Note: If you enter `no`, you are routed back to step 4. If you enter `q` (quit), the application terminates.

Step 2. Rebuild the Tree

After completing the clone process, go to the new directory and use Ant, the ADK's build tool, to rebuild the entire tree. For more information about Ant, see [“Ant-Based Build Process” on page 3-3](#).

Note: You must explicitly enter the file extension when running Ant commands on UNIX systems. For example, enter `ant.sh release`. Entering `ant release` on UNIX systems does not locate `ant.sh`.

To rebuild the tree by using Ant, do the following:

1. Edit `antEnv.cmd` (Windows) or `antEnv.sh` (UNIX) in `WLI_HOME/adapters/ADAPTER/utils`.
2. Set the following variables to valid paths:
 - `BEA_HOME` - The top-level directory for your BEA products, such as `c:/bea`.
 - `WLI_HOME` - The location of your WebLogic Integration directory.
 - `WL_HOME` - The location of your WebLogic Server directory.
 - `JAVA_HOME` - The location of your Java Development Kit.
 - `ANT_HOME` - The location of your Ant directory, typically `WLI_HOME/adapters/utils`.

Note: The installer performs this step for you, but you should be aware that these settings control the Ant process.

On a UNIX system, execute permission for all must be set for the Ant file in `WLI_HOME/adapters/utils`. To add execute permission, enter the following command:

```
chmod u+x ant.sh
```

3. Execute `antEnv.cmd` (Windows) or `antEnv.sh` (UNIX) from the command line to set the necessary environment variables for your shell.
4. Execute `ant.cmd release` (Windows) or `ant.sh release` (UNIX) from the `WLI_HOME/adapters/ADAPTER/project` directory to build the adapter. (Replace `ADAPTER` with the name of the new adapter development root.)

When you execute `ant release`, Javadoc is generated for the adapter. You can view the Javadoc by going to:

```
WLI_HOME/adapters/ADAPTER/docs/overview.html
```

This file provides environment-specific instructions for deploying your adapter in a WebLogic Integration environment. Specifically, it provides `config.xml` entries and replacements for the path already created. In addition, the file provides mapping information.

To facilitate adapter deployment, as described in [“Step 4. Deploy the Adapter to WebLogic Integration” on page 4-6](#), you can copy the contents of `overview.html` directly into `config.xml`.

Step 3. Test the Adapter

Before deploying the adapter, you should test the behavior of the adapter. Create a service and an event and test them. You can use the steps described in [“Learning to Develop Adapters Using the DBMS Sample Adapters” on page E-1](#) as a guide to using the adapter. Once testing is successfully complete, you are ready to customize the cloned adapter to meet the needs of the EIS.

Step 4. Deploy the Adapter to WebLogic Integration

You can deploy the adapter either manually or from the WebLogic Server Administration Console. See [Chapter 10, “Deploying Adapters,”](#) for complete information.

Using the Logging Toolkit

Logging is an essential feature of an adapter component. Most adapters are used to integrate different applications; they do not interact with end users while data is being processed. Unlike a front-end component, when an adapter encounters an error or warning condition, it cannot stop processing and wait for an end-user to respond.

With the ADK, you can log adapter activity by implementing a logging framework. This framework gives you the ability to log internationalized and localized messages to multiple output destinations. It provides a range of configuration parameters you can use to tailor message category, priority, format, and destination.

This section contains information about the following subjects:

- [Logging Toolkit](#)
- [Logging Configuration File](#)
- [Logging Concepts](#)
- [How to Set Up Logging](#)
- [Logging Framework Classes](#)
- [Internationalization and Localization of Log Messages](#)
- [Saving Contextual Information in a Multithreaded Component](#)

Logging Toolkit

The ADK logging toolkit allows you to log internationalized messages to multiple output destinations. The logging toolkit leverages the work of the Apache Log4j open source project. This product includes software developed by the Apache Software Foundation (<http://www.apache.org>).

The logging toolkit is a framework that wraps the necessary Log4j classes to provide added functionality for J2EE-compliant adapters. It is provided in the `logtoolkit.jar` file under `WLI_HOME/lib`. This JAR file depends on DOM, XERCES, and Log4j. The XERCES dependency is satisfied by the `weblogic.jar` and `xmlx.jar` files provided with WebLogic Server. The required version of Log4j, `log4j.jar`, is provided in `WLI_HOME/common/lib`.

The Log4j package is distributed under the Apache public license, a full-fledged open source license certified by the open source initiative. The latest Log4j version, including full-source code, class files, and documentation, can be found at the Apache Log4j Web site (<http://www.apache.org>).

Logging Configuration File

Throughout this section, you will see references to and code excerpts from the logging configuration file. This file is an `.xml` file that is identified by the adapter logical name, such as `BEA_WLS_DBMS_ADK.xml`. It contains the base information for the four logging concepts discussed in “Logging Concepts” on page 5-2 and can be modified for your specific adapter.

The ADK provides a basic logging configuration file, `BEA_WLS_SAMPLE_ADK.xml`, in `WLI_HOME/adapters/sample/src`. To modify this file for your adapter, run `GenerateAdapterTemplate`. This utility customizes the sample version of the logging configuration file with information pertinent to your new adapter and places the customized version in the new adapter’s development environment. For more information about `GenerateAdapterTemplate`, see Chapter 4, “Creating a Custom Development Environment.”

Logging Concepts

Before using the logging toolkit provided with the ADK, you should understand a few key concepts of the logging framework. Logging has four main components:

- [Message Categories](#)
- [Message Priority](#)
- [Message Appenders](#)

- [Message Layout](#)

These components work together to enable you to log messages according to message type and priority, and to control, at run time, how these messages are formatted and where they are reported.

Message Categories

Categories identify log messages according to criteria you define and are a central concept of the logging framework. In the ADK, a category is identified by its name, such as `BEA_WLS_SAMPLE_ADK.DesignTime`.

Categories are hierarchically defined and any category can inherit properties from a parent category. The hierarchy is defined as follows:

- A category is an ancestor of another category if its name, followed by a dot, is a prefix of the descendant category name.
- A category is a parent of a child category if there are no ancestors between itself and the descendant category.

For example, `BEA_WLS_SAMPLE_ADK.DesignTime` is a descendant of `BEA_WLS_SAMPLE_ADK` which, in turn, is a descendant of the root category, as shown in the following diagram.

```

ROOT CATEGORY
|
| ->BEA_WLS_SAMPLE_ADK
|
| ->BEA_WLS_SAMPLE.ADK.DesignTime
    
```

The root category resides at the top of the hierarchy; it cannot be deleted or retrieved by name.

When you create categories, you should name them according to components in the adapter to which they belong. For example, if an adapter has a design-time user interface component, the adapter might have a category with the following name: `BEA_WLS_SAMPLE_ADK.DesignTime`.

Message Priority

Every message has a priority that indicates its importance. Message priority is determined by the `ILogger` interface method used to log the message. For example, if you call the `debug` method on an `ILogger` instance, a debug message is generated.

The logging toolkit supports five possible priorities for a given message. These priorities are listed, in descending order of importance, in [Table 5-1](#).

Table 5-1 Logging Toolkit Priorities

Priority	Indicates
AUDIT	An extremely important log message related to the business processing performed by an adapter. Messages with this priority are always written to the log.
ERROR	An error in the adapter. Error messages are internationalized and localized for the user.
WARN	A situation that is not an error, but that might cause problems in the adapter. Warning messages are internationalized and localized for the user.
INFO	An informational message that is internationalized and localized for the user.
DEBUG	A debug message, that is, information used to determine how the internals of a component are working. Debug messages are typically not internationalized.

The `BEA_WLS_SAMPLE_ADK` category has priority `WARN` because of the following child element:

```
<priority value='WARN' class='com.bea.logging.LogPriority' />
```

The class for the priority must be `com.bea.logging.LogPriority`.

Assigning a Priority to a Category

You can assign a priority to a category. If a given category is not assigned a priority, it inherits one from its closest ancestor with an assigned priority; that is, the inherited priority for a given category is equal to the first non-null priority above the given category in the hierarchy.

A log message is sent to the log destination if its priority is higher than or equal to the priority of its category. Otherwise, the message is not written to the log. A category without an assigned priority inherits one from the hierarchy. To ensure that all categories can eventually inherit a priority, the root category always has an assigned priority. A log statement of priority p , in a category with inherited priority q , is enabled if $p \geq q$. This rule is based on the assumption that priorities are ordered as follows: `DEBUG < INFO < WARN < ERROR < AUDIT`.

Message Appenders

The logging framework allows an adapter to log messages to multiple destinations by using an interface called an appender. Log4j provides appenders for:

- Console
- Files
- Remote socket servers
- NT event loggers
- Remote UNIX Syslog daemons

In addition, the ADK logging toolkit provides an appender that you can invoke to send a log message to your WebLogic Server log.

A category may refer to multiple appenders. Each enabled logging request for a given category is forwarded to all the appenders in that category, as well as all the appenders higher in the hierarchy. In other words, appenders are inherited cumulatively from the category hierarchy.

For example, if a console appender is added to the root category, then all enabled logging requests are displayed, at a minimum, on the console. If, in addition, a file appender is added to category `C`, then enabled logging requests for `C` and `C`'s children are printed in a file and displayed on the console. It is possible to override this default behavior (that is, to stop appender inheritance from being cumulative) by setting the `additivity` flag to `false`.

Note: If you also add the console appender directly to C, you get two messages—one from C and one from root—on the console. The root category always logs to the console.

Listing 5-1 shows an appender for the WebLogic Server log.

Listing 5-1 Sample Code Showing an Appender for the WebLogic Server Log

```
<!--  
A WeblogicAppender sends log output to the Weblogic log. If running outside  
of WebLogic, the appender writes messages to System.out  
-->  
<appender name="WebLogicAppender"  
    class="com.bea.logging.WeblogicAppender" />  
</appender>
```

Message Layout

Log4j enables you to customize the format of a log message by associating a layout with an appender. The layout determines the format of a log message, while an appender directs the formatted message to its destination. The logging toolkit typically uses `PatternLayout` to format its log messages. `PatternLayout`, part of the standard Log4j distribution, lets you specify the output format according to conversion patterns similar to the C language `printf` function.

For example, if you invoke `PatternLayout` with the conversion pattern `%-5p%d{DATE} %c{4} %x - %m%n`, a message such as the following is generated:

```
AUDIT 21 May 2001 11:00:57,109 BEA_WLS_SAMPLE_ADK - admin opened connection  
to EIS
```

In this conversion pattern:

- The value of `%-5p` is the priority of the message; in the example shown here, the priority is `AUDIT`.
- The value of `%d{DATE}` is the date of the message; in the example shown here, the date is `21 May 2001 11:00:57,109`.
- The value of `%c{4}` is the category for the log message; in the example shown here, the category is `BEA_WLS_SAMPLE_ADK`.

The text after the dash (-) is the message of the statement.

Putting the Components Together

[Listing 5-2](#) declares a new category for the sample adapter, assigns a priority to the new category, and declares an appender in order to specify the type of file to which log messages should be sent.

Listing 5-2 Sample XML Code for Declaring a New Log Category

```
<!--
IMPORTANT!!! ROOT Category for the adapter; making this unique prevents other
adapters from logging to your category
-->
<category name='BEA_WLS_SAMPLE_ADK' class='com.bea.logging.LogCategory'>
  <!--
    Default Priority Level; may be changed at runtime
    DEBUG means log all messages from the adapter's code base
    INFO means log informationals, warnings, errors, and audits
    WARN means log warnings, errors, and audits
    ERROR means log errors and audits
    AUDIT means log audits only
  -->

  <priority value='WARN' class='com.bea.logging.LogPriority' />
  <appender-ref ref='WebLogicAppender' />
</category>
```

Note: You must specify the class as `com.bea.logging.LogCategory`.

How to Set Up Logging

Note: The following procedure is based on the assumption that you have cloned a development environment by running the `GenerateAdapterTemplate` utility. For more information about this utility, see [Chapter 4, “Creating a Custom Development Environment.”](#)

To set up the logging framework for your adapter:

1. Identify all the basic components used in the adapter. For example, if your adapter has an EventGenerator, you might want an EventGenerator component; if it supports a design-time GUI, you need a design-time component.
2. Open the base log configuration file from the cloned adapter. This file is found in `WLI_HOME/adapters/ADAPTER/src/`. Its name includes the `.xml` extension. For example, the DBMS sample adapter configuration file is `WLI_HOME/adapters/dbms/src/BEA_WLS_DBMS_ADK.xml`.
3. In the base log configuration file, add the category elements for all adapter components you identified in step 1. For each category element, establish a priority. Listing 5-3 shows how a category for an EventGenerator with a priority of DEBUG is added.

Listing 5-3 Sample Code for Adding an EventGenerator Log Category with a Priority of DEBUG

```
<category name='BEA_WLS_DBMS_ADK.EventGenerator'  
        class='com.bea.logging.LogCategory'>  
    <priority value='DEBUG'  
        class='com.bea.logging.LogPriority' />  
</category>
```

4. Determine which appender is needed and specify it in the configuration file. If necessary, add message formatting information. Listing 5-4 shows how a basic file appender is added within the `<appender>` element. Instructions within the `<layout>` element identify the message format.

Note: By default, `WebLogicAppender` is used in all sample adapters provided by WebLogic Integration.

Listing 5-4 Sample Code for Adding a File Appender and Layout Pattern

```
<!-- A basic file appender -->  
  
<appender name='FileAppender'  
    class='org.apache.Log4j.FileAppender'>  
  
    <!-- Send output to a file -->  
  
    <param name='File' value='BEA_WLS_DBMS_ADK.log' />  
  
    <!-- Truncate existing -->
```

```

<param name="Append" value="true"/>

<!-- Use a basic LOG4J pattern layout -->

<layout class='org.apache.Log4j.PatternLayout'>
  <param name='ConversionPattern' value='%-5p %d{DATE} %c{4}
    %x - %m%n' />
</layout>

</appender>

```

At this point, you should check the setting in the following configuration files:

- *WLI_HOME/adapters/ADAPTER/src/rar/META-INF/ra.xml* and *weblogic-ra.xml*—The `AbstractManagedConnectionFactory` uses the logging information entered in the base configuration file to configure the log framework at initialization time.
- *WLI_HOME/adapters/ADAPTER/src/war/web-inf/web.xml*—The `RequestHandler` (the parent of `AbstractDesignTimeRequestHandler`) uses the logging information entered in the base configuration file to configure the log framework at initialization time.

In the preceding paths, *ADAPTER* represents the name of your adapter. For example, the name of the DBMS sample adapter appears in the pathname for the associated configuration file, as follows:

WLI_HOME/adapters/dbms/src/rar/META-INF/ra.xml

Logging Framework Classes

In addition to understanding the basic concepts of the logging framework, you also need to understand the three main classes provided in the logging toolkit:

- [com.bea.logging.ILogger](#)
- [com.bea.logging.LogContext](#)
- [com.bea.logging.LogManager](#)

`com.bea.logging.ILogger`

This class is the main interface to the logging framework. It provides numerous convenience methods for logging messages.

The “[How to Set Up Logging](#)” procedure explains how you can configure logging in the base log configuration file. You can also configure logging programmatically by implementing the following logging methods:

- `logger.setPriority("DEBUG")` changes the minimum priority of messages printed from the current `ILogger`.
- `logger.addRuntimeDestination (writer)` adds the appender that is used when the container passes its `PrintWriter` to the adapter.
- `logger.warn("Some message", true)` logs a message with the priority level of `WARN`, without using the `ResourceBundle`. The boolean indicates that the string is a message, not a key.
- `logger.warn("someKey")` logs a message with the priority level `WARN`, by looking it up with "someKey" in `ResourceBundle`.
- `logger.info("someKey", anObjArray)` logs a message with the priority level of `INFO` by looking up a template with `someKey` in `ResourceBundle` and filling in the blanks with the elements of `anObjArray`.
- `logger.error(exception)` logs a message with the priority level of `ERROR`, by passing an exception (`Throwable`) to this method. It calls `getMessage()` and includes a stack trace. (All logging methods that take a `Throwable` as an argument log a stack trace.)

`com.bea.logging.LogContext`

This class encapsulates the information needed to identify an `ILogger` instance in the logging framework. Currently, the `LogContext` class encapsulates a log category name and a locale, such as `en_US`. This class is the primary key for uniquely identifying an `ILogger` instance in the log manager.

`com.bea.logging.LogManager`

This class provides a method that allows you to configure the logging framework and gain access to `ILogger` instances.

To ensure that you can properly configure the logging toolkit for your adapter, the ADK implements the `LogManager`'s `configure()` method with the arguments shown in [Listing 5-5](#).

Listing 5-5 Sample Code for Configuring the Logging Toolkit

```

public static LogContext
    configure(String strLogConfigFile,
              String strRootLogContext,
              String strMessageBundleBase,
              Locale locale,
              ClassLoader classLoader)

```

[Table 5-2](#) describes the arguments passed by `configure()`.

Table 5-2 `configure()` Arguments

Argument	Description
<code>strLogConfigFile</code>	File that contains the log configuration information for your adapter. The file's location should be included in the classpath. We recommend that you include this file in your adapter's main JAR file so that it can be included in the WAR and RAR files for your adapter. This file should conform to the <code>Log4j.dtd</code> . The <code>Log4j.dtd</code> file is provided in the <code>Log4j.jarfile</code> provided with WebLogic Integration.
<code>strRootLogContext</code>	Name of the logical root of the category hierarchy for your adapter. For the sample adapter, its value is <code>BEA_WLS_SAMPLE_ADK</code> .
<code>strMessageBundleBase</code>	Base name of the message bundles for your adapter. The ADK requires the use of message bundles. For the sample adapter, its value is <code>BEA_WLS_SAMPLE_ADK</code> .
<code>locale</code>	Nation and language of the users. The logging toolkit organizes categories into different hierarchies, based on locale. For example, if your adapter supports two locales, <code>en_US</code> and <code>fr_CA</code> , the logging toolkit maintains two hierarchies: one for <code>en_US</code> and one for <code>fr_CA</code> .
<code>classLoader</code>	<code>ClassLoader</code> that should be used by the <code>LogManager</code> to load resources, such as <code>ResourceBundles</code> and log configuration files.

Once the configuration is complete, you can retrieve `ILogger` instances for your adapter by supplying a `LogContext` object.

Listing 5-6 Sample Code for Supplying a LogContext Object

```
LogContext logContext = new LogContext("BEA_WLS_SAMPLE_ADK",
    java.util.Locale.US);

ILogger logger = LogManager.getLogger(logContext); logger.debug("I'm
logging now!");
```

The ADK hides most of the log configuration and setup from you. The `com.bea.adapter.spi.AbstractManagedConnectionFactory` class configures the logging toolkit for service connections and the `AbstractEventGenerator` configures the logging toolkit for event connections. In addition, all of the Client Connector Interface (CCI) and Service Provider Interface (SPI) base classes included in the ADK provide access to an `ILogger` and the `LogContext` associated with it.

An adapter may also include layers that support the CCI/SPI layer, such as a socket layer used for establishing communication with the EIS. To make it possible for such adapters to access the correct `ILogger` object, you can take either of two approaches:

- The CCI/SPI layers can pass the `LogContext` object into the lower layers. This method works, but it adds overhead.
- The CCI layer can establish the `LogContext` for the current running thread at the earliest possible place in the code. The ADK's `com.bea.adapter.cci.ConnectionFactoryImpl` class sets the `LogContext` for the current running thread in the `getConnection()` methods. The `getConnection()` methods are the first point of contact between a client program and your adapter. Consequently, lower layers in an adapter can safely access the `LogContext` for the current running thread by using the following code:

Listing 5-7 Code Accessing LogContext for the Current Thread

```
public static LogContext getLogContext(Thread t)
    throws IllegalStateException, IllegalArgumentException
```

Additionally, we supply the following convenience method on `LogManager`:

```
public static ILogger getLogger() throws IllegalStateException
```

This method provides an `ILogger` for the current running thread. There is one caveat to using this approach: lower layers should not store `LogContext` or `ILogger` as members. Rather, they should dynamically retrieve them from `LogManager`. An `IllegalStateException` is thrown if this method is called before a `LogContext` is set for the current running thread.

Internationalization and Localization of Log Messages

Internationalization (I18N) and localization (L10N) are central concepts to the ADK logging framework. All logging convenience methods on the `ILogger` interface, except the debug methods, allow I18N. The implementation follows the Java Internationalization standards, using `ResourceBundle` objects to store locale-specific messages or templates. Sun Microsystems provides a good online tutorial on using the I18N and L10N standards of the Java language.

Saving Contextual Information in a Multithreaded Component

Most real-world systems must manage multiple clients simultaneously. In a typical multithreaded implementation of such a system, different threads handle different clients. Logging is especially well suited to tracing and debugging complex distributed applications. A common way of differentiating between the logging output of two clients is to instantiate a separate category for each client. This approach has a drawback however: categories proliferate and the overhead required to manage them increases.

A lighter technique is to stamp each log request initiated from the same client interaction with a unique identifier. Neil Harrison describes this method in “Patterns for Logging Diagnostic Messages” in *Pattern Languages of Program Design 3*, edited by R. Martin, D. Riehle, and F. Buschmann (Addison-Wesley, 1997).

To stamp each request with a unique identifier, the user pushes contextual information into the Nested Diagnostic Context (NDC). The logging toolkit provides a separate interface for accessing NDC methods. The interface is retrieved from the `ILogger` by using the `getNDCInterface()` method.

NDC printing is turned on in the XML configuration file (with the symbol `%x`). Every time a log request is made, the appropriate logging framework component includes the entire NDC stack for the current thread in the log output. The user does not need to intervene in this process. In fact, the user is responsible only for placing the correct information in the NDC by using the push and pop methods at a few well-defined points in the code.

Listing 5-8 Sample Code

```
public void someAdapterMethod(String aClient) {
    ILogger logger = getLogger();
    INestedDiagnosticContext ndc = logger.getNDCInterface();
    // I'm keeping track of this client name for all log messages
    ndc.push("User name=" + aClient);
    // method body ...
    ndc.pop();
}
```

A good place to use the NDC is in your adapter's CCI Interaction object.

Developing a Service Adapter

A service connection receives an XML request document from a client and invokes the associated function in the underlying EIS. Service connections are consumers of messages; they may or may not provide responses. They perform the following four functions:

- They receive service requests from an external client.
- They transform an XML request document into the EIS-specific format. The request document conforms to the request XML schema for the service. The request XML schema is based on metadata in the EIS.
- They invoke the underlying function in the EIS and wait for a response from that function.
- They transform the response from the EIS-specific data format to an XML format that conforms to the response XML schema for the service. The response XML schema is based on metadata in the EIS.

This section contains information about the following subjects:

- [Service Connections in a Run-Time Environment](#)
- [Flow of Events](#)
- [Step 1: Research Your Environment Requirements](#)
- [Step 2: Configure the Development Environment](#)
- [Step 3: Implement the SPI](#)
- [Step 4: Implement the CCI](#)

- [Step 5: Enable Environment Variable Support \(Optional\)](#)
- [Step 6: Test the Adapter](#)
- [Step 7: Deploy the Adapter](#)

J2EE-Compliant Adapters Not Specific to WebLogic Integration

The steps outlined in this section are directed primarily at developing adapters for use with WebLogic Integration. You can also use the ADK to develop adapters for use outside the WebLogic Integration environment, however, by following the same steps with certain modifications. For instructions, see [Appendix A, “Creating an Adapter Not Specific to WebLogic Integration.”](#)

Service Connections in a Run-Time Environment

[Figure 6-1](#) and [Figure 6-2](#) show the processes that are executed when a service connection is used in a run-time environment. [Figure 6-1](#) shows an asynchronous service connection; [Figure 6-2](#), a synchronous adapter.

Figure 6-1 Asynchronous Service Connection in a Run-Time Environment

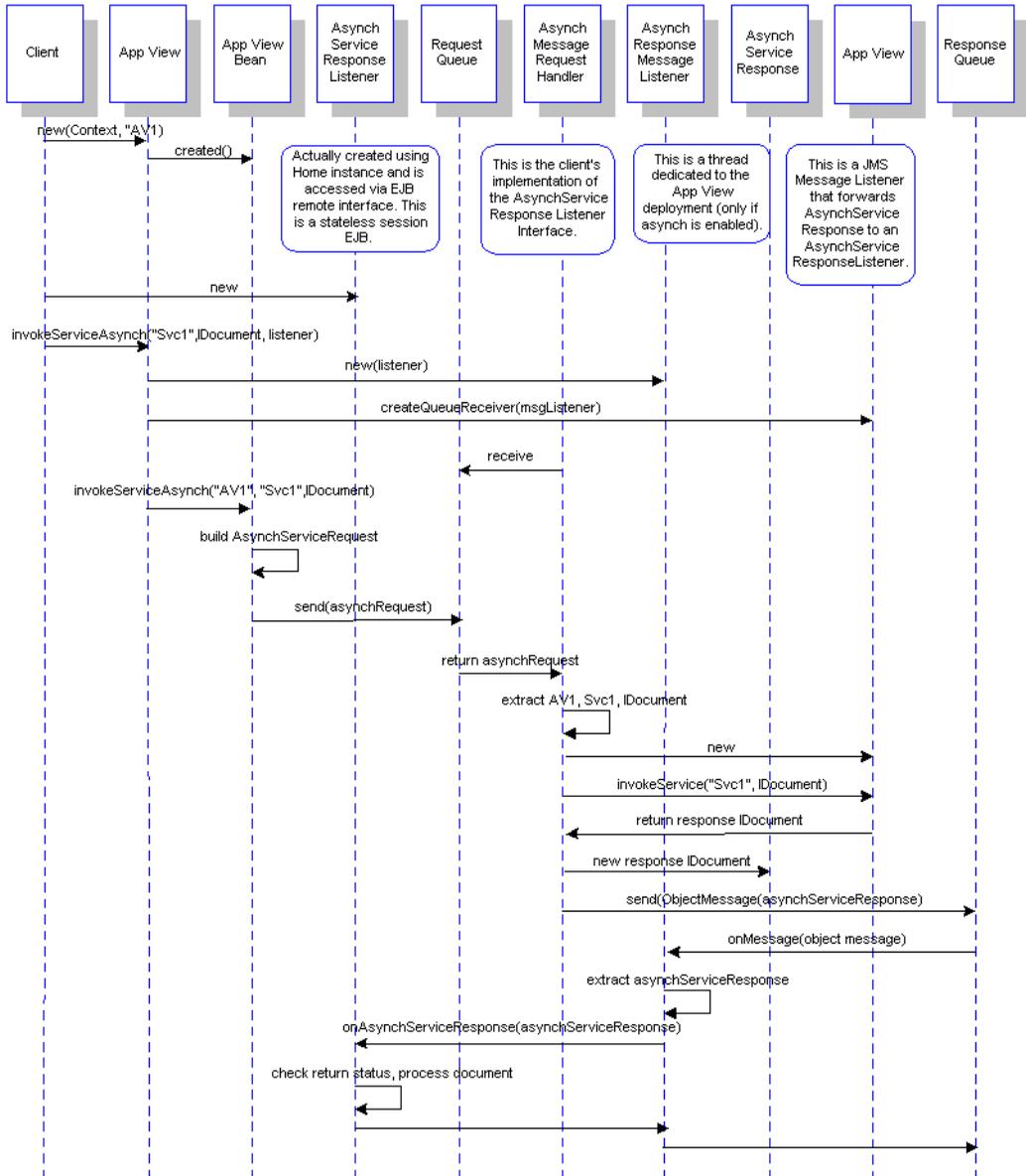
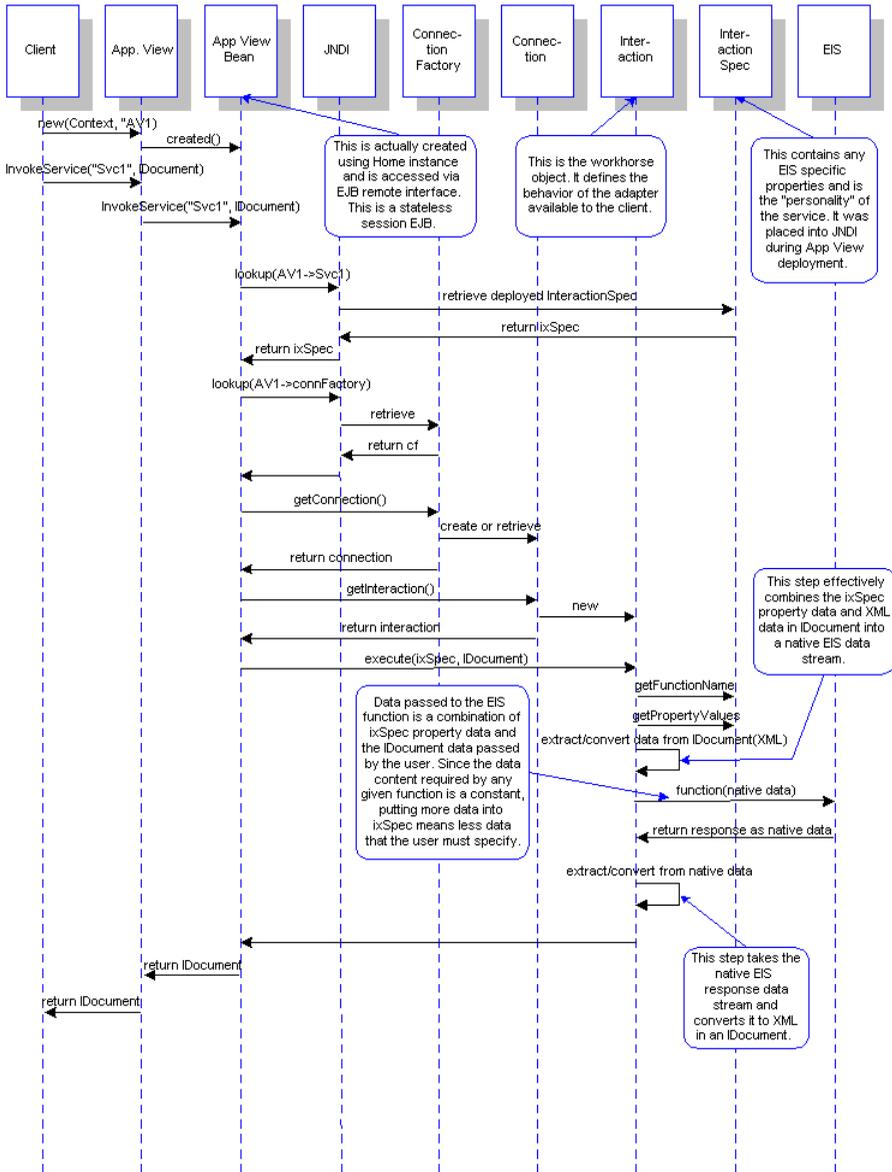


Figure 6-2 Synchronous Service Connection in a Run-Time Environment

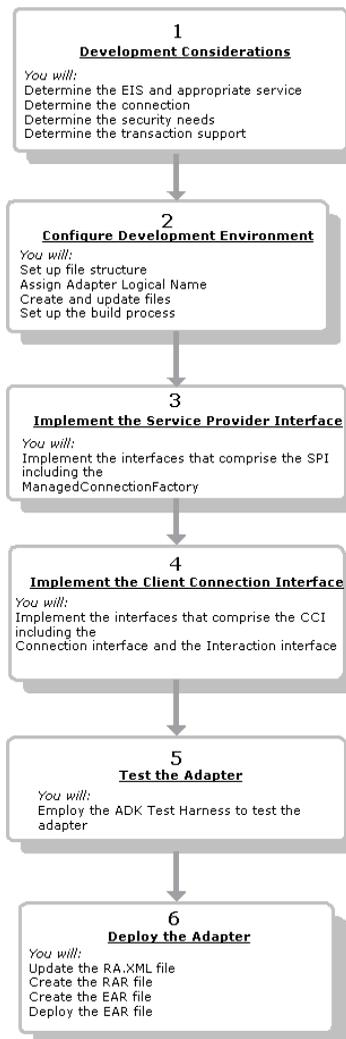
Developing a Service Adapter



Flow of Events

Figure 6-3 outlines the steps required to develop a service connection.

Figure 6-3 Flow of Events in Service Connection Development Process



Step 1: Research Your Environment Requirements

Before you start developing your service connection, you must identify the resources needed in your environment to support it. This section provides a high-level description of the prerequisites for a development environment. For a complete list of required resources, see [Appendix C, “Adapter Setup Worksheet.”](#)

- Identify the required EIS and the service appropriate for it.

Based on your knowledge of the EIS, identify the interface to the back-end functionality.

- Identify the expensive connection object.

An *expensive* connection object is an object required to invoke a function within the EIS. This function, in turn, is required for communicating with the EIS.

An expensive connection object requires an allocation of system resources, such as a socket connection or DBMS connection. A valuable benefit of using the J2EE Connector Architecture is that the application server pools these objects. Because the object for your adapter will be pooled by the application server, you need to identify it.

- Identify your security needs.

To pass connection authentication across the connection request path, your adapter must implement a `ConnectionRequestInfo` class. To facilitate such an implementation, the ADK provides the class `ConnectionRequestInfoMap`. You can use this class to map authorization information, such as username and password, to the connection.

The ADK conforms to the *J2EE Connector Architecture Specification 1.0*. For more information about connection architecture security, see the “Security” section of that document. You can download the specification in PDF format (for easy printing) from the following URL:

<http://java.sun.com/j2ee/>

- Identify the type of transaction support needed for your adapter.

Decide which of the following types of transaction demarcation support to implement with your adapter:

- No transaction
- Local transaction demarcation
- XA-compliant transaction demarcation

Note: For more information about transaction demarcation support, see “[Transaction Demarcation](#)” on page 6-24, or see:

http://java.sun.com/blueprints/guidelines/designing_enterprise_applications/transaction_management/platform/index.html

- Decide if you will support environment variables for services.

Environment variables are used to isolate environment specific information in a way that allows system administrators and application deployers to modify it as necessary for the target environment. If your adapter defines service properties that include environment-specific identifiers or other information, you should implement environment variable support in your service adapter.

Step 2: Configure the Development Environment

This section provides a four-step procedure (steps 2a-2d) for configuring your environment.

Note: A simple way of completing this procedure is by running the `GenerateAdapterTemplate` utility. For more information, see [Chapter 4, “Creating a Custom Development Environment.”](#)

Step 2a: Set Up the Directory Structure

When you install WebLogic Integration, you also create the directory structure necessary not only to run an adapter, but also to use the ADK. The ADK files reside under `WLI_HOME/adapters/`, where `WLI_HOME` is the directory in which you installed WebLogic Integration. Be sure to verify that your `WLI_HOME` directory is populated with the necessary directories and files at installation time.

The following table describes the directory structure under `WLI_HOME`.

Table 6-1 ADK Directory Structure

Pathname	Description
<code>adapters</code>	Directory containing the ADK.
<code>adapters/src/war</code>	Directory containing <code>.jsp</code> files, images, and so on. All files in this directory should be included in the WAR file for an adapter.
<code>adapters/utills</code>	Directory containing files used by the build process, including a file that timestamps JAR files.

Table 6-1 ADK Directory Structure

Pathname	Description
adapters/dbms	Directory containing a sample J2EE-compliant adapter built with the ADK.
adapters/sample	Directory containing a sample adapter that you can use to start developing your own adapter.
adapters/sample/project	Directory containing the Apache Jakarta Ant build file: <code>build.xml</code> . This file contains build information for compiling the source code, generating the JAR and EAR files, and generating Javadoc information. For details about building the adapter, see “Step 2c: Set Up the Build Process” on page 6-11 .
adapters/sample/src	Directory containing all the source code for an adapter. The decision about whether to provide source code with your adapter is yours.
adapters/sample/src/ BEA_WLS_SAMPLE_ADK- base.properties	File containing messages used by the adapter for internationalization and localization. This file is merged with the ADK properties file (<code>WLI_HOME/adapters/src/ADK.properties</code>) to form the <code>BEA_WLS_SAMPLE_ADK.properties</code> file, the final properties file that will be used by the adapter. This merge happens in the <code>build.xml</code> file in the <code>packages</code> target.
adapters/sample/src/ BEA_WLS_ SAMPLE_ADK.xml	File that provides a basic configuration file for the logging framework. You should use this file to develop your own adapter logging configuration file.
adapters/sample/src/rar/ META-INF/ra.xml	File containing configuration information about a J2EE-compliant adapter. Use this file as a guide to which parameters are needed by the ADK run-time framework.
adapters/sample/src/rar/META- INF/weblogic-ra.xml	File containing configuration information for a J2EE-compliant adapter that is specific to the WebLogic Server J2EE engine. Use this file as an example for setting up the <code>weblogic-ra.xml</code> file for your adapter. The file is required for WebLogic Server.
adapters/sample/src/sample	Directory containing the source code for the adapter.

Table 6-1 ADK Directory Structure

Pathname	Description
<code>adapters/sample/src/war</code>	Directory containing <code>.jsp</code> files, <code>.html</code> files, images, and so on. All files in this directory should be included in the Web application archive (<code>.war</code>) file for an adapter.
<code>adapters/sample/src/war/WEB-INF/web.xml</code>	Web application descriptor. See “ The web.xml and weblogic.xml Descriptor Files ” on page 6-9 for details.
<code>adapters/sample/src/war/WEB-INF/weblogic.xml</code>	File containing WebLogic Server-specific attributes for a Web Application. See “ The web.xml and weblogic.xml Descriptor Files ” on page 6-9 for details.
<code>adapters/sample/src/ear/META-INF/application.xml</code>	J2EE application that contains a connector and a Web application for configuring application views for the adapter.

The web.xml and weblogic.xml Descriptor Files

The `web.xml` and `weblogic.xml` descriptors for your adapter generally follow a very simple pattern. They list the names of all the JSP pages in your design-time web application and provide additional setup information. Because most adapters contain very similar web descriptors, the ADK provides a means to automatically generate them. This frees the adapter developer from maintaining a large descriptor that is mostly identical to other adapter’s web descriptors.

The generation of the web application descriptors may be requested by including and calling a special Ant target in your Ant `build.xml` file for your adapter. If you clone the ADK sample adapter using `GenerateAdapterTemplate`, the resulting `build.xml` already includes the necessary Ant target and a call to use that target. Look at the `WLI_HOME/adapters/sample/project/build.xml` file and find the `generate_web_descriptors` target. This Ant target takes in a file called `web-gen.properties` and generates the `web.xml` and `weblogic.xml` descriptors from the information contained in it. Notice that in the sample adapter `build.xml`, this target is called near the top of the `packages` target.

The sample adapter includes a `web-gen.properties` file as a template that can be modified for your adapter. The following properties are listed in this file:

- `display-name`—This is the value used in the `display-name` element of `web.xml`. It should be the adapter logical name for your adapter (for example, `BEA_WLS_SAMPLE_ADK` for the sample adapter).

- `version`—The version of your design-time web application and the value that is used in the `version` element of `web.xml` (for example, `8.1.0` for the sample adapter). Always use the same value you specified in `GenerateAdapterTemplate`.
- `request-handler-class`—This is the full class name of the design-time request handler implementation class for your adapter. This class that extends `AbstractDesignTimeRequestHandler` and normally resides in `adapter_package.web.DesignTimeRequestHandler`.
- `adapter-logical-name`—The adapter logical name for your adapter. It should be the same value you used in `GenerateAdapterTemplate` for example, `BEA_WLS_SAMPLE_ADK` for the sample adapter)
- `debug-setting`—This setting enables or disables debug capabilities and is specified as `on` or `off`. Turning `debug` `on` enables any debug statements you have placed in your source code (or that were there from cloning the sample adapter source) that use the `ILogger.debug()` method. Turning `debug` `off` disables these debug statements from being logged to the log file.
- `extra-jsp-list`—a comma separated list of additional JSPs. The standard ones are:
 - `addevent`, `addservc`, `confconn`, `edtevent`, `edtservc`, `event`, `service`, `testform`, and `varset`

You do not need to include those in your list. For example, if you add a JSP called `mybrowser.jsp`, your `extra-jsp-list` would be: `extra-jsp-list=mybrowser`

Fill in the information appropriate for your adapter in the copy of this file made for you by `GenerateAdapterTemplate`. Do not modify the original file.

At build time, the `generate_web_descriptors` target substitutes the information in your `web-gen.properties` file and generates the `web.xml` and `weblogic.xml` descriptors appropriate for your adapter's design-time web application. The descriptors are placed in the `src/war/WEB-INF` directory for your adapter.

If you wish to tightly control your `web.xml` and `weblogic.xml` descriptors, you can comment out any call to `generate_web_descriptors` in your `build.xml` file, and maintain the `web.xml` and `weblogic.xml` descriptors manually.

Creating A Development Tree Within the Directory Structure

When you clone a development tree by using `GenerateAdapterTemplate`, the contents of all the directories under `WLI_HOME/adapters/sample` are automatically cloned and updated to reflect the new development environment.

The changes are also reflected in the file

`WLI_HOME/adapters/ADAPTER/src/overview.html`, where the value of `ADAPTER` is the name of the new development directory. This file also contains code that you can copy and paste into the `config.xml` file for the new adapter that sets up WebLogic Integration to host the adapter.

Step 2b: Assign the Adapter Logical Name

Assign a logical name to the adapter. By convention, this name is made up of three items—the vendor name, the type of EIS connected to the adapter, and the version number of the EIS—separated by underscores, as follows:

vendor_EIS-type_EIS-version

For example:

`BEA_WLS_SAMPLE_ADK`

For more information about the logical name of an adapter, see [“Adapter Logical Name” on page 2-5](#).

Step 2c: Set Up the Build Process

The ADK employs a build process based on Ant, a 100% pure Java-based build tool. For more information about Ant, see [“Ant-Based Build Process” on page 3-3](#). For more information about using Ant, see:

<http://jakarta.apache.org/ant/index.html>

The sample adapter provided with the ADK (in `WLI_HOME/adapters/sample/project`) contains `build.xml`, the Ant build file for the sample adapter. It contains the tasks needed to build a J2EE-compliant adapter. When you run the `GenerateAdapterTemplate` utility to clone a development tree for your adapter, a `build.xml` file is created specifically for that adapter. This automatic file generation frees you from having to customize the sample `build.xml` and ensures that the code is correct. For information about using the `GenerateAdapterTemplate` utility, see [Chapter 4, “Creating a Custom Development Environment.”](#)

Manifest File

Among the files created by `GenerateAdapterTemplate` is `MANIFEST.MF`, the manifest file. This file contains classloading instructions for each component that uses the file. A manifest file is created for each `/META-INF` directory except `ear/META-INF`.

[Listing 6-1](#) shows an example of the manifest file included with the sample adapter.

Listing 6-1 Manifest File Example

```
Manifest-Version: 1.0

Created-By: BEA Systems, Inc.

Class-Path: shared.jar
```

The first two lines of the file contain version and vendor information. The third line contains the relevant classpath or classloading instructions. The `Class-Path` property contains references to resources required by the component and a list of shared JAR files. (Filenames in the list are separated by spaces.) Make sure the JAR files are included in the shared area of the EAR file. (For details, see [“Enterprise Archive \(EAR\) Files” on page 2-8.](#))

The JAR tool imposes a 72-character limit on the length of the `Class-Path:` line. Lines longer than 72 characters should carry over to the next line and begin with a preceding space, as in the following:

```
Class-Path: .....72 chars of classpath
<space>more classpath
```

In the sample ADK adapters, all shared JAR files are combined into a single JAR file (`shared.jar`) using the following Ant commands:

Listing 6-2 Shared JAR File Creation

```
<jar jarfile='${LIB_DIR}/shared.jar'>
<zipfileset src='${LIB_DIR}/${JAR_FILE}'>
<exclude name='META-INF/MANIFEST.MF' />
</zipfileset>
<zipfileset src='${WLI_LIB_DIR}/adk.jar'>
<exclude name='META-INF/MANIFEST.MF' />
</zipfileset>
<zipfileset src='${WLI_LIB_DIR}/adk-eventgenerator.jar'>
<exclude name='META-INF/MANIFEST.MF' />
</zipfileset>
<zipfileset src='${WLI_LIB_DIR}/wlai-core.jar'>
```

```

<exclude name='META-INF/MANIFEST.MF' />
</zipfileset>
<zipfileset src='${WLI_LIB_DIR}/wlai-client.jar'>
<exclude name='META-INF/MANIFEST.MF' />
</zipfileset>
</jar>

```

Note: When it is included in a WAR file, the filename `MANIFEST.MF` must be spelled in all uppercase letters. If it is spelled otherwise, it is not recognized on a UNIX system and an error occurs.

build.xml Components

To learn how `build.xml` works, open it and review its components. This section provides descriptions of the main file elements. Refer to these descriptions as you review the contents of `build.xml`.

Note: The examples in this section are taken from the sample adapter itself, *not* from a cloned version of it.

1. The first line sets the name attribute of the root project element:

```
<project name='BEA_WLS_SAMPLE_ADK' default='all' basedir='.'>
```

2. The following line reads the value of the `WL_HOME` environment variable from your system.

```
<property environment="env" />
```

3. The location of the `weblogic.jar` file and the integration home directory (`WLI_HOME`) are defined, ensuring access to required parsers and J2EE packages:

```
<property name="WEBLOGIC_JAR" value="${env.WL_HOME}/lib/weblogic.jar" />
<property name="WLI_HOME" value="${basedir}/../../../../" />
```

4. Names are assigned to the archive files (JAR, WAR, and RAR files), as shown in the following example listing. Note that as of WebLogic Integration 8.1 the `EventRouter` is no longer a separate component within the adapter, so its JAR and WAR files are no longer built

Listing 6-3 Setting Archive Filenames

```
<property name="JAR_FILE" value="BEA_WLS_SAMPLE_ADK.jar" />
<property name="EIS_JAR_FILE" value="sample-eis.jar" />
<property name="EIS_JAR_PATH" value="APP-INF/lib/${EIS_JAR_FILE}" />
<property name="RAR_FILE" value="BEA_WLS_SAMPLE_ADK.rar" />
<property name="WAR_FILE" value="BEA_WLS_SAMPLE_ADK_Web.war" />
<property name="EAR_FILE" value="BEA_WLS_SAMPLE_ADK.ear" />
```

5. The standard properties for the ADK are listed as shown in [Listing 6-4](#).

Listing 6-4 Standard ADK Properties

```
<property name="ADAPTER_DIR" value="${WLI_HOME}/adapters/sample" />
<property name="SRC_DIR" value="${ADAPTER_DIR}/src" />
<property name="LIB_DIR" value="${ADAPTER_DIR}/lib" />
<property name="DOC_DIR" value="${ADAPTER_DIR}/docs/api" />
<property name="WLI_LIB_DIR" value="${WLI_HOME}/lib" />
<property name="METAMATA_JAR" value="${WLI_LIB_DIR}/metamata.jar" />
<property name="LOG4J_JAR" value="${WLI_LIB_DIR}/log4j.jar" />
<property name="JUNIT" value="${WLI_LIB_DIR}/junit.jar" />
<property name="HTTPUNIT" value="${WLI_LIB_DIR}/httpunit.jar" />
<property name="TIDY" value="${WLI_LIB_DIR}/Tidy.jar" />
<property name="ADK" value="${WLI_LIB_DIR}/adk.jar" />
<property name="ADK_WEB" value="${WLI_LIB_DIR}/adk-web.jar" />
<property name="ADK_TEST" value="${WLI_LIB_DIR}/adk-test.jar" />
<property name="ADK_EVENTGENERATOR"
    value="${WLI_LIB_DIR}/adk-eventgenerator.jar" />
<property name="BEA" value="${WLI_LIB_DIR}/bea.jar" />
<property name="LOGTOOLKIT" value="${WLI_LIB_DIR}/logtoolkit.jar" />
<property name="WEBTOOLKIT" value="${WLI_LIB_DIR}/webtoolkit.jar" />
<property name="WLAI_CORE" value="${WLI_LIB_DIR}/wlai-core.jar" />
<property name="WLAI_CLIENT" value="${WLI_LIB_DIR}/wlai-client.jar" />
<property name="XMLTOOLKIT" value="${WLI_LIB_DIR}/xmltoolkit.jar" />
<property name="XCCI" value="${WLI_LIB_DIR}/xcci.jar" />
```

You should not need to alter these properties. After them, however, you can add any other JAR files and/or classes needed by your adapter.

6. The classpath is set up for compiling as shown in the following listing.

Listing 6-5 Setting the Classpath

```

<path id='CLASSPATH'>
  <pathelement location='${SRC_DIR}' />
  <pathelement path='${ADK}:${ADK_EVENTGENERATOR}:
    ${ADK_WEB}:${ADK_TEST}:${WEBTOOLKIT}:${WLAI_CORE}:
    ${WLAI_CLIENT}' />
  <pathelement path='${WEBLOGIC_JAR}:${env.BEA_HOME}' />
  <pathelement path='${JUNIT}:${HTTPUNIT}:${TIDY}' />
  <pathelement path='${env.JAVA_HOME}/lib/tools.jar' />
</path>

```

After this information, you have the option of calling any of the following three combinations of files:

- All the binaries and archives for the adapter
 - All the binaries and archives for the adapter, plus the Javadoc:


```
<target name='release' depends='all,apidoc' />
```
 - A `version_info` file for inclusion with the archive files
-

7. The contents of the JAR file for the adapter are specified: run-time aspects of the adapter are included in the main JAR, while additional classes, such as the design-time GUI support classes, are included in the WAR or other JAR files, as shown in the following listing.

Listing 6-6 Sample Code for Setting Values in a JAR File

```

<target name='jar' depends='packages,version_info'>
  <delete file='${LIB_DIR}/${JAR_FILE}' />
  <mkdir dir='${LIB_DIR}' />
  <jar jarfile='${LIB_DIR}/${JAR_FILE}'>

```

The `includes` list from the adapter's source directory is specified. For the sample adapter described in this section, all the classes in the `sample/cci` and `sample/spi` packages are included, as well as the logging configuration file and message bundles.

Listing 6-7 Sample Code for Including the *Includes* List

```
<fileset dir='${SRC_DIR}'
  includes='sample/cci/*.class,sample/spi/*.class,
  *.xml,*.properties' />
```

Version information about the JAR file is provided, as shown in the following listing.

Listing 6-8 Setting JAR File Version Information

```
<!-- Include version information about the JAR file -->
  <fileset dir='${basedir}'
    includes='version_info.xml' />
</jar>
```

8. The JAR file for the dummy EIS used by this adapter is created. The classes making up this dummy EIS are placed in a separate JAR file to allow for the control of which ClassLoader loads these classes. You can remove this target and any references to it when implementing your own EIS communication logic.

Listing 6-9 Sample Code for Creating the Dummy EIS

```
<target name="eis_jar" depends="packages,version_info">
  <delete file="${LIB_DIR}/${EIS_JAR_PATH}" />
  <mkdir dir="${LIB_DIR}" />
  <mkdir dir="${LIB_DIR}/APP-INF" />
  <mkdir dir="${LIB_DIR}/APP-INF/lib" />
  <jar jarfile="${LIB_DIR}/${EIS_JAR_PATH}">
    <fileset dir="${SRC_DIR}" includes="sample/eis/*.class" />
    <fileset dir="${basedir}" includes="version_info.xml" />
  </jar>
</target>
```

9. The J2EE adapter archive (RAR) file is created. This file should contain all the classes and JAR files needed by the adapter. It can be deployed on any J2EE-compliant application server on which the adapter depends. Our example adapter includes the following targets:

- Version information for this RAR file
- The deployment descriptor for the adapter

The following listing shows how the RAR file for the sample adapter is created.

Listing 6-10 Sample Code for Creating the Connection Architecture RAR File

```
<target name='rar' depends='jar'>
<delete file='${LIB_DIR}/${RAR_FILE}' />
<mkdir dir='${LIB_DIR}' />
<jar jarfile='${LIB_DIR}/${RAR_FILE}'
  manifest='${SRC_DIR}/rar/META-INF/MANIFEST.MF'>
  <fileset dir='${SRC_DIR}/rar' includes='META-INF/ra.xml,
    META-INF/weblogic-ra.xml' excludes=
    'META-INF/MANIFEST.MF' />
</jar>
</target>
```

10. The J2EE Web application archive (WAR) file is created. This file also includes code that cleans up the existing environment.

Listing 6-11 Sample Code for Producing the WAR File

```
<target name='war' depends='jar'>
<!-- Clean-up existing environment -->
  <delete file='${LIB_DIR}/${WAR_FILE}' />
  <copy file='${WLI_HOME}/adapters/src/war/WEB-INF/taglibs/
    adk.tld' todir='${SRC_DIR}/war/WEB-INF/taglibs' />
  <java classname='weblogic.jspc' fork='yes'>
    <arg line='-d ${SRC_DIR}/war -webapp ${SRC_DIR}/
      war -compileAll -depend' />
    <classpath refid='CLASSPATH' />
  </java>
<!-- The first adapter should compile the common ADK JSPs -->
  <java classname='weblogic.jspc' fork='yes' failonerror="true">
    <arg line='-d ${WLI_HOME}/adapters/src/war -webapp
      ${WLI_HOME}/adapters/src/war -compileAll
      -depend' />
    <classpath refid='CLASSPATH' />
  </java>
```

Developing a Service Adapter

```
<war warfile='${LIB_DIR}/${WAR_FILE}'
    webxml='${SRC_DIR}/war/WEB-INF/web.xml'
    manifest='${SRC_DIR}/war/META-INF/MANIFEST.MF'>
<!--
IMPORTANT! Exclude the WEB-INF/web.xml file from
the WAR as it already gets included via the webxml attribute above
-->

    <fileset dir="${SRC_DIR}/war" >
        <patternset >
            <include name="WEB-INF/taglibs/adk.tld" />
            <include name="WEB-INF/weblogic.xml"/>
            <include name="**/*.html"/>
            <include name="**/*.gif"/>
        </patternset>
    </fileset>

<!--
IMPORTANT! Include the ADK design time framework into the
adapter's design time Web application.
-->

    <fileset dir="${WLI_HOME}/adapters/src/war" >
        <patternset >
            <include name="**/*.css"/>
            <include name="**/*.html"/>
            <include name="**/*.gif"/>
            <include name="**/*.js"/>
        </patternset>
    </fileset>

<!-- Include classes from the adapter that support the design
time UI -->

    <classes dir='${SRC_DIR}' includes='sample/web/*.class'/>
    <classes dir='${SRC_DIR}/war' includes='**/*.class'/>
    <classes dir='${WLI_HOME}/adapters/src/war' includes=
        '**/*.class'/>

<!--
Include all JARs required by the Web application under the
WEB-INF/lib directory of the WAR file that are not shared in the
EAR
-->
```

All JAR files needed by the Web application are included in the `<lib>` component of the `build.xml` file.

Listing 6-12 Including JAR Files Needed by Web Application

```
<lib dir='${WLI_LIB_DIR}' includes='adk-web.jar,
    webtoolkit.jar,wlai-client.jar' />
```

11. The EAR file is created. This listing also shows all common or shared JAR files.

Listing 6-13 Including the EAR File

```
<target name='ear' depends='eis_jar,rar,war'>
  <delete file='${LIB_DIR}/${EAR_FILE}' />
  <jar jarfile="${LIB_DIR}/shared.jar">
    <zipfileset src="${LIB_DIR}/${JAR_FILE}">
      <exclude name="META-INF/MANIFEST.MF" />
    </zipfileset>
    <zipfileset src="${WLI_LIB_DIR}/adk.jar">
      <exclude name="META-INF/MANIFEST.MF" />
    </zipfileset>
    <zipfileset src="${WLI_LIB_DIR}/adk-eventgenerator.jar">
      <exclude name="META-INF/MANIFEST.MF" />
    </zipfileset>
    <zipfileset src="${WLI_LIB_DIR}/wlai-core.jar">
      <exclude name="META-INF/MANIFEST.MF" />
    </zipfileset>
    <zipfileset src="${WLI_LIB_DIR}/wlai-client.jar">
      <exclude name="META-INF/MANIFEST.MF" />
    </zipfileset>
  </jar>
  <jar jarfile='${LIB_DIR}/${EAR_FILE}'>
    <fileset dir='${basedir}' includes='version_info.xml' />
    <fileset dir='${SRC_DIR}/ear' includes=
      'META-INF/application.xml' />
    <fileset dir='${LIB_DIR}' includes='shared.jar,
      ${RAR_FILE}, ${WAR_FILE}' />
    <fileset dir='${EIS_JAR_PATH}' />
  </jar>
  <delete file='${LIB_DIR}/${WAR_FILE}' />
```

```
<delete file='${LIB_DIR}/${RAR_FILE}' />
<delete file='${LIB_DIR}/${JAR_FILE}' />

<!--
Need to keep this out of the EAR and on the system classpath until we get
WLS Connector to fix the RAR ClassLoader so logical RAR's can see all
the classes/resources the base RAR can. Currently, the logical RAR can
only see things in the BASE RAR's archive or on its Manifest Class-Path
-->

<copy file="${LIB_DIR}/${EIS_JAR_PATH}" toFile="${LIB_DIR}/
  ${EIS_JAR_FILE}" />
<delete file="${LIB_DIR}/${EIS_JAR_PATH}" />
<delete dir="${LIB_DIR}/APP-INF/lib" />
<delete dir="${LIB_DIR}/APP-INF" />
<delete file="${LIB_DIR}/shared.jar" />
</target>

<jar jarfile='${LIB_DIR}/${EAR_FILE}'>
  <fileset dir='${basedir}' includes='version_info.xml' />
  <fileset dir='${SRC_DIR}/ear' includes='META-INF/application.xml' />
  <fileset dir='${LIB_DIR}' includes='shared.jar,${RAR_FILE},
    ${WAR_FILE}' />
  <fileset dir='${EIS_JAR_PATH}' />
</jar>
```

12. All the Java source files for this project are compiled.

Listing 6-14 Compiling Java Source

```
<target name="packages">
  <echo message="Building ${ant.project.name}..." />
<!--
Generate web descriptors. NOTE: You can turn this off if you want to
tightly control your web.xml/weblogic.xml. In this case, simply maintain
these files in your src/war/WEB-INF instead of web-gen.properties
-->

  <ant dir="${WLI_HOME}/adapters/utils/ant"
    target="generate_web_descriptors" inheritAll="false">
    <property name="web_gen_props_file"
      value="${SRC_DIR}/war/WEB-INF/web-gen.properties" />
  </ant>
```

```

<!--
Merge the ADK.properties file and your adapter-specific properties into
the final properties file that will be used by the adapter
-->

    <ant dir="${WLI_HOME}/adapters/utils/ant" target="merge_properties"
        inheritAll="false">
        <property name="props_dir" value="${SRC_DIR}" />
        <property name="adapter_props_file"
            value="BEA_WLS_SAMPLE_ADK-base.properties" />
        <property name="target_props_file"
            value="BEA_WLS_SAMPLE_ADK.properties" />
    </ant>

- <!--
Compile the java source files for the adapter
-->

    <javac deprecation="true" debug="true">
    <classpath refid="CLASSPATH" />
    <src path="${SRC_DIR}" />
    <include name="**/*.java" />
    <exclude name="sample/event/OfflineEventGeneratorTestCase.java" />
    <exclude name="war/jsp_servlet/**" />
    </javac>
</target>

```

13. The Javadoc is generated.

Listing 6-15 Generating Javadoc

```

<target name='apidoc'>
    <mkdir dir='${DOC_DIR}' />
    <javadoc sourcepath='${SRC_DIR}'
        destdir='${DOC_DIR}'
        packagenames='sample.*'
        author='true'
        version='true'
        use='true'
        overview='${SRC_DIR}/overview.html'
        windowtitle='WebLogic BEA_WLS_SAMPLE_ADK Adapter
            API Documentation'
        doctitle='WebLogic BEA_WLS_SAMPLE_ADK Adapter
            API Documentation'
        header='WebLogic BEA_WLS_SAMPLE_ADK Adapter'
        bottom='Built using the WebLogic Adapter'
    </javadoc>
</target>

```

```

        Development Kit (ADK) '>
        <classpath refid='CLASSPATH' />
    </javadoc>
</target>

```

14. The targets that clean the files created by their counterparts are listed.

Listing 6-16 Including Cleanup Code

```

<target name='clean' depends='clean_release' />
<target name='clean_release' depends='clean_all, clean_apidoc' />
<target name='clean_all' depends='clean_ear, clean_rar, clean_war,
    clean_test' />
<target name='clean_test'>
    <delete file='${basedir}/BEA_WLS_SAMPLE_ADK.log' />
    <delete file='${basedir}/mcf.ser' />
</target>
<target name='clean_ear' depends='clean_jar'>
    <delete file='${LIB_DIR}/${EAR_FILE}' />
</target>
<target name='clean_rar' depends='clean_jar'>
    <delete file='${LIB_DIR}/${RAR_FILE}' />
</target>
<target name='clean_war' depends='clean_jar'>
    <delete file='${LIB_DIR}/${WAR_FILE}' />
    <delete dir='${SRC_DIR}/war/jsp_servlet' />
</target>
<target name='clean_jar' depends='clean_packages, clean_version_
    info'>
    <delete file='${LIB_DIR}/${JAR_FILE}' />
</target>
<target name='clean_version_info'>
    <delete file='${basedir}/version_info.xml' />
</target>
<target name='clean_packages'>
    <delete>
        <fileset dir='${SRC_DIR}' includes='**/*.class' />
    </delete>
</target>
<target name='clean_apidoc'>
    <delete dir='${DOC_DIR}' />
</target>
</project>

```

Step 2d: Create the Message Bundle

Any message destined for an end-user should be placed in a *message bundle*: a `.properties` text file containing *key=value* pairs that allow you to generate messages in more than one natural language. When a locale and a language are specified at run time, the contents of a message are interpreted in accordance with the relevant *key=value* pairs, and the message is presented to the user in the language appropriate for his or her locale.

The message bundle is generally located at the root of the `src` tree,

`WLI_HOME/adapters/adapter/src/bundle_name-base.properties`, where *bundle_name* is the adapter logical name. Remember that this base message bundle is merged with the `ADK.properties` bundle (in `WLI_HOME/adapters/src/ADK.properties`) to form the final `bundle_name.properties` file that is loaded at runtime. This allows ADK defined properties to be placed in the final bundle without having to physically copy or maintain them in the adapter's message bundle file. This allows for updating or upgrading these properties as new WebLogic Integration releases are available and new locales are supported.

For instructions on creating a message bundle, see the JavaSoft tutorial on internationalization at:

<http://java.sun.com/docs/books/tutorial/i18n/index.html>

Step 3: Implement the SPI

The Service Provider Interface (SPI) contains the objects that provide and manage connectivity to the EIS, establish transaction demarcation, and provide a framework for service invocation. All J2EE-compliant adapters must provide an implementation for these interfaces in the `javax.resource.spi` package.

This section contains descriptions of the interfaces you can use to implement the SPI. A minimum of three interfaces are necessary to complete the task (see “[Basic SPI Implementation](#)” on [page 6-24](#)). Each interface is described in detail, followed by a discussion of how it is extended in the sample adapter included with the ADK.

First, we describe the three required interfaces. Then we describe the additional interfaces in detail, and discuss why you might use them and how they can be beneficial when used in an adapter.

Basic SPI Implementation

To implement the SPI for your adapter, you must extend *at least* the following three interfaces:

- `ManagedConnectionFactory`, which supports connection pooling by providing methods for matching and creating a `ManagedConnection` instance.
- `ManagedConnection`, which represents a physical connection to the underlying EIS
- `ManagedConnectionMetaData`, which provides information about the underlying EIS instance associated with a `ManagedConnection` instance

Ideally, these interfaces are implemented in the order specified here.

In addition to these three interfaces, you can implement any of the other interfaces described in this step, as your adapter needs dictate.

ManagedConnectionFactory

`javax.resource.spi.ManagedConnectionFactory`

The `ManagedConnectionFactory` interface is a factory of both `ManagedConnection` and EIS-specific connection factory instances. This interface supports connection pooling by providing methods for matching and creating a `ManagedConnection` instance.

Transaction Demarcation

A critical component of the `ManagedConnectionFactory` interface is transaction demarcation. You must be able to determine which statements in your program are included in a single transaction. J2EE defines a transaction management contract between an application server and an adapter (and its underlying resource manager). The transaction management contract has two parts. The contract differs, depending on the type of transaction for which it is used. There are two types of transactions:

- XA-compliant transactions
- Local transactions

XA-Compliant Transaction

In a distributed transaction processing (DTP) environment, a `javax.transaction.xa.XAResource`-based contract is established between a transaction manager and a resource manager. A JDBC driver or a JMS provider implements this interface to

support the association between a global transaction and a database or message service connection.

The `XAResource` interface can be supported by any transactional resource that is intended for use by application programs in an environment in which transactions are controlled by an external transaction manager.

An example of such a resource is a database management system set up in such a way that an application accesses data through multiple database connections. Each database connection is enlisted with the transaction manager as a transactional resource. The transaction manager obtains an `XAResource` for each connection participating in a global transaction. The transaction manager uses the `start()` method to associate the global transaction with the resource; it uses the `end()` method to disassociate the transaction from the resource. The resource manager associates the global transaction with all work performed on its data between invocations of the `start()` and `end()` methods.

At transaction commit time, the resource managers are instructed, by the transaction manager, to prepare, commit, or roll back a transaction, according to the two-phase commit protocol.

Local Transaction

When an adapter implements the `javax.resource.spi.LocalTransaction` interface to support local transactions that are performed on the underlying resource manager, a local transaction management contract is established. This contract enables an application server to provide the infrastructure and run-time environment for transaction management. Application components rely on this transaction infrastructure to support the component-level transaction model that they use.

For more information about transaction demarcation support, enter the following URL:

[http://java.sun.com/blueprints/guidelines/
designing_enterprise_applications/transaction_management/
platform/index.html](http://java.sun.com/blueprints/guidelines/designing_enterprise_applications/transaction_management/platform/index.html)

ADK Implementations

The ADK provides an abstract foundation for an adapter called the `AbstractManagedConnectionFactory`. This foundation provides the following features:

- Basic support for internationalization and localization of exception and log messages for an adapter
- Hooks into the logging toolkit

- Getter and setter methods for standard connection properties (username, password, server, connectionURL, and port)
- Access to adapter metadata gathered from a `java.util.ResourceBundle` for an adapter
- Support for the ability to plug license checking into the initialization process for the factory. If license verification fails, client applications cannot get a connection to the underlying EIS, which makes the adapter useless.
- State verification checking to support JavaBeans-style post-constructor initialization

You must provide your own implementations for the following key methods:

- `createConnectionFactory()`
- `createManagedConnection()`
- `checkState()`
- `equals()`
- `hashCode()`
- `matchManagedConnections()`

The following sections describe these methods.

`createConnectionFactory()`

`createConnectionFactory()` is the factory for application-level connection handles for the adapter. In other words, clients of your adapter will use the object returned by this method to obtain a connection handle to the EIS.

If the adapter supports a CCI interface, we recommend that you return an instance of `com.bea.adapter.cci.ConnectionFactoryImpl` or an extension of this class. The key to implementing this method correctly is to propagate the `ConnectionManager`, `LogContext`, and `ResourceAdapterMetaData` into the client API.

Listing 6-17 `createConnectionFactory()` Example

```
protected Object
    createConnectionFactory(ConnectionManager connectionManager,
                           String strAdapterName,
                           String strAdapterDescription,
```

```

        String strAdapterVersion,
        String strVendorName)
    throws ResourceException

```

createManagedConnection()

`createManagedConnection()` is used to construct a `ManagedConnection` instance for your adapter. The following listing shows an example of this method.

Listing 6-18 `createManagedConnection()` Example

```

public ManagedConnection
    createManagedConnection(Subject subject, ConnectionRequestInfo
        info)
    throws ResourceException

```

The `ManagedConnection` instance encapsulates the expensive resources needed to communicate with the EIS. This method is called by the `ConnectionManager` when it determines that a new `ManagedConnection` is required to satisfy a client's request. A common design pattern used in adapters is to open the resources needed to communicate with the EIS in this method and then pass the resources into a new `ManagedConnection` instance.

checkState()

The `checkState()` method is called by the `AbstractManagedConnectionFactory` before it attempts to perform any factory responsibilities. Use this method to verify that all members that need to be initialized before the `ManagedConnectionFactory` can perform its SPI responsibilities have been initialized correctly.

Implement this method as follows:

```
protected boolean checkState()
```

equals()

The `equals()` method tests the object argument for equality. It is important to implement this method correctly because it is used by the `ConnectionManager` for managing the connection pools. This method should include all important members in its equality comparison.

Implement this method as follows:

```
public boolean equals(Object obj)
```

hashCode()

The `hashCode()` method provides a hash code for the factory. It is also used by the `ConnectionManager` for managing the connection pools. Consequently, this method should generate a `hashCode` based on properties that determine the uniqueness of the object.

Implement this method as follows:

```
public int hashCode()
```

matchManagedConnections()

The `ManagedConnectionFactory` must supply an implementation of the `matchManagedConnections()` method. The `AbstractManagedConnectionFactory` provides an implementation of the `matchManagedConnections()` method that relies on the `compareCredentials()` method of `AbstractManagedConnection`.

To provide logic that can match managed connections, you must override the `compareCredentials()` method provided by the `AbstractManagedConnection` class. This method is invoked when the `ManagedConnectionFactory` attempts to match a connection with a connection request for the `ConnectionManager`.

Currently, the `AbstractManagedConnectionFactory` implementation extracts a `PasswordCredential` from the `Subject/ConnectionRequestInfo` parameters that are supplied. If both parameters are null, this method returns true because it has already been established that the `ManagedConnectionFactory` for this instance is correct. This implementation is shown in the following listing.

Listing 6-19 compareCredentials() Implementation

```
public boolean compareCredentials(Subject subject,
                                 ConnectionRequestInfo info)
    throws ResourceException
{
    ILogger logger = getLogger();
```

Next, you must extract a `PasswordCredential` from either the JAAS Subject or the SPI `ConnectionRequestInfo` using the ADK's `ManagedConnectionFactory`. An example is shown in the following listing.

Listing 6-20 Extracting a PasswordCredential

```

PasswordCredential pc = getFactory().
getPasswordCredential(subject, info);
    if (pc == null)
    {
        logger.debug(this.toString() + ": compareCredentials

```

In the previous listing, JAAS Subject and `ConnectionRequestInfo` are null, which means that a match is assumed. This method is not invoked unless it has already been established that the factory for this instance is correct. Consequently, if the Subject and `ConnectionRequestInfo` are both null, then the credentials match by default. Therefore, the result of pinging this connection determines the outcome of the comparison. The following listing shows how to ping the connection programmatically.

Listing 6-21 Pinging a Connection

```

return ping();
}

    boolean bUserNameMatch = true;
    String strPcUserName = pc.getUserName();
        if (m_strUserName != null)
    {

logger.debug(this.toString() + ": compareCredentials >>> comparing
    my username ["+m_strUserName+"] with client username
        ["+strPcUserName+"]");

```

Next, you need to check whether the user specified in either the `Subject` or `ConnectionRequestInfo` is the same as our user. We do not support reauthentication in this adapter, so if the usernames do not match, this instance cannot satisfy the request. The following code satisfies the request:

```
bUserNameMatch = m_strUserName.equals(strPcUserName);
```

If the usernames match, ping the connection to determine whether it is still good. If the names do not match, there is no reason to ping.

To ping the connection, use the following code:

```
return bUserNameMatch ? ping() : false;
```

Explanation of the Implementation

In a managed scenario, the application server invokes the `matchManagedConnections()` method on the `ManagedConnectionFactory` for an adapter. The specification does not indicate how the application server determines which `ManagedConnectionFactory` to use to satisfy a connection request. The ADK's `AbstractManagedConnectionFactory` implements `matchManagedConnections()`.

The first step in this implementation is to compare “this” (that is, the `ManagedConnectionFactory` instance on which the `ConnectionManager` invoked `matchManagedConnections()`) to the `ManagedConnectionFactory` on each `ManagedConnection` in the set supplied by the application server. For each `ManagedConnection` in the set that has the same `ManagedConnectionFactory`, the implementation invokes the `compareCredentials()` method. This method allows each `ManagedConnection` object to determine whether it can satisfy the request.

`matchManagedConnections()` is called by the `ConnectionManager` (as shown in [Listing 6-22](#)) to search for a valid connection in the pool it is managing. If this method returns null, then the `ConnectionManager` allocates a new connection to the EIS via a call to `createManagedConnection()`.

Listing 6-22 `matchManagedConnections()` Method Implementation

```
public ManagedConnection
matchManagedConnections(Set connectionSet,
                        Subject subject,
                        ConnectionRequestInfo info)
    throws ResourceException
```

This class uses the following approach to match a connection:

1. For each object in the set, it iterates over the appropriate `connectionSet` until a match is found. Then it determines whether the object is an `AbstractManagedConnection` class.
2. If it is, this connection is compared to the `ManagedConnectionFactory` for the `AbstractManagedConnection` from the set.
3. If the factories are equal, then the `compareCredentials()` method is invoked on the `AbstractManagedConnection`.
4. If the `compareCredentials()` method returns `true`, then the instance is returned.

AbstractManagedConnectionFactory Properties Required at Deployment

To use the base implementation of `AbstractManagedConnectionFactory`, you must, at deployment time, provide the properties described in the following table.

Table 6-2 AbstractManagedConnectionFactory Properties

Property Name	Property Type	Applicable Values	Description	Default
<code>LogLevel</code>	<code>java.lang.String</code>	ERROR, WARN, INFO, DEBUG	Logs verbosity level	WARN
<code>LanguageCode</code>	<code>java.lang.String</code>	For a valid ISO language code, see http://ftp.ics.uci.edu/pub/ietf/http/related/iso639.txt .	Determines the desired locale for log messages	en
<code>CountryCode</code>	<code>java.lang.String</code>	For a valid ISO country code, see http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html .	Determines the desired locale for log messages	US
<code>MessageBundleBase</code>	<code>java.lang.String</code>	Any valid Java class name or filename	Determines the message bundle for log messages	None, required

Table 6-2 AbstractManagedConnectionFactory Properties (Continued)

Property Name	Property Type	Applicable Values	Description	Default
LogConfigFile	java.lang. String	Any valid filename	Configures the LOG4J system	None, required
RootLogContext	java.lang. String	Any valid Java string	Categorizes log messages from this connection factory	None, required
AdditionalLog Context	java.lang. String	Any valid Java string	Adds additional information to uniquely identify messages from this factory	None, optional

Other Key ManagedConnectionFactory Features in the ADK

The ADK sample adapter provides a class called `sample.spi.ManagedConnectionFactoryImpl` that extends `AbstractManagedConnectionFactory`. Use this class as an example of how to extend the ADK's base class.

For a complete code listing of an implementation of the sample adapter called `ManagedConnectionFactory`, see:

```
WLI_HOME/adapters/sample/src/sample/spi/  
ManagedConnectionFactoryImpl.java
```

ManagedConnection

javax.resource.spi.ManagedConnection

The `ManagedConnection` object is responsible for encapsulating all the expensive resources needed to establish connectivity to the EIS. A `ManagedConnection` instance represents a physical connection to the underlying EIS. `ManagedConnection` objects are pooled by the application server in a managed environment.

ADK Implementation

The ADK provides an abstract implementation of `ManagedConnection`. The base class provides logic for managing connection event listeners and multiple application-level connection handles for each instance of `ManagedConnection`.

When implementing the `ManagedConnection` interface, you need to determine the transaction demarcation support provided by the underlying EIS. For more information about transaction demarcation, see [“Transaction Demarcation” on page 6-24](#).

The ADK provides `AbstractManagedConnection`, an abstract implementation for the `javax.resource.spi.ManagedConnection` interface that:

- Provides access to the ADK logging framework
- Manages a collection of connection event listeners
- Provides convenience methods for notifying all connection event listeners of connection-related events
- Simplifies the cleanup and destruction of a `ManagedConnection` instance

The sample adapter provided with the ADK includes `ManagedConnectionImpl`, which extends `AbstractManagedConnection`. For a complete code listing for a sample adapter called `ManagedConnection`, see:

```
WLI_HOME/adapters/sample/src/sample/spi/  
ManagedConnectionFactoryImpl.java
```

ManagedConnectionMetaData

`javax.resource.spi.ManagedConnectionMetaData`

The `ManagedConnectionMetaData` interface provides information about the underlying EIS instance associated with a `ManagedConnection` instance. An application server uses this information to get run-time information about a connected EIS instance.

ADK Implementation

The ADK provides `AbstractManagedConnectionMetaData`, an abstract implementation of the `javax.resource.spi.ManagedConnectionMetaData` and `javax.resource.cci.ConnectionMetaData` interfaces that:

- Simplifies exception handling

- Provides access to an `AbstractManagedConnection` instance
- Allows you to focus on implementing EIS-specific logic
- Makes it unnecessary for you to have separate metadata classes for the CCI and SPI implementations

The sample adapter provided with the ADK includes `ConnectionMetaDataImpl`, which extends `AbstractManagedConnectionMetaData`. For the complete code listing for the adapter, see:

```
WLI_HOME/adapters/sample/src/sample/spi/ConnectionMetaDataImpl.java
```

ConnectionEventListener

`javax.resource.spi.ConnectionEventListener`

The `ConnectionEventListener` interface provides an event callback mechanism that enables an application server to receive notifications from a `ManagedConnection` instance.

ADK Implementation

The ADK provides two concrete implementations of `ConnectionEventListener`:

- `com.bea.adapter.spi.ConnectionEventLogger`, which logs connection-related events to the adapter's log by using the ADK logging framework.
- `com.bea.adapter.spi.NonManagedConnectionEventListener`, which destroys `javax.resource.spi.ManagedConnection` instances when the adapter is running in an unmanaged environment. This implementation:
 - Logs connection-related events using the ADK logging framework
 - Destroys `ManagedConnection` instances when a connection-related error occurs

In most cases, the implementations provided by the ADK are sufficient; you should not need to provide your own implementation of this interface.

ConnectionManager

`javax.resource.spi.ConnectionManager`

The `ConnectionManager` interface provides a hook that can be used by the adapter to pass a connection request to the application server.

ADK Implementation

The ADK provides a concrete implementation of this interface:

`com.bea.adapter.spi.NonManagedConnectionManager`. This implementation provides a basic connection manager for adapters running in an unmanaged environment. In a managed environment, this interface is provided by the application server. In most cases, you can use the implementation provided by the ADK.

`NonManagedConnectionManager` is a concrete implementation of the `javax.resource.spi.ConnectionManager` interface. It serves as the `ConnectionManager` in the unmanaged scenario for an adapter; it does not provide any connection pooling or any other quality of service.

ConnectionRequestInfo

`javax.resource.spi.ConnectionRequestInfo`

The `ConnectionRequestInfo` interface enables an adapter to pass its own request-specific data structure across a connection request flow. An adapter extends the empty interface to support its own data structures for a connection request.

ADK Implementation

The ADK provides a concrete implementation of the

`javax.resource.spi.ConnectionRequestInfo` interface. This interface is called `ConnectionRequestInfoMap`. It provides a `java.util.Map` interface to information requested when a connection is being established, such as username and password.

LocalTransaction

`javax.resource.spi.LocalTransaction`

The `LocalTransaction` interface provides support for transactions that are managed within an EIS resource manager, and do not require an external transaction manager.

ADK Implementation

The ADK provides an abstract implementation of this interface called

`AbstractLocalTransaction`, thus allowing you to focus on implementing the EIS-specific aspects of a `LocalTransaction`. Specifically, it:

- Simplifies exception handling

- Allows adapter providers to focus on implementing EIS-specific transaction logic
- Makes it unnecessary to have separate metadata classes for the CCI and SPI implementations

Step 4: Implement the CCI

The client interface allows a J2EE-compliant application to access back-end systems. The client interface manages the flow of data between the client application and the back-end system; it does not have any visibility into what either the container or the application server are doing with the adapter. The client interface specifies the format of both the request records and the response records for a given interaction with the EIS.

First, you must determine whether your adapter must support the J2EE-compliant Common Client Interface (CCI). Although not required by the current J2EE specification, the CCI is likely to be required in a later version. Consequently, the ADK focuses on helping you implement a CCI interface for your adapter.

How to Use This Section

This section (“[Step 4: Implement the CCI](#)”) describes some of the interfaces you can use to implement the CCI. At a minimum, two interfaces are necessary to complete the task. (See “[Basic CCI Implementation](#)” on [page 6-36](#).) Each interface is described in detail, followed by a discussion of how it is extended in the sample adapter included with the ADK.

Following the description of the two required interfaces, detailed descriptions of the additional interfaces are provided, along with a discussion of reasons why you might use these interfaces and the benefits they provide.

Basic CCI Implementation

To implement the CCI for your adapter, you need to extend *at least* the following two interfaces:

- [Connection](#), which represents an application-level handle that is used by a client to access the underlying physical connection
- [Interaction](#), which enables a component to execute EIS functions

If possible, implement these interfaces in the order specified here.

In addition, you can implement any of the following interfaces needed for your adapter:

- [ConnectionFactory](#)

- `ConnectionMetaData`
- `ConnectionSpec`
- `InteractionSpec`
- `LocalTransaction`
- `Record`
- `ResourceAdapterMetaData`

Connection

`javax.resource.cci.Connection`

A `Connection` represents an application-level handle that is used by a client to access the underlying physical connection. The actual physical connection associated with a `Connection` instance is represented by a `ManagedConnection` instance.

A client gets a `Connection` instance by using the `getConnection()` method on a `ConnectionFactory` instance. A `Connection` can be associated with zero or more `Interaction` instances.

ADK Implementation

The ADK provides an abstract implementation of this interface called `AbstractConnection`. This implementation provides the following functionality:

- Access to the ADK logging framework
- Access to an `AbstractManagedConnection` instance
- State management and assertion checking

You must extend this class by providing an implementation for:

```
public Interaction createInteraction()
    throws ResourceException
```

This method creates an interaction associated with this connection. An interaction enables an application to execute EIS functions. This method:

- Returns an `Interaction` instance
- Throws a `ResourceException` if the create operation fails

Interaction

javax.resource.cci.Interaction

The `javax.resource.cci.Interaction` enables a component to execute EIS functions. An `Interaction` instance supports the following ways of interacting with an EIS instance:

- An `execute()` method may take an input `Record`, an output `Record`, and an `InteractionSpec`. This method executes the EIS function represented by the `InteractionSpec` and updates the output `Record`.
- An `execute()` method may take an input `Record` and an `InteractionSpec`. This method implementation executes the EIS function represented by the `InteractionSpec` and produces the output `Record` as a return value.

An `Interaction` instance is created from a connection and is required to maintain the association between the `Interaction` and the `Connection` instances. The `close()` method releases all resources maintained by the adapter for the interaction. The close of an `Interaction` instance should not trigger the close of the associated `Connection` instance.

ADK Implementation

The ADK provides an implementation of this interface called `AbstractInteraction`. This implementation:

- Provides access to the ADK logging framework
- Manages warnings

You must supply a concrete extension to `AbstractInteraction` that implements `execute()`. Two versions of `execute()` are available. They are described in the following sections.

`execute()` Version 1

The `execute()` method declared in [Listing 6-23](#) shows an interaction represented by `InteractionSpec`.

Listing 6-23 Example of `execute()` Version 1

```
public boolean execute(InteractionSpec ispec,
                      Record input,
                      Record output)
    throws ResourceException
```

When invoked in this way, `execute()` takes an input record and updates the output record. It returns the following:

- Returns `true` if execution of the EIS function is successful and the output (`Record`) has been updated; otherwise it returns `false`.
- Throws `ResourceException` if the `execute` operation fails.

The parameters for `execute()` version 1 are described in the following table.

Table 6-3 Parameters for `execute()` Version 1

Parameter	Description
<code>ispec</code>	<code>InteractionSpec</code> representing a target EIS data or function module
<code>input</code>	Input record
<code>output</code>	Output record

`execute()` Version 2

The `execute()` method declared in [Listing 6-24](#) also executes an `Interaction` represented by `InteractionSpec`.

Listing 6-24 Example of `execute()` Version 2

```
public Record execute(InteractionSpec ispec,
                    Record input)
    throws ResourceException
```

When invoked in this way, `execute()` takes an input `Record` and, if the execution of `Interaction` is successful, it returns an output `Record`.

This method:

- Returns an output `Record` if execution of the EIS function has been successful; otherwise it throws an exception.

- Throws `ResourceException` if the `execute` operation fails.

If an exception occurs, this method notifies its `Connection`, which takes the appropriate action, including closing itself.

The parameters for `execute()` version 2 are listed in the following table.

Table 6-4 Parameters for `execute()` Version 2

Parameter	Description
<code>ispec</code>	<code>InteractionSpec</code> representing a target EIS data or function module
<code>input</code>	Input record

Using XCCI to Implement the CCI

XML-CCI is a dialect of the Client Connector Interface, in which XML-based record formats are used to represent data. These formats are supported by a framework and tools. XML-CCI is usually referred to by its abbreviation: XCCI.

XCCI is made up of two components: Services and DocumentRecords.

Services

A service represents functionality available in an EIS. It includes four components:

- Unique business name

Every service has a unique business name that indicates its role in an integration solution. For example, in an integration solution involving a Customer Relationship Management (CRM) system, you may have a service named `CreateNewCustomer`. It is important to give a service a name that reflects the business purpose of the service; it is an abstraction of the name of the functions invoked by your service in the EIS.

- Request document definition

The request document definition describes the input requirements for a service. The `com.bea.document.IDocumentDefinition` interface embodies all the metadata about a document type. It includes the document schema (structure and usage), and the root element name for all documents of this type. The root element name is needed because an XML schema can define more than one root element.

- Response document definition

A response document definition describes the output of a service.

- Additional metadata

A service is a higher-order component in an integration solution that hides most of the complexity involved in executing functionality in an EIS. In other words, a service does not expose many of the details required to interact with the EIS in its public interface. As a result, some of the information required to invoke a function in an EIS is not provided by the client in the request. Consequently, most services need to store additional metadata. In WebLogic Integration, this additional metadata is encapsulated by an adapter's `javax.resource.cci.InteractionSpec` implementation class.

To indicate that a given service does not require request or response data, create an empty or null `IDocumentDefinition` for the request or response in your `DesignTimeRequestHandler`. You may also set the `IDocumentDescriptor` for the request or response on the `IServiceDescriptor` for the service with an empty or null `IDocumentDescriptor` instance. Create empty or null `IDocumentDefinition` instances using the static `DocumentFactory.createNullDocumentDefinition()` method, and empty or null `IDocumentDescriptor` instances by using the static `DescriptorFactory.createNullDocumentDescriptor()` method.

If you choose to use empty or null document definitions or descriptors in the generated `IServiceDescriptor` or `IApplicationDescriptor` generated by the adapter at design-time, you must ensure that the null request or response documents for these services are handled at runtime. In other words, an adapter that uses empty or null document descriptors must not assume a request or response document is non-null at runtime.

The Application View runtime engine ensures that services requiring a request or response receive non-null request or response documents, and ensures that services not requiring a request or response receive null request or response documents.

DocumentRecord

`com.bea.connector.DocumentRecord`

At run time, the XCCI layer expects `DocumentRecord` objects as input to a service and returns `DocumentRecord` objects as output from a service. `DocumentRecord` implements the `javax.resource.cci.Record` and the `com.bea.document.IDocument` interfaces. For a description of the `Record` interface, see [“Record” on page 6-51](#).

`IDocument`, which facilitates XML input and output from the CCI layer in an adapter, is described in the following section.

IDocument

`com.bea.document.IDocument`

An `IDocument` is a higher-order wrapper around the W3C Document Object Model (DOM). The most important value added by the `IDocument` interface is an XPath interface to elements in an XML document. In other words, `IDocument` objects can be queried and updated using XPath strings. For example, the XML document shown in [Listing 6-25](#) shows how XML is used to record details about a person named *Bob*.

Listing 6-25 XML Example

```
<Person name="Bob">
  <Home squareFeet="2000"/>
  <Family>
    <Child name="Jimmy">
      <Stats sex="male" hair="brown" eyes="blue"/>
    </Child>
    <Child name="Susie">
      <Stats sex="female" hair="blonde" eyes="brown"/>
    </Child>
  </Family>
</Person>
```

By using `IDocument`, you can retrieve Jimmy's hair color using the XPath code shown in [Listing 6-26](#).

Listing 6-26 Sample Code for Retrieving IDocument Data

```
System.out.println("Jimmy's hair color: " +
  person.getStringFrom("//Person[@name=\"Bob\"]/Family/Child
  [@name=\"Jimmy\"]/Stats/@hair");
```

On the other hand, if DOM is used, you must use the code shown in [Listing 6-27](#) to submit a query.

Listing 6-27 Sample Code for Retrieving DOM Data

```
String strJimmysHairColor = null;
org.w3c.dom.Element root = doc.getDocumentElement();
if (root.getTagName().equals("Person") &&
    root.getAttribute("name").equals("Bob") {
    org.w3c.dom.NodeList list = root.
        getElementsByTagName("Family");
    if (list.getLength() > 0) {
        org.w3c.dom.Element family = (org.w3c.dom.
            Element)list.item(0);

        org.w3c.dom.NodeList childList = family.getElementsByTagName("Child");
        for (int i=0; i < childList.getLength(); i++) {
            org.w3c.dom.Element child = childList.item(i);
            if (child.getAttribute("name").equals("Jimmy")) {
                org.w3c.dom.NodeList statsList =
                    child.getElementsByTagName("Stats");
                if (statsList.getLength() > 0) {
                    org.w3c.dom.Element stats = statsList.item(0);
                    strJimmysHairColor = stats.getAttribute("hair");
                }
            }
        }
    }
}
```

As you can see, IDocument enables you to simplify your code.

Proper Use of Namespaces in IDocument Instances

If you pass in XML instance documents within an `IDocument` instance, do not use any namespace or prefix within the instance document text, unless the adapter or consumer of the `IDocument` instance explicitly uses qualified XPath statements to query the data.

Failure to observe this rule will cause runtime failures in code using `IDocument`.

Also, because the sample adapters and most clients using `IDocument` do not use namespace prefixes to qualify the steps in XPath expressions, use of XML namespace declarations on XML documents represented as `IDocument` instances should be avoided.

For example, an XML namespace declaration is included on the XML instance document for an Application View service. The DBMS sample adapter uses the `IDocument` interface to retrieve the request data fields. `IDocument`, because of the underlying processor, cannot retrieve data fields in an XML instance document with a default XML namespace using XPaths with unqualified steps.

The result is that the DBMS Sample adapter (or any code using `IDocument` or the Xalan XPath to get data fields from an XML instance) does not get the proper field data from the request document.

Given the following document:

```
<Input>
  <FirstName>Joe</FirstName>
</Input>
```

The call `IDocument.getDocumentData("/Input/FirstName")` returns "Joe".

Using the following document with a default namespace:

```
<Input xmlns="my URI">
  <FirstName>Joe</FirstName>
</Input>
```

The call `IDocument.getDocumentData("/Input/FirstName")` returns "". The XPath processor does not detect any XPath step that selects data from the `my URI` namespace, just the empty namespace.

ADK-Supplied XCCI Classes

To help you implement XCCI for your adapters, the ADK provides the following classes and interfaces:

- [AbstractDocumentRecordInteraction](#)

- [DocumentDefinitionRecord](#)
- [DocumentInteractionSpecImpl](#)

This section describes those classes and interfaces.

AbstractDocumentRecordInteraction

com.bea.adapter.cci.AbstractDocumentRecordInteraction

This class extends the ADK's abstract base `Interaction`, `com.bea.adapter.cci.AbstractInteraction`. The purpose of this class is to provide convenience methods for manipulating `DocumentRecords` and to reduce the amount of error handling that you need to implement. Specifically, this class declares:

```
protected abstract boolean execute(
    InteractionSpec ixSpec,
    DocumentRecord inputDoc,
    DocumentRecord outputDoc
) throws ResourceException
```

and

```
protected abstract DocumentRecord execute(
    InteractionSpec ixSpec,
    DocumentRecord inputDoc
) throws ResourceException
```

These methods are not invoked on the concrete implementation until it has been verified that the output records are `DocumentRecord` objects.

DocumentDefinitionRecord

com.bea.adapter.cci.DocumentDefinitionRecord

This class allows the adapter to return an `IDocumentDefinition` from its `DocumentRecordInteraction` implementation. This class is useful for satisfying design-time requests to create the request and/or response document definitions for a service.

DocumentInteractionSpecImpl

com.bea.adapter.cci.DocumentInteractionSpecImpl

This class allows you to save a request document definition and response document definition for a service in the `InteractionSpec` provided to the `execute` method at run time. This capability

is useful when the `Interaction` for an adapter needs access to the XML schemas for a service at run time.

Note: `DocumentInteractionSpecImpl` also implements the `com.bea.connector.ClientDataInteractionSpec` interface. This allows it to receive `IClientData` instances containing environment variables and other client information at runtime. For more information, see [“Step 5: Enable Environment Variable Support \(Optional\)”](#) on page 6-52.

XCCI Design Pattern

A design pattern that is frequently used with the XCCI is support for the definition of services in the `Interaction` implementation. When this design pattern is used, the `javax.resource.cci.Interaction` implementation for an adapter allows a client program to retrieve metadata from the underlying EIS in order to define a WebLogic Integration service. As a result, the interaction must be able to generate the request and response XML schemas and additional metadata for a service. The `Interaction` may also allow a client program to browse a catalog of functions provided by the EIS. This approach facilitates a thin-client architecture for your adapter.

The ADK provides the `com.bea.adapter.cci.DesignTimeInteractionSpecImpl` class to help you implement this design pattern. The `sample.cci.InteractionImpl` class demonstrates how to implement this design pattern using the `DesignTimeInteractionSpecImpl` class.

Using NonXML J2EE-Compliant Adapters

The ADK provides a plug-in mechanism for using nonXML adapters with WebLogic Integration. Not all prebuilt adapters use XML as the `javax.resource.cci.Record` data type. For example, XML may not be used in the following circumstances:

- You have developed a J2EE-compliant adapter with a proprietary record format.
- You have purchased a third-party J2EE-compliant adapter that uses a proprietary record format in the adapter’s CCI layer.

To facilitate implementation of these types of adapters, the ADK provides the `com.bea.connector.IRecordTranslator` interface. At run time, the application integration engine uses an adapter’s `IRecordTranslator` implementation to translate request and response records before executing the adapter’s service.

Because the application integration engine supports only `javax.resource.cci.Record` of type `com.bea.connector.DocumentRecord`, you must translate this proprietary format to a document record for request and response records. You do not need to rewrite the adapter's CCI interaction layer. By including a class in your adapter's EAR file that implements the `IRecordTranslator` interface, the application integration engine can execute the translation methods in your translator class on each record for request and response.

There is a one-to-one correlation between an `InteractionSpec` implementation class and an `IRecordTranslator` implementation class. An adapter with more than one type of `InteractionSpec` implementation requires an `IRecordTranslator` implementation class for each. The plug-in architecture loads the translator class by name, using the full class name of the adapter's `InteractionSpec`, plus the phrase `RecordTranslator`. For example, if the name of the adapter's `InteractionSpec` class is

`com.bea.adapter.dbms.cci.InteractionSpecImpl`, then the engine loads the `com.bea.adapter.dbms.cci.InteractionSpecImplRecordTranslator` class (if the latter class is available).

For a description of the methods that must be implemented, see the Javadoc for `com.bea.connector.IRecordTranslator` at the following URL:

<http://e-docs.bea.com/wli/docs81/javadoc/adk/>

ConnectionFactory

`javax.resource.cci.ConnectionFactory`

`ConnectionFactory` provides an interface for getting a connection to an EIS instance. An implementation of the `ConnectionFactory` interface must be provided by an adapter.

The application looks up a `ConnectionFactory` instance from JNDI namespace and uses it to get EIS connections.

To support JNDI registration, `java.io.Serializable` and `javax.resource.Referenceable` interfaces must be implemented. For this purpose, an implementation class for `ConnectionFactory` is required.

ADK Implementation

The ADK provides `ConnectionFactoryImpl`, a concrete implementation of the `javax.resource.cci.ConnectionFactory` interface that provides the following functionality:

- Access to the ADK logging framework

- Access to adapter metadata
- Implementation of the `getConnection()` method

Usually you can use this class as is, without extending it.

ConnectionMetaData

`javax.resource.cci.ConnectionMetaData`

`ConnectionMetaData` provides information about an EIS instance connected through a `Connection` instance. A component calls the method `Connection.getMetaData` to get a `ConnectionMetaData` instance.

ADK Implementation

By default, the ADK provides an implementation of this class via the `com.bea.adapter.spi.AbstractConnectionMetaData` class. You must extend this abstract class and implement its four abstract methods for your adapter.

ConnectionSpec

`javax.resource.cci.ConnectionSpec`

`ConnectionSpec` is used by an application component to pass connection request-specific properties to the `ConnectionFactory.getConnection()` method.

We recommend that you implement the `ConnectionSpec` interface as a `JavaBean` so that it can support tools. Define the properties of the `ConnectionSpec` implementation class through the getter and setter methods pattern.

The CCI specification defines a set of standard properties for a `ConnectionSpec`. The properties are defined on either a derived interface or an implementation class of an empty `ConnectionSpec` interface. In addition, an adapter may define additional properties specific to its underlying EIS.

ADK Implementation

Because the `ConnectionSpec` implementation must be a `JavaBean`, the ADK does not supply an implementation for this class.

InteractionSpec

`javax.resource.cci.InteractionSpec`

An `InteractionSpec` holds properties for driving an interaction with an EIS instance. Specifically, it is used by an interaction to execute the specified function on an underlying EIS.

The CCI specification defines a set of standard properties for an `InteractionSpec`. An `InteractionSpec` implementation is not required to support a standard property if that property does not apply to the underlying EIS.

The `InteractionSpec` implementation class must provide getter and setter methods for each of its supported properties. The getter and setter methods convention should be based on the JavaBeans design pattern.

The `InteractionSpec` interface must be implemented as a `JavaBean` in order to support tools. An implementation class for the `InteractionSpec` interface is required to implement the `java.io.Serializable` interface.

The `InteractionSpec` contains information that is not in `Record` but that helps to determine which EIS function to invoke.

The standard properties are described in the following table.

Table 6-5 Standard InteractionSpec Properties

Property	Description
<code>FunctionName</code>	Name of an EIS function
<code>InteractionVerb</code>	Mode of interaction with an EIS instance: <code>SYNC_SEND</code> , <code>SYNC_SEND_RECEIVE</code> , or <code>SYNC_RECEIVE</code>
<code>ExecutionTimeout</code>	The number of milliseconds an <code>Interaction</code> waits for an EIS to execute the specified function

The following standard properties are used to give hints to an interaction instance about the `ResultSet` requirements:

- `FetchSize`
- `FetchDirection`
- `MaxFieldSize`

- `ResultSetType`
- `ResultSetConcurrency`

A CCI implementation can provide properties other than the one described in the `InteractionSpec` interface.

Note: The format and type of any additional properties are specific to an EIS; they are outside the scope of the CCI specification.

ADK Implementation

The ADK contains a concrete implementation of `javax.resource.cci.InteractionSpec` called `InteractionSpecImpl`. This interface provides a base implementation that you can extend by using getter and setter methods for the standard interaction properties described in [Table 6-5](#).

LocalTransaction

`javax.resource.cci.LocalTransaction`

The `LocalTransaction` interface is used for application-level local transaction demarcation. It defines a transaction demarcation interface for resource manager local transactions. The system contract level `LocalTransaction` interface (as defined in the `javax.resource.spi` package) is used by the container for managing local transactions.

A local transaction is managed within a resource manager. No external transaction manager is involved in the coordination of such transactions.

A CCI implementation can (but is not required to) implement the `LocalTransaction` interface. If the `LocalTransaction` interface is supported by a CCI implementation, then the method `Connection.getLocalTransaction()` should return a `LocalTransaction` instance. A component can then use the returned `LocalTransaction` to demarcate a resource manager local transaction (associated with the `Connection` instance) on the underlying EIS instance.

The `com.bea.adapter.spi.AbstractLocalTransaction` class also implements this interface.

For more information about local transactions, see [“Transaction Demarcation” on page 6-24](#).

Record

`javax.resource.cci.Record`

The `javax.resource.cci.Record` interface is the base interface for representing an input to or output from the `execute()` methods defined for an Interaction. For more information about the `execute()` methods, see [“execute\(\) Version 1” on page 6-38](#) and [“execute\(\) Version 2” on page 6-39](#).

A `MappedRecord` or `IndexedRecord` can contain another `Record`. This means that you can use `MappedRecord` and `IndexedRecord` to create a hierarchical structure of arbitrary depth. A basic Java type is used as the leaf element of a hierarchical structure represented by a `MappedRecord` or `IndexedRecord`.

The `Record` interface can be extended to form one of the representations shown in the following table.

Table 6-6 Record Interface Representations

Representation	Description
<code>MappedRecord</code>	A set of key-value pairs that represents a record. This interface is based on the <code>java.util.Map</code> .
<code>IndexedRecord</code>	An ordered and indexed collection representing a record. This interface is based on the <code>java.util.List</code> .
JavaBean-based representation of an EIS abstraction	An example is a custom record generated to represent a purchase order in an ERP system.
<code>javax.resource.cci.ResultSet</code>	This interface extends both <code>java.sql.ResultSet</code> and <code>javax.resource.cci.Record</code> . A <code>ResultSet</code> represents tabular data.

If the adapter implements a CCI interface, the next question to consider is which record format to use for a service. For each service, a format must be specified for the request records (which provide input to the service) and response records (which provide the EIS responses).

ADK Implementation

The ADK focuses on helping you implement an XML-based record format in the CCI layer. To this end, the ADK provides the `DocumentRecord` class. In addition, you can use BEA's schema toolkit to develop schemas to describe the request and response documents for a service.

The ADK provides `RecordImpl`, a concrete implementation of the `javax.resource.cci.Record` interface that provides getter and setter methods for the name and description of a record.

For an adapter provider who wants to use an XML-based record format (which is highly recommended), the ADK also provides the `com.bea.adapter.cci.AbstractDocumentRecordInteraction` class. This class ensures that the client passes `DocumentRecord` objects. In addition, this class provides convenience methods for accessing content in a `DocumentRecord`.

ResourceAdapterMetaData

`javax.resource.cci.ResourceAdapterMetaData`

The interface `javax.resource.cci.ResourceAdapterMetaData` provides information about the capabilities of an adapter implementation. A CCI client uses a `ConnectionFactory.getMetaData` to get metadata information about the adapter. The `getMetaData()` method does not require an active connection to an EIS instance. The `ResourceAdapterMetaData` interface can be extended to provide more information specific to an adapter implementation.

Note: This interface does not provide information about an EIS instance that is connected through the adapter.

ADK Implementation

The ADK provides an interface that encapsulates adapter metadata and provides getters and setters for all properties: `ResourceAdapterMetaDataImpl`.

Step 5: Enable Environment Variable Support (Optional)

This optional step is required if you allow the use of environment variables in service definitions. For more information on how environment variables are defined and what they mean to your adapter, see [“Developing a Design-Time GUI” on page 9-1](#).

At runtime, your adapter uses a set of environment variables to obtain values a system administrator or application deployer has configured for use in the target environment. Use of

variables generally involves replacing a variable reference within one or more property values for the service with a runtime variable value. What properties, where in the property value, and what the variable reference looks like are all adapter-specific.

In order to use environment variables, your adapter must obtain the set of environment variables intended for use with the currently executing service. Your adapter can obtain this set of variables in one of two ways.

- By implementing the `com.bea.connector.ClientDataInteractionSpec` interface.
- By extending the ADK's `com.bea.adapter.cci.DocumentInteractionSpecImpl` class.

Implementing ClientDataInteractionSpec

This interface provides an `IClientData` instance in the `setClientData(IClientData)` method. You obtain the variable set by calling `client_data_object.getVariableSet()` method.

Extending DocumentInteractionSpecImpl

In this case, you listen for `java.beans.PropertyChange` events in your own implementation of the `java.beans.VetoableChangeListener.vetoableChange` method. `IClientData` is obtained when the `PropertyChangeEvent.getPropertyName()` method returns `clientData` and by calling the `getOldValue()` method on the event. The returned `IClientData` instance can be used to obtain the variable set (`IVariableSet` instance) by calling the `client_data_object.getVariableSet()` method.

Once you have obtained an `IVariableSet` instance, you can retrieve variables from the set by calling `getVariable()` and use the variable's value to generate a runtime property value (by replacing any variable references in the original property value).

If desired, you can listen for changes on this variable set, by implementing the `com.bea.connector.VariableChangeListener` interface and adding an instance of your implementation class to the variable sets listener list by calling `IVariableSet.addListener(VariableChangeListener)` method. Any time the variables in the variable set change, or their values change, you will be notified by a call to `variableChange()` on your listener.

For examples of how to use environment variables at runtime, see the source code for the DBMS sample adapter located at `WLI_HOME/adapters/dbms/src`.

Step 6: Test the Adapter

To help you test your adapter, the ADK provides `com.bea.adapter.test.TestHarness`, a test harness that leverages JUnit, an open-source tool for unit testing. The `com.bea.adapter.test.TestHarness` performs the following functions:

- Reads a properties file containing test configuration information
- Initializes the logging toolkit
- Initializes JUnit TestSuite
- Loads test classes and executes them using JUnit
- Allows you to test code offline and outside WebLogic Server

You can find more information about JUnit at:

<http://www.junit.org>

Using the Test Harness

To use the test harness in the ADK, complete the following steps:

1. Create a class that extends `junit.framework.TestCase`. The class must provide a static method named `suite` that returns a new `junit.framework.TestSuite`.
2. Implement test methods. The name of each method should begin with `test`.
3. Create or alter the `test.properties` in the project directory. (If you clone the sample adapter, then your adapter will have a base `test.properties` in the project directory.) The properties file should contain any configuration properties needed for your test case.
4. Invoke the test using Ant. Your Ant `build.xml` file needs a test target that invokes the `com.bea.adapter.test.TestHarness` class with the properties file for your adapter. For example, the sample adapter uses the Ant target shown in [Listing 6-28](#).

Listing 6-28 Ant Target Specified in the Sample Adapter

```
<target name='test' depends='packages'>
  <java classname='com.bea.adapter.test.TestHarness'>
    <arg value='-DCONFIG_FILE=test.properties' /><classpath
      refid='CLASSPATH' />
  </java>
```

This target invokes the JVM with the following main class:
`com.bea.adapter.test.TestHarness`. This class uses the classpath established for the sample adapter and passes the following command-line argument:

```
-DCONFIG_FILE=test.properties
```

Test Case Extensions Provided by the ADK

The sample adapter provides two basic `TestCase` extensions:

- `sample.spi.NonManagedScenarioTestCase`
- `sample.event.OfflineEventGeneratorTestCase`

sample.spi.NonManagedScenarioTestCase

`NonManagedScenarioTestCase` allows you to test your SPI and CCI classes in a nonmanaged scenario. Specifically, this class tests the following:

- Initialization of the `ManagedConnectionFactory` implementation.
- Serialization or deserialization of the `ManagedConnectionFactory` instance.
- Opening of a connection to the EIS.
- Closing of a connection to the EIS. Make sure all associated resources are closed when a connection is closed.

sample.event.OfflineEventGeneratorTestCase

`sample.event.OfflineEventGeneratorTestCase` allows you to test the inner workings of your event generator outside WebLogic Server. Specifically, this class tests the following for the event generator:

- It simulates the event router and instantiates a new instance of the adapter's event generator.
- It passes the `test.properties` to the event generator for initialization so you can test your initialization logic.
- It refreshes the event generator randomly so you can test your `setupNewTypes()` and `removeDeadTypes()` methods.

- It receives event postings and displays them in the log file for the adapter.

sample.client.ApplicationViewClient

The `sample.client.ApplicationViewClient` class offers an additional way to test your adapter. This class is a Java program that demonstrates how to invoke a service and listen for an event on an application view. An Ant `build.xml` file provides the *client* target so you can use the `ApplicationViewClient` program. When you execute `ant client`, the default configuration is to display the usage for the program. You can change the input parameters for the client program by editing the `build.xml` file.

To see an example of `sample.client.ApplicationViewClient.java`, go to `WLI_HOME/adapters/sample/src/sample/client`.

Note: `sample.client.ApplicationViewClient` is not integrated with the test harness.

Step 7: Deploy the Adapter

After implementing the SPI and CCI interfaces for an adapter, and then testing the adapter, you can deploy the adapter in a WebLogic Integration environment, either manually or from the WebLogic Server Administration Console. For complete information, see [Chapter 10, “Deploying Adapters.”](#)

Developing an Event Adapter

This section contains information about the following subjects:

- [Introduction to Event Connections](#)
- [Event Adapters in a Run-Time Environment](#)
- [Flow of Events](#)
- [Step 1: Define the Adapter](#)
- [Step 2: Configure the Development Environment](#)
- [Step 3: Implement the Adapter](#)
- [Step 4: Test the Adapter](#)
- [Step 5. Deploy the Adapter](#)

Introduction to Event Connections

Event connections propagate information from an EIS to the WebLogic Integration environment; they can be described as publishers of information.

All WebLogic Integration event connections perform the following functions:

- They respond to *events* that occur inside the running EIS by extracting and storing data about the event from the EIS.

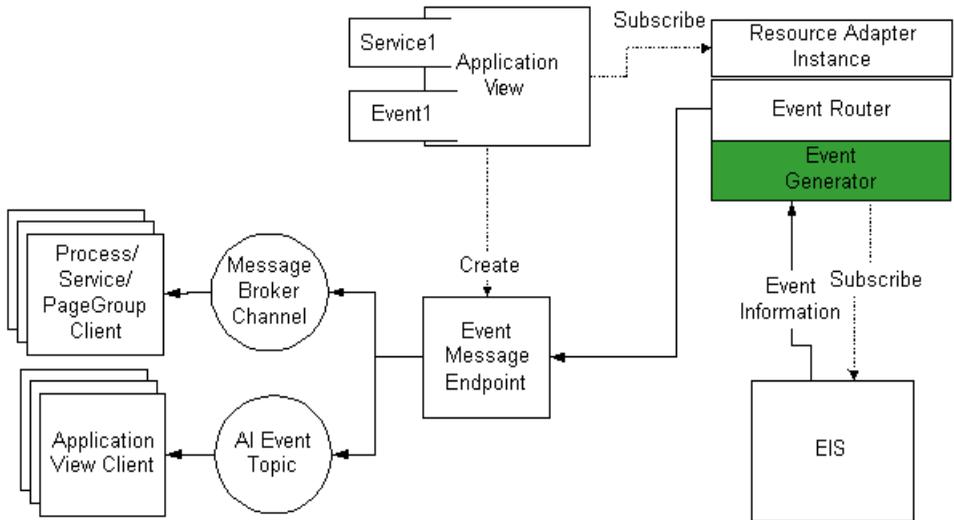
- They transform event data from an EIS-specific format to an XML document that conforms to the XML schema for the event. The XML schema is based on metadata in the EIS.
- They propagate the event in the WebLogic Integration environment by using the event router.
- Optionally, they respond to requests by the event router to suspend and resume event generation.
- Optionally, they can indicate their status, and the status of the EIS to which they are connected, back to the event router.
- Optionally, they can define logical event generator instances that allow system administrators to control the distribution of event generation work within a WebLogic Integration cluster, thus improving load balancing and fault tolerance.
- Optionally, they can make use of environment variables. Environment variables allow adapters to isolate information that is specific to a given deployment environment such that it can be updated by an administrator when the adapter instances using that adapter are moved between environments.

WebLogic Integration implements the aspects of the first three functions that are common to all event connections, allowing you to focus on only the EIS-specific aspects of your adapter. WebLogic Integration provides framework support for the four optional functions, but no direct implementation.

Event Adapters in a Run-Time Environment

The behavior of an event in a run-time environment is depicted in [Figure 7-1](#).

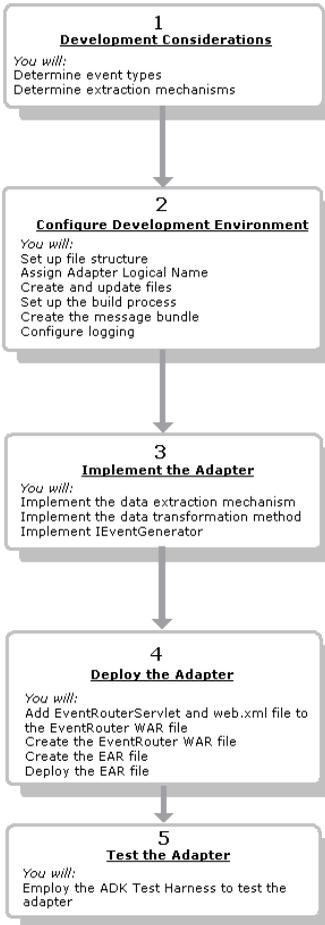
Figure 7-1 Event Connections in a Run-time Environment



Flow of Events

Figure 7-2 outlines the steps required to develop an Event Connection.

Figure 7-2 Event Connection Flow of Events



Step 1: Define the Adapter

Before you start developing an event connection, you must define your requirements for it. For a complete list of the information you need to do so, see [Appendix C, “Adapter Setup Worksheet.”](#) This section provides a summary of the most important tasks to be completed for step 1:

1. Define an event in terms of the following questions:
 - What will be the contents of the event?
 - How will the event be defined in the XML schema?
 - What will trigger the event?
2. Select one of the following data extraction methods:
 - *Push*—The EIS notifies the adapter of an event. Use this method when your adapter needs to poll the EIS to determine a change of state. This is the most efficient implementation for event generators, but involves EIS-side configuration and may not be possible with all EIS types.
 - *Pull*—The adapter polls the EIS and pulls event data from it. Use this method when you want to implement event generation that works like a publish-and-subscribe model. This is less efficient than the push model above, but is often easier to implement, is less dependent on the capabilities of the EIS, and doesn't generally require EIS-side configuration.
3. Decide if you will allow your event generator to be suspended and resumed.
 - Allow this if your EIS holds event information persistently, and it can be retrieved at a later time.
 - Do not allow this if your EIS cannot store event information for later retrieval (for example, event information is delivered via a non-persistent remote call)
4. Decide if you will indicate your event generator's status and the status of your EIS instance to the event router.
 - BEA strongly recommends you consider implementing this support. Implementing it greatly enhances the manageability of your adapter.
 - Implement this support if you can clearly determine the internal state of your event generator (for example, Healthy if your connection to the EIS is established and working properly) and/or you can clearly determine the status of the EIS instances (for example, Available if you have an active connection to the EIS).
5. Decide if you will implement event generator instance support.

- BEA strongly recommends you consider implementing this support. Implementing it greatly enhances the manageability of your adapter with regards to load balancing and fault tolerance within a WebLogic Integration cluster.
 - If your event generator can coexist with other event generators using the same EIS instance without conflicting with the operation of the other generators, you should consider implementing generator instance support. For example, the DBMS sample adapter is capable of sharing a single set of events within a DBMS instance with multiple generator instances. It has implemented generator instance support to enable it to be managed effectively in a WebLogic Integration cluster.
 - If your event generator cannot coexist with other event generators using the same EIS instance, you do not need to implement event generator instance support.
6. Decide if you will implement environment variable support.
- Environment variables allow adapters to isolate information that is specific to a given deployment environment such that it can be updated by a WLI administrator when the adapter instances using that adapter are moved between environments.
 - Adapters that require environment specific information in their interaction spec object or in the request document for services or the event definition for events, should consider implementing environment variable support. Examples of environment specific information are resource identifiers like table names for a DBMS adapter, or email folder names for an email adapter.
- Note:** Environment specific information does not include the information used to create a connection to the EIS. This information is almost always, by its nature, environment specific.
- Adapters that contain no environment specific information in their interaction spec object, request documents, or event definitions, need not implement environment variable support.

Step 2: Configure the Development Environment

This step involves completing a five-step procedure to prepare your computer for adapter development:

- [Step 2a: Set Up the File Structure](#)
- [Step 2b: Assign a Logical Name to the Adapter](#)
- [Step 2c: Set Up the Build Process](#)

- [Step 2d: Create the Message Bundle](#)
- [Step 2e: Configure Logging](#)

Step 2a: Set Up the File Structure

The file structure needed for an event connection development environment is the same as that required for developing service connections. For details, see “[Step 2a: Set Up the Directory Structure](#)” in [Chapter 6, “Developing a Service Adapter.”](#)

Step 2b: Assign a Logical Name to the Adapter

Assign a logical name to your adapter. By convention, this name comprises the vendor name, the type of EIS connected to the adapter, and the version number of the EIS, and it is expressed as *vendor_EIS-type_EIS version*. For example:

```
BEA_WLS_SAMPLE_ADK
```

This name includes the following components:

- BEA_WLS is the vendor.
- SAMPLE is the EIS type.
- ADK is the EIS version.

Step 2c: Set Up the Build Process

WebLogic Integration employs a build process based on Ant, a 100% pure Java-based build tool. For more information about how Ant works, see “[Ant-Based Build Process](#)” on [page 3-3](#). For more information about how to use Ant, go to:

<http://jakarta.apache.org/ant/index.html>

The sample adapter provided by WebLogic Integration contains an Ant build file: *WLI_HOME/adapters/sample/project/build.xml*. This file, in turn, contains the tasks needed to build a J2EE-compliant adapter. When you run the `GenerateAdapterTemplate` utility to clone a development tree for your adapter, a `build.xml` file is created specifically for that adapter. Because this file is generated automatically, you do not need to customize the sample `build.xml` file and you can be sure that the code is correct. For information about using the `GenerateAdapterTemplate` utility, see [Chapter 4, “Creating a Custom Development Environment.”](#)

For more information about the build process, see “[Step 2c: Set Up the Build Process](#)” in [Chapter 6, “Developing a Service Adapter.”](#)

Step 2d: Create the Message Bundle

Any message destined for an end-user should be placed in a *message bundle*: a `.properties` text file containing *key=value* pairs that allow you to internationalize messages. When a geographic locale and a natural language are specified for a message at run time, the contents of the message are interpreted on the basis of the *key=value* pair, and the message is presented to the user in the specified language.

For instructions on creating a message bundle, see the JavaSoft tutorial on internationalization at:

<http://java.sun.com/docs/books/tutorial/i18n/index.html>

Step 2e: Configure Logging

Logging is performed with a logging tool called Log4j, which was developed as part of the Apache Jakarta project.

Before you begin this step, we recommend that you read more about logging in [Chapter 2, “Basic Development Concepts,”](#) and about how to use Log4j in [Chapter 5, “Using the Logging Toolkit.”](#)

Create an Event Generation Logging Category

If you are planning to use an event connection, you must create a logging category specifically for event generation. (For more information about logging categories, see “[Message Categories](#)” [on page 5-3.](#)) To edit the logging configuration file for a specific adapter

(`WLI_HOME/adapters/YOUR_ADAPTER/src/adapter_logical_name.xml`), add the code shown in the following listing.

Listing 7-1 Sample Code for Creating an Event Generation Logging Category

```
<category name='BEA_WLS_SAMPLE_ADK.EventGenerator' class='com.bea.  
    logging.LogCategory'>  
</category>
```

Replace `BEA_WLS_SAMPLE_ADK` with the logical name of your adapter.

If you do not set any parameters for this category, it inherits all the property settings of the parent category. In this example, the parent category is `BEA_WLS_SAMPLE_ADK`. Although you are not required to use the adapter logical name as the root category, you must use a unique identifier so that there is no impact on other adapters in a multi-adapter environment.

Step 3: Implement the Adapter

To implement an event connection, you must complete the following two-step procedure:

1. Create an event generator. This process implements the data extraction method (in either push or a pull mode) and the `IEventGenerator` interface. (The latter interface is used by the event router to drive the event generation process.) This step is described in [“Step 3a: Create an Event Generator.”](#)
2. Implement the data transformation method. This step is described in [“Step 3b: Implement the Data Transformation Method.”](#)

Step 3a: Create an Event Generator

Event generation provides an adapter with a mechanism to either receive notification from an EIS or poll an EIS for a specific occurrence of an event. The WebLogic Integration engine provides a powerful event generator that can support multiple types of events. An event type is defined by the configuration properties for an event.

Typically event properties are defined by the properties associated with an event at design time. When configuring an event connection, keep in mind that you may designate one or more Web pages from which the adapter will collect event properties. These properties are saved with the application view descriptor and passed back to the event at run time. The WebLogic Integration engine uses the properties and the source application view to determine how to route back to the listeners. For instance, if two separate deployments of the same event generator with identical properties are used, only one `IEventDefinition` is created by the WebLogic Integration engine. If different properties are specified, however, a single `IEventDefinition` is created for each deployment of a single event connection. The event generator must determine which `IEventDefinition` to use in the routing process. This determination is typically made on the basis of property values and specific event occurrences.

`IEventDefinition` objects are used by your implementation of the event generator to route specific events back to their listener. As discussed elsewhere, the WebLogic Integration engine creates `IEventDefinition` objects for deployed application views containing events.

`IEventDefinition` objects can be used to extract specific properties associated with the

deployment of an application view, or to access schema and routing objects. You must employ these attributes when routing an event.

How the Data Extraction Mechanism Is Implemented

WebLogic Integration supports two modes of data extraction:

- Push event generation—A state change is recognized when the object generating events pushes a notification to the event generator. When the push event generator receives the event the WebLogic Integration engine then routes the event to a deployed application view. The push event generator uses a publish-and-subscribe model.
- Pull event generation—Used when polling is required to confirm a state change. A process continually queries an object until it detects a change in state, at which point it creates an event, which the WebLogic Integration engine then routes to a deployed application view.

Pull Mode

Pull mode relies on a polling technique to determine whether an event has taken place. To implement it, you must derive your event generator from the `AbstractPullEventGenerator` in the `com.bea.adapter.event` package.

Note: The `adk-eventgenerator.jar` file contains the ADK base classes required to implement an event generator. It must be included in your WAR make file.

In the `AbstractPullEventGenerator`, the ADK supplies several abstract methods that you must override in your implementation. These methods are described in the following table.

Table 7-1 AbstractPullEventGenerator Methods

Method	Description
<code>postEvents()</code>	Control method for the remainder of your event generation, message transformation, and routing code; allows you to add polling and routing code. Called from the <code>run</code> method in the <code>AbstractPullEventGenerator</code> at an interval determined by the Event Router configuration files.
<code>setupNewTypes()</code>	Method for preprocessing any <code>IEventDefinition</code> object being deployed. Only valid new <code>IEventDefinition</code> objects can be passed to this method.

Table 7-1 AbstractPullEventGenerator Methods (Continued)

Method	Description
<code>removeDeadTypes()</code>	Handles any cleanup required for <code>IEventDefinition</code> objects being undeployed. The WebLogic Integration engine calls this method when application views with associated events are being undeployed.
<code>doInit()</code>	Method called while the event generator is being constructed. During the initialization process the event generator can use predefined configuration values to set up the necessary state or connections for the event generation process.
<code>doCleanUpOnQuit()</code>	Frees resources allocated by your event generation process. Called before the thread driving the event generation process is ended.

Push Mode

Push mode uses notification to trigger the routing of an event. To implement it, you must derive your event generator from the `AbstractPushEventGenerator` class in the `com.bea.adapter.event` package. Several other supporting classes are included in the event package. These classes are described in [Table 7-2](#).

Note: The `adk-eventgenerator.jar` file contains the WebLogic Integration base classes required to implement an event generator. It must be included in your WAR make file.

Table 7-2 AbstractPushEventGenerator Classes

Class	Description
AbstractPushEventGenerator	Class containing the same abstract and concrete methods as the AbstractPullEventGenerator. The methods in both implementations (AbstractPullEventGenerator and AbstractPushEventGenerator) are intended to be used in the same manner. For a list of the methods and responsibilities associated with each, see Table 7-1 .
IPushHandler	Interface provided primarily to abstract the generation of an event from the routing of an event. It is not required for the implementation of the push mode of data extraction. The IPushHandler is designed to be tightly coupled with the PushEventGenerator. The PushEventGenerator initializes, subscribes, and cleans up the PushHandler implementation. The IPushHandler provides a simple interface to abstract the generation logic. The interface provides methods to initialize, subscribe to push events, and clean up resources.
PushEvent	PushEvent is an event object derived from java.util.EventObject. The PushEvent object is designed as a wrapper for an EIS notification, which is sent to any IPushEventListener objects.
EventMetaData	The EventMetaData class is intended to wrap any data necessary for event generation. The EventMetaData class is passed to the IPushHandler on initialization.

How the Event Generator Is Implemented

An event generator typically implements the following flow of control:

1. The `doInit()` method creates and validates connections to the EIS.

2. The `setupNewTypes()` method processes `IEventDefinition` objects, creating any structures required for processing.
3. The `postEvents()` method iteratively invokes one of the two modes of data extraction:
 - Push—The `postEvents()` method polls the EIS for an event and, if an event exists, `postEvent()` determines which `IEventDefinition` objects will receive it. The method then transforms the event data into an `IDocument` object, using the associated schema, and routes the `IDocument` object using the `IEvent` associated with the `IEventDefinition` object.
 - Pull—The `postEvents()` method waits for notification of an event. When it receives such notification, it extracts the event data from the `PushEvent` object and transforms it into an `IDocument` object in accordance with the schema associated with the event connection. When all the necessary event data has been put into the `IDocument`, the `IDocument` is routed to the correct `IEventDefinition` objects.
4. The `removeDeadTypes()` method removes dead `IEventDefinition` objects from any data structures being used for event processing. Any resources associated with those objects are also freed. `IEventDefinition` objects are considered *dead* when the application view to which they belong is undeployed.
5. The `doCleanUpOnQuit()` method removes any resources allocated during event processing.

[Listing 7-2](#) shows the class declaration for the sample adapter’s (pull-mode) event generator.

Listing 7-2 Sample Implementation of the Pull Mode of Data Extraction

```
public class EventGenerator
    extends AbstractPullEventGenerator
```

Note: The `AbstractPullEventGenerator` implements the `Runnable` interface, which enables it to run on its own thread.

The remaining sections in “[Step 3a: Create an Event Generator](#)” provide more code examples that show how an event generator is implemented with the pull mode of data extraction.

Sample EventGenerator

[Listing 7-3](#) shows a simple constructor for an event generator. You must invoke the parent's constructor so that the parent's members get initialized correctly. The listing then shows how the `doInit()` method receives configuration information from the `map` variable and validates the parameters. The sample contains any parameters associated with the event generator at design time.

Listing 7-3 Sample Constructor for an EventGenerator

```
public EventGenerator()
{
    super();
}

protected void doInit(Map map)
    throws java.lang.Exception
{
    ILogger logger = getLogger();

    m_strUserName = (String)map.get("UserName");
    if (m_strUserName == null || m_strUserName.length() == 0)
    {
        String strErrorMsg =
            logger.getI18NMessage("event_generator_no_UserName");
        logger.error(strErrorMsg);
        throw new IllegalStateException(strErrorMsg);
    }

    m_strPassword = (String)map.get("Password");
    if (m_strPassword == null || m_strPassword.length() == 0)
    {
        String strErrorMsg = logger.getI18NMessage
            ("event_generator_no_Password");
        logger.error(strErrorMsg);
        throw new IllegalStateException(strErrorMsg);
    }
}
```

`postEvents()` is called from the `run` method of the parent class, as shown in [Listing 7-4](#). This method polls the EIS to detect the occurrence of a new event. This method is invoked at a fixed interval, which is defined in the `web.xml` file for the event router.

Listing 7-4 Sample Implementation of `postEvents()`

```
protected void postEvents(IEventRouter router)
    throws java.lang.Exception
{
    ILogger logger = getLogger();

    // TODO: a real adapter would need to call into the EIS to
    // determine if any new events occurred since the last time
    // this method was invoked. For the sake of example, we'll just
    // post a single event every time this method gets invoked...
    // event data will be the current time on the
    // The system formatted according to the event definition...
    // we'll look for several event types...

    Iterator eventTypesIterator = getEventTypes();
    if (eventTypesIterator.hasNext())
    {
        do
        {
            // The event router is still interested in this type of event
            IEventDefinition eventDef = (IEventDefinition)
                eventTypesIterator.next();
            logger.debug("Generating event for " + eventDef.getName());

            // Create a default event (just blank/default data)
            IEvent event = eventDef.createDefaultEvent();

            // Get the format for the event
            java.util.Map eventPropertyMap = eventDef.
                getPropertySet();
            String strFormat = (String)eventPropertyMap.get
                ("Format");
```

Developing an Event Adapter

```
        if( logger.isDebugEnabled() )
            logger.debug("Format for event type '"+eventDef.
                getName()+"' is '"+strFormat+"'");
        java.text.SimpleDateFormat sdf =
            new java.text.SimpleDateFormat(strFormat);
        IDocument payload = event.getPayload();
        payload.setStringInFirst("/SystemTime", sdf.format(new
            Date()));

        // let's log an audit message for this...

        try
        {
            logger.audit(toString() + ": postEvents >>> posting event
                ["+payload.toXML()+"] to router");
        }

        catch (Exception exc)

        {
            logger.warn(exc);
        }

        // This call actually posts the event to the IEventRouter
        router.postEvent(event);
    } while (eventTypesIterator.hasNext());
}

} // end of postEvents
```

A real adapter must query the EIS to determine whether any new events have occurred since the last time this method was invoked. A concrete example of such a call, available in the DBMS sample adapter included with the ADK, is the `postEvent()` method in the `EventGenerator.java` file:

`WLI_HOME/adapters/dbms/src/com/bea/adapter/dbms/event/EventGenerator.java`

Adding New Event Types

`setupNewTypes()` is called during refresh to handle any new event types. Typically, an event generator needs to allocate resources in the EIS in order to be able to receive events from the EIS.

In the DBMS sample adapter, for example, a trigger is created in the DBMS in order to handle a new event type. The `setupNewTypes()` method allows you to set up any definitions required to handle a new type. The parent class has already performed (and logged) a sanity-check on the `listOfNewTypes()` file, so you do not need to perform those tasks.

Listing 7-5 Sample Template for `setupNewTypes()`

```
protected void setupNewTypes(java.util.List listOfNewTypes)
{
    Iterator iter = listOfNewTypes.iterator();
    while (iter.hasNext())
    {
        IEventDefinition eventType = (IEventDefinition)iter.next();
    }
}
```

Removing Event Types for Undeployed Application Views

`removeDeadTypes()` is called during refresh to remove any event types for application views that have been undeployed.

You must execute a cleanup process

To ensure that obsolete event types are no longer handled, you must perform a cleanup process. You should, for example, close resources needed to handle the obsolete event type. [Listing 7-6](#) shows how `removeDeadTypes()` is implemented.

Listing 7-6 Sample Code Based on `removeDeadTypes()` Template

```
protected void removeDeadTypes(java.util.List listOfDeadTypes)
{
    Iterator iter = listOfDeadTypes.iterator();
    while (iter.hasNext())
    {
        IEventDefinition eventType = (IEventDefinition)iter.next();
    }
}
```

Removing Resources

`doCleanUpOnQuit()` is called during shutdown of the event generator. This method removes any resources allocated during event processing. The sample adapter stubs in this method. The template for implementing this method is shown in the following listing.

Listing 7-7 Sample `doCleanUpOnQuit()` Method Call

```
protected void doCleanUpOnQuit()
    throws java.lang.Exception
{
    ILogger logger = getLogger();
    logger.debug(this.toString() + ": doCleanUpOnQuit");
}
}
```

Step 3b: Implement the Data Transformation Method

Data transformation is the process of taking data from the EIS and transforming it into an XML schema that can be read by the application server. For each event, a schema defines the appearance of the XML output, using the `SOM` and `IDocument` class libraries. The following code listings show the sequence of events during the data transformation process:

- [Listing 7-8](#) shows the code that transforms data from the EIS into XML schema.
- [Listing 7-9](#) shows the XML schema created by the code in [Listing 7-8](#).
- [Listing 7-10](#) shows the valid XML document created by the schema shown in [Listing 7-9](#).

Listing 7-8 Sample Code for Transforming EIS Data into XML Schema

```
SOMSchema schema = new SOMSchema();
SOMElement root = new SOMElement("SENDINPUT");
SOMComplexType mailType = new SOMComplexType();
root.setType(mailType);
SOMSequence sequence = mailType.addSequence();
SOMElement to = new SOMElement("TO");
```

```

to.setMinOccurs("1");
to.setMaxOccurs("unbounded");
sequence.add(to);
SOMElement from = new SOMElement("FROM");
from.setMinOccurs("1");
from.setMaxOccurs("1");
sequence.add(from);
SOMElement cc = new SOMElement("CC");
cc.setMinOccurs("1");
cc.setMaxOccurs("unbounded");
sequence.add(cc);
SOMElement bcc = new SOMElement("BCC");
bcc.setMinOccurs("1");
bcc.setMaxOccurs("unbounded");
sequence.add(bcc);
SOMElement subject = new SOMElement("SUBJECT");
subject.setMinOccurs("1");
subject.setMaxOccurs("1");
sequence.add(subject);
SOMElement body = new SOMElement("BODY");
if (template == null)
    { body.setMinOccurs("1");
      body.setMaxOccurs("1");
    }else
    { Iterator iter = template.getTags();
      if (iter.hasNext())
        { SOMComplexType bodyComplex = new SOMComplexType();
          body.setType(bodyComplex);
          SOMAll all = new SOMAll();
          while (iter.hasNext())
            { SOMElement eNew = new SOMElement((String)iter.next());
              all.add(eNew);
            }//endwhile
          bodyComplex.setGroup(all);
        }//endif
    }//endif
sequence.add(body);
schema.addElement(root);

```

Listing 7-9 XML Schema Created by Code in Listing 7-8

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="SENDINPUT">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="TO" maxOccurs="unbounded"
        type="xsd:string"/>
      <xsd:element name="FROM" type="xsd:string"/>
      <xsd:element name="CC" maxOccurs="unbounded"
        type="xsd:string"/>
      <xsd:element name="BCC" maxOccurs=
        "unbounded" type="xsd:string"/>
      <xsd:element name="BCC" maxOccurs="unbounded"
        type="xsd:string"/>
      <xsd:element name="BODY" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Listing 7-10 Valid XML Document Created by Schema in Listing 7-9

```
</xsd:schema>
<?xml version="1.0"?>
<!DOCTYPE SENDINPUT>
<SENDINPUT>
  <TO/>
  <FROM/>
  <CC/>
  <BCC/>
  <BCC/>
  <BODY/>
</SENDINPUT> <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

Step 3c: Implement Suspend/Resume Support

This is an optional step. Suspend/resume support allows your event generator to respond to requests from the event router to suspend or resume generation of events. The event router makes these requests when an administrator indicates event delivery from your adapter should be suspended. If you do not implement suspend/resume support directly, the WebLogic Integration engine will store any events generated by your event generator during the time your adapter is suspended.

Whether you implement suspend/resume support in your adapter depends on whether your EIS instance can store event information for later retrieval. If your EIS can store this information, you should implement suspend/resume support in your adapter. This allows you to store events (possibly) more efficiently on the EIS than if WebLogic Integration were to store them for you.

If your EIS cannot effectively store event information for later retrieval, you should not implement suspend/resume support directly in your event generator. Rather, you should just allow WebLogic Integration to store the events for you.

If you decide to implement suspend/resume support, simply implement the `com.bea.wlai.event.ISuspendableEventGenerator` interface on your event generator implementation class. If you subclass `AbstractPullEventGenerator`, you can simply delegate the `suspend()` and `resume()` calls back to the superclass. The following code listing shows how the DBMS sample adapter implements `ISuspendableEventGenerator`.

Listing 7-11 Sample Adapter

```
import com.bea.adapter.event.AbstractPullEventGenerator;
import com.bea.wlai.event.ISuspendableEventGenerator;

public class EventGenerator
extends AbstractPullEventGenerator
implements ISuspendableEventGenerator {

...

/**
 * Suspend (temporarily) the generation of events. Any events that occur
 * on the EIS instance must be preserved either on the EIS instance itself,
 * or within some EventGenerator-specific store. This method simply delegates
```

```
    * the call back to its superclass.
 */
public void suspend()
    throws Exception
{
    super.suspend();
}

/**
 * Resume the generation of events. Any events that have occurred while this
 * EventGenerator was suspended, should now be delivered as soon as possible
 * to the EventRouter. This method simply delegates the call back to its
 * superclass.
 */
public void resume()
    throws Exception
{
    super.resume();
}

...
}
```

If your event generator does not extend from `AbstractPullEventGenerator`, you will need to determine for yourself the proper way to implement `ISuspendableEventGenerator`.

Step 3d: Implement Event Generator and EIS Status Reporting

This is an optional step. WebLogic Integration 8.1 adds the ability for event generators to indicate their own status and the status of their EIS instance back to their event router. This ability greatly enhances the manageability of your event generator. BEA strongly recommends adapter developers implement this support in their adapters.

Status reporting provides two important benefits to the WebLogic Integration administrator:

- Within the WebLogic Integration Administration Console, when your event generator moves to any abnormal state, the administrator can see the status message your event generator has itself reported to the event router. This provides good diagnostic information to the administrator.
- When your event generator detects any problems communicating with the EIS, it can indicate this directly back to the router, allowing the adapter instance it belongs to be

suspended, and thus clearly indicate the EIS is unavailable. Without this support, EIS availability is only detected second-hand during service invocation through the service adapter or by errors logged by the event generator.

To implement status reporting, you will use the following methods on the `IEventRouter` interface.

Note: You are passed an `IEventRouter` instance in the call to `postEvents`, `setupNewTypes` and `removeDeadTypes`.

- `setEventGeneratorStatus(Status status)`—This allows you to indicate the status of your event generator. The `Status` class is an enumeration class that defines standard status values (for example, `UNKNOWN`, `HEALTHY`, `SUSPENDED`, `MARGINAL`, `INOPERATIVE`). You may choose any of these values, or you may define your own status values as integer values greater than 100. The status class also provides a description field that allows you describe the reason your event generator is in the indicated state.
- `eisUnavailable(String reason)`—This method is used to indicate that your event generator cannot contact its EIS instance. Call this method with a descriptive reason why you couldn't contact the EIS. Calling this method will cause the adapter instance for the event generator to be suspended, and will cause the adapter instance to appear in the WLI management console as `Suspended`.
- `eisAvailable(String reason)`—This method is used to indicate that your event generator can again contact its EIS instance. Call this method with a descriptive reason why you now can contact the EIS. Examples would include resetting the connection to the EIS, or waiting some predefined amount of time and retrying the connection. Calling this method will cause the adapter instance for the event generator to be resumed, and will cause the adapter instance to appear in the WLI management console as `Deployed`.

Listing 7-12 DBMS Sample Adapter Status Reporting

```
/**
 * This method is called each time the event generator accesses
 * the connection to the DBMS.
 */
private final boolean verifyEISStatus(IEventRouter router)
{
    Status status = null;
    if (router != null)
    {
```

Developing an Event Adapter

```
        status = router.getEventGeneratorStatus();
    }

    boolean pingSucceeded = false;
    Exception pingException = null;
    try
    {
        // Perform a trivial operation on the connection to see if its alive.
        // Try to revive the connection if the ping fails the first time.
        pingSucceeded = pingAndRevive(true);
    }
    catch (Exception e)
    {
        pingException = e;
    }

    // See where we are, and act on it

    if (pingSucceeded)
    {
        if (status == null || status.status != Status.HEALTHY)
        {
            router.eisAvailable("Ping on DBMS connection succeeded");
        }
    }
    else
    {
        if (status == null || status.status != Status.SUSPENDED)
        {
            String msg = "Ping on DBMS connection failed";
            if (pingException != null)
            {
                msg += ": " + pingException;
            }
            router.eisUnavailable(msg);
        }
    }
}
```

```
    return pingSucceeded;
}
```

Step 3e: Implement Event Generator Instance Support

Note that as of WebLogic Integration 8.1 Service Pack 2, you must set event generator targets when in a clustered environment. A blank target (" ") does not trigger events on any node in the cluster. (In a single node environment, no entry is needed; a blank event generator target value kicks off events for the one server.)

WebLogic Integration 8.1 has added the ability to distribute event generators among nodes in a WebLogic Integration cluster. This improves load balancing and fault tolerance of event generation. However, the ability to do this is predicated in the requirements and implementation of the adapter itself.

Some event generators assume that they own any resources in the EIS they use to detect the occurrence of an event. Such event generators cannot coexist with other event generators using the same EIS instance, as resource usage conflicts will arise. An example of this is the WebLogic Integration 7.0 DBMS sample adapter. It assumed that it owned the information in the event staging tables within the DBMS. Because of this it could not coexist with other WebLogic Integration 7.0 DBMS sample adapter event generators, because they would each compete for the event information in the event staging tables, often causing starvation of one or more generators or database locking/update conflicts.

Note: Even event generators that do not implement event generator instance support can benefit from the new event generation targeting capabilities. However, the WebLogic Integration administrator is only able to designate nodes in cluster to have a single instance (or no instance) of the event generator. Implementing event generator instance support allows the adapter to have finer control over event generation and greater manageability.

Example: DBMS Sample Adapter

In WebLogic Integration 8.1, the DBMS sample adapter has been enhanced to allow the coexistence of multiple event generators using the same DBMS instance. It does this by implementing a concept known as event generator instance support. The DBMS sample adapter recognizes any number of administrator defined generator instances. This section provides an example of what generator instances are and how they are used in the DBMS sample adapter.

We have a 3 node WebLogic Integration cluster. The cluster nodes are named Server1, Server2, and Server3. The administrator wishes to evenly distribute event generation responsibilities for a single adapter instance (and by convention a single EIS instance) among the nodes. Before this can be done, the administrator must know if the adapter supports multiple event generators or not. This information is included in the adapter documentation.

If the adapter does not support multiple generator instances, a single event generator must be associated with a single node in the cluster. This is done by specifying the following value for the Event Generation Targets field in the WebLogic Integration Administration Console (Event Connection Details page).

Event Generation Targets: Server1

If the adapter does support multiple generator instances, event generators can be distributed to each node in the cluster. This is done by specifying the Event Generation Targets field as follows:

Event Generation Targets: Server1,Server2,Server3

This allows 3 event generators to share the work of generating events for the given EIS instance.

In the case of the DBMS sample adapter, having multiple event generators selecting/deleting/updating in the same tables caused database locking conflicts. To overcome this, we define a way to identify each generator instance such that the individual events are destined to a specific generator instance. This allows multiple event generators to coexist because they each have their own set of events to handle. The use of generator instance identifiers effectively partitions the single set of event staging tables for the DBMS sample adapter into multiple logical sets of tables.

For the DBMS sample adapter, a generator instance has a numeric identifier, or ID. The system administrator, knowing that the DBMS sample adapter supports event generator instances, can define a set of generator instances by defining a list of numeric identifiers. He can then distribute these generator instances among the nodes in the cluster by associating the generator instance's numeric ID with the server name in the Event Generation Targets field value.

A typical Event Generation Targets setting for our example 3 node cluster, for the DBMS sample adapter is as follows:

Event Generation Targets: Server1=[1/3],Server2=[2/3],Server3=[3/3]

This specification defines one generator instance per node in the cluster. Each instance specification is represented as:

`= [instance_id/number_of_instances]`

The instance specification follows the name of the server to which the instances are to be associated.

Note: The *instance_id/number_of_instances* format is specific to the DBMS sample adapter. You are free to define the format of the generator instance however you wish. However, the list of instances is always enclosed in square brackets [] and each instance is separated from the others by one more space characters. Because of this, your instance format cannot allow the following characters:

- open/close square bracket ([])
- comma (,)
- whitespace of any kind (such as spaces, tabs, or newline).

Why Implement Event Generator Instance Support?

To see why event generator instance support is a powerful feature, let us consider a single-node failure scenario within the WebLogic Integration cluster, using the DBMS sample adapter. Let's assume that the administrator has configured the three nodes to have one generator instance each, and that the generators have been initialized and are steadily processing, and delivering events. Let's assume that sometime later, Server2 fails.

In this case, since the administrator assigned generator instance 2 to Server2, all events destined for instance 2 simply stop being processed when Server2 fails. When the administrator detects the failure of Server2, they can use the WebLogic Integration Administration Console to reset the Event Generation Targets field to retarget instance 2 to a live node in the cluster:

Event Generation Targets: Server1=[1/3 2/3],Server3=[3/3]

By making this change, the administrator has migrated the event generation responsibilities for instance 2 from the failed Server2 to the live Server1. In the next section, we describe how the event generator will detect changes to the generator instance specification, and how it should respond to those changes.

Detecting and Responding to Changes in Event Generator Instance Specifications

Any time the WebLogic Integration administrator changes the value of Event Generation Targets, the event generator receives a call to `refresh()`. This is the generator's opportunity to refresh the list of generator instance IDs it has been made responsible for.

The DBMS sample adapter extends `AbstractPullEventGenerator`.

`AbstractPullEventGenerator` intercepts the call to `refresh()`, collates the current event

types for the event generator, and calls `setupNewTypes()` or `removeDeadTypes()` only as needed. To make generator instance handling work reliably, we override the `refresh()` method as shown in [Listing 7-13](#).

Listing 7-13 Getting List of Generator Instance IDs in the `refresh()` Method

```
/**
 * Refresh this generator, taking into account the current list of event
 * types (definitions) this router is responsible for, and also the list
 * of event router instance IDs its responsible for.
 *
 */
public void refresh()
    throws Exception
{
    // See what instance IDs we're being asked to handle
    refreshEventRouterInstanceIDs();

    // Make sure the EVENT_GENERATOR table reflects our new max count (this
    // call is a no-op if we're not the 'primary' generator instance
    // (e.g. instance id 1).
    updateGeneratorID(getConnection(), true);

    // Let our super class decide whether the event types for this router have
    // changed.
    super.refresh();
}

private void refreshEventRouterInstanceIDs()
{
    m_isPrimaryInstance = false;
    String[] eventRouterInstanceIDs =
        getRouter().getEventRouterInstanceIDs();

    ... Store the new list of instance IDs, and if we see instance '1', we have
    become the 'primary' instance. The 'primary' instance is responsible for
    certain singleton lifecycle updates. The concept of principal instance
```

```

allows us to avoid collisions that might result if more than one instance
attempted these lifecycle operations ...
}

```

Note that we call the superclass `refresh()` method at the end of the overridden method. This ensures proper calls to `setupNewTypes()` and `removeDeadTypes()` if needed. Note also that within the call to `refresh()` we retrieve the current list of generator instance IDs this generator is responsible for. The DBMS sample adapter event generator will then begin selecting events from the event staging tables destined for any generator instance it has found in the new list.

Note: The DBMS sample adapter event generator only uses the one thread created for it by `AbstractPullEventGenerator`. A powerful way to scale the processing power of your event generator would be to allocate one processing thread per generator instance ID. This way, in our single-node failure scenario above, you could have two threads processing events on `Server1` instead of just one. In addition, having more threads as more generator instances are associated with an event generator provides a way for the administrator to increase event processing throughput. He can simply define a larger number of generator instances (say two or more per node in the cluster), and distribute them among the nodes.

Step 3f: Implement Environment Variable Support

This is an optional step. Environment variables allow adapters to isolate information that is specific to a given deployment environment such that it can be updated by a WLI administrator when the adapter instances using that adapter are moved between environments.

Adapters that require environment specific information in their interaction spec object, service request documents, or event definitions, should consider implementing environment variable support. Examples of environment specific information are resource identifiers like table names for a DBMS adapter, or email folder names for an email adapter.

Note: Environment specific information does not include the information used to create a connection to the EIS. This information is almost always, by its nature, environment specific.

Adapters that contain no environment specific information in their interaction spec object, request documents, or event definitions, need not implement environment variable support.

Environment variables are defined for a given application view at design-time in the Application Integration Design Console. The set of variable definitions created there are propagated to the

Developing an Event Adapter

event adapter instance, event router, and finally event generator by way of the `IEventDefinition` object used to represent an event subscription in the event generator.

For more information on how environment variables are defined at design-time, see [“Developing a Design-Time GUI” on page 9-1](#).

You get a set of event definitions in each call to `IEventGenerator.refresh()` or `AbstractEventGenerator.setupNewTypes()` or `AbstractEventGenerator.removeDeadTypes()`.

An environment variable at runtime is represented as an instance of the `IVariable` interface. The `IVariable` interface is as follows:

```
/**
 * Contains an adapter defined environment variable
 */
public interface IVariable
    extends java.io.Serializable
{
    public String getName();
    public String getType();
    public String getDescription();
    public String getDefaultValue();
    public String getValue();
}
```

The variables are contained in a variable set defined as follows:

```
/**
 * Contains a set of adapter defined environment variables for an adapter
 */
public interface IVariableSet
    extends java.io.Serializable
{
    public void addListener(VariableChangeListener listener);
    public void removeListener(VariableChangeListener listener);
    public String[] listVariableNames();
    public IVariable getVariable(String name);
    public IVariable[] listVariables();
}
```

Event generator instances will likely only be interested in the name and value of variables. How you use variables in your event generator implementation is up to you. It depends on the format of the metadata your design-time component places into the event definition for an event. Typically, adapters will place marker fields into the text for an event definition property indicating a replaceable variable value.

For example, the DBMS sample adapter allows application view designers to use environment variables when defining the table, catalog, and schema names for an event. The DBMS sample adapter treats events as any insert/delete/update on a designated table. A table is identified by its name, and the name of the catalog and schema that contain it. These names often vary between environments, and thus using variables to represent them allows the administrator an easy way to change those names as he moves the application view between environments. A typical event descriptor for the DBMS sample adapter contains the following property name/value pairs:

- `tableName`—`CUSTOMER_TABLE`
- `schemaName`—`{mySchema}`
- `catalogName`—`{myCatalog}`

Notice the curly braces enclosing the `mySchema` and `myCatalog` values. These braces are used by the DBMS sample adapter to indicate a variable value. The variables in this case are `mySchema`, and `myCatalog`. With this event definition, the DBMS sample adapter event generator expects the event definition to be accompanied at runtime by a set of variables containing two variables named `mySchema` and `myCatalog`. At runtime, the event generator retrieves the variable values in effect at the time, and substitutes those values for the `{mySchema}` and `{myCatalog}` text in the `schemaName` and `catalogName` property values, respectively. Here is the code from the DBMS sample adapter:

```
IEventDefinition eventDef = ... One of the event defs passed in setupNewTypes() ...

IClientData clientData = eventDef.getClientData();
if (clientData != null)
{
    IVariableSet varSet = clientData.getVariableSet();
    catalogName = DBMSSQLUtils.applyVariables(varSet, catalogName);
    schemaName = DBMSSQLUtils.applyVariables(varSet, schemaName);
    tableName = DBMSSQLUtils.applyVariables(varSet, tableName);
}
```

`DBMSSQLUtils.applyVariables()` simply does a string replacement of the variables it find in the given property with the value of the corresponding variable.

Step 4: Test the Adapter

You can test the adapter by using the adapter test harness provided with WebLogic Integration. For a complete description of this tool and instructions for using it, see [“Step 6: Test the Adapter”](#) in [Chapter 6, “Developing a Service Adapter.”](#)

Step 5. Deploy the Adapter

After rebuilding the new adapter, deploy it in a WebLogic Integration environment. You can deploy an adapter either manually or from the WebLogic Server Administration Console. For complete information, see [Chapter 10, “Deploying Adapters.”](#)

Developing a Resource Adapter

This section contains information about the following subjects:

- [Introduction to Resource Adapters](#)
- [Resource Adapters in a Run-Time Environment](#)
- [Step 1: Define the Adapter](#)
- [Step 2: Configure the Development Environment](#)
- [Step 3: Implement the Adapter](#)
- [Step 4: Implement the Event and Service Adapters](#)
- [Step 5. Deploy the Adapter](#)

Introduction to Resource Adapters

The `ResourceAdapter` interface is new in WebLogic Integration 8.1. It serves as a single object to unify both event and service handling. In prior WebLogic Integration releases, the event adapter and service adapter were treated as separate adapter components. Each was deployed and configured separately.

In WebLogic Integration 8.1, the event and service adapters are unified under the `ResourceAdapter` interface, and the adapter's implementation of that interface. It provides access to 0 or 1 event connection (created by the event adapter), and 0 or more service connections (sometimes referred to as connection factories, and created by the service adapter)

Configuration of event and service connections are now done under the umbrella of the `ResourceAdapter` interface. You will see the term Resource Adapter used in the Application Integration Design Console to represent a container of event and service connections.

WebLogic Integration defines an extension to the generic `ResourceAdapter` interface (called `WLIResourceAdapter`) that contains enhancements designed to improve usability and adds methods needed to support optional features of event adapters. The optional features enabled by these added methods are:

- Suspend/Resume Support
- Event Generator and EIS Status Reporting
- Event Generator Instance Support

Each adapter must implement `WLIResourceAdapter` in a concrete class in order to operate within WebLogic Integration 8.1. The ADK provides an abstract base implementation of the `WLIResourceAdapter` class in `com.bea.adapter.spi.AbstractWLIResourceAdapter`. WebLogic Integration also provides an adapter repackaging utility that allows existing adapters to be utilized in WebLogic Integration 8.1. This utility code generates the required `WLIResourceAdapter` implementation for any repackaged adapter.

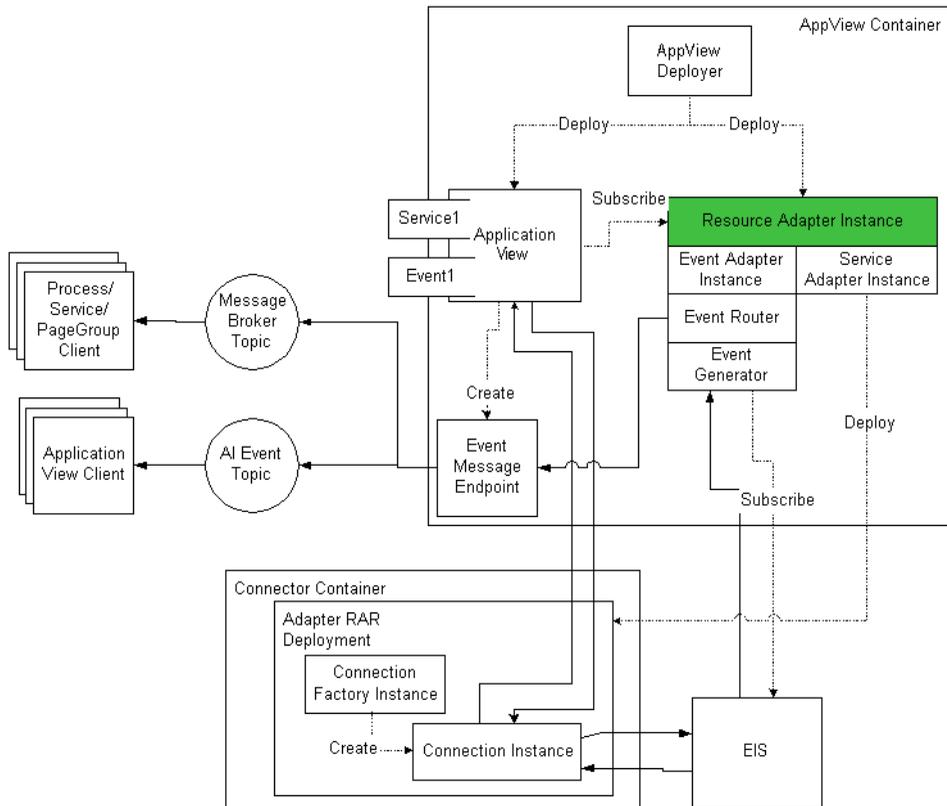
The `WLIResourceAdapter` implementation should be implemented according to the JavaBeans specification. This means WebLogic Integration should be able to instantiate an instance of the `WLIResourceAdapter` implementation, and set properties on that instance via simple setter/getter pairs of methods (one pair per property). The properties defined for a `WLIResourceAdapter` implementation can be used support event connections, service connections or both. In most cases, however, properties on the `WLIResourceAdapter` implementation are used to support event connections, whereas service connections are usually supported by properties of the `ManagedConnectionFactory` implementation in the service adapter.

The flow of events is roughly the same as that for the development of a service adapter.

Resource Adapters in a Run-Time Environment

The behavior of a resource adapter in a run-time environment is depicted in [Figure 8-1](#).

Figure 8-1 Resource Adapters in a Run-time Environment



Step 1: Define the Adapter

Before you start developing a resource adapter, you must define your requirements for it. For a complete list of the information you need to do so, see [“Adapter Setup Worksheet” on page C-1](#). This section provides a summary of the most important tasks to be completed for step 1:

1. Decide if your adapter will support events, services, or both:
 - Services do work within an EIS instance
 - Events notify listeners that work has been done in an EIS instance
2. Define properties for `WLIResourceAdapter` implementation

- Define one setter method and one getter method per property on the `WLIResourceAdapter` implementation for each property used to support an event connection.
 - Define one setter method and one getter method per property on the `WLIResourceAdapter` implementation for each property used to support a service connection. In this step you must take into account the properties required for service connections that are already designated to reside directly on the `ManagedConnectionFactory` implementation for the service adapter. You generally will want to have a given property stored in only one place.
3. Define any validation logic needed for the properties you defined
- Some properties may depend on or relate to other properties
 - Optionally, you can implement the `internalValidate` method to get an opportunity to do a final check on all properties for a resource adapter instance

Step 2: Configure the Development Environment

This step involves completing a five-step procedure to prepare your computer for adapter development:

- [Step 2a: Set Up the File Structure](#)
- [Step 2b: Assign a Logical Name to the Adapter](#)
- [Step 2c: Set Up the Build Process](#)
- [Step 2d: Create the Message Bundle](#)
- [Step 2e: Configure Logging](#)

Step 2a: Set Up the File Structure

The file structure needed for an event connection development environment is the same as that required for developing service connections. For detailed information on setting up the file structure, see “[Step 2a: Set Up the Directory Structure](#)” on page 6-7.

Step 2b: Assign a Logical Name to the Adapter

Assign a logical name to your adapter. By convention, this name comprises the vendor name, the type of EIS connected to the adapter, and the version number of the EIS, and it is expressed as `vendor_EIS-type_EIS version`. For example:

BEA_WLS_SAMPLE_ADK

This name includes the following components:

- BEA_WLS is the vendor
- SAMPLE is the EIS type
- ADK is the EIS version

Step 2c: Set Up the Build Process

WebLogic Integration employs a build process based on Ant, a 100% pure Java-based build tool. For more information about how Ant works, see “[Ant-Based Build Process](#)” on page 3-3. For more information about how to use Ant, go to:

<http://jakarta.apache.org/ant/index.html>

The sample adapter provided by WebLogic Integration contains an Ant build file:

`WLI_HOME/adapters/sample/project/build.xml`. This file, in turn, contains the tasks needed to build a J2EE-compliant adapter. When you run the `GenerateAdapterTemplate` utility to clone a development tree for your adapter, a `build.xml` file is created specifically for that adapter. Because this file is generated automatically, you do not need to customize the sample `build.xml` file and you can be sure that the code is correct. For information about using the `GenerateAdapterTemplate` utility, see “[Creating a Custom Development Environment](#)” on page 4-1.

For more information about the build process, see “[Step 2c: Set Up the Build Process](#)” on page 6-11.

Step 2d: Create the Message Bundle

Any message destined for an end-user should be placed in a *message bundle*: a `.properties` text file containing `key=value` pairs that allow you to internationalize messages. When a geographic locale and a natural language are specified for a message at run time, the contents of the message are interpreted on the basis of the `key=value` pair, and the message is presented to the user in the specified language.

For instructions on creating a message bundle, see the JavaSoft tutorial on internationalization at:

<http://java.sun.com/docs/books/tutorial/i18n/index.html>

Step 2e: Configure Logging

Logging is performed with a logging tool called Log4j, which was developed as part of the Apache Jakarta project.

Before you begin this step, we recommend that you read more about logging in [“Basic Development Concepts” on page 2-1](#), and about how to use Log4j in [“Using the Logging Toolkit” on page 5-1](#).

Create an Event Generation Logging Category

If you are planning to use an event connection, you must create a logging category specifically for event generation. (For more information about logging categories, see [“Message Categories” on page 5-3](#).) To edit the logging configuration file for a specific adapter (*WLI_HOME/adapters/YOUR_ADAPTER/src/adapter_logical_name.xml*), add the code shown in the following listing.

Listing 8-1 Sample Code for Creating an Event Generation Logging Category

```
<category name='BEA_WLS_SAMPLE_ADK.EventGenerator'  
    class='com.bea.logging.LogCategory'>  
</category>
```

Replace `BEA_WLS_SAMPLE_ADK` with the logical name of your adapter.

If you do not set any parameters for this category, it inherits all the property settings of the parent category. In this example, the parent category is `BEA_WLS_SAMPLE_ADK`. Although you are not required to use the adapter logical name as the root category, you must use a unique identifier so that there is no impact on other adapters in a multi-adapter environment.

Step 3: Implement the Adapter

To implement a resource adapter, you must complete the following two-step procedure:

1. Create a `WLIResourceAdapter` implementation. This process implements the unified resource adapter implementation and the `WLIResourceAdapter` interface. (The latter interface is used by the application integration engine to control the lifecycle of the resource adapter.) This step is described in [“Step 3a: Create a Resource Adapter.”](#)

2. Implement the property setter/getter methods. This step is described in “[Step 3b: Implement the Resource Adapter Properties.](#)”

Step 3a: Create a Resource Adapter

The `GenerateAdapterTemplate` utility does most of the work necessary to implement your resource adapter class. What remains is to implement the setter/getter methods for each property you chose to define on your `WLIResourceAdapter` implementation. By way of example, we list below the `WLIResourceAdapter` implementation for the `Sample` adapter. This code should look nearly identical to the code you now have in your adapter development project. The difference in your project is that your adapter logical name, package names, and other identifiers will have the values you specified in the `GenerateAdapterTemplate` utility.

Listing 8-2 Sample `WLIResourceAdapter` Implementation

```
package sample.spi;

import com.bea.adapter.spi.AbstractWLIResourceAdapter;
import javax.resource.ResourceException;

// Sample Resource Adapter Implementation

public class ResourceAdapterImpl
    extends AbstractWLIResourceAdapter
{
    // NOTE: If the class named here implements ISuspendableEventGenerator
    //       then the AbstractWLIResourceAdapter implementation will handle
    //       advertising this WLIResourceAdapter instance as suspendable, and
    //       will handle calling suspend/resume on the EventGenerator

    public static String EVENT_GENERATOR_CLASS_NAME =
        "sample.event.EventGenerator";

    // Sample Adapter specific members

    private String m_userName = null;
    private String m_password = null;
    private String m_sleepCount = "4000";

    public ResourceAdapterImpl()
        throws ResourceException
    {
        super();
    }
}
```

Developing a Resource Adapter

```
// Tell AbstractWLIResourceAdapter what EventGenerator we're using
setEventGeneratorClassName(EVENT_GENERATOR_CLASS_NAME);

// Set properties that never change

setAdapterName("BEA_WLS_SAMPLE_ADK Adapter");
setAdapterDescription("__BEA_Sample_Description__");
setAdapterVersion("__MAJOR_V__.__MINOR_V__");
setAdapterVendorName("__BEA__");

// Set default values for standard properties
setRootLogContext("BEA_WLS_SAMPLE_ADK");

setLogConfigFile("BEA_WLS_SAMPLE_ADK.xml");
setMessageBundleBase("BEA_WLS_SAMPLE_ADK");
setLogLevel("WARN");
}

public String getUsername()
{
    return m_username;
}

public void setUsername(String userName)
{
    m_username = userName;
}

public String getPassword()
{
    return m_password;
}

public void setPassword(String password)
{
    m_password = password;
}

}

public String getSleepCount()
{
    return m_sleepCount;
}

public void setSleepCount(String sleepCount)
{
    m_sleepCount = sleepCount;
}

/**
```

```

* TODO: Modify this method to do the proper checking for your adapter's
*       properties. This method is key in forcing users to give you valid
*       configuration for the ResourceAdapter instance.
*/
protected void internalValidate()
    throws ResourceException
{
    if (m_userName == null || m_userName.trim().length() < 1 ||
        m_sleepCount == null || m_sleepCount.trim().length() < 1)
    {
        throw new ResourceException("UserName and SleepCount properties are
            required");
    }

    try
    {
        int sleepCount = Integer.parseInt(m_sleepCount);
        if (sleepCount < 1)
        {
            throw new IllegalArgumentException("sleepCount");
        }
    }
    catch (Exception e)
    {
        throw new ResourceException("SleepCount '" + m_sleepCount + "' is
            invalid");
    }
}
}

```

Note that the constructor of the `ResourceAdapterImpl` calls several setter methods from the `AbstractWLIResourceAdapter` base class. These setters establish default values for the standard properties defined by WebLogic Integration. These standard properties are as follows. For more information on logging properties, see [“Using the Logging Toolkit” on page 5-1](#).

- `EventGeneratorClassName`—the name of the class you define to implement the event generator for your event adapter (optional). If you will not be implementing an event adapter, you can skip setting this property. This will leave its value as the default of null.
- `AdapterName`—adapter display name
- `AdapterVersion`—version to display for the adapter
- `AdapterVendorName`—name of the vendor for this adapter

- `RootLogContext`—logging configuration
- `LogConfigFile`—logging configuration
- `MessageBundleBase`—logging configuration
- `LogLevel`—logging configuration

Step 3b: Implement the Resource Adapter Properties

As you can see, the sample adapter's `ResourceAdapterImpl` class is comprised of a simple constructor, and setter/getter methods to define the following properties:

- `UserName`
- `Password`
- `SleepCount`

These three properties are used to support event connections in the sample adapter. Thus, the `ResourceAdapterImpl` class defines a `setUserName/getUserName` pair of methods, a `setPassword/getPassword` set of methods and a `setSleepCount/getSleepCount` set of methods. Each of these sets of methods operate on a single property. These methods can do simple validity checking such as ensuring an integer value doesn't exceed some set minimum or maximum value, or a string property doesn't exceed some maximum length or violate some syntactical conventions.

However, sometimes individual properties must be compared with other properties on the same object. With individual setter/getter methods, this can become difficult, as you must take into account the order in which properties have been set, and account for properties that have not yet been given valid values.

For this reason, the `WLIResourceAdapter` interface defines a `validate` method that is called after all properties have been set on the `WLIResourceAdapter` instance. This method gives you the opportunity to inspect and validate the values of all properties at one time. The `AbstractWLIResourceAdapter` base class (which implements `WLIResourceAdapter`) defines the `validate` method, and makes a call to a method called `internalValidate` to allow subclasses to hook into the validation process. You should implement `internalValidate` such that it checks all properties for validity and throws `ResourceException` if any invalid properties are encountered.

Step 4: Implement the Event and Service Adapters

For a complete description of how to develop an event adapter, see [“Developing an Event Adapter” on page 7-1](#). For a complete description of how to develop a service adapter, see [“Developing a Service Adapter” on page 6-1](#).

Step 5. Deploy the Adapter

After rebuilding the new adapter, deploy it in a WebLogic Integration environment. You can deploy an adapter either manually or from the WebLogic Server Administration Console. For complete information, see [“Deploying Adapters” on page 10-1](#).

Developing a Resource Adapter

Developing a Design-Time GUI

The ADK's design-time framework provides tools for building a web-based GUI for defining, deploying, and testing adapter users' application views. Although each adapter has EIS-specific functionality, adapters require a GUI for deploying application views. The design-time framework minimizes the effort required to create and deploy such a GUI, primarily through the use of the following components:

- A Web application component that allows you to build an HTML-based GUI by using Java Server Pages (JSP). This component is augmented by tools such as the JSP templates and tag library and the JavaScript library.
- The `DesignTimeHelper` class, which provides a simple API for deploying, undeploying, copying, and editing application views on WebLogic Server.

This section includes information about the following subjects:

- [Introduction to Design-Time Form Processing](#)
- [Design-Time Features](#)
- [File Structure](#)
- [Flow of Events](#)
- [Step 1: Defining the Design-Time GUI Requirements](#)
- [Step 2: Defining the Page Flow](#)
- [Step 3: Configuring the Development Environment](#)

- [Step 4: Implement the Design-Time GUI](#)
- [Step 5: Write the HTML Forms](#)
- [Step 6. Implement the Look and Feel](#)
- [Step 7. Implement Environment Variables](#)
- [Step 8. Test the Sample Adapter Design-Time Interface](#)

Introduction to Design-Time Form Processing

A variety of approaches are available for processing forms using Java Servlets and JSPs. All approaches share several basic requirements, however:

- Displaying an HTML form.
To create this functionality, you must:
 - Generate the form layout using HTML.
 - Indicate to the user which fields are mandatory.
 - Prepopulate fields with defaults, if any.
- Validating the field values in the HTTP request included in the data on a form submitted by a user.

To create this functionality, you must:

- Supply logic that can determine whether all mandatory fields contain a value.
 - Validate each value submitted against a set of constraints. For example, you may want your Web application to determine whether the value in an age field is a valid integer between 1 and 120.
- Redisplaying a form on which an invalid value has been entered, along with an error message beside each erroneous field on the form. If the Web application supports multiple locales, the error message should be localized for the user's preferred locale.

The Web application must also be capable of redisplaying the last input of the user is not required to re-enter valid information. The Web application should continue with Step 2 and loop as many times as necessary until the values entered in all fields are valid.

- Processing the form data after all fields have passed coarse-grained validation. While processing the data, the Web application may encounter an error condition that is unrelated to individual field validation, such as a Java exception. The form must be redisplayed to

the user with a localized error message at the top of the page. As stipulated in step 3, all input fields should be saved so the user is not required to re-enter any valid information.

The Web application developer must determine:

- Which object or method implements the form-processing API.
- How and when to advance the user to the next page in the Web application.
- If the form is processed successfully, the next page in the Web application is displayed to the user.

Form Processing Classes

Implementing all the form-processing functionality for every form in a Web application is a tedious and error-prone process. The ADK design-time framework simplifies this process by using a Model-View-Controller (MVC) paradigm. This paradigm, in turn, is based on the following five classes:

- [RequestHandler](#)
- [ControllerServlet](#)
- [ActionResult](#)
- [Word and Its Descendants](#)
- [AbstractInputTagSupport and Its Descendants](#)

RequestHandler

`com.bea.web.RequestHandler`

This class provides HTTP request-processing logic. It is the model component of the MVC-based mechanism. The `RequestHandler` object is instantiated by the `ControllerServlet` and saved in the HTTP session under the key `handler`. The ADK provides the `com.bea.adapter.web.AbstractDesignTimeRequestHandler`. This abstract base class implements the functionality needed to deploy an application view that is common to all adapters. You must extend this class to supply adapter or EIS-specific logic.

ControllerServlet

com.bea.web.ControllerServlet

This class is responsible for receiving an HTTP request, validating each value in the request, delegating the request to a `RequestHandler` for processing, and determining which page to display to the user. The `ControllerServlet` uses Java reflection to determine which method to invoke on the `RequestHandler`. The `ControllerServlet` looks for an HTTP request parameter named `doAction` to indicate the name of the method that implements the form-processing logic. If this parameter is not available, the `ControllerServlet` does not invoke any methods on the `RequestHandler`.

The `ControllerServlet` is configured in the `web.xml` file for the Web application. The `ControllerServlet` is responsible for delegating HTTP requests to a method on a `RequestHandler`. You are not required to provide any code to use the `ControllerServlet`. However, you must supply the initial parameters listed in [Table 9-5](#).

ActionResult

com.bea.web.ActionResult

`ActionResult` encapsulates the outcome of processing a request. It also provides information to the `ControllerServlet` to help that class determine which page to display next to the user.

Word and Its Descendants

com.bea.web.validation.Word

All fields in a Web application require validation. The `com.bea.web.validation.Word` class and its descendants supply logic to validate form fields. If any fields are invalid, the `Word` object uses a message bundle to retrieve an internationalized or localized error message for the field. The ADK supplies the custom validators described in [Table 9-1](#).

Table 9-1 Custom Validators for Word Object

Validator	Determines whether the value for a field
Integer	Is an integer within a specified range
Float/Double	Is a floating point value within a specified range
Identifier	Is a valid Java identifier

Table 9-1 Custom Validators for Word Object (Continued)

Validator	Determines whether the value for a field
Perl 5 Regular Expression	Matches a Perl 5 regular expression
URL	(Supplied by the user) is a valid URL
Email	(Supplied by the user) contains a list of valid e-mail addresses
Date	(Supplied by the user) is a valid date using a specified date/time forma

AbstractInputTagSupport and Its Descendants

`com.bea.web.tag.AbstractInputTagSupport`

The tag classes provided by the Web toolkit are responsible for:

- Generating the HTML for a form field and prepopulating it with a default value, if applicable.
- Displaying a localized error message beside the form field if the supplied value is invalid.
- Initializing a `com.bea.web.validation.Word` object and saving it in Web application scope so that the validation object is accessible by the `ControllerServlet` using the name of the field on the form.

Submit Tag

Additionally, the ADK provides a submit tag, such as:

```
<adk:submit name='xyz_submit' doAction='xyz' />
```

This tag ensures that the `doAction` parameter is passed to the `ControllerServlet` in the request. As a result, the `ControllerServlet` invokes the `xyz()` method on the registered `RequestHandler`.

Form Processing Sequence

This section discusses the sequence in which forms are processed.

Prerequisites

Before a form can be processed, the following must occur:

1. When a JSP containing a custom ADK input tag is written to an HTTP response object, the tag ensures that the object initializes an instance of `com.bea.web.validation.Word` and places it in the Web application scope, keyed by the input field name. Such a tag makes the validation object available to the `ControllerServlet` so that it can perform coarse-grained validation on an HTTP request before submitting the request to the `RequestHandler`. For example:

```
<adk:int name='age' minInclusive='1' maxInclusive='120'  
required='true' />
```

2. The HTML for this tag is generated when the JSP engine invokes the `doStartTag()` method on an instance of `com.bea.web.tag.IntegerTagSupport`. The `IntegerTagSupport` instance instantiates a new instance of `com.bea.web.validation.IntegerWord` and adds it to Web application scope under the key `age`. Consequently, the `ControllerServlet` can retrieve the `IntegerWord` instance from its `ServletContext` whenever it must validate a value for `age`. The validation ensures that any value passed for `age` is greater than or equal to one, and less than or equal to 120.
3. The HTML form must also submit a hidden field named `doAction`. The value of this field is used by the `ControllerServlet` to determine which method on the `RequestHandler` can process the form.

Once these prerequisites are met, the JSP form is displayed, as shown in [Listing 9-1](#).

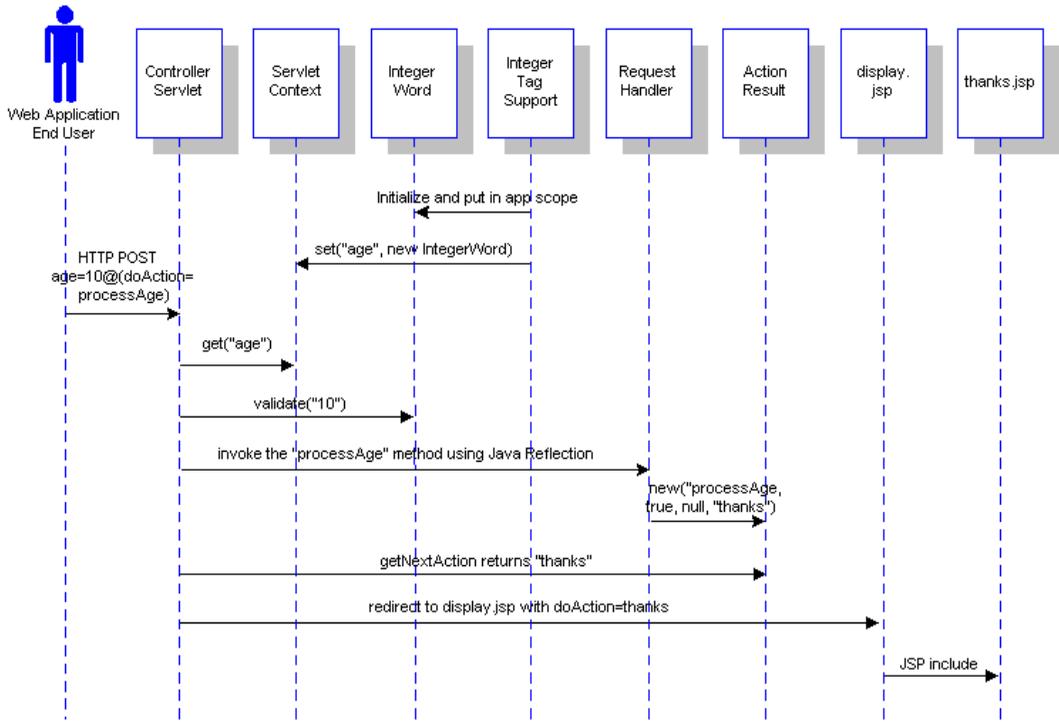
Listing 9-1 Sample JSP Form

```
<form method='POST' action='controller'>  
  Age: <adk:int name='age' minInclusive='1' maxInclusive='120'  
    required='true' />  
  <adk:submit name='processAge_submit' doAction='processAge' />  
</form>
```

Steps in the Sequence

The following diagram illustrates, step by step, how form processing is performed.

Figure 9-1 UI Form Processing



The sequence is as follows:

1. A user submits a form with the following data: `age=10, doAction=processAge`.
2. **ControllerServlet** retrieves the `age` field from the **HTTP** request.
3. **ControllerServlet** retrieves a `com.bea.web.validation.Word` object from its **ServletContext** using `age` as the key. The object is an instance of `com.bea.web.validation.IntegerWord`.
4. The **ControllerServlet** invokes the `validate()` method on the `Word` instance and passes `10` as a parameter.
5. The `Word` instance determines that the value `10` is greater than or equal to `1`, and it is less than or equal to `120`. The `Word` instance returns `true` to indicate that the value is valid.

6. The `ControllerServlet` retrieves the `RequestHandler` from the session (or creates it) and adds it to the session as handler.
7. The `ControllerServlet` uses the Java Reflection API to locate and invoke the `processAge()` method on the `RequestHandler`. An exception is generated if the method does not exist. The method signature is:

```
public ActionResult processAge(HttpServletRequest request) throws
Exception
```

8. The `RequestHandler` processes the form input and returns an `ActionResult` object to indicate the outcome of the processing. The `ActionResult` contains information used by the `ControllerServlet` to determine the which page to display next to the user. The next page information should be the name of another JSP or HTML page in your Web application. For example, `thanks` might display the `thanks.jsp` page to the user.
9. If the `ActionResult` is a success, then the `ControllerServlet` redirects the HTTP response to the display page for the Web application. In the ADK, the display page is typically `display.jsp`.
10. The `display.jsp` page includes the JSP indicated by the `content` parameter (for example, `thanks.jsp`). It displays that JSP to the user.

Design-Time Features

Design-time development has its own features, different from those associated with run-time adapter development. This section describes those features.

Java Server Pages

A design-time GUI comprises a set of Java Server Pages (JSPs). JSPs are simply HTML pages that call Java servlets to invoke a transaction. To the user, a JSP looks like any other web page.

The following table describes the JSPs that make up a design-time GUI.

Table 9-2 Design-Time GUI JSPs

Filename	Description
<code>display.jsp</code>	The display page, also called the Adapter Home Page, contains the HTML necessary to create the look-and-feel.
<code>login.jsp</code>	The Adapter Design-Time Login page.

Table 9-2 Design-Time GUI JSPs (Continued)

Filename	Description
<code>confconn.jsp</code>	The Confirm Connection page provides a form on which the user can specify connection parameters for the EIS.
<code>appvwadmin.jsp</code>	The Application View Administration page provides a summary of an undeployed application view.
<code>addevent.jsp</code>	The Add Event page allows the user to add an event to the application view.
<code>addservc.jsp</code>	The Add Service page allows the user to add a service to the application view.
<code>edtevent.jsp</code>	The Edit Event page is an optional page that allows users to edit events.
<code>edtservc.jsp</code>	The Edit Service page is an optional page that allows users to edit services.
<code>depappvw.jsp</code>	The Deploy Application View page allows users to specify deployment properties.

For a discussion of how to implement these JSPs, see [“Step 2: Defining the Page Flow” on page 9-17](#).

JSP Templates

A template is an HTML page that is dynamically generated by a Java Servlet based on parameters provided in an HTTP request. Templates are used to minimize the number of custom pages and the amount of custom HTML needed for a Web application.

The design-time framework provides a set of JSP templates for rapidly assembling a Web application to define, deploy, and test a new application view for an adapter. The templates supplied by the ADK offer three advantages to adapter developers:

- They provide most of the HTML forms needed to deploy an application view. In most cases, you need to supply only three custom forms:
 - Form that collects the EIS-specific connection parameters

- Form that collects the EIS-specific information needed to add an event. You can use either the same form or a different form to collect the information needed to edit an event.
- Form that collects the EIS-specific information needed to add a service. You also have the option of supplying a JSP for browsing a metadata catalog for an EIS. You can use either the same form or a different form to collect the information needed to edit a service.
- They leverage the internationalization and localization features of the Java platform. The content of every page in the Web application is stored in a message bundle. Consequently, the web interface for an adapter can be internationalized quickly.
- They guarantee a consistent look and feel for all templates

For a complete list of JSP templates provided by the ADK, see [“JSP Templates” on page 9-9](#).

ADK Library of JSP Tags

The custom JSP tag library provided by the ADK helps developers create user-friendly HTML forms. Custom tags for HTML form input components allow page developers to seamlessly link to a validation mechanism. The following table describes the custom tags provided by the ADK.

Table 9-3 ADK JSP Tags

Tag	Description
<code>adk:check_box</code>	Determines whether the checkbox form field should be checked when a form is displayed. (This tag does not perform validation.)
<code>adk:content</code>	Provides access to a message in a message bundle.
<code>adk:date</code>	Verifies that the user's input is a date value in a specific format.
<code>adk:double</code>	Verifies that the user's input is a double value.
<code>adk:email</code>	Verifies that the user's input is a valid list of e-mail addresses (one or more).
<code>adk:float</code>	Verifies that the user's input is a float value.
<code>adk:identifier</code>	Verifies that the user's input is a valid Java identifier.
<code>adk:int</code>	Verifies that the user's input is an integer value.

Table 9-3 ADK JSP Tags (Continued)

Tag	Description
<code>adk:label</code>	Displays a label from the message bundle.
<code>adk:password</code>	Verifies the user's input in a text field against a Perl 5 regular expression and marks the input with an asterisk (*).
<code>adk:submit</code>	Links the form to a validation mechanism.
<code>adk:text</code>	Verifies the user's input against a Perl 5 regular expression.
<code>adk:textarea</code>	Verifies that the user's input in a text area matches a Perl 5 regular expression.
<code>adk:url</code>	Verifies that the user's input is a valid URL.

JSP Tag Attributes

You can further customize the JSP tags by applying the attributes listed in [Table 9-4](#).

Table 9-4 JSP Tag Attributes

Tag	Requires Attributes	Optional Attributes
adk:int, adk:float, adk:double	name - field name	default - value displayed on the page by default maxlength - maximum length of value size - size of display minInclusive - value supplied by user must be greater than or equal to this value maxInclusive - value supplied by user must be less than or equal to this value minExclusive - value supplied by user must be strictly greater than this value maxExclusive - value supplied by user must be strictly less than this value required - true or false (default is false, meaning field is not required) attrs - additional HTML attributes
adk:date	name - field name	default - value displayed on the page by default maxlength - maximum length of value size - size of display required - true or false (default is false, meaning field is not required) attrs - additional HTML attributes lenient - true or false (default is false, meaning the date formatter should not be lenient in its parsing) format - expected format of the user input (default is <i>mm/dd/yyyy</i>)
adk:email, adk:url, adk:identifier	name - field name	default - value displayed on the page by default maxlength - maximum length of value size - size of display required - true or false (default is false, meaning field is not required) attrs - additional HTML attributes

Table 9-4 JSP Tag Attributes (Continued)

Tag	Requires Attributes	Optional Attributes
adk:text, adk:password	name - field name	default - value displayed on the page by default maxlength - maximum length of value size - size of display required - true or false (default is false, meaning field is not required) attrs - additional HTML attributes pattern - a Perl 5 regular expression
adk:textarea	name - field name	default - value displayed on the page by default required - true or false (default is false, meaning field is not required) attrs - additional HTML attributes pattern - a Perl 5 regular expression rows - number of rows to be displayed columns - number of columns to be displayed

Note: For more information about tag usage, see `adk.tld` in:

`WLI_HOME/adapters/src/war/WEB-INF/taglibs`

The Application View

An application view is a business-level interface to the functionality specific to an application. For more information, see [“Application Views” on page 1-6](#).

File Structure

The file structure necessary to build a design-time GUI adapter is the same as that required for service connections. See [“Step 2a: Set Up the Directory Structure” on page 6-7](#). In addition to the structure described there, you should also be aware that:

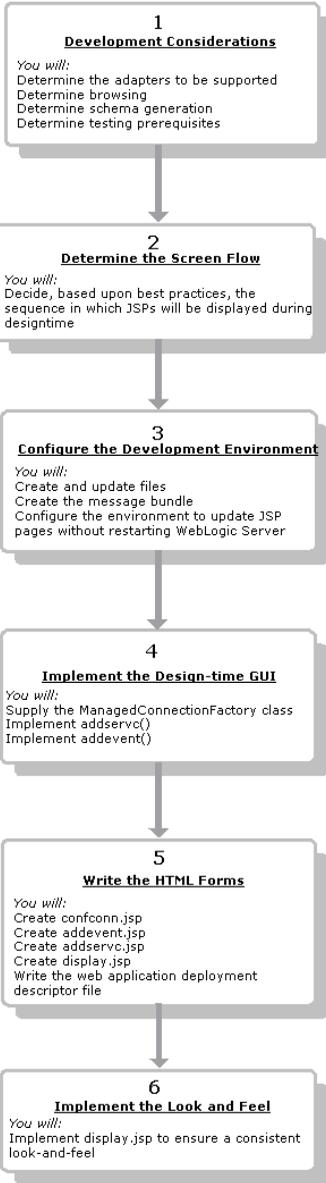
- The design-time interface for each adapter is a J2EE Web application that is bundled as a WAR file.
- A Web application is a bundle of `.jsp`, `.html`, and image files.

- The Web application descriptor is `WLI_HOME/adapters/ADAPTER/src/war/WEB-INF/web.xml`. This descriptor instructs the J2EE web container how to deploy and initialize the Web application.

Flow of Events

[Figure 9-2](#) outlines the steps required to develop a design-time GUI.

Figure 9-2 Design-Time GUI Development Flow of Events



Step 1: Defining the Design-Time GUI Requirements

Before you start developing your design-time GUI, you must define your requirements for it by answering the following questions:

- Will this GUI support event connections? Service connections? Both?
- How does the user browse event and service catalogs?

The EIS must supply functions to access the event and service catalogs. If the EIS does not supply these, the user cannot browse the catalogs. If the EIS does supply them, we recommend the following design principle: invoke a call from the design-time UI to get metadata from the EIS. Such a call is really no different from a call from a run-time component. Both types of call execute functions on the back-end EIS.

Consequently, you need to leverage your run-time architecture as much as possible to provide design-time metadata features. You should invoke design-time-specific functions that use a CCI Interaction object. The sample adapter included with the ADK provides an example or framework of this approach. You can find the sample adapter in

WLI_HOME/adapters/sample.

- How will the adapter generate the request/response schema for a service? Will it make a call to the EIS or use some other methodology?

Generally, an adapter must call the EIS to get metadata about a function or event. The adapter then transforms the EIS metadata into XML schema format. To make this process happen, you must invoke the SOM API. Again, the sample adapter provides instructions for implementing the SOM API. For more information about this API, see [“ADK Library of JSP Tags” on page 9-10.](#)

- Will some sort of service testing be supported? If your design-time GUI will support service testing, you must provide:

- A class that transforms the XML response schema into an HTML form. For an example, see:

WLI_HOME/adapters/dbms/docs/api/com/bean/adapter/dbms/utills/class-use/TestFormBuilder.html

A JSP named `testform.jsp` that invokes the transformation and displays the HTML form. To see an example of this file, see: *WLI_HOME/adapters/dbms/src/war/*

- Will environment variables be supported? This may be the case if your event or service property values can contain environment specific identifiers or other information. Environment variable support can be added using JSPs and request handler methods provided by the ADK.

Step 2: Defining the Page Flow

You must specify the order in which the JSPs will be displayed when the user invokes an application view. This section describes the basic, required flow of pages for a successful application view. Note that these requirements are minimal; you can also add pages to the flow to meet your specific needs.

Page 1: Logging In

Because an application view is a secure system, the user must log in before implementing the view. Thus, the Application Integration Design Console Logon page must be the first page the user sees.

To use this page, the user supplies a valid username and password. That information is then validated to ensure that the user is a member of the adapter group in the default WebLogic Server security realm.

Note: The security requirements for Application View Web applications are specified in the `WLI_HOME/adapters/ADAPTER/src/war/WEB-INF/web.xml` file, which is available in the `adapter.war` file.

Page 2. Managing Application Views

Once the user successfully logs in, the Application Integration Design Console page is displayed. This page lists the folders that contain application views, the status of these folders, and any actions taken on them. From this page, the user can either view existing application views or add new ones.

- To view an existing application view, the user selects the appropriate folder and drills down to the desired application view. The user then selects the desired application view and the Application View Summary page is displayed (`appvwsum.jsp`). For details about this page, see [“Page 9: Summarizing an Application View” on page 9-21](#)).
- To add a new application view, the user clicks Add Application View. The Define New Application View page is displayed.

Page 3: Defining the New Application View

The Define New Application View page (`defappvw.jsp`) allows the user to define a new application view in any folder in which the client is located. To do this, the user must provide a description that associates the application view with an adapter. This form provides fields in

which the user can enter the application view name and a description of it, and a drop-down list of adapters with which the user can associate the application view.

Once the new application view is defined, the user selects OK and the Configure Connection page is displayed.

Page 4: Configuring the Connection

If the new application view is valid, the user must configure the connection. Therefore, once the application view is validated, the Configure Connection Parameters page (`confconn.jsp`) should be displayed. This page provides a form on which the user can specify connection parameters for the EIS. Because connection parameters are EIS-specific, the appearance of this page differs from one adapter to another.

When the user submits the connection parameters, the adapter attempts to open a new connection to the EIS using the parameters. If it succeeds, the user is forwarded to the next page, Application View Administration.

Page 5: Administering the Application View

The user needs a means of administering the new application view. The Application View Administration page (`appvwadmin.jsp`) provides a summary of an undeployed application view. Specifically, it shows the following:

- Connection criteria—The connection criteria section provides a link that returns the user to the Configure Connection page so that he or she can change connection parameters.
- List of events—For each event listed in the application view, the user can do the following:
 - View the XML schema
 - Remove the event
 - Provide event properties
- List of services—For each service listed in the application view, the user can do the following:
 - View the request XML schema
 - View the response XML schema
 - Remove the service
 - Provide service properties

In addition to providing a list of events and a list of services in the application view, the page provides a link to a page that allows you to add a new event or service.

Page 6: Adding an Event

Now the user needs to add events to the application view. The Add Event page (`addevent.jsp`) allows the user to do so.

The following rules apply to a new event:

- Every event must have a unique name.
 - The event name can contain only the following characters: a-z, A-Z, 0-9, and underscore (`_`). It must begin with a letter. No other characters are valid.
 - The length of the name may not exceed 256 characters.
 - The event name must be unique within the application view. If the user specifies an event name that is not unique, the form is reloaded with an error message indicating that the event is already defined.
- Optionally, the user can provide a description of the event. This description cannot exceed 2048 (2K) characters.
- In addition to a name and a description, every event requires EIS-specific parameters. The collection of EIS-specific parameters defines an event type for the adapter.
- Optionally, the adapter can provide a mechanism for browsing the event catalog for an EIS.
- Optionally, the adapter can allow the user to define environment variables for use in the configuration of the event. For more details, see [“Step 7. Implement Environment Variables” on page 9-41](#).

After defining and saving a new event, the user is returned to the Application View Administration page.

Page 7: Adding a Service

The user also needs to add new services to an application view. The Add Service page (`addservc.jsp`) allows the user to do so.

The following rules apply to a new event:

- Every service must have a unique name.

- The service name can contain only the following characters: a-z, A-Z, 0-9, and underscore (_). It must begin with a letter. No other characters are valid.
- The length of the name may not exceed 256 characters.
- The service name must be unique within the application view. If the user specifies a service name that is not unique, the form is reloaded with an error message indicating that the service is already defined.
- Optionally, the user can provide a description of the service. This description cannot exceed 2048 (2K) characters.
- In addition to a name and a description, every service requires EIS-specific parameters. The collection of EIS specific parameters defines an service type for the adapter.
- Optionally, the adapter can provide a mechanism for browsing the service catalog for an EIS.
- Optionally, the adapter can allow the user to define environment variables for use in the configuration of the service. For more information, see [“Step 7. Implement Environment Variables” on page 9-41](#).

After defining and saving a new service, the user is returned to the Application View Administration page.

Page 8: Testing an Application View

After adding at least one service or event, the user can test the application view. When an application view is tested, it becomes available to process events and services for testing purposes. If the user chooses to test the application view, he or she has the following two choices:

- Test—Go directly to the Application View Summary page and begin testing the application view.
- Set Variables and Test—Optionally, set any environment variable values to non-default values, and then proceed to the Application View Summary page to begin testing the application view (`depappvw.jsp`).

Publishing an Application View

Once the user has successfully tested an application view, they can publish it from the Application View Summary page, thus making it available for use from within WebLogic Workshop applications.

Saving an Application View

The user can save an application view (even one which is untested and unpublished) and return to it later via the Application Integration Design Console. This process saves the application view to the application integration repository within your target WebLogic Workshop application. Testing an application view will automatically save it, so this step is only necessary if you wish to leave a design session before actually testing the application view.

Page 9: Summarizing an Application View

When an application view is deployed successfully, the user is forwarded to the Application View Summary page (`appvwsum.jsp`). This page provides the following information about an application view:

- Testing state (testing or not testing)

- If the application view is deployed:

The page includes an option to stop testing the application view. If the user clicks the Stop Testing link, a child window is displayed, prompting the user to confirm this choice. If the user confirms, the application view test is stopped and the summary page is redisplayed. Application views that are not being tested in this way continue to be saved in the repository. As a result, the user can edit or remove the application view.

If the adapter supports the testing of events, the Summary page displays a test link for each event. Testing of events is not supported directly by the ADK.

If the adapter supports the testing of services, the Summary page displays a test link for each service. The ADK demonstrates one possible approach to testing services by providing the `testsvc.jsp` and `testrslt.jsp` files. You are free to use these pages to devise your own service testing strategy.

- If the application view is not deployed:

The page includes an option to test the application view. If the user clicks the Test link, the application view test is started and the Application View Summary page is reloaded.

The page includes an option to edit the application view. If the user clicks the Edit link, the Application View Administration page is displayed.

The page includes an option to remove the application view. If the user clicks the Remove link, a child window is displayed, prompting the user to confirm the choice to remove the application view from the ADK repository. If the user confirms, the

application view is deleted from the repository and the user is redirected to the Application Integration Design Console.

- Event and service connection information (pooling configuration, log level, and security)
- List of events: For each event, the Summary page offers the option of viewing the schema and, if event testing is supported, the option of testing. The user cannot remove events from this page; instead the user must choose to edit first.
- List of services: For each service, the Summary page offers the option of viewing the request schema and the response schema, and, if service testing is supported, the option of testing. The user cannot remove services from this page; instead the user must undeploy and edit first.

Step 3: Configuring the Development Environment

In this step, you set up your software environment to support design-time GUI development.

Step 3a: Create the Message Bundle

Any message destined for an end-user should be placed in a *message bundle*. A message bundle is simply a `.properties` text file that contains *key=value* pairs that allow you to internationalize messages. When a locale and national language are specified for a message at run time, the contents of the message are interpreted, on the basis of the *key=value* pair, and the message is presented to the user in the language appropriate for the specified locale.

For instructions on creating a message bundle, see the JavaSoft tutorial on internationalization at:

<http://java.sun.com/docs/books/tutorial/i18n/index.html>

Step 3b: Configure the Environment to Update JSPs Without Restarting WebLogic Server

The design-time UI is deployed as a J2EE Web application from a WAR file. A WAR file is simply a JAR file with a Web application descriptor in `WEB-INF/web.xml` in the JAR file. However, the WAR file does not allow the J2EE Web container in WebLogic Server to recompile JSPs on the fly. Consequently, you normally have to restart WebLogic Server just to change a JSP file. Because this approach contradicts the spirit of JSP, the ADK suggests the following workaround for updating JSPs without restarting WebLogic Server:

1. Construct a valid WAR file for your adapter's design-time UI. For the sample adapter, you can do so by using Ant. [Listing 9-2](#) shows the target that produces the J2EE WAR file.

Listing 9-2 Target that Creates a WAR File

```

<target name='war' depends='jar'>

<!-- Clean-up existing environment -->
  <delete file='${LIB_DIR}/${WAR_FILE}'/>

<war warfile='${LIB_DIR}/${WAR_FILE}'
webxml='${SRC_DIR}/war/WEB-INF/web.xml'
manifest='${SRC_DIR}/war/META-INF/MANIFEST.MF'>

<!--
IMPORTANT! Exclude the WEB-INF/web.xml file from the WAR as it
already gets included via the webxml attribute above
-->
  <fileset dir="${SRC_DIR}/war" >
    <patternset >
      <include name="WEB-INF/weblogic.xml"/>
      <include name="**/*.html"/>
      <include name="**/*.gif"/>
    </patternset>
  </fileset>

<!--
IMPORTANT! Include the ADK design time framework into the adapter's
design time Web application.
-->
  <fileset dir="${WLI_HOME}/adapters/src/war" >
    <patternset >
      <include name="**/*.css"/>
      <include name="**/*.html"/>
      <include name="**/*.gif"/>
      <include name="**/*.js"/>
    </patternset>
  </fileset>

<!--
Include classes from the adapter that support the design time UI

```

```
-->
<classes dir='${SRC_DIR}' includes='sample/web/*.class' />

<classes dir='${SRC_DIR}/war' includes='**/*.class' />
<classes dir='${WLI_HOME}/adapters/src/war'
  includes='**/*.class' />

<!--
Include all JARs required by the Web application under the
WEB-INF/lib directory of the WAR file that are not shared in the EAR
-->
<lib dir='${WLI_LIB_DIR}'
  includes='adk-web.jar,webtoolkit.jar,wlai-client.jar' />
  </war>
</target>
```

This Ant target constructs a valid WAR file for the design-time interface in the *PROJECT_ROOT/lib* directory, where *PROJECT_ROOT* is the location under the WebLogic Integration installation where the developer is constructing the adapter; for example, the DBMS sample adapter is being constructed in:

```
WLI_HOME/adapters/DBMS
```

2. Load your Web application into WebLogic Server using the WebLogic Server Administration Console.
3. Configure the development environment. Sample development environment information is shown in [Listing 9-3](#).

Listing 9-3 Name of Adapter Development Tree

```
<Application Deployed="true" Name="BEA_WLS_SAMPLE_ADK_Web"
  Path="WLI_HOME\adapters\PROJECT_ROOT\lib">
  <WebAppComponent Name="BEA_WLS_SAMPLE_ADK_Web"
    ServletReloadCheckSecs="1" Targets="myserver" URI=
      "BEA_WLS_SAMPLE_ADK_Web" />
</Application>
```

Set the adapter logical name and directory values as follows:

- a. Replace `BEA_WLS_SAMPLE_ADK_Web` with the logical name of your adapter.
- b. Replace `WLI_HOME` with the pathname of the directory in which WebLogic Integration is installed. Replace `PROJECT_ROOT` with the name of the top-level directory of your adapter development tree, as shown in [Listing 9-3](#).

Note: If you run `GenerateAdapterTemplate`, the information in [Listing 9-3](#) is updated automatically. You can then open `WLI_HOME/adapters/ADAPTER/src/overview.html`, copy this information and paste the copy into your `config.xml` entry.

4. To change a JSP, do so in the `src/war` directory and then rebuild the WAR target. Do not change a JSP in the temporary directory. When the WAR file is created, it is also extracted into the directory monitored by WebLogic Server, which picks up changes only to a specific JSP. The duration of the monitoring operation performed by WebLogic Server is set by the `pageCheckSeconds` parameter in `WEB-INF/weblogic.xml`. [Listing 9-4](#) shows how this parameter is set.

Listing 9-4 Setting the Monitoring Interval

```
<jsp-descriptor>
  <jsp-param>
    <param-name>compileCommand</param-name>
    <param-value>/jdk130/bin/javac.exe</param-value>
  </jsp-param>
  <jsp-param>
    <param-name>keepgenerated</param-name>
    <param-value>>true</param-value>
  </jsp-param>
  <jsp-param>
    <param-name>pageCheckSeconds</param-name>
    <param-value>1</param-value>
  </jsp-param>
  <jsp-param>
    <param-name>verbose</param-name>
    <param-value>>true</param-value>
  </jsp-param>
</jsp-descriptor>
```

This approach also tests whether your WAR file is being constructed correctly.

Step 4: Implement the Design-Time GUI

Implementing the procedure provided in “[Introduction to Design-Time Form Processing](#)” for every form in a Web application is a tedious and error-prone process. The design-time framework simplifies this process by supporting a Model-View-Controller paradigm.

To implement the design-time GUI, you must implement the `DesignTimeRequestHandler` class. This class accepts user input from a form and performs a design-time action. To implement this class, you must extend the `AbstractDesignTimeRequestHandler` provided with the ADK. For a detailed overview of the methods provided by this object, see the Javadoc for the `DesignTimeRequestHandler` class.

Extend `AbstractDesignTimeRequestHandler`

The `AbstractDesignTimeRequestHandler` provides utility classes for deploying, editing, copying, and removing application views on the WebLogic Server. It also provides access to an application view descriptor. The application view descriptor provides the connection parameters, list of events, list of services, log levels, and pool settings for an application view. The parameters are shown on the Application View Summary page.

At a high level, the `AbstractDesignTimeRequestHandler` provides an implementation for all actions that are common to all adapters. These actions include:

- Defining an application view
- Configuring the connection

Note: The ADK provides a method for processing connection parameters to obtain a CCI connection, but it does not supply the `confconn.jsp` page. For instructions on creating this form, see “[Step 5a: Create the confconn.jsp Form](#)” on page 9-29.

- Deploying an application view
- Providing application view security
- Editing an application view
- Undeploying an application view

Methods to Include

To ensure that these actions are performed successfully, you must supply the following methods when you implement `AbstractDesignTimeRequestHandler`:

- `initServiceDescriptor()`;

This method adds a service to an application view at design time. (See “[Step 4b. Implement `initServiceDescriptor\(\)`](#)” on page 9-27.)

- `initEventDescriptor()`;

This method adds an event to an application view at design time. (See “[Step 4c. Implement `initEventDescriptor\(\)`](#)” on page 9-28.)

In every concrete implementation of `AbstractDesignTimeRequestHandler`, you also need to provide the following two methods:

- `protected String getAdapterLogicalName()`;

This method returns the logical name of the adapter. It is used to deploy an application view under that name.

- `protected Class getManagedConnectionFactoryClass()`;

This method returns the SPI `ManagedConnectionFactory` implementation class for the adapter.

Step 4a. Supply the `ManagedConnectionFactory` Class

To supply the `ManagedConnectionFactory` class, you need to implement the following method:

```
protected Class getManagedConnectionFactoryClass();
```

This method returns the SPI `ManagedConnectionFactory` implementation class for the adapter. This class is needed by the `AbstractManagedConnectionFactory` when it tries to get a connection to the EIS.

Step 4b. Implement `initServiceDescriptor()`

For service connections, you need to implement `initServiceDescriptor()` so that the adapter user can add services at design time. This method is implemented as shown in [Listing 9-5](#).

Listing 9-5 `initServiceDescriptor()` Implementation

```
protected abstract void initServiceDescriptor(ActionResult result,
                                             IServiceDescriptor sd,
                                             HttpServletRequest request)
    throws Exception
```

This method is invoked by the `addservc()` implementation of the `AbstractDesignTimeRequestHandler`. It is responsible for initializing the EIS-specific information associated with the `IServiceDescriptor` parameter. The base class implementation of `addservc()` handles error handling, and so on. The `addservc()` method is invoked when the user submits the `addservc` JSP.

Step 4c. Implement `initEventDescriptor()`

For event connections, you must implement `initEventDescriptor()` so that the adapter user can add events at design time. This method is implemented as shown in [Listing 9-6](#).

Listing 9-6 `initEventDescriptor()` Implementation

```
protected abstract void
    initEventDescriptor(ActionResult result,
                        IEventDescriptor ed,
                        HttpServletRequest request)
    throws Exception;
```

This method is invoked by the `addevent()` implementation of the `AbstractDesignTimeRequestHandler`. It is responsible for initializing the EIS-specific information associated with the `IServiceDescriptor` parameter. The base class implementation of `addevent()` handles such concepts as error handling. The `addevent()` method is invoked when the user submits the `addevent` JSP. You should not override `addevent()`, as it contains common logic and delegates EIS-specific logic to `initEventDescriptor()`.

Note: When adding properties to a service descriptor, make sure that the names you give them conform to the bean attribute naming standard. When property names do not conform to that standard, the service descriptor does not update the `InteractionSpec` correctly.

Step 5: Write the HTML Forms

The final step to implementing a design-time GUI is to write the various forms that the interface comprises. To familiarize yourself with the forms you must create, see the following sections:

- See “[Java Server Pages](#)” on page 9-8 for a list of the necessary forms and a high-level description of them.
- See “[Step 2: Defining the Page Flow](#)” on page 9-17 for details about each form.

The following sections describe how to write code for these forms. A sample of code for a form is included.

Step 5a: Create the confconn.jsp Form

This page provides an HTML form for users to supply connection parameters for the EIS. You are responsible for providing this page with your adapter’s design-time Web application. This form posts to the `ControllerServlet` with `doAction=confconn`. This implies that the `RequestHandler` for your design-time interface must provide the following method:

```
public ActionResult confconn(HttpServletRequest request) throws
    Exception
```

The implementation of this method is responsible for using the supplied connection parameters to create a new instance of the adapter’s `ManagedConnectionFactory`. The `ManagedConnectionFactory` supplies the `CCI ConnectionFactory`, which is used to obtain a connection to the EIS. Consequently, the processing of the `confconn` form submission verifies that the supplied parameters are sufficient for obtaining a valid connection to the EIS.

The `confconn` form for the sample adapter is shown in [Listing 9-7](#).

Listing 9-7 Coding confconn.jsp

```
1 <%@ taglib uri='/WEB-INF/taglibs/adk.tld' prefix='adk' %>
2 <form method='POST' action='controller'>
3   <table>
4     <tr>
5       <td><adk:label name='userName' required='true' /></td>
6       <td><adk:text name='userName' maxlength='30' size='8' /></td>
7     </tr>
8     <tr>
9       <td><adk:label name='password' required='true' /></td>
10      <td><adk:password name='password' maxlength='30' size='8' /></td>
11    </tr>
12    <tr>
13      <td colspan='2'><adk:submit name='confconn_submit'
14        doAction='confconn' /></td>
```

```
15 </table>  
16 </form>
```

The following sections describe the contents of [Listing 9-7](#):

- [Including the ADK Tag Library](#)
- [Posting the ControllerServlet](#)
- [Displaying the Label for the Form Field](#)
- [Displaying the Text Field Size](#)
- [Displaying a Submit Button on the Form](#)
- [Implementing confconn\(\)](#)

Including the ADK Tag Library

Line 1 in [Listing 9-7](#) instructs the JSP engine to include the ADK tag library:

```
<%@ taglib uri='/WEB-INF/taglibs/adk.tld' prefix='adk' %>
```

The tags provided by the ADK are listed in [Table 9-3](#).

Posting the ControllerServlet

Line 2 in [Listing 9-7](#) instructs the form to post to the ControllerServlet:

```
<form method='POST' action='controller'>
```

The ControllerServlet is configured in the `web.xml` file for the Web application. It is responsible for delegating HTTP requests to a method on a RequestHandler. You are not required to provide any code to use the ControllerServlet; however, you must supply the initial parameters, which are described in [Table 9-5](#):

Table 9-5 Initial Parameters for ControllerServlet

Parameter	Description
<code>MessageBundleBase</code>	Specifies the base name for all message bundles supplied with an adapter. The ADK always uses the logical names for its sample adapters. However, you are free to choose your own naming convention for message bundles. This property is also established in the <code>ra.xml</code> file.
<code>DisplayPage</code>	Specifies the name of the JSP that controls both the flow and the look-and-feel of the pages in the application. In the sample adapter, this page is <code>display.jsp</code> .
<code>LogConfigFile</code>	Specifies the log4j configuration file for the adapter.
<code>RootLogContext</code>	Specifies the root log context. Log context is helpful for classifying log messages according to modules in a program. The ADK uses the adapter logical name for the root log context so that all messages from a specific adapter are classified accordingly.
<code>RequestHandlerClass</code>	Provides the fully qualified name of the request handler class for the adapter. In the sample adapter, this value is <code>sample.web.DesignTimeRequestHandler</code> .

Displaying the Label for the Form Field

Line 5 in [Listing 9-7](#) displays a label for a field on the form:

```
<adk:label name='userName' required='true' />
```

The value that is displayed is retrieved from the message bundle for the user. The `required` attribute indicates whether the user must supply this parameter to be successful.

Displaying the Text Field Size

Line 6 in [Listing 9-7](#) sets a text field of size 8 with a maximum length (max length) of 30:

```
<adk:text name='userName' maxLength='30' size='8' />
```

Displaying a Submit Button on the Form

Line 13 in [Listing 9-7](#) displays a button on the form that allows an adapter user to submit input:

```
<adk:submit name='confconn_submit' doAction='confconn' />
```

The label on the button is retrieved from the message bundle using the `confconn_submit` key. When the form data is submitted, the `ControllerServlet` locates the `confconn` method on the registered request handler (see the `RequestHandlerClass` property) and passes the request data to it.

Implementing `confconn()`

The `AbstractDesignTimeRequestHandler` provides an implementation of the `confconn()` method. This implementation leverages the Java Reflection API to map connection parameters supplied by the user to setter methods on the adapter's `ManagedConnectionFactory` instance. You need to supply only one item: the concrete class for your adapter's `ManagedConnectionFactory`. To provide this class, implement the following method:

```
public Class getManagedConnectionFactoryClass()
```

Step 5b: Create the `addevent.jsp` form

This form allows a user to add a new event to an application view. The form is EIS-specific. The `addevent.jsp` form for the sample adapter is shown in [Listing 9-8](#).

Listing 9-8 Sample Code Creating the `addevent.jsp` Form

```
1 <%@ taglib uri='/WEB-INF/taglibs/adk.tld' prefix='adk' %>
2 <form method='POST' action='controller'>
3     <table>
4         <tr>
5             <td><adk:label name='eventName' required='true' /></td>
6             <td><adk:text name='eventName' maxLength='100' size='50' /></td>
7         </tr>
8         <tr>
9             <td colspan='2'><adk:submit name='addevent_submit'
10                doAction='addevent' /></td>
11         </tr>
12     </table>
</form>
```

The following sections describe the contents of `addevent.jsp`.

Including the ADK Tag Library

Line 1 in [Listing 9-8](#) instructs the JSP engine to include the ADK tag library.

```
<%@ taglib uri='/WEB-INF/taglibs/adk.tld' prefix='adk'%>
```

The tags provided by the ADK are described in [Table 9-3](#).

Posting the ControllerServlet

Line 2 in [Listing 9-8](#) instructs the form to post to the `ControllerServlet`.

```
<form method='POST' action='controller'>
```

The `ControllerServlet` is configured in the `web.xml` file for the Web application. It is responsible for delegating HTTP requests to a method on a `RequestHandler`. You are not required to provide any code to use the `ControllerServlet`; however, you must supply the initial parameters, as described in [Table 9-5](#), “`ControllerServlet` Parameters.”

Displaying the Label for the Form Field

Line 5 in [Listing 9-8](#) displays a label for a field on the form.

```
<adk:label name='eventName' required='true' />
```

The value that is displayed is retrieved from the message bundle for the user. The `required` attribute indicates whether the user must supply this parameter to be successful.

Displaying the Text Field Size

Line 6 in [Listing 9-8](#) sets a text field of size 50 with a maximum length (max length) of 100.

```
<adk:text name='eventName' maxlength='100' size='50' />
```

Displaying a Submit Button on the Form

Line 9 in [Listing 9-8](#) displays a button on the form that allows an adapter user to submit input.

```
<adk:submit name='addevent_submit' doAction='addevent' />
```

The label on the button is retrieved from the message bundle using the `addevent_submit` key. When the form data is submitted, the `ControllerServlet` locates the `addevent()` method on the registered request handler (see the `RequestHandlerClass` property) and passes the request data to it.

Adding Additional Fields

You must also add any additional fields that the user requires for defining an event. See [Appendix E, “Learning to Develop Adapters Using the DBMS Sample Adapters,”](#) for examples of forms with multiple fields.

Step 5c: Create the addservc.jsp form

This form allows a user to add a new service to an application view. The form is EIS-specific. The `addservc.jsp` form for the sample adapter is shown in [Listing 9-9](#).

Listing 9-9 Coding addservc.jsp

```

1 <%@ taglib uri='/WEB-INF/taglibs/adk.tld' prefix='adk' %>
2 <form method='POST' action='controller'>
3   <table>
4     <tr>
5       <td><adk:label name='serviceName' required='true' /></td>
6       <td><adk:text name='serviceName' maxlength='100' size='50' /></td>
7     </tr>
8     <tr>
9       <td colspan='2'><adk:submit name='addservc_submit'
10        doAction='addservc' /></td>
11     </tr>
12 </table>
</form>

```

Including the ADK Tag Library

Line 1 in [Listing 9-9](#) instructs the JSP engine to include the ADK tag library.

```
<%@ taglib uri='/WEB-INF/taglibs/adk.tld' prefix='adk' %>
```

The tag library supports the user-friendly form validation provided by the ADK. The ADK tag library provides the tags described in [Table 9-3](#).

Posting the ControllerServlet

Line 2 in [Listing 9-9](#) instructs the form to post to the `ControllerServlet`.

```
<form method='POST' action='controller'>
```

The `ControllerServlet` is configured in the `web.xml` file for the Web application. It is responsible for delegating HTTP requests to a method on a `RequestHandler`. You are not required to provide any code to use the `ControllerServlet`; however, you must supply the initial parameters, as described in [Table 9-5](#), “`ControllerServlet` Parameters.”

Displaying the Label for the Form Field

Line 5 in [Listing 9-9](#) displays a label for a field.

```
<adk:label name='serviceName' required='true' />
```

The value that is displayed is retrieved from the message bundle for the user. The `required` attribute indicates whether the user must supply this parameter to be successful.

Displaying the Text Field Size

Line 6 in [Listing 9-9](#) sets a text field of size 50 with a maximum length (`max length`) of 100.

```
<adk:text name='serviceName' maxlength='100' size='50' />
```

Displaying a Submit Button on the Form

Line 9 in [Listing 9-9](#) displays, on a form, a button that allows an adapter user to submit input.

```
<adk:submit name='addservc_submit' doAction='addservc' />
```

The label on the button is retrieved from the message bundle using the `addservc_submit` key. When the form data is submitted, the `ControllerServlet` locates the `addservc` method on the registered `RequestHandler` (see the `RequestHandlerClass` property) and passes the request data to it.

Adding Additional Fields

You must also add any additional fields that the user requires for defining a service. See [Appendix E](#), “[Learning to Develop Adapters Using the DBMS Sample Adapters](#),” for examples of forms with multiple fields.

Step 5d: Implement Editing Capability for Events and Services (optional)

If you want to give adapter users the ability to edit events and services at design time, you must edit the adapter properties, create `edtservc.jsp` and `edtevent.jsp` forms, and implement some specific methods. This step describes those tasks.

Note: This step is optional. You are not required to provide users with these capabilities.

Update the Adapter Properties File

First, update the system properties in the adapter properties file for the sample adapter by making the following changes to that file:

- Add the following properties:

```
- edtservc_title=Edit Service
- edtservc_description=On this page, you edit service properties.
- edtevent_description=On this page, you edit event
  properties.edtevent_title=Edit Event
- glossary_description=This page provides definitions for commonly
  used terms.
- service_submit_add=Add
- service_label_serviceDesc=Description:
- service_submit_edit=Edit
- service_label_serviceName=Unique Service Name:
- event_submit_add=Add
- event_label_eventDesc=Description:
- event_label_eventName=Unique Event Name:
- event_submit_edit=Edit
- eventLst_label_edit=Edit
- serviceLst_label_edit=Edit
- event_does_not_exist=Event {0} does not exist in application view
  {1}.
- service_does_not_exist=Service {0} does not exist in Application
  View {1}.
```

- `no_write_access={0}` does not have write access to the Application View.

- Remove the following properties:

- `addservc_submit_add=Add`
- `addevent_label_eventDesc=Description:`
- `addservc_label_serviceName=Unique Service Name:`
- `addevent_submit_add=Add`
- `pingTable_invalid=The ping table cannot be reached. Please enter a valid table in the existing database to ping.`
- `pingTable=Ping Table`
- `addevent_label_eventName=Unique Event Name:`
- `addservc_label_serviceDesc=Description:`

After updating the adapter properties file, compare your new version of the file to the original file and make sure that they are now synchronized.

Create `edtservc.jsp` and `addservc.jsp`

These Java server pages are called in order to provide editing capabilities. The main difference between the edit JSP file and the add JSP file is the loading of descriptor values; the edit JSP file loads the existing descriptor values. For this reason, the same HTML files are used for both editing and adding in the DBMS sample adapter.

These HTML files are statically included in each JSP page. This saves duplication of JSP/HTML and properties. The descriptor values are mapped to the controls displayed on the edit page. From there, you can submit any changes.

To initialize the controls with values defined in the descriptor, call the `loadEvent/ServiceDescriptorProperties()` method on the `AbstractDesignTimeRequestHandler`. This method sets all the service's properties in the `RequestHandler`. Once these values are set, the `RequestHandler` maps the values to the ADK controls being used in the JSP file.

The default implementation of `loadEvent/ServiceDescriptorProperties()` uses the property name associated with the ADK tag to map the descriptor values. If you use values other than the ADK tag names to map the properties for a service or event, you must override these values to provide the descriptor to the ADK tag-name mapping.

You must also initialize the `RequestHandler` before the HTML is resolved. This initialization should be performed only once. [Listing 9-10](#) shows an example of code used to load the `edtevent.jsp`.

Listing 9-10 Sample Code Used to Load `edtevent.jsp`

```
if(request.getParameter("eventName") != null){  
handler.loadEventDescriptorProperties(request);  
}
```

The `edtservc.jsp` file should submit to `edtservc`:

```
<adk:submit name='edtservc_submit' doAction='edtservc' />
```

The `edtevent.jsp` file should submit to `edtevent`:

```
<adk:submit name='edtevent_submit' doAction='edtevent' />
```

For examples, see the DBMS sample adapter at the following location:

```
WLI_HOME/adapters/dbms/src/war
```

Implement Methods

Finally, implement the methods described in [Table 9-6](#).

Table 9-6 Methods to Implement with `edtservc.jsp` and `edtevent.jsp`

Methods	Description
<code>loadServiceDescriptorProperties</code> and <code>loadEventDescriptorProperties</code>	These methods load the <code>RequestHandler</code> with the ADK tag-to-value mapping. If the developer uses the same values to name the ADK tag and load the Service/Event Descriptor, then the mapping is free. Otherwise, to provide these mappings, the developer must override these methods in <code>DesignTimeRequestHandlers</code> .

Table 9-6 Methods to Implement with `edtservc.jsp` and `edtevent.jsp` (Continued)

Methods	Description
<code>boolean supportsEditableServices()</code> and <code>boolean supportsEditableEvents()</code>	These methods are used as markers. If they return <code>true</code> , the edit link is displayed on the Application View Administration page. Override in the <code>DesignTimeRequestHandler</code> is supported.
<code>editServiceDescriptor</code> and <code>editEventDescriptor</code>	These methods are used to persist the edited service or event data. These methods extract the ADK tag values from the request and add them back into the Service or Event Descriptor. In addition, these methods handle any special processing for the schemas associated with the event or service. If the schemas need modification, they should be updated here. Once the values read in from the request are no longer needed, they should be removed from the <code>RequestHandler</code> .

For an example of how these methods are implemented, see the sample adapters.

Step 5e: Write the Web Application Deployment Descriptors

The `web.xml` and `weblogic.xml` descriptors for your adapter generally follow a very simple pattern, and list the names of all the JSP pages in your design-time web application, plus some other setup information for the design-time web application. Because most adapters contain very similar web descriptors, the ADK provides a means to automatically generate them. This frees the adapter developer from maintaining a large descriptor that is mostly identical to other adapters web descriptors.

The generation of the web application descriptors may be requested by including and calling a special Ant target in your Ant `build.xml` file for your adapter. If you clone the ADK sample adapter using `GenerateAdapterTemplate`, the resulting `build.xml` will already include the necessary Ant target, and a call to use that target. Look at `WLI_HOME/adapters/sample/project/build.xml` and find the `generate_web_descriptors` target. This Ant target takes in a file called `web-gen.properties`, and generates the `web.xml` and `weblogic.xml` descriptors from the information contained in it. Notice in the sample adapter `build.xml`, this target is called near the top of the `packages` target.

The sample adapter includes a `web-gen.properties` file that acts as a template for you to fill in for your adapter. Following is a list of properties in this file, and their meanings:

- `display-name` - This is the value to be used in the `display-name` element of `web.xml`. It should be the adapter logical name for your adapter (e.g. `BEA_WLS_SAMPLE_ADK` for the sample adapter).
- `version` - The version of your design-time web application, and the value that will be used in the `version` element of `web.xml` (e.g. `8.1.0` for the sample adapter, but you should use the same value as you used in `GenerateAdapterTemplate`).
- `request-handler-class` - This is the full class name of the design-time request handler implementation class for your adapter. Its the class that extends `AbstractDesignTimeRequestHandler` and normally resides in `<your adapter package>.web.DesignTimeRequestHandler`
- `adapter-logical-name` - The adapter logical name for your adapter. It should be the same value you used in `GenerateAdapterTemplate` (e.g. `BEA_WLS_SAMPLE_ADK` for the sample adapter)
- `debug-setting` - Can be the value 'on' or 'off' (without the quotes). Turning debug on will enable any debug statements you have placed in your source code (or that were there from cloning the sample adapter source) that use the `ILogger.debug()` method. Using a value of off disables any such statements from being logged to the log file.
- `extra-jsp-list` - a comma separated list of 'extra' JSPs. The 'standard' ones are:
`addevent, addservc, confconn, edtevent, edtservc, event, service, testform, var set`
You do not need to include these in your list. For example, if you add a JSP called `mybrowser.jsp`, your `extra-jsp-list` would be as follows:
`extra-jsp-list=mybrowser`

Step 6. Implement the Look and Feel

An important programming practice you should observe when developing a design-time GUI is to implement a consistent look and feel in all the pages of your application view. The look-and-feel is determined by `display.jsp`. This page, included with the ADK, provides the following benefits for a design-time Web application:

- Creates a template that establishes the look and feel for all pages.
- Includes other JSPs based on the `content HTTP` request parameter. If the `content HTTP` request parameter is not supplied, the `display.jsp` file must include `main.jsp`.
- Registers the error page for Java exceptions as `error.jsp` from the ADK.

To implement a consistent look and feel across a set of pages, do the following:

1. Use `display.jsp` from the sample adapter as a starting point. For example, see `WLI_HOME/adapters/sample/src/war/WEB-INF/web.xml`.
2. Using HTML, alter the look-and-feel markup on this page to reflect your own look and feel or your company's identity standards.
3. Somewhere in your HTML markup, be sure to include:

```
<%pageContext.include(sbPage.toString());%>
```

This code is a custom JSP tag used to include other pages. This tag uses the JSP scriptlet `sbPage.toString()` to include an HTML or JSP page in the display page. `sbPage.toString()` evaluates to the value of `content` (the HTTP request parameter) at run time.

Step 7. Implement Environment Variables

This is an optional step. Environment variables are intended to isolate environment-specific information within an application view in a way that allows system administrators and WebLogic Integration application deployers to specify new values for this information depending on the environment they are deploying to. If your adapter does not include any event or service properties that can be considered environment-specific, then you do not need to implement environment variables in your adapter. Examples of environment-specific information in event or service properties for the DBMS sample adapter are:

- Name of the catalog/schema holding the table for a given event type. Different database instances will use different catalog/schema names for the same tables, and thus this information might need to change when moving between database instances.
- Qualified names of tables used in a SQL statement for a service. If the SQL statement refers to tables in more than one catalog/schema, qualified names are required, and these names can change between database instances.

Event and service connection information should not be considered when deciding whether or not to implement environment variables. Connection information is, by its very nature, specific to the EIS instance being connected to. Environment variables are used only to isolate environment specific information in event and service definitions.

If you choose to implement environment variable support in your adapter, you must provide this support in both the design-time GUI and the runtime portion of the adapter. For more information on runtime considerations, see [“Developing a Service Adapter” on page 6-1](#).

For the design-time GUI, environment variable support can be added in two major steps:

- Include the `varset.jsp` from your `addevent/edtevent/addservc/edtservc` JSP pages. This extra JSP provides an HTML table used to display and edit a set of environment variables. This set of variables will be stored along with the application view that is being defined.
- Provide fields and field parsers/validators that accept variable markers referring to the variables shown in the `varset.jsp` table.

Step 7a – Displaying/Editing the Variable Set

The preferred method for displaying and editing the variable set for an application view is to include the `varset.jsp` JSP page in your event and service definition JSP pages. You can do this by including the following JSP code:

```
<p>
<hr>
<p>

<jsp:include page='varset.jsp'>
  <jsp:param name='mode' value='writeable' />
  <jsp:param name='hostPage' value='addservc' />
</jsp:include>
```

This will be shown as a horizontal line with a table below it for displaying and editing the environment variables for an application view. The `varset.jsp` table allows the user to add, remove, and edit variable definitions. See the DBMS sample adapter for examples of using the `varset.jsp` JSP. The `mode` parameter should always be writable, and the `hostPage` parameter should be the name of the JSP hosting the variable set JSP (without the `.jsp` extension).

Note: It is possible for you to define your own variables without using the `varset.jsp` JSP page. You can control the variables defined for an application view by using the `IMutableVariableSet` interface defined in the `com.bea.connector` package. The following code shows how to get the application view descriptor, and define a new variable called `myVariable` on it.

```
import com.bea.wlai.common.IApplicationViewDescriptor;
import com.bea.connector.IMutableVariableSet;

IApplicationViewDescriptor avd = getApplicationViewDescriptor();
IMutableVariableSet varSet = avd.getMutableVariableSet();
```

```
// Add a variable
IVariable myVar = varSet.addVariable("myVariable", "string", "My Variable",
    "the default value");

// And then remove it
varSet.removeVariable("myVariable");
```

Step 7b – Using the Variable Set

Once you have included the `varset.jsp` JSP page, you can begin using the environment variables it will allow you to define. How you use environment variables depends on the nature of your adapter and the environment-specific information it maintains.

In general, environment variables will be used to provide either a part of or the entire value for an event or service property. The value provided by the variable at runtime can be a default value or a specific value configured for the variable by a system administrator or application deployer. The decision of whether to use variables for part of a property's value, or the entire value is up to you.

For the DBMS sample adapter, variables are used in some cases as the entire property value (for example, catalog/schema name for events) or as a part of a property value (for example, catalog/schema name within a SQL statement for services). The DBMS sample adapter denotes the use of an environment variable by including the variable's name enclosed in curly braces: `{var_name}`. It uses this syntax for both events and services. The curly braces were chosen to delimit the variable name because curly braces are not reserved characters and are not legal characters in SQL syntax. This allows the DBMS sample adapter to reliably parse the curly braces within a SQL statement without risk of confusing the parser.

Your adapter may require different conventions to uniquely or safely identify variable references in your property values.

Once you have decided how to represent variable references in your event and service property values, you can write the code needed to validate variable use in those values. The following rules apply:

- All variable references in your event/service properties must match with an environment variable defined within the application view (via the `varset.jsp`).
- You must verify that all variable references in event/service properties conform to the type/usage expectations for those variables.

At runtime, the application integration engine will provide you with a set of variables and their values. Your runtime adapter code should substitute the variable's runtime value for any variable reference in the configured property value. For more information, see [“Developing a Service Adapter” on page 6-1](#) or [“Developing an Event Adapter” on page 7-1](#).

To get a list of variables defined (at design-time) via the `varset.jsp`, obtain the `IApplicationViewDescriptor` being configured by calling the superclass (`AbstractDesignTimeRequestHandler`) method:

```
import com.bea.connector.IVariableSet;

IApplicationViewDescriptor avd = getApplicationViewDescriptor();
IVariableSet varSet = avd.getVariableSet();

// List variable names

String[] varNames = varSet.listVariableNames();

// Get the first variable

IVariable var = varSet.getVariable(varNames[0]);
```

Step 8. Test the Sample Adapter Design-Time Interface

WebLogic Integration provides a test driver that verifies the basic functionality of the sample adapter design-time interface. The test driver is based on HTTP Unit, a framework for testing web interfaces which is available from <http://www.httpunit.org>. HTTP Unit is related to the JUnit test framework (available from <http://www.junit.org>). Versions of both HTTP Unit and JUnit are also included with WebLogic Integration.

The test driver executes a number of tests. It creates application views, adds both events and services to application views, deploys and undeploys application views, and tests both events and services. After it finishes running successfully, the test driver removes all application views.

Files and Classes

All test cases are available in the `DesignTimeTestCase` class or its parent class, `AdapterDesignTimeTestCase`. The `DesignTimeTestCase` class (in the `sample.web` package and the `WLI_HOME/adapters/sample/src/sample/web` folder) contains tests specific to the sample adapter. `AdapterDesignTimeTestCase` (in the `com.bea.adapter.web` package and

the `WLI_HOME/lib/adk-web.jar` file) contains tests that apply to all adapters and several convenience methods.

Run the Tests

To test the design-time interface, complete the following procedure:

1. Start WebLogic Server with the sample adapter deployed. Next, change the current working folder to the specific project folder and execute the `setenv` command, as shown in the following steps.

2. Go to `WLI_HOME` and, at the command prompt, enter `setenv`.

The `setenv` command creates the necessary environment for the next step.

3. Go to the web folder for the sample adapter by entering the following at the command prompt:

```
cd WLI_HOME/adapters/sample/project
```

4. Edit the `designTimeTestCase.properties` file: in the line containing the list of test cases to be executed, add `web.DesignTimeTestCase`. The line should read:

```
test.case=web.DesignTimeTestCase
```

5. Near the end of the file, you might need to change the value of two entries: username and password. Specify the username and password that the test driver should use to connect to WebLogic Integration.
6. After editing the `test.properties` file, start WebLogic Server.
7. Run the tests by entering the following command at the command prompt:

```
ant designtimetest
```

Developing a Design-Time GUI

Deploying Adapters

After you create an adapter, you must deploy it by using an Enterprise Archive (EAR) file. An EAR file simplifies this task by deploying all adapter components in a single step. You can deploy an EAR file from the WebLogic Server Administration Console.

This section contains information about the following subjects:

- [Using Enterprise Archive \(EAR\) Files](#)
- [Deploying Adapters Using the WebLogic Server Administration Console](#)
- [Editing Web Application Deployment Descriptors](#)

Using Enterprise Archive (EAR) Files

Each adapter is deployed from a single Enterprise Archive (EAR) file. An EAR file contains a design-time Web application WAR file, an adapter RAR file, an adapter JAR file, and any shared JAR files required for deployment. The EAR file should be structured as shown in [Listing 10-1](#).

Listing 10-1 EAR File Structure

```
adapter.ear
  application.xml
  sharedJar.jar
  adapter.jar
  adapter.rar
  META-INF
```

```
        ra.xml
        weblogic-ra.xml
        MANIFEST.MF
designertime.war
    WEB-INF
        web.xml
    META-INF
        MANIFEST.MF
```

The EAR file for the sample adapter is shown in [Listing 10-2](#).

Listing 10-2 EAR File for the Sample Adapter

```
sample.ear
  application.xml
    adk.jar (shared .jar between .war and .rar)
    bea.jar (shared .jar between .war and .rar)
    BEA_WLS_SAMPLE_ADK.jar (shared .jar between .war and .rar)
    BEA_WLS_SAMPLE_ADK.war (Web application with
      META-INF/MANIFEST.MF entry Class-Path:
      BEA_WLS_SAMPLE_ADK.jar adk.jar bea.jar log4j.jar
      logtoolkit.jar xcci.jar xmltoolkit.jar)
    BEA_WLS_SAMPLE_ADK.rar (Resource Adapter with
      META-INF/MANIFEST.MF entry Class-Path:
      BEA_WLS_SAMPLE_ADK.jar adk.jar bea.jar log4j.jar
      logtoolkit.jar xcci.jar xmltoolkit.jar)
    log4j.jar (shared .jar between .war and .rar)
    logtoolkit.jar (shared .jar between .war and .rar)
    xcci.jar (shared .jar between .war and .rar)
    xmltoolkit.jar (shared .jar between .war and .rar)
```

Notice that neither the RAR nor WAR file includes the shared JAR files; them; instead, both types of files refer to the shared JAR files using the `<manifest.classpath>` attribute.

Using Shared JAR Files in an EAR File

The design-time application uses an adapter's SPI classes in an unmanaged scenario. Consequently, an adapter's SPI and CCI classes should be contained in a shared JAR file that resides in the same directory as the EAR file. To allow the WAR and RAR classloaders to access the classes in the shared JAR, you must specify, in the `MANIFEST.MF` files, a request for inclusion of the shared EAR files. For more information about `MANIFEST.FM`, see either “Manifest File” on page 6-11 or “Understanding the Manifest” at the following URL:

<http://java.sun.com/docs/books/tutorial/jar/basics/manifest.html>

The `BEA_WLS_SAMPLE_ADK.rar` and `BEA_WLS_SAMPLE_ADK.war` files contain `META-INF/MANIFEST.MF`, as shown in Listing 10-3:

Listing 10-3 Manifest File Example

```
Manifest-Version: 1.0
Created-By: BEA Systems, Inc.
Class-Path: BEA_WLS_SAMPLE_ADK.jar adk.jar wlai-core.jar
            wlai-client.jar
```

Note: The name of the file, `MANIFEST.MF`, is spelled in all uppercase. If it is not spelled correctly, it is not recognized on a UNIX system and an error occurs.

EAR File Deployment Descriptor

Listing 10-4 shows the deployment descriptor, which declares the components of an EAR file. In this case, these components include the design-time WAR and adapter RAR modules.

Listing 10-4 Deployment Descriptor for the EAR File

```
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE
Application 1.3//EN"
'http://java.sun.com/dtd/application_1_3.dtd'>
<application>
  <display-name>BEA_WLS_SAMPLE_ADK</display-name>
```

```
<description>This is a J2EE application that contains a sample
  connector and Web application for configuring
  application views for the adapter.</description>
<module>
  <connector>BEA_WLS_SAMPLE_ADK.rar</connector>
</module>
<module>
  <web>
    <web-uri>BEA_WLS_SAMPLE_ADK.war</web-uri>
    <context-root>BEA_WLS_SAMPLE_ADK_Web</context-root>
  </web>
</module>
</application>
```

You can deploy the adapter via the WebLogic Server Administration Console. This procedure is described in [“Deploying Adapters Using the WebLogic Server Administration Console”](#).

Deploying Adapters Using the WebLogic Server Administration Console

To configure and deploy an adapter from the WebLogic Server Administration Console, complete the following procedure:

1. Open the WebLogic Server Administration Console.
2. In the navigation tree (in the left pane), choose Deployments→Applications.
The Applications page is displayed.
3. Select Configure a new application.
The Configure a new Application page is displayed.
4. Enter values in the following fields:
 - In the Name field, enter the logical name of the adapter.
 - In the Path field, enter the path for the appropriate EAR file.
 - In the Deployed field, make sure that the check box is selected.

5. Click Apply to create the new entry.
6. Select Configure Components.
7. Set the target for each component individually.

When you install an application (or application component) via the WebLogic Server Administration Console, you also create entries for that application or component in the configuration file for the relevant domain (`/config/DOMAIN_NAME/config.xml`, where *DOMAIN_NAME* is your domain). WebLogic Server also generates JMX Management Beans (MBeans) that enable you to configure and monitor the application and application components.

Adapter Auto-registration

WebLogic Integration uses an automatic registration process during adapter deployment. Autoregistration is performed during the adapter deployment phase. You can invoke this process in either of two ways:

- [Using a Naming Convention](#)
- [Using a Text File](#)

Using a Naming Convention

The preferred approach is to use a naming convention for the design-time Web application and connector deployment.

When deploying an EAR file in a WebLogic Integration environment, identify the file in `config.xml` by using the logical name of the adapter as the filename, as shown in the example, in [Listing 10-5](#).

Listing 10-5 Adding the Adapter Logical Name to config.xml

```
<Application Deployed="true" Name="ALN"
  Path="WLI_HOME/adapters/ADAPTER/lib/ALN.ear">
  <ConnectorComponent Name="ALN" Targets="myserver"
    URI="ALN.rar"/>
  <WebAppComponent Name="ALN_EventRouter" Targets="myserver"
    URI="ALN_EventRouter.war"/>
  <WebAppComponent Name="ALN_Web" Targets="myserver"
```

```
        URI="ALN_Web.war" />  
</Application>
```

In this listing, *ALN* is the logical name of the adapter. You must use this name as the value of the `Name` attribute of the `<ConnectorComponent>` element.

If you assign the name *ALN_Web* to your design-time Web application deployment, the design-time Web application is registered automatically, through the Weblogic Server Administration Console, during deployment. This naming convention is used in the DBMS and sample adapters.

Using a Text File

Alternatively, you can include a text file named `webcontext.txt` in the root directory of the pathname for your EAR file. The `webcontext.txt` file should contain the context for the design-time Web application for your adapter. This file must be encoded in UTF-8 format.

Editing Web Application Deployment Descriptors

For some adapters, you may need to change the deployment parameters of the Event Router Web application. For the DBMS sample adapter, for example, you might need to change the database URL used by its event generator.

This section explains how to use the Deployment Descriptor Editor provided by the WebLogic Server Administration Console to edit the following Web application deployment descriptors:

- `web.xml`
- `weblogic.xml`

Deployment Parameters

You can change any parameter of the Event Router Servlet. These parameters are:

- `eventGeneratorClassName`
- `userID`
- `password`
- `dataSource`
- `jdbcDriverClassName`

- dbURL
- dbAccessFlag
- eventCatalog
- eventSchema
- RootLogContext
- AdditionalLogContext
- LogConfigFile
- LogLevel
- MessageBundleBase
- LanguageCode
- CountryCode
- sleepCount

Editing the Deployment Descriptors

To edit the Web application deployment descriptors, complete the following procedure:

1. Open the WebLogic Server Administration Console in your browser by accessing the following URL:

```
http://host:port/console
```

In this URL, replace *host* with the name of the computer on which WebLogic Server is running, and *port*, with the number of the port on which WebLogic Server is listening. For example:

```
http://localhost:7001/console
```

2. In the left pane, expand two nodes: the Deployments node and the Web Applications node below it.
3. Right-click the name of the Web application for which you want to edit the deployment descriptors. From the drop-down menu select Edit Web Application Descriptor. The WebLogic Server Administration Console is displayed in a new browser.

The Console consists of two panes. The left pane contains a navigation tree composed of all the elements in the two Web application deployment descriptors. The right pane contains a form for the descriptive elements of the `web.xml` file.

4. To edit, delete, or add elements in the Web application deployment descriptors, expand the node in the left pane that corresponds to the deployment descriptor file you want to edit. The following nodes are available:
 - The WebApp Descriptor node contains the elements of the `web.xml` deployment descriptor.
 - The WebApp Ext node contains the elements of the `weblogic.xml` deployment descriptor.
5. To edit an existing element in one of the Web application deployment descriptors, complete the following procedure:
 - a. Navigate the tree in the left pane, clicking parent elements until you find the element you want to edit.
 - b. Click the name of the appropriate element. A form is displayed in the right pane with a list of either the attributes or the subelements of the selected element.
 - c. Edit the text in the form in the right pane.
 - d. Click Apply.
6. To add a new element to one of the Web application deployment descriptors, complete the following procedure:
 - a. Navigate the tree in the left pane, clicking parent elements until you find the name of the element you want to create.
 - b. Right-click the name of the appropriate element and select Configure a New Element from the drop-down menu. A form is displayed in the right pane.
 - c. Enter the element information in the form in the right pane.
 - d. Click Create.
7. To delete an existing element from one of the Web application deployment descriptors, complete the following procedure:
 - a. Navigate the tree in the left pane, clicking parent elements until you find the name of the element you want to delete.
 - b. Right-click the name of the appropriate element and select Delete Element from the drop-down menu. A confirmation page is displayed.
 - c. Click Yes on the Delete confirmation page to verify that you want to delete the element.

8. Once you have made all your changes to the Web application deployment descriptors, click the root element of the tree in the left pane. The root element is either the name of the Web application *.war archive file or the name that is displayed for the Web application.
9. Click Validate if you want to ensure that the entries in the Web application deployment descriptors are valid.
10. Click Persist to write your edits of the deployment descriptor files to disk in addition to WebLogic Server memory.

Deploying Adapters in a WebLogic Integration Cluster

Adapters can be deployed to a WebLogic Integration cluster. For more information about deploying adapters in a clustered WebLogic Integration environment, see “[Understanding WebLogic Integration Clusters](#)” in *Deploying BEA WebLogic Integration Solutions* at the following URL:

<http://edocs.bea.com/wli/docs81/deploy/cluster.html>

Redeploying Adapter Instances

If you have made changes to the event connection or service connection for an adapter instance, you must redeploy the instance for those changes to take effect. Redeploying an adapter instance causes its dependent application views to be redeployed, as well. You can replot adapter instances using the WebLogic Integration Administration Console. For more information, see “[Redeploying an Adapter Instance](#)” in *Managing WebLogic Integration Solutions* at the following URL:

<http://edocs.bea.com/wli/docs81/manage/ai.html>

Warning: The adapter instance redeploy operation does not guarantee any transaction recovery for in-flight transactions or transaction requests during undeployment and redeployment. Redeploying during transaction processing may result in unpredictable behavior and data mismatches between the database and the WebLogic Integration Administration Console.

Deploying Adapters

Creating an Adapter Not Specific to WebLogic Integration

The procedures for developing J2EE-compliant adapters outlined in [Chapter 6, “Developing a Service Adapter,”](#) and [Chapter 7, “Developing an Event Adapter,”](#) primarily pertain to adapters developed for use with WebLogic Integration. By making modifications to the procedures described in those chapters, you can build an adapter that complies with the J2EE Connector Architecture specification but is not specific to WebLogic Integration.

This section describes those modifications. Specifically, it provides information about the following subjects:

- [Using This Section](#)
- [Building the Adapter](#)
- [Updating the Build Process](#)

Using This Section

This section shows you how to modify the procedure for developing a J2EE-compliant adapter in order to build one that is not specifically designed to run with WebLogic Integration. Each step in this section refers to a corresponding step in [Chapter 6, “Developing a Service Adapter,”](#) and describes how to modify that step. You should understand each step thoroughly before proceeding with the modifications described here.

Building the Adapter

This procedure is based on the assumption that you have installed WebLogic Integration as described in *Installing BEA WebLogic Integration*.

1. Identify the requirements for your development environment as described in “[Step 1: Research Your Environment Requirements](#)” in [Chapter 6, “Developing a Service Adapter.”](#) Ignore the final item in the bullet list, which refers to transaction support; WebLogic Server does not support local or XA transactions.
2. Run `GenerateAdapterTemplate`, as described in [Chapter 4, “Creating a Custom Development Environment.”](#)
3. Assign a logical name to the adapter, as described in “[Step 2b: Assign the Adapter Logical Name](#)” on [page 6-11](#).
4. Implement the SPI, as described in “[Basic SPI Implementation](#)” on [page 6-24](#). You must extend the following classes:
 - `AbstractManagedConnectionFactory` (described in “[ManagedConnectionFactory](#)” on [page 6-24](#))
 - `AbstractManagedConnection` (described in “[ManagedConnection](#)” on [page 6-32](#))
 - `AbstractConnectionMetaData` (described in “[ManagedConnectionMetaData](#)” on [page 6-33](#))As you implement these classes, keep the following considerations in mind:
 - WebLogic Server does not support adapters that use transactional semantics.
 - Do not implement the `ConnectionManager` interface; the adapters you are developing here are managed adapters (that is, they are designed to be plugged in to WebLogic Server).
5. Extend `AbstractConnectionFactory`.

Updating the Build Process

In addition to the procedure provided in “[Building the Adapter](#)” on [page A-2](#), you need to modify the `build.xml` file to create an adapter that is not specific to WebLogic Integration. To update the build process, do the following:

1. In your code editor, open the ADK’s `build.xml` file.

2. See “[Step 2c: Set Up the Build Process](#)” on page 6-11. This step includes a section called “[build.xml Components](#)” on page 6-13. In that section, the contents of the `build.xml` file are shown in a set of code listings.
3. Find [Listing 6-11](#) and [Listing 6-12](#).
4. Remove the code shown in those listings from the adapter’s `build.xml` file.

Creating an Adapter Not Specific to WebLogic Integration

XML Toolkit

The XML Toolkit provided with BEA WebLogic Integration's Adapter Development Kit (ADK) helps you develop valid XML documents to transmit information from an EIS to the application on the other side of the adapter. It consolidates, in a single location, many of the operations required for XML manipulation, relieving you of the need to perform these often tedious chores separately.

This section contains information about the following subjects:

- [Toolkit Packages](#)
- [IDocument](#)
- [Schema Object Model \(SOM\)](#)

Toolkit Packages

The XML Toolkit is composed primarily of two Java packages:

- `com.bea.document`
- `com.bea.schema`

These packages are available in the `xmltoolkit.jar` file, which is installed with the ADK when you install WebLogic Integration. They include complete Javadoc for each class, interface, and method.

IDocument

`com.bea.document.IDocument`

An `IDocument` is a container that combines the W3C Document Object Model (DOM) with an XPath interface to elements in an XML document. This combination makes it possible to query and update `IDocument` objects simply by using XPath strings. These strings eliminate the need to parse an entire XML document to find specific information by allowing you to specify only those elements you want to query, and returning responses to those queries.

For example, the XML document shown in [Listing B-1](#) describes a person named *Bob*.

Listing B-1 XML Example

```
<Person name="Bob">
  <Home squareFeet="2000" />
  <Family>
    <Child name="Jimmy">
      <Stats sex="male" hair="brown" eyes="blue" />
    </Child>
    <Child name="Susie">
      <Stats sex="female" hair="blonde" eyes="brown" />
    </Child>
  </Family>
</Person>
```

Suppose you want to retrieve Jimmy's hair color from the `<child>` element. If you use DOM, you must use the code shown in [Listing B-2](#).

Listing B-2 Sample Retrieval of DOM Data

```
String strJimmysHairColor = null;
org.w3c.dom.Element root = doc.getDocumentElement();
if (root.getTagName().equals("Person") && root.getAttribute("name").
equals("Bob")) {
    org.w3c.dom.NodeList list = root.getElementsByTagName("Family"); if
        (list.getLength() > 0) {
        org.w3c.dom.Element family = (org.w3c.dom.Element)list.item(0);
```

```

org.w3c.dom.NodeList childList = family.getElementsByTagName ("Child");
for (int i=0; i < childList.getLength(); i++) {
    org.w3c.dom.Element child = childList.item(i);
    if (child.getAttribute("name").equals("Jimmy")) {
        org.w3c.dom.NodeList statsList = child.
            getElementsByTagName("Stats");
        if (statsList.getLength() > 0) {
            org.w3c.dom.Element stats = statsList.item(0);
            strJimmysHairColor = stats.getAttribute("hair");
        }
    }
}
}
}
}

```

By using `IDocument`, however, you can retrieve Jimmy's hair color by creating the XPath string that seeks exactly that information, as shown in [Listing B-3](#).

Listing B-3 Sample Retrieval of IDocument Data

```

System.out.println("Jimmy's hair color: " + person.getStringFrom
    ("//Person[@name=\"Bob\"] /Family/Child[@name=\"Jimmy\"]/Stats/@hair");

```

As you can see, by using `IDocument` you can simplify the code necessary to query and find information in a document.

Schema Object Model (SOM)

SOM is an interface for programmatically building XML schemas. An adapter calls an EIS for specific request and response metadata, which then must be programmatically transformed into an XML schema. SOM is a set of tools that extracts and validates many of the common details XML schemas—such as syntactical complexities of schemas—so that you can focus on the more fundamental aspects of the XML document.

How SOM Works

An XML schema is similar to a contract between an EIS and an application on the other side of the adapter. This contract specifies how data coming from the EIS must be displayed in order to

be manipulated by the application. A document (that is, an XML-rendered collection of metadata from the EIS) is considered valid if it meets the rules specified in the schema, regardless of whether or not the XML code in the document is correct. For example, if a schema requires a name to be shown in a `<name>` element and that element requires two child elements, `<firstname>` and `<lastname>`, then, in order to be valid, the document from the EIS must be written in the form shown in [Listing B-4](#) and the schema must be written as shown in [Listing B-5](#).

Listing B-4 Document Example

```
<name>
  <firstname>Joe</firstname>
  <lastname>Smith</lastname>
</name>
```

Listing B-5 Schema Example

```
<schema>
  <element name="name">
    <complexType>
      <sequence>
        <element name="firstname" />
        <element name="lastname" />
      </sequence>
    </complexType>
  </element>
</schema>
```

No other form of `<name></name>` is valid, even if it is written as correct XML code. The following XML, for example, is not valid:

```
<name>Joe Smith</name>
```

Creating the Schema

You can create an XML schema programatically by using the classes and methods provided with SOM. The benefit of using this tool is that it allows you to tailor a schema for your needs simply by populating the variables in the program components. For instance, the following code examples create a schema that validates a purchase order document. [Listing B-6](#) sets up the schema and adds the necessary elements.

Listing B-6 Purchase Order Schema

```
import com.bea.schema.*;
import com.bea.schema.type.SOMType;

public class PurchaseOrder
{
    public static void main(String[] args)
    {
        System.out.println(getSchema().toString());
    }

    public static SOMSchema getSchema()
    {
        SOMSchema po_schema = new SOMSchema();

        po_schema.addDocumentation("Purchase order schema for
        Example.com.\nCopyright 2000 Example.com.\nAll rights
        reserved.");

        SOMElement purchaseOrder =
            po_schema.addElement("purchaseOrder");

        SOMElement comment = po_schema.addElement("comment");

        SOMComplexType usAddress =
            po_schema.addComplexType("USAddress");

        SOMSequence seq2 = usAddress.addSequence();

        // adding an object to a SOMSchema defaults to type="string"
        seq2.addElement("name");
        seq2.addElement("street");
        seq2.addElement("city");
    }
}
```

```
seq2.addElement("state");
seq2.addElement("zip", SOMType.DECIMAL);
```

Attributes can be set in the same way that elements are created, as shown in [Listing B-7](#). To set these attributes correctly, you must maintain their addressability.

Listing B-7 Setting Parent Attributes

```
SOMAttribute country_attr = usAddress.addAttribute("country",
    SOMType.NMTOKEN);
country_attr.setUse("fixed");
country_attr.setValue("US");
```

Like `complexType`s, `simpleTypes` can be added to the root of the schema, as shown in [Listing B-8](#).

Listing B-8 Adding SimpleTypes to the Schema Root

```
SOMSimpleType skuType = po_schema.addSimpleType("SKU");
SOMRestriction skuRestrict = skuType.addRestriction
    (SOMType.STRING);
skuRestrict.setPattern("\\d{3}-[A-Z]{2}");

SOMComplexType poType =
    po_schema.addComplexType("PurchaseOrderType");
purchaseOrder.setType(poType);
poType.addAttribute("orderDate", SOMType.DATE);
```

The `addSequence()` method of a `SOMComplexType` object returns a `SOMSequence` reference, allowing you to modify the element that was added to the schema. As shown in [Listing B-9](#), objects are added to the schema in this way.

Listing B-9 Implementing addSequence() to Modify an Element

```
SOMSequence poType_seq = poType.addSequence();
    poType_seq.addElement("shipTo", usAddress);
    poType_seq.addElement("billTo", usAddress);
```

The attributes of an element within a schema can be set by calling the setter methods of the `SOMElement` object. For example, [Listing B-10](#) shows the implementation of `setMinOccurs()` and `setMaxOccurs()`.

Listing B-10 Implementing setMinOccurs() and setMaxOccurs()

```
SOMElement commentRef = new SOMElement(comment);
    commentRef.setMinOccurs(0);
    poType_seq.add(commentRef);
SOMElement poType_items = poType_seq.addElement("items");
SOMComplexType itemType = po_schema.addComplexType("Items");
SOMSequence seq3 = itemType.addSequence();
SOMElement item = new SOMElement("item");
    item.setMinOccurs(0);
    item.setMaxOccurs(-1);
    seq3.add(item);
SOMComplexType t = new SOMComplexType();
    item.setType(t);
SOMSequence seq4 = t.addSequence();
    seq4.addElement("productName");
SOMElement quantity = seq4.addElement("quantity");
SOMSimpleType st = new SOMSimpleType();
    quantity.setType(st);
SOMRestriction restrict =
    st.addRestriction(SOMType.POSITIVEINTEGER);
    restrict.setMaxExclusive("100");
```

In this example, the `items` element for `PurchaseOrderType` was created before the `Items` type. Therefore when the `Items` type object becomes available, you must create the reference and set the type by using the code shown in [Listing B-11](#).

Listing B-11 Setting the Type When the Items Type Object Is Available

```
poType_items.setType(itemType);
```

Finally, you need to add an element to the schema. You can do so by implementing either the `addElement()` method of `SOMSequence` or the `add()` method from a previously created `SOMElement`. [Listing B-12](#) shows both methods.

Listing B-12 Adding an Element to the Schema

```
seq4.addElement("USPrice", SOMType.DECIMAL);

SOMElement commentRef2 = new SOMElement(comment);
commentRef2.setMinOccurs(0);
seq4.add(commentRef2);

SOMElement shipDate = new SOMElement("shipDate", SOMType.DATE);
shipDate.setMinOccurs(0);
seq4.add(shipDate);
t.addAttribute("partNum", skuType);

return po_schema;
}
}
```

Resulting Schema

When you run the code shown in the previous seven listings ([Listing B-6](#) through [Listing B-12](#)), the schema shown in [Listing B-13](#) is created.

Listing B-13 XML Schema Definition Document

```

<?xml version="1.0" ?>
<!DOCTYPE schema (View Source for full doctype...)>
<xsd:schema xmlns:xsd="http://www.w3.org/2000/XMLSchema">
  <xsd:annotation>
    <xsd:documentation>Purchase order schema for Example.com.
      Copyright 2000 Example.com. All rights
      reserved.</xsd:documentation>
  </xsd:annotation>
  <xsd:simpleType name="SKU">
    <xsd:annotation>
    </xsd:annotation>
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d{3}-[A-Z]{2}" />
    </xsd:restriction>
  </xsd:simpleType>
  <xsd:complexType name="PurchaseOrderType">
    <xsd:sequence>
      <xsd:element type="USAddress" name="shipTo" />
      <xsd:element type="USAddress" name="billTo" />
      <xsd:element ref="comment" minOccurs="0" />
      <xsd:element type="Items" name="items" />
    </xsd:sequence>
    <xsd:attribute name="orderDate" type="xsd:date" />
  </xsd:complexType>
  <xsd:complexType name="Items">
    <xsd:sequence>
      <xsd:element maxOccurs="unbounded" name="item"
        minOccurs="0">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element type="xsd:string"
              name="productName" />
            <xsd:element name="quantity">

```

```

        <xsd:simpleType>
            <xsd:restriction base=
                "xsd:positiveInteger">
                <xsd:maxExclusive value="100" />
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:element>
    <xsd:element type="xsd:decimal" name=
        "USPrice" />
    <xsd:element ref="comment"
        minOccurs="0" />
    <xsd:element type="xsd:date"
        name="shipDate" minOccurs="0" />
    </xsd:sequence>
    <xsd:attribute name="partNum" type="SKU" />
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
<xsd:complexType name="USAddress">
    <xsd:sequence>
        <xsd:element type="xsd:string" name="name" />
        <xsd:element type="xsd:string" name="street" />
        <xsd:element type="xsd:string" name="city" />
        <xsd:element type="xsd:string" name="state" />
        <xsd:element type="xsd:number" name="zip" />
    </xsd:sequence>
    <xsd:attribute name="country" use="fixed" value="US"
        type="xsd:NMTOKEN" />
</xsd:complexType>
<xsd:element type="PurchaseOrderType" name="purchaseOrder" />
<xsd:element type="xsd:string" name="comment" />
</xsd:schema>

```

Validating an XML Document

The schema shown in [Listing B-13](#) is then used to validate a document sent from the EIS. For example, the document described in [Listing B-14](#) passes schema validation based on the schema we just created.

Listing B-14 Validated XML Document

```
<?xml version="1.0" ?>
<!DOCTYPE PurchaseOrder (View Source for full doctype...)>

<purchaseOrder orderDate="1/14/00">
  <shipTo Country="US">
    <name>Bob Jones</name>
    <street>1000 S. 1st Street</street>
    <city>Denver</city>
    <state>CO</state>
    <zip>80111</zip>
  </shipTo>

  <billTo Country="US">
    <name>Bob Jones</name>
    <street>1000 S. 1st Street</street>
    <city>Denver</city>
    <state>CO</state>
    <zip>80111</zip>
  </billTo>

  <comment>None</comment>

  <items>
    <item partNum="123-AA">
      <productName>Washer</productName>
      <quantity>20</quantity>
      <USPrice>0.22</USPrice>
      <comment>Only shipped 10</comment>
      <shipDate>1/14/00</shipDate>
    </item>
    <item partNum="123-BB">
      <productName>Screw</productName>
```

```
<quantity>10</quantity>
<USPrice>0.30</USPrice>
<comment>None</comment>
<shipDate>1/14/00</shipDate>
</item>
</items>
</purchaseOrder>
```

How the Document Is Validated

SOM can be used to validate XML DOM documents by using the `SOMSchema` method `isValid()`. The `SOMElement` class includes a corresponding `isValid()` method for validating an element instead of a DOM document.

The `isValid()` method determines whether a document or element is valid and, if it is not, `isValid()` compiles a list of errors. If the document is valid, `isValid()` returns `true` and the list of errors is empty.

Implementing `isValid()`

[Listing B-15](#) shows two ways to implement `isValid()`.

Listing B-15 Examples of `isValid()` Implementation

```
public boolean isValid(org.w3c.dom.Document doc,
                      java.util.List errorList)
public boolean isValid(IDocument doc,
                      List errorList)
```

The following parameters are used:

- `doc` - The document instance to be validated
- `errorList` - A list of errors found in the document, `doc`

`isValid()` returns a boolean value of `true` if the document is valid with respect to this schema. If the document is not valid with respect to the schema, `isValid()` returns `false` and the `errorList` is populated.

`errorList` is a `java.util.List` for reporting errors found in the document, `doc`. The error list is cleared before validating the document. Therefore, the list implementation used must support the `clear()` method. If `isValid()` returns `false`, the error list is populated with a list of errors found during the validation procedure. The items in the list are instances of the class `com.bea.schema.SOMValidationException`. If `isValid()` returns `true`, `errorList` is empty.

For complete information about the API, see the Javadoc for `isValid()` at the following URL:

<http://e-docs.bea.com/wli/docs81/javadoc/>

`isValid()` Sample Implementation

Listing B-16 shows a sample implementation of `isValid()`.

Listing B-16 Sample Implementation of `isValid()`

```
SOMSchema schema = ...;

IDocument doc = DocumentFactory.createDocument(new FileReader(f));
java.util.LinkedList errorList = new java.util.LinkedList();
boolean valid = schema.isValid(doc, errorList);...
if (! valid){
    System.out.println("Document was invalid. Errors were:");
    for (Iterator i = errorList.iterator; i.hasNext(); )
    {
        System.out.println(((SOMValidationException) i.next()).
            toString());
    }
}
```

Adapter Setup Worksheet

Use the worksheet on the following page to collect critical information about the adapter you are developing. The questions on the worksheet will help you define components, such as the logical name of the adapter and the basename of the Java package. They can also help you determine the locales for which you need to localize message bundles. Your answers to these questions will help you define your adapter before you start coding.

Note: If you are using the `GenerateAdapterTemplate` utility, it is especially important for you to use the worksheet; the answers you provide are essential to your ability to run this utility successfully.

Adapter Setup Worksheet

Before you begin developing an adapter, answer as many of the following questions as you can. If you plan to use the `GenerateAdapterTemplate` utility, you must answer every question marked by an asterisk (*).

1. *What is the name of the EIS for which you are developing an adapter?
2. *Which version of the EIS are you using?
3. *Which type (such as DBMS or ERP) of the EIS are you using?
4. *What is the name of the vendor for this adapter?
5. *Which version of the adapter are you using?
6. *What is the logical name of the adapter?
7. Does the adapter need to invoke functionality within the EIS?
If so, then your adapter needs to support services.
8. What mechanism and/or API is provided by the EIS to allow an external program to invoke functionality provided by the EIS?
9. What information is needed to create a session and/or connection to the EIS for this mechanism?
10. What information is needed to determine which function(s) will be invoked in the EIS for a given service?
11. Does the EIS allow you to query it for input and output requirements for a given function?
If so, what information is needed to determine the input requirements for the service?
12. Which of the input requirements are static across all requests? Your adapter should encode static information in an `InteractionSpec` object.
13. Which of the input requirements are dynamic per request? Your adapter should provide an XML schema that describes the input parameters required by this service per request.
14. What information is needed to determine the output requirements for the service?
15. Does the EIS provide a mechanism to browse a catalog of functions your adapter can invoke? If so, your adapter should support browsing of services.

16. Does the adapter need to receive notifications of changes that occur inside the EIS? If so, then your adapter needs to support events.
17. What mechanism and/or API is provided by the EIS to allow an external program to receive notification of events in the EIS? The answer to this question will help determine whether a pull mechanism or a push mechanism is developed.
18. Does the EIS provide a way to determine which events your adapter can support?
19. Does the EIS provide a way to query for metadata for a given event?
20. What locales (defined by language and country) does your adapter need to support?

Adapter Setup Worksheet

Upgrading Adapters to WebLogic Integration 8.1

For information on converting a WebLogic Integration 7.0 SP2 Adapter Development Kit (ADK) adapter development tree to an 8.1 ADK adapter development tree, see “[Upgrading an Adapter Development Project](#)” in the *Upgrade Guide* at the following URL:

<http://edocs.bea.com/wli/docs81/upgrade/adk.html>

For information on upgrading adapters developed with the ADK to interact with WebLogic Integration 8.1 SP2, see “[Upgrading WebLogic Integration 7.0 SP2 Adapters Developed by ADK to WebLogic Integration 8.1 SP2](#)” in the *Upgrade Guide* at the following URL:

<http://edocs.bea.com/wli/docs81/upgrade/addtionalinfo.html>

Upgrading Adapters to WebLogic Integration 8.1

Learning to Develop Adapters Using the DBMS Sample Adapters

This section contains information about the following subjects:

- [Introduction to the DBMS Sample Adapters](#)
- [How the DBMS Sample Adapters Work](#)
- [How the DBMS Sample Adapters Were Developed](#)
- [How the DBMS Sample Adapter Design-Time GUI Was Developed](#)

Introduction to the DBMS Sample Adapters

The DBMS sample adapters are J2EE-compliant adapters that includes a JSP-based GUI. They provides concrete examples of how an adapter can be constructed by using the WebLogic Integration ADK. A relational database is used as the adapters' EIS to allow adapter providers to focus on details of the adapters and the ADK, instead of investing time to learn about a particular proprietary EIS.

Two DBMS sample adapters are provided:

- **BEA_WLS_DBMS_ADK**—A sample DBMS adapter that includes XA transaction support. This adapter is used for the tour of the sample adapter and the description of adapter development.
- **BEA_WLS_DBMS_ADK_LOCALTX**—A sample DBMS adapter that includes only support for local transactions.

The DBMS Sample Adapters are intended to help you understand the tasks required to design and develop your own adapter. They are not intended for use in a production environment, nor are they supported in such an environment. Because the adapters are intended as examples, rather than production-ready adapters, they do not include a full set of features and have the following limitations: the adapters are unable to execute complex queries or stored procedures.

Whether you are a developer or a business analyst, the DBMS sample adapters can help you understand the possibilities at your disposal when you use the ADK to build adapters. If you are a business analyst, you might find it useful to run through the interface to get a better understanding of an *application view*, *service*, and *event*, as described in [“How the DBMS Sample Adapters Work” on page E-3](#).

If you are an adapter developer, we suggest you start by learning how you can extend and use the ADK classes to build a J2EE-compliant adapter. To do so, review the following:

- [“How the DBMS Sample Adapters Were Developed” on page E-26](#)
- [“How the DBMS Sample Adapter Design-Time GUI Was Developed” on page E-43](#)
- DBMS sample adapter code
- DBMS sample adapter Javadoc

The DBMS sample adapters satisfy the following requirements:

- Provides a GUI that allows end-users to connect to a Pointbase, or Oracle database.
- Uses the classes and tools of the ADK.
- Allows users to create application views with events and services.
- Allows users to test events and services.
- Provides a GUI that enables users to browse, from the GUI, the catalogs, schemas, tables, and columns of the underlying database.
- Supports the creation of services that select, insert, delete, and update data in the database (EIS).

Connection Parameters for DBMS Sample Adapters

When setting connection parameters for these adapters, specify a JDBC Driver and URL. The sample DBMS adapters do not support the specification of a DataSource. Existing service connections which use a DataSource operate as before, but the Application Integration Design Console does not allow you to specify a DataSource for a new adapter.

If you configure a domain using an XA driver, create an additional connection pool for whatever DataSource you create for the event connection.

Do not create two Tx Data Sources that point to the same connection pool. If a transaction uses two different Tx Data Sources which are both pointed to the same connection pool, you will get an XA_PROTO error when you try to access the second connection.

For more information on JDBC DataSources and WebLogic Server, see “JDBC DataSources” in *Administration Console Online Help* at the following location:

http://e-docs.bea.com/wls/docs81/ConsoleHelp/jdbc_datasources.html

When using the DBMS sample adapter with a Microsoft SQL Server or Sybase database, use the WebLogic Server Administration Console to enable the `TestReservedConnection` parameter for the connection pool used for application integration. If the parameter is not enabled, the auto resume or manual resume features do not work and a `SQLException` similar to the following is thrown:

```
java.sql.SQLException: [BEA][SQLServer JDBC Driver]No more data
available to read
```

How the DBMS Sample Adapters Work

This section shows how the DBMS sample adapter works. If you are a business analyst, you might enjoy running through the interface to get a feel for how the adapter works. The example in this section shows how to create a service that inserts a customer in the underlying database, and how an event is generated to notify others that this action has taken place.

This section contains information about the following subjects:

- [Before You Begin](#)
- [Accessing the DBMS Sample Adapter](#)
- [Tour of the DBMS Sample Adapter](#)

Before You Begin

Before you try to access the DBMS sample adapter, make sure you complete the following tasks:

- Install WebLogic Integration. For instructions, see [Installing WebLogic Platform](#)
- Set up the ADK Ant-based make process. For instructions, see “[Step 2c: Set Up the Build Process](#)” on page 6-11.

- Deploy the DBMS sample adapter in such a way that the design-time GUI is accessible. For more information, see [Installing WebLogic Platform](#).
- Start WebLogic Server.
- Assign the system administrator role to Microsoft SQL Server user accounts. To deploy an application integration event for a Microsoft SQL Server database, the user account from which event generator tables are created must be assigned the system administrator role. If the user account is not assigned the system administrator role, the deployment fails and an error message is generated stating that the tables are invalid objects.

Accessing the DBMS Sample Adapter

To access the DBMS sample adapter:

1. Open a new browser window.
2. Enter the URL for your system's Application Integration Design Console:

```
http://HOSTNAME:7001/wlai
```

The Application Integration Design Console Logon page is displayed as shown in [Figure E-1](#).

Warning: You should only have one instance of the Application Integration Design Console running on a single client machine. Running multiple consoles on a single machine may interfere with proper navigation between screens in your web browser.

Tour of the DBMS Sample Adapter

This section provides a short tour through the DBMS sample adapter. To begin, open the Application Integration Design Console Logon page for the DBMS sample adapter in your browser. For instructions, see [“Accessing the DBMS Sample Adapter” on page E-4](#).

Figure E-1 Application Integration Design Console - Logon

WebLogic Integration - Application Integration Design Console Logon

Please supply a valid WebLogic username and password.

Username

Password

1. To log on to the Application Integration Design Console, enter your WebLogic Server username and password, and click Login. The Select Application page is displayed.

Select Application

Current Application: --- Application Root Folder: ---

Application Integration Server Configuration Switch Application WebLogic Server Console WebLogic Integration Console

This page allows you to define the application context for this design-time session. This context includes the name of the J2EE application that will host the ApplicationViews shown within this console, and the root folder of the J2EE applications BEA Platform application or EAR folder.

You may select a deployed application from the list below...

Deployed Applications:

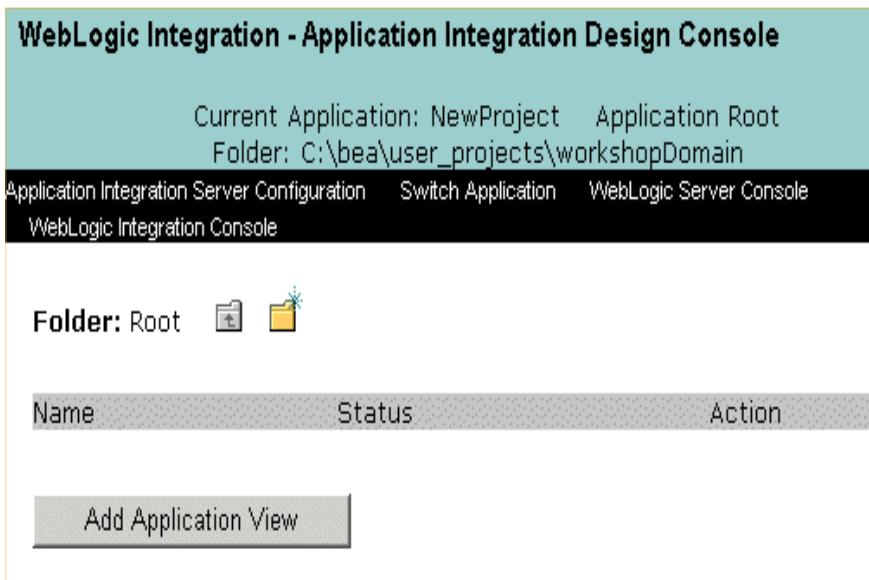
Or manually type in the name and root folder of the application.

Application Name (BEA Application or J2EE App):

Root Folder for Application (absolute):

2. To create a repository for a new application, specify an application name and the root directory for the application. Click OK. The Application View Design Console page is displayed.

Figure E-2 Application View Management Console



3. Click Add Application View. The Define New Application View page is displayed. When you create an application view, you should provide a description that associates that application view with the DBMS sample adapter.

For detailed information about application views and how to define them, see [“Defining an Application View”](#) in *Using the Application Integration Design Console*.

Figure E-3 Define New Application View Page

This page allows you to define a new application view

Folder: [Root](#)

Application View Name: *

Description:

Associated Adapter:

4. To define an application view:
 - a. In the Application View Name field, enter `AppViewTest`.

The name should describe the set of functions performed by this application. The name of each application view must be unique to its adapter. All characters are valid except the following: period (.), hash mark (#), backslash (\), plus sign (+), ampersand (&), comma (,), apostrophe (‘), double quotes (“), and a space.
 - b. In the Description field, enter a brief description of the application view.
 - c. From the Associated Adapters list, select a DBMS sample adapter to use to create your application view.
 - d. Click **Create New Connection**. The **Create New Browsing Connection** page is displayed. Alternatively, to reuse an existing connection, click on **Reuse Existing Connection** and select a browsing connection.

Connection Name: * AppViewTest_Default_Browsing

Description:

Connection Parameters:

Define... (Needed)

Connection Pool Parameters

Use these parameters to configure the connection pool for this ConnectionFactory.

Minimum Pool Size* 0

Maximum Pool Size* 5

Allow Pool to Shrink?

Log Configuration

Set the log verbosity level for this ConnectionFactory.

Log warnings, errors, and audit messages

Ok Cancel

5. To create a new connection:
 - a. In the Connection Name field, enter a unique suffix for to the default connection name prefix or use the default suffix provided.
 - b. In the Description field, enter a description for the new connection.
 - c. Edit the connection pool and log parameters as required.
 - d. Click Define to define the connection parameters for the browsing connection. The Configure Connection Parameters page is displayed.

Figure E-4 Configure Connection Parameters Page

On this page, you supply parameters to connect to your DBMS. You specify a JDBC drive and JDBC URL to connect. Note the supported database types and their respective drive names and URL formats at the bottom of the page.

This adapter **REQUIRES** an XA-capable driver. You must use a JDBC Driver that supports connections.

Database User Name:

Database Password:

JDBC Driver:

JDBC URL:

Extra Properties:

6. On the Configure Connection Parameters page, enter the network-related information that enables the application view to interact with the target EIS. It is not necessary for you to enter this information more than once per application view:
 - a. Enter the database username and password.
 - b. In the JDBC Driver and JDBC URL fields, enter the driver specification and the JDBC URL. Depending on the database type, you can also specify additional properties. Note the supported database types and their respective URL formats the bottom of the page.
 - c. Click Continue in the Configure Connection Parameters page to return to the Create New Browsing Connection page.
 - d. Click OK. The Application View Administration page is displayed.

The Application View Administration page summarizes the connection criteria. After events and services are defined, you can view schemas and summaries and delete an event or service from this page.

You have finished creating an application view; you can now add a service to it.

Figure E-5 Application View Administration Page for AppViewTest

This page allows you to add events and/or services to an Application View.

Description: An application view used to test one of the DBMS sample adapters. [Edit](#)

Adapter: BEA_WLS_DBMS_ADK

Version: ADK_SAMPLE

Connections: [Select/Edit](#)

Events

Services



7. To add a service to your new application view, you must supply a name for the service, a description of it, and an SQL statement.

Use the browse link to browse the DBMS sample adapter database schemas and tables and to specify the database table `CUSTOMER_TABLE`.

To add a service:

- a. On the Application View Administration page, click Add in the Services group. The Add Service page is displayed.

Figure E-6 Add Service Page

Unique Service Name:*

Description:

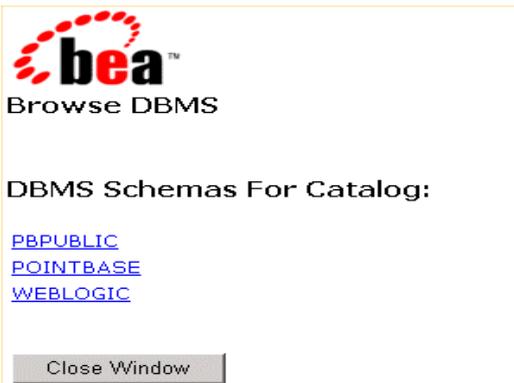
SQL Statement:*

[Browse DBMS...](#)

Syntax Help:
1. Use fully qualified table name (i.e. catalog.schema.table);
2. to gather user input, bracket the column name and type as follows:
"[ColumnName ColumnType]"
3. use variables to parameterize table qualifiers (e.g. myCatalog.mySchema.myTable becomes {myTableQualifiers}myTable)
Parameters created in SQL only support JDBC types. To view the supported types follow the link.
[JDBC Types](#)

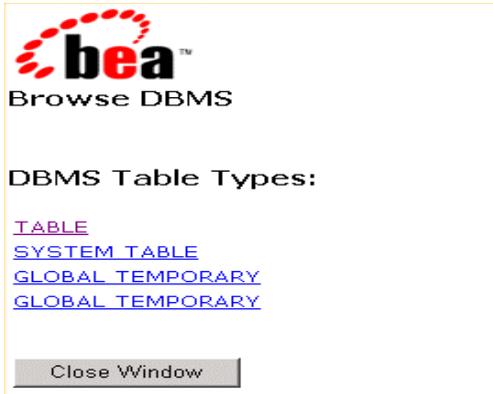
- b. In the Unique Service Name field, enter InsertCustomer.
- c. In the Description field, enter a description of the service.
- d. Click Browse DBMS to view the table and column structure of the database. If you are writing a complex query, you may want to leave the Browse window open so you can later cut and paste table or column names into your query.

Figure E-7 Browse DBMS Page



- e. On the DBMS Schemas for Catalog page, click WEBLOGIC.

Figure E-8 Browse DBMS Table Types Page



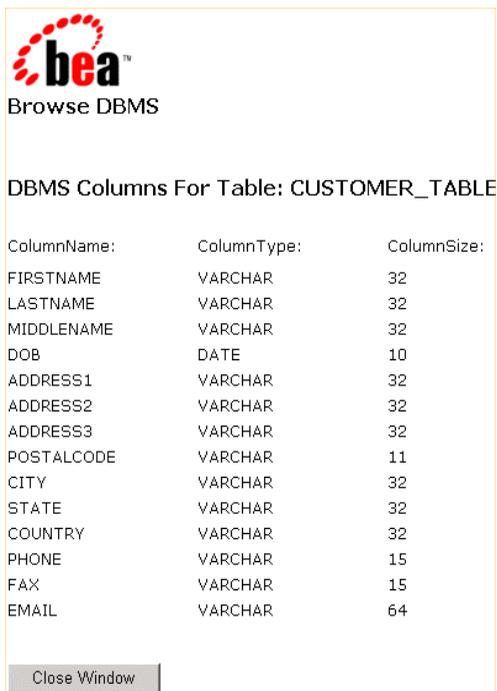
- f. On the DBMS Table Types page, click TABLE.

Figure E-9 DBMS Browse Tables Page



- g. On the Tables list for WEBLOGIC page, click CUSTOMER_TABLE. The Browse window now displays the names and types of the columns.

Figure E-10 Browse DBMS for Table Page



- h. Click Close Window to close the window and return to the Add Service Page.

This window is included in the tour to introduce you to available functionality; you are not required to select any text for this exercise.

- i. On the Service Page, add the following information to the SQL Statement field:

```
Insert into WEBLOGIC.CUSTOMER_TABLE (FIRSTNAME, LASTNAME, DOB)
VALUES ([FIRSTNAME VARCHAR], [LASTNAME VARCHAR], [DOB DATE])
```

- j. Click Add. The Application View Administration page is displayed.

For additional information about adding services, see [“Defining an Application View”](#) in *Using the Application Integration Design Console*.

8. Add an event to your application view. To do so, you must provide a unique name and a description of the event. Then you must specify the database table to which a trigger should be added for the event. You must also specify whether the event is an insert, update, or delete event.

You can use the **Browse DBMS** link to browse the DBMS database schemas and tables and to specify the database table. Then you can click the **Fill table name with selected table** button to populate the field automatically with the specified table name.

To add an event:

- a. On the **Application View Administration** page, click **Add** in the **Events** field. The **Add Event** page is displayed.

Figure E-11 Add Event Page

On this page, you add events to your Application View.

Unique Event Name:*	<input type="text"/>
Description:	<input type="text"/>
Table Name:*	<input type="text"/> Browse DBMS...
Schema Name:	<input type="text"/>
Catalog Name:	<input type="text"/>

Table/Catalog/Schema Name Help...
ORACLE: SCHEMA.TABLENAME, MS SQLSERVER: catalog.schema.tablename, SYBASE: catalog.schema.tablename,
DB2: SCHEMA.TABLENAME, or POINTBASE: SCHEMA.TABLENAME
NOTE: You can specify catalog and schema name as variable references {varName}.
For databases that don't require catalog or schema just leave them blank or define them as variable references
and leave the variable value blank.

Please select the type of event to create:

Insert Event
 Update Event
 Delete Event

- b. In the **Unique Event Name** field, enter `CustomerInserted`.
- c. In the **Description** field, enter a description of the event.
- d. Click the **Browse DBMS** link to view the table and column structure of the database.

Figure E-12 Browse DBMS Tables Page

ACTIVECOLLABORATOR	<input type="radio"/>
ACTIVECONVDEF	<input type="radio"/>
ACTIVECONVERSATION	<input type="radio"/>
ACTIVECONVSTATE	<input type="radio"/>
ACTIVECPA	<input type="radio"/>
ACTIVECSpace	<input type="radio"/>
ACTIVEHUB	<input type="radio"/>
ACTIVEMESSAGE	<input type="radio"/>
ACTIVEMESSAGEData	<input type="radio"/>
ACTIVEMESSAGEEnvelope	<input type="radio"/>
ACTIVEMESSAGEStore	<input type="radio"/>
ACTIVEMESSAGEToken	<input type="radio"/>
ACTIVEPAYLOAD	<input type="radio"/>
ACTIVEPROCESS	<input type="radio"/>
ACTIVEROLE	<input type="radio"/>
ACTIVEROLEDEF	<input type="radio"/>
ACTIVWFInstance	<input type="radio"/>
ACTIVWLCID	<input type="radio"/>
ADDRESSEdMESSAGE	<input type="radio"/>
BUSINESSCALENDAR	<input type="radio"/>
BUSINESSOPERATION	<input type="radio"/>
BUSINESS PROCESS	<input type="radio"/>
BUSINESS PROTOCOL DEFINITION	<input type="radio"/>
CA	<input type="radio"/>
CERTIFICATE	<input type="radio"/>
CONTAINED OBJECT	<input type="radio"/>
CPACOLLABORATORMAP	<input type="radio"/>
CPACONVERSATIONMAP	<input type="radio"/>
CPACSPACEMAP	<input type="radio"/>
CUSTOMER TABLE	<input checked="" type="radio"/>
CXML BINDING	<input type="radio"/>
...	<input type="radio"/>

- e. Select the CUSTOMER TABLE option. Then click the Fill table name with selected table button.

Figure E-13 Add Event Page

On this page, you add events to your Application View.

Unique Event Name:*

Description:

Table Name:* [Browse DRMS...](#)

Schema Name:

Catalog Name:

Table/Catalog/Schema Name Help...
ORACLE: SCHEMA.TABLENAME, MS SQLSERVER: catalog.schema.tablename, SYBASE: catalog.schema.tablename,
DB2: SCHEMA.TABLENAME, or POINTBASE: SCHEMA.TABLENAME
NOTE: You can specify catalog and schema name as variable references {varName}.
For databases that don't require catalog or schema just leave them blank or define them as variable references
and leave the variable value blank.

Please select the type of event to create:

- Insert Event
- Update Event
- Delete Event

- f. Select the Insert Event option.
- g. Click Add. The Application View Administration page is displayed.

Figure E-14 Application View Administration Page for AppViewTest

This page allows you to add events and/or services to an Application View.

Description: An application view used to test one of the DBMS sample adapters. [Edit](#)

Adapter: BEA_WLS_DBMS_ADK

Version: ADK_SAMPLE

Connections: [Select/Edit](#)

Events

CustomerInserted [Edit](#) [Remove Event](#) [View Summary](#) [View Event Schema](#)

Services

InsertCustomer [Edit](#) [Remove Service](#) [View Summary](#) [View Request Schema](#) [View Response Schema](#)



9. Click Save to save the current application view information. You must create or select connections for services and events before testing and publishing the application view.
10. Click Select/Edit. The Connection Information page is displayed. The Connection Information page is organized by connection type: service invocation, event delivery, and design-time browsing connections.

On this page you select or edit the service, event, and browsing connections to use for your Application View

On this page you select or edit the service, event, and browsing connections to use for your Application View

- Service Connection:
 - [Browsing](#) [Create New...](#) [Select Existing...](#)
- Event Connection:
 - [Event](#) [Select Existing...](#)
- Browsing Connection:
 - [Browsing](#) [Create New...](#) [Select Existing...](#)

11. For the purposes of this tour, use an existing service connection. Click on the Select Existing link for service connections and select an existing connection.
12. Click Event to configure the connection properties and log level of your event connection. The Edit Event Connection page is displayed.

On this page you configure the connection properties, log level of your event connection.

Connection Name: * EastCoast.Sales.VendorManagement_Default_Event

Description:

Connection Parameters:

Log Configuration

Set the log verbosity level for this ConnectionFactory.

The Edit Event Connection page allows you to define event parameters and configure what information will be logged for the connection factory. Select one of the following settings for the log:

- Log errors and audit messages
- Log warnings, errors, and audit messages
- Log informational, warning, error, and audit messages
- Log all messages

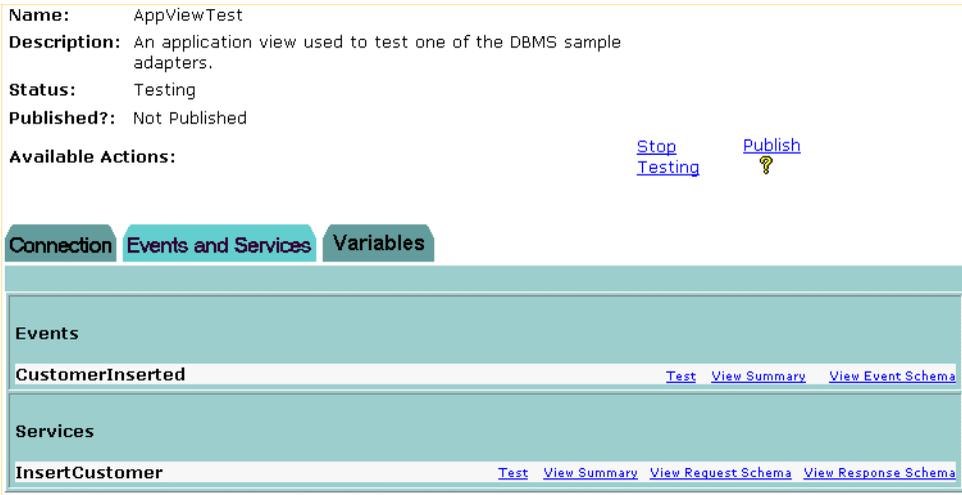
Locate Connection Parameters and click Define to set the event delivery parameters. The Configure Event Delivery Parameters page is displayed.

The event delivery parameters you enter on this page enable connection to an EIS instance and are used when generating events. The properties are specific to the associated adapter and are defined in the `wli-ra.xml` file within the base adapter.

13. After you have set the event delivery parameters, click on Continue to return to the Edit Event Connection page and then click OK to return to the Connection Information page.
14. Click Back to return to the Application View Administration page.

15. In the Application View Administration page, click Test. The Summary for Application View page is displayed. Note that the Status is Testing and a Stop Testing link is displayed.

Figure E-15 Summary for Application View Page



16. Test an event. To ensure that the application view is working correctly, you can test the events and services shown in it. You can test an event by invoking a service or by manually creating the event. The user can also specify how long the application should wait to receive the event.
 - a. In the Events group, on the CustomerInserted line, click Test. The Test Event page is displayed.

Figure E-16 Test Event Page

This page allows you to test an event. You may create the event by invoking a service, or by manually creating the event.

If you want to use a service invocation to create an event, select the service option below, and select the service to invoke. Optionally, you can create the event manually using any tools your EIS provides (for example an interactive SQL tool for the DBMS adapter used to insert a new row to create an insert event).

How do you want to create the event?

Service

Manual

How long should we wait to receive the event?

Time (in milliseconds):

- b. On the Test Event page select the Service option and, from the Service menu, InsertCustomer.
- c. In the How long should we wait to receive the event? field, enter 6000.
- d. Click Test. The Test Service page is displayed.

Figure E-17 Test Service Page

Please fill in any inputs to the service query and click Test.

Test Service: InsertCustomer on application view 'AppViewTest'

Insert into WEBLOGIC.CUSTOMER_TABLE (FIRSTNAME, LASTNAME, DOB) VALUES
([FIRSTNAME VARCHAR], [LASTNAME VARCHAR], [DOB DATE])

Input

FIRSTNAME *text*

LASTNAME *text*

DOB *XML date 'CCYY-MM-DD', e.g. 2003-07-03-05:00*

Enable Response Namespace Enforcement?:

- e. In the FIRSTNAME field, enter a first name.
- f. In the LASTNAME field, enter a last name.
- g. In the DOB field, enter a date of birth. The correct format is specified to the right of the DOB field.
- h. Click Test. The Test Result page is displayed. It shows the contents of the XML documents representing the event you generated and the response generated by the application view.

Figure E-18 Test Result Page

This page shows the results from testing a event.

Generated event of type CustomerInserted on application view AppViewTest

```
<?xml version="1.0"?>
<!DOCTYPE CUSTOMER_TABLE.insert>
<CUSTOMER_TABLE.insert>
  <ADDRESS1></ADDRESS1>
  <ADDRESS2></ADDRESS2>
  <ADDRESS3></ADDRESS3>
  <CITY></CITY>
  <COUNTRY></COUNTRY>
  <DOB>2001-09-12 06:07:15</DOB>
  <EMAIL></EMAIL>
  <FAX></FAX>
  <FIRSTNAME>Jane</FIRSTNAME>
  <LASTNAME>Doe</LASTNAME>
  <MIDDLENAME></MIDDLENAME>
  <PHONE></PHONE>
```

Input to service InsertCustomer on application view AppViewTest

```
<?xml version="1.0"?>
<!DOCTYPE Input>
<Input>
  <FIRSTNAME>Jane</FIRSTNAME>
  <LASTNAME>Doe</LASTNAME>
  <DOB>2001-10-05 04:27:24-05:00</DOB>
</Input>
```

Output from service InsertCustomer on application view AppViewTest

```
<?xml version="1.0"?>
<!DOCTYPE RowsAffected>
```

After successfully testing events and services, you can return to the Summary page and publish the application view. This generates an EJB within the application directory, publishes schema files to the WebLogic Workshop application for use in the XML Mapper, and makes the application view visible within WebLogic Workshop.

For more information on publishing application views, see [“Defining an Application View”](#) in *Using the Application Integration Design Console*.

How the DBMS Sample Adapters Were Developed

This section describes each interface used to develop the DBMS sample adapters. The ADK provides many of the necessary implementations required by a Java Connector Architecture-compliant adapter, but some interfaces cannot be implemented fully until the EIS and its environment are defined. For this reason the DBMS sample adapters were created as a concrete implementation of the abstract classes provided by the ADK.

The procedure for creating the DBMS sample adapters includes the following steps:

- [Step 1: Learn About the DBMS Sample Adapters](#)
- [Step 2: Define Your Environment](#)
- [Step 3: Implement the Server Provider Interface Package](#)
- [Step 4: Implement the Common Client Interface Package](#)
- [Step 5: Implement the Event Package](#)
- [Step 6: Deploy the DBMS Sample Adapter](#)

Step 1: Learn About the DBMS Sample Adapters

To learn how the implementations provided by the ADK are leveraged in the DBMS sample adapters, we recommend that you review the Javadoc and code for the methods defined in this section.

- For the Javadoc, see:
<http://e-docs.bea.com/wli/docs81/javadoc/adk/>
- For the code listing for this package, see:
`WLI_HOME/adapters/dbms/src/com/bea/adapter/dbms/spi`

Note: *WLI_HOME* is the drive or directory in which WebLogic Integration is installed.

Step 2: Define Your Environment

To help you identify and collect critical information about the adapter you are developing before you begin coding, see the [Appendix C, “Adapter Setup Worksheet”](#). For the DBMS sample adapter `BEA_WLS_DBMS_ADK`, the worksheet questions are answered as follows:

Note: Questions preceded by an asterisk (*) are required to use the GenerateAdapterTemplate utility.

1. **What is the name of the EIS for which you are developing an adapter?*
PointBase or Oracle databases.
2. **What version of the EIS are you using?*
PointBase 4.4, Oracle 8.1.7., or Oracle 91 Release 2
3. **Which type of EIS (such as DBMS or ERP) are you using?*
DBMS
4. **What is the name of the vendor for this adapter?*
BEA
5. **What is the version number of this adapter?*
None - Sample Only
6. **What is the logical name of the adapter?*
BEA_WLS_DBMS_ADK
7. *Does the adapter need to invoke functionality within the EIS?*
Yes
If so, then your adapter must support services.
Yes
8. *What mechanism or API is provided by the EIS to allow an external program to invoke EIS functionality?*
JDBC
9. *What information is needed to create a session or connection to the EIS for this mechanism?*
Database URL, driver class, user name, password
10. *What information is needed to determine which function(s) will be invoked in the EIS for a given service?*
Function name, executeUpdate, executeQuery

11. *Does the EIS allow you to query it for input and output requirements for a given function?*

Yes, you can browse data structures.

If so, what information is needed to determine the input requirements for the service?

SQL

12. *Which of the input requirements are static across all requests? Your adapter should encode static information in an InteractionSpec object.*

SQL

13. *Which of the input requirements are dynamic per request? Your adapter should provide an XML schema that describes the input parameters required by this service per request.*

The input requirements would change depending on the SQL expression for the service.

14. *What information is needed to determine the output requirements for the service?*

N/A

15. *Does the EIS provide a mechanism to browse a catalog of functions that can be invoked by your adapter? If so, your adapter should support the browsing of services.*

Yes

16. *Does the adapter need to receive notifications of changes that occur inside the EIS? If so, then your adapter must support events.*

Yes

17. *What mechanism or API is provided by the EIS to allow an external program to receive notification of events in the EIS? The answer to this question will help you determine whether a pull mechanism or a push mechanism is developed.*

None. The DBMS sample adapter was built on the WebLogic Integration event generator using a pull mechanism.

18. *Does the EIS provide a way to determine which events can be supported by your adapter?*

Yes

19. *Does the EIS provide a way to query for metadata for a given event?*

Yes

20. *What locales (defined by language and country) does your adapter need to support?*

Multiple

Step 3: Implement the Server Provider Interface Package

To implement the DBMS sample adapter Server Provider Interface (SPI) and meet the J2EE-compliant SPI requirements, the classes in the ADK were extended to create the following concrete classes:

Table E-1 SPI Class Extensions

This concrete class...	Extends this ADK class...
<code>ManagedConnectionFactoryImpl</code>	<code>AbstractManagedConnectionFactory</code>
<code>ManagedConnectionImpl</code>	<code>AbstractManagedConnection</code>
<code>ConnectionMetaDataImpl</code>	<code>AbstractConnectionMetaData</code>
<code>LocalTransactionImpl</code>	<code>AbstractLocalTransaction</code>

These classes provide connectivity to an EIS and establish a framework for event listening and request transmission, establish transaction demarcation, and allow management of a selected EIS.

ManagedConnectionFactoryImpl

The first step in implementing an SPI for the DBMS sample adapter was to implement the `ManagedConnectionFactory` interface. A `ManagedConnectionFactory` supports connection pooling by providing methods for matching and creating a `ManagedConnection` instance.

Basic Implementation

The ADK provides `com.bea.adapter.spi.AbstractManagedConnectionFactory`, an implementation of the Java Connector Architecture interface `javax.resource.spi.ManagedConnectionFactory`. The DBMS sample adapter extends this class in `com.bea.adapter.dbms.spi.ManagedConnectionFactoryImpl`. [Listing E-1](#) shows the derivation tree for `ManagedConnectionFactoryImpl`.

Listing E-1 `com.bea.adapter.dbms.spi.ManagedConnectionFactoryImpl`

```

javax.resource.spi.ManagedConnectionFactory
|
|-->com.bea.adapter.spi.AbstractManagedConnectionFactory

```

```
|  
|-->com.bea.adapter.dbms.spi.ManagedConnectionFactoryImpl
```

Developers' Comments

The `ManagedConnectionFactory` is the central class of the Java Connector Architecture SPI package. The ADK's `AbstractManagedConnectionFactory` provides much of the required implementation for the methods declared in Sun Microsystems' interface. To extend the ADK's `AbstractManagedConnectionFactory` for the DBMS sample adapter, the key `createConnectionFactory()` and `createManagedConnection()` methods were implemented. Overrides for `equals()`, `hashCode()`, `checkState()` were also written to provide specific behaviors for the databases supported by the DBMS sample adapter.

There are private attributes about which the superclass has no knowledge. When creating your adapters, you must provide setter/getter methods for these attributes. The abstract `createConnectionFactory()` method is implemented to provide an EIS-specific `ConnectionFactory` using the input parameters.

Additionally, `createManagedConnection()` is the main factory method of the class. It checks to see if the adapter is configured properly before doing anything else. It then implements methods of the superclass to get a connection and a password credential object. It then attempts to open a physical database connection; if this succeeds, it instantiates and returns a `ManagedConnectionImpl` (the DBMS sample adapter implementation of `ManagedConnection`), which is given the physical connection.

Other methods are getter/setter methods for member attributes.

ManagedConnectionImpl

A `ManagedConnection` instance represents a physical connection to the underlying EIS in a managed environment. `ManagedConnection` objects are pooled by the application server. For more information, read about how the ADK implements the `AbstractManagedConnection` instance in [“ManagedConnection” on page 6-32](#).

Basic Implementation

The ADK provides `com.bea.adapter.spi.AbstractManagedConnection`, an implementation of the J2EE interface `javax.resource.spi.ManagedConnection`. The DBMS sample adapter extends this class in `com.bea.adapter.dbms`.

`spi.ManagedConnectionImpl`. [Listing E-2](#) shows the derivation tree for `ManagedConnectionImpl`.

Listing E-2 `com.bea.adapter.dbms.spi.ManagedConnectionImpl`

```

javax.resource.spi.ManagedConnection
|
|-->com.bea.adapter.spi.AbstractManagedConnection
|
|   |-->com.bea.adapter.dbms.spi.ManagedConnectionImpl

```

Developers' Comments

This class is thoroughly documented in the Javadoc comments because the `ManagedConnection` is a crucial part of the Java Connector Architecture SPI specification. You should focus on our implementation of the following methods:

- `java.lang.Object getConnection(javax.security.auth.Subject subject, javax.resource.spi.ConnectionRequestInfo connectionRequestInfo)`
- `protected void destroyPhysicalConnection(java.lang.Object objPhysicalConnection)`
- `protected void destroyConnectionHandle(java.lang.Object objHandle)`
- `boolean compareCredentials(javax.security.auth.Subject subject, javax.resource.spi.ConnectionRequestInfo info)`

The `ping()` method is used to check whether a physical database connection (not our `cci.Connection`) is still valid. If an exception occurs, `ping()` is specific about checking the type so that a connection is not needlessly destroyed.

Other methods are EIS-specific or are simply required setters or getters.

ConnectionMetaDataImpl

The `ManagedConnectionMetaData` interface provides information about the underlying EIS instance associated with a `ManagedConnection` instance. An application server uses this information to get run-time information about a connected EIS instance. For more information, read about how the ADK implements the `AbstractConnectionMetaData` instance in [“ManagedConnection” on page 6-32](#).

Basic Implementation

The ADK provides `com.bea.adapter.spi.AbstractConnectionMetaData`, an implementation of the J2EE interface `javax.resource.spi.ManagedConnectionMetaData`. The DBMS sample adapter extends this class in `com.bea.adapter.dbms.spi.ConnectionMetaDataImpl`. [Listing E-3](#) shows the derivation tree for `ConnectionMetaDataImpl`.

Listing E-3 `com.bea.adapter.dbms.spi.ConnectionMetaDataImpl`

```
javax.resource.spi.ManagedConnectionMetaData
|
|-->com.bea.adapter.spi.AbstractConnectionMetaData
|
|   |-->com.bea.adapter.dbms.spi.ConnectionMetaDataImpl
```

Developers' Comments

The ADK's `AbstractConnectionMetaData` class implements the following:

- `javax.resource.cci.ConnectionMetaData`
- `javax.resource.spi.ManagedConnectionMetaData`

This implementation of the `ConnectionMetaData` class uses a `DatabaseMetaData` object. Because the ADK's abstract implementation was used, you must provide EIS-specific knowledge when implementing the abstract methods in this class.

LocalTransactionImpl

The `LocalTransaction` interface provides support for transactions that are managed within an EIS resource manager (that is, transactions that do not require an external transaction manager). For more information, read about how the ADK implements the `AbstractLocalTransaction` instance in "[LocalTransaction](#)" on page 6-35.

Basic Implementation

The ADK provides `com.bea.adapter.spi.AbstractLocalTransaction`, an implementation of the J2EE interface `javax.resource.spi.LocalTransaction`. The DBMS sample adapter extends this class in `com.bea.adapter.dbms.spi.LocalTransactionImpl`. [Listing E-4](#) shows the derivation tree for `LocalTransactionImpl`.

Listing E-4 com.bea.adapter.dbms.spi.LocalTransactionImpl

```

javax.resource.spi.LocalTransaction
|
|-->com.bea.adapter.spi.AbstractLocalTransaction
|
|   |-->com.bea.adapter.dbms.spi.LocalTransactionImpl

```

Developers' Comments

This class utilizes the ADK's abstract superclass which provides logging and event notification. The superclass implements both the CCI and SPI LocalTransaction interfaces provided by Sun. The DBMS sample adapter's concrete class implements the three abstract methods of the superclass:

- doBeginTx()
- doCommitTx()
- doRollbackTx()

Step 4: Implement the Common Client Interface Package

To implement the DBMS sample adapter Common Client Interface (CCI) and meet the J2EE-compliant CCI requirements, several classes in the ADK were extended to create the following concrete classes.

Table E-2 CCI Class Extensions

This concrete class . . .	Extends this ADK class . . .
ConnectionImpl	AbstractConnection
InteractionImpl	AbstractInteraction
InteractionSpecImpl	InteractionSpecImpl

These classes provide access back-end systems. The client interface specifies the format of both the request and response records for a given interaction with the EIS.

Note: Although implementation of the CCI is optional in the *Java Connector Architecture 1.0* specification, it is likely to be required in the future. For your reference, the DBMS sample adapter provides a complete implementation.

ConnectionImpl

A `Connection` represents an application-level handle that is used by a client to access an underlying physical connection. The actual physical connection associated with a `Connection` instance is represented by a `ManagedConnection` instance. For more information, read about how the ADK implements the `AbstractConnection` instance in [“Connection” on page 6-37](#).

Basic Implementation

The ADK provides `com.bea.adapter.cci.AbstractConnection`, an implementation of the J2EE interface `javax.resource.cci.Connection`. The DBMS sample adapter extends this class in `com.bea.adapter.dbms.cci.ConnectionImpl`. [Listing E-5](#) shows the derivation tree for `ConnectionImpl`.

Listing E-5 `com.bea.adapter.dbms.cci.ConnectionImpl`

```
javax.resource.cci.Connection
|
|-->com.bea.adapter.cci.AbstractConnection
|
|   |-->com.bea.adapter.dbms.cci.ConnectionImpl
```

Developers' Comments

The `ConnectionImpl` class is the DBMS sample adapter's concrete implementation of the `javax.resource.cci.Connection` interface. It extends the ADK's `AbstractConnection` class. The actual physical connection associated with a connection instance is represented by a `ManagedConnection` instance.

A client gets a connection instance by using the `getConnection()` method on a `ConnectionFactory` instance. A connection can be associated with zero or more interaction instances. The simplicity of this concrete class is a testament to the power of extending the ADK's base classes.

InteractionImpl

The `Interaction` instance enables a component to execute EIS functions. An interaction instance is created from a connection and is required to maintain its association with the `Connection` instance. For more information, read about how the ADK implements the `AbstractInteraction` instance in [“Interaction” on page 6-38](#).

Basic Implementation

The ADK provides `com.bea.adapter.cci.AbstractInteraction`, an implementation of the J2EE interface `javax.resource.cci.Interaction`. The DBMS sample adapter extends this class in `com.bea.adapter.dbms.cci.InteractionImpl`. [Listing E-6](#) shows the derivation tree for `InteractionImpl`.

Listing E-6 `com.bea.adapter.dbms.cci.InteractionImpl`

```

javax.resource.cci.Interaction
|
|-->com.bea.adapter.cci.AbstractInteraction
|
|   |-->com.bea.adapter.dbms.cci.InteractionImpl

```

Developers' Comments

The `InteractionImpl` class is the concrete implementation of the ADK's `Interaction` object. The methods are EIS-specific implementations of methods required by the Java Connector Architecture and the ADK.

Both versions of the Java Connector Architecture's

`javax.resource.cci.InteractionExecute()` method (the central method of this class) were implemented for the DBMS sample adapter. The main logic for the `execute()` method includes the following signature:

```
public Record execute(InteractionSpec ispec, Record input)
```

Because this method returns the actual output record from the interaction, it is called more often than other methods.

The second implementation is provided as a convenience method. This form of `execute()` includes the following signature: `public boolean execute(InteractionSpec ispec,`

Record input, Record output). The second implementation's logic returns a boolean, which indicates only the success or failure of the interaction.

InteractionSpecImpl

An `InteractionSpecImpl` holds properties for driving an interaction with an EIS instance. An `InteractionSpec` is used by an interaction to execute the specified function on an underlying EIS.

The CCI specification defines a set of standard properties for an `InteractionSpec`, but an `InteractionSpec` implementation is not required to support a standard property if that property does not apply to its underlying EIS.

The `InteractionSpec` implementation class must provide getter and setter methods for each of its supported properties. The convention followed in the getter and setter methods should be based on the Java Beans design pattern. For more information, read about how the ADK implements the `InteractionSpecImpl` instance in [“InteractionSpec” on page 6-49](#).

Basic Implementation

The ADK provides `com.bea.adapter.cci.InteractionSpecImpl`, an implementation of the J2EE interface `javax.resource.cci.InteractionSpec`. The DBMS sample adapter extends this class in `com.bea.adapter.dbms.cci.InteractionSpecImpl`. [Listing E-7](#) shows the derivation tree for `InteractionSpecImpl`.

Listing E-7 `com.bea.adapter.dbms.cci.InteractionSpecImpl`

```
javax.resource.cci.InteractionSpec
|
|-->com.bea.adapter.cci.InteractionSpecImpl
|
|-->com.bea.adapter.dbms.cci.InteractionSpecImpl
```

Developers' Comments

An implementation class for the `InteractionSpec` interface is required to implement the `java.io.Serializable` interface. `InteractionSpec` extends the ADK `InteractionSpec` in order to add setter and getter methods for the String attribute `m_sql`. The getter and setter

methods should be based on the Java Beans design pattern, as specified in the *Java Connector Architecture 1.0* specification.

Step 5: Implement the Event Package

This package contains only one class: the DBMS sample adapter `EventGeneratorWorker`. It does the work for the event generator for the DBMS sample adapter.

EventGenerator

The `EventGenerator` class implements the following interfaces:

- `com.bea.wlai.event.IEventGenerator`
- `java.lang.Runnable`

Basic Implementation

The DBMS sample adapter event generator extends the ADK's `AbstractPullEventGenerator` because a database cannot *push* information to the event generator; you therefore need to *pull* or *poll* the database for changes about which you want to be notified. [Listing E-8](#) shows the derivation tree for `EventGenerator`.

Listing E-8 EventGenerator

```
com.bea.adapter.event.AbstractEventGenerator
|
|-->com.bea.adapter.event.AbstractPullEventGenerator
|
|   |-->com.bea.adapter.dbms.event.DbmsEventGeneratorWorker
```

Developers' Comments

This concrete implementation of the ADK's `AbstractPullEventGenerator` implements the following abstract methods:

- `protected abstract void postEvents(IEventRouter router) throws Exception`
- `protected abstract void setupNewTypes(List listOfNewTypes)`
- `protected abstract void removeDeadTypes(List listOfDeadTypes).`

It also overrides the following methods:

- `void doInit(Map map)`
- `void doCleanUpOnQuit()`.

These methods are EIS-specific: they are used to identify an event within the context of the EIS while interacting with the database to create and remove event definitions and events.

Additionally, these methods can be used to create and remove triggers on the database that are activated when an event occurs.

The key method of the class is `postEvents()`. It creates the `IEvent` objects of the data taken from rows in the `EVENT` table of the database. This method takes an `IEventRouter` as an argument. After creating an `IEvent` using the `IEventDefinition` object's `createDefaultEvent()` method, the `postEvents()` method populates the event data propagates the event to the router by calling `router.postEvent(event)`. Once the event is sent to the router, the method deletes the relevant rows of event data from the database.

The method `setupNewTypes()` creates new event definitions, making sure that the appropriate triggers are created for the database. For each event definition, the method creates a trigger information object that describes the catalog, schema, table, `triggerType`, and trigger key represented by the event definition. A map of trigger keys is maintained so that triggers are not redundantly added to the database. If the map does not contain the new key, trigger text for the database is generated.

The method `removeDeadTypes()` also creates a trigger information object, but this object also searches for one or more matching event types. All event definitions that match this trigger are removed from the map, and then the trigger itself is removed from the database.

Step 6: Deploy the DBMS Sample Adapter

After implementing the SPI, CCI, and event interfaces, deploy the adapter by completing the following steps:

- [Step 6a: Set Up Your Environment](#)
- [Step 6b: Update the ra.xml File](#)
- [Step 6c: Create the RAR File](#)
- [Step 6d: Build the JAR and EAR Files](#)
- [Step 6e: Create and Deploy the EAR File](#)

Step 6a: Set Up Your Environment

Before deploying the adapter in a WebLogic Integration environment, determine the location of the adapter on your computer. The adapter resides in `WLI_HOME/adapters/dbms`. You must replace `WLI_HOME` with the pathname for the directory in which WebLogic Integration is installed. We refer to this location as `ADAPTER_ROOT`.

Step 6b: Update the ra.xml File

The DBMS sample adapter provides the `ra.xml` file in the adapter's RAR file (`META-INF/ra.xml`). Because the DBMS sample adapter extends the `AbstractManagedConnectionFactory` class, the following properties are provided in the `ra.xml` file:

- `LogLevel`
- `LanguageCode`
- `CountryCode`
- `MessageBundleBase`
- `LogConfigFile`
- `RootLogContext`
- `AdditionalLogContext`

The DBMS sample adapter also requires the declarations listed in the following table.

Table E-3 `ra.xml` Properties

Property	Example
<code>UserName</code>	Username for DBMS sample adapter login
<code>Password</code>	Password for username
<code>DataSourceName</code>	Name of the JDBC connection pool

You can view the complete `ra.xml` file for the DBMS sample adapter in:

`WLI_HOME/adapters/dbms/src/rar/META-INF/`

Step 6c: Create the RAR File

Class files, logging configuration information, and message bundle(s) should be collected in a JAR file, which should then be bundled, along with `META-INF/ra.xml`, into a RAR file. The `Ant build.xml` file demonstrates how to construct this RAR file properly.

Step 6d: Build the JAR and EAR Files

To build the JAR and EAR files, complete the following procedure:

1. In a text editor, open either `antEnv.cmd` (Windows) or `antEnv.sh` (UNIX) in `WLI_HOME/adapters/utils`. Assign valid pathnames to the following variables:
 - `BEA_HOME` - The top-level directory for your BEA products.
 - `WLI_HOME` - The location of your Application Integration directory.
 - `JAVA_HOME` - The location of your Java Development Kit.
 - `WL_HOME` - The location of your WebLogic Server directory.
 - `ANT_HOME` - The location of your Ant installation, such as `WLI_HOME/adapters/utils`.
2. Execute `antEnv` from the command line so the new values of the required environment variables take effect.
3. Go to `WLI_HOME/adapters/dbms/project`.
4. Execute `ant.cmd release` (Windows) or `ant.sh release` (UNIX) from the `WLI_HOME/adapters/dbms/project` directory to build the adapter.

Step 6e: Create and Deploy the EAR File

The DBMS sample adapter is displayed by creating and deploying an EAR file. To do so, complete the following procedure:

1. Declare the adapter's EAR file in your domain's `config.xml` file, as shown in the following listing.

Listing E-9 Declaring the DBMS Sample Adapter's EAR File

```
<!-- This deploys the EAR file -->
```

```

<Application Deployed="true" Name="BEA_WLS_DBMS_ADK"
Path="WLI_HOME/adapters/dbms/lib/BEA_WLS_DBMS_ADK.ear">
    <ConnectorComponent Name="BEA_WLS_DBMS_ADK" Targets="myserver"
        URI="BEA_WLS_DBMS_ADK.rar" />
    <WebAppComponent Name="DbmsEventRouter" Targets="myserver"
        URI="BEA_WLS_DBMS_ADK_EventRouter.war" />
    <WebAppComponent Name="BEA_WLS_DBMS_ADK_Web" Targets="myserver"
        URI="BEA_WLS_DBMS_ADK_Web.war" />
</Application>

```

Note: Replace *WLI_HOME* with the pathname of the WebLogic Integration installation directory for your environment.

2. Open the WebLogic Server Administration Console by entering:

```
http://host:port/console
```

In this URL *host* is the name of your server and *port* is the port at which your server listens. For example:

```
http://localhost:7001/console
```

3. In the WebLogic Server Administration Console:

- a. Add the adapter group to the default WebLogic Server security realm.
- b. Add a user to the adapter group.
- c. Save your changes.

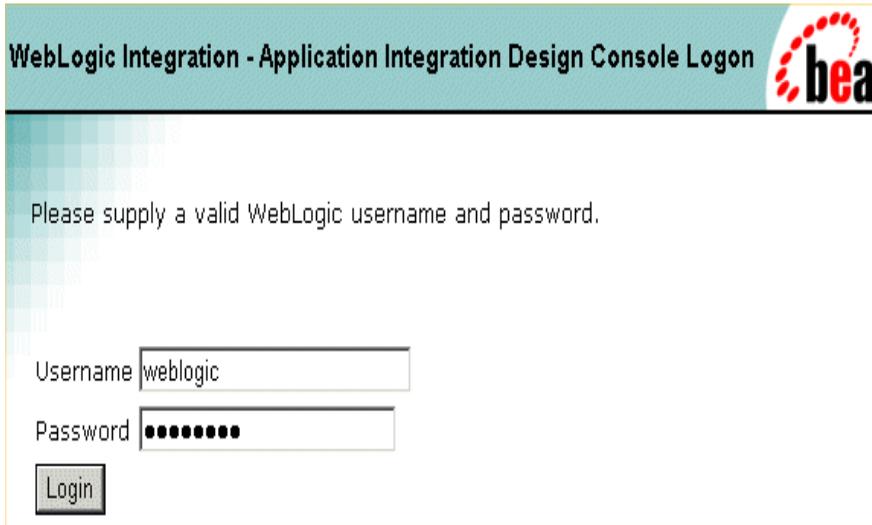
4. To configure and deploy application views, go to:

```
http://host:port/wlai
```

In this URL *host* is the name of your server and *port* is the port at which your server listens. For example:

```
http://localhost:7001/wlai
```

The Application Integration Design Console Logon page is displayed.



5. Log on to WebLogic Integration by entering your username and password in the appropriate fields.
6. Configure and publish your application views by completing the procedures described in [“Defining Application Views”](#) in *Using Application Integration*.

Database-Specific Error Messages

When using the DBMS sample adapter, you may see database-specific error messages written to the log file. The section identifies some of the more common database-specific error messages.

From time to time, when using Sybase or MSSQL databases, warnings are issued stating that the active database and/or language has been changed. These warnings come from the Sybase and MSSQL databases when the active database or language is changed on a connection. Since the catalog is changed at various points in the DBMS sample adapter, users of the sample adapter will inevitably see these messages. These warnings are harmless and can be ignored.

When an adapter instance in an MSSQL XA environment is automatically suspended and resumed, error messages similar to the following are thrown.

```
<Oct 15, 2003 4:40:30 PM PDT> <Error> <JDBC> <BEA-001112>  
<Test "SELECT COUNT(*) FROM sysobjects" set up for pool "wlaiPool" failed with  
exception: "javax.transaction.xa.XAException: [BEA][SQLServer JDBC Driver]  
No more data available to read.">
```

This is a JDBC-level error, generated as the JDBC container cleans up existing/dead connections to the restarted DBMS instance. This type of exception is normal in this case; the EIS is available, the adapter instance is resumed successfully, and the JDBC container recovers.

When using the DBMS sample adapter with a Microsoft SQL Server database, use the WebLogic Server Administration Console to enable the `TestReservedConnection` parameter for the connection pool used for application integration. If the parameter is not enabled, the auto resume or manual resume features do not work and a `SQLException` similar to the following is thrown:

```
java.sql.SQLException: [BEA][SQLServer JDBC Driver]No more data
available to read
```

How the DBMS Sample Adapter Design-Time GUI Was Developed

The design-time GUI is an interface that allows a user to create application views, add services and events, and deploy an adapter that is hosted in a WebLogic Integration environment. This section explains how the design-time GUI for the DBMS sample adapter was developed:

- [Step 1: Identify Requirements](#)
- [Step 2: Identify Required Java Server Pages](#)
- [Step 3: Create the Message Bundle](#)
- [Step 4: Implement the Design-Time GUI](#)
- [Step 5: Write Java Server Pages](#)

Step 1: Identify Requirements

Before development of the design-time GUI for the DBMS was begun, values for the following parameters needed to be determined:

- Which database(s) will be supported?
- How many levels of browsing will be supported?
- Determine the DBMS schema generation.
- Will the adapter support testing of services and events?

Step 2: Identify Required Java Server Pages

The DBMS sample adapter uses the Java Server Pages (JSPs) delivered with the ADK for a design-time GUI. Additional JSPs have been added, however, to provide adapter-specific functionality. The additional JSPs are described in the following table.

Table E-4 Additional ADK JSPs

Filename	Description
<code>addevent.jsp</code>	The Add Event page allows a user to add a new event to the application view.
<code>addservice.jsp</code>	The Add Service page allows the user to add a new service to the application view.
<code>browse.jsp</code>	<p>The Browse page handles the logic flow and display for the Browse window of the DBMS sample adapter. Although this functionality was developed specifically for this adapter, it illustrates a fairly common interaction between a design-time interface and an underlying adapter.</p> <p>The Browse page calls the <code>DesignTimeRequestHandler</code> (handler) of the DBMS sample adapter, which extends the ADK's <code>AbstractDesignTimeRequestHandler</code>. The best way to understand the browse functionality of the DBMS sample adapter is to deploy the adapter and use your Web browser to access the design-time framework.</p>
<code>confconn.jsp</code>	The Confirm Connection page provides a form on which a user can specify connection parameters for the EIS.
<code>testform.jsp</code>	<p>The Testform page is included (<code><jsp:include page='testform.jsp' /></code>) in the ADK's <code>testsrcv.jsp</code> page. It accesses the <code>InteractionSpec</code> for this interaction and displays the SQL for the service. It then creates a form for gathering the user input required to test a service.</p> <p>To create a form, Testform gets the <code>RequestDocumentDefinition</code> from the handler's application view and then passes it, with the <code>.jsp</code> Writer, to a utility class called <code>com.bea.adapter.dbms.utils.TestFormBuilder</code>. This class creates the form.</p>

Step 3: Create the Message Bundle

To support the internationalization of all text labels, messages, and exceptions, the DBMS sample adapter uses a message bundle based on a text property file. The property file includes both name-value pairs copied from the `BEA_WLS_SAMPLE_ADK` property file, and new entries, added for properties specific to the DBMS sample adapter.

The message bundle for the DBMS sample adapter resides in the `WLI_HOME/adapters/dbms/src` directory, which was installed with the ADK. For details, see the `BEA_WLS_DBMS_ADK.properties` file in the same directory.

For additional instructions on creating a message bundle, see the JavaSoft tutorial on internationalization at:

<http://java.sun.com/docs/books/tutorial/i18n/index.html>

Step 4: Implement the Design-Time GUI

To implement the design-time GUI, you need to create a `DesignTimeRequestHandler` class. This class accepts user input from a form and provides methods to perform a design-time action. For more information about the `DesignTimeRequestHandler`, see “[Step 4: Implement the Design-Time GUI](#)” on page 9-26.

The DBMS sample adapter public class `DesignTimeRequestHandler` extends `AbstractDesignTimeRequestHandler`, and it provides the methods shown in the following table.

Table E-5 Methods for the DBMS Sample Adapter Design-Time GUI

Method	Description
<code>browse(java.lang.String dbtype, com.bea.connector.DocumentRecord input)</code>	Handles the back-end behavior for the Browse functionality of the <code>addevent.jsp</code> and <code>addservc.jsp</code> pages.
<code>getAdapterLogicalName()</code>	Returns the adapter’s logical name and helps the parent class when entities such as application views are deployed.
<code>getManagedConnectionFactoryClass()</code>	Returns the adapter’s <code>ManagedConnectionFactory</code> implementation class, which is then used by a parent class to get a CCI connection to the EIS.

Table E-5 Methods for the DBMS Sample Adapter Design-Time GUI (Continued)

Method	Description
supportsServiceTest ()	Indicates that this adapter supports the testing of services at design time.
initServiceDescriptor (ActionResult result, IServiceDescriptor sd, HttpServletRequest request)	<p>Initializes a service descriptor, which involves creating the request and response schemas for a service. A typical approach is to execute an Interaction against the EIS to retrieve metadata and then transform that metadata into an XML schema.</p> <p>Consequently, the CCI interface provided by the adapter was used. This method is called from the <code>addsrvc</code> method of the <code>AbstractDesignTimeRequestHandler</code>.</p>
initEventDescriptor (ActionResult result, IEventDescriptor ed, HttpServletRequest request)	<p>Initializes an event descriptor. The event descriptor provides information about an event on an application view. Subclasses must supply an implementation of this method.</p> <p>If events are not supported, then the implementation should throw an <code>UnsupportedOperationException</code>. This method is not called (by the <code>AbstractDesignTimeRequestHandler</code>) until the name and definition of the event have been validated and it is confirmed that the event does not already exist for the application view.</p>
GetDatabaseType ()	Used to determine the type of database management system being used. WebLogic Integration supports PointBase and Oracle.

Step 5: Write Java Server Pages

Consider incorporating the following practices into your development process:

- [Use Custom JSP Tags](#)
- [Save an Object's State](#)
- [Write the WEB-INF/web.xml Deployment Descriptor](#)

Use Custom JSP Tags

Because the Java Server Pages (JSPs) are displayed on the `display.jsp` page, `display.jsp` is the first `.jsp` file that needs to be written. The ADK provides a library of custom JSP tags, which are used extensively throughout the Java server pages of the ADK and DBMS sample adapter. These tags support numerous features, such as the ability to add validation and to save a value entered in a field when the user clicks a button.

Save an Object's State

While writing the JSPs, with the ADK, for your adapter, you may need to save an object's state. You can do so in any of a number of ways. The `AbstractDesignTimeRequestHandler` maintains an `ApplicationDescriptor` of the application view being edited. This [file?] is often the best place in which to save state; calls to the handler from this file are fast and efficient.

As an alternative, you can request a Manager Bean from the `AbstractDesignTimeRequestHandler`, using its convenience methods (`getApplicationManager()`, `getSchemaManager()`, and `getNamespaceManager()`) to retrieve information from the repository about an application view. This method is more time-consuming but, occasionally, it may be necessary. Because it is a JSP, you can also use the session object, although everything put in the session must explicitly implement the `java.io.Serializable` interface.

Write the WEB-INF/web.xml Deployment Descriptor

Write the `WEB-INF/web.xml` deployment descriptor. In most cases, you should use the adapter's `web.xml` file as a starting point and modify components as necessary. To see the `web.xml` file for this adapter, go to:

`WLI_HOME/adapters/dbms/src/war/WEB-INF/web.xml`

For detailed information, see the BEA WebLogic Server product documentation at:

<http://edocs.bea.com>

Run-Time Considerations

You can manage your application integration run-time environment using the WebLogic Integration Administration Console. For more information, see “[Application Integration](#)” in *Managing WebLogic Integration Solutions* at the following URL:

<http://edocs.bea.com/wli/docs81/manage/ai.html>

The following sections describe administration issues specific to the DBMS sample adapter.

Changing Event Connections

Please consider the following when modifying the `EventSchema` parameter of the event connection for DBMS sample adapter instances:

- Ensure that the new event schema has all the required event staging tables. You can run `evnt.sql` to ensure that this is the case. The location for `evnt.sql` is as follows:

```
WLI_HOME\adapters\dbms\sql\database_type
```

where `WLI_HOME` is the WebLogic Integration installation directory and `database_type` is the directory for the database you are using.

Note: In WebLogic Integration 8.1 Service Pack 3 or later, you must run both `evnt.sql` and `evnt_trigger.sql` if your database type is Oracle.

- Ensure that the user for the source schema has insert privileges on the event schema’s event staging tables. Consult your database documentation for instructions on granting insert privileges to users.

For example, if `user1` owns the schema `user1` that contains `CUSTOMER_TABLE`, the trigger on `CUSTOMER_TABLE` tries to insert event data into the staging tables in the `user2` schema. Make sure `user1` has insert privileges on the event staging tables in `user2`’s schema.

Note: You may need to redeploy servers if there is a change in the database instance used by the JDBC connection pool for the application integration event.

The schema name/table owner associated with an event trigger retains the information provided at design time. When you edit the event connection from the WebLogic Integration Administration Console, you point to a new connection setting for the event tables; any event metadata is associated with the new connection after the application is redeployed from the WebLogic Integration Administration Console. The event trigger still points to the old schema name.

When changing event connections to a new database, you must ensure that the new database has a schema and a table with the same name as the previous settings. The new table is used for triggering events.

You can point the event trigger to the new schema/table name using environment variables. Note that the old trigger is not deleted and that the adapter instance must be redeployed. To delete the old trigger, redeploy the application from the WebLogic Server Administration Console. The environment variable for the schema name is capable of changing the schema name/table owner where the trigger polls.

It is good practice to limit events to one per table in the database. This avoids multiple event pollers attempting to access one table.

Administering a DBMS Sample Adapter Instance Used by Multiple Application Views

Please consider the following when setting up DBMS sample adapter instances used by multiple application views. Only one adapter instance should be used for all application views needing an event for a given table/trigger type. If you have two application views that each depend on `CUSTOMER_TABLE` inserts, make sure they both use the same adapter instance. This is because the event generator for the adapter instance deletes the insert trigger from `CUSTOMER_TABLE` when all its event subscriptions are undeployed. If `AppView1` uses `adapter_instance_1` and is undeployed, `adapter_instance_1` deletes the insert trigger for `CUSTOMER_TABLE`. If `AppView2`, which also needs `CUSTOMER_TABLE` inserts, uses `adapter_instance_2`, it will be starved of events because `adapter_instance_1` previously deleted the trigger. If `AppView2` uses `adapter_instance_1`, `adapter_instance_1` knows that `AppView2` still needs the `CUSTOMER_TABLE` insert event and does not delete the trigger even if `AppView1` is undeployed.

Multiple Event Generators Using the Same DBMS Instance

You can distribute event generators among nodes in a WebLogic Integration cluster. This improves load balancing and fault tolerance of event generation. For an example of multiple event generators using the same DBMS instance, see [“Example: DBMS Sample Adapter” on page 7-25](#).

You must set event generator targets when in a clustered environment. A blank target (" ") does not trigger events on any node in the cluster. (In a single node environment, no entry is needed; a blank event generator target value kicks off events for the one server.)

Learning to Develop Adapters Using the DBMS Sample Adapters

Index

A

- abstract base class 2-2
- AbstractConnection 6-37
- AbstractConnectionFactory A-2
- AbstractConnectionMetaData A-2
- AbstractDesignTimeRequestHandler 9-26, 9-28, 9-32
- AbstractDocumentRecordInteraction 6-45
- AbstractInputTagSupport 9-5
- AbstractInteraction 6-38
- AbstractLocalTransaction 6-35
- AbstractManagedConnection 6-34, 6-37, A-2
- AbstractManagedConnectionFactory A-2
- AbstractManagedConnectionMetaData 6-33, 6-34
- AbstractPullEventGenerator 7-10, 7-12, 7-13
- AbstractPushEventGenerator 7-12
- ActionResult 9-4
- adapter 1-6
 - event 1-7
 - service 1-7
- adapter logical name 2-5, 4-4, 5-2, 7-7, A-2
- Adapter Setup Worksheet 7-5
- adapter setup worksheet 4-1
- adapter, deploying 2-8
- addevent.jsp 9-32
- addservc 9-28
- addservc.jsp 9-34
- ADK 1-2
- ADK tag library 9-33, 9-34
- adk-eventgenerator.jar 7-10, 7-11
- Ant 3-3, 3-4, 4-5, 6-54, 7-7, 9-24
 - why use 3-4

- ant release 4-6
- ANT_HOME 4-5, E-40
- antEnv.cmd 4-5
- antEnv.sh 4-5, 4-6
- Apache Project 2-4, 5-2, 7-8
- Apache Software Foundation 5-2
- appender 5-5, 5-8
- Application Integration Design Console 1-3
- application view 1-6, 1-7, 2-3, 9-1, 9-26, 9-32
- application view descriptor 9-26
- application view security 9-26
- Application View Summary page 9-26
- assertion checking 6-37
- avaScript library 2-2

B

- BEA_HOME 4-5, E-40
- build.xml 6-54

C

- category
 - ancestor 5-3
 - assigning a priority to 5-5
 - child 5-3
 - naming 5-4
 - parent 5-3
 - properties 5-3
 - referring to multiple appenders 5-5
 - root 5-4
- CCI 6-34, 6-36, 6-40, 6-41, 6-48, 6-49, 6-50, 6-51, 6-52, 6-55, 6-56, 9-26, 9-29
- chmod u+x ant 4-5

- classes
 - abstract 3-1
- com.bea.adapter.cci.AbstractDocumentRecordInteraction 6-52
- com.bea.adapter.cci.AbstractDocumentRecordInteraction 6-45
- com.bea.adapter.cci.AbstractInteraction 6-45
- com.bea.adapter.cci.DesignTimeInteractionSpecImpl 6-46
- com.bea.adapter.cci.DocumentDefinitionRecord 6-45
- com.bea.adapter.cci.ServiceInteractionSpecImpl 6-45
- com.bea.adapter.event 7-11
- com.bea.adapter.spi.AbstractConnectionMetaData 6-48
- com.bea.adapter.spi.ConnectionEventLogger 6-34
- com.bea.adapter.spi.NonManagedConnectionEventListener 6-34
- com.bea.adapter.spi.NonManagedConnectionManager 6-35
- com.bea.adapter.test.TestHarness 6-54
- com.bea.connector.DocumentRecord 6-41
- com.bea.document.IDocument 6-41, 6-42, B-2
- com.bea.web.ActionResult 9-4
- com.bea.web.ControllerServlet 9-4
- com.bea.web.RequestHandler 9-3
- com.bea.web.tag.AbstractInputTagSupport 9-5
- com.bea.web.tag.IntegerTagSupport 9-6
- com.bea.web.validation.IntegerWord 9-6, 9-7
- com.bea.web.validation.Word 9-4, 9-5, 9-6
- Common Client Interface
- confconn.jsp 9-26, 9-29
- config.xml 4-6
- Connection 6-36, 6-37
- connection 6-38
 - event 1-5
- ConnectionEventListener 6-34
- ConnectionFactory 6-47
- ConnectionFactory.getMetaData 6-52

- ConnectionFactoryImpl 6-47
- ConnectionManager 6-34, 6-35, A-2
- ConnectionMetaData 6-48
- ConnectionRequestInfo 6-35
- ConnectionSpec 6-48
- ControllerServlet 9-4, 9-5, 9-6, 9-7, 9-29, 9-30, 9-33, 9-34
- Creating a Custom Development Environment 4-1

D

- Data Extraction 7-10
- data transformation 7-9, 7-18
- DbmsEventGeneratorWorker.java 7-16
- deployment helper 1-6, 2-2
- design time 2-1, 9-27
 - GUI 1-6
- designtime
 - GUI 1-6
- design-time GUI 1-1
- DesignTimeHelper 1-6
- DesignTimeInteractionSpecImpl 6-46
- DesignTimeRequestHandler 9-26
- Developing an Event Adapter 7-1, 8-1
- display.jsp 9-40, 9-41
- DisplayPage 9-31
- Document Object Mode
- DocumentDefinitionRecord 6-45
- DocumentRecord 6-41, 6-45
- DocumentRecordInteraction 6-45
- DOM 5-2, 6-42, B-2

E

- EAR file 2-8, 2-9
- EAR files 2-9
- Enterprise Adapter Archive file 2-8
- enterprise information system (EIS) 1-4
- Enterprise Java Beans (EJB) 1-5
- error.jsp 9-40
- event connection 1-3

- Event Generator 7-9, 7-10, 7-12
- event generator 2-2, 7-9
- event listener 6-33
- event router 6-55
- EventGenerator 7-14, 7-18
- EventMetaData 7-12
- EventRouter 7-9, 7-15
- exception handling 6-35
- ExecutionTimeout 6-49

F

- form processing 9-2
 - classes 9-3
 - prerequisites 9-6
 - sequence 9-5
- framework 1-2
 - designtime 1-2, 1-6, 3-4, 9-1
 - logging 1-2, 2-2, 2-4, 2-5, 5-1, 5-3, 6-33, 6-34, 6-37, 6-38, 6-47
 - packaging 1-2, 1-7
 - runtime 1-2, 2-1, 2-2
- FunctionName 6-49

G

- GenerateAdapterTemplate 3-2, 3-3, 4-1, 5-2, 7-7, A-2
- GenerateAdapterTemplate.cmd 4-2
- GenerateAdapterTemplate.sh 4-2
- GUI 1-1

I

- I18N 5-13
- IDocument 3-4, 6-41, 6-42, 7-13, B-2, B-3
- IDocumentDefinition 6-45
- IEventDefinition 7-9, 7-10, 7-13
- ILogger 5-4
- IndexedRecord 6-51
- installer 4-5
- Interaction 6-36, 6-38, 6-46

- interaction 6-38
- InteractionSpec 6-38, 6-39, 6-45, 6-49
- InteractionSpecImpl 6-50
- InteractionVerb 6-49
- internationalization 7-8, 9-4
- IPushHandler 7-12

J

- Jakarta project 7-8
- JAR file 2-9
- Java 2-5
- Java exception 9-2, 9-40
- Java package base name 4-4
- Java Reflection 9-8, 9-32
- Java Server Page. see JSP
- java.io.Serializable 6-47
- java.util.Map 6-35
- JAVA_HOME 4-5, E-40
- JavaBean 6-48
- Javadoc 3-3, 4-6
- JavaScript library 1-6
- javax.resource.cci.Connection 6-37
- javax.resource.cci.ConnectionFactory 6-47
- javax.resource.cci.ConnectionMetaData 6-33, 6-48
- javax.resource.cci.ConnectionSpec 6-48
- javax.resource.cci.Interaction 6-38, 6-46
- javax.resource.cci.InteractionSpec 6-49, 6-50
- javax.resource.cci.LocalTransaction 6-50
- javax.resource.cci.Record 6-41, 6-51, 6-52
- javax.resource.cci.ResourceAdapterMetaData 6-52
- javax.resource.ReferenceableInterfaces 6-47
- javax.resource.spi 6-23, 6-50
- javax.resource.spi.ConnectionEventListener 6-34
- javax.resource.spi.ConnectionManager 6-34, 6-35
- javax.resource.spi.ConnectionRequestInfo 6-35
- javax.resource.spi.LocalTransaction 6-35

- javax.resource.spi.ManagedConnection 6-34
- javax.resource.spi.ManagedConnectionMetaDat
a 6-33
- JNDI 6-47
- JSP 1-6, 2-2, 9-1, 9-6, 9-8, 9-28, 9-30, 9-40, 9-41
- JSP template 1-6
- JSP templates 2-2
- JUnit 6-54
- junit.framework.TestCase 6-54
- junit.framework.TestSuite 6-54

L

- L10N 5-13
- label, displaying for a form field 9-31, 9-33, 9-35
- local transaction 6-50
- localization 6-23, 7-8, 9-4, 9-5
- LocalTransaction 6-35, 6-50
- log categories 2-5
- Log4j 2-4, 5-2
- log4j 5-2, 7-8
- LogConfigFile 9-31
- Logging 2-4, 5-1
- logging 5-2, 7-8
 - appender 5-2
 - appenders 5-5
 - AUDIT 5-4
 - categories 5-2, 5-3
 - category 7-8
 - concepts 5-2
 - DEBUG 5-4
 - ERROR 5-4
 - INFO 5-4
 - internationalization 2-5, 5-1, 5-2, 5-4, 9-4
 - localization 2-4, 2-5, 5-1, 5-4, 9-4
 - message layout 5-3
 - priorities 5-4
 - priority 5-2, 5-4
 - WARN 5-4
- logging configuration file 5-2
- logging toolkit 2-4, 5-2

M

- main.jsp 9-40
- ManagedConnection 6-24, 6-33, 6-34, 6-37
- ManagedConnectionFactory 6-24, 6-55, 9-27,
9-29, 9-32
- ManagedConnectionImpl 6-33
- ManagedConnectionMetaData 6-24, 6-33
- ManagedConnectionMetaDataImpl 6-34
- manifest 6-11
- manifest file 6-11
- MappedRecord 6-51
- Message Bundle 6-23, 7-8
- message bundle 2-5, 6-23, 9-33
- message bundles 9-31
- MessageBundleBase 9-31
- metadata 3-4, 6-34, 6-36, 6-41, B-3

N

- namespace 6-47
- NDC 5-13
- NonManagedScenarioTestCase 6-55

O

- overview.html 4-6

P

- package format 4-4
- PatternLayout 5-6
- priority 5-4
- pull data extraction 7-9, 7-10, 7-13
- push data extraction 7-9, 7-10, 7-13
- PushEvent 7-12, 7-13

R

- ra.xml 9-31
- RAR file 2-8, 2-9
- Record 6-38, 6-49, 6-51
- RecordImpl 6-52

- Request Document Definition 6-40
- RequestHandler 9-3, 9-4, 9-5, 9-6, 9-8, 9-29, 9-30, 9-33
- RequestHandlerClass 9-31
- resource adapter, see adapter
- ResourceAdapterMetaData 6-52
- ResourceAdapterMetaDataImpl 6-52
- Response Document Definition 6-40
- RootLogContext 9-31
- Runtime 2-1
- runtime 2-5, 9-41
- run-time engine 2-2

S

- Sample Adapter 3-1
- sample adapter 3-1, 3-3, 4-1, 6-34
- sample.client.ApplicationViewClient 6-56
- sample.event.EventGenerator 3-3
- sample.event.OfflineEventGeneratorTestCase 6-55
- sample.spi.ConnectionMetaDataImpl 3-3
- sample.spi.ManagedConnectionFactoryImpl 3-2
- sample.spi.ManagedConnectionImpl 3-2
- sample.spi.NonManagedScenarioTestCase 6-55
- sample.spi.ResourceAdapterImpl 3-3
- sample.web.DesignTimeRequestHandler 3-3
- Schema Object Model
 - see CCI
 - see DOM
 - see SOM
 - see SPI
 - service
 - synchronous 1-4
- service connection 1-3
- service descriptor 9-28
- Service Provider Interface
- SOM 3-4
- SPI 6-34, 6-36, 6-55, 6-56, 9-27, A-2
- State management 6-37

- submit button, displaying on a form 9-31, 9-33, 9-35

T

- tag library 2-2
- test harness 7-32
- test.properties 6-54, 6-55
- TestSuite 6-54
- text field, displaying size 9-31, 9-33, 9-35
- transaction, local A-2
- transaction, XA A-2

U

- Unique Business Name 6-40

V

- validation 9-2
- validator 9-4

W

- WAR file 2-8
- web application 2-3, 3-4, 9-2, 9-30
- web application descriptor 2-3
- web.xml 2-3, 7-15, 9-4, 9-30, 9-33, 9-41
- WebLogic 6.0 5-2
- WebLogic Integration A-2
- WebLogic Server 5-5, A-2
- WebLogic Server 6.0 A-2
- WL_HOME 4-5, E-40
- WLAI_HOME E-40
- WLI_HOME 4-5
- Word 9-4, 9-7

X

- XA transaction A-2
- XCCI 6-40, 6-44
 - design pattern 6-46

- DocumentRecords 6-40
- Services 6-40
- XERCES 5-2
- XML 2-3
 - document 6-42, 7-18, B-2
 - request document 1-5
 - schema 1-5, 2-3, 3-4, 6-46, 7-2, 7-5, 7-18,
B-3
- XML Tools 3-4
- XPath 6-42, B-2