



BEA AquaLogic Data Services Platform™

Client Application Developer's Guide

Version: 2.0.1
Document Date: June 2005
Revised: September 2005

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. Introducing Data Services Platform for Client Applications

Simplifying Data Programming	1-1
What is a Data Services Platform Client?	1-2
Deciding Which Programming Model to Use	1-3
Service Data Objects (SDO).	1-5
Development Steps	1-6
Security Considerations in Client Applications.	1-6
Runtime Client JAR Files.	1-7
Enabling Data Source Updates	1-7

2. Client Programming with Service Data Objects (SDO)

What is Service Data Objects (SDO) Programming?	2-1
SDO and the Data Services Platform	2-2
Looking at an SDO Client Application	2-4
Looking at a Data Graph	2-6
XML Schema-to-Java Type Mapping	2-8
ArrayOf Types	2-11
Static versus Dynamic Interfaces	2-12
Static Interface	2-13
Dynamic Data Object Interface.	2-16

Common SDO Operations and Examples	2-18
Instantiating and Populating Data Objects	2-18
Static Interface Instantiation	2-18
Dynamic Interface Instantiation	2-19
Accessing Data Object Properties	2-19
Typed Property Access	2-19
Untyped Property Access	2-20
Setting Data Object Properties	2-21
Adding New Data Objects	2-22
Deleting Data Objects	2-23
Submitting Data Object Changes	2-24
Typed Interface Submit	2-24
Untyped Interface Submit	2-25
Introspecting a Data Object	2-25
Working with Data Graphs	2-27
XPath Support in the Untyped SDO API	2-28
For More Information	2-29

3. Enabling SDO Data Source Updates

Overview	3-1
How Data Source Updates Work	3-2
Decomposition	3-2
Update Processing Sequence	3-4
Update Overrides	3-5
Update Behavior	3-6
Update Order	3-6
Understanding Property Maps	3-7
Multi-Level Data Services	3-7

Transaction Management	3-8
SDO Submit Inside a Containing Transaction	3-8
When to Customize Updates	3-8
Developing an Update Override Class	3-10
UpdateOverride Interface	3-10
Development Steps	3-12
Testing Submit Results	3-14
Understanding Update Override Context	3-15
Physical Level Update Override Considerations	3-17
Update Programming Patterns	3-19
Override Decomposition and Update	3-19
Augment Original Data Object Content	3-20
Accessing the Data Service Mediator Context	3-20
Accessing the Decomposition Map	3-20
Customizing an Update Plan	3-23
Executing an Update Plan	3-25
Retrieving the Container of the Current Data Object	3-25
Retrieving and Updating Data Through Other Data Services	3-26
Setting the Log Level	3-26
Configuring Optimistic Locking	3-29
Handling Foreign and Primary Keys	3-30
Returning Computed Primary Keys	3-30
Managing Key Dependencies	3-30
Foreign Keys	3-31

4. Accessing Data Services from Java Clients

Overview of the Data Services Platform Mediator API	4-1
What's in the Data Service Mediator API?	4-3

Setting the Classpath	4-4
Creating the Mediator Client JAR File from the Command Line.....	4-5
Build an EAR File	4-5
Build the Client JAR	4-6
How to Use the Mediator API.....	4-7
Getting a WebLogic JNDI Context for DSP.....	4-8
Using the Static Data Service Interface	4-9
Using the Dynamic Data Service Interface.....	4-12
Using Navigation Functions	4-14
Bypassing a Function's Data Cache When Using the Mediator API.....	4-15

5. Accessing Data Services from Workshop Applications

WebLogic Workshop and Data Services Platform	5-1
Data Service Controls	5-2
Use With Page Flow, Web Services, Portals, Business Processes	5-2
Data Service Control (JCX) File	5-3
Design View	5-3
Source View	5-4
Running Ad Hoc Queries Through a Data Service Control	5-7
Creating Data Service Controls	5-8
Step 1: Create a Project in an Application	5-8
Step 2: Start WebLogic Server, If Not Already Running.....	5-8
Step 3: Create a Folder in a Project.....	5-8
Step 4: Create the Data Service Control.....	5-9
Step 5: Enter Connection Information to the WebLogic Server	5-11
Step 6: Select Data Service Functions to Add to Your Control.....	5-12
Modifying Existing Data Service Controls.....	5-14
Changing a Method Used by a Control	5-14

Adding a New Method to a Control	5-15
Updating an Existing Control if Schemas Change	5-15
Using Data Services Platform with NetUI	5-16
Generating a Page Flow From a Control	5-16
To Generate a Page Flow From a Data Service Control	5-16
Adding a Data Service Control to an Existing Page Flow	5-18
Adding Service Data Objects (SDO) Variables to the Page Flow	5-19
To Add a Variable to a Page Flow	5-21
To Initialize the Variable in the Page Flow	5-21
Working with Data Objects	5-22
Displaying Array Values in a Table or List	5-23
Adding a Repeater to a JSP File	5-23
Adding a Nested Level to an Existing Repeater	5-25
Adding Code to Handle Null Values	5-26
Using the Data Services Platform in Business Process Projects	5-27
Creating a Data Service Control	5-27
Adding a Data Service Control to a JPD File	5-28
Setting Up the Control in the Business Process	5-28
Submitting Changes from a Business Process	5-29
Caching Considerations When Using Data Service Controls	5-29
Bypassing the Cache When Using a Data Service Control	5-29
Cache Bypass Example When Using a Data Service Control	5-30
Security Considerations When Using Data Service Controls	5-30
Security Credentials Used to Create Data Service Controls	5-31
Testing Controls With the Run-As Property in the JWS File	5-31
Trusted Domains	5-31
Configuring Trusted Domains	5-32

6. Exposing Data Services through Web Services

Exposing Data Services as Web Services.	6-1
Adding a Data Service Control to a Web Service Project	6-2
Creating a Web Service From a Data Service Control	6-6

7. Using the Data Services Platform JDBC Driver

About the Data Services Platform JDBC Driver.	7-2
Features of the Data Services Platform JDBC Driver	7-2
Data Services Platform and JDBC Driver Terminology	7-3
Installing the Data Services Platform JDBC Driver with JDK 1.4x	7-3
Using the JDBC Driver	7-5
Obtaining a Connection	7-5
Using the preparedStatement Interface	7-6
Getting Data Using JDBC	7-6
Connecting to the JDBC Driver from a Java Application	7-7
Connecting to Data Services Platform Client Applications Using the ODBC-JDBC Bridge from	
Non-Java Applications.	7-12
Using the EasySoft ODBC-JDBC Bridge	7-12
Using OpenLink ODBC-JDBC Bridge.	7-16
Using Reporting Tools with the Data Services Platform ODBC-JDBC Driver	7-23
Crystal Reports 10 - ODBC	7-23
Crystal Reports 10 - JDBC.	7-33
Business Objects 6.1 - ODBC	7-36
Microsoft Access 2000 - ODBC	7-49
DSP and SQL Type Mappings	7-54
SQL-92 Support	7-55
Supported Features.	7-55
Limitations.	7-58

8. Advanced Topics

Applying Filter Data Service Results	8-1
Using Filters	8-2
Specifying Filter Effects	8-3
Filter Operators	8-5
Ordering and Truncating Data Service Results	8-6
Consuming Large Result Sets (Streaming API and Writing Results To a File)	8-7
Using the Streaming Interface	8-7
Writing Data Service Function Results to a File	8-11
Using Ad Hoc Queries	8-11
Transaction Considerations	8-14
Setting Up Data Source Aliases for Relational Sources Accessed by DSP	8-15
Setting Up Handlers for Web Services Accessed by DSP	8-17

Introducing Data Services Platform for Client Applications

This chapter introduces you to developing BEA AquaLogic Data Services Platform client applications. It covers the following topics:

- [Simplifying Data Programming](#)
- [What is a Data Services Platform Client?](#)
- [Deciding Which Programming Model to Use](#)
- [Service Data Objects \(SDO\)](#)
- [Development Steps](#)
- [Security Considerations in Client Applications](#)
- [Runtime Client JAR Files](#)
- [Enabling Data Source Updates](#)

Note: Data Services Platform was initially named Liquid Data. Some artifacts of the original name remain in the product, installation path, and components.

Simplifying Data Programming

The Data Services Platform (DSP) significantly simplifies how client applications access and use data. In a typical organization, data comes from a variety of sources, including distributed databases, files, applications from partners or e-commerce exchange markets. With DSP, client applications can use

heterogeneous data through a unified service layer without having to contend with the complexity of working with distributed data sources using various connection mechanisms and data formats.

DSP provides a uniform, consolidated interface for accessing and updating heterogeneous back-end data. It enables a services-oriented approach to information access using data services.

From the perspective of a client application, a data service typically represents a distinct business entity, such as a customer or order. Behind the scenes, the data service may aggregate the data that comprises a single view of the data, for example, from multiple sources and transform it in a number of ways. A data service may be related to other data services, and it is easy to follow these relationships in DSP. Data services insulate the client application from the details of the composition of each business entity. The client application only has to know the public interface of the data service.

This document describes how to create DSP-aware client applications. It explains the various client access mechanisms that DSP supports and its main client-side data programming model, including Service Data Objects (SDO). It also describes how to create update-capable data services using the DSP update framework.

What is a Data Services Platform Client?

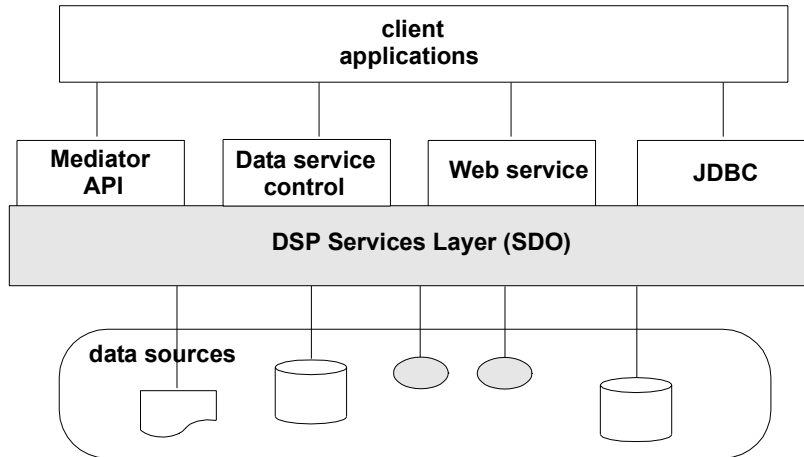
A DSP client is any process that consumes data services. A client application may be, for example, a Java program, non-Java programs such as .NET applications, BEA WebLogic Workshop applications, or JDBC/ODBC clients.

As illustrated in [Figure 1-1](#), DSP supports several access mechanisms:

- Java clients can use data service functions through the *Mediator API*.
- Workshop applications (such as portals, business processes, and web applications) can use a data service control.
- By generating a Web service for data services, you can make DSP services available to a wide array of WebLogic and non-WebLogic applications and integration channels.
- The DSP JDBC driver provides JDBC and ODBC clients, such as reporting tools, with SQL-based access to DSP information.

Whatever the client type, DSP gives developers a uniform, services-oriented mechanism for accessing and modifying heterogeneous data from external sources. Developers can focus on the business logic of the application rather than details of various data source connections and formats.

Figure 1-1 Accessing DSP Services



Deciding Which Programming Model to Use

Developers can choose from several models for accessing DSP services. The model chosen will depend on the access mechanism you decide to use. The possible access methods are:

- Data Mediator API
- Data service control
- Web Services
- JDBC/ODBC

Each access method has its own advantages and use. [Table 1-2](#) provides a description of each of these access methods and summarizes the advantages of the various models for accessing DSP services.

Table 1-2 Data Services Platform Access Models

Access mechanism	Description	Advantages/When to use...
Data Service Mediator API	<p>A Java interface for using data services. Returns data as data objects, providing full support for Service Data Objects (SDO) programming.</p> <p>For more information, see Chapter 4, “Accessing Data Services from Java Clients.”</p>	<ul style="list-style-type: none"> • Can be developed with standard Java IDEs such as BEA WebLogic Workshop, Eclipse, IntelliJ, JBuilder, and others. • Easy-to-use approach to developing Java programs that use external data. • Provides several access modes, including a dynamic (untyped) interface, a static (typed) interface, and an ad hoc query interface. • Seamless ability to submit data changes.
Data Service Control	<p>A WebLogic Workshop control for accessing DSP resources.</p> <p>For more information, see Chapter 5, “Accessing Data Services from Workshop Applications.”</p>	<ul style="list-style-type: none"> • Best suited for BEA WebLogic Workshop applications, including portals, business process workflows, and pageflows. • Leverages BEA WebLogic Workshop features for working with controls, such as drag-and-drop method and variable generation. • Provides an ad hoc query interface for a highly dynamic approach to querying information. • Seamless ability to submit data changes.
Web Service	<p>A data service can be wrapped as a Web service, providing the data service with the benefits of web service features.</p> <p>For more information, see Chapter 6, “Exposing Data Services through Web Services.”</p>	<ul style="list-style-type: none"> • Makes standard Web service features available to data services, such as WS-Security, WSDL descriptors, and more. • Makes data services usable from .NET applications, or other non-Java programs. • Ideal for XML-based SOA architectures
JDBC/ODBC	<p>Client applications can use JDBC or ODBC to access DSP services using SQL queries. The DSP JDBC driver supports SQL-92.</p> <p>For more information, see Chapter 7, “Using the Data Services Platform JDBC Driver.”</p>	<ul style="list-style-type: none"> • Works with applications designed for JDBC access, such as Cognos business intelligence software and Crystal Reports. • Enables users to leverage existing SQL skills and resources. • Limited to "flat" views of data.

Service Data Objects (SDO)

Service Data Objects (SDO), a specification proposed jointly by BEA and IBM, is a Java-based architecture and API for data programming. SDO unifies data programming against heterogeneous data sources. It simplifies data access, giving data consumers a consistent, uniform approach to using data whether it comes from a database, web service, application, or any other system.

SDO uses the concept of *disconnected data graphs*. Under this architecture, a client gets a copy of externally persisted data in a data graph, which is a structure for holding data objects. The client operates on the data remotely; that is, disconnected from the data source. If data changes need to be saved to the data source, a connection to the source is re-acquired. Holding connections and locks the data at the source for the minimum time possible in this way maximizes the scalability and performance of applications.

To SDO clients, the data has a uniform appearance no matter where it came from or what its source format is. Enabling this unified view of data in the SDO model is the Data Service Mediator. The mediator is the intermediary between data clients and back-end systems. It allows clients to access data services and invoke their functions to acquire data or submit data changes. DSP serves as such a SDO mediator.

On the client side, information takes the form of data objects. Data objects are the basic unit of information prescribed by the SDO architecture. SDO has both static (or strongly typed) and dynamic (or loosely typed) interfaces for working with data objects.

Static interfaces provide a programmer-friendly model for getting and setting properties in a data object. Accessors are generated for each property in the data type of a data service, for example `getCustomerName()` and `setCustomerName()` for a Customer data object. The dynamic interface, on the other hand, is useful when a predefined static interface is unknown or undefined at runtime. Dynamic interface calls are in the form `get("CustomerName")` and `set("CustomerName", "J. Dough")`.

In keeping with the goals of a service-oriented architecture (SOA), data graphs are self-describing. The metadata API enables applications, tools, and frameworks to inspect information on the data contained in a data graph. The data is described by an XML schema, which describes the names of properties, their types, and more.

For details on using SDO, see [Chapter 2, “Client Programming with Service Data Objects \(SDO\).”](#)

Development Steps

There are several steps you will take to when developing your application:

1. Choose the data access approach that best suits your needs. ([Table 1-2, “Data Services Platform Access Models,”](#) on [page 1-4](#) describes the advantages of the different access mechanisms.)
2. Determine what data services you want to use in your client application by looking at the available services using the Data Services Platform Console. The DSP Console acts as a sort of service registry in the DSP architecture; it shows what data services are available and what functions they provide.
3. Make sure you have the required JAR files for developing DSP client applications (see [“Runtime Client JAR Files,”](#) below). To use the typed data service and SDO interfaces, acquire the generated mediator client JAR from the DSP administrator.
4. Develop the DSP client application.

A prior or possibly parallel task is creation of DSP services and their functions, which are accessed by client applications. Development of these entities is described in the [Data Services Developer's Guide](#).

Security Considerations in Client Applications

Data Services Platform administrators can control access to deployed DSP resources through role-based security policies. DSP leverages and extends the security features of the underlying WebLogic platform. Roles can be set up in the WebLogic Administration Console. (Refer to the Data Services Platform [Administration Guide](#) for information about the DSP Console.)

Access policies for DSP resources can be defined at any level— on all data services in a deployment, individual data services, individual data service functions, or even on individual elements returned by the functions of a data service.

For complete information on WebLogic security, see:

<http://e-docs.bea.com/wls/docs81/security/index.html>

Runtime Client JAR Files

The DSP API includes the packages listed in [Table 1-3](#).

Table 1-3 Required Java Archive Files

Name	Description	Location
[App]-ld-client.jar	Contains the generated static interfaces for data services and their data types. The name of the file is prefixed by the name of the DSP application from which the typed interface is generated.	(Supplied by your Data Services Platform administrator.)
ld-client.jar	The dynamic, or untyped, data service APIs, including generic data service interfaces and ad hoc query interfaces.	<bea_home>\weblogic81\liquiddata\lib\
wlsdo.jar	The interfaces defined in the SDO specification, including untyped data interfaces and data graph interfaces.	<bea_home>\weblogic81\liquiddata\lib\
weblogic.jar	The common WebLogic APIs.	<bea_home>\weblogic81\server\lib\
xbean.jar xqrl.jar wlxbean.jar	XMLBean classes and interfaces on which the Data Services Platform SDO implementation relies. Also enables XPath expressions in untyped data accessors.	<bea_home>\weblogic81\server\lib\

Enabling Data Source Updates

SDO gives client applications create, read, update, and delete access to external data. Changes to data object property values can be persisted to back-end data sources. The programming details associated with data changes are hidden from the client. The client can update data in several heterogeneous, distributed sources with a single update call.

DSP makes it easy to create data services that can apply changes to information as well as access from back-end data sources. For relational data sources, DSP propagates the updates to the back-end source automatically. For non-relational sources, such as web services, or when you want to customize relational updates, DSP provides an update framework that you can use to implement your own data source updates.

For details on using SDO to update data, see [Chapter 3, “Enabling SDO Data Source Updates.”](#)

Introducing Data Services Platform for Client Applications

Client Programming with Service Data Objects (SDO)

This chapter describes the BEA AquaLogic Data Services Platform (DSP) client-side data programming model and framework based on Service Data Objects (SDO). It introduces SDO and describes common programming tasks undertaken with SDO. It covers the following topics:

- [What is Service Data Objects \(SDO\) Programming?](#)
- [SDO and the Data Services Platform](#)
- [Common SDO Operations and Examples](#)
- [XPath Support in the Untyped SDO API](#)

What is Service Data Objects (SDO) Programming?

The Service Data Object (SDO) specification defines a Java-based programming architecture and API for data access. A central goal of SDO is to provide client applications with a unified programming model for working with data in a disconnected way, regardless of its physical source or format. SDO thereby simplifies the way applications use data.

SDO specifies a data programming API as well as an architecture. The architecture part of the specification describes the components for enabling data access, such as mediators which serve as the intermediary between the client and back-end sources. In SDO terms, DSP is a member.

In terms of client data programming, SDO has characteristics in common with other data access technologies, such as JDBC and Java Data Objects (JDO). Like JDO, SDO provides a static API for accessing data through typed accessors (for example, `getCustomerName()`). Like JDBC's `RowSet` interface, SDO has a dynamic API for accessing data through untyped accessors (for example,

`getString("CUSTOMER_NAME")`). What distinguishes SDO from other technologies, however, is that SDO gives applications both a static and a dynamic API for accessing data, along with a disconnected model for accessing externally persisted data.

SDO and the Data Services Platform

DSP implements the SDO specification as its Java client programming model. In concrete terms, this means that when a client application invokes a read function on a data service through the Data Service Mediator API (also called the Mediator API) or a data service control, it gets the information back as a *data object*. A data object is the fundamental component of the SDO programming model and represents a unit of structured information.

The role of data objects, along with other key components in the SDO framework, are summarized as follows:

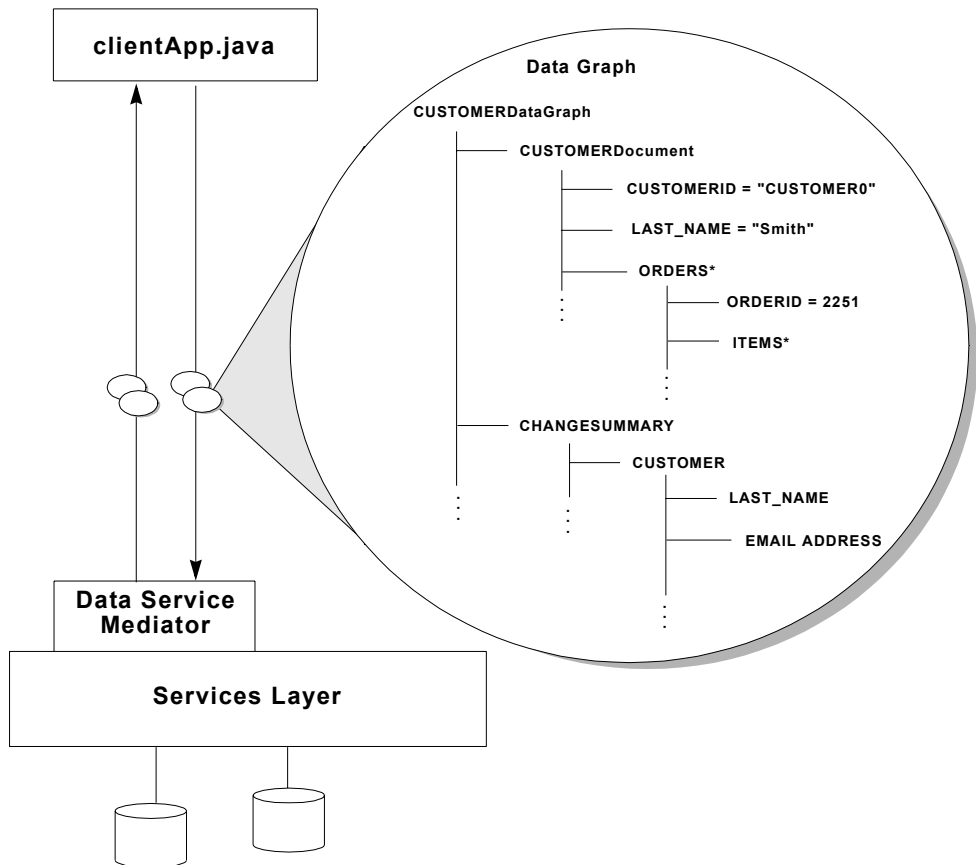
- **Data Object.** A data object holds values as properties, which can be either simple values (such as the CUSTOMERID property shown in [Figure 2-1](#)) or other data objects (such as ORDERS). Static methods are generated for reading, setting, and adding data object properties.
- **Properties.** Properties are the attributes of a data object. A property can be either a simple type or a complex type. A simple type corresponds to a leaf node in an XML document tree, usually of a primitive type such as `String` or `int`. A complex type corresponds to a branch node in the tree, and contains another data object.
- **Data Graph.** A structure for holding data objects, a data graph consists of a single root object, any number of additional objects and properties, the XML schema for the object, and a change summary (a log of data changes). Data users exchange information with the DSP server components by passing data graphs back and forth.
- **Data Service Mediator.** A mediator resides between SDO clients and data sources and acts as the intermediary between them. It receives data requests and change submissions from the client and relays information back to the client in the form of a data graph.

According to the SDO specification, there can be many types of mediators, each intended for a particular type of query language or back-end system. DSP includes the Data Service Mediator, a mediator specialized for acting as the intermediary between DSP services and clients.

Note: For more on the Data Service Mediator API, see [Chapter 4, “Accessing Data Services from Java Clients.”](#)

See also [“For More Information” on page 2-29](#).

Figure 2-1 SDO Components with Data Graph



Data objects are passed between the mediator and client applications in a data graph. A data graph can have only a single root object (for example, **CUSTOMERDocument** in Figure 2-1). For result involving repeating objects, therefore, a single root element prefixed by "ArrayOf" is introduced to serve as the data graph root node. For more information, see ["ArrayOf Types" on page 2-11](#).

DSP leverages XMLBeans technology to generate static interfaces from XML. As a result, many features of the underlying XMLBeans technology are available in SDO as well. For more information on XMLBeans, see <http://xmlbeans.apache.org>.

Furthermore, all SDO types inherit from **XmlObject**, so factory classes for creating instances and parsing data objects are present from the inherited **XmlObject** interface.

Looking at an SDO Client Application

This section presents a simple example ([Listing 2-1](#)) of an SDO client application. The example gets information from DSP through the Mediator API by instantiating a remote interface to a data service and invoking data service functions. It extracts information for a customer, modifies it, and submits the change to the mediator for update to the source or sources.

Listing 2-1 Sample SDO Client Application

```
import java.util.Hashtable;
import javax.naming.InitialContext;
import dataServices.customerDB.customer.ArrayOfCUSTOMERDocument;
import dataServices.customerdb.CUSTOMER;

public class ClientApp {

    public static void main(String[] args) throws Exception {

        Hashtable h = new Hashtable();
        h.put(Context.INITIAL_CONTEXT_FACTORY,
            "weblogic.jndi.WLInitialContextFactory");
        h.put(Context.PROVIDER_URL, "t3://localhost:7001");
        h.put(Context.SECURITY_PRINCIPAL, "weblogic");
        h.put(Context.SECURITY_CREDENTIALS, "weblogic");
        Context context = new InitialContext(h);

        // get the Customer data service and run dynamic invocation of data service
        CUSTOMER custDS = CUSTOMER.getInstance(context, "RTLApp");
        ArrayOfCUSTOMERDocument myCustomer =
            (ArrayOfCUSTOMERDocument) custDS.invoke("CUSTOMER", null);

        // get and show customer name
        String existingFName =
            myCustomer.getArrayOfCUSTOMER().getCUSTOMERArray(0).getFIRSTNAME();
        String existingLName =
            myCustomer.getArrayOfCUSTOMER().getCUSTOMERArray(0).getLASTNAME();

        System.out.println(" \n----- \n Before Change: \n");
        System.out.println(existingFName + existingLName);

        // change the customer name
        myCustomer.getArrayOfCUSTOMER().getCUSTOMERArray(0).setFIRSTNAME("J.B.");
        myCustomer.getArrayOfCUSTOMER().getCUSTOMERArray(0).setLASTNAME("Kwik");
        custDS.submit(myCustomer, "ld:DataServices/CustomerDB/CUSTOMER");

        // re-query and print new name
```



```

myCustomer = (ArrayOfCUSTOMERDocument) custDS.invoke("CUSTOMER",null);
String newFName =
    myCustomer.getArrayOfCUSTOMER().getCUSTOMERArray(0).getFIRSTNAME();
String newLName =
    myCustomer.getArrayOfCUSTOMER().getCUSTOMERArray(0).getLASTNAME();
String newName = newFName.concat(newLName);

System.out.println(" \n----- \n After Change: \n");
System.out.println(newFName + newLName);    }
}

```

The example above includes the following processing steps:

1. First the application instantiates a remote interface to the Customer data service, passing a JNDI context that identifies the WebLogic Server where DSP is deployed.
2. It then calls the `invoke()` function of the data service, pouring the results into an `ArrayOfCUSTOMERDocument` object.
3. A new value for the `FIRSTNAME` and `LASTNAME` property of the `CUSTOMER` is set and the change is submitted.
4. The `invoke()` function is executed again, and the results are printed to output.

In the sample, an SDO is acquired through the Data Service Mediator API and modified through the SDO static API. Keep in mind that you can acquire data objects through the data service control as well. Therefore, it is useful to note the difference in the sample between mediator API calls and SDO calls.

The data service interface is instantiated and invoked through mediator API calls as follows:

```

ArrayOfCUSTOMERDocument myCustomer =
    ( ArrayOfCUSTOMERDocument) ds.invoke("CUSTOMER", null);

```

Once the data object is created, its properties are accessed using the SDO static interface (which returns the actual type of that node):

```

myCustomer.getArrayOfCUSTOMER().getCUSTOMERArray(0).getFIRSTNAME();

```

As mentioned elsewhere, the SDO client data programming model includes both static and dynamic interfaces. The equivalent call using the dynamic interface would be as follows:

```

myCustomer.getArrayOfCUSTOMER().getCUSTOMERArray(0).get("FIRSTNAME");

```

(This will returns an `Object` instance that you will need to cast to the type ordinarily returned by the static interface.)

Finally, the change is submitted to the data service mediator for propagation to the back-end source through the `submit()` function in the mediator interface.

Although code for handling exceptions is not shown in the example, a runtime error in SDO throws an `SDOException`. If an exception is generated by a data source, it is wrapped in an `SDOException`.

Note: For more information on the Mediator API, see [Chapter 4, “Accessing Data Services from Java Clients.”](#)

For complete documentation on the mediator and SDO APIs, refer to the [SDO Update Javadoc](#).

Looking at a Data Graph

Data objects and data graphs can be serialized and printed to standard output. In fact, viewing a printed data graph when developing client applications can help you understand how data objects are composed.

[Listing 2-2](#) shows a data graph associated with a modified data object. The printout is produced by the following code:

```
myCustomer.getArrayOfCUSTOMER().getCUSTOMERArray(0).setFIRSTNAME("J.B.");  
myCustomer.getArrayOfCUSTOMER().getCUSTOMERArray(0).setLASTNAME("Nimble");  
System.out.println(myCustomer.getDataGraph());
```

Notice the data graph features in the following listing examples:

- Metadata, in the form of an XML schema description, applicable to the contained data.
- A change summary, which shows the former value of a changed property (the `FIRSTNAME` and `LASTNAME` is changed from J.B. Nimble to Jack B. Quick).
- The contents of the data object itself, in this case, is a customer record.

For more information on data graphs, see [“Working with Data Graphs” on page 2-27](#).

Listing 2-2 Serialized Data Graph

```

<com:datagraph xmlns:com="commonj.sdo">
  <xsd>
    <xs:schema
      targetNamespace="ld:DataServices/CustomerDB/CUSTOMER"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:single="ld:DataServices/CustomerDB/CUSTOMER">
      <xs:include schemaLocation="CUSTOMER.xsd"/>
      <xs:element name="ArrayOfCUSTOMER">
        <xs:complexType>
          <xs:sequence>
            <xs:element ref="single:CUSTOMER" minOccurs="0" maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </xsd>

  <changeSummary>
    <CUSTOMER com:ref="/ArrayOfCUSTOMER/CUSTOMER[1]">
      <FIRST_NAME>J. B.</FIRST_NAME>
      <LAST_NAME>Nimble</LAST_NAME>
    </CUSTOMER></changeSummary>

  <ns0:ArrayOfCUSTOMER xmlns:ns0="ld:DataServices/CustomerDB/CUSTOMER">
    <ns0:CUSTOMER>
      <CUSTOMER_ID>CUSTOMER1</CUSTOMER_ID>
      <FIRST_NAME>Jack B. </FIRST_NAME><LAST_NAME>Quick</LAST_NAME>
      <CUSTOMER_SINCE>2001-10-01</CUSTOMER_SINCE>
      <EMAIL_ADDRESS>Jack@hotmail.com</EMAIL_ADDRESS>
      <TELEPHONE_NUMBER>2145134119</TELEPHONE_NUMBER>
      <SSN>295-13-4119</SSN>
      <BIRTH_DAY>1970-01-01</BIRTH_DAY>
      <DEFAULT_SHIP_METHOD>AIR</DEFAULT_SHIP_METHOD>
      <EMAIL_NOTIFICATION>1</EMAIL_NOTIFICATION>
      <NEWS_LETTTER>0</NEWS_LETTTER>
      <ONLINE_STATEMENT>1</ONLINE_STATEMENT>
      <LOGIN_ID>Jack</LOGIN_ID>
    </ns0:CUSTOMER><ns0:CUSTOMER>
      <CUSTOMER_ID>CUSTOMER2</CUSTOMER_ID>
      <FIRST_NAME>Kevin</FIRST_NAME>
      <LAST_NAME>Smith</LAST_NAME>
      <CUSTOMER_SINCE>2001-10-01</CUSTOMER_SINCE>
      <EMAIL_ADDRESS>JOHN_2@yahoo.com</EMAIL_ADDRESS>
      <TELEPHONE_NUMBER>3607467964</TELEPHONE_NUMBER>
      <SSN>087-46-7964</SSN>
      <BIRTH_DAY>1978-09-21</BIRTH_DAY>
    </ns0:CUSTOMER>
  </ns0:ArrayOfCUSTOMER>
</com:datagraph>

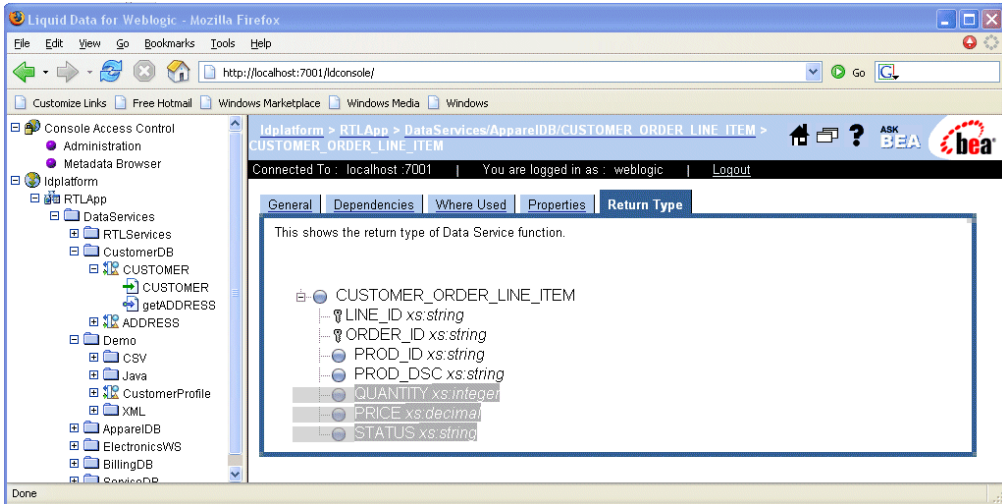
```

```
<DEFAULT_SHIP_METHOD>AIR</DEFAULT_SHIP_METHOD>
<EMAIL_NOTIFICATION>1</EMAIL_NOTIFICATION>
<NEWS_LETTTER>0</NEWS_LETTTER>
<ONLINE_STATEMENT>1</ONLINE_STATEMENT>
<LOGIN_ID>Jerry</LOGIN_ID></ns0:CUSTOMER>
<ns0:CUSTOMER>
.
.
.
</ns0:ArrayOfCUSTOMER>
</com:datagraph>
```

XML Schema-to-Java Type Mapping

DSP client developers can use the Data Services Platform Console to view the XML schema types associated with data services. (See [Figure 2-2](#)) The return type tab indicates, for example, whether an element is a string, int, or complex type.

Figure 2-2 Return Types Display in DSP Console



The [Table 2-3](#) shows XML schema types correspondence to SDO Java types.

Table 2-3 Schema to Java Data Type Mapping

XML Schema Type	SDO Java Type
xs:anyType	Sequence
xs:anySimpleType	String
xs:anyURI	String
xs:base64Binary	byte[]
xs:boolean	boolean
xs:byte	byte
xs:date	java.util.Calendar (Date)
xs:dateTime	java.util.Calendar
xs:decimal	java.math.BigDecimal
xs:double	double
xs:duration	String
xs:ENTITIES	String
xs:ENTITY	String
xs:float	float
xs:gDay	java.util.Calendar
xs:gMonth	java.util.Calendar
xs:gMonthDay	java.util.Calendar
xs:gYear	java.util.Calendar
xs:gYearMonth	java.util.Calendar
xs:hexBinary	byte[]
xs:ID	String
xs:IDREF	String

Table 2-3 Schema to Java Data Type Mapping

XML Schema Type	SDO Java Type
<code>xs:IDREFS</code>	<code>String</code>
<code>xs:int</code>	<code>int</code>
<code>xs:integer</code>	<code>java.math.BigInteger</code>
<code>xs:language</code>	<code>String</code>
<code>xs:long</code>	<code>long</code>
<code>xs:Name</code>	<code>String</code>
<code>xs:NCName</code>	<code>String</code>
<code>xs:negativeInteger</code>	<code>java.math.BigInteger</code>
<code>xs:NMTOKEN</code>	<code>String</code>
<code>xs:NMTOKENS</code>	<code>String</code>
<code>xs:nonNegativeInteger</code>	<code>java.math.BigInteger</code>
<code>xs:nonPositiveInteger</code>	<code>java.math.BigInteger</code>
<code>xs:normalizedString</code>	<code>String</code>
<code>xs:NOTATION</code>	<code>String</code>
<code>xs:positiveInteger</code>	<code>java.math.BigInteger</code>
<code>xs:QName</code>	<code>javax.xml.namespace.QName</code>
<code>xs:short</code>	<code>short</code>
<code>xs:string</code>	<code>String</code>
<code>xs:time</code>	<code>java.util.Calendar</code>
<code>xs:token</code>	<code>String</code>
<code>xs:unsignedByte</code>	<code>short</code>
<code>xs:unsignedInt</code>	<code>long</code>
<code>xs:unsignedLong</code>	<code>java.math.BigInteger</code>

Table 2-3 Schema to Java Data Type Mapping

XML Schema Type	SDO Java Type
xs:unsignedShort	Int
xs:keyref	String

ArrayOf Types

As mentioned elsewhere, data graphs are used to pass data objects between clients and the data service mediator. While a data service function can return multiple elements, a data graph can only have a single root element.

To accommodate functions that return multiple array types, DSP manufactures a root element to serve as the single container for array types. The elements are named with the prefix "ArrayOf". For example, for a function defined to return multiple CUSTOMER elements, the root element is ArrayOfCUSTOMER, as shown in [Listing 2-3](#).

Listing 2-3 Array Root Element

```
<ns0:ArrayOfCUSTOMER xmlns:ns0="ld:DataServices/CustomerDB/CUSTOMER">
  <ns0:CUSTOMER>
    <CUSTOMER_ID>CUSTOMER1</CUSTOMER_ID>
    <FIRST_NAME>Jack B.</FIRST_NAME>
    <LAST_NAME>Quick</LAST_NAME>
    <CUSTOMER_SINCE>2001-10-01</CUSTOMER_SINCE>
    <EMAIL_ADDRESS>Jack@hotmail.com</EMAIL_ADDRESS>
    <TELEPHONE_NUMBER>2145554119</TELEPHONE_NUMBER>
    <SSN>295-00-4119</SSN>
    <BIRTH_DAY>1970-01-01</BIRTH_DAY>
    <DEFAULT_SHIP_METHOD>AIR</DEFAULT_SHIP_METHOD>
    <EMAIL_NOTIFICATION>1</EMAIL_NOTIFICATION>
    <NEWS_LETTTER>0</NEWS_LETTTER>
    <ONLINE_STATEMENT>1</ONLINE_STATEMENT>
    <LOGIN_ID>Jack</LOGIN_ID>
  </ns0:CUSTOMER>
</ns0:ArrayOfCUSTOMER>
```

The array type does not appear in the return type displayed in the DSP Console. However, interface functions for it are generated and included in client packages. If `ArrayOf` types are included in the client package for the data type you want to use, they will include the generated root element. An array is indicated in the console with an asterisk appended to the return type of the function. For example, `CUSTOMER*` indicates a `CUSTOMER` array.

Static versus Dynamic Interfaces

As otherwise mentioned (see [“What is Service Data Objects \(SDO\) Programming?” on page 2-1](#)), SDO specifies both static (strongly typed) and dynamic interfaces for data objects:

- The strongly typed SDO interface is an XML-to-Java API binding that produces functions corresponding to the elements of the data shape returned by the data service, such as `getCUSTOMERNAME()`.
- In the dynamic interface, the element is passed as an argument to the function, such as `get("CUSTOMER")`.

Equivalent typed and dynamic interfaces are provided by the Data Service Mediator API, allowing you to work with data objects in either a dynamic or static model from end-to-end, that is, from data acquisition to client-side manipulation.

[Table 2-4](#) outlines the advantages of each approach.

Table 2-4 Typed versus Untyped Interfaces

Access Mode	Advantages...
typed	<ul style="list-style-type: none">• Easy-to-use interface, resulting in code that is more intuitive and easier read and maintain.• Compile-time type checking.• Enables a pop-up menu in BEA Workshop Source View.• Easier for developers to implement.
dynamic	<ul style="list-style-type: none">• Allows discovery• Code is easier to maintain— changes to the interface do not require the library to be applied.• Allows for a general-purpose coding style.

Note: For more information on the Mediator API, see [Chapter 4, “Accessing Data Services from Java Clients.”](#)

The following sections provide more information on the DSP implementation of both kinds of interfaces.

Static Interface

The static interface is a Java interface generated from a data service's XML schema definition, similar to JAXB or XMLBean static interfaces. The interface files, packaged in a JAR, are typically generated by the DSP implementor using WebLogic Workshop.

Note: For information on generating the client JAR file using WebLogic Workshop, see the [Data Services Developer's Guide](#).

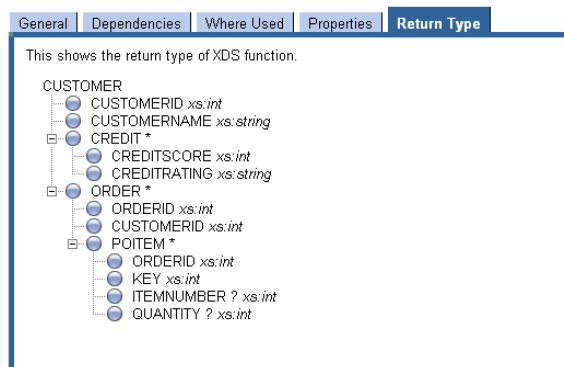
There is another way to generate a JAR file. See [“Setting Up Data Source Aliases for Relational Sources Accessed by DSP,”](#) in [Chapter 8, “Advanced Topics.”](#)

The generated typed interface contains static accessors for all properties of the XML datatype. If the property is complex (such as CREDIT and ORDER in [Figure 2-5](#)), an interface class is generated for the property in the containing package. The interface includes accessors for the properties that make up the complex property.

When developing Data Service Mediator client applications, it is helpful to browse the contents of the generated client package in a development tool (such as Eclipse) to get acquainted with how DSP generates interfaces from data service types. The types of functions that are generated depend on the XML Schema definition for the type. For example, for properties that can have multiple occurrences, as defined in their schema, `getPROPERTYArray()` functions are generated.

Consider the return type shown in the metadata browser illustrated in [Figure 2-5](#).

Figure 2-5 CUSTOMER Return Type



For each complex property—such as the global CUSTOMER element and properties CREDIT, ORDER, and POITEM—separate interfaces are generated with accessors for their contained properties. For each simple attribute, DSP generates set and get methods. For example, the following are generated in the CUSTOMER interface for the CUSTOMERNAME string attribute:

```
getCUSTOMERNAME ()
setCUSTOMERNAME (String)
```

For multiple occurrence properties in the return type (indicated by an asterisk in the Return Type tab of the DSP console), functions for getting the array and manipulating array items are generated. In the XML Schema, a property may occur multiple times if it has a maxOccurs attribute set to unbounded or greater than 1. Attributes cannot occur multiple times.

For example, the following functions are generated for the CREDIT element:

```
getCREDITArray ()
getCREDITArray (int)
addNewCREDIT ()
insertNewCREDIT (int)
removeCREDIT (int)
setCREDITArray (int, CREDIT)
setCREDITArray (CREDIT [])
sizeofCREDITArray ()
```

Because CREDIT is a complex attribute, a separate CREDIT interface with these functions is generated for it in the customer package:

```
getCREDITRATING ()
getCREDITSCORE ()
setCREDITRATING (String)
setCREDITSCORE (int)
```

Similar methods are generated for the ORDER and its POITEM interfaces.

Note that these additional methods are also generated:

- For the root CUSTOMERDocument interface, the following methods are generated:

```
getCUSTOMER ()
setCUSTOMER (CUSTOMER)
addNewCUSTOMER ()
```

- The `get()` methods are derived from the underlying XMLBeans technology. They are methods for getting and setting values as XML types. While `get` methods are not a part of the SDO specification, they can be used if desired. They are generated for elements or attributes whose type is a simple type.
- Factory classes are derived from the `XmlObject` interface that data objects extend. Factory classes exist for creating instances and parsing XML and more. Also, because `DataObjects` are derived from `XmlObject`, they can be cast to `Strings` using the `toString()` method, for example, for printing to output.

Typed Accessor Method Signatures

The following table lists the rules for the typed (or static) method generation.

Signature Format	Description
<code>Type get [PROPERTY] ()</code>	Returns the value of the property. Generated when <code>PROPERTY</code> is an attribute or element with single occurrence.
<code>void set [PROPERTY] (Type newValue)</code>	Sets the value of the property to the <code>newValue</code> . Generated when <code>PROPERTY</code> is an attribute or an element with single occurrence.
<code>boolean isSet [PROPERTY] ()</code>	Determines whether the <code>[PROPERTY]</code> element or attribute exists in the document. Generated for elements and attributes that are optional. In schema, any optional element has a <code>minOccurs</code> attribute set to 0; an optional attribute has a <code>use</code> attribute set to optional.
<code>void unSet [PROPERTY] ()</code>	Removes the <code>[PROPERTY]</code> element or attribute from the document. Generated for elements and attributes that are optional. In schema, and optional element has an <code>minOccurs</code> attribute set to 0; an optional attribute has a <code>use</code> attribute set to optional.
<code>Type[] get [PROPERTY]Array()</code>	For multiple occurrence elements, returns all <code>[PROPERTY]</code> elements.
<code>void set [PROPERTY]Array(Type[] newValue)</code>	Sets all <code>[PROPERTY]</code> elements.
<code>Type get [PROPERTY]Array(int index)</code>	Returns the <code>[PROPERTY]</code> child element at the specified index.
<code>void set [PROPERTY]Array(Type newValue, int index)</code>	Sets the <code>[PROPERTY]</code> child element at the specified index.

Signature Format	Description
<code>int sizeof [PROPERTY]Array ()</code>	Returns the current number of property child elements.
<code>void remove [PROPERTY] (int index)</code>	Removes the <code>[PROPERTY]</code> child element at the specified index.
<code>void insert [PROPERTY] (int index, [PROPERTY]Type newValue)</code>	Inserts the specified <code>[PROPERTY]</code> child element at the specified index.
<code>void add [PROPERTY] ([PROPERTY]Type newValue)</code>	Adds the specified <code>[PROPERTY]</code> to the end of the list of <code>[PROPERTY]</code> child elements.
<code>boolean isSet [PROPERTY]Array (int index)</code>	Determines whether the <code>[PROPERTY]</code> element at the specified index is null.
<code>void unset [PROPERTY]Array (int index)</code>	Unsets the value of <code>[PROPERTY]</code> element at the specified index; that is, sets it to null. Note that after you call <code>unset</code> and then call <code>set</code> , the return value is <i>false</i> .

Dynamic Data Object Interface

The dynamic interface has generic property accessors (such as `set ()` and `get ()`) as well as accessors that get or set data as a specified type, such as `String`, `Date`, `List`, `BigInteger`, and `BigDecimal`. These accessor methods take the following form:

- `set (String propertyName, Object propertyValue)`
- `set (int n, Object propertyValue)`
- `set (int n)`
- `get (commonj.sdo.Property propertyName)`
- `get (int n, Object propertyValue)`
- `get (int n)`
- `set [Type] (String propertyName, Object propertyValue)`
- `get [Type] (String propertyName)`
- `createDataObject (commonj.sdo.Property propertyName)`
- `createDataObject (commonj.sdo.Property propertyName, commonj.sdo.Property propertyName, ...)`
- `createDataObject (String propertyName)`

- `createDataObject(int n)`
- `createDataObject(int n, String propertyName, String propertyValue)`
- `createDataObject(String propertyName, String propertyValue, String propertyName)`
- `delete()`
- `unset(commonj.sdo.Property propertyName)`
- `unset(int n)`
- `unset(String propertyName)`

The generic get methods return an Object type. Also, with the generic set and get methods you can specify an *n*th property to get or set. Type in the above list indicates the specific data type to be set or retrieved; for example, `setBigDecimal` and `getBigDecimal`. This includes the accessors provided for getting and setting properties as primitive types, which include, for example, `setInt()`, `setDate()`, `getString()`, and so on. For a full list, see the SDO Update Javadoc available at:

<http://e-docs.bea.com/liquiddata/docs85/sdoUpdateJavadoc/index.html>

The `propertyName` argument indicates the property whose value you want to get or set, and `propertyValue` is the new value. For example, given a data object `myCustomer`, the following code sets a value by its property name, `LAST_NAME`:

```
myCustomer.set("LAST_NAME", "Nimble");
```

XPath provides considerable flexibility in how you identify nodes in XML-based information. You can identify properties in SDO accessor arguments by SDO extension of XPath expressions. XPath can find nodes by position, relative position, type, and other criteria.

Common SDO Operations and Examples

This section describes common programming tasks involving SDOs. It covers the following:

- [Instantiating and Populating Data Objects](#)
- [Accessing Data Object Properties](#)
- [Submitting Data Object Changes](#)
- [Introspecting a Data Object](#)
- [Adding New Data Objects](#)
- [Deleting Data Objects](#)
- [Working with Data Graphs](#)

Instantiating and Populating Data Objects

The first step in using DSP in a client application is to acquire a data object through a typed or untyped interface.

When you instantiate a data object through either interface, in addition to the data object, you get a data graph to which the object is attached and a handle to the root data object in the data graph. By default, change tracking (logging) on the data graph is enabled, which means that any changes performed on the object values are recorded in the change summary, enabling data source updates and rollbacks.

Note: You can populate a new data object by calling a function in the Mediator API or using a data service control function. The code samples in this chapter generally use the Mediator API. For more information on using the mediator and data service control interfaces, see [Chapter 4, “Accessing Data Services from Java Clients,”](#) and [Chapter 5, “Accessing Data Services from Workshop Applications.”](#)

Static Interface Instantiation

To instantiate a data object using a typed data service, import the packages that contain the generated data service and data type interfaces from the `<app>-ld-client.jar` and instantiate the data service object using the `getInstance()` method. The data type interfaces are contained in a package that has the following prefix to its package path:

```
org.openuri.temp.schemas
```

The following shows a sample of using the typed interface to instantiate a data object:

```
import dataservices.rtlservices.CustomerView;
import retailer.ArrayOfCUSTOMERDocument;

CustomerView custViewDS = CustomerView.getInstance(context, "RTLApp");
ArrayOfCUSTOMERDocument arrCustDoc =
    custViewDS.getCustomerView("CUSTOMER3");
```

Once you have the data service, you can call its public read method, `getCustomerView()`, to get an instance of the root schema element of the data service, `ArrayOfCUSTOMERDocument`.

A *document* type, such as `ArrayOfCUSTOMERDocument`, is a construct for representing a global, top-level element in a data service schema. It lets you access the contents of the entire result returned by a data service function.

Dynamic Interface Instantiation

To instantiate a data object through a dynamic interface, create a `DataService` object using the `DataServiceFactory` class.

The libraries to import from the DSP application client JAR provide the interfaces to the dynamic data services, which can be used as follows:

```
import com.bea.ld.dsmediator.client.DataServiceFactory;

DataService custDS =
    DataServiceFactory.newXmlService(
        context, "RTLApp", "ld:DataServices/CustomerDB/CUSTOMER");

ArrayOfCUSTOMERDocument myCustomer =
    (ArrayOfCUSTOMERDocument) custDS.invoke("CUSTOMER", null);
```

Accessing Data Object Properties

After obtaining a data object, you can access its properties. To access a data object property (similar to instantiating data objects), you can use a typed or untyped interface functions.

Typed Property Access

DSP generates typed (or static) accessors based on the XML type returned by a data service function. As an alternative to untyped data access, you can use typed accessor functions. (Typed functions provide greater ease-of-use than untyped functions.)

For example, the following code example shows how to get the LAST_NAME property of a CUSTOMER instance using typed accessors:

```
ArrayOfCUSTOMER arrCust = myCustomer.getArrayOfCUSTOMER();
CUSTOMER[] customer = arrCust.getCUSTOMERArray();
String lastName = customer[0].getLASTNAME();
```

Untyped Property Access

You can use the untyped (dynamic) API with types that are unknown or not yet deployed at development time. In the untyped interface, the type names are passed as parameters in the untyped accessor call, and the returned object is cast to the type needed. However, in cases where returned type is unbound, you will need to cast the returned object to a List and use an iterator, if necessary. The following is the untyped implementation of the code sample shown in [“Typed Property Access”](#) above, and gets a single CUSTOMER object:

```
ArrayOfCUSTOMER arrCust =
    (ArrayOfCUSTOMER) myCustomer.get("ArrayOfCUSTOMER");
List customerList = (List) arrCust.get("CUSTOMER[1]");
CUSTOMER customer = (CUSTOMER) customerList.get(0);
String lastName = (String) customer.get("LAST_NAME");
```

In cases where you are working with an unbounded type (such as ArrayOfCustomer in the above example), and you want to traverse all the objects in the type, you implement an iterator, such as:

```
List customerList = (List) arrCust.get("CUSTOMER");
Iterator iterator = customerList.iterator();
while ( iterator.hasNext() ){
    if (iterator.next instanceof CUSTOMER){
        String lastName = (String) customer.get("LAST_NAME");
    }
}
```

Note that the string specified in the get method matches the name of the element as specified in the data service. For example, the typed get method for returning the customer’s last name is getLASTNAME() while the untyped method is get("LAST_NAME") rather than get("LASTNAME").

You can identify properties in SDO accessor arguments by element name, such as LAST_NAME. Accessor functions take property identifiers specified as XPath expressions, as follows:

```
customer.get("CUSTOMER_PROFILE[1]/ADDRESS[AddressID='ADDR_10_1']")
```


The example gets the ADDRESS at the specified path with the specified addressID. If elements have identical identifier values, all elements are returned. For example, the ADDRESS element also has a CustomerID (a customer can have more than one address), so all addresses would be returned. (Note that the get method returns a DataObject, so you will need to cast the returned object to the appropriate type. For unbound objects, you will need use a List.)

SDO augments standard XPath notation in how you specify index positions in a path in order to identify an instance in an array. The following SDO call can be used to retrieve the last name of a customer in an array of customers:

```
String lastName = (String) arrCust.get("CUSTOMER.0/LAST_NAME")
```

SDO also supports bracketed-style index notations. The following gets the name of the same department as in the previous example:

```
String lastName = (String) arrCust.get("CUSTOMER[1]/LAST_NAME")
```

Notice that the index for the dot-number notation is zero-based, whereas standard XPath notation is one-based. Therefore, both notation examples retrieve the last name of the first customer in an array of properties. Zero-based indexing is more familiar to Java language programmers and allows zero-based counter values in loop constructs to be used in path expressions without having to add 1.

Note: A “query too complex” exception is raised if required JAR files are not in the JVM’s CLASSPATH when an XPath path expression is executed. If you encounter this error, make sure that the JAR files `xqrl.jar` and `wlxbean.jar` are in the CLASSPATH.

You can get a data object’s containing parent data object by doing the following:

```
myCustomer.get("..")
```

You can get the root containing data object by doing the following:

```
myCustomer.get("/")
```

(This is similar to executing `myCustomer.getDataGraph().getRootObject()`.)

Note: For more information on XPath in SDO, see [“XPath Support in the Untyped SDO API.”](#)

Setting Data Object Properties

You can modify data object property values using `set()` methods. Like `get()` methods, there are both static and dynamic interfaces for setting properties. However, set methods differ from get methods in that they have an additional argument: the new value of the property. For example, to set the last name of a customer using the dynamic API, you would do the following:

```
CUSTOMER customer = (CUSTOMER) myCustomer.get("CUSTOMER");
customer.set("LAST_NAME", "Smith");
```

The example sets the `LAST_NAME` field to a new value “Smith”. By comparison, an operation that sets a value for a typed property using the static API would be:

```
myCustomer.getCUSTOMER().setLASTNAME("Smith");
```

A very important behavioral property of the SDO model is that the back-end data source associated with a modified object (if there is one) is not changed until a `submit()` method is called on the data service bound to the object. Meanwhile, the old value is recorded in a change summary, the change log kept in the data graph that holds the object. For more information on data graphs, see [“Working with Data Graphs” on page 2-27](#).

Adding New Data Objects

You can create new a data object (and have a corresponding changed applied to the data sources associated with its data service) by using an add method and then calling `submit()` on the data service bound to the data object. The lineage of the data (the back-end data sources associated with it) is derived from the data service.

A new data object can be added to a root data object or, more commonly, as a new element in a data object array. In addition, whole new arrays can be added to data objects as well.

The following example demonstrates how to add a data object to an array of objects.

```
CUSTOMERDocument.CUSTOMER newCustomer =
    myCustomer.getArrayOfCUSTOMER().addNewCUSTOMER();
int idNo =
    myCustomer.getArrayOfCUSTOMER().getCUSTOMERArray().length;
newCustomer.setCUSTOMERID("CUSTOMER" + String.valueOf(idNo));
newCustomer.setFIRSTNAME("Clark");
newCustomer.setLASTNAME("Kent");
newCustomer.setCUSTOMERSINCE(java.util.Calendar.getInstance());
newCustomer.setEMAILADDRESS("kent@dailyplanet.com");
newCustomer.setTELEPHONENUMBER("555-555-5555");
newCustomer.setSSN("509-00-3683");
newCustomer.setDEFAULTSHIPMETHOD("Air");
```

There are few points to note about adding data objects:

- Be sure to set any fields in the new object that are required—as specified by the XML schema for the object—before calling `submit()`.
- Foreign key fields in the data object are automatically populated by DSP based on the value of the corresponding foreign key in the container object.
- In a database schema, tables sometimes have auto-generated values as their primary key. When adding an object to such a database, the primary key is generated and returned to the client through the `submit()` call. For more information on primary key computation, see [“Submitting Data Object Changes” on page 2-24](#).

Deleting Data Objects

Just as records can be added to a data source by creating new data objects and submitting the changed data graph, you can similarly remove records by deleting data objects from an existing data graph.

A data object is deleted by removing it from the context of its containing object. When you remove an object from a container, the reference to the item is deleted but not the values. (The values are cleaned up later by Java garbage collection.)

To delete a data object, use the `delete()` method. For example, the following searches a CUSTOMER array for a customer's name and deletes that customer.

```
CUSTOMERDocument.CUSTOMER[] customers =
    myCustomer.getArrayOfCUSTOMER().getCUSTOMERArray();
for (int i=0; i < customers.length; i++){
    if (customers[i].getFIRSTNAME().equals("Clark") &&
        customers[i].getLASTNAME().equals("Kent") )
    {
        customers[i].delete();
        custDS.submit(myCustomer, "ld:DataServices/CustomerDB/CUSTOMER");
    }
}
```

If the deleted object contains any child elements, they are deleted as well. However, note that only the data object on which a delete call has explicitly been performed is tracked in the change summary as having been deleted.

Submitting Data Object Changes

To submit data changes, call the `submit()` method on the data service bound to an object, passing the root changed object and the fully qualified name of the data service bound to the object. Note that the `submit()` method is part of the mediator API.

The untyped interface `submit()` method has the following signature:

```
abstract public void submit (DataObject do, String dataservice)
                        throws java.lang.Exception
```

The `dataservice` argument is the fully qualified name of the data service to which you want the data object to be bound. The function for decomposition for that data service is used to establish the lineage of the data object (the correlation between data object properties and back-end data sources), for example:

```
custDS.submit(myCustomer, "ld:DataServices/CustomerDB/CUSTOMER");
```

In this example, the `dataservice` argument specifies that the `CUSTOMER` data service to which the `myCustomer` data object is to be bound.

The typed version of the `submit()` function only takes the data object as an argument:

```
custDS.submit(myCustomer);
```

After submitting the change, if you want to continue using the object in the client application, it is recommended that you rerun the method used to acquire the data object. This ensures that any side effects of the update operation (at the physical data service level) are incorporated in the data object.

Note that if new objects were added that correspond to relational records in back-end data sources, and if the records have auto-generated primary key fields, the fields are generated in the database source and returned to the client in a property array. The properties include name-value items corresponding to the column name and new auto-generated key value.

Typed Interface Submit

The following example shows how to modify a data object and submit the change using the typed interface:

```
CUSTOMER custDS = CUSTOMER.getInstance(ctx, "RTLApp");
ArrayOfCUSTOMERDocument myCustomer =
    (ArrayOfCUSTOMERDocument) custDS.invoke("CUSTOMER", null);

myCustomer.getArrayOfCUSTOMER().getCUSTOMERArray(0).setLASTNAME("Nimble");

custDS.submit(myCustomer);
```

Note: For more information, see [Chapter 3, “Enabling SDO Data Source Updates.”](#)

Untyped Interface Submit

The following example shows how to modify a data object and submit the change using the untyped interface:

```
CUSTOMER custDS = CUSTOMER.getInstance(ctx, "RTLApp");
ArrayOfCUSTOMERDocument myCustomer =
    (ArrayOfCUSTOMERDocument) custDS.invoke("CUSTOMER", null);
myCustomer.getArrayOfCUSTOMER().getCUSTOMERArray(0).setLASTNAME("Nimble");
custDS.submit(myCustomer, "ld:DataServices/CustomerDB/CUSTOMER");
```

Introspecting a Data Object

When using the untyped interface, it is often necessary to check the properties of a data object once it is acquired. The Type interface gives client applications the ability to discover information on a data object at runtime. The information includes the type of the data object and its list of properties with their types.

The `getType()` method returns the Type interface for a data object. The Type interface gives the client application access to the data object's properties, both elements and attributes.

The following example shows how to get the type of a data object and print a property's value:

```
DataObject o = ...;
Type type = o.getType();
if (type.getName().equals("CUSTOMER") {
    System.out.println(o.getString("CUSTOMERNAME")); }
```

Once you have the type of the object, you can get the list of properties defined for the type and access their values using the `getProperties()` method. The following example iterates through the property list of a data object and prints out information about each property:

```
public void printDataObject(DataObject dataObject, int indent) {
    Type type = dataObject.getType();
    List properties = type.getProperties();
    for (int p=0, size=properties.size(); p < size; p++) {
        if (dataObject.isSet(p)) {
            Property property = (Property) properties.get(p);
            // For many-valued properties, process a list of values
            if (property.isMany()) {
                List values = dataObject.getList(p);
                for (int v=0; count=values.size(); v < count; v++) {
                    printValue(values.get(v), property, indent);
                }
            }
        }
    }
}
```

```
        }
        else {
            // For single-valued properties, print out the value
            printValue(dataObject.get(p), property, indent);
        }
    }
}
}
```

The following table lists other useful methods in the `Type` interface.

Table 2-6 Type Interface Methods

Method	Description
<code>java.lang.Class getInstanceClass()</code>	Returns the Java class that this type represents.
<code>java.lang.String getName()</code>	Returns the name of the type.
<code>java.lang.List getProperties</code>	Returns the list of the properties of this type.
<code>Property getProperty(java.lang.String propertyName)</code>	Returns from all the properties of this type, the one with the specified name. (See Table 2-7 for a list of the methods in the <code>Property</code> class.)
<code>java.lang.String getURI()</code>	Returns the namespace URI of the type.
<code>boolean isInstance(java.lang.Object object)</code>	Returns whether the specified object is an instance of this type.

[Table 2-7](#) lists the methods of the `Property` interface.

Table 2-7 Property Interface Methods

Method	Description
<code>Type getContainingType()</code>	Returns the containing type of this property.
<code>java.lang.Object getDefault()</code>	Returns the default value this property will have in a data object where the property hasn't been set
<code>java.lang.String getName()</code>	Returns the name of the property.
<code>Type getType()</code>	Returns the type of the property. (See Table 2-6 for a list of the methods in the <code>Type</code> class.)

Table 2-7 Property Interface Methods

Method	Description
<code>boolean isContainment()</code>	Returns whether the property is containment; that is, whether it represents by-value composition.
<code>boolean isMany()</code>	Returns whether the property is many-valued.

Working with Data Graphs

A data graph is the container for objects passed between the client application and the DSP mediator. The root object of the data graph is typically a data object corresponding to the root type of the data service return type, such as a Customer object.

In addition to data objects, a data graph contains metadata on the data object and a change summary. A change summary is a record of client-side data changes. The mediator uses the change summary to propagate those changes to the back-end data sources.

You get a data graph automatically with the data object when you invoke a data service function. You can also create a data graph and attach a data object to it on the client side or replace the root object returned from a data service function invocation.

Note: For a print-out of a data graph, see [“Looking at a Data Graph” on page 2-6](#).

There are several useful methods on the data graph interface. You can access the root data object of a data graph using the `getRootObject()` method. To add a root object to an empty data graph, use the `createRootObject()` method. Note that if the data graph already has a root object, it is overwritten. To get the data graph of an existing data object, use the following form:

```
CUSTOMERDocument.getDataGraph()
```

The `getChangeSummary()` method allows you to access the data change log. This is particularly useful when creating data update overrides. These are classes for customizing how data updates are propagated to back-end data sources. For more information on update overrides, see [Chapter 3, “Enabling SDO Data Source Updates.”](#)

XPath Support in the Untyped SDO API

XPath expressions give you a great deal of flexibility in how you locate data objects and attributes in accessors in the untyped interface. For example, you can filter the results of a `get()` function invocation based on data elements and values:

```
company.get("CUSTOMER[1]/POITEMS/ORDER[ORDERID=3546353] ")
```

Note: For more examples of using XPath expressions with SDO, see [“Accessing Data Object Properties” on page 2-19](#).

The SDO implementation extends support of XPath 1.0 as specified by the SDO language specification. However, there are a few points to keep in mind regarding the DSP implementation:

- Expressions with double adjacent slashes ("`//`") are not supported. As specified by XPath 1.0, you can use an empty step in a path to effect a wildcard. For example:

```
("CUSTOMER//ArrayOfPOITEM")
```

In this example, the wildcard matches all purchase order arrays below the CUSTOMER root, which includes either of the following:

```
CUSTOMER/ORDERS/ArrayOfPOITEM
```

```
CUSTOMER/RETURNS/ArrayOfPOITEM
```

Because this notation introduces type ambiguity (types can be either ORDERS or RETURNS), it is not supported by the DSP SDO implementation.

- Attribute notation ("`@`") cannot be used to identify elements. According to the SDO specification, the notation for denoting an attribute "`@`" can be used anywhere in the path because attributes and elements are used interchangeably as properties. However, because DSP implements SDO to XML data binding, the distinction between attributes and elements must be preserved. Attribute notation can only be used to identify what the attributes are in the DSP data type. For example, the ID attribute of the following element:

```
<ORDER ID="3434">
```

is accessed with the following path:

```
ORDER/@ID
```

- SDO identifies an augmentation to the standard XPath notation to specify array index positions. SDO adds the "`.index_from_0`" form of index notation to the standard bracketed, 1-based notation of XPath. The SDO augmentation is 0-based.

For example, the following paths refer to the same element, the first ORDER child node under CUSTOMER:


```
o.get ("CUSTOMER/ORDER [1] ") ;
```

The same expression in SDO's augmented notation:

```
o.get ("CUSTOMER/ORDER.0") ;
```

A 0-based index is more convenient for Java programmers who are accustomed to 0-based counters, and who want to use counter values as index values without adding 1. DSP fully supports both the traditional index notation and the augmented notation. However, note that the augmented form of expression are replaced with the traditional form by the SDO preprocessor. This avoids conflicts with elements named with a dot-number, such as `<myAcct.12>`.

For More Information

This chapter introduces SDO and covers common operations. For detailed information on SDO, use the following references.

- For complete information on the SDO API, see the `commonj.sdo` Javadoc at the following location:

<http://dev2dev.bea.com/technologies/commonj/sdo/index.jsp>

The SDO specification page contains introductory information on SDO:

<http://dev2dev.bea.com/technologies/commonj/sdo/index.jsp>

- The static type interface implementation for DSP SDO is built on top of XMLBeans technology. For more information on XMLBeans, see:

<http://xmlbeans.apache.org/>

- For more information on the version of XPath implemented by DSP, see:

<http://www.w3.org/TR/2004/WD-xquery-20040723/>

Client Programming with Service Data Objects (SDO)

Enabling SDO Data Source Updates

This chapter explains how to implement data services that support data source updates. It includes the following topics:

- [Overview](#)
- [How Data Source Updates Work](#)
- [Update Behavior](#)
- [When to Customize Updates](#)
- [Developing an Update Override Class](#)
- [Update Programming Patterns](#)

Overview

As it does for reading data, BEA AquaLogic Data Services Platform (DSP) gives client applications an easy-to-use, unified interface for updating data. DSP allows client applications to modify, create, and delete data from heterogeneous, distributed data sources as if it were a single entity. The complexity of propagating the changes to the diverse data sources is hidden from the client programmer by the Data Services Platform integration layer.

From the data service implementor's point of view, building a library of update-capable data services is made significantly easier by the DSP update framework. For relational sources, DSP propagates changes to the data source automatically. For other sources, the data service implementor can use the DSP update framework artifacts and APIs to quickly implement update-capable services.

Note: This chapter discusses data service design considerations and programming tasks for enabling updates. For instructions on invoking updates from client applications, see [“Submitting Data Object Changes,” in Chapter 2, “Client Programming with Service Data Objects \(SDO\).”](#)

This chapter covers these areas:

- What the default DSP update behavior is — how it works, its automated behavior as well as its limitations — so that you know when to override the default update process.
- How to implement custom processing classes that extend or override DSP’s default processing behavior.
- Programming patterns used in processing updates. (See [“Update Programming Patterns” on page 3-19.](#))

How Data Source Updates Work

After operating on a data object (for example, by changing, adding, or deleting values), a client application initiates the update process by calling the `submit()` operation on the data service. The data graph, which contains the modified data object and a change summary (the list of old values), is passed to the update mediator service. If the data service that is bound to a changed object is a physical data service, the mediator simply checks for an update override class for the data service and calls the override class if it is present. For relational data sources, the mediator simply propagates the changes to the data source if an update override class is not present.

Decomposition

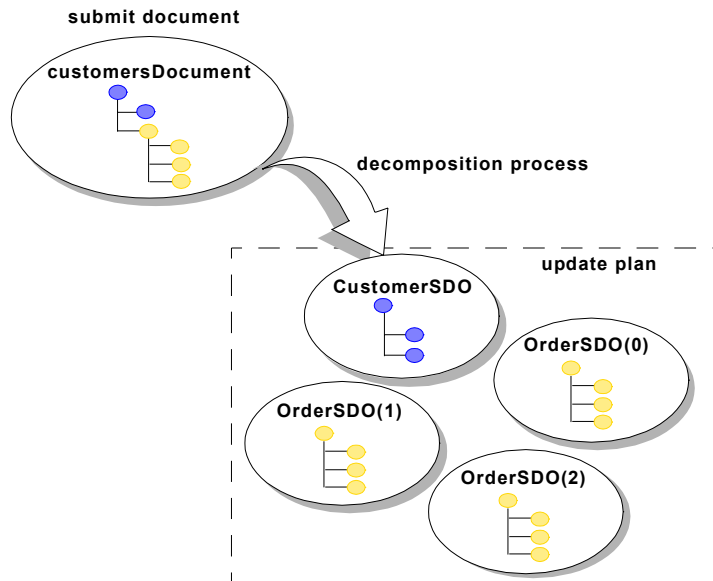
For data objects bound to logical data services, the mediator must first identify where the changed information came from. It does so by analyzing (essentially inverting) the function designated as the *decomposition function* of the data service bound to the data object. The decomposition function enables the mediator to determine the lineage of data, that is, the physical source for each individual element in the data object. This lineage information is expressed in the form of a decomposition map.

Note: If a function for decomposition is not explicitly specified in a data service, the mediator uses the first read function in the data service as the decomposition function. In most cases any of the read functions will do.

If data comes from more than one source, the incoming SDO object is decomposed into its constituent parts. The physical level data objects corresponding to the changed values in the updated data object are instantiated.

For example, a `customersDocument` object that is made up of an updated customer information from a Customer data service and three updated Order objects from an Orders data service would be decomposed into four objects, as illustrated in [Figure 3-1](#).

Figure 3-1 Sample Update Plan



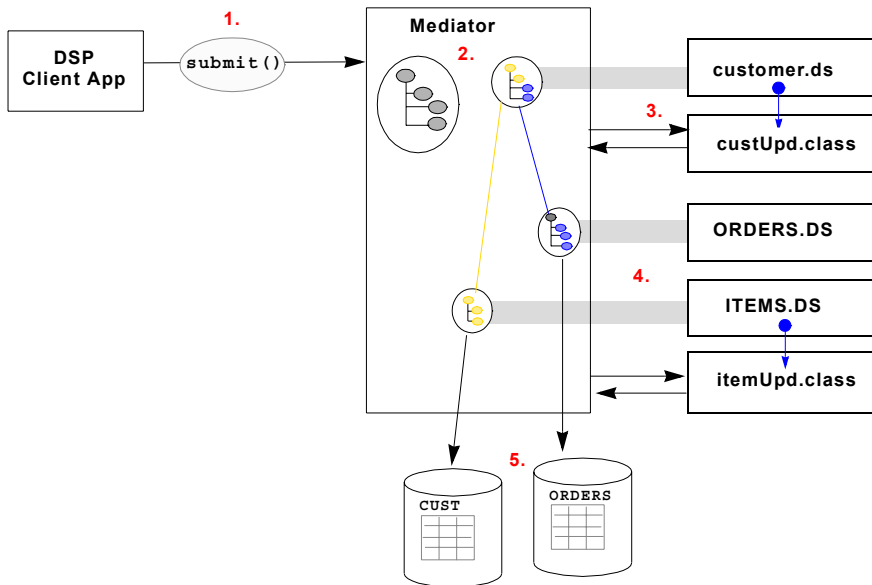
By analyzing the change summary and decomposition map the mediator automatically derives an update plan. The update plan indicates what physical resources will be modified and how. (See [“Accessing the Decomposition Map”](#) on page 3-20 for a description of decomposition maps.) The update plan only has access to the modified objects in a submitted data graph. Unchanged objects do not appear in the plan, and data services for unchanged objects will not be accessed during the update.

Update Processing Sequence

There are a number of steps performed in the updating of a data source. The following are the steps in the update process:

1. The client calls `submit`, passing the data graph with the changed object. The data graph has a change summary detailing the changes to the object.
2. If present, the update override class associated with the top-level data service is instantiated and its `performChange()` function is run. (See [“Update Overrides” on page 3-5](#).) The function can access and modify the update plan and decomposition map, or perform any other custom processing desired. In this case, when finished, the update override class returns control to the mediator. Alternatively, the class could have taken over the remaining processing steps.
3. The mediator determines what data sources need to be changed (the changed object's lineage) and how to change it.
4. Finally, the tree of "SDO objects to update" are applied to the respective data sources.
5. If any are present, update override procedures in the physical data services are called. Note that an update override can exist at each layer of data service composition (and there can be many such levels in the most general case.) Thus, a logical data service of several layers of services would check for update overrides for each component part.

[Figure 3-2](#) illustrates the steps in the update processing sequence used to update a data source. In this example, the mediator performs the final step of the update, known as data change propagation.

Figure 3-2 Update Processing Sequence

Update Overrides

An update override is a Java class associated with a data service. An update override lets you hook custom code into the default update process and is useful for customizing the default behavior, validating data, propagating data changes to a non-relational source, or applying any other processing action desired.

An update plan is generated for any change submitted for a data object bound to a logical data service. However, automatic update propagation will occur only if the data source is relational. If it is not, an update override class must be implemented to propagate changes to the physical data sources. (However, you will get an exception if no update override is available for an updated physical data service that is non-relational.) The override class must implement the `UpdateOverride` interface and contain a `performChange` method, which indicates to the mediator whether or not it should resume the normal course of processing after the update to apply any further changes. (For more information about the update override class, see [“Developing an Update Override Class” on page 3-10.](#))

For logical data services (such as Customers) the update override class is called before the update plan is generated. For physical services, it is called immediately before update propagation. If decomposition yields multiple instances of a changed data object (for example, multiple Orders for a

customer), an update override for the Order data service would be called multiple times, once for each changed object.

Update Behavior

This section provides additional information regarding the behavior of data source updates. It covers these topics:

- **Update Order.** The order in which changes are applied to data objects.
- **Understanding Property Maps.** Property maps are where the mediator saves logical foreign-primary key relationship information.
- **Multi-Level Data Services.** Logical data services can be built upon other logical data services.
- **Transaction Management.** Each SDO submit operates as a transaction.

Update Order

As previously described (see “[How Data Source Updates Work](#)”), the mediator produces an update tree in the decomposition process. The tree contains a data service object for each changed data source instance. When propagating an update, the mediator walks the update plan and submits the indicated changes to the lower-level data service.

The order of objects in the tree and their hierarchical relationships (that is, container-containment relationships) determines the order in which the changes are applied.

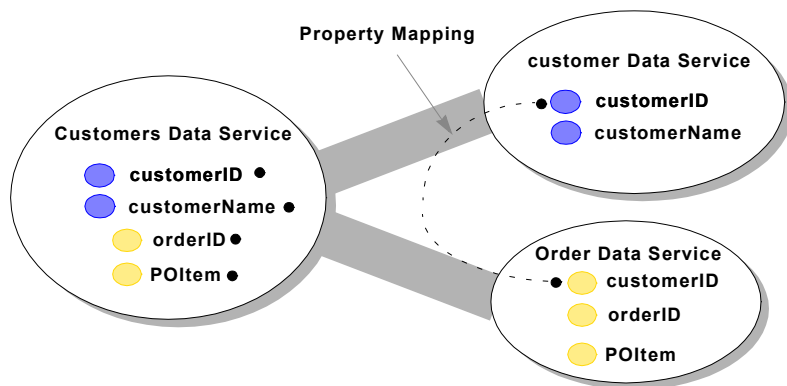
By default, the following order is observed:

- Siblings are processed in the order in which they were encountered in the data object.
- Container objects to be updated are processed before contained objects, unless the container is being deleted. In that case, contained objects are processed first.
- If there is a property map (see the following section, “[Understanding Property Maps](#)”) specified for an object to be updated, the values are mapped from its container before submitting the change. (Changes made to the container SDO during its update, such as primary key computations, are visible in the contained SDO.)

Understanding Property Maps

The mediator saves logical foreign-primary key relationship information in a property map. A property map is used to populate foreign key fields when the parent is new and does not yet have a value for its primary key field. A property map ensures that after the primary key for a parent is generated, the generated value is propagated to the foreign key field of the contained element. In other words, the property map identifies a correspondence between data elements at adjacent levels of the decomposition. [Figure 3-3](#) illustrates the decomposition and property mapping for the decomposition of a Customers data service.

Figure 3-3 Decomposition Map



Multi-Level Data Services

Logical data services can be built upon other logical data services. When functions in the top-level data services are executed, any mid-level logical data services are “folded in” so that the function appears to be written directly against physical data services.

The outcome of the decomposition process differs depending on whether the mid-level data service has a update override class, as follows:

- Without an update override at the mid-level data service, the mediator decomposes the updated object into submit() calls against the underlying physical data services.
- With an update override at the mid-level data service, the mid-level data service there is treated as if it were a physical data service for purposes of decomposition. This means that the mid-level object is instantiated from the top-level SDO so that the mid-level update override is called. The submit() to the mid-level data service is then processed as usual.

Note that any information not projected in the top-level data service will not be able to be resupplied to the intermediate data service.

For example, say that a top-level data service provides a list of orders. It gets the information by calling a function in another logical data service that returns all customer information with their orders. When the orders-only function is called, the view is flattened so that only order information is retrieved from the data source.

Transaction Management

Each SDO `submit()` operates as a transaction. The change log associated with each SDO is unchanged whether or not the `submit()` succeeds. Additional changes may have occurred after the `submit()` call, but those changes are kept separately—the changes are not reflected in the values or change summary of the originally submitted SDO.

If the `submit()` succeeds, the SDO should be re-queried to be sure it matches the current data because side effects of the update may have changed the result of the query. This has the side effect of clearing the change summary as well. If the `submit()` fails, reinvoking `submit()` on the data object would cause an attempt at performing the same updates again because the original data object and change summary are still available. If the SDO `submit()` is not inside a broader containing transaction, the transaction will be committed if the `submit()` succeeds and rolled back if it fails.

SDO Submit Inside a Containing Transaction

All submits perform immediate updates to data sources. If a data object submit occurs within the context of a broader containing transaction, commits or rollbacks of the containing transaction have no effect on the SDO or its change summary, but they will affect any data source updates that participated in the transaction.

When to Customize Updates

You will need to create custom update classes whenever you want to support updates for non-relational sources. For relational sources, you may also want to use custom update classes to apply custom logic to the update process, or if an aspect of the data service design prevents automated updates.

Some examples of when custom updates are required include:

- Specification of your own computation of a primary key value for a data object that is being added as a new record in a database.
- Handling circular dependencies. A circular dependency exists when multiple objects are being modified or added that have mutual dependencies. For example, say a department is being added and one of the department's required fields is a manager. The manager is also being added. However, the new manager must be added as a member of a department. The logic for accommodating the dependency would be: 1) add department with manager set to a temporary value, 2) add the employee manager, 3) reset the department manager to the new employee.

For relational sources, you will need to create custom updates if the XQuery design prevents DSP from being able to perform updates. Factors that might prevent DSP from performing updates include:

- The lineage of the value is ambiguous. For example, the data service decomposition function cannot contain “if-then-else” constructs that provide alternate composition from lower level data services.
- The lineage involves a transformation other than `data()` or `rename`. For example, the following would not be supported by automatic updates:

```
<ACCOUNT> { sum(data($C/ACCOUNT)) }; </ACCOUNT>
```

- Multiple lineages exist for a composed property. The following provides an example of a property with more than one lineage, or data source, for a property:

```
<customerName>{ cat(data($C/FNAME), " ", data($WS/LAST_NAME)) };
</customerName>
```

- Nesting matching logic is not expressed in a where predicate clause. Typically, nested containment is expressed in XQuery using a where dependency clause. If the query does not use a where clause to implement nesting, the foreign-primary key association will be indeterminable.

For instance, if an element of a complex type has values from more than one source (that is, a data object has fields from more than one source), the where predicate does not indicate a 1-*N* cardinality between the two source because the where predicate does not involve a primary key. For example, any *M:N* join like Orders with Payments is not usually a common join, and in this case neither Orders nor Payments would be decomposed.

- The tuple identity is ambiguous. For example, distinct-values or group-by would lead to an arbitrary tuple remaining from a set of duplicate tuples.

- If the same source value instance gets projected in the SDO (or the same physical data source value), and if it is updated in the SDO, it will not be automatically decomposed.
- In some complex types (such as Part and Item values), the Part values may repeat and are therefore not decomposed. For example:
 - You can determine whether a primary key is projected or derivable by knowing the cardinality between two tuples providing the values for the data object. If the predicate between the tuples identifies a primary key on one side (tuple1) but not on the other side (tuple2), values from tuple1 may repeat. Tuple1 values would not be decomposed, but tuple2 values would be decomposed. If the predicate identifies that both tuples primary keys are equal, then values for both tuples would be decomposed.
 - If two Lists of Orders occur in a data object, the predicates used to produce them may or may not make them disjointed. No attempt is made to detect this case. Updates from each instance will be decomposed as separate updates. Depending on the chosen optimistic locking strategy for the data service, the second update may or may not succeed and may overwrite changes made in the first update.
- If the query plan of the decomposition function has a “typematch” node, the decomposition will stop at that point for the SDO.

Developing an Update Override Class

This section describes how to create an update override class. It includes the following topics:

- [UpdateOverride Interface](#)
- [Development Steps](#)
- [Testing Submit Results](#)
- [Understanding Update Override Context](#)
- [Physical Level Update Override Considerations](#)

UpdateOverride Interface

As described in the section [“Update Overrides” on page 3-5](#), a data service needs to specify an update override class to customize the behavior for updates. This class must implement the `UpdateOverride` public interface shown in [Listing 3-1](#).

To implement the interface, your class must implement the `performChange()` method defined in the `UpdateOverride` interface. The method will be executed whenever a submit is issued for objects

bound to the overridden data service. In cases where the submit is for an array of data service data objects, the array is decomposed into a list of singleton `DataService` objects. Some of these objects may have been added, deleted, or modified; therefore, the update override might be executed more than once (that is, once per changed object.)

The `performChange()` takes an on object of type `DataGraph`, which will be passed to it by the mediator. This object is the SDO on which your update override class will operate. The `DataGraph` object contains the data object, the changes to the object, and other artifacts, such as metadata.

The `performChange()` method returns a Boolean value where:

- True signals the mediator to continue with the automated update process.
- False signals the mediator to discontinue the automated update process.

It is a good practice to verify at runtime that the root data object for the data graph being passed in is an instance of the singleton data object bound to the data service for which the update override is written.

Note: The Javadoc for the `UpdateOverride` class is available at:

<http://e-docs.bea.com/liquiddata/docs85/sdoUpdateJavadoc/index.html>

Listing 3-1 UpdateOverride Interface

```
package com.bea.ld.dsmediator.update;

import commonj.sdo.DataGraph;
import commonj.sdo.Property;

public interface UpdateOverride
{
    public boolean performChange(DataGraph sdo)
    {

    }
}
```

Development Steps

To create an update override class, perform the following steps:

1. Add a Java class to the DSP project. (If it is not in the project, it should be in the classpath.) You can put the class anywhere in the application folder; however, for simple projects, it might be most convenient to add the class to the same directory as your data services. For larger projects, you may choose to keep the update classes in their own folder.
2. In the new Java file, implement the `UpdateOverride` interface. For example, the class signature may be:

```
public class OrderDetailUpdate implements UpdateOverride
```

3. Import the following packages into your class, in which you are implementing the `UpdateOverride` class:

```
import com.bea.ld.dsmediator.update.UpdateOverride;  
import commonj.sdo.DataGraph;
```

4. Add a `performChange()` method to the class. This public method takes a `DataGraph` object (containing the modified data object) and returns a Boolean value. For example:

```
public boolean performChange(DataGraph graph)
```

5. In the body of the method, implement your processing logic. You can access the changed object, instantiate other data objects and modify and submit them, or access the mediator context's update plan and decomposition map.

For general examples of the types of activities update overrides may implement, see [“Update Behavior” on page 3-6](#).

6. Associate the class with a data service by referring to it from the data service by placing a `javaUpdate` element in the pragma statement of the data service. For example, the `OrderDetailUpdate` class in step 2 could be referred to from an data service named `ApplOrder` by
`<javaUpdateExit className="RTLServices.OrderDetailUpdate"/>.`

While it will make sense in most cases to have a single update class apply for specific data services, you can have multiple data services use a single update override class.

[Listing 3-2](#) shows a simple update override implementation.

Listing 3-2 Sample Update Override

```

package RTLServices;

import com.bea.ld.dsmediator.update.UpdateOverride;
import commonj.sdo.DataGraph;
import java.math.BigDecimal;
import java.math.BigInteger;
import retailer.ORDERDETAILDocument;
import retailerType.LINEITEMTYPE;
import retailerType.ORDERDETAILTYPE;

public class OrderDetailUpdate implements UpdateOverride
{
    public boolean performChange(DataGraph graph){
        ORDERDETAILDocument orderDocument =
            (ORDERDETAILDocument) graph.getRootObject();

        ORDERDETAILTYPE order =
            orderDocument.getORDERDETAIL().getORDERDETAILArray(0);
        BigDecimal total = new BigDecimal(0);
        LINEITEMTYPE[] items = order.getLINEITEMArray();
        for (int y=0; y < items.length; y++) {
            BigDecimal quantity =
                new BigDecimal(Integer.toString(items[y].getQuantity()));
            total = total.add(quantity.multiply(items[y].getPrice()));
        }
        order.setSubTotal(total);
        order.setSalesTax(
            total.multiply(new BigDecimal(".06")).setScale(2,BigDecimal.ROUND_UP));
        order.setHandlingCharge(new BigDecimal(15));
        order.setTotalOrderAmount(
            order.getSubTotal().add(
                order.getSalesTax().add(order.getHandlingCharge())));
        System.out.println(">>> OrderDetail.ds Exit completed");
        return true;
    }
}

```

In the sample class shown in [Listing 3-2](#), a `OrderDetailUpdate` class implements the `UpdateOverride` class, and, as required by the interface, defines a `performChange()` method. The class illustrates the some basic concepts regarding update override classes:

- The `performChange()` method is the entry point for the class, as called by the mediator. This method can either contain all of the custom update code itself, or it can call external modules to implement the update override code. It can invoke data service functions to populate data objects and submit changes on the object to the mediator. Such changes are automatically added to the current update transaction.
- The method gets the modified data graph as an argument. This is the data graph on which the submit was called.
- The following code gets the first object in the graph, casts it to `ORDERDETAILDocument`, and instantiates a data object with the result:

```
ORDERDETAILDocument orderDocument =  
    (ORDERDETAILDocument) graph.getRootObject();
```

- Objects in the changed object list are accessed through the appropriate `get` call and index value. For example, to get the first such object:

```
ORDERDETAILTYPE order =  
    orderDocument.getORDERDETAIL().getORDERDETAILArray(0)
```

- Finally, notice that the method returns `true`. This tells the mediator that it should continue with its normal course of update processing (with the modified update plan). The value returned from a `performChange()` method in an `UpdateOverride` class must return a Boolean value that will indicate that action the mediator should take after the method completes. The possibilities are:
 - **True.** After control returns from the method, the mediator resumes its normal course of processing. Note that a new update plan is automatically generated so that any new changes against the passed-in SDO made in the update override plan can be accounted for. The new plan combines the previously indicated changes with any new change.
 - **False.** The mediator does not attempt to further apply the changes. The method would return `false`, for example, if it has already propagated all the changes itself. (If you want to handle an error that would require the update to be aborted, your method should throw an exception.) An update override on a physical data service other than a relational one must return `false`.

Testing Submit Results

In the DSP development environment view, the test pane lets you try submitting a change to a data service. Whenever you implement a submit-capable data service, you should similarly test your update results to ensure that changes occur as expected.

You can test submits using the Test View in BEA WebLogic Workshop. For information on testing submits, refer to the *Data Services Developer's Guide*.

While Test View gives you a quick way to test simple update cases in the data services you create, for more substantial testing and troubleshooting you can use an update override class to inspect the decomposition mapping and update plan for the update.

The override class is also the mechanism you can use to extend and override the Mediator's default update processing. You can use it to implement updates for data services that would otherwise not support updates, such as non-relational sources.

See “[Developing an Update Override Class](#)” on page 3-10 for information about override classes.

Understanding Update Override Context

An update override class can programmatically access several update framework artifacts:

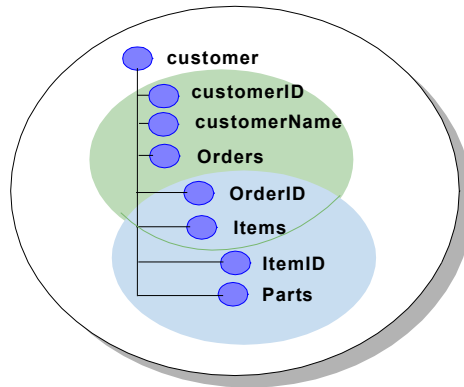
- Modified data object tree
- Update plan
- Decomposition map

The content of the artifacts is determined by the context from which they are accessed:

- The update override class of the top-level data service object has access to the full changed data object tree
- In the update override class for an intermediate or physical data service instantiated as a result of decomposition, only the objects in the change tree that are bound to it are available, along with the contents of the immediate container object. You cannot directly access objects from above the immediate container level in the method.

[Figure 3-4](#) illustrates the context visibility within an update override.

Figure 3-4 Context Visibility in Update Override



The `performChange()` method class can perform gets and sets on the changed SDO, which is passed to the method. Any changes to the SDO values are added to the change summary, just as if the change had occurred in the client application.

Within the `performChange()` method, you can gain access to the decomposition map and the update plan. You can modify the update plan for a particular submit operation, giving you significant control over how updates are applied to a data source. The following are the types of changes that can be effected by modifying this method:

- Check the validity of the new values or change the values in some way.
- Propagate the data changes yourself, for example, by passing modified values to a workflow or web service.
- Use the update plan to directly drive SQL statements.
- Perform audits (log changes).

Although you can access the default decomposition map, you should not modify it. However, you can use access to the decomposition map to understand how decomposition will work, and this could be used to drive your own custom decomposition.

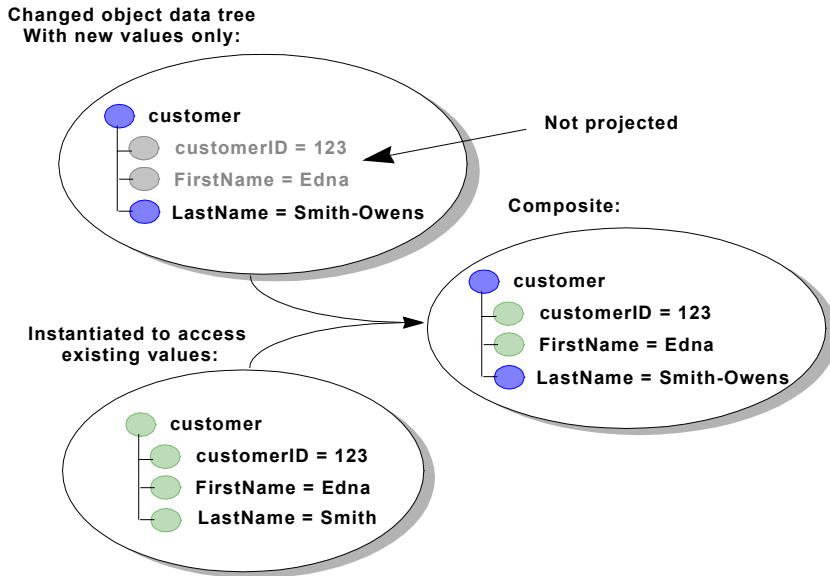
In addition to accessing the decomposition map, you can access the update plan (that is, the tree of changed objects) in the override class. You can modify values in the tree, remove nodes, or rearrange them (to change the order in which they are applied). However, if you modify the update plan, you should execute the plan within the override if you want to keep the changes. As you modify the values in the tree, remove nodes or rearrange them, the update plan will track your changes automatically in the change list.

Physical Level Update Override Considerations

Considerations for implementing update override classes for physical level data services include the following:

- For updated data objects bound to physical data services, further decomposition does not occur. Therefore, requesting a decomposition map or update plan in the override class of an object bound to such a service returns null.
- If the data service is bound to a relational data source, returning true causes the mediator to apply the changes currently indicated by the data object to the database. It does so using the optimistic locking strategy specified for the data service. (Note that if the Data Service is not bound to a relational data source, returning true will cause an exception.)
- For physical data services, the update override can calculate a primary key value or perform other validations or calculations on the submitted data object. If an object bound to a physical data service is being updated in the context of an update to a higher-level data service object (that is, as a product of decomposition), changes in the physical update override (such as the primary key calculation) will be available when the higher-level update plan is applied. Therefore, if a primary key is calculated in the physical update override as part of a data object insert, the key will be available in the logical update plan, so that it can be assigned as a foreign key for the containing object.
- Keep in mind that the modified SDO that is passed to the physical level update override only gets to see those properties of a data object that were projected in the higher level data service. (See [Figure 3-5](#).) To access the unprojected values as well, the update override must re-instantiate the data object.

Figure 3-5 Projected Data Objects



Additional considerations concerning update overrides for relational data services include:

- If `performChange()` returns true, the mediator applies the changes indicated in the data object to the source database using the optimistic locking strategy specified for the data service.
- If an object is inserted with unset property values:
 - If default values for the property are indicated by the data service schema, they are used.
 - If default values are not configured, NULL is used.
- If a primary key was not projected or specified, the automated update signals an error and cancels the update request.

For physical non-relational data services, the following additional considerations apply:

- `performChange()` must provide the implementation for propagating the data change because the mediator does not provide automatic updates for non-relational sources. Using the change summary information in the data object, the method can identify the changes to make and submit them to the data source using any interface or mechanism supported by the source.

- If no update override exists for a non-relational physical data service object for which an update call is made, an error occurs indicating that the change cannot be persisted.

Update Programming Patterns

This section contains code samples that illustrate many of the concepts previously discussed.

- [Override Decomposition and Update](#)
- [Augment Original Data Object Content](#)
- [Accessing the Data Service Mediator Context](#)
- [Accessing the Decomposition Map](#)
- [Customizing an Update Plan](#)
- [Executing an Update Plan](#)
- [Retrieving the Container of the Current Data Object](#)
- [Retrieving and Updating Data Through Other Data Services](#)
- [Setting the Log Level](#)
- [Configuring Optimistic Locking](#)
- [Handling Foreign and Primary Keys](#)

Override Decomposition and Update

In this pattern, the override function takes over the entire decomposition and update processing for the submitted data object. Typical activities include:

- Instantiating lower level data objects and submit them for update.
- Calling a web service passing the appropriate data.
- Using JDBC to execute SQL statements.

In this case, the function would return false to indicate to the mediator not to attempt to proceed with automated decomposition.

Augment Original Data Object Content

The override function inspects or modifies the object values to be changed and returns control to the mediator. If validating values, it can raise a `DataServiceException` to signal errors, and roll back the transaction. The function returns true to have the mediator proceed with update propagation using the objects as changed.

Accessing the Data Service Mediator Context

To get the change plan and decomposition map for an update, you first need to get the data mediator service context.

The context enables you to view the decomposition map, produce an update plan, execute the update plan, and access the container data service instance for the data service object currently being processed.

The following code snippet shows how to get the context:

```
DataServiceMediatorContext context =  
    DataServiceMediatorContext().getInstance();
```

Accessing the Decomposition Map

Once you have the context (see [“Accessing the Data Service Mediator Context”](#)), you can access the decomposition map with the following method:

```
DecompositionMapDocument.DecompositionMap dm =  
    context.getCurrentDecompositionMap();
```

If you want the string form returned, you use the `toString()` method. The returned string will contain the XML of the decomposition map, such as the following:

```
<xml-fragment xmlns:upd="update.dsmediator.ld.bea.com">  
  <Binding>  
    <DSName>ld:DataServices/CUSTOMERS.ds</DSName>  
    <VarName>f1603</VarName>  
  </Binding>  
  <AttributeLineage>  
    <ViewProperty>CUSTOMERID</ViewProperty>  
    <SourceProperty>CUSTOMERID</SourceProperty>  
    <VarName>f1603</VarName>  
  </AttributeLineage>
```

```

<AttributeLineage>
  <ViewProperty>CUSTOMERNAME</ViewProperty>
  <SourceProperty>CUSTOMERNAME</SourceProperty>
  <VarName>f1603</VarName>
</AttributeLineage>
<upd:DecompositionMap>
  <Binding>
    <DSName>ld:DataServices/getCustomerCreditRatingResponse.ds</DSName>
    <VarName>getCustomerCreditRating</VarName>
  </Binding>
  <AttributeLineage>
    <ViewProperty>CREDITSCORE</ViewProperty>
    <SourceProperty>
      getCustomerCreditRatingResult/TotalScore
    </SourceProperty>
    <VarName>getCustomerCreditRating</VarName>
  </AttributeLineage>
  <AttributeLineage>
    <ViewProperty>CREDITRATING</ViewProperty>
    <SourceProperty>
      getCustomerCreditRatingResult/OverAllCreditRating
    </SourceProperty>
    <VarName>getCustomerCreditRating</VarName>
  </AttributeLineage>
</upd:DecompositionMap>
<upd:DecompositionMap>
  <Binding>
    <DSName>ld:DataServices/PO_CUSTOMERS.ds</DSName>
    <VarName>f1738</VarName>
  </Binding>
  <Predicate>
    <LeftVarName>f1738</LeftVarName>
    <LeftProperty>CUSTOMERID</LeftProperty>
    <RightVarName>CUSTOMERID</RightVarName>
    <RightProperty>CUSTOMERID</RightProperty>
  </Predicate>
  <AttributeLineage>
    <ViewProperty>ORDERID</ViewProperty>

```

Enabling SDO Data Source Updates

```
<SourceProperty>ORDERID</SourceProperty>
  <VarName>f1738</VarName>
</AttributeLineage>
<AttributeLineage>
  <ViewProperty>CUSTOMERID</ViewProperty>
  <SourceProperty>CUSTOMERID</SourceProperty>
  <VarName>f1738</VarName>
</AttributeLineage>
<upd:DecompositionMap>
  <Binding>
    <DSName>ld:DataServices/PO_ITEMS.ds</DSName>
    <VarName>f1740</VarName>
  </Binding>
  <Predicate>
    <LeftVarName>f1740</LeftVarName>
    <LeftProperty>ORDERID</LeftProperty>
    <RightVarName>ORDERID</RightVarName>
    <RightProperty>ORDERID</RightProperty>
  </Predicate>
  <AttributeLineage>
    <ViewProperty>ORDERID</ViewProperty>
    <SourceProperty>ORDERID</SourceProperty>
    <VarName>f1740</VarName>
  </AttributeLineage>
  <AttributeLineage>
    <ViewProperty>KEY</ViewProperty>
    <SourceProperty>KEY</SourceProperty>
    <VarName>f1740</VarName>
  </AttributeLineage>
  <AttributeLineage>
    <ViewProperty>ITEMNUMBER</ViewProperty>
    <SourceProperty>ITEMNUMBER</SourceProperty>
    <VarName>f1740</VarName>
  </AttributeLineage>
  <AttributeLineage>
    <ViewProperty>QUANTITY</ViewProperty>
    <SourceProperty>QUANTITY</SourceProperty>
    <VarName>f1740</VarName>
```



```

        </AttributeLineage>
    </upd:DecompositionMap>
</upd:DecompositionMap>
    <ViewName>ld:DataServices/Customer.ds</ViewName>
</xml-fragment>

```

Customizing an Update Plan

After possibly validating or modifying the values in the submitted data object, the function retrieves the update plan by passing in the current data object to the following function:

```
DataServiceMediatorContext.getCurrentUpdatePlan()
```

The update plan can be augmented in several ways, including:

- Additional value set operations could be made on the decomposed data objects.
- The tree of data object instances to be updated could be altered by removing, adding, or rearranging items.
- The modified update plan could be processed by passing it to the function

```
DataServiceMediatorContext.executeUpdatePlan()
```

After executing the update plan, the function should return false so that the mediator does not attempt to apply the update plan.

The update plan lets you modify the values to be updated to the source. It also lets you modify the update order.

You can walk the update plan to view its contents. To walk the plan, you can use a call similar to the method `navigateUpdatePlan()` shown in [Listing 3-3](#) where the method is called from a `performChange` method (see “[UpdateOverride Interface](#)” on [page 3-10](#) for information about this method) and recursively walks the plan.

Listing 3-3 Walking an Update Plan

```

public boolean performChange(DataGraph datagraph) {

    UpdatePlan up = DataServiceMediatorContext.currentContext().
        getCurrentUpdatePlan( datagraph );
    navigateUpdatePlan( up.getDataServiceList() );
    return true;
}

```

Enabling SDO Data Source Updates

```
}

private void navigateUpdatePlan( Collection dsCollection ) {
    DataServiceToUpdate ds2u = null;
    for (Iterator it=dsCollection.iterator();it.hasNext();) {
        ds2u = (DataServiceToUpdate)it.next();

        // print the content of the SDO
        System.out.println (ds2u.getDataGraph() );

        // walk through contained SDO objects
        navigateUpdatePlan (ds2u.getContainedDSToUpdateList() );
    }
}
```

A sample update plan report would look like the following

```
UpdatePlan
  SDOToUpdate
    DSName: ... :PO_CUSTOMERS
    DataGraph:      ns3:PO_CUSTOMERS to be added
      CUSOTMERID   = 01
      ORDERID      = unset
    PropertyMap = null
```

Now consider an example in which a line item is deleted along with the order that contains it. Given the original data, the following listing illustrates an update plan in which item 1001 will be deleted from Order 100, and then the Order is deleted.

```
UpdatePlan
  SDOToUpdate
    DSName:...:PO_CUSTOMERS
    DataGraph:      ns3:PO_CUSTOMERS to be deleted
      CUSTOMERID    = 01
      ORDERID       = 100
    PropertyMap = null
```

```

SDOToUpdate
  DSName:...:PO_ITEMS
  DataGraph:    ns4:PO_ITEMS to be deleted
    ORDERID = 100
    ITEMNUMBER = 1001
  PropertyMap = null

```

In this case, the execution of the update plan is as follows: before deleting the PO_CUSTOMERS, its contained SDOToUpdates are visited and processed. So the PO_ITEMS is deleted first and then the PO_CUSTOMERS.

If the contents of the Update Plan are changed then the new update plan can be executed and the update exit should then return false, signaling that no further automation should occur.

The plan can then be propagated to the data source, as described in [“Executing an Update Plan.”](#)

Executing an Update Plan

After modifying an update plan, you can execute it. Executing the update plan causes the data service mediator to propagate the changes in the update plan to the indicated data sources.

Given a modified update plan named `up`, the following statement executes it:

```
context.executeUpdatePlan(up);
```

Retrieving the Container of the Current Data Object

On a data service that is being processed for an update plan, you can get the container of the SDO being processed. The container must exist in the original changed object tree, as decomposed. If no container exists, null is returned. Consider the following example:

```

String containerDS = context.getContainerDataServiceName();
DataObject container = context.getContainerSDO();

```

In this example, if in the update override class for the Orders data service the you ask to see the container, the Customer data service object for the Order instance being processed would be returned. If that Customer instance was in the update plan, then it would be returned. If it was not in the update plan, then it would be decomposed from CustOrders and returned. The update plan only shows what has been changed. In some cases, the container will not be in the update plan. When the code asks for the container, it will be returned from the update plan if present; otherwise, it will be decomposed from the source SDO.

Retrieving and Updating Data Through Other Data Services

Other data services may be accessed and updated from an update override. The data service mediator API can be used to access data objects, modify and submit them. Alternatively, the modified data objects can be added to the update plan and updated when the update plan is executed. If the data object is added to the update plan, it will be updated within the current context and its container will be accessible inside its Data Service update override.

If the DataService Mediator API is used to perform the update, a new DataService context is established for that submit, just as if it were being executed from the client. This `submit()` acts just like a client submit—changes are not reflected in the data object. Instead, the object must be re-fetched to see the changes made by the submit.

Setting the Log Level

DSP utilizes the underlying WebLogic Server for logging. WebLogic logging is based on the JDK 1.4 logging APIs, which are available in the `java.util.logging` package.

In an update override, you can contribute to the log by acquiring a `DataServiceMediatorContext` instance, and calling the `getLogger()` method on the context, as follows:

```
DataServiceMediatorContext context =  
    DataServiceMediatorContext().getInstance();  
Logger logger = context.getLogger();
```

You can then contribute to the log by issuing the appropriate logger call with a specified level.

The log level implies the severity of the event. When WebLogic Server message catalogs and the `NonCatalogLogger` generate messages, they convert the message severity to a `weblogic.logging.WLLevel` object. A `WLLevel` object can specify any of the following values, from lowest to highest impact:

- **DEBUG.** Debug information, including execution times.
- **INFO.** Normal events with informational value. This will allow you to see SQL that is executed against the underlying databases.
- **WARNING.** Events that may cause errors.
- **ERROR.** Events that cause errors.
- **NOTICE.** Normal but significant events.
- **CRITICAL, ALERT, EMERGENCY.** Significant events that require immediate intervention.

Development_time logging is written to the following location:

```
<bea_home>\user_projects\domains\<domain_name>
```

Given the specified logging level, the mediator logs the following information:

- **Notice or summary** level. For each submit method on a data service from a client the following information is provided:
 - Fully qualified data service name
 - Invocation time
 - Total execution time
 - Invocation by user/group
- **Information or Detail** level. For each submit method invocation on a data service at any level, the following information is provided:
 - For a fully qualified data service name:
 - Invocation time
 - Number of times executed
 - Total execution time
 - For relational sources, per SQL statement type per table:
 - SQL script
 - Total execution time
 - Number of times executed
 - For each update override invoked:
 - Name of Data Service being overridden, which includes: number of times called and total execution time

The following ([Listing 3-4](#)) is a sample of a log entry:

Listing 3-4 Sample Log Entry

```
<Nov 4, 2004 11:50:10 AM PST> <Notice> <LiquidData> <000000> <Demo - begin
client submitted DS: ld:DataServices/Customer.ds>
<Nov 4, 2004 11:50:10 AM PST> <Notice> <LiquidData> <000000> <Demo -
ld:DataServices/Customer.ds number of execution: 1 total execution
time:171>
<Nov 4, 2004 11:50:10 AM PST> <Info> <LiquidData> <000000> <Demo -
ld:DataServices/CUSTOMERS.ds number of execution: 1 total execution time:0>
<Nov 4, 2004 11:50:10 AM PST> <Info> <LiquidData> <000000> <Demo - EXECUTING
SQL: update WEBLOGIC.CUSTOMERS set CUSTOMERNAME=? where CUSTOMERID=? AND
CUSTOMERNAME=? number of execution: 1 total execution time:0>
<Nov 4, 2004 11:50:10 AM PST> <Info> <LiquidData> <000000> <Demo -
ld:DataServices/PO_ITEMS.ds number of execution: 3 total execution
time:121>
<Nov 4, 2004 11:50:10 AM PST> <Info> <LiquidData> <000000> <Demo - EXECUTING
SQL: update WEBLOGIC.PO_ITEMS set ORDERID=? , QUANTITY=? where ITEMNUMBER=?
AND ORDERID=? AND QUANTITY=? AND KEY=? number of execution: 3 total
execution time:91>
<Nov 4, 2004 11:50:10 AM PST> <Notice> <LiquidData> <000000> <Demo - end
clientsubmitted ds: ld:DataServices/Customer.ds Overall execution time:
381>
```

Configuring Optimistic Locking

Concurrency control helps to prevent data conflicts in systems in which multiple clients access the same data source. What if two client read the same information, a customer order total, for example, and attempt to change its value by adding an order? Because the second update does not take into account the first, it can result in invalid data.

DSP uses optimistic locking as its concurrency control policy. With optimistic locking, a database lock is not held for a data record that has been read. Instead, locking reoccurs only when an update is being attempted. At that time, the value of the data when it was read is compared to its current value. If the values differ, the update is not applied and the client is notified.

You can specify which fields are to be compared at the time of the update for each table. Note that primary key column must match, and BLOB and floating types might not be compared. By default, Projected is used. [Table 3-6](#) describes the other options.

Table 3-6 Optimistic Locking Update Policy Options

Optimistic Locking Update Policy	Effect
Projected	<p>Projected is the default setting. It uses a 1-to-1 mapping of elements in the SDO data graph to the data source to verify the “updateability” of the data source.</p> <p>This is the most complete means of verifying that an update can be completed, however if many elements are involved updates will take longer due to the greater number of fields needing to be verified.</p>
Unprojected	Only fields that have changed in your SDO data graph are used to verify the changed status of the data source.
Selected Fields	Selected fields are used to validate the changed status of the data source.

Note: In some instances, DSP may not be able to read data from a database table because another application has locked the table, causing queries issued by DSP to be queued until the application releases the lock. To prevent this, you can set the transaction isolation to read uncommitted in the JDBC connection pool on your WebLogic Server. See ["Setting the Transaction Isolation Level"](#) in the *Administration Guide* for details on how to set the transaction isolation level.

Handling Foreign and Primary Keys

This section describes how relational source updates that affect primary and foreign keys in some way are managed by DSP. For data inserts of autonumber primary keys, the new primary key value is generated and returned to the client. DSP also propagates the effects of changes to a primary or foreign key, as described in the following sections.

Returning Computed Primary Keys

If top-level data objects have been added which have primary keys that are automatically generated by the RDBMS, the values of the primary keys for the inserted tuples will be returned as an array of Java properties (XPath name/value pairs) after a successful update submit. This only applies to the primary keys of the top-level data object. Primary keys for nested data objects that have computed primary keys are not returned.

Returning the top-level primary keys of inserted tuples allows the developer to refetch tuples based on their new primary keys if desired.

For example, given an array of Customer objects with a primary key field CustID into which two customers are inserted, the submit would return an array of two properties with the name being CustID, relative to the Customer type, and the value being the new primary key value for each inserted Customer.

Managing Key Dependencies

DSP manages primary key dependencies when updates are performed. It identifies primary and foreign keys using predicate statements in the decomposition function. For example, if a query joins data records using a value comparison, such as `where customer/id = order/id`, the mediator performs various services given the inferred key/foreign key relationship when updating the data source.

If a predicate dependency exists between two SDOToUpdate instances (data objects in the update plan) and the *container* SDOToUpdate instance is being inserted or modified and the *contained* SDOToUpdate instance is being inserted or modified, then a key pair list is identified that indicates which values from the *container* SDO should be moved to the *contained* SDO after the *container* SDO has been submitted for update. This Key Pair List is based on the set of fields in the *container* SDO and the *contained* SDO that were required to be equal when the current SDO was constructed, and the key pair list will only identify those primary key fields from the predicate fields. The Property Map will only identify container primary key to *container* field mappings. If the full primary key of the container is not identified by the map then no properties are specified to be mapped.

A Key Pair List contains one or more items, identifying the node names in the container and contained objects that are mapped. Mapping means that the value of the property will be propagated from the parent to the child, if the property is an autonumber primary key in the container, which is a new record in the data source after the autonumber has been generated.

Foreign Keys

When computable by SDO submit decomposition, foreign key values will be set to match the parent key values. Foreign keys are computed when an update plan is produced.

Enabling SDO Data Source Updates

Accessing Data Services from Java Clients

This chapter describes how to access data services in Java client applications. It covers the following topics:

- [Overview of the Data Services Platform Mediator API](#)
- [What's in the Data Service Mediator API?](#)
- [How to Use the Mediator API](#)
- [Getting a WebLogic JNDI Context for DSP](#)
- [Using the Static Data Service Interface](#)
- [Using the Dynamic Data Service Interface](#)
- [Using Navigation Functions](#)

Overview of the Data Services Platform Mediator API

The BEA AquaLogic Data Services Platform (DSP) Mediator API gives Java client applications an easy-to-use programmatic interface for accessing information from data services. To use the Mediator API, Java applications simply instantiate a remote data service interface and invoke any public functions on the interface, including read, submit, and navigate functions. When a read function or navigation function is invoked through the Mediator API, the client application gets back information as an SDO data object, also known as a data graph.

As discussed in the [Chapter 3, “Enabling SDO Data Source Updates,”](#) SDO is the client-side data programming model for DSP. The SDO API consists primarily of functions for getting and manipulating data objects and their properties.

Note: For more information on working with Data Objects, see [Chapter 2, “Client Programming with Service Data Objects \(SDO\).”](#)

Like the SDO data programming interfaces, the Mediator API enables client applications to use either a typed or untyped approach to using data services. The untyped interface lets client applications use data services that are either unknown or not created at client development time.

A data object acquired through the untyped mediator API can be cast to a typed object, as long as its structure is compatible with the schema of the type to which it is being cast. In fact, a development pattern that can streamline programming when working with multiple data services is to use untyped APIs for invoking data service functions and subsequently casting the acquired objects to their appropriate types.

This chapter discusses the typed and untyped Mediator interfaces. The Mediator API contains several more advanced features as well. These are discussed in [Chapter 8, “Advanced Topics.”](#) They include:

- **Return value filtering, ordering, and truncating.** Filter and order features in the Mediator API give client applications flexibility for controlling how results are returned by data service function invocation. For more information, see [“Applying Filter Data Service Results,”](#) and [“Ordering and Truncating Data Service Results,”](#) in [Chapter 8, “Advanced Topics.”](#)
- **Streaming DSP information.** Most of the data service interfaces return data as XML objects. When a function is invoked, DSP materializes the resulting XML in memory. Certain queries, such as periodic, administrative-related queries used to inventory a data set, for example, are too large for in-memory materialization to be practical. The streaming interfaces provide for a stream-oriented accessor and file-based materialization of the result object. For more information, see [“Consuming Large Result Sets \(Streaming API and Writing Results To a File\),”](#) in [Chapter 8, “Advanced Topics.”](#)
- **Ad hoc query interface.** The PreparedExpression interface gives DSP clients the ability to invoke ad hoc XQuery expressions against data service results. The ad hoc query interface is similar in spirit to the prepared expression interface of JDBC. Ad hoc queries return data as XMLObjects.

What's in the Data Service Mediator API?

The Mediator API exposes data service functionality to Data Services Platform clients. It contains interfaces and classes for instantiating remote interfaces to the data services and executing functions on the interface. The functions defined for the data service are available in the Mediator API.

The generic, untyped Mediator API classes and interfaces are in the following JAR file:

```
ld-client.jar
```

The Data Service Mediator package is named as follows:

```
com.bea.ld.dsmediator.client
```

The API consists of the classes and interfaces listed in the following table.

Table 4-1 Data Services Platform Mediator API

Interface or Class Name	Description
<code>DataService</code>	Interface for data services that returns data as Data Objects.
<code>PreparedExpression</code>	Interface for preparing and executing ad hoc queries that return information as XML objects. An ad hoc query is one that is defined in the client program, not in the data service.
<code>DataServiceFactory</code>	Factory class for creating Untyped data service interface instances.
<code>StreamingDataService</code>	Interface for data services that returns data as a token stream.
<code>StreamingPreparedExpression</code>	Interface for preparing and executing ad hoc query functions that return information as a stream. An ad hoc query is one that is defined in the client program, not in the data service.

The static mediator interface extends the generic mediator interface, and gives clients a typed approach for instantiating and invoking data service functions. For example, the following class definition represents a typed data service interface:

```
public class dataservices.Customer extends
    com.bea.ld.dsmediator.client.XMLDataServiceBase { ... }
```

The typed data service interface is in the SDO Mediator Client JAR files generated from an DSP project.

The exception class for mediator errors is in the following package:

```
com.bea.ld.dsmediator.client.exception
```

In addition to an exception for general mediator errors (`SDOMediatorException`) there are exceptions for ad hoc queries (`ServerPrepareException`) and streaming access (`StreamingException`).

Exceptions that are generated by the data source (such as `SQLException`) are wrapped in an `SDOException`, and can be accessed at the label `#sdoException.detail`.

Setting the Classpath

To develop mediator client programs, include the preceding JARs in the system CLASSPATH of the development computer.

For example, on Microsoft Windows operating system, the command for setting the class path would be:

```
set CLASSPATH=%CLASSPATH%;Demo-ld-client.jar;  
C:\bea\weblogic81\server\lib\weblogic.jar;  
C:\bea\weblogic81\liquiddata\lib\wlsdo.jar;  
C:\bea\weblogic81\server\lib\xbean.jar;  
C:\bea\weblogic81\liquiddata\lib\ld-client.jar;
```

Note that this assumes that the first item, `Demo-ld-client.jar`, is in the current directory and that the WebLogic home directory is `C:\bea\weblogic81`. If different on your system, modify the path to the locations where these resides on your system.

Also note that when developing your own applications, you will need to substitute the name of `Demo-ld-client.jar` with the name of the JAR file generated from your DSP-enabled application.

Creating the Mediator Client JAR File from the Command Line

Client applications can access the classes representing the typed data service interface using a JAR (Java Archive) file generated from the DSP project. The JAR file needs to be on the client application development machine. The file is named in the form:

```
<AppName>-ld-client.jar
```

There are two ways to generate the JAR file:

- From the Workshop UI, as described in the [Data Services Developer's Guide](#).
- From the command line of the data service development machine.

In most cases, the JAR file would be generated by the Data Services Platform administrator and distributed to data client programmers.

This section describes how to generate the client JAR from the command line. To perform the procedure, first build the EAR file and then the client JAR file.

Build an EAR File

To create a mediator client JAR file, you first need to create an EAR (Enterprise Archive) file from the DSP application. An EAR file is similar to a JAR file — it contains a set of deployable application artifacts.

To build an EAR file, perform the following steps:

1. Open the application by opening the Workshop application file in WebLogic Workshop, typically found in a directory location such as the following:

```
<bea_home>\user_projects\applications\<WLDomain>\<AppName>.work
```

2. Build the application by pressing the F7 key, or select Build Application from the Build menu. This builds the entire application. For your own applications, you will only need to build and deploy the DSP specific artifacts, such as web services, schemas, and so on.
3. Select build Ear from the Build menu. The following file is produced:

```
<bea_home>\user_projects\applications\<WLDomain>\<AppName>.ear
```

4. If it is not already running, start the WebLogic Server. From the Tools menu, select WebLogic Server → Start WebLogic Server.

For more information about building an EAR file, refer to the [Data Services Developer's Guide](#).

Build the Client JAR

To use data service of a DSP project in a client application, you need to generate a client version of the DSP project Java archive. The client version includes wrapper classes that allow the client to call the data service functions through a dynamic API.

Generate the client JAR file from the EAR file you created earlier (see [“Build an EAR File”](#) above) by performing the following steps:

- 1. At a command prompt, navigate to the directory:

```
<bea_home>\weblogic81\liquiddata\lib\sdoclientmediator
```

The directory contains several files, including a build script and a ANT build configuration file. You should not have to modify these files.

- 2. Run the build script using the format:

```
ld_client_gen <LocationOfArchive> [<LocationOfDirectory>] [<LocationForTempDir>]
```

Where *<LocationOfArchive>*, *<LocationOfDirectory>*, and *<LocationForTempDir>* arguments are defined as described in the following table.

Argument	Description
<i><LocationOfArchive></i>	The fully qualified name of the EAR file you generated in step 4 of the Build an EAR File instructions.
<i><LocationOfDirectory></i>	The folder where you want the generated client JAR file to be placed. This is an optional parameter. If not specified, the current directory is used.
<i><LocationForTempDir></i>	The folder where you want the temporary, expanded EAR directory to be placed. This is an optional parameter. If not specified, the current directory is used.

For example:

```
ld_client_gen C:\bea\user_projects\applications\danube\Demo\Demo.ear
C:\test
```

When you run `ld_client_gen`, the following file is produced:

```
C:\test\Demo-ld-client.jar
```


When working with your own applications, "Demo" in the generated name will be replaced by a name derived from your EAR file.

How to Use the Mediator API

To use the Data Service Mediator API to invoke data services, follow these general steps in your application:

- **Step 1:** Import the `com.bea.ld.dsmediator.client` package.
- **Step 2:** Create a JNDI context for the WebLogic Server that hosts the DSP application. (For more information, see [“Getting a WebLogic JNDI Context for DSP”](#))

For complete information, see:

<http://e-docs.bea.com/wls/docs81/javadocs/weblogic/jndi/WLInitialContextFactory.html>

- **Step 3:** Instantiate remote interfaces for the data service. You can use either the typed data service interface or untyped data service interface. The untyped data service interface is generic—the data service name is passed as an argument. For example:

```
DataService ds = DataServiceFactory.newXmlService(
    JndiCntxt, "RTLApp", "ld:DataServices/RTLServices/Customer");
Object params[] = {"CUSTOMER1"};
DataObject myCustomer =
    (DataObject) ds.invoke("getCustomerByCustID", params);
```

Here is the same operation using the typed interface:

```
Customer ds = Customer.getInstance(JndiCntxt, "RTLApp");
```

- **Step 4:** Invoke a function on the data service. The following is the operation using the untyped interface:

```
Object params[] = {"CUSTOMER1"};
ds.invoke("getCustomerByCustID", params);
```

Here is the same operation using the typed interface:

```
CUSTOMERDocument myCust = ds.getCustomerByCustID("CUSTOMER1");
```

When a read or navigate function is invoked, the function returns an SDO data object. For more information, see [Chapter 2, “Client Programming with Service Data Objects \(SDO\).”](#)

Getting a WebLogic JNDI Context for DSP

In general, WebLogic JNDI services allow client applications to access named objects on a WebLogic Server. For DSP, you use JNDI calls to obtain references to remote data services. (Only one JNDI call is needed because the call is created in the context and is passed to the data services factory.) Once you have the server context, you can invoke functions and acquire information from data services.

To get the WebLogic server context, set up the JNDI initial context by specifying the `INITIAL_CONTEXT_FACTORY` and `PROVIDER_URL` environment properties:

- The value of `INITIAL_CONTEXT_FACTORY` should be set to:

```
weblogic.jndi.WLInitialContextFactory
```

- The value of `PROVIDER_URL` should reflect the location (URI) of the WebLogic Server hosting DSP (for example, `t3://localhost:7001`).

A local client (that is, a client that resides on the same computer as the WebLogic Server) may bypass these steps by using the settings in the default context obtained by invoking the empty initial context constructor; that is, by calling `new InitialContext()`.

At this stage, the client may also authenticate itself by passing its security context to the corresponding JNDI environment properties `SECURITY_PRINCIPAL` and `SECURITY_CREDENTIALS`.

The code excerpt below is an example of a remote client obtaining a JNDI initial context using a hashtable.

```
Hashtable h = new Hashtable();  
h.put(Context.INITIAL_CONTEXT_FACTORY,  
"weblogic.jndi.WLInitialContextFactory");  
h.put(Context.PROVIDER_URL, "t3://machinename:7001");  
h.put(Context.SECURITY_PRINCIPAL, <username>);  
h.put(Context.SECURITY_CREDENTIALS, <password>);
```

Be sure to replace the machine name and username/password with values appropriate for your environment.

Using the Static Data Service Interface

Once you have obtained an initial context to the server containing DSP artifacts, you can instantiate a remote interface for a data service. If you know the data service type at development time, you can use the static data service interface, which uses strongly typed data objects. Alternatively, the dynamic interface lets you use data services specified at runtime. The static interface gives you a number of advantages, including type validation and code completion when using development tools, such as Eclipse or your favorite development tool.

To use the static data service interface, you must have the SDO Mediator Client JAR file that was generated from the desired DSP-enabled application. If you do not have the JAR file, contact your administrator to acquire it.

Add the JAR file to your client application's build path and import the data service package into your application. For example, to use a data service named Customer in a DSP project named data services, use the following import statement:

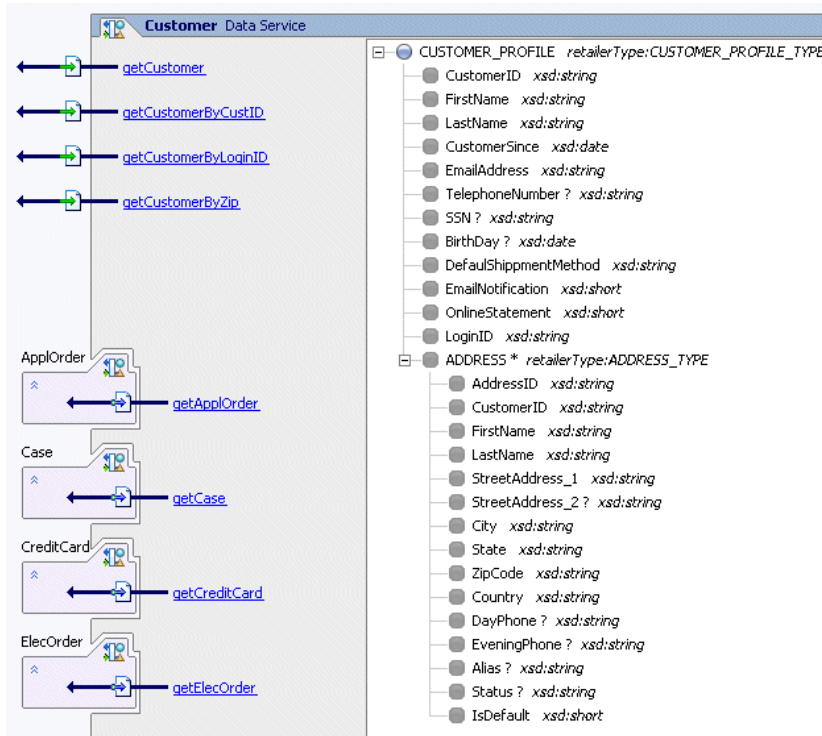
```
import dataservices.Customer;
```

From there, you can instantiate the desired data service interface using the `getInstance()` function. In the function call, pass the following arguments:

- The server context object
- The name of the DSP application that is deployed on the server

Once you have a remote data service instance, you can invoke functions on the data service. Any public function defined on the data service is available in the generated class. For example, consider the public data service functions shown in [Figure 4-2](#).

Figure 4-2 Customer Data Service



The list of methods that are generated for the typed data service are listed below. Notice that methods are created for each function in the data service, such as `getCustomer()` and `getAppOrder()`

```
Customer(Context, String)
getInstance(Context, String)
prepareExpress(Context, String, String)
submit(DataObject)
getCustomer()
getCustomerToFile(String)
getCustomerByCustID(String)
getCustomerByCustIDToFile(String, String)
getCustomerByZip(String)
getCustomerByZipToFile(String, String)
getCase(CUSTOMERPROFILEDocument)
getCaseToFile(CUSTOMERPROFILEDocument, String)
getCreditCard(CUSTOMERPROFILEDocument)
getCreditCardToFile(CUSTOMERPROFILEDocument, String)
getAppOrder(CUSTOMERPROFILEDocument)
```

```

getApplOrderToFile(CUSTOMERPROFILEDocument, String)
getElecOrder(CUSTOMERPROFILEDocument)
getElecOrderToFile(CUSTOMERPROFILEDocument, String)
getCustomerByLoginID(String)
getCustomerByLoginIDToFile(String, String)

```

Several additional functions are generated as well. The `submit()` function is used to save changes to the data objects served by the data service. The *"ToFile"* functions, such as `getCustomerToFile()` and `getAllCustomersToFile()`, are also generated for each function defined in the data service. It allows a client to write results returned from the function call to a file specified as an argument. For more information, see [“Consuming Large Result Sets \(Streaming API and Writing Results To a File\),”](#) in [Chapter 8, “Advanced Topics.”](#)

Another function that is automatically provided is the `prepareExpression()` function. This function is for creating ad hoc queries against the data provided by the data service.

[Listing 4-1](#) shows a small but complete example of using the typed interface.

Listing 4-1 Mediator Client Sample Using the Static Interface to a Data Service

```

import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import dataservices.rtlservices.Customer;
import retailerType.ArrayOfCUSTOMERPROFILEDocument;

public class MyTypedCust
{
    public static void main(String[] args) throws Exception {
        //Get access to Liquid Data
        Hashtable h = new Hashtable();
        h.put(Context.INITIAL_CONTEXT_FACTORY,
            "weblogic.jndi.WLInitialContextFactory");
        h.put(Context.PROVIDER_URL, "t3://localhost:7001");
        h.put(Context.SECURITY_PRINCIPAL, "weblogic");
        Context context = new InitialContext(h);

        // Use the Mediator API
        Customer customerDS = xds.getInstance(context, "RTLApp");
        ArrayOfCUSTOMERPROFILEDocument myCust =
            (ArrayOfCUSTOMERPROFILEDocument)xds.getCustomerByCustID("CUSTOMER2");
        System.out.println(" CUST" + myCustomer);
    }
}

```

Using the Dynamic Data Service Interface

The dynamic data service interface is useful for programming with data services that are unknown or do not exist at development time. It is useful, for example, for developing tools and user interfaces that work across data services.

In the dynamic interface, specific data service names are passed as parameters of the function calls instead of explicitly reflected in the function call names themselves. Like the Mediator API, the SDO API has both static (or strongly typed) and dynamic interfaces for working with data. In most cases, the static Mediator API would be used alongside the equivalent dynamic SDO interfaces. Both have the same use case—working with data when the type is unknown beforehand, for example:

```
DataService ds =
    DataServiceFactory.newDataService(
        context, "RTLApp", "ld:DataServices/RTLServices/Customer");
Object params = {"CUSTOMER2"};
DataObject myCustomer =
    (DataObject) ds.invoke("getCustomerByCustomerID", params);
println(myCustomer.get("Customer/LastName"));
```

A data object returned by the dynamic interface can be down cast to a typed object, as follows:

```
DataService ds =
    DataServiceFactory.newDataService(
        context, "RTLApp", "ld:DataServices/Customer");
Object params = {"CUSTOMER2"};
CUSTOMERDocument myCustomer =
    (CUSTOMERDocument) ds.invoke("getCustomer", params);
println(myCustomer.getCUSTOMER().getCUSTOMERNAME());
```

For an dynamic data service, use the `newDataService()` method of the `DataServiceFactory` class. In the method call, pass the following arguments:

- The server context object.
- The name of the DSP application that is deployed on the server.
- The DSP URI pointing to the location of the data service inside the DSP application.

[Listing 4-2](#) shows a full example.

Listing 4-2 Mediator Client Sample Using the Dynamic Interface to a Data Service

```

import com.bea.ld.dsmediator.client.DataService;
import com.bea.ld.dsmediator.client.DataServiceFactory;
import commonj.sdo.DataObject;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;

public class MyUntypedCust
{
    public static void main(String[] args) throws Exception {

        //Get access to Liquid Data
        Hashtable h = new Hashtable();
        h.put(Context.INITIAL_CONTEXT_FACTORY,
            "weblogic.jndi.WLInitialContextFactory");
        h.put(Context.PROVIDER_URL, "t3://localhost:7001");
        h.put(Context.SECURITY_PRINCIPAL, "weblogic");
        h.put(Context.SECURITY_CREDENTIALS, "weblogic");
        Context context = new InitialContext(h);

        // Use the Mediator API
        DataService ds =
            DataServiceFactory.newXmlService(context, "RTLApp",
                "ld:DataServices/RTLServices/Customer");
        DataObject myCustomer = (DataObject) ds.invoke("getCustomer", null);
        System.out.println(" Customer Information: \n" + myCustomer);
    }
}

```

Using Navigation Functions

A navigation function lets you get data from a related data service. Relationships between data services serve to model a logical connection between them. They also streamline your client programming because you can invoke the relationship function from an instance of the current data service. For example, from a Customer data service you can get a credit card list for a customer instance, as in the following:

```
Customer myCustomer = Customer.getInstance(ctx, "RTLApp");
CUSTOMERPROFILEDocument custProfileDoc =
    CUSTOMERPROFILEDocument.Factory.newInstance();
ArrayOfCREDITCARDDocument cc = myCustomer.getCreditCard(custProfileDoc);
```

A navigation function is called with an object of the calling data service being passed as an argument, as shown in the sample.

To use a navigation function, include the interface for the related data service in the import statements of your application, as shown in [Listing 4-3](#).

Listing 4-3 Calling a Navigation Function Sample

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;

import dataservices.rtl.services.Customer;
import retailerType.CUSTOMERPROFILEDocument;
import retailerType.CUSTOMERPROFILETYPE;
import retailerType.ArrayOfCREDITCARDDocument;

public class CustomerClientNavigation
{
    public static void main(String[] args) throws Exception {

        Hashtable h = new Hashtable();
        h.put(Context.INITIAL_CONTEXT_FACTORY,
            "weblogic.jndi.WLInitialContextFactory");
        h.put(Context.PROVIDER_URL, "t3://localhost:7001");
        h.put(Context.SECURITY_PRINCIPAL, "weblogic");
        h.put(Context.SECURITY_CREDENTIALS, "weblogic");
        Context context = new InitialContext(h);

        Customer myCustomer = Customer.getInstance(context, "RTLApp");

        CUSTOMERPROFILEDocument custProfileDoc =
            CUSTOMERPROFILEDocument.Factory.newInstance();
```



```

CUSTOMERPROFILETYPE newCustProfile =
    custProfileDoc.addNewCUSTOMERPROFILE();

CUSTOMERPROFILETYPE myCustPfl =
    myCustomer.getCustomerByCustID("CUSTOMER0") .
        getArrayOfCUSTOMERPROFILE().
            getCUSTOMERPROFILEArray(0);

newCustProfile.setFirstName(myCustPfl.getFirstName());
newCustProfile.setLastName(myCustPfl.getLastName());
newCustProfile.setADDRESSArray(myCustPfl.getADDRESSArray());
newCustProfile.setCustomerID(myCustPfl.getCustomerID());
newCustProfile.setCustomerSince(myCustPfl.getCustomerSince());
newCustProfile.setEmailAddress(myCustPfl.getEmailAddress());
newCustProfile.setEmailNotification(myCustPfl.getEmailNotification());
newCustProfile.setDefaultShippmentMethod(
    myCustPfl.getDefaultShippmentMethod());

newCustProfile.setOnlineStatement(myCustPfl.getOnlineStatement());
newCustProfile.setLoginID(myCustPfl.getLoginID());
custProfileDoc.setCUSTOMERPROFILE(newCustProfile);

// Navigate to CreditCard data service
ArrayOfCREDITCARDDocument cc = myCustomer.getCreditCard(custProfileDoc);

System.out.println(cc);
}
}

```

Bypassing a Function's Data Cache When Using the Mediator API

Data retrieved by data service functions can be cached. See ["Configuring the Query Results Cache"](#), in the Data Services Platform *Administration Guide*.

However, the following scenario is very common: most of the time you can use cached data because it changes infrequently; however, on occasion, your application must fetch data directly the data source. At the same time, you want to update your cache with the most up-to-date information. A typical example would be to refresh the cache at the beginning of every week or month.

You can accomplish this by passing the attribute `GET_CURRENT_DATA` with your function call.

To bypass the Data Services Platform cache when using the Mediator API, your application will need to signal Liquid Data to retrieve results directly from the data source, rather than from its cache. You need to set the query attribute `GET_CURRENT_DATA` before you call a read function.

[Listing 4-4](#) shows sample Java code that causes the query function to get data from the original source, rather than the cache. As a byproduct, the cache is also refreshed.

Listing 4-4 Cache Bypass Example When Using Mediator API

```
DataService ds = DataServiceFactory.newXmlService(  
getInitialContext(), // Initial Context  
"Evaluation", // Application Name  
"ld:DataServices/CustomManagement/CustomProfile" // Data Service Name  
);  
String params[] = {"CUSTOMER3"};  
QueryAttributes attr = new QueryAttributes();  
attr.enableFeature(QueryAttributes.GET_CURRENT_DATA);  
ds.setQueryAttributes(attr);  
  
CustomerProfileDocument doc = (CustomerProfileDocument)  
ds.invoke("getCustomerProfile",params);
```

Accessing Data Services from Workshop Applications

This chapter describes how you can use a data service control in WebLogic Workshop to develop applications that consume data from data services. The following topics are included:

- [WebLogic Workshop and Data Services Platform](#)
- [Data Service Control \(JCX\) File](#)
- [Creating Data Service Controls](#)
- [Modifying Existing Data Service Controls](#)
- [Using Data Services Platform with NetUI](#)
- [Using the Data Services Platform in Business Process Projects](#)
- [Security Considerations When Using Data Service Controls](#)

WebLogic Workshop and Data Services Platform

Data service controls give WebLogic Workshop applications an easy way to use (or *consume*) data services. When you use a data service control to invoke data services, you get information back as a data object. A data object is a unit of information as defined by the Service Data Objects (SDO) specification. For more information on SDO, see [Chapter 2, “Client Programming with Service Data Objects \(SDO\).”](#)

Note that many of the features available through the Mediator API are also available through data service controls as well. These include:

- Function result filtering
- Ad hoc XQueries
- Result ordering, sorting, and truncating APIs

For more information on these features, see [Chapter 8, “Advanced Topics.”](#)

Data Service Controls

A data service control is a wizard-generated Java file that exposes a data service functions to a Workshop client application. You can add functions to a control from data services deployed on any WebLogic server that is accessible to the client application, whether on the same WebLogic server as the client application or on a remote WebLogic server. In either case, the data service control wizard retrieves all the data service functions on the server that you specify. It then lets you choose the ones to include in your control.

If accessing data services on a remote server, metadata describing information that the service functions return (in the form of XML schema files) is first downloaded from the remote server into the current application. These schema files are placed in a schema project named after the remote application. The directory structure within the project mirrors the directory structure of the remote server. DSP generates interface files for the target schemas associated with the queries and the data service control (.jcx) file.

Use With Page Flow, Web Services, Portals, Business Processes

Like any WebLogic Workshop control, you can use a data service control in applications such as web services, page flows, and WebLogic integration business processes. After applying the control to a client application, you can use the data returned from queries in the control in your application.

This chapter describes in detail how to use the control in a page flow-based web application. The steps for using it in Portals and other WebLogic Workshop Projects are similar.

Data Service Control (JCX) File

When you create a Data Services Platform control, WebLogic Workshop generates a Java Control Extension (.jcx) file. This file contains the methods included the control was created and a commented method that, when uncommented, allows you to pass any XQuery statement to the server (called *ad hoc queries*).

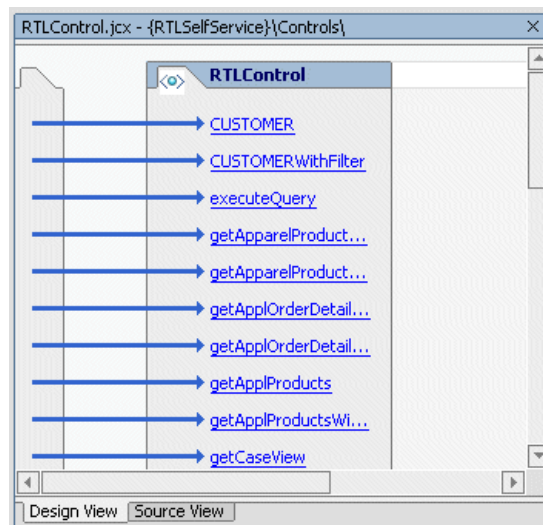
This section describes the data service control in detail and includes the following sections:

- [Design View](#)
- [Source View](#)
- [Running Ad Hoc Queries Through a Data Service Control](#)

Design View

The Design View tab of a data service control shows a graphical view of the data service methods that were selected for inclusion in the control.

Figure 5-1 Design View of a JCX File



Using the right-click menu, you can add or modify a control method (for example, by changing the data service function accessed by a method). The right-click menu is context sensitive—it displays different items if the mouse cursor is over a method or elsewhere in the control portion of the design pane.

Source View

The Source View tab shows the source code of the Java Control Extension (.jcx) file. It includes as comments the name of the data service function associated with each method. For update functions, the data service bound to the update is the data service specified by the locator attribute. (For example, locator="c:/DSP/DataServices/RTLServices/ApplOrderDetailView.ds")

The signature for the method shows its return type. The return type for a read method is an SDO object corresponding to the schema type of the data service that contains the referenced function. The SDO classes corresponding to the data services used in a data service control reside in the Libraries folder of the project. An interface is generated for each data service. The folder also contains a copy of the schema files associated with the functions in the JCX file.

The Java Control Extension instance is a generated file. The only time you should need to edit the source code is if you want to add a method to run an ad hoc query, as described in [“Running Ad Hoc Queries Through a Data Service Control” on page 5-7](#).

The the following example ([Listing 5-1](#)) shows a generated data service control (.jcx) file. It shows the package declaration, import statements, and data service URI used with the queries.

Listing 5-1 Java Control Extension Sample

```
package Controls;

import weblogic.jws.control.*;
import com.bea.ld.control.LDControl;
import com.bea.ld.filter.FilterXQuery;
import com.bea.ld.QueryAttributes;

/**
 * @jc:LiquidData application="RTLApp"
 urlKey="RTLApp.RTLSelfService.Controls.RTLControl"
 */
public interface RTLControl extends LDControl, com.bea.control.ControlExtension
{

    /* Generated methods corresponding to stored queries. */
    /**
```

```

*
* @jc:XDS locator="ld:DataServices/RTLServices/AplOrderDetailView.ds"
functionName="submitAplOrderDetailView"
*/
java.util.Properties[]
submitAplOrderDetailView(retailer.ORDERDETAILDocument rootDataObject)
throws Exception;

/**
*
* @jc:XDS locator="ld:DataServices/RTLServices/ProfileView.ds"
functionName="submitArrayOfProfileView"
*/
java.util.Properties[]
submitArrayOfProfileView(retailer.ArrayOfPROFILEDocument rootDataObject) throws
Exception;

/**
*
locator="ld:DataServices/RTLServices/ElecOrderDetailView.ds"
functionName="submitElecOrderDetailView"
*/
java.util.Properties[]
submitElecOrderDetailView(retailer.ORDERDETAILDocument rootDataObject) throws
Exception;

/**
*
* @jc:XDS functionURI="ld:DataServices/CustomerDB/CUSTOMER"
functionName="CUSTOMER" schemaURI="ld:DataServices/CustomerDB/CUSTOMER"
schemaRootElement="ArrayOfCUSTOMER"
*/
dataServices.customerDB.customer.ArrayOfCUSTOMERDocument CUSTOMER();

/**
*
* @jc:XDS functionURI="ld:DataServices/CustomerDB/CUSTOMER"
functionName="CUSTOMER" schemaURI="ld:DataServices/CustomerDB/CUSTOMER"
schemaRootElement="ArrayOfCUSTOMER"
*/
dataServices.customerDB.customer.ArrayOfCUSTOMERDocument
CUSTOMERWithFilter(FilterXQuery filter);

/**
*
* @jc:XDS functionURI="ld:DataServices/RTLServices/AplOrderDetailView"
functionName="getAplOrderDetailView"
*/
retailer.ORDERDETAILDocument getAplOrderDetailView(java.lang.String p0);

```

Accessing Data Services from Workshop Applications

```
.
.
.
/**
 *
 * @jc:XDS functionURI="ld:DataServices/RTLServices/ProfileView"
functionName="getProfileView" schemaURI="urn:retailer"
schemaRootElement="ArrayOfPROFILE"
 */
retailer.ArrayOfPROFILEDocument getProfileViewWithFilter(java.lang.String
p0, FilterXQuery filter);

/**
 * Default method to execute an ad hoc query.
 * This method can be customized to have a different method name (e.g.
 * runMyQuery), or to return an SDO generated class (e.g. Customer),
 * or to return the DataObject class, or to have one or both of the following
 * two extra parameters: com.bea.ld.ExternalVariables and
 * com.bea.ld.QueryAttributes
 * e.g. commonj.sdo.DataObject executeQuery(String xquery,
 *      ExternalVariables params);
 * e.g. commonj.sdo.DataObject executeQuery(String xquery,
 *      QueryAttributes attrs);
 * e.g. commonj.sdo.DataObject executeQuery(String xquery,
 *      ExternalVariables params, QueryAttributes attrs);
 */
com.bea.xml.XmlObject executeQuery(String query);
}
```

Running Ad Hoc Queries Through a Data Service Control

A client application can issue ad hoc queries against data service functions. You can use ad hoc queries when you need to change the way a data service function returns data. Ad hoc queries are most often used to process data returned by data services deployed on a WebLogic Server. Ad hoc queries are especially useful when it is not convenient or feasible to add functions to an existing data service.

A data service control generated from a wizard has a commented ad hoc query method that can serve as a starting point for generating an ad hoc query. To generate the ad hoc query, follow these steps:

1. If you do not already have a data service control (JCX) file, generate one using the data service control wizard.
2. Add the following lines of code in the JCX file:

```
com.bea.xml.XmlObject executeQuery(String query);
```

(You can replace the function name in with your own to impart meaning to the ad hoc query function. The ad hoc query returns an `XmlObject` by default, but you can return a typed SDO or typed `XMLBean` class that matches the return type XML for the ad hoc query. You can also optionally supply `ExternalVariables` or `QueryAttributes` (or both) to an ad hoc query.)

When invoking this ad hoc query function from a data service control, the caller needs to pass the query string (and the optional `ExternalVariables` binding and `QueryAttributes` if desired). For example, a ad-hoc query signature in a data service control will look like the following:

```
public interface MyLDControl extends LDControl,
                                   com.bea.control.ControlExtension
{
    ldcProduucerDataServices.address.ArrayOfADDRESSDocument
                                   adHocAddressQuery(String xquery);
}
```

The code to call this data service control (from a `WebService.jws` file, for example) would be:

```
/** @common:control */
public ldccontrol.MyLDControl myldcontrol;

/** @common:operation */
public ldcProduucerDataServices.address.ArrayOfADDRESSDocument
                                   adHocAddressQuery()
{
    String adhocQuery =
    "declare namespace f1 = \"ld:ldc_produucerDataServices/ADDRESS\";\n" +
    "declare namespace ns0=\"ld:ldc_produucerDataServices/ADDRESS\";\n" +
    "<ns0:ArrayOfADDRESS>\n"+"{for $i in f1:ADDRESS()\n" +
    "where $i/STATE = \"TX\"\n"+" return $i}\n" +
    "</ns0:ArrayOfADDRESS>\n";
```

```
        return myldcontrol.adHocAddressQuery(adhocQuery);  
    }
```

Creating Data Service Controls

This section describes the steps for creating a data service control and using it in a web project. The general steps to create a data service control are:

[Step 1: Create a Project in an Application](#)

[Step 2: Start WebLogic Server, If Not Already Running](#)

[Step 3: Create a Folder in a Project](#)

[Step 4: Create the Data Service Control](#)

[Step 5: Enter Connection Information to the WebLogic Server](#)

[Step 6: Select Data Service Functions to Add to Your Control](#)

The following sections describe each of these steps in detail.

Step 1: Create a Project in an Application

Before you can create a data service control in WebLogic Workshop, you must create an application and a project in that application. You can create a data service control in most types of Workshop projects. The most common projects in which you create data service controls are:

- Web Projects
- Web Service Projects
- Portal Web Projects
- Process Web Projects

Step 2: Start WebLogic Server, If Not Already Running

Make sure that the WebLogic Server that host the DSP-enabled application is running. WebLogic Server can be running locally (on the same domain as WebLogic Workshop) or remotely (on a different domain from Workshop).

Step 3: Create a Folder in a Project

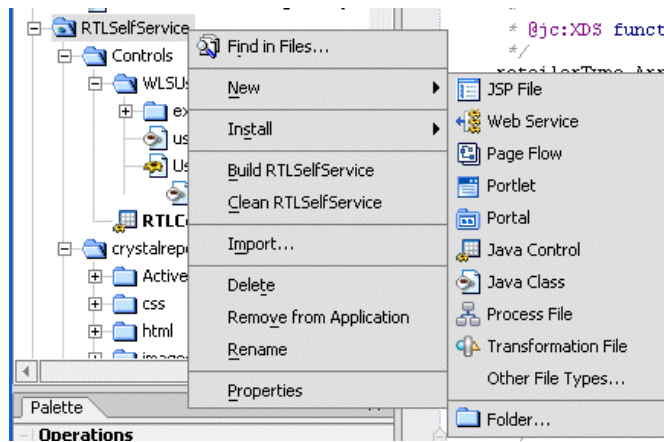
Create a folder in the project to hold the data service control by selecting a folder and right-clicking on that folder. You can also create other controls (database controls, for example) in the same folder

as needed. Workshop controls cannot be created at the top level of a project directory structure. Instead, they must be created in a folder. When you create the folder, enter a name that makes sense for your application.

Step 4: Create the Data Service Control

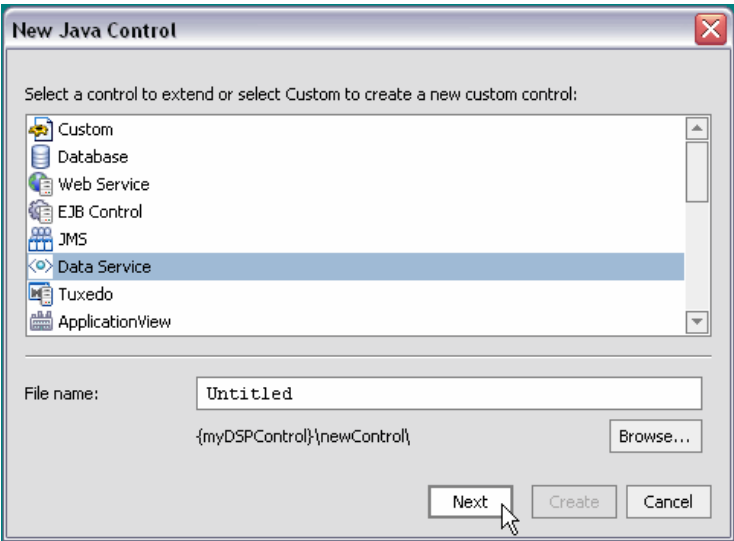
To create a data service control, start the Java Control Wizard by right-clicking on the new folder in your project and choosing **New** → **Java Control** as shown in [Figure 5-2](#). (You can also create a control using the **File** → **New** → **Java Control** menu item.)

Figure 5-2 Create a New Data Service Control



Next, select **Data Services Platform** from the **New Java Control** dialog as shown in [Figure 5-3](#). Enter a filename for the control (`.jcx`) file and click **Next**.

Figure 5-3 New Java Control Dialog



Step 5: Enter Connection Information to the WebLogic Server

The New Java Control - DSP dialog (Figure 5-4) allows you to enter connection information for the WebLogic Server that hosts your Data Services Platform application or project. If the server is local, a data service control uses the connection information stored in the application properties. (To view these settings, access the Tools → Application Properties menu item in Workshop.)

If the server is remote, choose the Remote option and fill in the appropriate server URL, user name, and password.

Note: You can specify a different username and password with which to connect to a local machine in the data service control Wizard as well. To do this, click the Remote button and enter the connection information (with a different username and password) for your local machine. The security credentials specified through the Application Properties or through the data service control wizard are only used for creating the JCX file, not for testing queries through the control. For more details, see [“Security Considerations When Using Data Service Controls” on page 5-30](#).

When the information is correct, click Create to go to the next step.

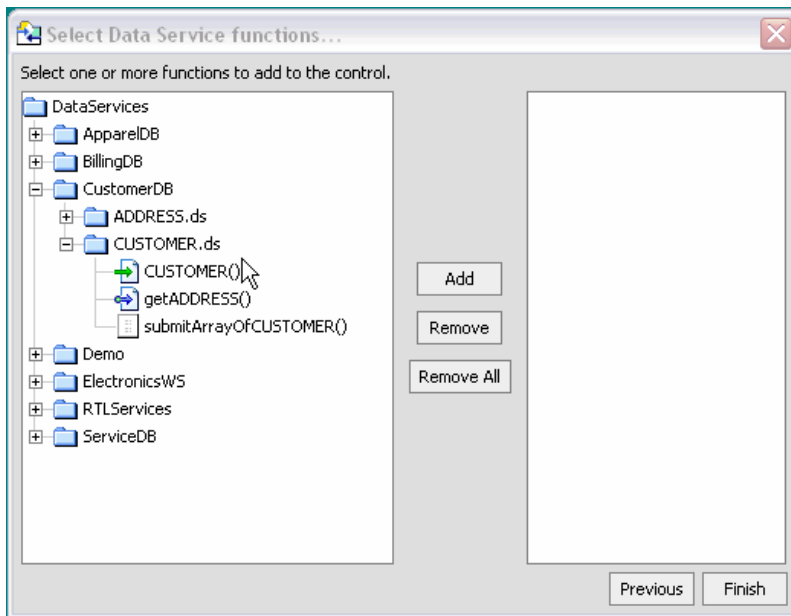
Figure 5-4 Data Service Control Wizard: Connection Information

The screenshot shows the 'New Java Control - Data Service' dialog box. It has a title bar with a close button. The dialog is divided into two steps. Step 1 is 'New JCX name:' with a text field containing 'Untitled'. Step 2 is 'Data Services Application' with two radio buttons: 'Current' (selected) and 'Other'. Below this, there are two radio buttons for 'Server Domain': 'Local' (selected) and 'Remote'. Under 'Local', there are four text fields: 'Server URL: (t3://localhost:7001)', 'User name: (installedadministrator)', 'Password:', and 'Application name:'. The 'Application name:' field has a 'Browse' button next to it. At the bottom, there are four buttons: 'Previous', 'Next', 'Create', and 'Cancel'. A mouse cursor is pointing at the 'Create' button.

Step 6: Select Data Service Functions to Add to Your Control

In the Select Data Services Platform Queries screen, select the data service functions you want to use in your application from the left pane and click Add. When done, click Finish. At that point, the data service control JCX file is generated, with a call for each selected function.

Figure 5-5 Control Wizard: Select Data Service Functions Dialog Box



The `LiquidDataControl.jar` file is copied into the `Libraries` directory of your application when you create your data service control.

The control appears with the functions you chose. Also, *WithFilter* functions are added for each function, such as `getCustomerWithFilter()`. A filter function lets you further filter the results normally returned by a function. For more information, see [“Applying Filter Data Service Results”](#) in [Chapter 8, “Advanced Topics.”](#)

After you have added all the queries you need in the wizard, click Finish. Workshop generates the JCX file for your Data Services Platform control. Each method in the file returns an SDO type corresponding to the appropriate (or corresponding) data service schema. The SDO classes are stored in the `Libraries` directory of the Workshop Application.

Note: In the unlikely case that you get a timeout error when attempting to create a data service control, you will may see a message related to the compiler being unable to find the XMLBean class for a particular schema element.

You can change the timeout value — by default that value is set at 5000 (5 seconds) — by adding a directive in the WebLogic Workshop configuration file:

```
<beahome>/weblogic81/workshop/workshop.cfg
```

For example to change the setting to 10000 add the following directive to the file:

```
-Dcom.bea.ld.control.notification.timeout=10000
```

Modifying Existing Data Service Controls

This section describes the ways you can modify an existing data service control. When you edit a control, the SDO classes that are available to the control are recompiled, which means that any changes to data service are incorporated to the controls at that point as well.

This section contains the following procedures:

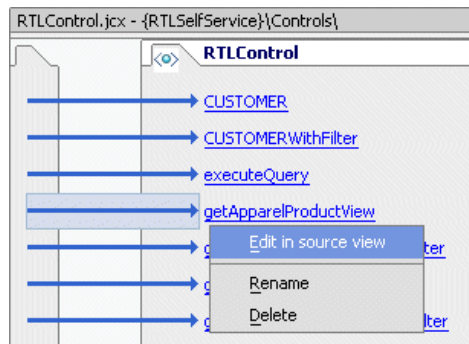
- [Changing a Method Used by a Control](#)
- [Adding a New Method to a Control](#)
- [Updating an Existing Control if Schemas Change](#)

Changing a Method Used by a Control

To change a data service function in a data service control, perform the following steps:

1. In WebLogic Workshop, open the Design View for a data service control (.jcx) file.
2. Select the method you want to change, right-click, and select Edit in source view to bring up the source editor. (See [Figure 5-6](#).)

Figure 5-6 Changing a Function in a Data Service Control



3. In the source view, change the comment for the function. Change the functionName value to the new function you want to use. If necessary, change the functionURI value as well. This should be the path to the data service that contains the function.
4. Change the return type, parameters, and name of the function.

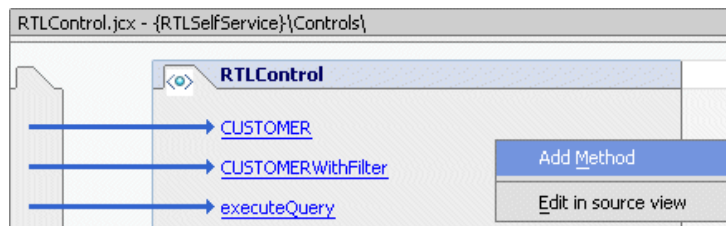
When you save your changes, the SDO classes based on the control are automatically recompiled.

Adding a New Method to a Control

To add a new method to an existing data service control, perform the following steps:

1. In Workshop, open an existing control in Design View.
2. In the control Design View, move your mouse inside the box showing the control methods, right-click, then select Add Method as shown in [Figure 5-7](#).

Figure 5-7 Adding a Method to a Control



3. Enter a name for the new method.
4. Right-click the new method, and select Edit in Source View to bring up the source editor.
5. In the Source View, add a comment for the function. Change the functionName value to the new function you want to use. If necessary, change the functionURI value as well. This should be the path to the data service that contains the function.
6. Change the return type, parameters, and name of the function.

Updating an Existing Control if Schemas Change

If any of the schemas corresponding to any methods in a data service control change, you must build the DSP data service folders to regenerate the SDO classes for the changed schemas. If the changes result in a different return type for any of the functions, you must also modify the function in the control.

When you edit the control, its SDO classes are automatically regenerated.

Using Data Services Platform with NetUI

The WebLogic NetUI tag library allows you to rapidly assemble JSP-based applications that display data returned by Data Services Platform. The following sections list the basic steps for using NetUI to display results from a data service control:

- [Generating a Page Flow From a Control](#)
- [Adding a Data Service Control to an Existing Page Flow](#)
- [Adding Service Data Objects \(SDO\) Variables to the Page Flow](#)
- [Displaying Array Values in a Table or List](#)

Generating a Page Flow From a Control

When you ask Workshop to generate a page flow, Workshop creates the page flow, a start page (`index.jsp`), and a JSP file and action for each method you specify in the Page Flow wizard.

To Generate a Page Flow From a Data Service Control

Perform the following steps to generate a page flow from a Data Services Platform control.

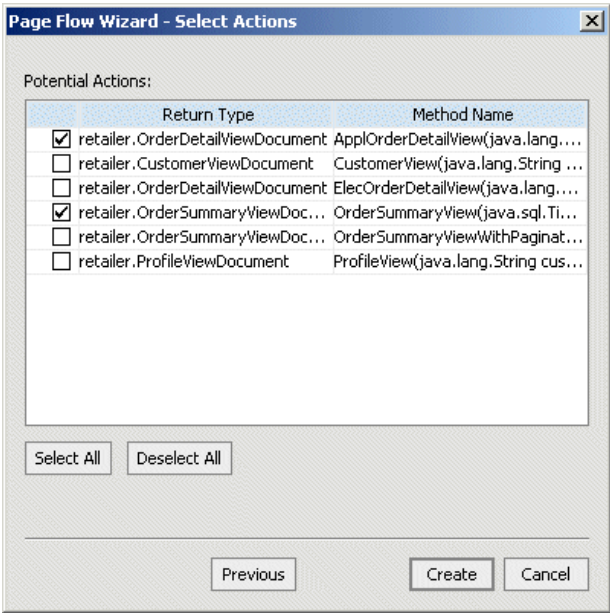
1. Select a Data Services Platform control JCX file from the application file browser, right-click, and select Generate Page Flow.
2. In the Page Flow Wizard (see [Figure 5-8](#)), enter a name for your Page Flow and click Next.

Figure 5-8 Enter a Name for the Page Flow

The screenshot shows a Windows-style dialog box titled "Page Flow Wizard - Page Flow Name". It has a standard close button (X) in the top right corner. The dialog is divided into two sections by horizontal lines. The first section, "Name And Location", contains three text input fields: "Page Flow Name:" with the value "myPageFlow", "Location:" with the value "{myTestWeb}/myPageFlow/" and a "Browse..." button to its right, and "Controller Name:" with the value "myPageFlowController.jspf". The second section, "Page Flow Nesting", contains a paragraph of text: "Nested page flows are used to gather and return information to a calling page flow." Below this text is a checkbox labeled "Make this a nested page flow", which is currently unchecked. At the bottom of the dialog, there are three buttons: "Next", "Create", and "Cancel".

3. On the Page Flow Wizard - Select Actions dialog, check the methods for which you want a new page created. The wizard has a check box for each method in the control. (See [Figure 5-9](#).)

Figure 5-9 Choose Data Services Platform Methods for the Page Flow



4. Click Create.

Workshop generates the .jpf Java Page Flow file, a start page (index.jsp), and a JSP file for each method you specify in the Page Flow wizard.

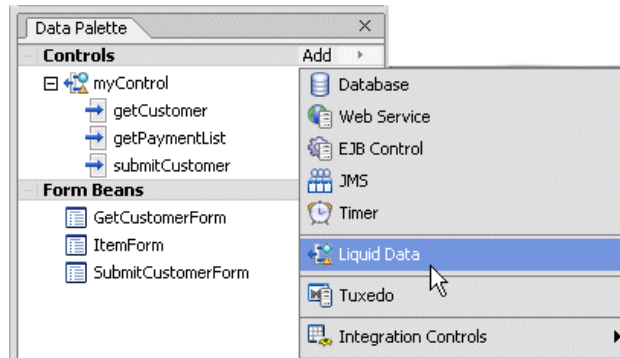
5. Add and initialize variables to the .jpf file based on the SDO classes. For details, see [“Adding Service Data Objects \(SDO\) Variables to the Page Flow”](#) on page 5-19.
6. Drag and drop the SDO variables to your JSPs to bind the data from Data Services Platform to your page layout. For details, see [“Displaying Array Values in a Table or List”](#) on page 5-23.
7. Build and test the application in WebLogic Workshop.

Adding a Data Service Control to an Existing Page Flow

You can add a data service control to an existing Page Flow .jpf file. The procedure is the same as adding a data service control to a Web Service as described in the section [“Adding a Data Service Control to a Web Service Project”](#) in Chapter 6, [“Exposing Data Services through Web Services.”](#) However, Instead of opening the Web service in Design View as described in that chapter, you open the Page Flow JPF file in Action View.

You can also add a control to an existing page flow from the Page Flow Data Palette (available in Flow View and Action View of a Page Flow) as shown in [Figure 5-10](#).

Figure 5-10 Adding a Control to a Page Flow from the Data Palette



Adding Service Data Objects (SDO) Variables to the Page Flow

To use the NetUI features to drag and drop data into a JSP, you must first create one or more variables in the page flow .jspx file. The variables must be of the data object type corresponding to the schema associated with the query.

Note: A data object is the fundamental component of the SDO architecture. For more information, see [Chapter 2, “Client Programming with Service Data Objects \(SDO\).”](#)

Defining a variable in the page flow .jspx file of the top-level class of the SDO function return type provides you access to all the data from the query through the NetUI repeater wizard. The top-level class, which corresponds to the global element of the data service type, has “Document” appended to its name, such as CUSTOMERDocument.

When you create a data service control and the SDO variables are generated, an array is created for each element in the schema that is repeatable. You may want to add other variables corresponding to other arrays in the classes to make it more convenient to drag and drop data onto a JSP, but it is not required. For example, when an array of CUSTOMER objects can contain an array of ORDER objects, you can define two variables: one for the CUSTOMER array and one for the ORDER array. You can then drag the variables to different JSP pages.

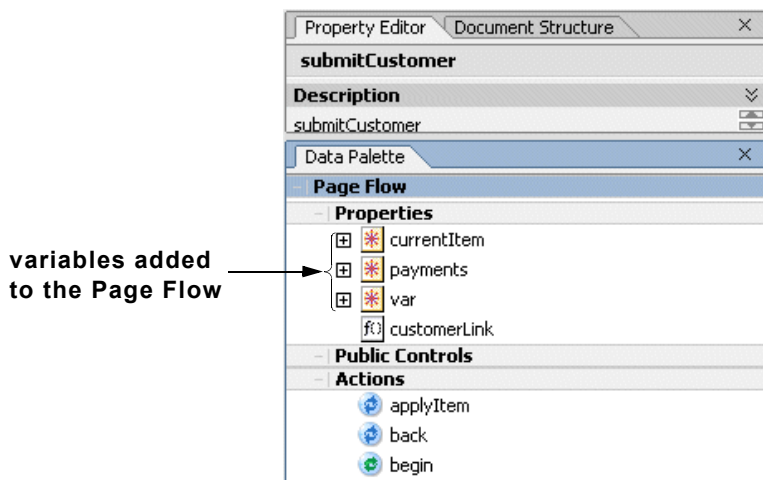
Define each variable with a type corresponding to an SDO object. Define the variables in the source view of the page flow controller class. The variables should be declared public. In the following example, the bold-typed variable declarations show an example of user variable declarations:

```
public class CustomerPFController extends PageFlowController
{
    /**
     * This is the control used to generate this pageflow
     * @common:control
     */
    private DanubeCtrl myControl;

    public CUSTOMERDocument var;
    public POITEM currentItem;
    public PAYMENTListDocument payments;
```

Once defined in the page flow controller, the variables appear on the Data Palette tab. From there, you can drag-and-drop them onto JSP files. When you drag-and-drop an array onto a JSP file, the NetUI Repeater Wizard appears and guides you through selecting the data you want to display. (See [Figure 5-11](#).)

Figure 5-11 Page Flow Variables for XMLBean Objects



To populate the variable with data, initialize the variable in the page flow method corresponding to the page flow action that calls the query. For details, see [“To Initialize the Variable in the Page Flow” on page 5-21](#).

To Add a Variable to a Page Flow

Perform the following steps to add a variable to the page flow:

1. Open your Page Flow (.jspx) file in Workshop.
2. Open the Source View tab.
3. In the variable declarations section of your Page Flow class, enter a variable with the SDO type corresponding to the schema elements you want to display. Depending on your schema, what you want to display, and how many queries you are using, you might need to add several variables.
4. To determine the SDO type for the variables, examine the method signature for each method that corresponds to a query in the data service control. The return type is the root level of the SDO class. Create a variable of that type. For example, if the signature for a control method is:

```
org.openuri.temp.schemas.customer.CUSTOMERDocument getCustomer(int p1);
```

create a variable as follows:

```
public org.openuri.temp.schemas.customer.CUSTOMERDocument var;
```

5. After you create the variables, initialize them as described in the following section.

To Initialize the Variable in the Page Flow

You can initialize the variable by calling a function in a data service control, which will populate the variable with the returned data. Initializing the variables ensures that the data bindings to the variables work correctly and that there are no tag exceptions when the JSP displays the results the first time.

Perform the following steps to initialize the variables in Page Flow:

1. Open your Page Flow (.jspx) file in Workshop.
2. Open the Source View.
3. In the page flow action that corresponds to the Data Services Platform query for which you are going to display the data, add the code to initialize the variable.

The following example shows how to initialize an object on the Page Flow. The code (and comments) in bold has been added. The rest of the code was generated when the Page Flow was created from the data service control (see [“Generating a Page Flow From a Control” on page 5-16](#)).

```
public class CustomerPFController extends PageFlowController
{
    /**
     * This is the control used to generate this pageflow
     * @common:control
     */
    private DanubeCtrl myControl;

    public CUSTOMERDocument var;
    ...
    /**
     * Action encapsulating the control method :getCustomer
     * @jpf:action
     * @jpf:forward name="success" path="viewCustomer.jsp"
     * @jpf:catch method="exceptionHandler" type="Exception"
     */
    public Forward getCustomer(GetCustomerForm aForm)
        throws Exception
    {
        var = myControl.getCustomer(aForm.pl);
        ...
        return new Forward("success");
    }
}
```

Working with Data Objects

After creating and initializing a data objects as a public variable in the Page Flow, you can drag and drop elements of the object onto your application pages (such as JSPs) from the Data Palette.

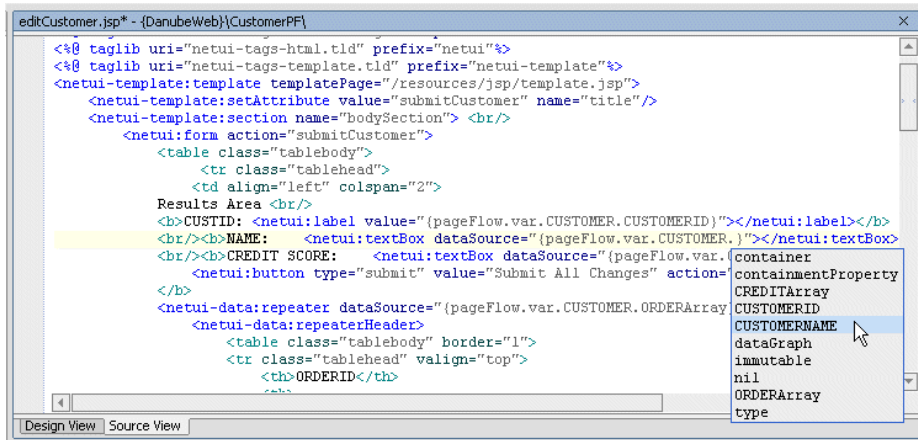
The elements appear in dot-delimited chain format, such as:

```
pageFlow.var.CUSTOMER.CUSTOMERNAME
```

Notice that the function that actually returns the element value is `getCUSTOMERNAME()`, which returns a `java.lang.String` value, the name of a customer.

As you edit code in the source view, Workshop offers code completion for method and member names as you type. A selection box of available elements appears in the data object variable as shown in

[Figure 5-12](#).

Figure 5-12 DataObject Method Name Completion

Note: For more information on programming with DSP data objects, see [Chapter 2, “Client Programming with Service Data Objects \(SDO\).”](#)

Displaying Array Values in a Table or List

DSP maps to an array any data element specified to have unbounded maximum cardinality in its XML schema definition. Unbounded cardinality means that there can be zero to many (unlimited) occurrences of the element (indicated by an asterisk in the return type view of the DSP Console). Such elements are named with the prefix `ArrayOf`.

When you drag and drop an array value onto a JSP File, BEA WebLogic Workshop displays the Repeater wizard to guide you through the process of selecting the data you want to display. The Repeater wizard provides choices for displaying the results in an HTML table or in a list.

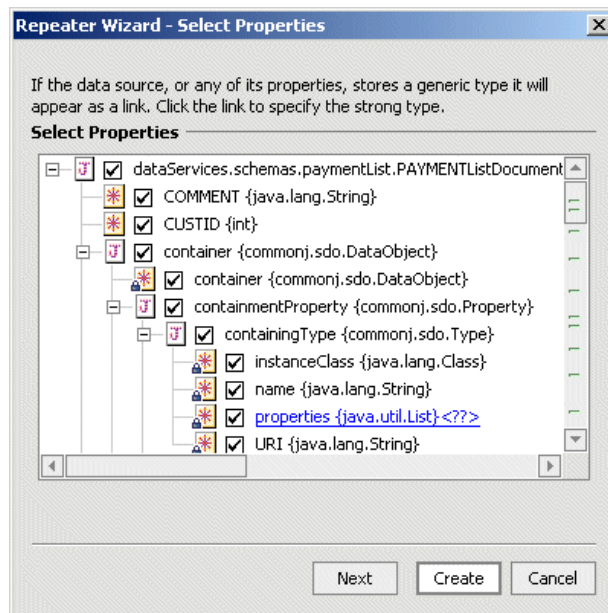
Adding a Repeater to a JSP File

To add a NetUI repeater tag (used to display the data from a Data Services Platform query) to a JSP file, perform the following steps:

1. Open a JSP file in your Page Flow project where you want to display data. This should be the page corresponding to the action in which the variable is initialized.
2. In the Data Palette → Page Flow Properties, locate the variable containing the data you want to display.

3. Expand the nodes of the variable to expose the node that contains the data you want to display. If the variable does not traverse deep enough into your schema, you will have to create another variable to expose the part of your schema you require. For details, see [“To Initialize the Variable in the Page Flow” on page 5-21](#).
4. Select the node you want, then drag and drop it onto the location of the JSP file in which you want to display the data. You can do this either in Design View or Source View. Workshop displays the repeater wizard as shown in [Figure 5-13](#).

Figure 5-13 Repeater Wizard



5. In the repeater wizard, navigate to the data you want to display and uncheck any fields that you do not want to display. There might be multiple levels in the repeater tag, depending on your schema.
6. Click Next. The Select Format screen appears as shown in [Figure 5-14](#).

Figure 5-14 Repeater Wizard Select Format Screen

Repeater Wizard - Select Format

Data Format

☒ Table
☐ List Labeled List ▼
☐ Text

Example:

Field1	Field2	Field3	Field4
Value1	Value1	Value1	Value1
Value2	Value2	Value2	Value2
Value3	Value3	Value3	Value3

Title Field (Optional)

<no title> ▼

Title Field does not apply to the Table data format

Previous Create Cancel

7. Choose the display format for your data and click Create.
8. Right-click on the JSP page and choose Run Page to see the results.

Adding a Nested Level to an Existing Repeater

You can create repeater tags inside other repeater tags. You can display nested repeaters on the same page (in nested tables, for example) or you can set up Page Flow actions to display the nested level on another page (with a link, for example).

To create a nested repeater tag, perform the following steps:

1. Add a repeater tag as described in [“Adding a Repeater to a JSP File”](#) on page 5-23.
2. Add a column to the table where you want to add the nested level.
3. Drag and drop the array from your variable corresponding to your nested level into the data cell you created in the table.
4. In the repeater wizard, select the items you want to display.
5. Click the Create button in the repeater wizard to create the repeater tags.
6. Right-click on the JSP page and choose Run Page to see the results.

Adding Code to Handle Null Values

It is a common JSP design pattern to add conditional code to handle null checks. If you do not check for null values returned by function invocations, your page will display tag errors if it is rendered before the functions on it are executed.

To add code to handle null values, perform the following steps:

1. Add a repeater tag as described in [“Adding a Repeater to a JSP File” on page 5-23](#).
2. Open the JSP file in source view.
3. Find the `netui-data:repeater` tag in the JSP file.
4. If the `dataSource` attribute of the `netui-data:repeater` tag directly accesses an array variable from the page flow, then you can set the `defaultText` attribute of the `netui-data:repeater` tag. For example:

```
<netui-data:repeater dataSource="{pageFlow.promo}" defaultText="no data">
```

If the `dataSource` attribute of the `netui-data:repeater` tag accesses a child of the variable from the page flow, you must add `if/else` logic in the JSP file as described below.

5. If the `defaultText` attribute can have a null value for your `netui-data:repeater` tag, add code before and after the tag to test for null values. The following is sample code. The code in bold is added, the rest is generated by the repeater wizard. This code uses the profile variable initialized in [“To Initialize the Variable in the Page Flow” on page 5-21](#).

```
<%
PageFlowController pageFlow = PageFlowUtils.getCurrentPageFlow(request);
if ( ((pF2Controller)pageFlow).profile == null
    ||
    ((pF2Controller)pageFlow).profile.getPROFILEVIEW().getCUSTOMERPROFILEArray
    () == null
    ||
    ((pF2Controller)pageFlow).profile.getPROFILEVIEW().getCUSTOMERPROFILEArray
    ().length == 0) {
    %>
    <p>No data</p>
    <% } else { %>
<netui-data:repeater dataSource=
    "{pageFlow.profile.PROFILEVIEW.CUSTOMERPROFILEArray}">
<netui-data:repeaterHeader>
```

```

        <table cellpadding="2" border="1" class="tablebody" >
        <tr>
<!-- the rest of the table and NetUI code goes here -->
<td><netui:label value
        ="{container.item.PROFILE.DEFAULTSHIPMETHOD}"></netui:label></td>
        </tr>
        </netui-data:repeaterItem>
        <netui-data:repeaterFooter></table></netui-data:repeaterFooter>
</netui-data:repeater>
        <% }%>

```

6. Test the application.

Using the Data Services Platform in Business Process Projects

You can use DSP in WebLogic Integration (WLI) business process applications through a Data Services Platform control. DSP information can be used, for example, in decision-making logic in the business process. The procedure for adding a data service control to a business process application is similar to adding a control to a web project.

However, an important difference exists in how data objects are unmarshalled in business processes from web applications. As a result, you need to serialize the data graph manually when submitting changed data objects as described in this section.

There are three basic steps to adding Data Services Platform Queries to a WebLogic Integration business processes:

- [Creating a Data Service Control](#)
- [Adding a Data Service Control to a JPD File](#)
- [Setting Up the Control in the Business Process](#)

Note: For comprehensive information about WebLogic Integration, see the [WebLogic Integration](#) documentation.

Creating a Data Service Control

Before you can run a Data Services Platform query in a WLI business process, you must create a data service control that accesses the query or queries you want to run in your business process. For details, see [“Data Service Controls” on page 5-2](#).

Adding a Data Service Control to a JPD File

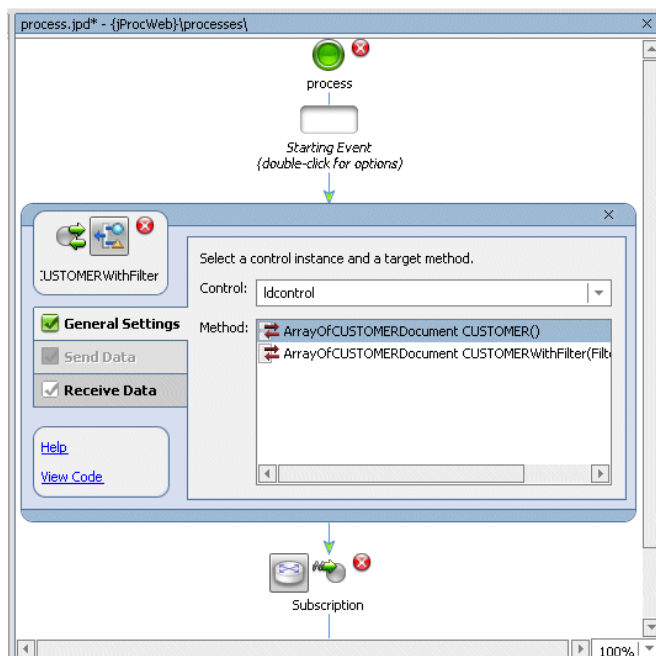
Once you have created a data service control, you can add it to a business process the same way you add any other control to a business process. For example, you can drag and drop the control into the WebLogic Integration business process in the place where you want to run your Data Services Platform query or you can add the data service control to the Data Palette. For comprehensive information about using WebLogic Integration, see the [WebLogic Integration](#) documentation.

Setting Up the Control in the Business Process

Once you have added the data service control to your business process, you can use its functions in the business process.

As shown in [Figure 5-15](#), you must select the query in the General Settings section of the data service control portion of the business process, specify input parameters for the query in the Send Data section, and specify the output of the query in the Receive Data section.

Figure 5-15 WebLogic Integration Business Process Accessing a Data Service Control



Submitting Changes from a Business Process

By default, a business processes (JPD) converts XML objects to an XML proxy class by implementing an interface named `ProcessXML`. However, `ProcessXML` is not completely compatible with SDO. In particular, it does not accommodate SDO specific features such as change summaries. As a result, the default XML processing performed in a business process must be overridden.

You can override the business process by performing the following steps in the workflow:

1. In the JPD you need to turn off default `ProcessXML` deserialization and enable XBean serialization on the XML object factory by calling the `XmlObjectVariableFactory.setXBean()`.
2. Invoke the data service control.
3. In the JPD you need to disable the XBean serialization and turn on the default `ProcessXML` deserialization on the XML object by calling `XmlObjectVariableFactory.unset()`.

Caching Considerations When Using Data Service Controls

The following scenario is very common: most of the time you can use cached data because it changes infrequently; however, on occasion, your application must fetch data directly the data source. At the same time, you want to update your cache with the most up-to-date information. A typical example would be to refresh the cache at the beginning of every week or month.

You can accomplish this by passing the attribute `GET_CURRENT_DATA` with your function call.

Bypassing the Cache When Using a Data Service Control

To bypass the data in a cached query function result, your application will need to signal Liquid Data to retrieve results directly from the data source, rather than from its cache. The steps required to accomplish this include:

- Adding an additional function to the set already defined in your `LiquidData` control (`.jcx`) file. This function will take a `QueryAttribute` object as a parameter.
- Instantiate a `QueryAttribute` object in your application and call the `enableFeature()` method, passing the `GET_CURRENT_DATA` attribute.
- Call the function you defined in your `LiquidData` control, passing the `QueryAttribute` object.

Cache Bypass Example When Using a Data Service Control

[Listing 5-2](#) shows example Java Page Flow (JPF) code that tests whether the user has requested a bypass of any cached data. If `refreshCache` is set to `false` then cached data (if any is available) is used. Otherwise the function will be invoked with the `GET_CURRENT_DATA` attribute and data will be retrieved from the data source. As a byproduct, any cache is automatically refreshed.

Listing 5-2 Cache Bypass Example When Using Data Services Platform Control

```
if (refreshCache == false) {
    customerDocument = LDControl.getCustomerProfile(CustomerID);
} else {
    QueryAttributes attr = new QueryAttributes();
    attr.enableFeature(QueryAttributes.GET_CURRENT_DATA);
    customerDocument =
        LDControl.getCustomerProfileWithAttr(CustomerID, attr);
}
```

As mentioned above, an additional function is also needed in the your Liquid Data control (.jcx) file. For the code shown in [Listing 5-2](#), you would add the following definition to your Liquid Data control:

```
/**
 * @jc:XDS functionURI="ld:DataServices/CustomerProfile"
 * functionName="getCustomerProfile"
 */
CUSTOMERPROFILEDocument getCustomerProfileWithAttr (java.lang.String p0,
QueryAttributes attr);
```

Security Considerations When Using Data Service Controls

This section describes security considerations for applications using a data service control. The following sections are included:

- [Security Credentials Used to Create Data Service Controls](#)
- [Testing Controls With the Run-As Property in the JWS File](#)
- [Trusted Domains](#)

Security Credentials Used to Create Data Service Controls

The WebLogic Workshop Application Properties (Tools → Application Properties) allow you to set the connection information to connect to the domain in which you are running. You can either use the connection information specified in the domain `boot.properties` file or override that information with a specified username and password.

When you create a Data Services Platform control JCX file and are connecting to a local Data Services Platform server (Data Services Platform on the same domain as Workshop), the user specified in the Application Properties is used to connect to the Data Services Platform server. When you create a data service control and are connecting to a remote Data Services Platform server (a WebLogic Server on a different domain from Workshop), you specify the connection information in the data service control wizard connection information dialog (see [Figure 5-4](#)).

When you create a data service control, the Control Wizard displays all queries to which the specified user has access privileges. The access privileges are defined by security policies set on the queries, either directly or indirectly.

Note: The security credentials specified through the Application Properties or through the data service control wizard are only used for creating the data service control JCX file, not for testing queries through the control. To test a query through the control, you must get the user credentials either through the application (from a login page, for example) or by using the run-as property in the Web Service file.

Testing Controls With the Run-As Property in the JWS File

You can use the run-as property to test a control running as a specified user. To set the run-as property in a Web Service, open the Web Service and enter a user for the run-as property in the WebLogic Workshop property editor.

When a query is run from an application, the application must have a mechanism for getting the security credential. The credential can come from a login screen, it can be hard-coded in the application, or it can be imbedded in a J2EE component (for example, using the run-as property in a `.jws` Web Service file).

Trusted Domains

If the WebLogic Server that hosts the DSP project is on a different domain than WebLogic Workshop, then both domains must be set up as trusted domains.

Domains are considered trusted domains if they share the same security credentials. With trusted domains, a user known to one domain need not be authenticated on another domain, if the user is already known on that domain.

Note: After configuring domains as trusted, you must restart the domains before the trusted configuration takes effect.

Configuring Trusted Domains

To configure domains as a trusted user, perform the following steps:

1. Log into the WebLogic Administration Console as an administrator.
2. In the left-frame navigation tree, click the node corresponding to your domain.
3. At the bottom of the General tab for the domain configuration, click the link labeled View Domain-wide Security Settings Links.
4. Click the Advanced tab. (See [Figure 5-16.](#))

Figure 5-16 Setting up Trusted Domains

The screenshot shows the 'Domain Wide Security Settings' page in the WebLogic Administration Console. The page title is 'liquiddata> Domain Wide Security Settings'. The top navigation bar shows 'Connected to : localhost:7001', 'You are logged in as : ldsystem', and a 'Logout' link. The 'Configuration' tab is selected, and the 'Advanced' sub-tab is active. The page content includes a warning icon and the text 'This page allows you to define the advanced security settings for this WebLogic Server domain.' Below this, there is a checkbox labeled 'Enable Generated Credential' which is currently unchecked. A descriptive text explains that this credential is used to establish trust between domains. There are two input fields labeled 'Credential:' and 'Confirm Credential:', both containing masked text. An 'Apply' button is located at the bottom right of the form area.

5. Uncheck the Enable Generated Credential box, enter and confirm a credential (usually a password), and click Apply.
6. Repeat this procedure for all of the domains you want to set up as trusted. The credential must be the same on each domain.

For more details on WebLogic security, see:

- [“Configuring Security for a WebLogic Domain”](#) in the WebLogic Server documentation.

For information on Data Services Platform security, see:

- ["Securing DSP Resources"](#) in the *Administration Guide*.

Accessing Data Services from Workshop Applications

Exposing Data Services through Web Services

This chapter describes how to expose data services as standard web services. It covers these topics:

- [Exposing Data Services as Web Services](#)
- [Adding a Data Service Control to a Web Service Project](#)
- [Creating a Web Service From a Data Service Control](#)

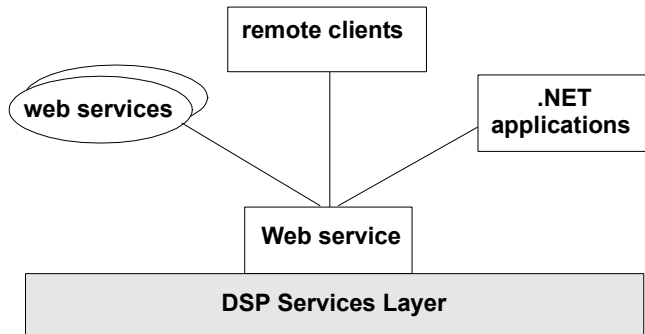
Exposing Data Services as Web Services

Using WebLogic Workshop and a BEA AquaLogic Data Services Platform (DSP) control, you can easily add a web service wrapper to a data service. Doing so gives your data services the benefits of standard web service features, including loose client/server coupling, UDDI capability and WS-Security.

WS-Security is particularly useful because it provides encryption-based, message-level security for your data.

Exposing data services as web services can make your data service information accessible to a wide variety of client types including other web services, .NET, or any non-Java application. [Figure 6-1](#) illustrates the relationship between these client types and the DSP services layer.

Figure 6-1 Web Service Clients



To expose data services through a web service interface, add a data service control to a web service project in WebLogic Workshop. Then add data service functions from the control to your web service. The web service function needs only to pass through the results of the data service function, as shown in the following sample of generated code:

```

public class myCustomerWS implements com.bea.jws.WebService {
...
    public dataServices.payments.CustomerDocument getCustomer(int p0) {
        return customerDsCtrl.getPayments(getCustomer(p0)); }
}
    
```

You can then generate a WSDL from the web service by right-clicking on the WSDL (.jws file). Once deployed, the data service function can be used from clients applications just like any other web service deployed to the WebLogic server. Keep in mind that data is returned from DSP as standard SOAP-encased XML data, not as service data objects.

Note: This chapter focuses on how to expose data services through a standard web service interface. For more information on consuming WebLogic web services, see [“Invoking Web Services”](#) in *Programming WebLogic Web Services* in the WebLogic Platform documentation.

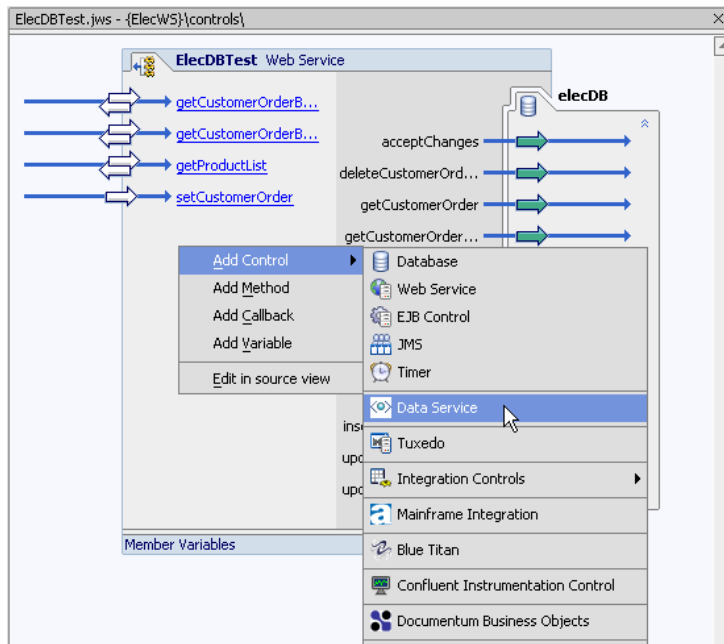
Adding a Data Service Control to a Web Service Project

To add a data service control to an existing Web Service file (.jws), perform the following steps:

1. Make sure the WebLogic Server is running.
2. In WebLogic Workshop, open the existing Web Service .jws file.
3. Click the Design View tab on the Web service.

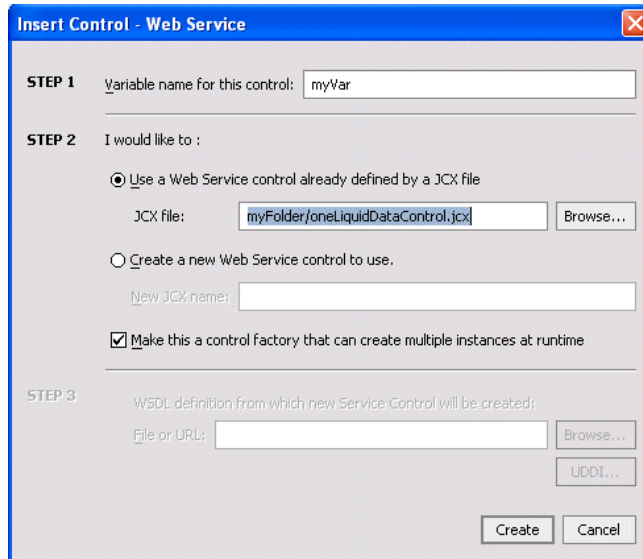
4. In the graphical representation of the Web service, right-click and select Add Control → Data Services Platform as shown in [Figure 6-2](#).

Figure 6-2 Adding a Data Service Control to a Web Service



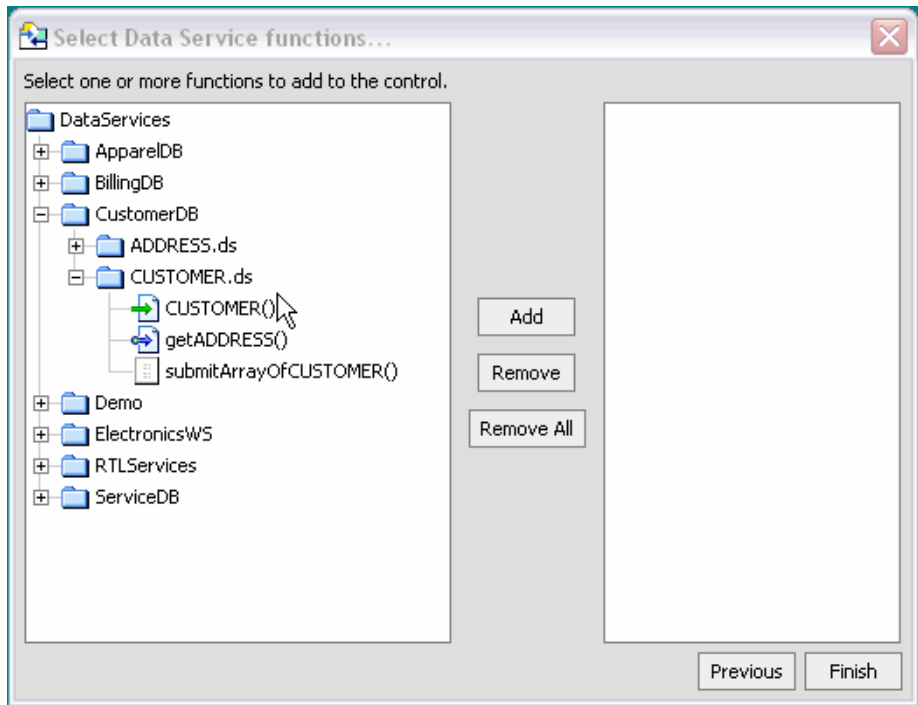
5. In the Insert Control wizard, enter a variable name for the control (STEP 1 in the dialog in [Figure 6-3](#)). The variable name can be any valid variable name that is unique in the Web Service.
6. In the Insert Control wizard, either browse to an existing Data Services Platform control (it must be in the same project as the Web Service) or click the Create a New Data Service Control button.
7. If you want the control to be a factory, check the Make This a Control Factory button as shown in [Figure 6-3](#). If the control is a factory, it will create multiple instances at runtime. Otherwise, requests to the control are queued and each request for a given query must complete before another can begin.

Figure 6-3 Insert Control Wizard



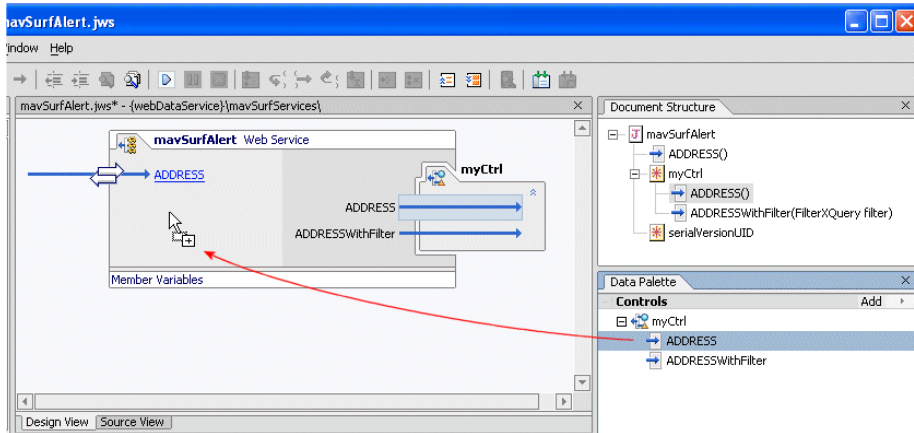
8. If Data Services Platform is deployed on a WebLogic Server that is running on a separate domain from Workshop, click remote (in STEP 3 of the Insert Control Wizard dialog). For details about specifying a local or remote Data Services Platform server, see [“Step 5: Enter Connection Information to the WebLogic Server”](#) on page 5-11 in Chapter 5, [“Accessing Data Services from Workshop Applications.”](#)
9. Click the Create button on the Insert Control Wizard.
10. If you created a new control, in the Select Data Services Platform Queries dialog select the data service functions you want from the left pane and click Add as shown in [Figure 6-4](#). When done, click Finish. At that point, the data service control JCX file is generated, with a call for each selected function.

The `LiquidDataControl.jar` file is copied into the `Libraries` directory of your application when you finish creating the data service control.

Figure 6-4 Data Service Control Wizard—Select Queries

11. Now add functions to the callable interface of the web service. To add a data service function to the web service, choose the function from below the control node in the Data Palette then drag and drop the function onto the left side of the web service diagram as illustrated in [Figure 6-5](#).

Figure 6-5 Adding a Function to the Web Service



12. Click the test button (or select Debug → Start from the Workshop menu) to test your web service.

From there, you can work with the Data Services Platform web service as you can any other web service in WebLogic Workshop. You can generate a WSDL file for it, and more.

For example, to create a WSDL file, right-click the JWS file in the application tree and choose Generate WSDL File from the menu.

For more information, see:

<http://e-docs.bea.com/wls/docs81/webservices.html>

Creating a Web Service From a Data Service Control

Perform the following steps to generate and test a web service from a data service control. You can create either conversational or stateless web services with data service. The following instructions describe how to create a conversational web service.

1. Select a data service control JCX file, right-click, and select Generate Stateless JWS File. Workshop generates the .jws Java Web Service file for your data service control.
2. Select your Web Service project, right-click, and select Build Project. Workshop builds a Web Service project.
3. When the build is complete, double-click the .jws file to open it.

4. On the Design View of the Web Service, notice the `startTestDrive` and `finishTestDrive` methods, as well as a method for each of the queries you specified in the data service control wizard.
5. Click the test button (or select Debug → Start from the Workshop menu) to test the web service.
6. Click the `startTestDrive` button to start the conversation for the Web Service.
7. Click the Continue this Conversation link (in the left corner of the test page).
8. Enter values for any query parameters (if the query has parameters) and click the button with the name corresponding to the query you want to execute.

The Web Service executes the query and the results are returned to a test browser.

9. If you want to run the query again or run other queries in the Web Service, click Continue this Conversation, enter any needed parameters and click the button with the name corresponding to the query you want to execute.
10. To end the Web Service conversation, click the Continue this Conversation link and then click the `finishTestDrive` button.

Once deployed, the web service can be used just like any other web service deployed on WebLogic servers. For more information, see ["Invoking Web Services"](#) in *Programming WebLogic Web Services* in the WebLogic Server documentation.

Exposing Data Services through Web Services

Using the Data Services Platform JDBC Driver

The BEA AquaLogic Data Services Platform (DSP) JDBC driver gives client applications a means to obtain JDBC access to the information made available by data services. The driver implements the `java.sql.*` interface in JDK 1.4x to provide access to an DSP server through the JDBC interface. You can use the JDBC driver to execute SQL92 SELECT queries, or stored procedures over DSP applications. This chapter explains how to install and use the Data Services Platform JDBC driver. It covers the following topics:

- [About the Data Services Platform JDBC Driver](#)
- [Installing the Data Services Platform JDBC Driver with JDK 1.4x](#)
- [Using the JDBC Driver](#)
- [Connecting to the JDBC Driver from a Java Application](#)
- [Connecting to Data Services Platform Client Applications Using the ODBC-JDBC Bridge from Non-Java Applications](#)
- [Using Reporting Tools with the Data Services Platform ODBC-JDBC Driver](#)
- [DSP and SQL Type Mappings](#)
- [SQL-92 Support](#)

Note: For data source and configuration pool information, refer to the WebLogic [Administration Guide](#). Your configuration settings may affect performance.

About the Data Services Platform JDBC Driver

The JDBC driver is intended to enable SQL access to data services. The Data Services Platform JDBC driver enables JDBC and ODBC clients to access information available from data services. The JDBC driver increases the flexibility of the DSP integration layer by enabling access from database visualization and reporting tools, such as Crystal Reports. From the point of view of the client, the DSP integration layer appears as a relational database, with each data service function comprising a table. Internally, DSP translates SQL queries into XQuery.

There are several constraints associated with the Data Services Platform JDBC driver. Because SQL provides a traditional, two-dimensional approach to data access (as opposed to the multiple level, hierarchical approach defined by XML), the Data Services Platform JDBC driver can only be used to access data through data services that have a flat data shape; that is, the data service type cannot have nesting.

Also, SQL tables do not have parameters; therefore, the Data Services Platform JDBC driver only exposes non-parameterized flat data service functions as tables. (Parameterized flat data services are exposed as SQL stored procedures.)

To expose non-flat data services, you can create flat views to be used from the JDBC driver.

Features of the Data Services Platform JDBC Driver

The Data Services Platform JDBC driver has the following features:

- Supports SQL-92 SELECT statements
- Implements JDBC 3.0 API
- Supports Data Services Platform with JDK 1.4
- Usable from both Java and ODBC clients

Notes:

- The Data Services Platform JDBC driver contains the following third party libraries: `xerces`
Java - 2.6.2 : `xercesImpl.jar`, `xmlParserAPIs.jar`, and `ANTLR 2.7.4` :
`antlr.jar`.
- The driver also contains the following DSP product libraries: `wlclient.jar`,
`ld-client.jar`, `Schemas_UNIFIED_Annotation.jar`, `jsr173_api.jar`, and
`xbean.jar`.

Data Services Platform and JDBC Driver Terminology

DSP views data retrieved from a database as comprised of data sources and functions. This means that Data Services Platform terminology and the terminology used when accessing data through the Data Services Platform JDBC driver, which provides access to a database, is different. The following table shows the equivalent terminology between the two.

Table 7-1 Data Services Platform and JDBC Driver Terminology

Data Services Platform Terminology	JDBC Driver Terminology
DSP Application Name	Database Catalog Name
Path from the DSP project folder up to the folder name of the data source separated by a ~ (tilde)	Database Schema Name
Function with parameters	Stored procedure
Function without parameters	Table
Function without parameters return type schema's elements	Table's Columns
Function with parameters return type schema's elements	Stored Procedure's Columns

For example, if you have an application Test with a project TestDataServices, and CUSTOMERS.ds with a function getCustomers() under a folder MyFolder, the table getCustomers can be describes as:

```
Test.TestDataServices~MyFolder.getCustomer
```

where Test is the catalog and TestDataServices~MyFolder is the schema.

Installing the Data Services Platform JDBC Driver with JDK 1.4x

The Data Services Platform JDBC driver is located in an archive file named `ldjdbc.jar`. In a DSP installation, the archive is in the following directory:

```
<WebLogicHome>/liquiddata/lib/
```

To use the driver on a client computer, perform the following steps:

1. Copy the `ldjdbc.jar` to the client computer.

2. Add `ldjdbc.jar` to the computer's classpath.
3. Set the appropriate supporting path by adding `%JAVA_HOME%\jre\bin` to your path.
4. To configure the JDBC driver:

- a. Set the driver class name to:

```
com.bea.ld.jdbc.LiquidDataJDBCdriver.
```

- b. Set the driver URL to:

```
jdbc:ld@<LDServerName>:<LDServerPortNumber>[:<LDCatalogAlias>]
```

For example, `jdbc:ld@localhost:7001` or

```
jdbc:ld@localhost:7001:ldCatalogName.
```

If you want to enable logging for debugging use, you can append the following to the driver URL

```
;debugStdOut=true;debugFile=ldjdbc.log;debugLog=true;
```

You can also specify configuration parameters as a Properties object or as a part of the JDBC URL. The following is an example of how to specify the parameters as part of a Properties object:

```
props = new Properties();
props.put(LiquidDataJDBCdriver.USERNAME_PROPERTY, "weblogic");
props.put(LiquidDataJDBCdriver.PASSWORD_PROPERTY, "weblogic");
props.put(LiquidDataJDBCdriver.APPLICATION_NAME_PROPERTY, "RTLApp");
props.put(
    LiquidDataJDBCdriver.PROJECT_NAME_PROPERTY, "DataServices-CustomerDB");
props.put(LiquidDataJDBCdriver.WLS_URL_PROPERTY, "t3://localhost:7001");
props.put(LiquidDataJDBCdriver.DEBUG_STDOUT_PROPERTY, "true");
props.put(LiquidDataJDBCdriver.DEBUG_LOG_PROPERTY, new Boolean(true));
props.put(
    LiquidDataJDBCdriver.DEBUG_LOG_FILENAME_PROPERTY, "ldjdbc.log");
Class.forName("com.bea.ld.jdbc.LiquidDataJDBCdriver");
con = DriverManager.getConnection(
    "jdbc:ld@localhost:7001:Demo:DemoLdProject", props);
```

Alternatively, you can specify all the parameters in the JDBC URL itself as shown in the following example:

```
Class.forName("com.bea.ld.jdbc.LiquidDataJDBCdriver");
con =
    DriverManager.getConnection("jdbc:ld@localhost:7001:Demo:DemoLdProject;
        ;debugStdOut=true;debugFile=ldjdbc.log;debugLog=true;username=weblogic;
        password=weblogic;", new Properties());
```


Using the JDBC Driver

The steps for connecting an application to DSP as a JDBC/SQL data source are substantially the same as for connecting to any JDBC/SQL data source. In the database URL, simply use the DSP application name as the database identifier with "ld" as the sub-protocol, in the form:

```
jdbc:ld@<WLServerAddress>:<WLServerPort>:<LDApplicationName>
```

For example:

```
jdbc:ld@localhost:7001:RTLApp
```

The name of the Data Services Platform JDBC driver class is:

```
com.bea.ld.jdbc.LiquidDataJDBCdriver
```

Note: If you are using the WebLogic Administration Console to configure the JDBC connection pool, set the initial connection capacity to 0. The Data Services Platform JDBC driver does not support connection pooling.

The following section describes how to connect using the driver class in a client application.

Obtaining a Connection

A JDBC client application can connect to a deployed DSP application in the same way as it can to any database. It loads the Data Services Platform JDBC driver and then establishes a connection to DSP.

For example:

```
Properties props = new Properties();
props.put("user", "weblogic");
props.put("password", "weblogic");

// Load the driver
Class.forName("com.bea.ld.jdbc.LiquidDataJDBCdriver");

//get the connection
Connection con =
    DriverManager.getConnection("jdbc:ld@localhost:7001", props);
```

Using the preparedStatement Interface

The following method demonstrates how to use the preparedStatement interface given a connection object (con) that is a valid connection obtained through the java.sql.Connection interface to a WebLogic server hosting DSP. (In the method, CUSTOMER refers to a CUSTOMER data service.)

```
public ResultSet storedQueryWithParameters() throws java.sql.SQLException {  
    PreparedStatement preStmt =  
        con.prepareStatement (  
            "SELECT * FROM CUSTOMER WHERE CUSTOMER.LAST_NAME=?");  
    preStmt.setString(1, "SMITH");  
    ResultSet res = preStmt.executeQuery();  
    return res;  
}
```

Note: You can create a preparedStatement for a non-parametrized query as well. The statement can also be used in the same manner.

Getting Data Using JDBC

Once a connection is established to a server where DSP is deployed, you can call a data service function to obtain data by using a parameterized data service function call.

The following method demonstrates calling a stored query with a parameter (where con is a connection to the Data Services Platform server obtained through the java.sql.Connection interface). In the snippet, a stored query named dtaQuery is executed where custid is the parameter name and CUSTOMER2 is the parameter value.

```
public ResultSet storedQueryWithParameters(String paramName)  
    throws java.sql.SQLException {  
    //prepare a stored query to execute  
    CallableStatement call = con.prepareCall("dtaQuery");  
    call.setString(1, "CUSTOMER2");  
    ResultSet resultSet = call.execute();  
    return resultSet;  
}
```

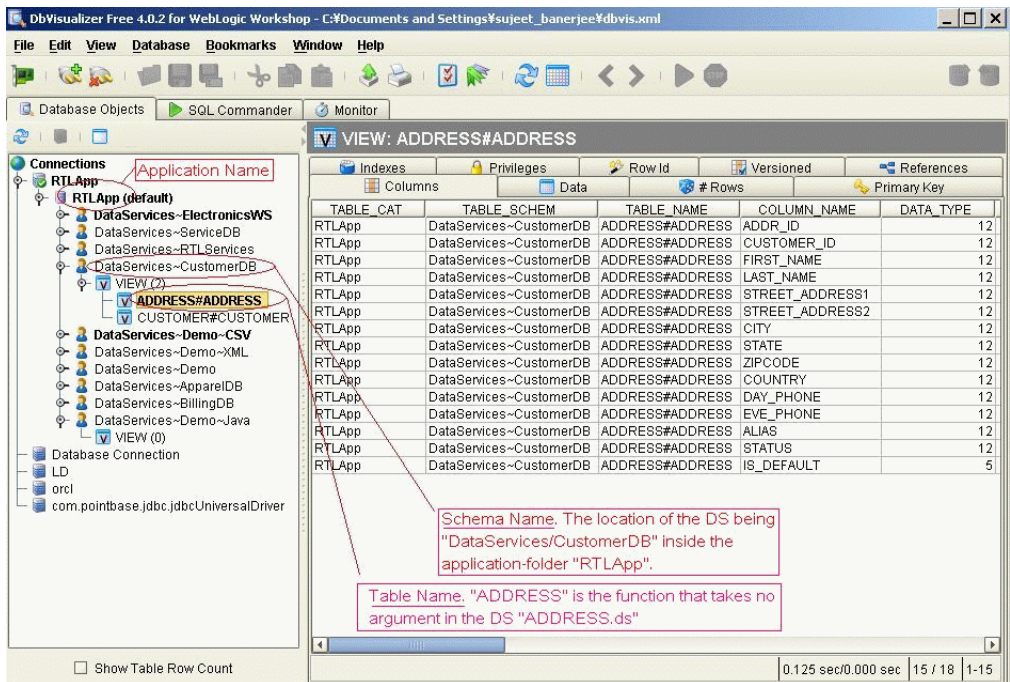
Connecting to the JDBC Driver from a Java Application

You can also use the Data Services Platform JDBC driver from client Java applications. This is a good way to learn how Data Services Platform exposes its artifacts through its JDBC/SQL driver.

Note: For details on supported reporting applications and connectivity software see "Configuring the Data Services Platform JDBC Driver for Reporting Applications" in the Preparing to Install Data Services Platform chapter of the DSP [Installation Guide](#).

This section describes how to connect to the driver from DBVisualizer. [Figure 7-2](#) shows a sample application as viewed from DbVisualizer for WebLogic Workshop.

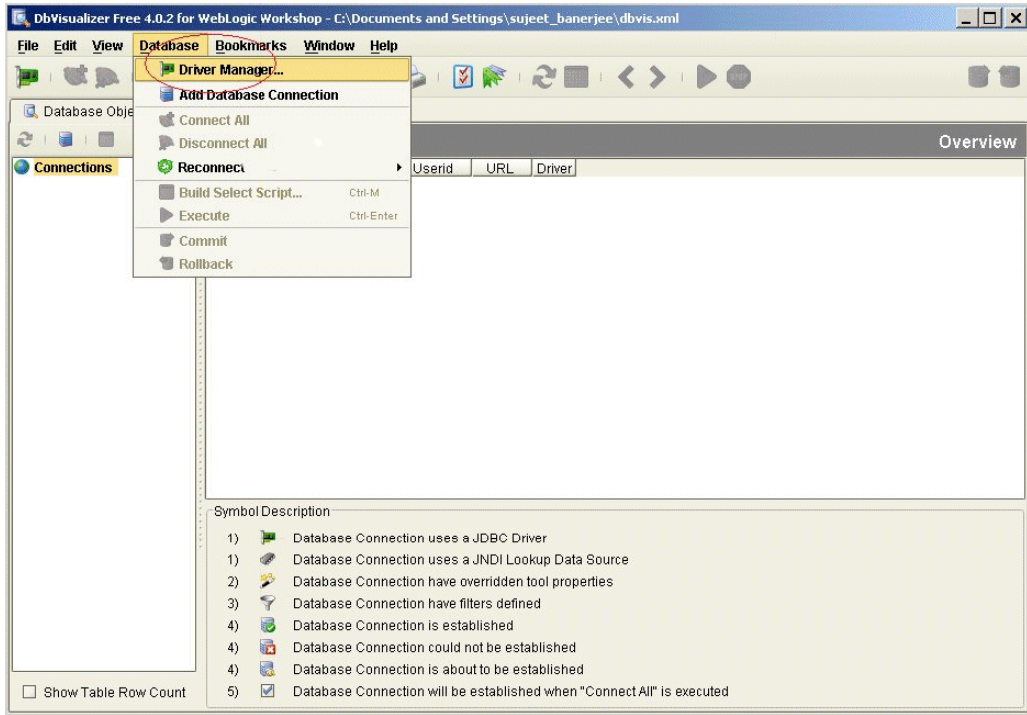
Figure 7-2 DbVisualizer View of DSP



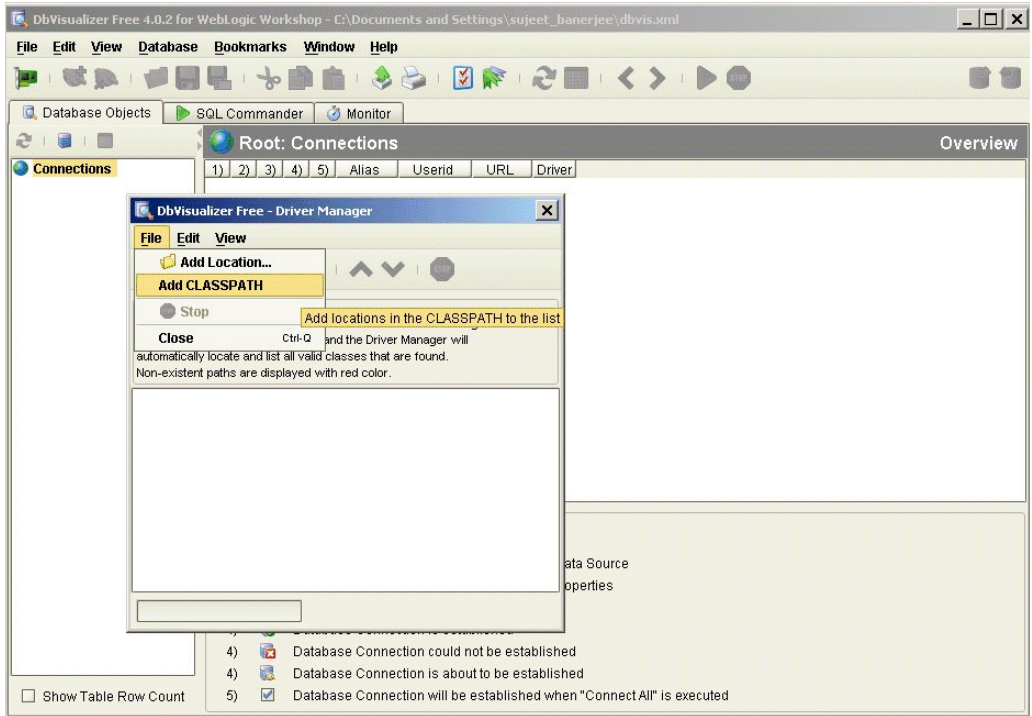
To use DBVisualizer, perform the following steps:

1. Configure DBVisualizer.
 - a. Ensure that `ldjdbc.jar` exists in your CLASSPATH. Start DBVisualiser from the Database menu select Driver Manager.

Using the Data Services Platform JDBC Driver

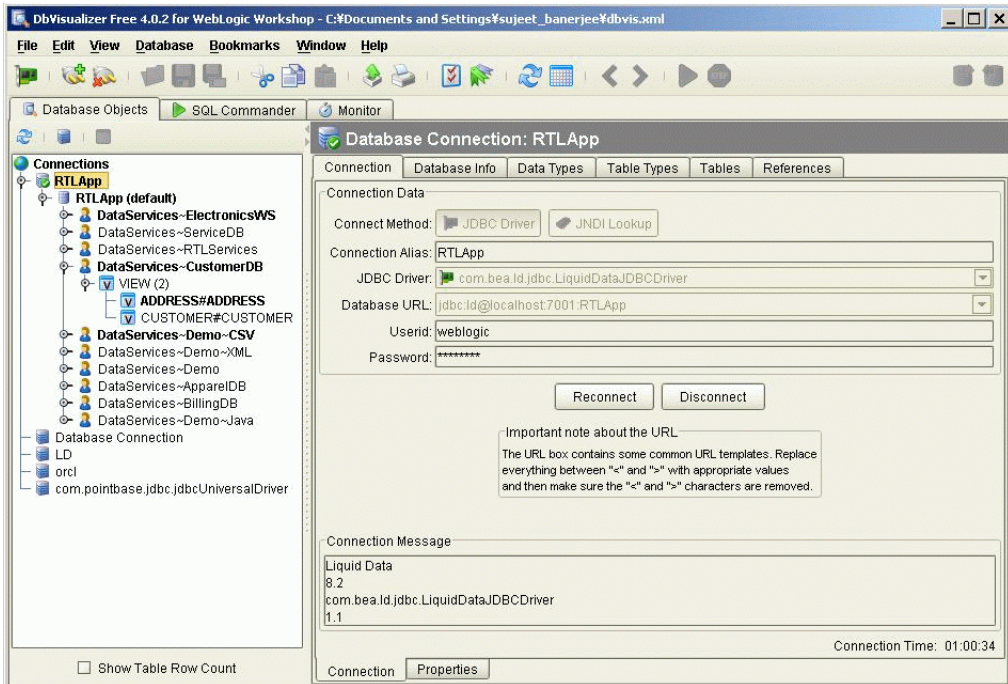


- b. Select Add CLASSPATH from the File menu of the driver manager dialog. You should see the `ldjdbc.jar` listed.
- c. Select `ldjdbc.jar` from the list shown then select Find Drivers from the Edit menu of the driver manager. You should see the `com.bea.ld.jdbc.LiquidDataJDBCdriver`. This means the JDBC driver has been located.



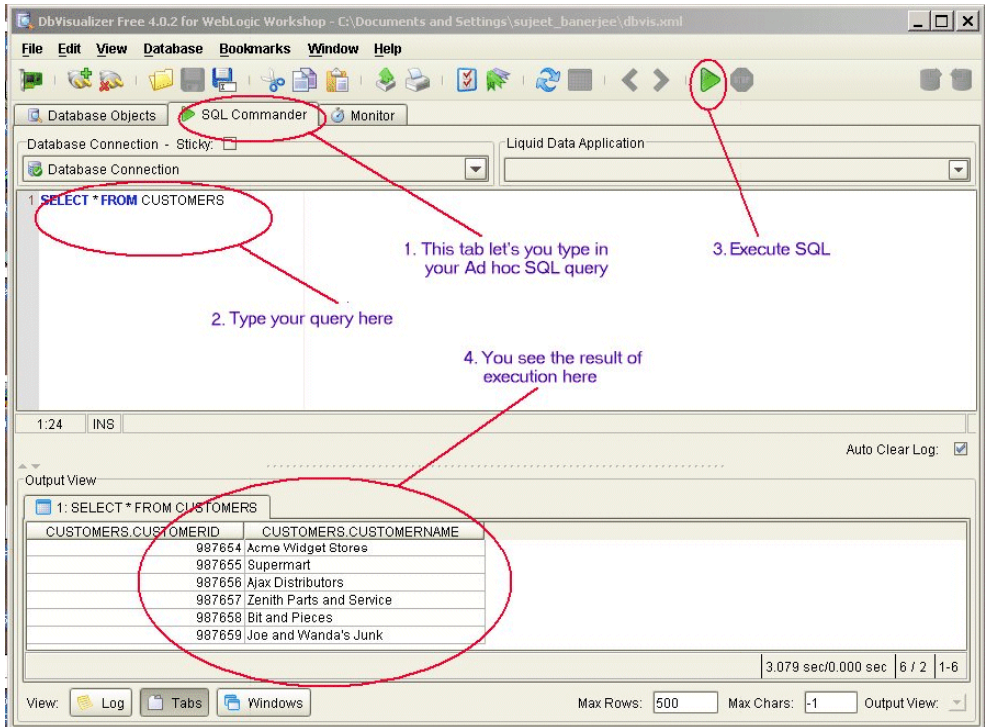
- d. Close the driver manager.
2. Add connection parameters by performing the following steps:
 - a. On the right pane select the JDBC Driver as `com.bea.ld.jdbc.LiquidDataJDBCdriver`, dropping down the list.
 - b. For the Database URL, enter `jdbc:ld@<machine_name>:<port>:<app_name>`. For example `"jdbc:ld@localhost:7001:RTLApp"`
 - c. Provide the username and password for connecting to the DSP application.
3. Click connect. On completion of a successful connection, you should see the following:

Using the Data Services Platform JDBC Driver



4. On the right pane of the window (see figure in step 3), you can see various tabs. The Tables tab helps you view the information about the tables, including their metadata. The References tab lets you view the field information and primary key of each table.
5. Execute ad hoc queries by activating the SQL Commander tab as shown in the following figure. Type in your SQL query and click the execute button.

Connecting to the JDBC Driver from a Java Application



Connecting to Data Services Platform Client Applications Using the ODBC-JDBC Bridge from Non-Java Applications

You can use an ODBC-JDBC bridge to connect to Data Services Platform JDBC driver from non-Java applications. This section describes how to configure the OpenLink and EasySoft ODBC-JDBC bridges to connect non-Java applications to the Data Services Platform JDBC driver.

Note: For details on supported reporting applications and connectivity software see "Configuring the Data Services Platform JDBC Driver for Reporting Applications" in the Preparing to Install Data Services Platform chapter of the DSP *Installation Guide*.

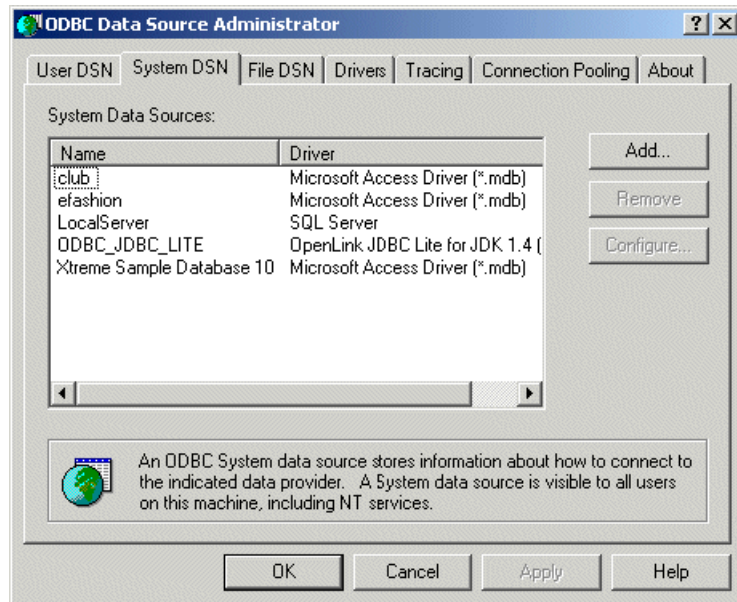
Using the EasySoft ODBC-JDBC Bridge

Applications can also communicate with the Data Services Platform JDBC Driver using EasySoft's ODBC-JDBC Gateway. The installation and use of the EasySoft Bridge is similar to the OpenLink bridge discussed in the previous section.

To use the EasySoft bridge, perform the following steps:

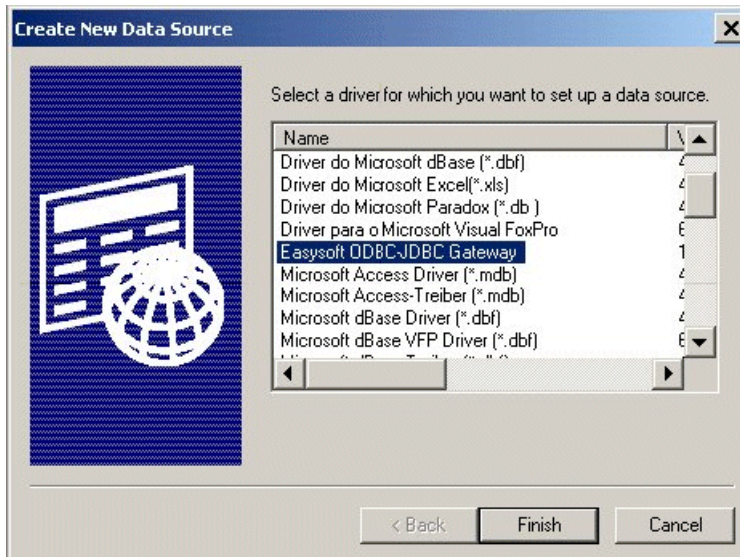
1. Install the EasySoft ODBC-JDBC bridge. Go to the EasySoft site for information about installation:
<http://www.easysoft.com>
2. Creating a system DSN and configuring it with respect to DSP by performing the following steps:

- a. Open Administrative tools → Data Sources (ODBC).



- b. Go to the System DSN tab and click Add.

- c. Select EasySoft ODBC-JDBC Gateway as shown in the figure below and click Finish.



- d. On the next screen, fill in the fields as follows:

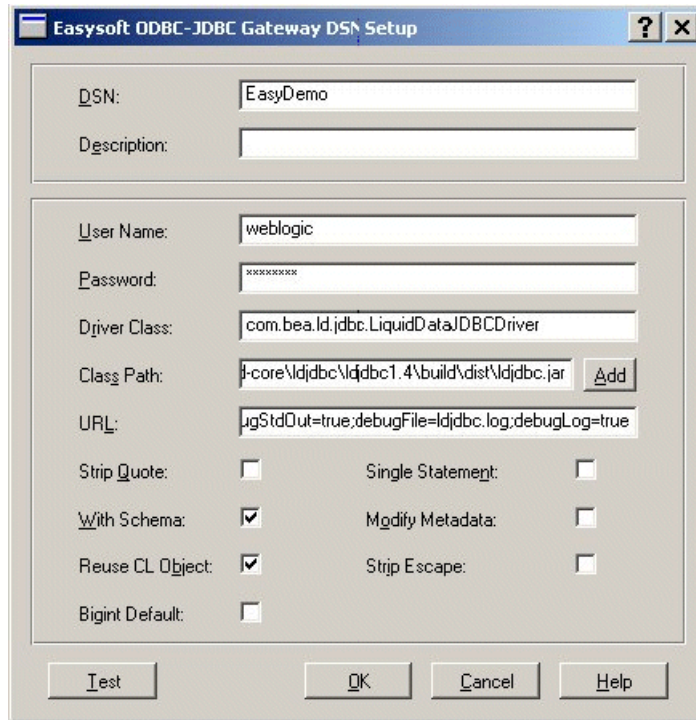
- For Class Path, enter the absolute path to the ldjdbc.jar

- For URL, enter:

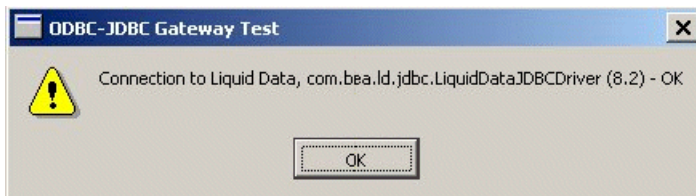
`jdbc:ld@<machine_name>:<port>:<app_name>`

- For Driver class, enter:

`com.bea.ld.jdbc.LiquidDataJDBCdriver`



- e. Click Test. The following screen will display, indicating the connection has completed successfully.



- f. Click OK to complete the set-up sequence.

Using OpenLink ODBC-JDBC Bridge

The Openlink ODBC-JDBC driver can be used to interface with the Data Services Platform JDBC driver to query DSP applications with client applications, such as Crystal Reports 10, Business Objects 6.1, and MS Access 2000.

To use the OpenLink bridge, you will need to install the bridge and create a system DSN using the bridge. The following are the steps for these two tasks:

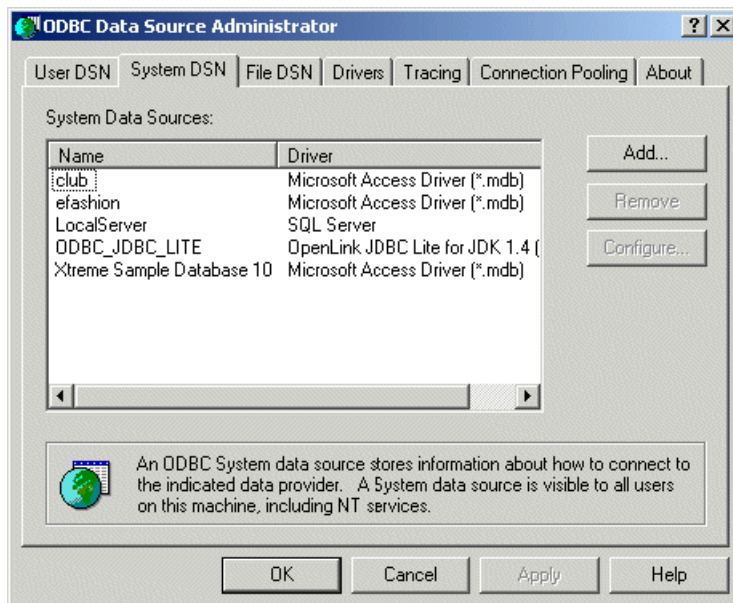
1. Install the OpenLink ODBC-JDBC bridge (called ODBC-JDBC-Lite). For information on the installation of OpenLink ODBC-JDBC-Lite, see:

<http://www.openlinksw.com/info/docs/uda51/lite/installation.html>

Warning: For Windows platforms, be sure that you preserve your CLASSPATH before installation. The installer might overwrite it.

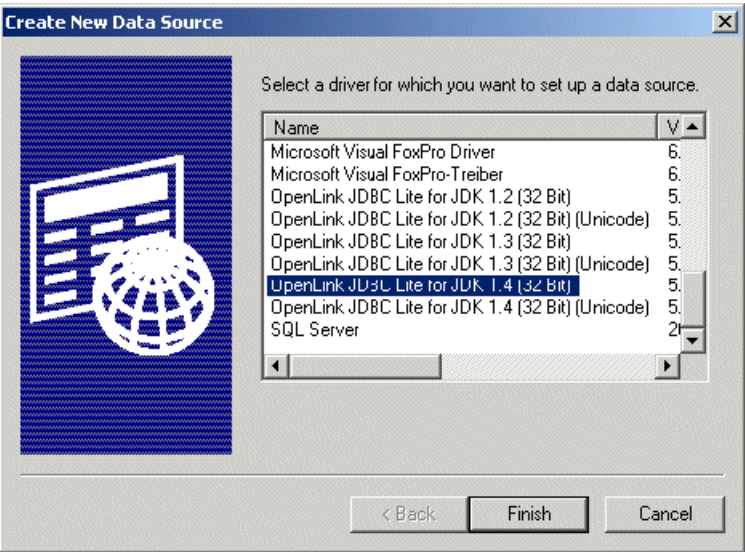
2. Create a system DSN and configure it for your DSP application by performing the following steps:
 - a. Ensure that the CLASSPATH contains the following jars required by ODBC-JDBC-Lite, as well as the `ldjdbc.jar`. A typical CLASSPATH might look like:

```
D:\lddriver\ldjdbc.jar; D:\odbc-odbc\openlink\jdk1.4\opljdbc3.jar;  
D:\odbc-jdbc\openlink\jdk1.4\megathin3.jar;
```
 - b. Update your system path to point to the `jvm.dll`, which should be under your `%javaroot%/jre/bin/server` directory.
 - c. Open Administrative tools Data Sources (ODBC). You should see the following:

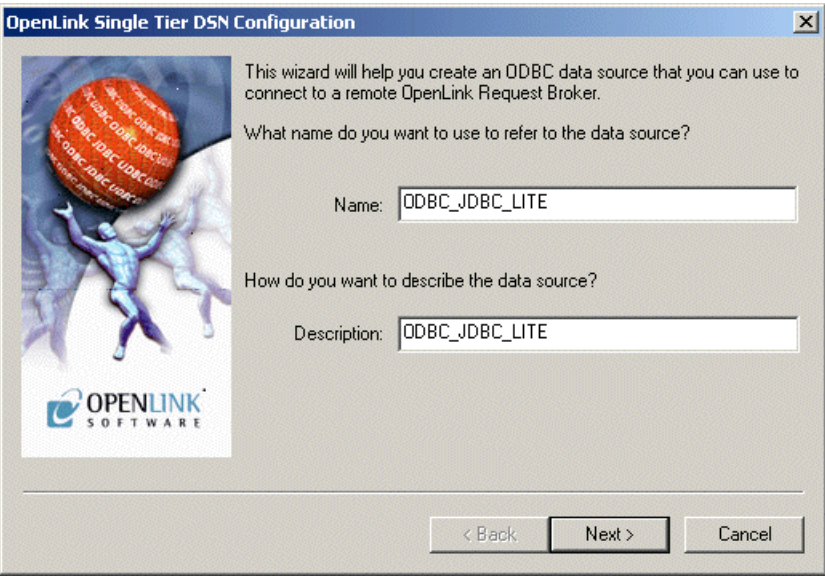


- d. Go to the System DSN tab and click Add.

- e. Select JDBC Lite for JDK 1.4 (32 bit) and click Finish.



- f. Write a name for the DSN. For example, ODBC_JDBC_LITE, as shown in the figure below.

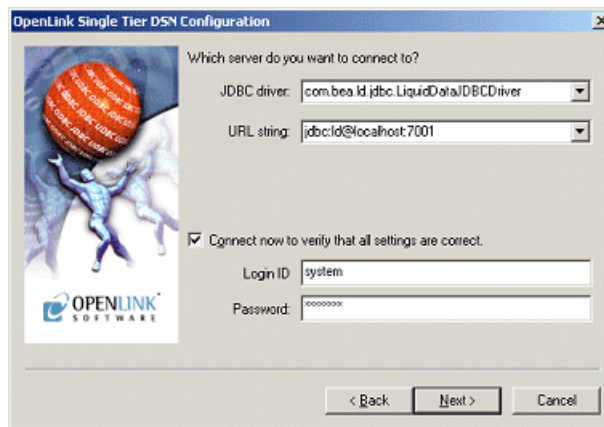


- g. Click Next. Then on the next screen, enter the following in the JDBC driver field:

`com.bea.ld.jdbc.LiquidDataJDBCDriver.`

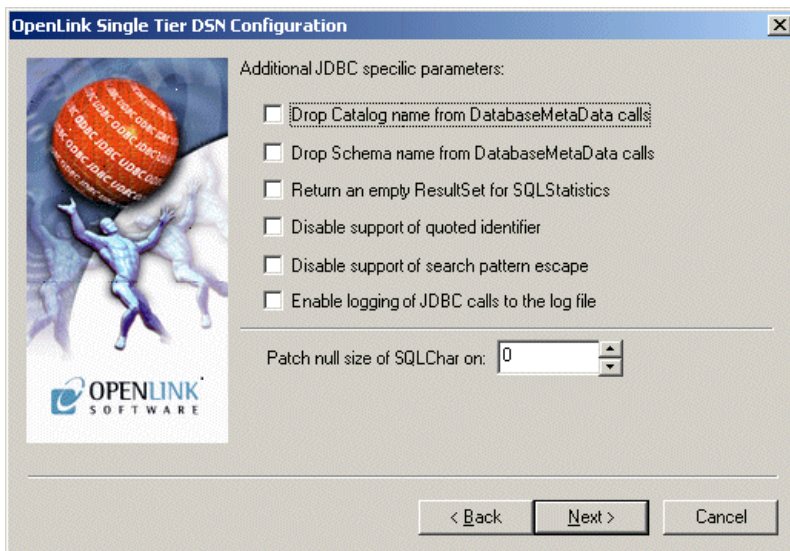
Enter the following in the URL string field:

`jdbc:ld@<machine_name>:<port>:<app_name>`

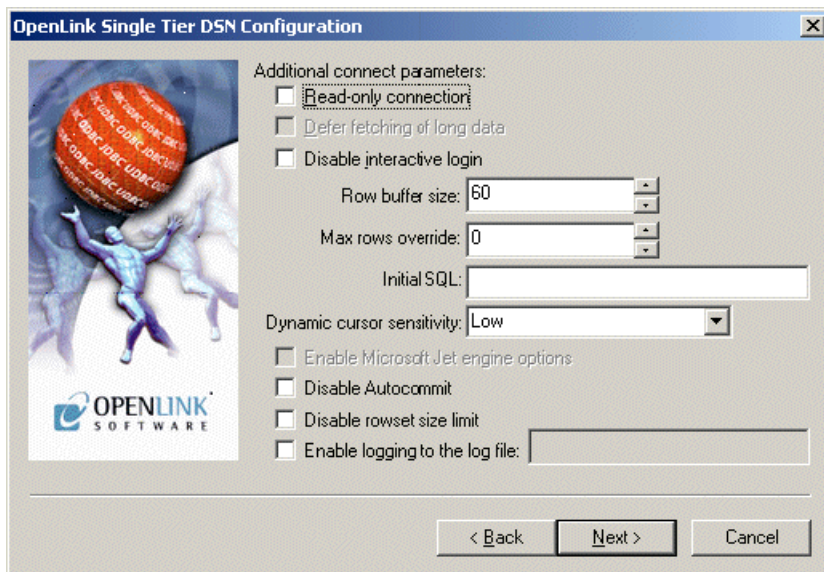


- h. Check the Connect now to verify that all settings are correct checkbox. Provide the login and password to connect to the Data Services Platform WebLogic server.

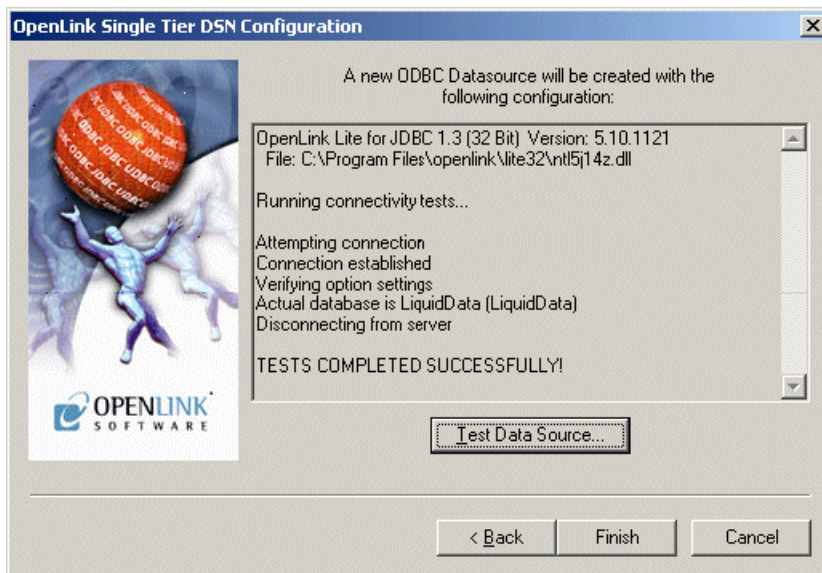
- i. Click **Next**. The screen shown below will display.



- j. Click **Next**. The following screen will display.



- k. Click Test Data Source. This screen will verify the setup is successful.



- l. Click Finish.

Using Reporting Tools with the Data Services Platform ODBC-JDBC Driver

Once you have configured your ODBC-JDBC Bridge, you can use your application to access the data source presented by DSP. The usual reason for doing so is to connect Data Services Platform to your favorite reporting tool.

Note: For details on supported reporting applications and connectivity software see "Configuring the Data Services Platform JDBC Driver for Reporting Applications" in the Preparing to Install Data Services Platform chapter of the DSP [Installation Guide](#).

This section describes how to configure the following reporting tools to use the Data Services Platform ODBC-JDBC driver:

- [Crystal Reports 10 - ODBC](#)
- [Crystal Reports 10 - JDBC](#)
- [Business Objects 6.1 - ODBC](#)
- [Microsoft Access 2000 - ODBC](#)

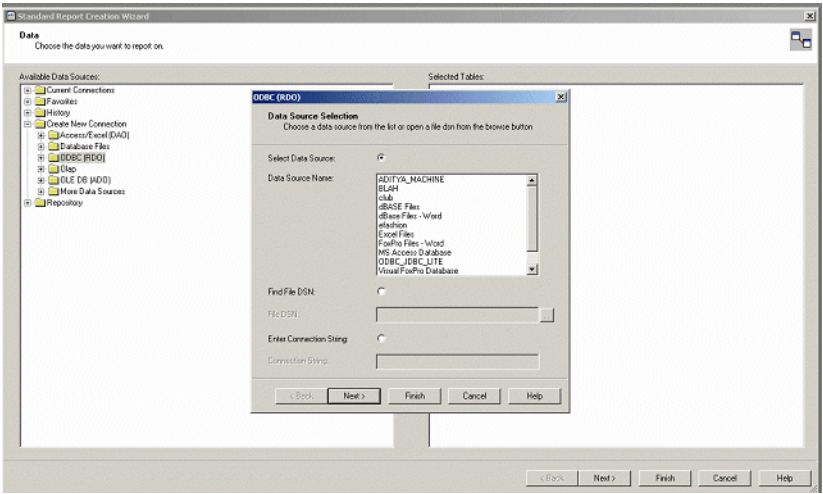
Note: Some reporting tools issue multiple SQL statement executions to emulate a scrollable cursor if the ODBC-JDBC bridge does not implement one. Some drivers do not implement a scrollable cursor, so the reporting tool issues multiple SQL statements. This can affect performance.

Crystal Reports 10 - ODBC

This section describes how to connect Crystal Reports to the Data Services Platform ODBC-JDBC driver. To connect Crystal Reports to the driver, perform the following steps:

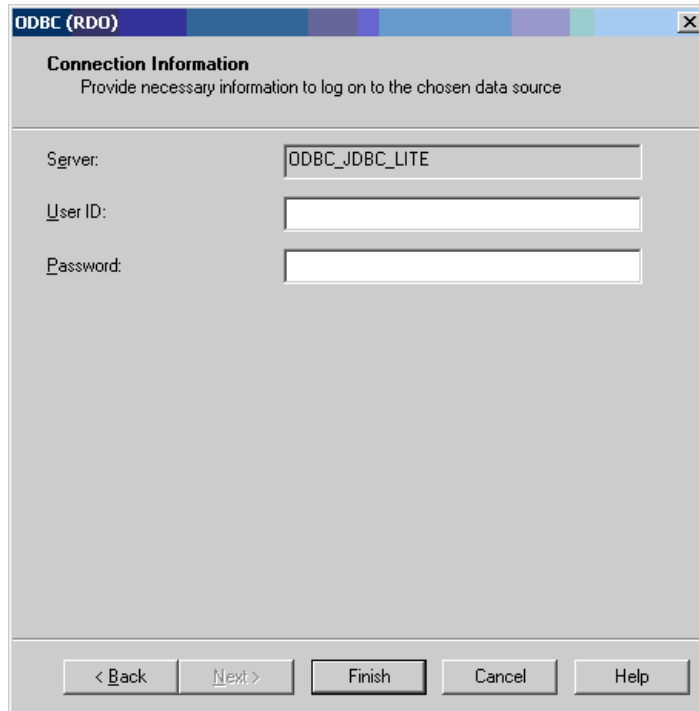
1. In Crystal Reports 10, you need to create a new Connection on ODBC RDO. You can do this by clicking on the New Report wizard button, which will prompt you immediately for a data source. Select the ODBC (RDO) option in the left-hand window as shown in the [Figure 7-3](#).

Figure 7-3 Data Source Selection



You can select the DSN you have created earlier (see the procedure in section [“Using OpenLink ODBC-JDBC Bridge”](#) or [“Using the EasySoft ODBC-JDBC Bridge”](#)). In this example, it is ODBC_JDBC_LITE.

Selecting ODBC_JDBC_LITE, prompts the following dialog:

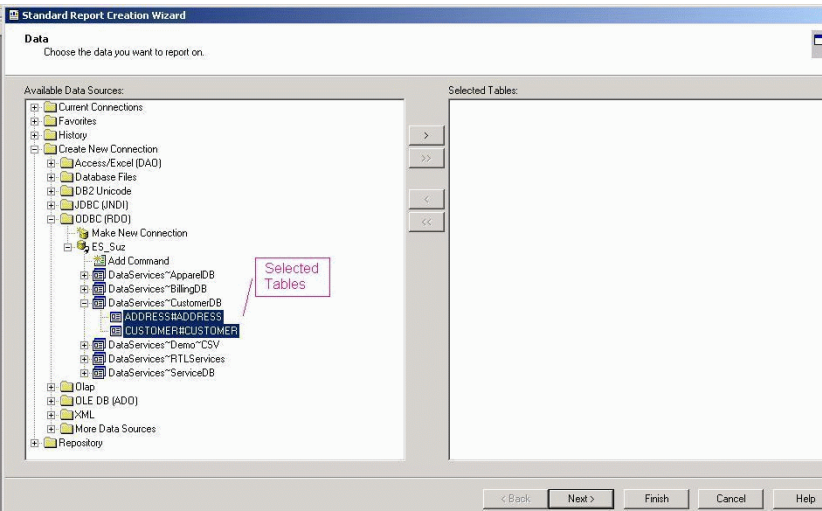


The screenshot shows a Windows-style dialog box titled "ODBC (RDO)". Below the title bar is a section header "Connection Information" followed by the instruction "Provide necessary information to log on to the chosen data source". The dialog contains three input fields: "Server:" with the text "ODBC_JDBC_LITE", "User ID:" (empty), and "Password:" (empty). At the bottom, there are five buttons: "< Back", "Next >", "Finish", "Cancel", and "Help".

2. Enter the domain login and password. Note that because the URL contains the Data Services Platform RTLApp application, you should use the domain login and password that the domain of the RTLApp application uses. (These will most likely be "weblogic".)

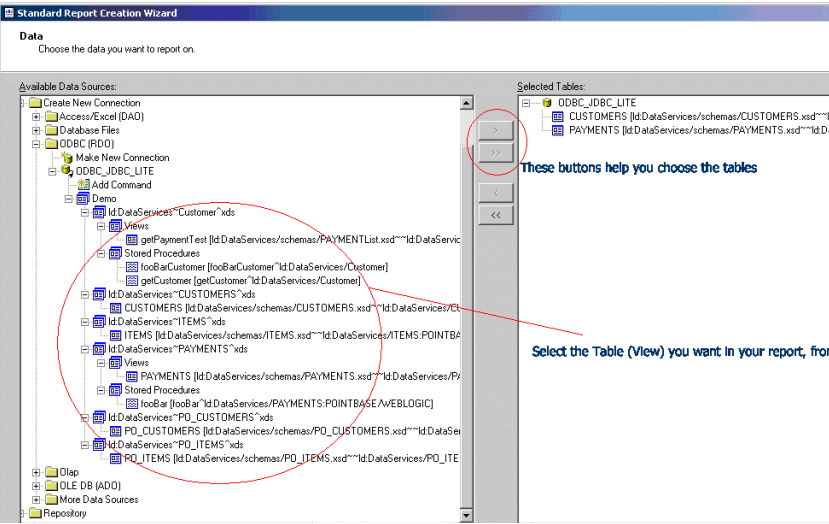
Once authenticated, Crystal Reports will show you a view of the DSP application on the server as shown in [Figure 7-4](#).

Figure 7-4 Available Data Sources



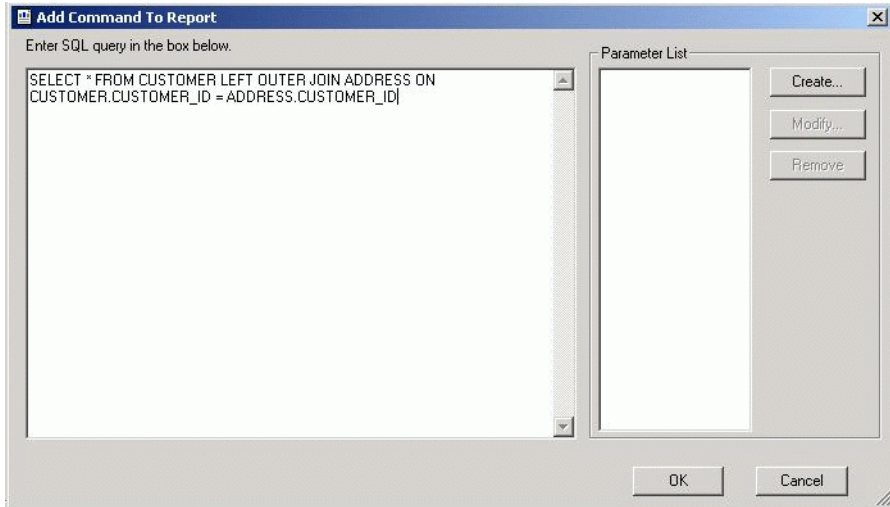
3. Generate a report using the Add command or by dragging the metadata to the right. In this example we will be using both options. You can choose the tables you want to use in the report as shown in [Figure 7-5](#).

Figure 7-5 Selecting the Table View



Alternatively, you can choose the Add Command option to type an SQL query directly, which will show you a window like one in the [Figure 7-6](#).

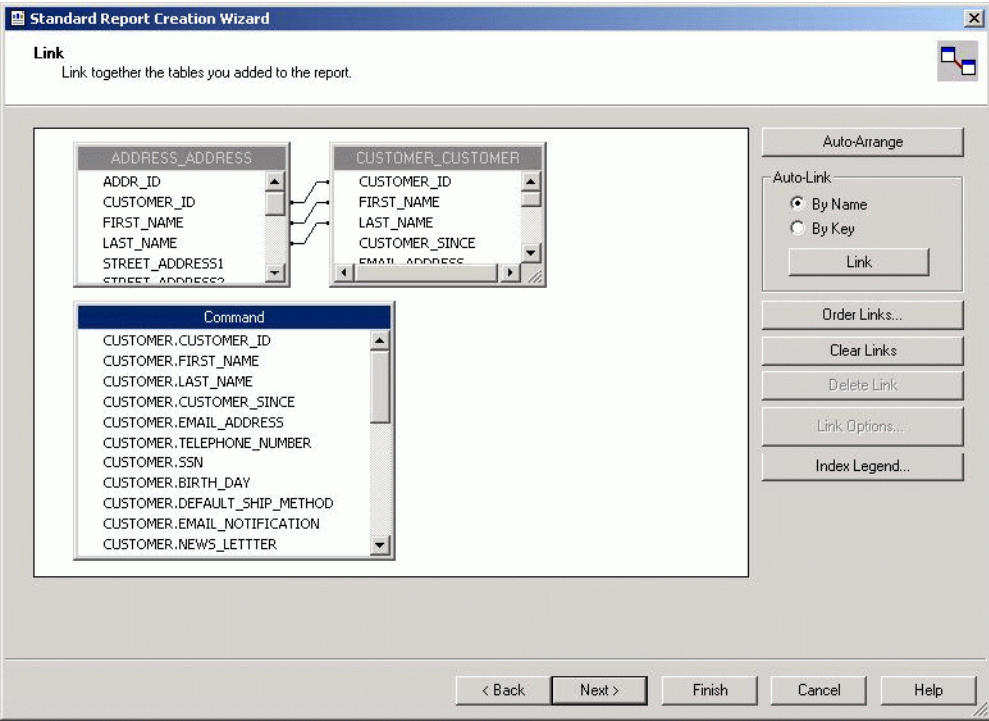
Figure 7-6 Add Command



4. Click the Ok Button to see the Command added to the Right hand side of the window.

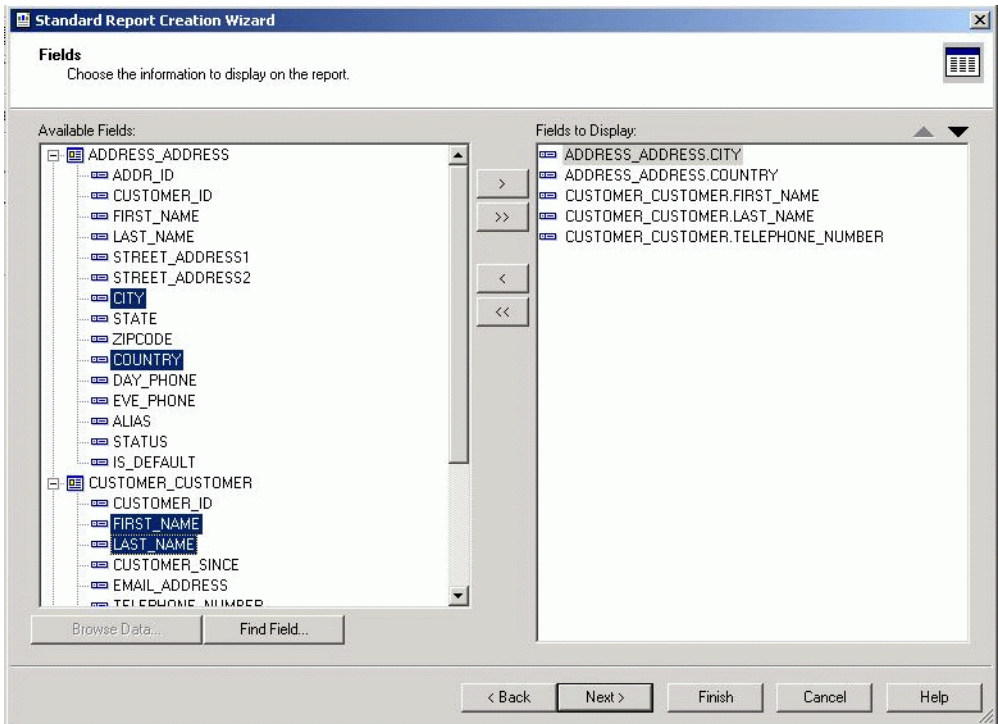
Clicking Next in the wizard shows you all the available views for this Report generation, as shown in [Figure 7-7](#).

Figure 7-7 Link Screen



Clicking Next again will take you to the Column chooser window, which allows you to select which Columns you want to see in the final Report, which appears as shown in [Figure 7-8](#).

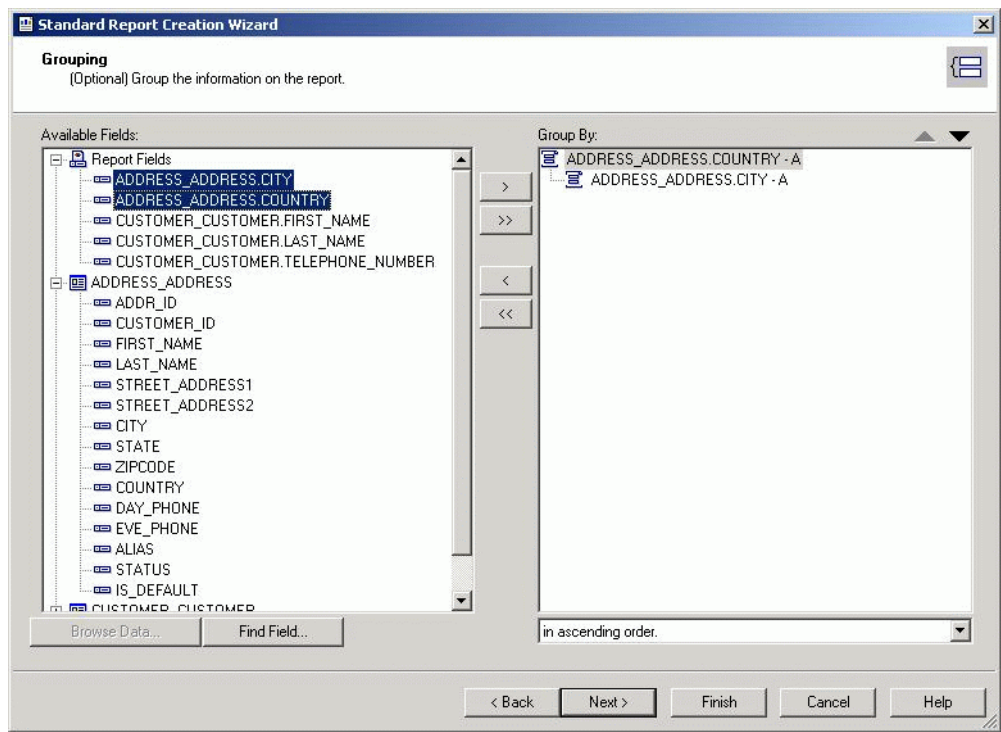
Figure 7-8 Column Chooser



Note: This example chooses columns from the user-generated Command and the view CUSTOMER.

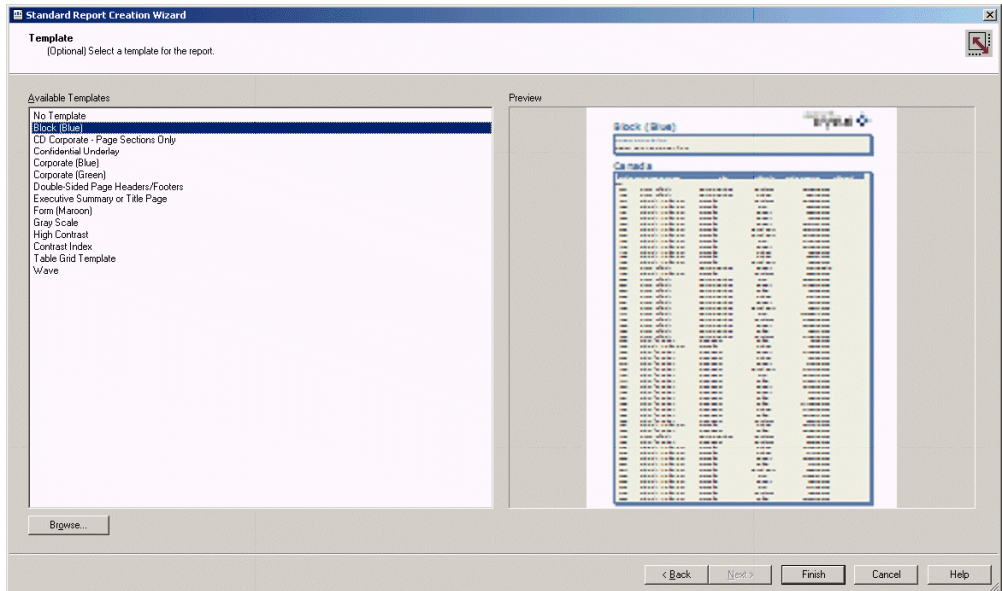
Clicking on Next again takes us to the Group by screen (as shown in [Figure 7-9](#)), which allows you to choose a column to group by. (This is grouping is performed by Crystal Reports. The Group-by information is not passed on to the JDBC driver.)

Figure 7-9 Group-by Screen



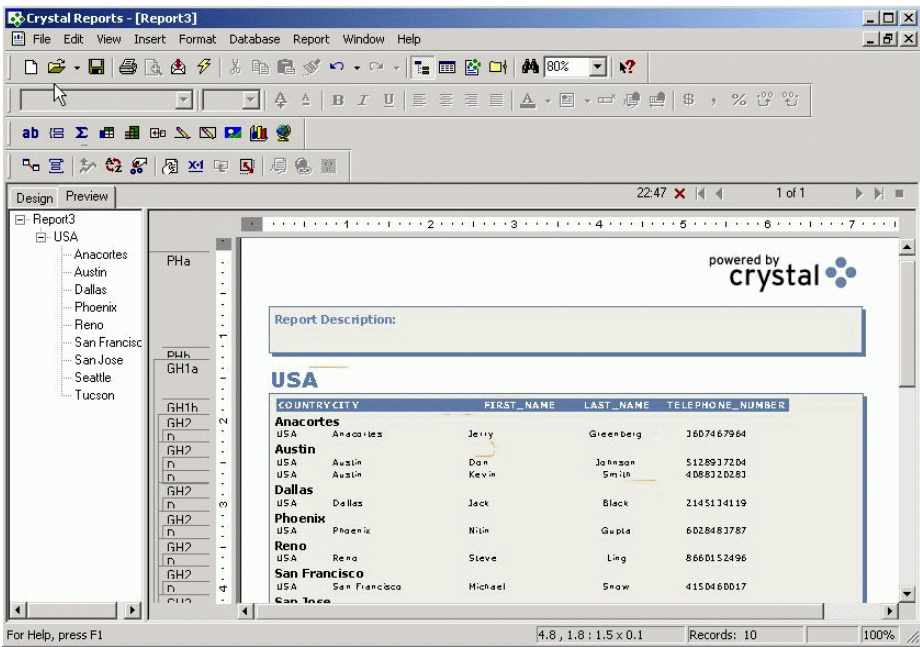
5. Skip the next few screens for now, clicking Next till you reach the Template Chooser Screen [Figure 7-10](#). Choose any appropriate Template. In this example, the user has chosen the Block (Blue) Template.

Figure 7-10 Template Chooser Screen



6. Click Finish. A Report similar to that shown in [Figure 7-11](#) is generated.

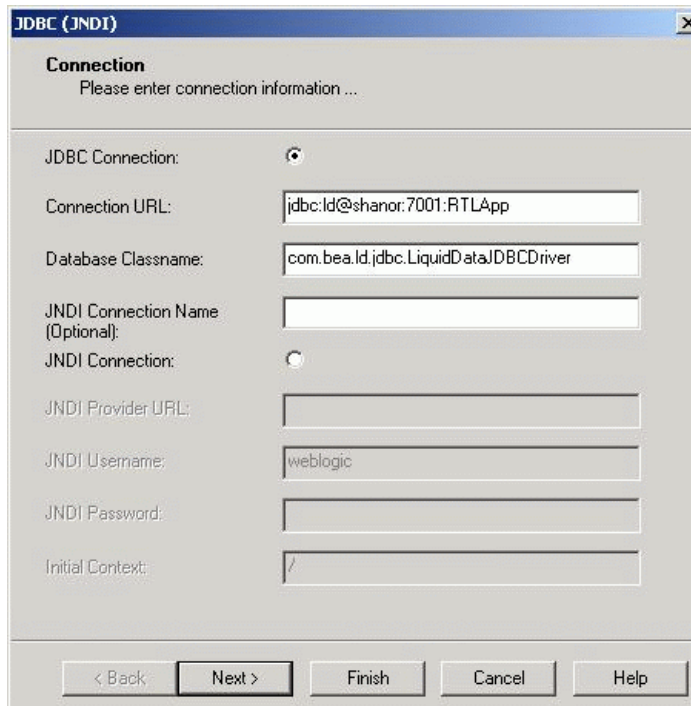
Figure 7-11 Generated Report



Crystal Reports 10 - JDBC

Crystal Reports 10.0 comes with a direct JDBC interface that can be used to interact directly with the Data Services Platform JDBC driver. The only difference between the ODBC and JDBC approach is that in JDBC, a new type of connection is used, as shown in [Figure 7-12](#).

Figure 7-12 Connection Dialog Box



The image shows a Windows-style dialog box titled "JDBC (JNDI)". Inside the dialog, there is a section labeled "Connection" with the instruction "Please enter connection information ...". Below this, there are several input fields and radio buttons. The "JDBC Connection:" radio button is selected. The "Connection URL:" field contains "jdbc:Id@shanor:7001:RTLApp". The "Database Classname:" field contains "com.bea.Id.jdbc.LiquidData/JDBCdriver". The "JNDI Connection Name (Optional):" field is empty. The "JNDI Connection:" radio button is unselected. The "JNDI Provider URL:" field is empty. The "JNDI Username:" field contains "weblogic". The "JNDI Password:" field is empty. The "Initial Context:" field contains "/". At the bottom of the dialog, there are five buttons: "< Back", "Next >", "Finish", "Cancel", and "Help".

[Figure 7-13](#) shows screen that requests the connection parameters for the JDBC Interface of Crystal Reports.

Figure 7-13 Connection Information Dialog Box

JDBC (JNDI)

Connection Information
Provide necessary information to log on to the chosen data source

Server:

User ID:

Password:

Database:
▼

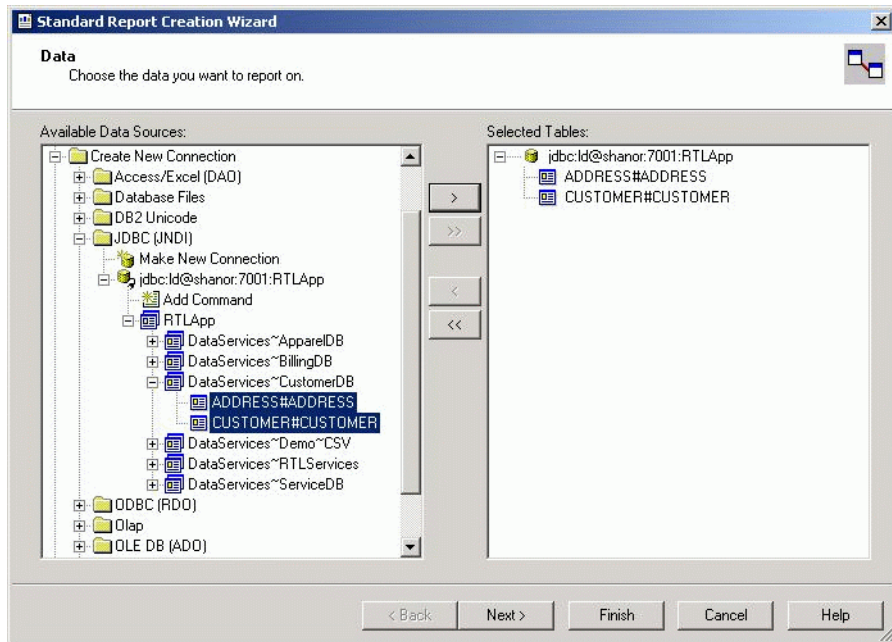
Trusted Connection: ☐

< Back Next > Finish Cancel Help

Note: The Database drop down box is populated with the available catalogs (DSP applications) once you have specified the correct parameters for User ID and, Password, as shown in [Figure 7-13](#).

Clicking the Finish button on the previous screen. This takes you the metadata browser shown in [Figure 7-14](#). The rest of the process is similar to the procedure described in the section “[Crystal Reports 10 - ODBC.](#)”

Figure 7-14 Metadata Browser Window



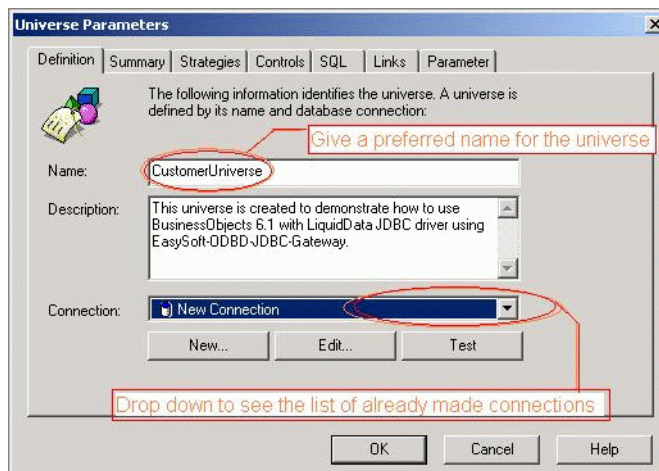
Business Objects 6.1 - ODBC

Business Objects 6.1 allows you to create a Universe and also allows you to generate reports based on the specified Universe. In addition, you can execute pass-through SQL queries against Business Objects that do not need the creation of a Universe.

To generate a report, perform the following steps:

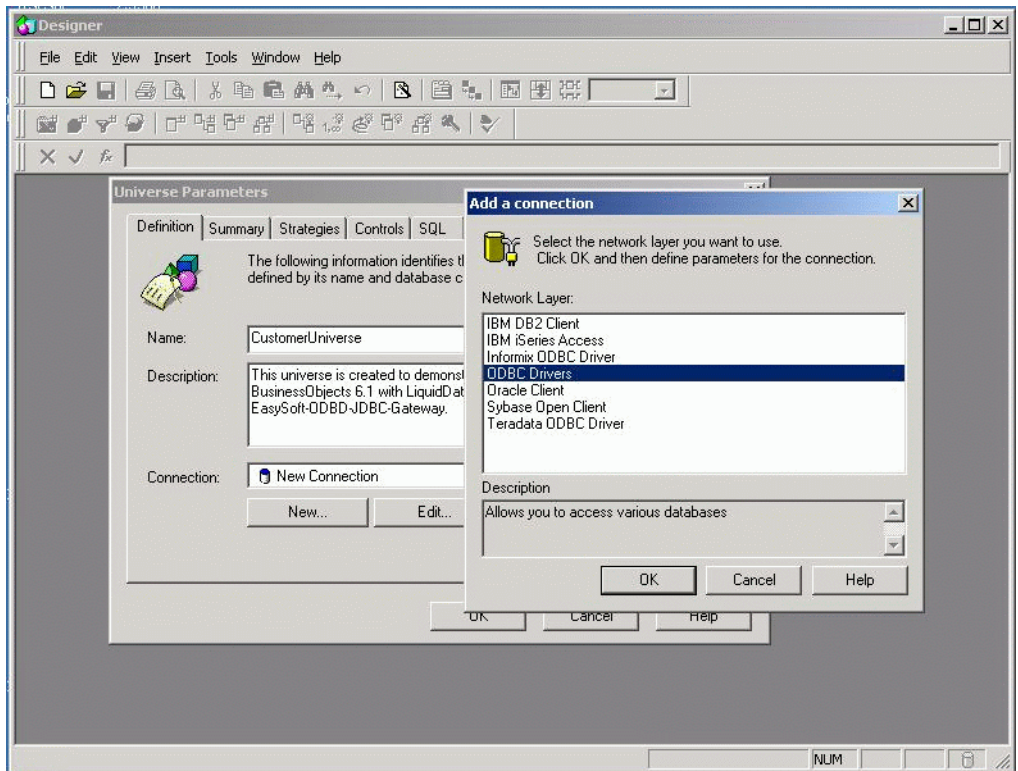
1. Creating a Universe by doing the following:
 - a. Run the Business Objects 6.1 Designer application and click New to create a new universe.
 - b. Fill in a name for your Universe and select the appropriate DSN connection from the drop-down list, as shown in [Figure 7-15](#).

Figure 7-15 Selecting the DSN Connection



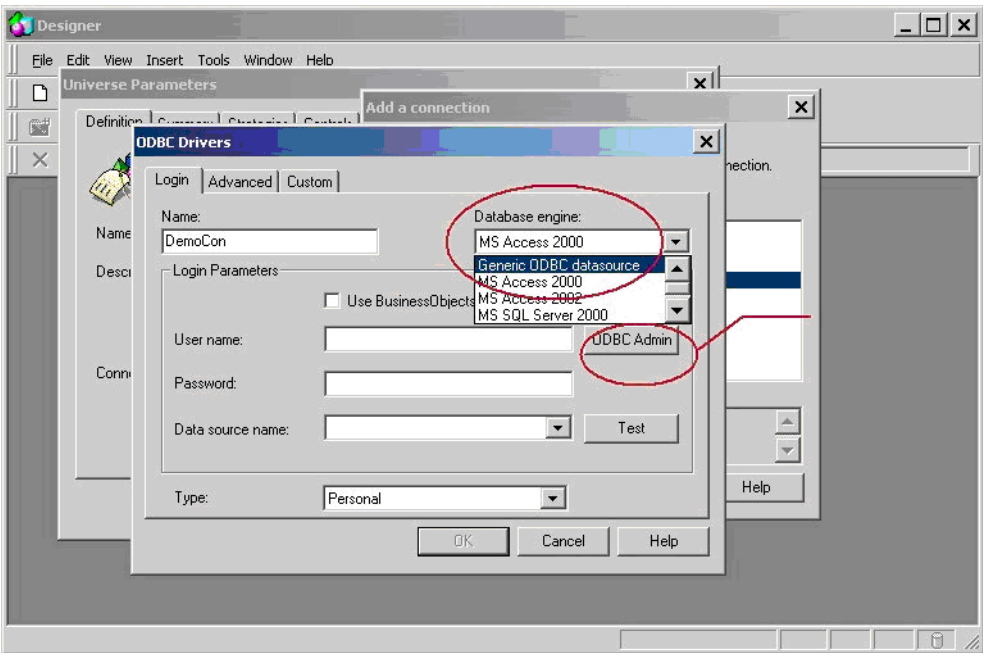
- c. If the DSN you wish doesn't appear in the list (this happens if you are using the application for the first time), use New to create a new connection. Select ODBC Drivers, as shown in [Figure 7-16](#), and click OK.

Figure 7-16 Selecting the ODBC Drivers



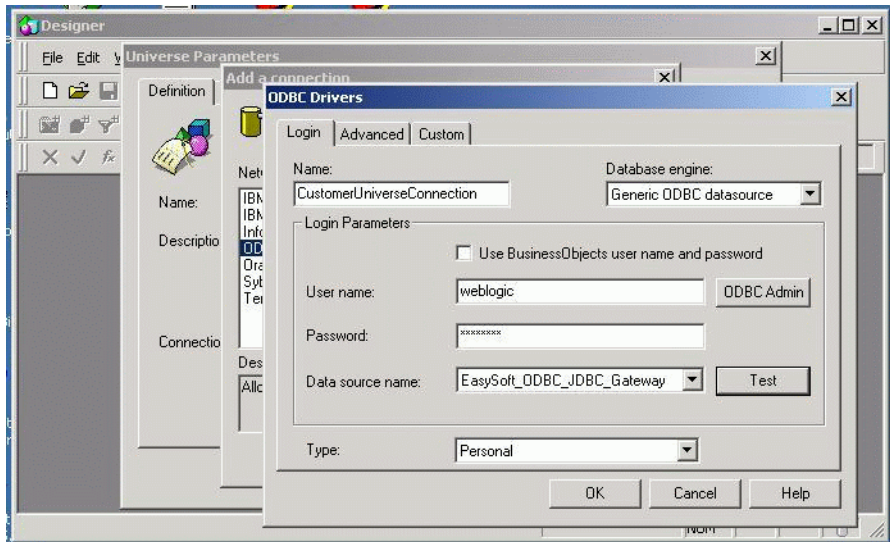
- d. Now select the database engine as a Generic ODBC data source, as shown in [Figure 7-17](#). Use the ODBC Admin button to check if the DSN you wish is already created. For any help creating a DSN using OpenLink or EasySoft please refer to the section ODBC-JDBC bridge of this document.

Figure 7-17 Selecting the Database Engine



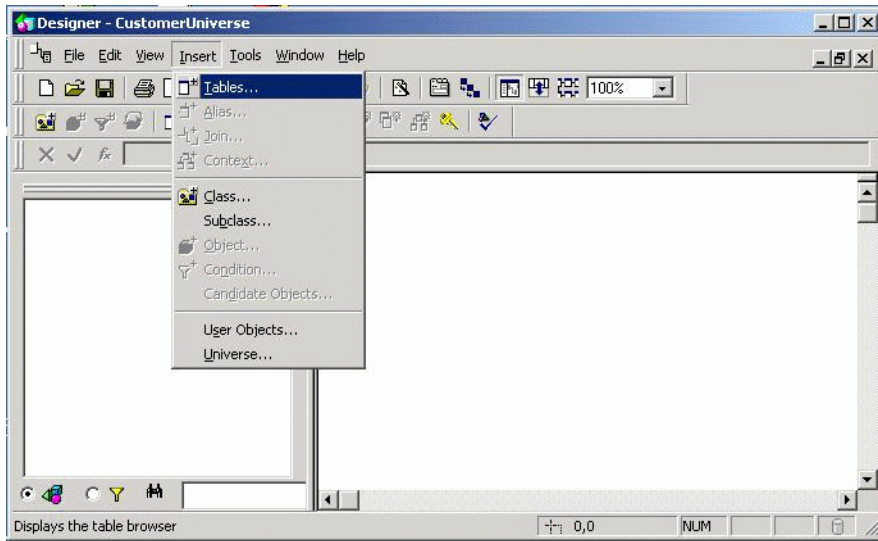
- e. Now select the data source name as shown in [Figure 7-18](#). This would be the name of DSN you wish to connect to. Refer to the picture below. Click OK to get back to the Universe creation window.

Figure 7-18 Selecting the Data Source Name



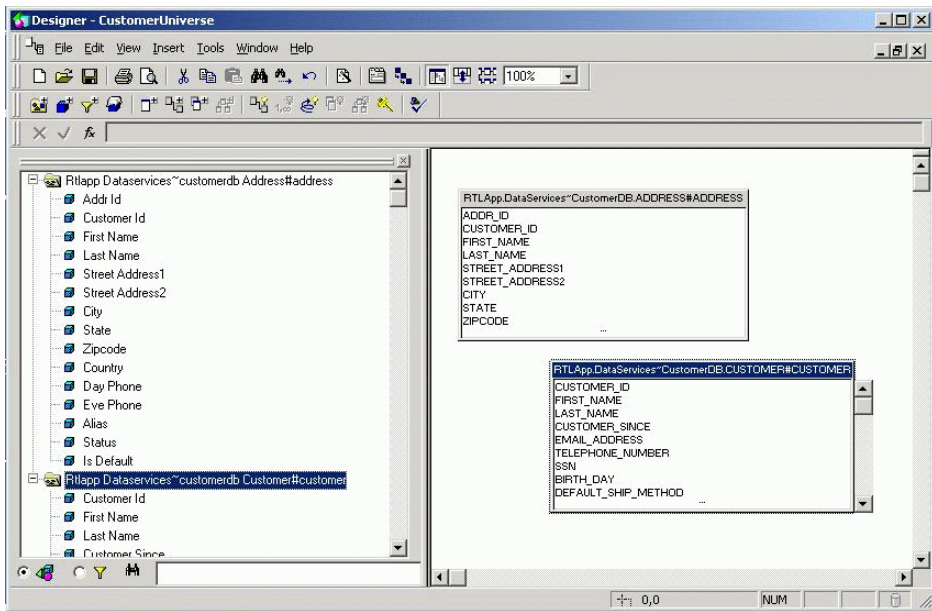
- f. Fill in the other details and click Test to see if the connection is successful. Click OK. You should see a new blank panel, as shown in [Figure 7-19](#).

Figure 7-19 Designer UI Screen



- g. From the Insert menu select Table, as shown in [Figure 7-19](#). Once the list of tables is shown in the Table Browser, double click on the tables you wish to put in the Universe you are creating. You should see a screen similar to that shown in [Figure 7-20](#).

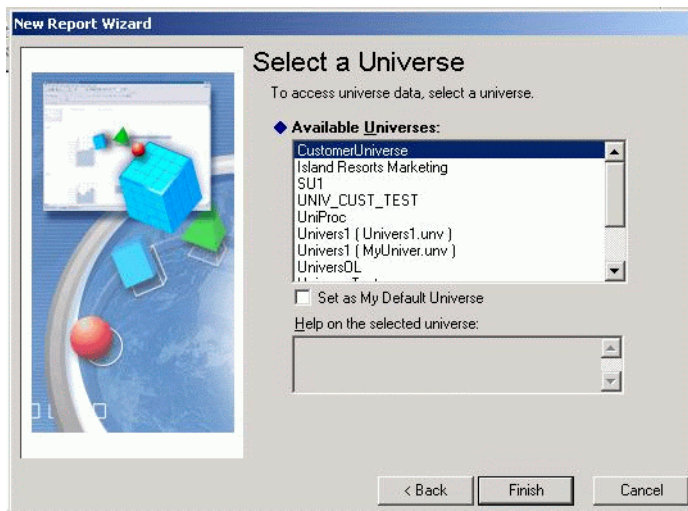
Figure 7-20 Table Browser



- h. Save the Universe and exit.

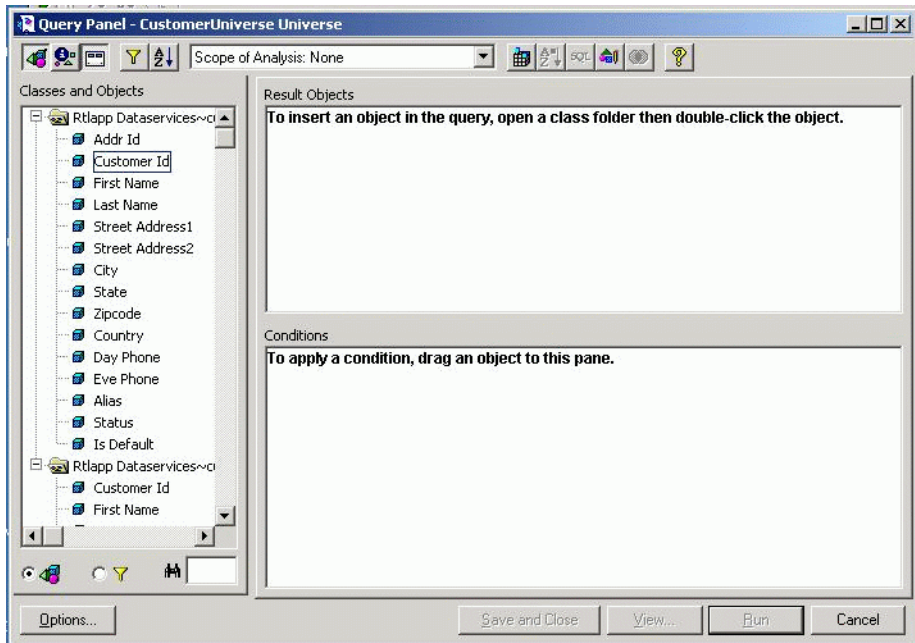
2. Creating a report using the New Report wizard. To create a new report, follow these steps:
 - a. Run the Business Objects application. Click New to open the New Report Wizard. Choose Specify to access data and click Begin. You should see the dialog-box shown in [Figure 7-21](#).

Figure 7-21 Available Universe Dialog Box



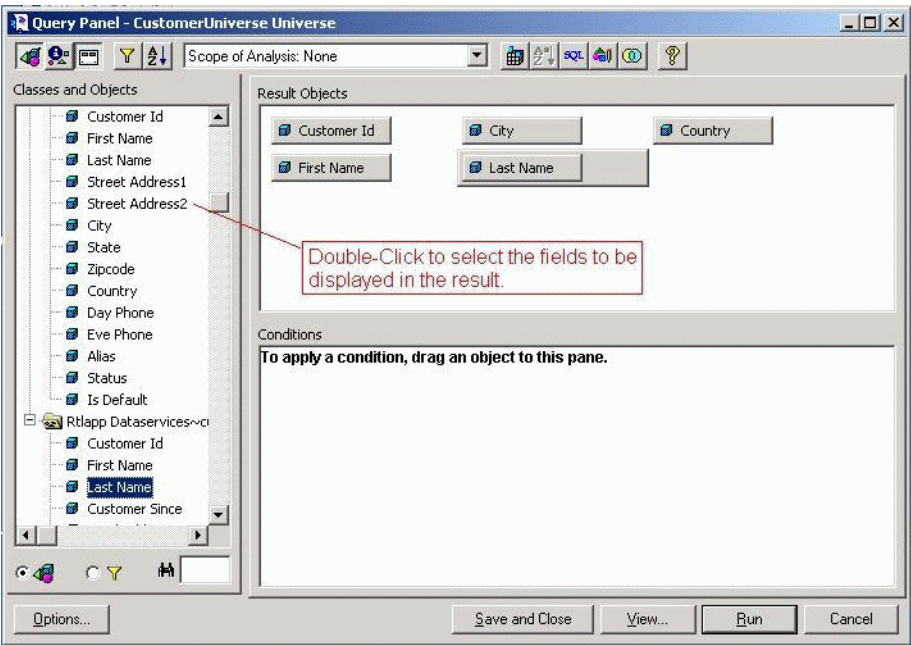
- b. Choose a Universe. Click Next. On the left pane, you should see the tables and their fields (columns) on expansion, as shown in [Figure 7-22](#).

Figure 7-22 Query Panel



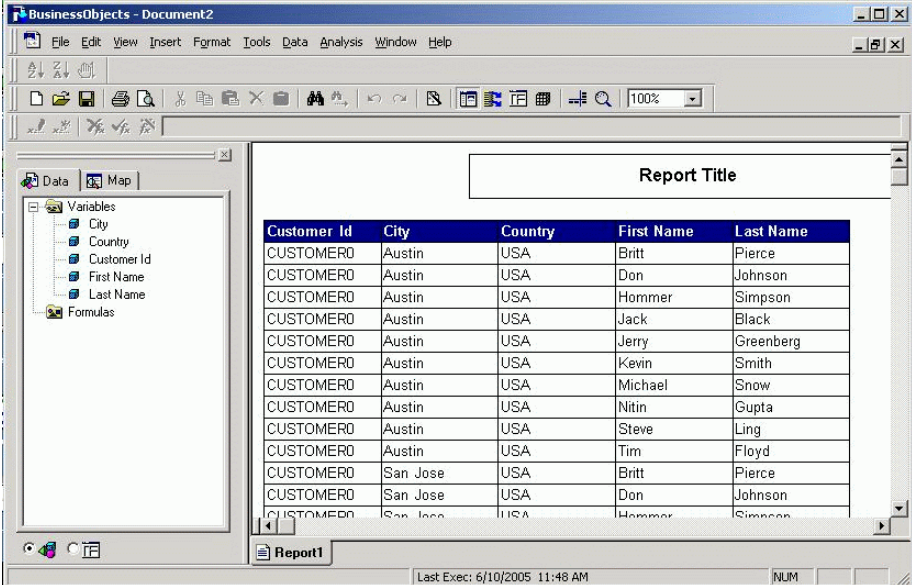
- c. Select the Universe of your choice and click Finish. Double-click a column (table-field) in the left pane to select it in the result, as shown in [Figure 7-23](#).

Figure 7-23 Selecting the Object.



- d. Click Run to execute the query. The result is seen as shown in [Figure 7-24](#).

Figure 7-24 Business Objects Panel.



The screenshot shows the BusinessObjects - Document2 application window. On the left is a 'Data' panel with a tree view containing 'Variables' (City, Country, Customer Id, First Name, Last Name) and 'Formulas'. The main area displays a report titled 'Report Title' containing a table with customer data. The status bar at the bottom indicates 'Last Exec: 6/10/2005 11:46 AM' and 'NUM'.

Customer Id	City	Country	First Name	Last Name
CUSTOMER0	Austin	USA	Britt	Pierce
CUSTOMER0	Austin	USA	Don	Johnson
CUSTOMER0	Austin	USA	Hommer	Simpson
CUSTOMER0	Austin	USA	Jack	Black
CUSTOMER0	Austin	USA	Jerry	Greenberg
CUSTOMER0	Austin	USA	Kevin	Smith
CUSTOMER0	Austin	USA	Michael	Snow
CUSTOMER0	Austin	USA	Nitin	Gupta
CUSTOMER0	Austin	USA	Steve	Ling
CUSTOMER0	Austin	USA	Tim	Floyd
CUSTOMER0	San Jose	USA	Britt	Pierce
CUSTOMER0	San Jose	USA	Don	Johnson
CUSTOMER0	San Jose	USA	Hommer	Simpson

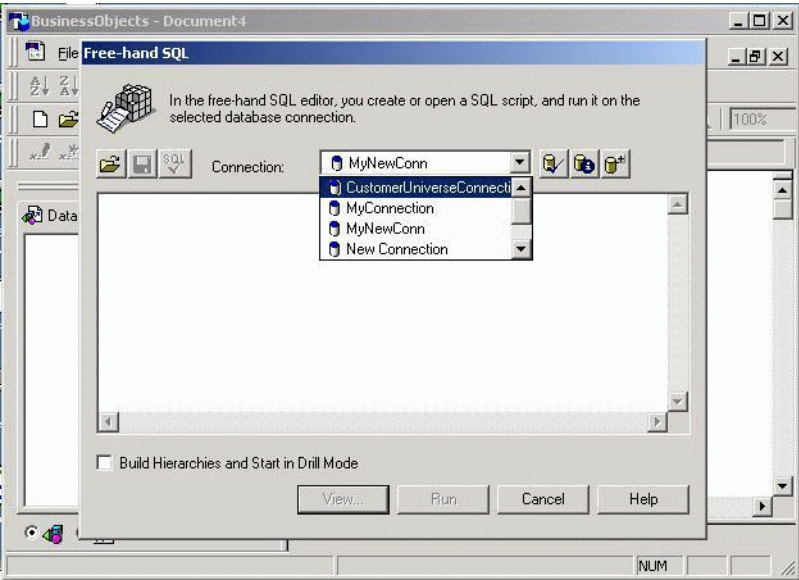
3. You can execute the pass-through queries as follows:
 - a. In the Business Object application, click New to create a new report.
 - b. In the New Report Wizard choose Others instead of Universe as shown in [Figure 7-25](#).

Figure 7-25 Data Access Dialog Box.

- c. Choose Free-hand SQL and click Finish.

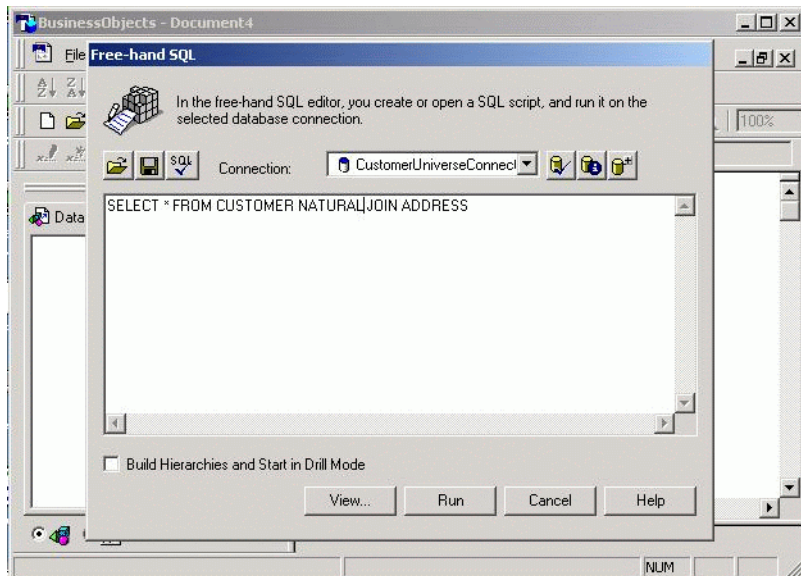
- d. Select the connection you made using Designer 6.1, as shown in [Figure 7-26](#).

Figure 7-26 Free Hand SQL Menu



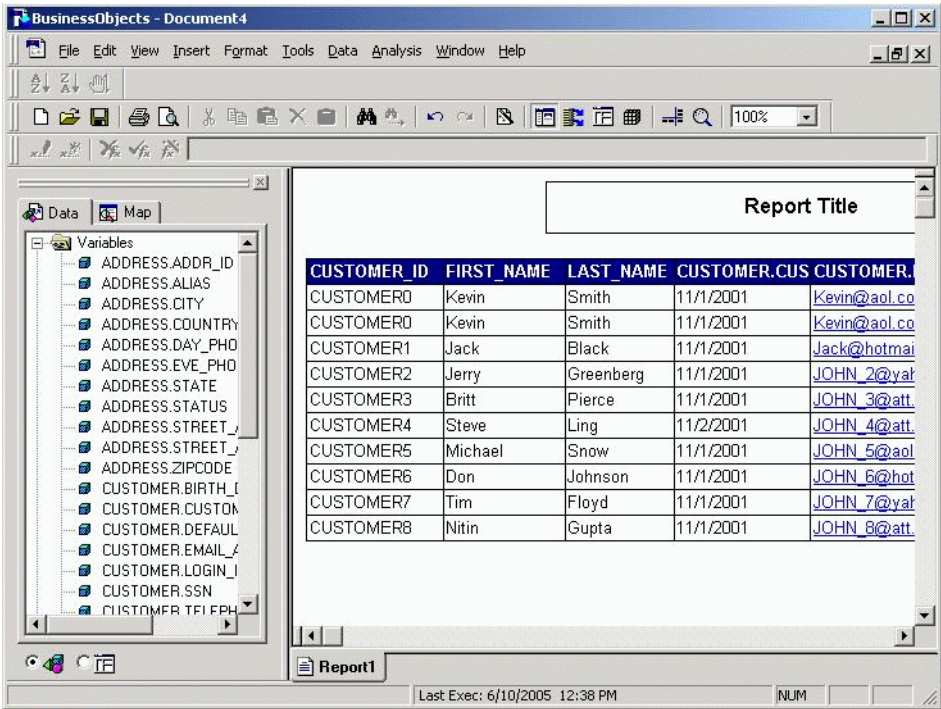
- e. Type in your SQL query and click Run to generate the report, as shown in [Figure 7-27](#).

Figure 7-27 Specifying the SQL Query



f. Click Run. You should see the report shown in [Figure 7-28](#).

Figure 7-28 Business Objects Report



Microsoft Access 2000 - ODBC

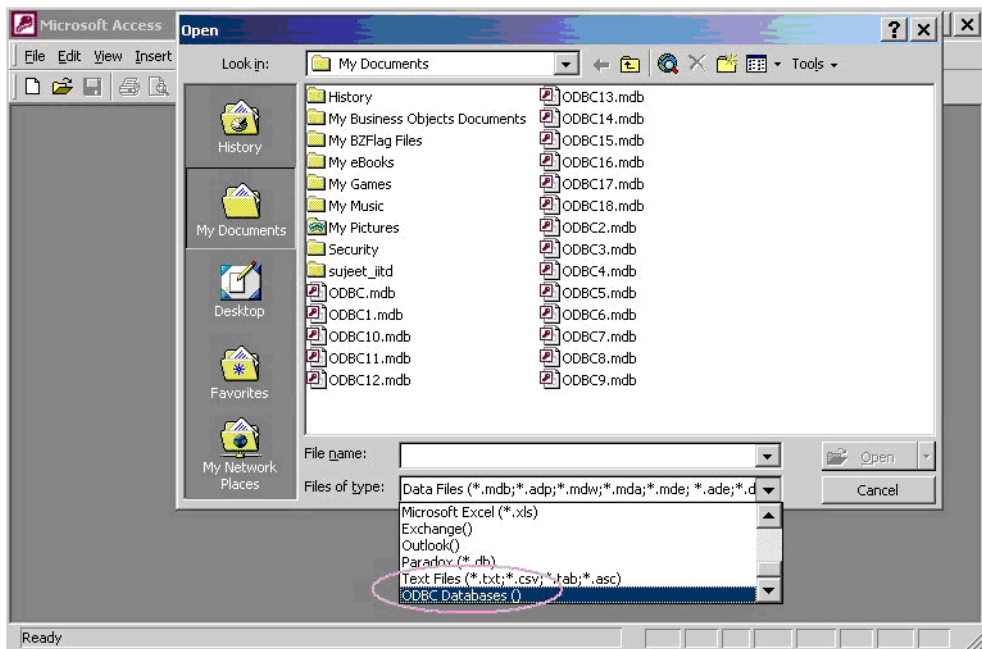
This section describes the procedure for connecting Microsoft Access 2000 to DSP through an ODBC-JDBC bridge.

Note: If you are using Microsoft Access 2000 you should use OpenLink's ODBC- JDBC bridge. The EasySoft bridge does not support Microsoft Access 2000.

To connect Access 2000 to the bridge, perform the following steps.

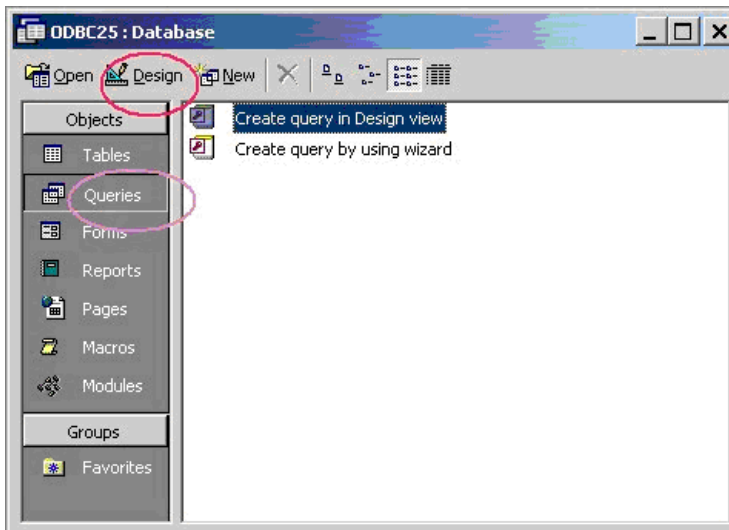
1. Run MS Access, click File Open, then select ODBC Databases as the file type as shown in the [Figure 7-29](#).

Figure 7-29 Selecting the ODBC Database in Access



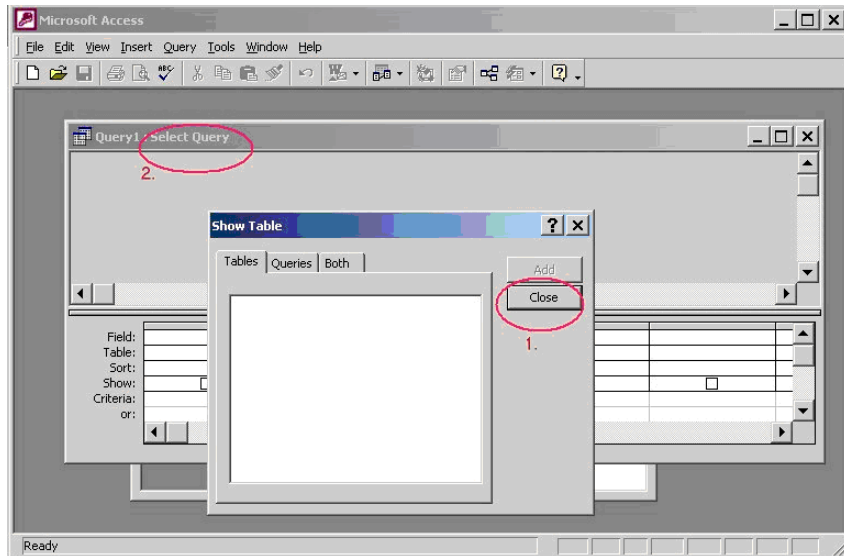
2. Once the dialog Select Data Source pops up, click Cancel to close it. You should see the window shown in [Figure 7-30](#).

Figure 7-30 ODBC23: Database Screen



3. Click Queries, then Design as indicated in [Figure 7-30](#). You should see a screen shown similar to that shown in [Figure 7-31](#).

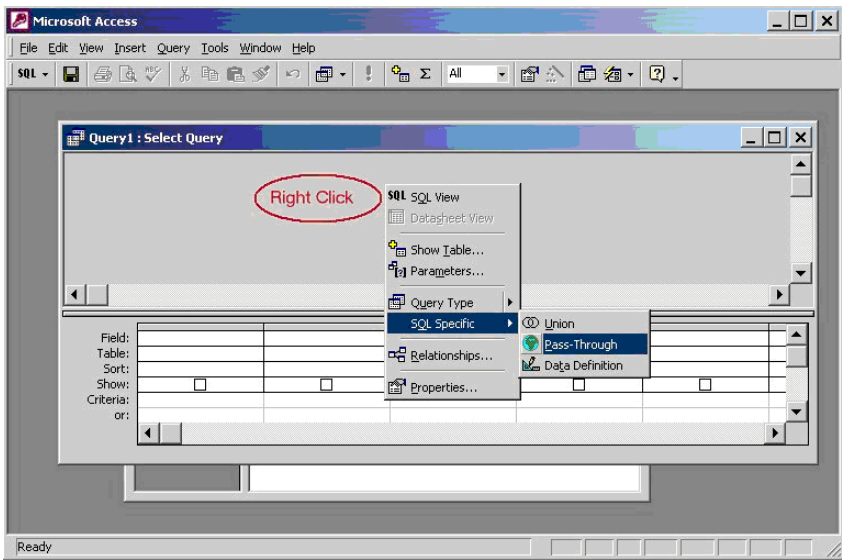
Figure 7-31 Select Query and Show Table Screens



4. Close the Show Table dialog box. You should now be able to see the Select Query dialog.

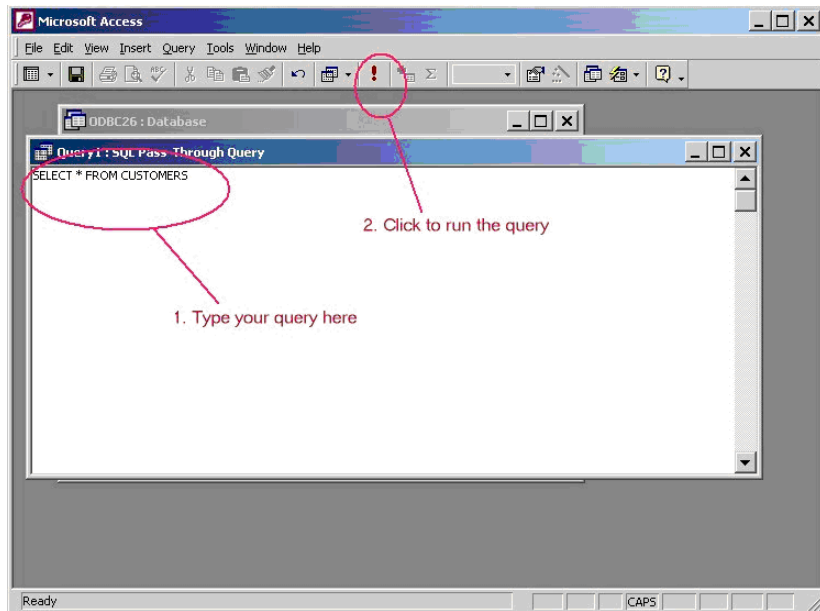
- 5. Right click in the upper pane and select SQL Specific → Pass-Through as indicated in Figure 7-32. This will open an editor.

Figure 7-32 Selecting SQL Specific and Pass Through



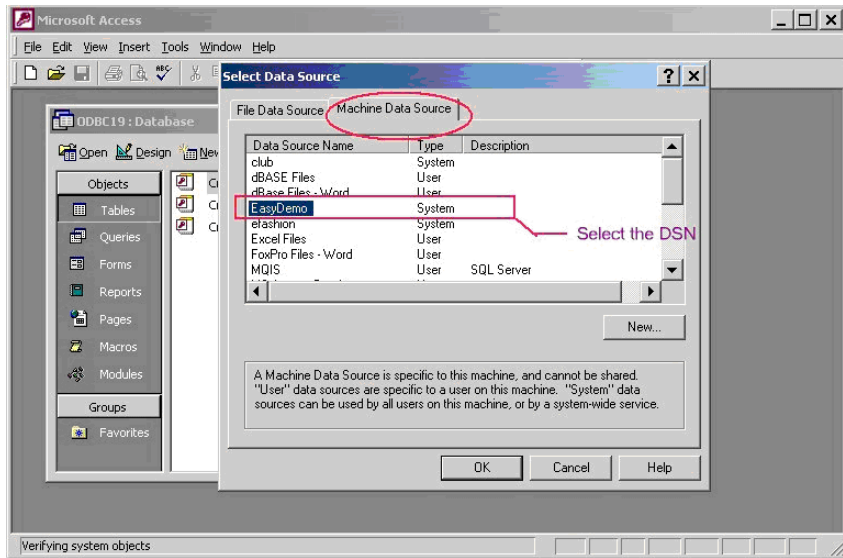
6. Type in your SQL query and click Run, as shown in the [Figure 7-33](#).

Figure 7-33 Running the SQL Query



7. In the dialog that pops up (as shown in [Figure 7-34](#)), move to the tab Machine Data Source and select the appropriate DSN for the database connectivity.

Figure 7-34 Selecting the DSN for the Database



DSP and SQL Type Mappings

When data service information is accessed from a JDBC client, the data is mapped from its XML Schema format to SQL types. The mapping between the types is shown in [Table 7-35](#).

The XML types are defined by `xmlns:xs="http://www.w3.org/2001/XMLSchema"`. The Java types are defined by `java.sql.Types`.

Table 7-35 XML to SQL Type Mapping

XML Type	SQL Types
xs:Boolean	Types.BOOLEAN.
xs:byte	Types.TINYINT
xs:dateTime	Types.TIMESTAMP
xs:date	Types.DATE

Table 7-35 XML to SQL Type Mapping

XML Type	SQL Types
xs:decimal	Types.DECIMAL
xs:double	Types.DOUBLE
xs:duration	Types.TIMESTAMP
xs:float	Types.FLOAT
xs:int	Types.INTEGER
xs:integer	Types.NUMERIC
xs:long	Types.BIGINT
xs:short	Types.SMALLINT
xs:string	Types.VARCHAR
xs:time	Types.TIME

SQL-92 Support

This section outlines the SQL-92 support in the Data Services Platform JDBC driver.

Supported Features

The Data Services Platform JDBC driver supports many standard SQL-92 features. In particular, supported features include:

- Only SELECT construct is supported. Inserts, updates, and deletes are not supported.
- SELECT clause with:
 - DISTINCT and ALL
 - Scalar expressions and functions, CASE statements, CAST, string and date literals, column wildcards.
- Projections (sub-queries) within the select clause are not supported.
- FROM clause with:

- Basic table names
 - Sub-queries
 - Joins
 - Set operations
- GROUP BY clause
- HAVING clause
- WHERE clause with:
 - Predicate expressions (arithmetic operators, functions, CASE statements)
 - Predicates involving non-correlated and correlated sub-queries
 - EXISTS
 - BETWEEN
 - LIKE
 - NULLIF
 - COALESCE
 - UNIQUE
 - IS NULL, IS NOT NULL, IS TRUE, IS FALSE
 - ALL, SOME, ANDY
- Joins of the following type:
- Cross joins, inner joins, and union joins
- Natural joins and joins with ON and USING
- Left, right, and full outer joins
- Set operations:
 - UNION
 - INTERSECT
 - MINUS
- Parameterized queries (with standard SQL-92 notation)

- ORDER by clause
- Functions:
 - STR
 - CONCAT
 - CURRENT_TIME
 - CURRENT_DATE
 - CURRENT_TIMESTAMP
 - ROUND
 - FLOOR
 - LOWER
 - UPPER
 - SUBSTRING
 - CASTTODATE
 - CASTTOTIME
 - COUNT
 - AVG
 - SUM
 - MIN
 - MAX
 - EXTRACT
 - TRIM

The Data Services Platform JDBC driver implements the following interfaces from `java.sql` package specified in JDK 1.4x:

- `java.sql.Connection`
- `java.sql.CallableStatement`
- `java.sql.DatabaseMetaData`
- `java.sql.ParameterMetaData`
- `java.sql.PreparedStatement`

- `java.sql.ResultSet`
- `java.sql.ResultSetMetaData`
- `java.sql.Statement`

Limitations

The following limitations are known to exist in the Data Services Platform JDBC driver:

- Each connection points to only one DSP application.
- An XML Schema name can contain special characters that are illegal for database schema names (such as "/" and "."). The Data Services Platform JDBC driver translates the characters to legal characters ("~" and "^", respectively).

The following table notes additional limitations that apply to SQL language features.

Unsupported Feature	Comments	Example
OVERLAPS	Intervals not supported	WHERE (, ,) OVERLAPS (, ,)
range-variable-comma-list	The <code>table_name</code> can have an alias, but you cannot specify the <code>colmn_name_alias_list</code> within it.	SELECT ID, NM, CT FROM STAFF AS (ID, NM, GD, CT);
Assignment in select	Not supported.	SELECT MYCOL = 2 FROM VTABLE WHERE COL4 IS NULL
The CORRESPONDING BY construct with the set-Operations(UNION, INTERSECT and EXCEPT)	<p>The SQL-92 specified default column ordering in the set operations is supported.</p> <p>Both the table-expressions (the operands of the set-operator) must conform to the same relational schema.</p>	<p>(SELECT NAME, CITY FROM CUSTOMER1) UNION CORRESPONDING BY (CITY, NAME) (SELECT CITY, NAME FROM CUSTOMER2)</p> <p>The supported query is: (SELECT NAME, CITY FROM CUSTOMER1) UNION (SELECT NAME, CITY FROM CUSTOMER2)</p>

Unsupported Feature	Comments	Example
"...table1 UNION table2..."	<p>Not supported. Also not supported are set operations between tables in a FROM clause, except through a sub-query.</p> <p>The TABLE keyword is not supported.</p>	<p>SELECT * FROM TABLE CUSTOMER1 UNION TABLE CUSTOMER2</p> <p>Where TABLE is a keyword not supported by the LDJDBC SQL interface.</p> <p>The supported version is:</p> <p>SELECT * FROM (SELECT * FROM CUSTOMER1 UNION SELECT * FROM CUSTOMER2) T1</p> <p>Other supported UNION constructs:</p> <p>SELECT * FROM CUSTOMER1 UNION SELECT * FROM CUSTOMER2</p> <p>SELECT * FROM CUSTOMER1 UNION (SELECT * FROM CUSTOMER2 UNION SELECT * FROM CUSTOMER3)</p>
SELECT-query within the SELECT clause	Not supported.	SELECT A, (SELECT B FROM C) FROM... WHERE...

Advanced Topics

This chapter provides information on miscellaneous topics related to client programming with BEA AquaLogic Data Services Platform (DSP). It covers the following topics:

- [Applying Filter Data Service Results](#)
- [Ordering and Truncating Data Service Results](#)
- [Consuming Large Result Sets \(Streaming API and Writing Results To a File\)](#)
- [Using Ad Hoc Queries](#)
- [Transaction Considerations](#)
- [Setting Up Data Source Aliases for Relational Sources Accessed by DSP](#)
- [Setting Up Data Source Aliases for Relational Sources Accessed by DSP](#)

Applying Filter Data Service Results

The Filter API enables client applications to apply filtering conditions to the information returned by data service functions. In a sense, filtering allows client applications to extend a data service interface by allowing them to specify more about how data objects are to be instantiated and returned by functions.

The Filter API alleviates data service designers from having to anticipate every possible data view that their clients may require and to implement a data service function for each view. Instead, the designer may choose to specify a broader, more generic interface for accessing a business entity and allow client applications to control views as desired through filters.

Objects in the function return set that do not meet the condition are blocked from the results. (The evaluation occurs at the server, so objects that are filtered are not passed over the network. Often they are not even retrieved from the underlying sources.) A filter is similar to a WHERE clause in an XQuery or SQL statement—it applies conditions to a possible result set. You can have multiple filter conditions using AND and OR operators.

Note: The Javadoc that describes the Filter API is available at:

<http://e-docs.bea.com/liquiddata/docs85/ldapiJavadoc/index.html>

Using Filters

Filtering capabilities are available to mediator and data service control client applications. You use filter conditions to specify what data you want returned, sort the data, or limit the number of records returned. To use filters in a mediator client application, import the appropriate package and use the supplied interfaces for creating and applying filter conditions. Data service control clients get the interface automatically. When a function is added to a control, a corresponding *"WithFilter"* function is added as well.

The filter package is named as follows:

```
com.bea.ld.filter.FilterXQuery;
```

To use a filter, perform the following steps:

1. Create an FilterXQuery object, such as:

```
FilterXQuery myFilter = new FilterXQuery();
```

2. Add a filter condition to the object u8d
3. od. With this function you can specify what node your filter condition will apply to and specify the number of records to be returned based on a limit; for example, you can specify the filter will apply to customer orders where only orders with an amount over a specified value will be returned.

The `addFilter()` method has the following signature:

```
public void addFilter(java.lang.String appliesTo,  
                     java.lang.String field,  
                     java.lang.String operator,  
                     java.lang.String value,  
                     java.lang.Boolean everyChild)
```

The method takes the following arguments:

- `appliesTo` indicates the node that filtering affects. That is, if a node specified by the field argument does not meet the condition, `appliesTo` nodes are filtered out.
- `field` is the node against which the filtering condition is tested.
- `operator` and `value` together compose the condition statement. The `operator` parameter specifies the type of comparison to be made against the specified `value`. The section [Filter Operators](#) describes the available operators.
- `everyChild` is an optional parameter. It is set to *false* by default. Specifying true for this parameter indicates that only those child elements that meet the filter criteria will be returned. For example, by specifying an operator of `GREATER_THAN` (or `>`) and a value of 1000, only records for customers where *all* orders are over 1000 will be returned. A customer that has an order amount less than 1000 will not be returned, although other order amounts might be greater than 1000.

The following is an example of an add filter method where those orders with an order amount greater than 1000 will be returned (note that `everyChild` is not specified, so order amounts below 1000 will be returned):

```
myFilter.addFilter("CUSTOMERS/CUSTOMER/ORDER",
                  "CUSTOMERS/CUSTOMER/ORDER/ORDER_AMOUNT",
                  ">",
                  "1000");
```

4. Use the Mediator API call `setFilterCondition()` to add the filter to a data service, passing the `FilterXQuery` instance as an argument. For example,

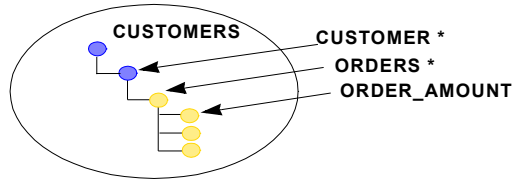
```
CUSTOMER custDS = CUSTOMER.getInstance(ctx, "RTLApp");
custDS.setFilterCondition(myFilter);
```

5. Invoke the data service function. (For more information on invoking data service functions, see [Chapter 4, “Accessing Data Services from Java Clients.”](#))

Specifying Filter Effects

If a filter condition applied to a specified element value resolves to false, an element is not included in the result set. The element that is filtered out is specified as the first argument to the `addFilter()` function.

The effects of a filter can vary, depending on the desired results. For example, consider the `CUSTOMERS` data object shown in [Figure 8-1](#). It contains several complex elements (`CUSTOMER` and `ORDERS`) and several simple elements, including `ORDER_AMOUNT`. You can apply a filter to any elements in this hierarchy.

Figure 8-1 Nested Value Filtering

In general, with nested XML data, a condition such as “CUSTOMER/ORDER/ORDER_AMOUNT > 1000” can affect what objects are returned in several ways. For example, it can cause all CUSTOMER objects to be returned, but filter ORDERS that have an amount less than 1000.

Alternatively, it can cause only CUSTOMER objects to be returned that have at least one large order, and all ORDER objects are returned for every CUSTOMER. Further, it can cause only CUSTOMER objects to be returned for which every ORDER is greater than 1000. For example,

```
XQueryFilter myFilter = new XQueryFilter();
myFilter.addFilter( "CUSTOMERS/CUSTOMER",
                   "CUSTOMERS/CUSTOMER/ORDER/ORDER_AMOUNT",
                   FilterXQuery.GREATER_THAN, "1000", true );
```

Note that in the optional fourth parameter `everyChild = true`, by default this attribute is false. By setting this parameter to true, only those CUSTOMER objects for which *every* ORDER is greater than 1000 will be returned.

The following examples show how filters can be applied in several different ways:

- Returns all CUSTOMER objects but only their large ORDER objects:

```
XQueryFilter myFilter = new XQueryFilter();
Filter f1 = myFilter.createFilter(
    "CUSTOMERS/CUSTOMER/ORDER/ORDER_AMOUNT",
    FilterXQuery.GREATER_THAN, "1000");
myFilter.addFilter("CUSTOMERS/CUSTOMER/ORDER", f1);
```

- Returns only CUSTOMER objects that have at least one large order but view *all* ORDER objects for such CUSTOMER:

```
XQueryFilter myFilter = new XQueryFilter();
myFilter.addFilter("CUSTOMERS/CUSTOMER",
    "CUSTOMERS/CUSTOMER/ORDER/ORDER_AMOUNT",
    FilterXQuery.GREATER_THAN, "1000");
```

- Returns only CUSTOMER objects that have at least one large order and return *only large* ORDER objects:

```
XQueryFilter myFilter = new XQueryFilter();
myFilter.addFilter("CUSTOMERS/CUSTOMER",
    "CUSTOMERS/CUSTOMER/ORDER/ORDER_AMOUNT",
    FilterXQuery.GREATER_THAN, "1000");
myFilter.addFilter("CUSTOMERS/CUSTOMER/ORDER",
    "CUSTOMERS/CUSTOMER/ORDER/ORDER_AMOUNT",
    FilterXQuery.GREATER_THAN, "1000");
```

The last example is a compound filter; that is, a filter with two conditions.

Filter Operators

You can use the following operators in filters:

- LESS_THAN (<)
- GREATER_THAN (>)
- LESS_THAN_EQUAL (<=)
- GREATER_THAN_EQUAL (>=)
- EQUAL (=)
- NOT_EQUAL (!=)
- matches (for string equality)
- sql-like (tests whether a string contains a specified pattern)

These compound operators can be applied to more than one filter:

- OR
- NOT
- AND

The following example uses the AND operator to apply a combination of filters to a result set, given a data service instance `customerDS`:

```
FilterXQuery myFilter = new FilterXQuery();
Filter f1 = myFilter.createFilter("CUSTOMER_PROFILE/ADDRESS/ISDEFAULT",
                                FilterXQuery.NOT_EQUAL, "0");
Filter f2 = myFilter.createFilter("CUSTOMER/ADDRESS/STATUS",
                                FilterXQuery.EQUAL,
                                "\"ACTIVE\"");
Filter f3 = myFilter.createFilter(f1, f2, FilterXQuery.AND);
Customer customerDS = Customer.getInstance(ctx, "RTLApp");
CustomerDS.setFilterCondition(myFilter);
```

Ordering and Truncating Data Service Results

An ordering condition is a type of filter that lets you specify the order in which results are returned from a data service. They allow you to arrange results in either ascending or descending order based on the value of a specified property.

The ordering methods are in the `FilterXQuery` class. The following example shows how to use ordering. It gets a list of customer profiles in ascending order based on the dates the person became a customer.

```
FilterXQuery myFilter = new FilterXQuery();
myFilter.addOrderBy("CUSTOMER_PROFILE", "CustomerSince" ,
                  FilterXQuery.ASCENDING);
ds.setFilterCondition(myFilter);
DataObject objArrayOfCust =
    (DataObject) ds.invoke("getCustomer", null);
```

Similarly, you can set the maximum number of results that can be returned from a function. The `setLimit()` function limits the number of elements in an array element to the specified number. And on a repeating node, it makes sense to specify a limit on the results to be returned. (Setting the limits on non-repeating nodes does not truncate the results.)

The following shows how to use the `setLimit()` method. It limits the number of active address in the result set (filtering out active addresses) to 10 given a data service instance `ds`:

```
FilterXQuery myFilter = new FilterXQuery();
Filter f2 = myFilter.createFilter("CUSTOMER_PROFILE/ADDRESS",
                                FilterXQuery.EQUAL, "\"INACTIVE\"");
myFilter.addFilter("CUSTOMER_PROFILE", f2);
```

```
myFilter.setLimit("CUSTOMER_PROFILE", "10");
ds.setFilterCondition(myFilter);
```

Consuming Large Result Sets (Streaming API and Writing Results To a File)

This section discusses further programming topics related to client programming with the Data Service Mediator API. It includes the following topics:

- [Using the Streaming Interface](#)
- [Writing Data Service Function Results to a File](#)

Using the Streaming Interface

When a function in the standard data service interface is called, the requested data is first materialized in the system memory of the server machine. If the function is intended to return a large amount of data, in-memory materialization of the data may be impractical. This may be the case, for example, for administrative functions that generate "inventory reports" of the data exposed by DSP. For such cases, DSP can serve information as an output stream.

DSP leverages the WebLogic XML Streaming API for its streaming interface. The WebLogic Streaming API is similar to the standard SAX (Streaming API for XML) interface. However, instead of contending with the complexity of the event handlers used by SAX, the WebLogic Streaming API lets you use stream-based (or *pull-based*) handling of XML documents in which you step through the data object elements. As such, the WebLogic Streaming API affords more control than the SAX interface, in that the consuming application initiates events, such as iterating over attributes or skipping ahead to the next element, instead of reacting to them.

Note: For more information on the WebLogic Streaming API, see "Using the WebLogic XML Streaming API" at http://e-docs.bea.com/wls/docs81/xml/xml_stream.html.

It is important to note that although serving data as a stream relieves the server from having to materialize large objects in memory, the server is using the request thread while output streaming occurs. This can tie up a thread for quite a while and affect the server's ability to respond to other service requests in a timely fashion. The streaming API is intended for use only for administrative sorts of uses, and should be avoided except at off-peak times or in non-production environments.

Data Services Platform streaming API can only be invoked from Java code that is part of the same application from which you are streaming data. That is, the client code needs to be in the same EAR application file in which the data services are hosted.

You can get DSP information as a stream by using either an ad hoc or an untyped data service interface.

Note: Streaming is not supported through static interfaces.

The streaming interface is in these classes in the `com.bea.ld.dsmediator.client` package:

- `StreamingDataService`
- `StreamingPreparedExpression`

Using these interfaces is very similar to using their SDO mediator client API equivalents. However, instead of a document object, they return data as an `XMLInputStream`. For functions that take complex elements (possibly with a large amount of data) as input parameters, `XMLInputStream` is supported as an input argument as well. The following is an example:

```
StreamingDataService ds = StreamingDataServiceFactory.getInstance(
    context,
    "ld:DataServices/RTLServices/Customer");
XMLInputStream stream = ds.invoke("getCustomerByCustID", "CUSTOMER0");
```

The previous example shows the dynamic streaming interface. The following example uses an ad hoc query:

```
String adhocQuery =
    "declare namespace ns0=\"ld:DataServices/RTLServices/Customer\";\n" +
    "declare variable $cust_id as xs:string external;\n" +
    "for $customer in ns0:getCustomerByCustID($cust_id)\n" +
    "return\n" +
    "    $customer\n";
StreamingPreparedExpression expr =
    DataServiceFactory.prepareExpression(context, adhocQuery);
```

If you have external variables in the query string (adhocQuery in the above example), you will also need to do the following:

```
expr.bindString("$cust_id", "CUSOMER0");
XMLInputStream xml = expr.executeQuery();
```

Note: For more information on using the dynamic and ad hoc interfaces, see [“Using the Dynamic Data Service Interface”](#) in Chapter 4, [“Accessing Data Services from Java Clients”](#) and [“Using Ad Hoc Queries”](#) on page 8-11. Also, a Javadoc that contains descriptions of the `StreamingDataService` interface is available in the Javadoc that describes the Filter API is available at:

<http://e-docs.bea.com/liquiddata/docs85/ldapiJavadoc/index.html>

Listing 8-1 shows an example of a method that reads the XML input stream. This method uses an attribute iterator to print out attributes and namespaces in an XML event and throws an XMLStream exception if an error occurs.

Listing 8-1 Sample Streaming Application

```
import weblogic.xml.stream.Attribute;
import weblogic.xml.stream.AttributeIterator;
import weblogic.xml.stream.ChangePrefixMapping;
import weblogic.xml.stream.CharacterData;
import weblogic.xml.stream.XMLEvent;
import weblogic.xml.stream.EndDocument;
import weblogic.xml.stream.EndElement;
import weblogic.xml.stream.EntityReference;
import weblogic.xml.stream.Space;
import weblogic.xml.stream.StartDocument;
import weblogic.xml.stream.XMLInputStream;
import weblogic.xml.stream.XMLInputStreamFactory;
import weblogic.xml.stream.XMLName;
import weblogic.xml.stream.XMLStreamException;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class ComplexParse {

    public void parse(XMLEvent event) throws XMLStreamException
    {
        switch(event.getType()) {
            case XMLEvent.START_ELEMENT:
                StartElement startElement = (StartElement) event;
                System.out.print("<" + startElement.getName().getQualifiedName() );
                AttributeIterator attributes = startElement.getAttributesAndNamespaces();
                while(attributes.hasNext()) {
                    Attribute attribute = attributes.next();
                    System.out.print(" " + attribute.getName().getQualifiedName() +
                        "=" + attribute.getValue() + "");
                }
            }
        }
    }
}
```

```
System.out.print(">");
    break;
case XMLEvent.END_ELEMENT:
    System.out.print("</" + event.getName().getQualifiedName() + ">");
    break;
case XMLEvent.SPACE:
case XMLEvent.CHARACTER_DATA:
    CharacterData characterData = (CharacterData) event;
    System.out.print(characterData.getContent());
    break;
case XMLEvent.COMMENT:
    // Print comment
    break;
case XMLEvent.PROCESSING_INSTRUCTION:
    // Print ProcessingInstruction
    break;
case XMLEvent.START_DOCUMENT:
    // Print StartDocument
    break;
case XMLEvent.END_DOCUMENT:
    // Print EndDocument
    break;
case XMLEvent.START_PREFIX_MAPPING:
    // Print StartPrefixMapping
    break;
case XMLEvent.END_PREFIX_MAPPING:
    // Print EndPrefixMapping
    break;
case XMLEvent.CHANGE_PREFIX_MAPPING:
    // Print ChangePrefixMapping
    break;
case XMLEvent.ENTITY_REFERENCE:
    // Print EntityReference
    break;
case XMLEvent.NULL_ELEMENT:
    throw new XMLStreamException("Attempt to write a null event.");
default:
    throw new XMLStreamException("Attempt to write unknown event["
```

```

        +event.getType()+" ] ");
    }
}

```

Writing Data Service Function Results to a File

You can write serialized results of a data service function to a file using a `WriteOutputToFile` method. Such a function is generated automatically for each function defined in the data service. For security reasons it writes only to a file on the server's file system.

These functions provide services that are similar to streaming APIs. They are intended for creating reports or an inventory of data service information. However, the `writeOutputToFile` method can be invoked from a remote mediator API (in contrast with the streaming API described in [“Using the Streaming Interface” on page 8-7](#)).

The following example shows how to write to a file from the untyped interface.

```

StreamingDataService sds =
    DataServiceFactory.newStreamingDataService(
        context, "RTLApp", "ld:DataServices/RTLServices/Customer");
sds.writeOutputToFile("getCustomer", null, "streamContent.txt");
sds.closeStream();

```

Note: No attempt to create folders is made. In the above example, if you want to write data inside a folder named `myData` that folder should be present in the server domain root prior to the write operation.

Using Ad Hoc Queries

An ad hoc query is an XQuery function that is not stored in a data service, but is instead defined by the client application. You can use an ad hoc query to execute any XQuery function, possibly against a data source defined on a remote DSP server or even with no back-end data source at all.

To use ad hoc queries, use the `PreparedExpression` interface of the Mediator API. The `PreparedExpression` interface is similar to the `PreparedStatement` interface of JDBC. You create the prepared expression by passing the function body as a string in the constructor (along with the JNDI server context and the application name), then call the `executeQuery()` method on the prepared expression as follows:

```

PreparedExpression adHocQuery =
    DataServiceFactory.prepareExpression(
        context, "RTLApp", "<CustomerID>CUSTOMER0</CustomerID>");
XmlObject adHocResult = adHocQuery.executeQuery();

```

The above sample merely returns an XML node named `CUSTOMER_ID`. A more useful ad hoc query, however, would typically invoke data service functions and process their results in some way.

To invoke data service functions in ad hoc queries, the query needs to import the namespace of the data service to be used. It can then invoke the data service's function. The following returns the results of a data service function named `getCustomers()`, which is in the name space `"Id:DataServices/RTLServices/Customer"`:

```

String queryStr =
    "declare namespace ns0=\"Id:DataServices/RTLServices/Customer\";" +
    "<Results>" +
    "  { for $customer_profile in ns0:getCustomer() " +
    "    return $customer_profile }" +
    "</Results>";

PreparedExpression adHocQuery =
    DataServiceFactory.prepareExpression(context, "RTLApp", queryStr );
XmlObject objResult = (XmlObject) adHocQuery.executeQuery();

```

DSP passes information back to the ad hoc query caller as an `XmlObject` object. Because all typed data objects implement the `XmlObject` interface, ad hoc query results that conform to a deployed schema can be downcast to the type of the schema.

For data service functions that return arrays, you must create a root element in the ad hoc query as a container for the array because an `XmlObject` must have a single root type. For example, the data service function `getCustomer()` invoked in the code sample above returns an array of `CUSTOMER_PROFILE` elements; therefore, the ad hoc query specifies a container `<Results>` to hold the returned array.

Security policies defined for a data service apply to the data service calls in an ad hoc query as well. Appropriate credentials must be passed when creating the JNDI initial context in an ad hoc query that uses secured resources. For more information, see [“Getting a WebLogic JNDI Context for DSP,”](#) in [Chapter 4, “Accessing Data Services from Java Clients.”](#)

Like the `PreparedStatement` interface of JDBC, you can bind variables dynamically in ad hoc query expressions. The ad hoc query `PreparedExpression` interface includes a number of methods for binding values of various types, named in the form `bindType`.

You bind a variable to a value by specifying the variable as a qualified name (*qname*) and passing the value in the bind method as follows:

```
PreparedExpression adHocQuery = DataServiceFactory.preparedExpression(
    context, "RTLApp",
    "declare variable $i as xs:int external;
    <result><zip>{fn:data($i)}</zip></result>");
adHocQuery.bindInt(new QName("i"), 94133);
XmlObject adHocResult = adHocQuery.executeQuery();
```

QName stands for qualified name. For more information on qnames, see:

<http://www.w3.org/TR/xmlschema-2/#QName>

Listing 8-2 shows a complete ad hoc query example, using the preparedExpression interface and qualified names to pass values in bind methods.

Listing 8-2 Sample Ad Hoc Query

```
import com.bea.ld.dsmediator.client.DataServiceFactory;
import com.bea.ld.dsmediator.client.PreparedExpression;
import com.bea.xml.XmlObject;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.xml.namespace.QName;
import weblogic.jndi.Environment;

public class AdHocQuery
{
    public static InitialContext getInitialContext() throws NamingException {
        Environment env = new Environment();
        env.setProviderUrl("t3://localhost:7001");
        env.setInitialContextFactory("weblogic.jndi.WLInitialContextFactory");
        env.setSecurityPrincipal("weblogic");
        env.setSecurityCredentials("weblogic");
        return new InitialContext(env.getInitialContext().getEnvironment());
    }

    public static void main (String args[]) {
        System.out.println("===== Ad Hoc Client =====");
        try {
            StringBuffer xquery = new StringBuffer();
            xquery.append("declare variable $p_firstname as xs:string external; \n");
            xquery.append("declare variable $p_lastname as xs:string external; \n");
```

```

xquery.append(
  "declare namespace ns1=\"ld:DataServices/MyQueries/XQueries\"; \n");
xquery.append(
  "declare namespace ns0=\"ld:DataServices/CustomerDB/CUSTOMER\"; \n\n");

xquery.append("<ns1:RESULTS>                                \n");
xquery.append("{                                           \n");
xquery.append("    for $customer in ns0:CUSTOMER()                \n");
xquery.append("    where ($customer/FIRST_NAME eq $p_firstname      \n");
xquery.append("        and $customer/LAST_NAME eq $p_lastname)      \n");
xquery.append("    return                                              \n");
xquery.append("        $customer                                       \n");
xquery.append(" }                                                       \n");
xquery.append("</ns1:RESULTS>                                \n");

    PreparedExpression pe = DataServiceFactory.prepareExpression(
        getInitialContext(), "RTLApp", xquery.toString());
    pe.bindString(new QName("p_firstname"), "Jack");
    pe.bindString(new QName("p_lastname"), "Black");
    XmlObject results = pe.executeQuery();
    System.out.println(results);

} catch (Exception e) {
    e.printStackTrace();
}
}

```

Transaction Considerations

The API to Data Services Platform is supported internally by stateless EJBs; therefore, the data sources used by DSP must support the trans-attribute settings of the EJB methods. The default settings for the methods are:

- NotSupported is the default for the execute query methods.
- Required is the default for submit().

The trans-attribute for the submit() method is Required and cannot be changed. Other methods allow you to set the attribute to a value other than the default value by resetting ReadTransactionAttribute when creating a data service. For the executeQuery and executeFunction methods, you have the option of setting the trans-attribute to Required. You can set trans-attribute for executeQueryToStream and executeFunctionToStream to Supported.

For detailed information about the trans-attribute values of the EJBs, refer to section 17.6.2 of the EJB 2.0 specification. The specification is available at:

<http://java.sun.com/products/ejb/docs.html>.

Setting Up Data Source Aliases for Relational Sources Accessed by DSP

When you import metadata from relational sources, you can provide logic in your application that maps users to different data sources depending on the user's role. This is accomplished by creating an interceptor and adding an attribute to the RelationalDB annotation for each data service in your application.

The interceptor is a Java class that implements the `SourceBindingProvider` interface. This class provides the logic for mapping a users, depending on their current credentials, to a logical data source name or names. This makes it possible to control the level of access to relational physical source based on the logical data source names. For example, you could have the data source names `cgDataSource1`, `cgDataSource2`, and `cgDataSource3` defined on your WebLogic server and define the logic in your class so that an user who is an administrator can access all three data sources, but a normal user only has access to the data source `cgDataSource1`.

Note: All relational, update overrides, stored procedure data services, or stored procedure XFL files that refer to the same relational data source should also use the same source binding provider; that is, if you specify a source binding provider for at least one of the data service (.ds) files, you should set it for the rest of them.

To implement the interceptor logic, do the following:

1. Write a Java class `SQLInterceptor` that implements the interface `com.bea.ld.binds.SourceBindingsProvider` and define a `getBindings()` public method within the class. The signature of this method is:

```
public String getBinding(String genericLocator, boolean isUpdate)
```

The `genericLocator` parameter specifies the current logical data source name. The `isUpdate` parameter indicates whether a read or an update is occurring. A value of `true` indicates an update. A value of `false` indicates a read. The string returned is the logical data source name to which the user is to be mapped. [Listing 8-3](#) shows an example `SQLInterceptor` class.

2. Compile your class into a `.jar` file.
3. In your application, save the `.jar` file in the `APP-INF/lib` directory of your WebLogic Workshop application.

4. Define the configuration interceptor for the data source in your `.ds` or `.xfl` files (or both if necessary) by adding a `sourceBindingProviderClassName` attribute to the `RelationalDB` annotation. The attribute must be assigned the name of a valid Java class, which is the name of as your interceptor class. For example (the attribute and Java class are in bold):

```
<relationalDB dbVersion="4" dbType="pointbase" name="cgDataSource"
sourceBindingProviderClassName="sql.SQLInterceptor"/>
```

5. Compile and run you application. The interceptor will be invoked on execution.

Listing 8-3 Interceptor Class Example

```
package sql;

public class SqlProvider implements com.bea.ld.bindings.SourceBindingProvider{
    public String getBinding(String dataSourceName, boolean isUpdate) {

        weblogic.security.Security security = new weblogic.security.Security();
        javax.security.auth.Subject subject = security.getCurrentSubject();
        weblogic.security.SubjectUtils subUtils =
            new weblogic.security.SubjectUtils();

        System.out.println(" the user name is " + subUtils.getUsername(subject));

        if (subUtils.getUsername(subject).equals("weblogic"))
            dataSourceName = "cgDataSource1";

        System.out.println("The data source is " + dataSourceName);
        System.out.println("SDO " + (isUpdate ? " YES " : " NO ") );

        return dataSourceName;
    }
}
```

Setting Up Handlers for Web Services Accessed by DSP

When you import metadata from web services for DSP, you can create SOAP handler for intercepting SOAP requests and responses. The handler will be invoked when a web service method is called. You can chain handlers that are invoked one after another in a specific sequence by defining the sequence in a configuration file.

To create and chain handlers, follow these two steps:

1. Create Java class implements the interface `javax.xml.rpc.handler.GenericHandler`. This will be your handler. (Note that you could create more than one handler. For, example you could have one named `WShandler` and one named `AuditHandler`.) [Listing 8-4](#) shows an example implementation of a `GenericHandler` class. Place your handlers in a folder named `WShandler` in Weblogic Workshop. (For detailed information on how to write handlers, refer to [“Creating SOAP Message Handlers to Intercept the SOAP Message”](#) in the *Programming WebLogic Web Services*.

Listing 8-4 Example Intercept Handler

```
package WShandler;

import java.util.Iterator;
import javax.xml.rpc.handler.MessageContext;
import javax.xml.rpc.handler.soap.SOAPMessageContext;
import javax.xml.soap.SOAPElement;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.GenericHandler;
import javax.xml.namespace.QName;

/**
 * Purpose: Log all messages to the Server console
 */
public class WShandler extends GenericHandler
{
    HandlerInfo hinfo = null;

    public void init (HandlerInfo hinfo) {
        this.hinfo = hinfo;
        System.out.println("*****");
        System.out.println("ConsoleLoggingHandler r: init");
        System.out.println(
            "ConsoleLoggingHandler : init HandlerInfo" + hinfo.toString());
        System.out.println("*****");
    }
}
```

```
/**
 * Handles incoming web service requests and outgoing callback requests
 */
public boolean handleRequest(MessageContext mc) {
    logSoapMessage(mc, "handleRequest");
    return true;
}

/**
 * Handles outgoing web service responses and
 * incoming callback responses
 */
public boolean handleResponse(MessageContext mc) {
    this.logSoapMessage(mc, "handleResponse");
    return true;
}

/**
 * Handles SOAP Faults that may occur during message processing
 */
public boolean handleFault(MessageContext mc){
    this.logSoapMessage(mc, "handleFault");
    return true;
}

public QName[] getHeaders() {
    QName [] qname = null;
    return qname;
}

/**
 * Log the message to the server console using System.out
 */
protected void logSoapMessage(MessageContext mc, String eventType){
    try{
        System.out.println("*****");
        System.out.println("Event: "+eventType);
        System.out.println("*****");
    }
    catch( Exception e ){
        e.printStackTrace();
    }
}

/**
 * Get the method Name from a SOAP Payload.
 */
protected String getMethodName(MessageContext mc){
```

```

String operationName = null;

try{
    SOAPMessageContext messageContext = (SOAPMessageContext) mc;
    // assume the operation name is the first element
    // after SOAP:Body element
    Iterator i = messageContext.

getMessage().getSOAPPart().getEnvelope().getBody().getChildElements();
    while ( i.hasNext() )
    {
        Object obj = i.next();
        if(obj instanceof SOAPElement)
        {
            SOAPElement e = (SOAPElement) obj;
            operationName = e.getElementName().getLocalName();
            break;
        }
    }
}
catch(Exception e){
    e.printStackTrace();
}
return operationName;
}
}

```

2. Define a configuration file in your application. This file specifies the handler chain and the order in which the handlers will be invoked. The XML in this configuration file must conform to the schema shown in [Listing 8-5](#).

Listing 8-5 Handler Chain Schema

```

<?xml version="1.0" encoding="UTF-8"?>

<xs:schema targetNamespace="http://www.bea.com/2003/03/wlw/handler/config/"
xmlns="http://www.bea.com/2003/03/wlw/handler/config/"
xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified">
<xs:element name="wlw-handler-config">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="handler-chain" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element name="handler">
              <xs:complexType>

```

```

<xs:sequence>
  <xs:element name="init-param"
    minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="description"
          type="xs:string" minOccurs="0"/>
        <xs:element name="param-name" type="xs:string"/>
        <xs:element name="param-value" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="soap-header"
    type="xs:QName" minOccurs="0" maxOccurs="unbounded"/>
  <xs:element name="soap-role"
    type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
<xs:attribute name="handler-name"
  type="xs:string" use="optional"/>
<xs:attribute name="handler-class"
  type="xs:string" use="required"/>
</xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>

```

The following is an example of the handler chain configuration. In this configuration, there are two chains. One is named `LoggingHandler`. The other is named `SampleHandler`. The first chain invokes only one handler named `AuditHandler`. The `handler-class` attribute specifies the fully qualified name of the handler.

```

<?xml version="1.0"?>
<hc:wlv-handler-config name="sampleHandler"
xmlns:hc="http://www.bea.com/2003/03/wlv/handler/config/">
  <hc:handler-chain name="LoggingHandler">
    <hc:handler
      handler-name="handler1" handler-class="WSHandler.AuditHandler"/>
  </hc:handler-chain>
  <hc:handler-chain name="SampleHandler">
    <hc:handler

```

```

        handler-name="TestHandler1" handler-class="WShandler.WShandler"/>
    <hc:handler handler-name="TestHandler2"
        handler-class="WShandler.WShandler"/>
    </hc:handler-chain>
</hc:wlv-handler-config>

```

3. In your DSP application, define the interceptor configuration for the method in the data service to which you want to attach the handler. To do this, add a line similar the bold text shown in the following example:

```

xquery version "1.0" encoding "WINDOWS-1252";

(::pragma xds <x:xds xmlns:x="urn:annotations.ld.bea.com"
    targetType="t:echoStringArray_return"
    xmlns:t="ld:SampleWS/echoStringArray_return">
<creationDate>2005-05-24T12:56:38</creationDate>
<webService targetNamespace=
"http://soapinterop.org/WSDLInteropTestRpcEnc"
wsdl="http://web.service.bea.com:7001/rpc/WSDLInteropTestRpcEncService?W
SDL"/></x:xds>::)

declare namespace f1 = "ld:SampleWS/echoStringArray_return";

import schema namespace t1 = "ld:AnilExplainsWS/echoStringArray_return"
at "ld:SampleWS/schemas/echoStringArray_param0.xsd";

(::pragma function <f:function xmlns:f="urn:annotations.ld.bea.com"
kind="read" nativeName="echoStringArray"
nativeLevel1Container="WSDLInteropTestRpcEncService"
nativeLevel2Container="WSDLInteropTestRpcEncPort" style="rpc">
<params>
    <param nativeType="null"/>
</params>
<interceptorConfiguration aliasName="LoggingHandler"
fileName="ld:SampleWS/handlerConfiguration.xml" />
</f:function>::)

declare function f1:echoStringArray($x1 as
element(t1:echoStringArray_param0)) as
schema-element(t1:echoStringArray_return) external;
<interceptorConfiguration aliasName="LoggingHandler"
fileName="ld:testHandlerWS/handlerConfiguration.xml">

```

Here the `aliasName` attribute specifies the name of the handler chain to be invoked and the `fileName` attribute specifies the location of the configuration file.

4. Include the JAR file in the library module that defines the handler class referred to in the configuration file.

5. Compile and run your application. Your handlers will be invoked in the order specified in the configuration file.