BEA AquaLogic Data Services Platform™

Client Application Developer's Guide

Note: Product documentation may be revised post-release and made available from the following BEA e-docs site:

http://e-docs.bea.com/aldsp/docs21/index.html

Version: 2.1
Document Date: June 2005
Revised: March 2006

Copyright

Restricted Rights Legend

Trademarks or Service Marks

March 16, 2006 1:05 pm

Contents

1. Introducing Data Services Platform for Client Application Developers

2. DSP's Data Programming Model and Update Framework

3. Accessing Data Services from Java Clients

4. Web Services and DSP-Enabled Applications

5. Accessing Data Services from WebLogic Workshop Applications

6. Supporting ADO.NET Clients

7. Using Workflow with DSP-Enabled Applications

8. Using the Data Services Platform JDBC Driver

9. Customizing Data Service Update Behavior

10. Advanced Topics

CHAPTER 1

Introducing Data Services Platform for Client Application Developers

BEA AquaLogic Data Services Platform (DSP) brings a service-oriented architecture (SOA) approach to data access. Data Services Platform enables organizations to consolidate, integrate, and transform as needed disparate data sources scattered throughout their enterprise, making enterprise data available as an easy-to-access, reusable commodity: a *data service*.

For client application developers, DSP provides a uniform, consolidated interface for accessing and updating the heterogeneous back-end data sources that comprise data services. This chapter provides an overview of Data Services Platform for client application developers. It includes the following topics:

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

Simplifying Application 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.

This guide provides information about how to leverage data services in your applications. For information about creating data services in WebLogic Workshop, see the *Data Services Developer's Guide*.

What is a Data Services Platform Client?

In the typical organization, data flows in a bidirectional manner from a wide variety of sources, including distributed databases, various files, applications from partners, or e-commerce exchange markets.

Creating an application that can access and update distributed, disparate data sources can be complex, challenging, and expensive: you must know how to use a wide variety of connection mechanisms and data formats, and how to use the variety of APIs required to access and update each back-end data source, for example.

Using DSP, data architects create data services that:

- Insulates applications from having to access and update multiple disparate data sources.
- Provides the ability to create data services that combine elements of multiple, disparate data sources into, essentially, virtual databases.

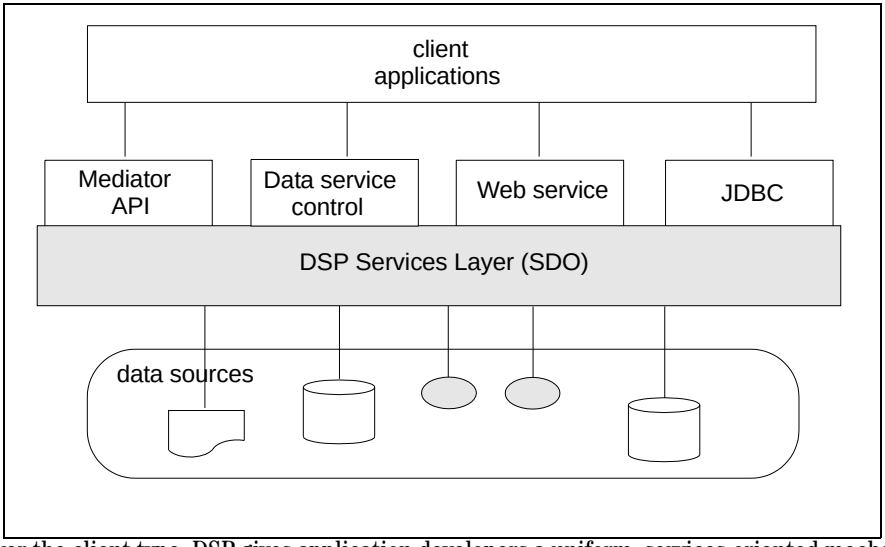
Data Your Way

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.

For client application developers needing to leverage such data assets, DSP supports multiple access methods (see Figure 1-1):

- Java clients can use data service functions through the DSP *Mediator API*.
- Workshop applications (such as portals, business processes, and web applications) can use a Data Service control.
- Web services enable you to 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.

Figure 1-1 Accessing DSP Services



Whatever the client type, DSP gives application 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.

The Role of WebLogic Server and WebLogic Workshop

Data services are created in WebLogic Workshop, BEA's integrated development environment for building and deploying many types of applications: portals, Web services, and integration applications, for example. The Data Services Platform application running under Workshop supports all aspects of Data Services Platform creation.

What is a Data Service?

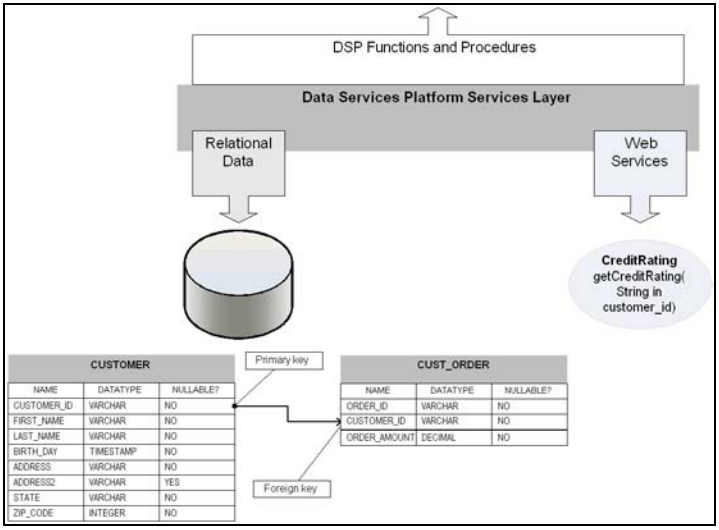
From a high-level perspective, a data service defines a distinct *business entity* such as a report that describes a customer and customer orders. The data service defines a unified view by aggregating data from any number of sources — relational database management systems (RDBMS), Web services, enterprise applications, flat files, and XML files, for example. Data services can also transform data from the original sources as needed.

In order to use data services, you need know only a few details, such as:

- The name of the data service.
- The functions and procedures exposed by the data service. (See What is a Data Services Platform Client Application?)
- The data types available.

Data service client applications can use data services in the same way that a Web service's client application invokes the operations of a Web service. Rather than invoking operations from a Java client application, the data service client application invokes a data service routine.

Figure 1-2 Data Services Layer Exposes Functions and Procedures to Client Application Developers



Note: For complete information on creating data services see the DSP *Data Services Developer's Guide*.

What is a Data Services Platform Client Application?

A DSP client application is any application that invokes data service routines. Client applications can include Java programs, non-Java programs such as Microsoft ADO.NET applications, BEA WebLogic Workshop applications, or JDBC/ODBC clients:

- Java client applications can use data service functions and procedures through the Data Services Mediator API (also known simply as the Mediator API).
- WebLogic Workshop applications (such as portals, business processes, and Web applications) can leverage data services by means of Data Service controls. (Controls are reusable Java components that can be used in WebLogic Workshop applications.) Data Service controls can be used as the basis of many DSP-enabled application scenarios. For example:
  - Data Service controls can be added to Web services, portal projects, and Web projects.
  - Data Service controls can be used to generate Web services that can make DSP services available to a wide variety of WebLogic and non-WebLogic applications and integration channels.
  - Data Service controls can be used within a JPD (Java process definition, a workflow component).

- The DSP JDBC driver provides JDBC and ODBC clients, such as reporting tools, with SQL-based access to DSP data.

Regardless of the client type, DSP provides a uniform, services-oriented mechanism for accessing and modifying distributed, heterogeneous data. Developers can focus on business logic, rather than on the details of various data source connections and formats.

In your client application code, simply invoke the data service routine: the DSP engine:

- Gathers data from the appropriate sources (via XQuery)
- Instantiating results as data objects, and
- Returns to your client application the materialized data objects.

These data objects conform to the Service Data Object (SDO) specification, a Java-based architecture and API for data programming that is the result of joint effort by BEA and IBM.

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 DSP *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`

Choosing a Data Services Programming Model

Application developers can choose from several models for accessing DSP services. The model chosen will depend on the access mechanism you decide to used. 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 1-3 provides a description of each of these access methods and summarizes the advantages of the various models for accessing DSP services.

Table 1-3 Data Services Platform Access Models

Access mechanism	Description	Advantages/When to use...
Data Service Mediator API	A Java interface for using data services. Returns data as data objects, providing full support for Service Data Objects (SDO) programming. For more information, see Chapter 2, "DSP's Data Programming Model and Update Framework."	• 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 through generic SDO, a static (typed) interface, and an ad hoc query interface. • Seamless ability to submit data changes.
Data Service Control	A Java control extension (JCX) file for accessing DSP resources. For more information, see Chapter 5, "Accessing Data Services from WebLogic Workshop Applications."	• 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.

Table 1-3 Data Services Platform Access Models

Access mechanism	Description	Advantages/When to use...
Web Service	A data service can be wrapped as a Web service, providing the data service with the benefits of web service features. For more information, see Chapter 4, "Web Services and DSP-Enabled Applications."	• 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	Client applications can use JDBC or ODBC to access DSP services using SQL queries. The DSP JDBC driver supports SQL-92. For more information, see Chapter 8, "Using the Data Services Platform JDBC Driver."	• 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.

Introducing 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. Keeping connections active for the minimum time possible maximizes 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 (*typed*) and dynamic (*untyped*) 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. Static interfaces depend on a schema for type information.

The dynamic interface, on the other hand, is useful when a type is unknown or undefined at runtime. Dynamic interface calls are in such forms as:

```
get("CustomerName")
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, "DSP's Data Programming Model and Update Framework."

Update Frameworks and the Data Service Mediator

The SDO specification does not specify an update framework, but it does discuss the need for mediator services, in general, to handle updates to data objects; the SDO specification leaves the details up to implementors.

The SDO specification allows for many types of mediators, each intended for a particular type of query language or back-end system. DSP provides a Data Service Mediator, a server-side component of DSP's XQuery processing engine that serves as the intermediary between data services and client applications or processes.

The Data Service Mediator facilitates updates to the various data sources that comprise any data service. DSP's Mediator service is the core mechanism for the DSP update framework. The Mediator handles updates to relational and non-relational data sources as follows:

- **Relational data sources**. Relational database management systems (RDBMS), such as IBM, Oracle, Microsoft SQL Server, and any other SQL-92 compliant RDBMS. For relational data sources, DSP propagates changes to relational data sources automatically. See "The Data Services Platform Update Framework" on page 2-15 for an overview of default behavior. (However, note that you can override the default update processing for a relational source if

you like, or when necessary. See "When Are Update Overrides Required for Relational Data Sources?" on page 9-4 for more information.)

- **Non-relational data sources**. This included non-relational data sources, such as Web services, XML files, and flat files. Updates to non-relational data sources always require custom server-side coding; specifically, an update override class. See "Developing the UpdateOverride Class" on page 9-6 for information about how to create a Java class to customize server-side behavior.

Typical Client Application Development Process

Developing client applications that invoke Data Services Platform functions and procedures presumes that a DSP project has been deployed or is being deployed. See the *Data Services Developer's Guide* for detailed information about developing data services.

Developing a DSP-enabled client applications encompasses these steps:

1. Identify the data services you want to use in your application. The Data Services Platform Console can be used to find all services available on your WebLogic Server. The DSP Console serves as a data service registry within the DSP architecture; it shows available data services, including the specific functions and procedures that each data service provides.

2. Choose the data access approach that best suits your needs. (Table 1-3, "Data Services Platform Access Models," on page 1-7 describes the advantages of the different access mechanisms.) The approach you choose also depends on precisely how the data service has been deployed.

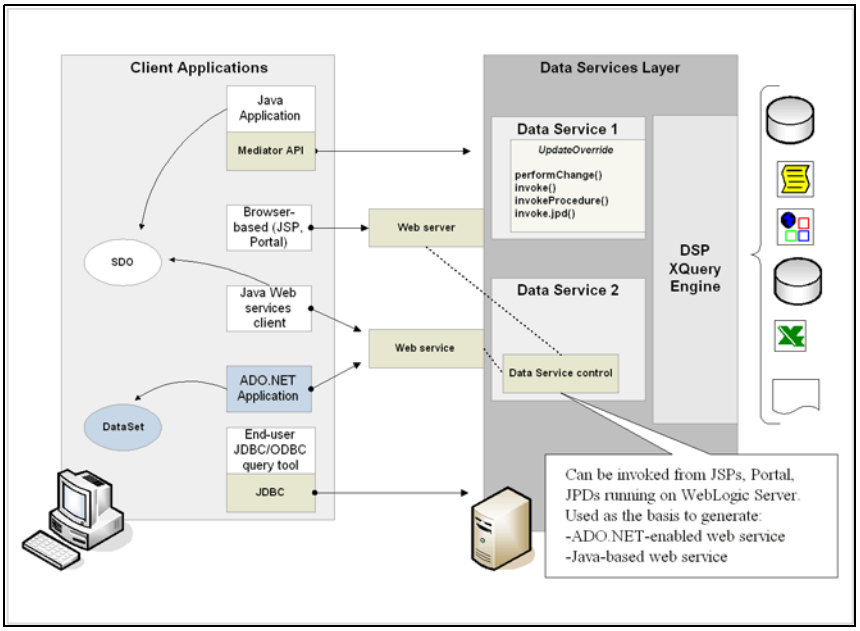
   For example, if the data service is hosted as a Web service, you can develop a Web service client application using Java in conjunction with the service's WSDL file.

   Similarly, if the data service is incorporated in a portal, business process, or Web application, your client application development process may take place entirely in the context of the server, as a set of pageflows or other server-side artifacts.

3. Obtain the required JAR files (see Table 1-5, "Data Services Platform Java Archive Files," on page 1-12). To use the typed data service and SDO interfaces, obtain the DSP application's generated Mediator client JAR file from your DSP administrator or data architect. Or generate the file yourself by following the steps outlined in "Generating a Static Mediator API JAR File" on page 3-5.

Figure 1-4 provides a conceptual overview of the various approaches, highlighting some of the relationships among various sub-systems and components.

Figure 1-4 Types of Data Services Platform Client Applications



Development Resources

Client application developers typically work with a small set of APIs, contained in JAR files. The APIs are primarily described through Javadocs (see "DSP Mediator API Javadoc" on page 1-13).

Runtime Client JAR Files

Data Services Platform APIs are contained the packages listed in Table 1-5.

Table 1-5 Data Services Platform Java Archive Files

Name	Description	Location
[*App*]-ld-client.jar	Contains generated typed interfaces for data services and their data types (static data APIs). The name of the file is is prefixed by the name of the DSP application from which the static interfaces are generated. Such an application-specific JAR file is not required if the only interface to Data Services Platform routines is through an untyped interface using generic SDO.	(Provided 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. [1]	<*bea_home*>\weblogic81\server\lib\

1. A "query too complex" exception is raised if the xqrl.jar and wlxbean.jar files are not in the JVM's CLASSPATH when an XPath expression is executed. If you encounter this error, make sure that these two JAR files are in the CLASSPATH.

DSP Mediator API Javadoc

The Data Services Platform Mediator API describes the routines needed by DSP client applications to invoke various DSP routines.

Client application developers will find Javadoc helpful for creating client applications that invoke data service routines. Data services developers and architects will find the Javadoc useful for understanding how to customize update behavior.

You can find javadoc for the Data Services Platform 2.1 Mediator API at:

`http://e-docs.bea.com/aldsp/docs21/javadoc/index.html`

Client applications built on earlier versions of Data Services Platform can continue to use the 2.0.1 mediator API routines. These are described in a Javadoc named javadoc-dsp201 and is available at:

`http://e-docs.bea.com/aldsp/docs21/javadoc-dsp201/index.html`

Javadoc is also provided in ZIP file format; it is available for download from the DSP e-docs Web site:

`http://e-docs.bea.com/aldsp/docs21/index.html`

Performance Considerations

Data service performance is the result of the end-to-end components that make up the entire system, including:

- **Data service design**. The number of data sources, complexity of logical data source consolidation, and other data service design considerations can affect performance.
- **Number of clients accessing the data service.** Number of simultaneous clients can affect performance.
- **Performance of the underlying data sources**. When data services access underlying data the availability and availability and performance of those systems can affect performance.
- **Network topology**. Overall available bandwidth must be measured against the number of applications running on the WebLogic Server, number of other applications in general consuming network bandwidth (can affect client response times).
- **Hardware resources.** The number of CPUs, processing power, memory allocation, and other factors for each and every platform throughout the system, client and server alike, can affect performance.

Before creating a client application for a data service, it is recommended that you benchmark performance of each underlying data source, and then benchmark the performance of the data service

as you develop it. Use load-testing tools to determine the maximum number of clients that your data service can support.

In addition, you can use DSP's auditing capabilities to instrument your code, thereby gaining performance profile information that you can use to identify and resolve performance problems if they occur. For detailed information on DSP audit capabilities see *DSP Administration Guide*.

Additional Technical and Product Information

A compendium of technical and product information related to BEA AquaLogic Data Services Platform can be found in the introductory chapter of the DSP *Concepts Guide*.

CHAPTER 2

DSP's Data Programming Model and Update Framework

BEA AquaLogic Data Services Platform (DSP) implements the Service Data Objects (SDOs) as its data client-application programming model. SDO is an architecture and set of APIs for working with data objects while disconnected from their source. In DSP, SDOs—whether typed or untyped data objects—are obtained from data services by using the Mediator APIs, or through Data Service controls. (See "Introducing Service Data Objects (SDO)" on page 1-8.)

Client applications manipulate the data objects as required for the business process at hand, and then submit changed objects to the data service, for propagation to the underlying data sources. Although the SDO specification does not define one, it does discuss the need for *mediator* services, in general, that can send and receive SDOs; the specification also discusses the need for handling updates to data sources, again, without specifying an implementation: The SDO specification leaves the details up to implementors as to how mediator services are implemented, and how they should handle updates to data objects.

As discussed in "Update Frameworks and the Data Service Mediator" on page 1-9, DSP's Mediator is the process that not only handles the back-and-forth communication between client applications and data services, it also facilitates updates to the various data sources that comprise any data service.

This chapter includes information about DSP's implementation of the SDO data programming model, as well as DSP's update framework. It includes:

Data Services Platform and Service Data Objects (SDOs)

When you invoke a data service's read or navigation function (through the Data Service Mediator API or from a Data Service control), the data service returns a *data graph* comprising one or more *data objects.* Data objects and data graphs are two fundamental artifacts of the SDO data programming model. As shown in Figure 2-1, a data graph comprises:

- A root object that typically corresponds to the root data type of a data service's return type.
- One or more data objects.
- A change summary.
- Metadata about the data objects; for example, the XML structure of a "CUSTOMER," comprising a LAST_NAME and an EMAIL_ADDRESS.

Each of these can be described in more detail, as follows:

- **Data Graph.** A data graph is a data type, the primary construct for SDO-based data programming. It is a data structure that serves as something of a container for related data objects. Data graphs encompass the data objects as instantiated from the data service, and track all changes made to those data objects.
- **Change Summary**. An object that tracks changes to data objects. A change summary exists only in the context of an associated data graph. As changes are made to the data objects that comprise the data graph—adds, deletes, or changes to the data objects or any of their properties—the changes are captured in the change summary.

  The change summary is used by the Mediator (in conjunction with a logical data service's decomposition map) to derive the update plan and ultimately, to update data sources. The change summary submitted with each changed SDO remains intact, regardless of whether or not the submit() succeeds, so it can support rollbacks when necessary.
- **Data Object.** A data object is a structure for containing property values. Properties can be simple types or complex types.
  - **Simple types.** Simple types comprise primitive data types, such as string or int, and correspond to leaf nodes in XML document trees.
  - **Complex types.** Complex types correspond to branch nodes in an XML document tree and may contain other data objects.

Figure 2-1 Client Applications and Data Service Mediator Send and Receive Data Graphs

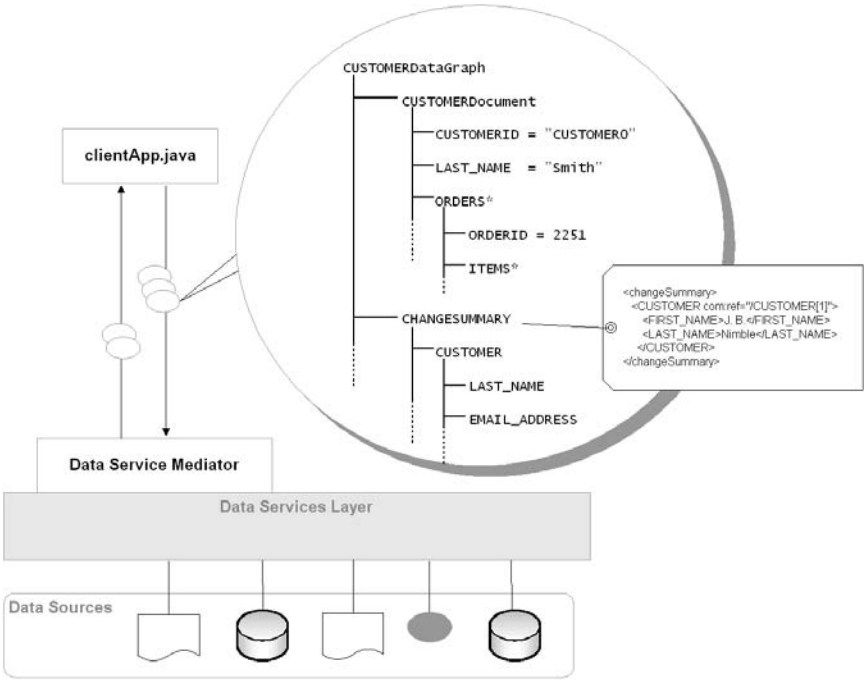


Table 2-2 summarizes the various SDO data programming artifacts and lists an example of each (as shown in Figure 2-1).

Table 2-2 Data Graph Example

Data Graph and Related Artifacts	Example
DataGraph	`CUSTOMERDataGraph`
DataObject	`CUSTOMER0, ORDERS`
Root Object	`CUSTOMERDocument`
ChangeSummary	`CHANGESUMMARY`
Property	`CUSTOMERID, LAST_NAME`
Simple Type	`CUSTOMERID`

Data Graph and Related Artifacts	Example
Complex Type	ORDERS
Metadata	<CUSTOMER> <LAST_NAME></LAST_NAME> <EMAIL_ADDRESS/> </CUSTOMER>

Static and Dynamic Data APIs

SDO specifies both static (typed) and dynamic (untyped) interfaces for data objects:

- **Static.** The static data API is an XML-to-Java API binding that contains methods that correspond to each element of the data object returned by the data service. These generated interfaces provide both getters and setters; for example, getCustomer(), setCustomer(). For examples see Table 2-5, "Static (Typed) Data API Getters and Setters," on page 2-6.
- **Dynamic.** The dynamic data API provides generic getters and setters for working with data objects. Elements are passed as arguments to the generic methods. For example, get("Customer") or set("Customer").

The dynamic data API can be used with data types that have not yet been deployed at development time.

Table 2-3 summarizes the advantages of each approach.

Table 2-3 Static and Dynamic Data APIs

Data Model	Advantages...
Static Data API	• Easy-to-implement interface; code is easy to read and maintain. • Compile-time type checking. • Enables code-completion in BEA WebLogic Workshop Source View.
Dynamic data API	• Dynamic; allows discovery. • Runtime type checking. • Allows for a general-purpose coding style.

Static Data API

SDO's static data API is a typed Java interface generated from a data service's XML schema definition. It is similar to JAXB or XMLBean static interfaces. The interface files, packaged in a JAR, are typically generated by the DSP data services developer using WebLogic Workshop, or by using one of the provided tools (see "Generating SDO Client Classes" on page 4-13 for more information).

The generated interfaces extend both the dynamic data API (specifically, the DataObject interface) and the XmlObject interface. Thus, the generated interfaces provide typed getters and setters for all properties of the XML datatype.

An interface is also generated for each complex property (such as CREDIT and ORDER shown in Figure 2-4), with getters and setters for each of the properties that comprise the complex type.

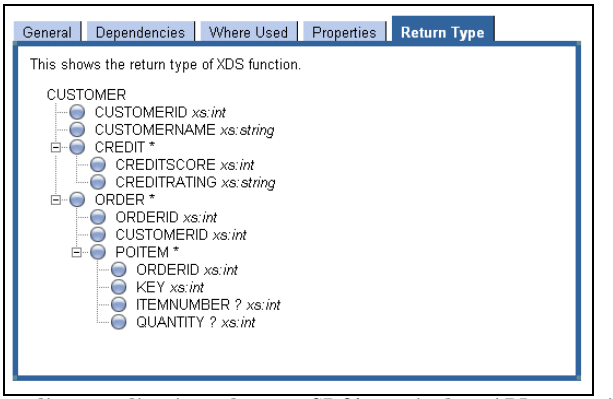
In addition, for properties that may have multiple occurrences, getters and setters are also generated for manipulating arrays and array elements. A multiple-occurring property is an XML schema element that has its maxOccurs attribute set to either unbounded or greater than one. In the DSP Console Metadata Browser, such elements are flagged with an asterisk—for example, ORDER* and POITEM* (see Figure 2-4) indicate that an array or order data objects (ORDERS[]) will be returned. For results involving repeating objects, you can cast the root element to an array of returned objects (*datatypename*[])

Note: In prior releases of Data Services Platform, an "ArrayOf..." schema element was created to serve as a container for array types returned as part of a Data Graph. Some references to the ArrayOf mechanism may remain in code samples and documentation.

As an example of how static data APIs get generated, given the CUSTOMER data type shown in Figure 2-4, generating typed client interfaces results in:

- CUSTOMER, CUSTOMERDocument, CREDIT, ORDER, and POITEM interfaces, each of which includes getters, setters, and factory classes (for instantiating typed data objects and their Properties).
- An interface for the CUSTOMERNAME string attribute.
- Getters and setters for working with members of arrays of CREDIT, ORDER, and POITEM elements.

Figure 2-4 CUSTOMER Return Type Displayed in DSP Console's Metadata Browser



When you develop Java client applications that use SDO's static data APIs, you will import these XMLBeans-generated typed interfaces into your Java client code. For example:

```
import appDataServices.AddressDocument;
```

The SDO API interfaces use XMLBeans for object serialization and deserialization. As a client application developer, you rarely need to know such details. However, developers who are integrating DSP with WebLogic Integration workflow components (JPDs, or Java process definitions) will need to modify the default serialization-deserialization in their JPD code that uses data objects. For more information, see Chapter 7, "Using Workflow with DSP-Enabled Applications."

Since DSP uses XMLBeans, many features of the underlying XMLBeans technology are available in SDO as well. For example, DataObjects can be cast to Strings using the XmlObjects toString( ) method, for printing to output.

Table 2-5 Static (Typed) Data API Getters and Setters

Static Data API (Generated)	Description	Examples
`Type getPropertyName()`	Returns the value of the property. A static get*PropertyName*( ) method is generated for each attribute or element that has a single occurrence.	`getCUSTOMER(), getCUSTOMERNAME(), getCREDITRATING(), getCREDITSCORE()`
`Type[] getPropertyNameArray()`	For multiple occurrence elements, returns all *PropertyName* elements.	`getCREDITArray()`
`Type getPropertyNameArray( int PropertyIndex)`	Returns the *PropertyName* child element at the specified index.	`getCREDITArray(int), setCREDITSCORE(int)`

Static Data API (Generated)	Description	Examples
`void set`*`PropertyName`*`(Type newValue)`	Sets the value of the property to the newValue. Generated when *PropertyName* is an attribute or an element with single occurrence.	`setCUSTOMER(CUSTOMER), setCUSTOMERNAME(String), setCREDITRATING(String)`
`void set`*`PropertyName`*`Array( Type[] newValue)`	Sets all *PropertyName* elements.	`setCREDITArray(CREDIT[])`
`void set`*`PropertyName`*`Array( Type newValue, int PropertyIndex)`	Sets the *PropertyName* child element at the specified index.	`setCREDITArray(int, CREDIT)`
`boolean isSet`*`PropertyName`*`()`	Determines whether the *PropertyName* element or attribute exists in the document. Generated for optional elements and attributes. (An optional element has a minOccurs attribute set to 0; an optional attribute has a use attribute set to optional.)	`isSetCustomerStreetAddress2()`
`void insert`*`PropertyName`*`(int index, `*`PropertyName`*`Type newValue)`	Inserts the specified *PropertyName* child element at the specified index.	`insertNewCREDIT(int)`
`int sizeOf`*`PropertyName`*`Array()`	Returns the current number of property child elements.	`sizeOfCREDITArray()`
`void unSet`*`PropertyName`*`()`	Removes the element or attribute of *PropertyName* from the document. Generated for elements and attributes that are optional. In schema, and optional element has an minOccurs attribute set to 0; an optional attribute has a use attribute set to optional.	`unSetCustomerStreetAddress2()`
`void remove`*`PropertyName`*`(int PropertyIndex)`	Removes the *PropertyName* child element at the specified index.	`removeCREDIT(int)`
`void add`*`PropertyName`*`( PropertyNameType newValue)`	Adds the specified *PropertyName* to the end of the list of *PropertyName* child elements.	addNewCREDIT(), addNewCUSTOMER()

Static Data API (Generated)	Description	Examples
`boolean isSet`*`PropertyName`*`Array(int PropertyIndex)`	Determines whether the *PropertyName* element at the specified index is null.	`isSetCustomerArray(3)`
`void unset`*`PropertyName`*`Array( int PropertyIndex)`	Sets the value of *PropertyName* element at the specified index to null. **Note:** After you call unset and then call set, the return value is false.	`unSetCustomerArray(3)`

XML Schema-to-Java Type Mapping Reference

DSP client application developers can use the Data Services Platform Console to view the XML schema types associated with data services (see Figure 2-4, "CUSTOMER Return Type Displayed in DSP Console's Metadata Browser," on page 2-6). The Return Type tab indicates the data type of each element—string, int, or complex type, for example. The XML schema data types are mapped to data objects in Java using the data type mappings shown in Table 2-6.

Table 2-6 XML Schema to Java Data Type Mapping

XML Schema Type	SDO Java Type	XML Schema Type	SDO Java Type
xs:anyType	Sequence	xs:integer	java.math.BigInteger
xs:anySimpleType	String	xs:language	String
xs:anyURI	String	xs:long	long
xs:base64Binary	byte[]	xs:Name	String
xs:boolean	boolean	xs:NCName	String
xs:byte	byte	xs:negativeInteger	java.math.BigInteger
xs:date	java.util.Calendar (Date)	xs:NMTOKEN	String
xs:dateTime	java.util.Calendar	xs:NMTOKENS	String
xs:decimal	java.math.BigDecimal	xs:nonNegativeInteger	java.math.BigInteger
xs:double	double	xs:nonPositiveInteger	java.math.BigInteger

XML Schema Type	SDO Java Type	XML Schema Type	SDO Java Type
xs:duration	String	xs:normalizedString	String
xs:ENTITIES	String	xs:NOTATION	String
xs:ENTITY	String	xs:positiveInteger	java.math.BigInteger
xs:float	float	xs:QName	javax.xml.namespace.QName
xs:gDay	java.util.Calendar	xs:short	short
xs:gMonth	java.util.Calendar	xs:string	String
xs:gMonthDay	java.util.Calendar	xs:time	java.util.Calendar
xs:gYear	java.util.Calendar	xs:token	String
xs:gYearMonth	java.util.Calendar	xs:unsignedByte	short
xs:hexBinary	byte[]	xs:unsignedInt	long
xs:ID	String	xs:unsignedLong	java.math.BigInteger
xs:IDREF	String	xs:unsignedShort	Int
xs:IDREFS	String	xs:keyref	String
xs:int	int		

Dynamic Data API

The dynamic data API has generic property getters and setters, such as set( ) and get( ), as well as getters and setters for specific Java data types (String, Date, List, BigInteger, and BigDecimal, for example). Table 2-7 lists representative APIs from SDO's dynamic data API. The propertyName argument indicates the name of the property whose value you want to get or set; propertyValue is the new value. The dynamic data API also includes methods for setting and getting a DataObject's property by indexValue. This includes methods for getting and setting properties as primitive types, which include setInt( ), setDate( ), getString( ), and so on.

Unlike the static data API, which eliminates underscores in method names generated from types that might include such characters ("LAST_NAME" results in a getLASTNAME( ) method, for example), the dynamic data API requires that field names be referenced precisely, as in get("LAST_NAME"). As an example, assuming that you have a reference to a CUSTOMER data object, you can use the dynamic data API to get the LAST_NAME property as follows:

```
String lastName = (String) customer.get("LAST_NAME");
```

For a complete reference of the dynamic data API, see the DSP Javadoc ("DSP Mediator API Javadoc" on page 1-13). For documentation on the SDO 1.0 API see the DataObject interface in the commonj.sdo package. It is available at:

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

Table 2-7 Dynamic Data API Getters and Setters

Dynamic Data API	Description	Example
`get(int PropertyIndex)`	Returns the *PropertyName* child element at the specified index.	`get(5)`
`set(int PropertyIndex, Object newValue)`	Sets the value of the property to the newValue.	`set(5, CUSTOMER3)`
`set(String PropertyName, Object newValue)`	Sets the value of the PropertyName to the newValue.	`set("LAST_NAME", "Nimble")`
`set(commonj.sdo.Property PropertyName, Object newValue)`	Sets the value of PropertyName to the NewValue	`set(LASTNAME, "Nimble")`
`getType(String PropertyName)`	Returns the value of the *PropertyNam*e. *Type* indicates the specific data type to obtain.	`getBigDecimal("CreditScore")`
`unset(int PropertyIndex)`	Sets the value of *PropertyName* element at the specified index to null.	`unset(5)`
`unset(commonj.sdo.Property PropertyName)`	Sets the value of the specified *PropertyName* to null.	`unset(LASTNAME)`
`unset(String PropertyName)`	Sets the value of the specified *PropertyName* to null.	`unset("LAST_NAME")`
`createDataObject(commonj.sdo.Property PropertyName)`	Returns a new DataObject for the specified containment property.	`createDataObject(LASTNAME)`

Dynamic Data API	Description	Example
`createDataObject(String PropertyName)`	Returns a new DataObject for the specified containment property.	`createDataObject("LAST_NAME")`
`createDataObject(int PropertyIndex)`	Returns a new DataObject for the specified containment property.	`createDataObject(5)`
`createDataObject(String PropertyName, String namespaceURI, String typeName)`	Returns a new DataObject for the specified containment property.	`createDataObject("LAST_NAME", "http://namespaceURI_here", "String")`
`delete()`	Removes the object from its container and unsets all writeable properties.	`delete(CUSTOMER)`

XPath Support in the Dynamic Data API

One of the benefits of DSP's use of XMLBeans technology is support for XPath in the dynamic data API. XPath expressions give you a great deal of flexibility in how you locate data objects and attributes in the dynamic data API's accessors. For example, you can filter the results of a get() method invocation based on data elements and values:

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

The SDO implementation goes beyond basic XPath 1.0 support by adding zero-based array index notation (".index_from_0") to XPath's standard bracketed notation ([n]). As an example, Table 2-8 compares the XPath standard and SDO augmented notations to refer to the same element, the first ORDER child node under CUSTOMER.

Table 2-8 XPath Standard and SDO Augmented Notation

XPath Standard Notation	SDO Augmented Notation
`get("CUSTOMER/ORDER[1]");`	`get("CUSTOMER/ORDER.0");`

Zero-based indexing is convenient for Java programmers who are accustomed to zero-based counters, and may 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 SDO pre-processor transparently replaces the zero-based form with one-based forms, to avoid conflicts with elements whose names include dot numbers, such as `<myAcct.12>`.

Keep in mind these other points regarding DSP's XPath support:

- 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//POITEM")
  ```

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

  ```
  CUSTOMER/ORDERS/POITEM
  ```

  ```
  CUSTOMER/RETURNS/POITEM
  ```

  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 be used to identify only the attributes that 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
  ```

Note: For more examples of using XPath expressions with SDOs, see "Step 2: Accessing Data Object Properties" on page 3-23.

Obtaining Type Information about Data Objects

The dynamic data API returns *generic* data objects. To obtain information about the properties of a data object, you can use methods available in SDO's Type interface. The Type interface (located in the `commonj.sdo` package) provides several methods for obtaining information, at runtime, about data objects, including a data object's type, its properties, and their respective types.

According to the SDO specification, the Type interface (see Table 2-9) and the Property interface (see Table 2-10) comprise a minimal metadata API that can be used for introspecting the model of data objects. For example, the following obtains a data object's type and prints a property's value:

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

Once you have an object's data type, you can obtain all its properties (as a list) and access their values using the Type interface's get Properties() method, as shown in Listing 2-1.

Listing 2-1 Using SDO's Type Interface to Obtain Data Object Properties

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

Table 2-9 lists other useful methods in the Type interface.

Table 2-9 Type Interface Methods

Method	Description
`java.lang.Class getInstanceClass()`	Returns the Java class that this type represents.
`java.lang.String getName()`	Returns the name of the type.

Method	Description
`java.lang.List getProperties`	Returns a list of the properties of this type.
`Property getProperty( java.lang.String propertyName)`	Returns from among all Property objects of the specified type the one with the specified name. For example, dataObject.get("name") or dataObject.get(dataObject.getType().getProperty ("name"))
`java.lang.String getURI()`	Returns the namespace URI of the type.
`boolean isInstance( java.lang.Object object)`	Returns True if the specified object is an instance of this type; otherwise, returns false.

Table 2-10 lists the methods of the Property interface.

Table 2-10 Property Interface Methods

Method	Description
`Type getContainingType()`	Returns the containing type of this property.
`java.lang.Object getDefault()`	Returns the default value this property will have in a data object where the property hasn't been set
`java.lang.String getName()`	Returns the name of the property.
`Type getType()`	Returns the type of the property.
`boolean isContainment()`	Returns True if the property represents by-value composition.
`boolean isMany()`	Returns True if the property is many-valued.

Role of the Mediator and SDOs

In DSP, data graphs are passed between data services and client applications: when a client application invokes a read function on a data service, for example, a data graph is sent to the client application. The client application modifies the content as appropriate—adds an order to a customer order, for example—and then submits the changed data graph to the data service. The Data Service

Mediator is the process that receives the updated data objects and propagates changes to the underlying data sources.

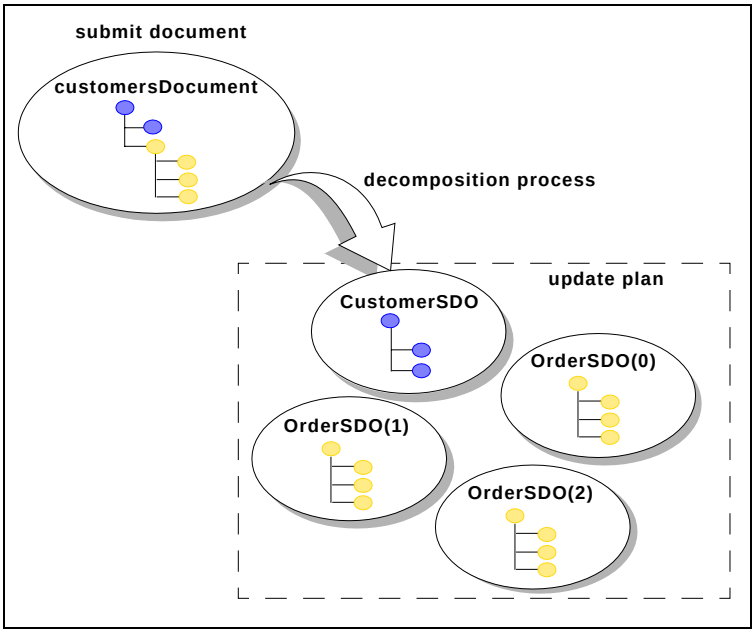
The Data Service Mediator is the linchpin of the update process. It uses information from submitted SDOs (change summary, for example) in conjunction with other artifacts to derive an update plan for changing underlying data sources. For relational data sources, updates are automatic. The artifacts that comprise DSP's update framework, including the Mediator, and how the default update process works, are described in more detail below.

The Data Services Platform Update Framework

As mentioned previously, the SDO specification does not define any specific mediators, but allows for the variety needed to support any type of back-end data sources. DSP's implementation of an SDO mediator service is the Data Service Mediator (or DSP Mediator) shown in Figure 2-1. The DSP Mediator plays an important role in facilitating updates to the various data sources that comprise any data service. It is the core mechanism for the DSP update framework; the update framework also encompasses several programming artifacts, as follows:

- **Decomposition function**. The decomposition function is the first read function contained in a data service, unless a different function has been specified by the data service architect through the Property Editor. The decomposition function is used by the Mediator to create a decomposition map (for logical data services only) that identifies the constituent data services. From these constituent data services, the Mediator instantiates data objects corresponding to the changed values in the updated data object.
- **Decomposition map**. A decomposition map (associated with logical data services only) provides information about how a data object based on that data service is constructed (from the underlying data sources or other data services).
- **Update plan**. The indicates the physical resources that should be modified, and how they should be modified. An update plan is generated for any changed data objects bound to logical data services (see Figure 2-11). The update plan does not include unchanged objects, nor does it have access to any data services that are not included in the update plan: only changed objects are included in the update plan.
- **KeyPair**. A keypair is an object used by DSP to keep track of primary-foreign key relationships, and the relationship of properties of data objects that have been populated from various layers of a multi-layer data service. (A keypair is sometimes described as a property map.)

Figure 2-11 DSP's Decomposition Process Populates an Update Plan with Constituent Data Objects



From a lower-level perspective, an *update plan* is a Java object that comprises a tree of DataServiceToUpdate instances — the names of the data services that comprise the changed data objects. DataServiceToUpdate, KeyPair, UpdatePlan, and DataServiceMediatorContext have been implemented as classes in the SDO Mediator APIs, specifically in:

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

See "DSP Mediator API Javadoc" on page 1-13 for information on product Javadocs.

How It Works: The Decomposition Process

An important characteristic of the SDO model is that back-end data sources associated with modified objects are not changed until the submit() method is called on the data service bound to the objects.

After receiving a data object (the changed SDO) from a calling client application, the Mediator always looks for an update override class first (regardless of whether the data service is a physical or logical data service). If an update override class is available, it is instantiated and executed.

Note: Update overrides are covered in detail in Chapter 9, "Customizing Data Service Update Behavior." This chapter covers the basics of the *default* update processing only.

The Mediator first determines the *data lineage*—the origins of the data—by using the data service's decomposition function to map each constituent in a data object to its underlying data source or data service. In addition, any inverse functions specified for the data service are used by the Mediator to define a complete decomposition map.

Note: The usage of inverse functions is described in "Best Practices and Advanced Topics", Data Services Developer's Guide.

As discussed above, for any logical data service, DSP's Mediator uses the decomposition function to create a decomposition map that identifies constituent data services and then instantiates data objects that correspond to the data objects' changed values. For example, as shown in Figure 2-11, a customersDocument object that comprises updated customer information (from a Customer data service) and three updated Orders (from an Orders data service) would be decomposed into four objects.

An important distinction between logical and physical data service updates is as follows:

- **Physical Data Service Update Process.** The data source is updated immediately. No decomposition is required.
- **Logical Data Service Update Process.** A logical data service must be decomposed into its constituent data services.

Physical Data Service Update Process

For a physical data service, changes to the data sources are propagated immediately (unless an update override class is associated with the data service).

Note: Neither a decomposition map nor an update plan is needed for a physical data service.

Upon receiving an SDO (whether from a submit( ) method invocation, or as a projection from a higher-level data service), the Mediator first checks for an UpdateOverride class associated with the data service.

- **No update override**. If there is no UpdateOverride the Mediator simply propagates the changes to the underlying data sources.
- **With update override**. If there is an UpdateOverride the Mediator executes the update override class.

  Note: For non-relational data sources, an update override is always required, since there is no automatic update processing for non-relational data sources.

For relational data sources without an update override, updates are handled automatically. However, non-relational data sources such as Web services, flat files, XML files, require an update override class that contains the processing logic necessary to make changes to the data source.

Logical Data Service Update Process

A logical data service can comprise any number of logical or physical data services. When a top-level data service function executes, the lower-level logical data services that it comprises are "folded in" so that the function appears to be written directly against physical data services. Only information that has been projected in the top-level data service is passed to the next lower-level data service.

Figure 2-12 provides an overview of the steps involved in updating a logical data service:

1. The client application invokes the submit() method, passing the changed data object and its associated data graph to the Mediator. The data graph has a change summary detailing the changes to the object.

2. The Mediator receives the submitted data object and begins the decomposition process by first checking for an update override class. The two possible logic branches are described below:

   – **No update override**. The Mediator decomposes the updated object into submit() calls against the underlying physical data services.

   – **With update override.** The Mediator instantiates mid-level data objects from the top-level SDO, then calls update override routine. The submit() on the mid-level data service is then processed as usual.

     Note: An update override class can exist at each layer of a multi-layered data service. Thus, a logical data service comprising several layers of other logical data services checks for an update override at each constituent layer. If a mid-layer data service has no update override, the update framework bypasses the instantiation of an SDO object, instead directly creating the SDO objects for the underlying data service. This is true in the case of a logical data service with an update override or a physical data service.

   The performChange( ) method can access and modify the update plan and decomposition map, or perform any other custom processing, including taking over complete processing.

   The performChange( ) method returns a Boolean value that either continues or aborts processing by the Mediator, as follows:

   – **True**. After control returns from the method, the Mediator resumes its normal course of processing. 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 apply the changes. The method would return false, for example, if all changes have already been made. (If you want to handle an error that would require the update to be aborted, your method should throw an exception.)

Note: See Chapter 9, "Customizing Data Service Update Behavior," for complete information about customizing behavior.

3. The Mediator determines the origins of the data sources that must be changed and how to change them. The Mediator calls the decomposition function associated with the data service and receives a decomposition map for the data service. By default, the Mediator uses the data service's first read function to create its decomposition map (if no other decomposition function is specified).

   a. The Mediator uses the information in the change summary and the data service's decomposition map to derive an update plan. The update plan comprises a tree of data service objects ("SDO objects to update") for each instance of a changed data source.

   b. For any lower-level data service, the Mediator also checks for an update override, and executes the update override class if one is present.

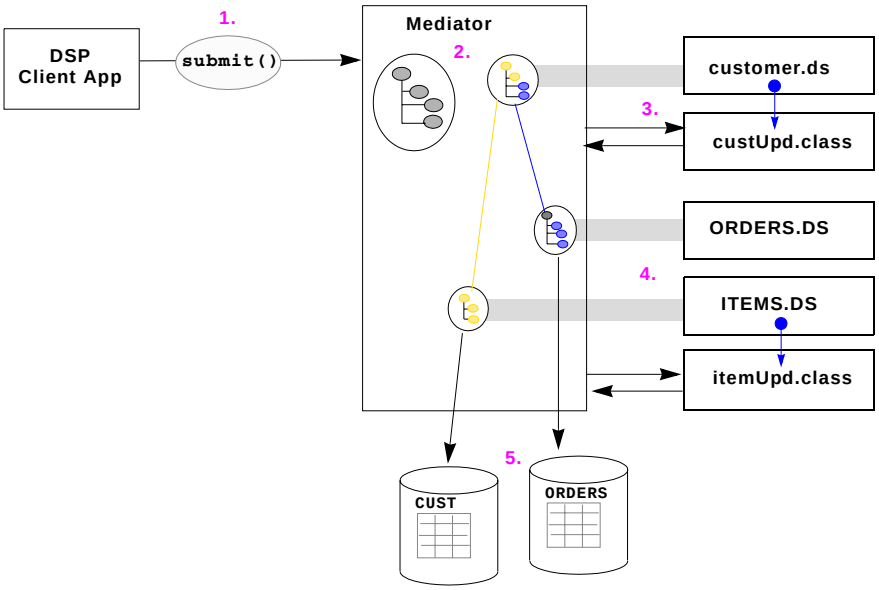
4. The Mediator iterates (walks) through the update plan, submitting changes to each of the lower level data services. The Mediator applies changes based on the order of objects in the tree and their container-containment relationships, as follows:

   a. Objects within the same level (sibling objects) are processed in the order in which they are encountered in the data object.

   b. Container objects are processed before contained objects—unless the container is being deleted, in which case changes are applied to the contained object before the containing object.

   c. If an object has a KeyPair specified, the values are mapped from its container before submitting the change. (Changes made to an SDO container during its update, such as primary key computations, are visible in the contained object.)

Figure 2-12 Logical Data Service Update Process



Primary-Foreign Key Relationships Mapped Using a KeyPair

Most RDBMSs can automatically generate primary keys, which means that if you are adding new data objects to a data service that is backed by a relational database, you may want or need to handle a primary key as a return value in your code. For example, if a submitted data graph of objects includes a new data object, such as a new Customer, DSP generates the necessary primary key.

For data inserts of autonumber primary keys, the new primary key value is generated and returned to the client. Only primary keys of top-level data objects (top-level of a multi-level data service) are returned; nested data objects that have computed primary keys are not returned.

By returning the top-level primary key of an inserted tuple, DSP allows you to re-fetch tuples based on their new primary keys, if necessary.

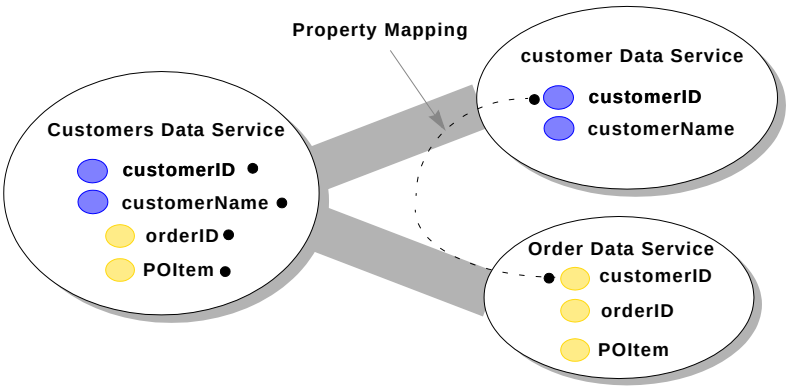
The Mediator saves logical primary-foreign keys as a KeyPair (see the `KeyPair` class in the Mediator API). A KeyPair object is a property map that is used to populate foreign-key fields during the process of creating a new data object:

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.

The KeyPair object is used to identify corresponding data elements at adjacent levels of a decomposition map; it ensures that a generated primary key value for a parent (container) object will be mapped to the foreign key field of the child (contained) element.

As an example, Figure 2-13 shows property mapping for the decomposition of a Customers data service.

Figure 2-13 Logical Data Services Use KeyPairs for Property Mapping (Primary-Foreign Key Mapping)



DSP manages the primary-foreign key relationships between data services; how the relationship is managed depends on the layer (of a multi-layered data service), as follows:

- **Top-level data service**. Inserts generate a new primary key, which is returned to the client. Adding a data object at the top-level of a data service 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:

  ```
  Properties[] keys = ds.submit(doc);
  ```

  A tuble is basically a record; in the context of data services, a tuble may comprise data that spans several layers of data services.

- **Nested data objects**. Generates and inserts a new primary key, but does not return to the client.

DSP propagates the effects of changes to a primary or foreign key.

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 during the update process. It identifies primary keys and can infer foreign keys in predicate statements. For example, in a query that joins data by comparing values, as in:

```
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 identify only those primary key fields from the predicate fields.

The KeyPair maps a container primary key to *container* field only. If the KeyPair does not container's complete primary key 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.

Foreign Keys

When computable by SDO submit decomposition, foreign key values are set to match the parent key values.

Foreign keys are computed when an update plan is produced.

Transaction Management

Each submit() to the Mediator operates as a transaction. Depending upon whether the submit() succeeds or fails, you should do one of two things:

- **Submit() succeeds**. You can re-query the SDO to be sure it matches the current data because side effects of the update may have changed the result of the query. (Re-querying the data service to obtain a new data object also clears the change summary.)
- **Submit() fails**. You can reinvoke submit() on the data object to execute the same updates (since the original data objects and change summary still exist).

Nested Transactions

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

CHAPTER 3

Accessing Data Services from Java Clients

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

Overview of the Data Services Platform Mediator API

The BEA AquaLogic Data Services Platform (DSP) Mediator API gives Java client application developers easy-to-use interfaces for using data service routines. To use the Mediator API, simply instantiate a remote data service interface and invoke public methods on the interface. Public methods can include read functions, navigation functions, and procedures.

The return type for the invocations depends on the type of method and whether the static or dynamic interfaces are used, as follows:

- **Read and navigation functions**. When a read function or navigation function is invoked through the Mediator API, the client application gets back information as a data graph that comprises the data objects constructed by the data service.

- **Procedures**. When a procedure is invoked through the Mediator API, the client application may or may not get back an SDO, depending on the implementation details of the procedure as configured for the data service.

The Mediator API provides both static and dynamic interfaces for working with data services.

- **Static Mediator APIs**. You can use the static mediator APIs to invoke functions on multiple data services, then cast the acquired objects to the appropriate data types. Static Mediator APIs are generated from a specific data service.
- **Dynamic Mediator APIs**. Use the dynamic mediator APIs to instantiate and invoke data service functions and procedures by name.

The Mediator API also supports several advanced features, including:

- **Ability to filter, sort, and truncate return values**. Your client applications can organize or limit returned results in several different ways using the Mediator API's Filter and FilterXQuery classes. For more information, see "Filtering, Sorting, and Fine-tuning Query Results" on page 10-5.
- **Ability to stream data service function results.** The static and dynamic interfaces data service materialize data service function call results as XML, in memory. However, in-memory materialization is not always practical. The Mediator API offers several different stream-oriented interfaces. For more information, see "Handling Large Result Sets with Streaming APIs" on page 10-15.
- **Ad hoc query interface**. The Mediator API's PreparedExpression interface enables client applications to invoke ad hoc XQuery expressions against data service results. Ad hoc queries can return anything, including simple data. Simple data is not represented as DataObjects (XmlObjects); however, in DSP ad hoc queries can return DataObjects if the returned XML is structured correctly and the appropriate static SDO classes are on the classpath. For more information, see "Using Ad Hoc Queries to Fine-tune Results from the Client" on page 10-11.

The Mediator APIs are used to instantiate interfaces to data services and invoke data service functions and procedures. Functions and procedures defined for a data service are available as methods in the Mediator API.

The dynamic 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.dsp.dsmediator.client
```

The API consists of the classes and interfaces listed in Table 3-1

Table 3-1 Data Services Platform Mediator API

Interface or Class Name	Description
DataService	Interface for data services that returns data as Data Objects. The interface includes invoke() method for invoking read and navigation functions; invokeProcedure() for invoking procedures; and submit() method for submitting data object changes.
PreparedExpression	Interface for preparing and executing ad hoc queries. An ad hoc query is one that is defined in the client program, not in the data service.
DataServiceFactory	Factory class for creating local interfaces to data services. Can be used for dynamic data service instantiation and ad hoc queries.
StreamingDataService	Interface for data services that returns data as a token stream.
StreamingPreparedExpression	Interface for preparing and executing ad hoc query functions that return information as a stream. An ad hoc query is an XQuery that is passed as a string from within a client program (rather than in the data service).

The static mediator API interface extends the static Mediator interface, as shown in this example of a class declaration for a typed data service:

```
public class dataservices.Customer extends
          com.bea.dsp.dsmediator.client.DataService { … }
```

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

The exception class for Mediator errors (SDOMediatorException) is in the following package:

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

Exceptions that are generated by the data source (such as SQLException) are wrapped in an SDO Exception, and can be accessed by calling getCause() on the SDOMediatorException.

Setting the Classpath

To develop Java-based client programs using the Mediator APIs, your development environment's CLASSPATH must include the JAR files listed in Table 1-5, "Data Services Platform Java Archive Files," on page 1-12.

In addition, to use static data APIs, you must include the *<app-name>*`-ld-client.jar` file (obtain from your data service architect or administrator).

Note: The *<app-name>*`-ld-client.jar` file is not needed for generic SDO.

As an example, for a data service named Demo using static APIs, your classpath on a Microsoft Windows operating system would include:

```
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\server\lib\xqrl.jar;
    C:\bea\weblogic81\server\lib\wlxbean.jar;
    C:\bea\weblogic81\liquiddata\lib\ld-client.jar;
```

This classpath assumes that the first item, `Demo-ld-client.jar`, is in the current directory and that the BEA WebLogic home directory is: `C:\bea\weblogic81`. Modify the path to the locations appropriate for your system, and change the name of `Demo-ld-client.jar` to the actual name of the JAR file generated from your DSP-enabled application.

Mediator API Summary and Reference

Client application developers can take two alternative approaches to working with SDOs:

- Mediator APIs, which encompass the two Java packages listed in Table 3-2
- Data Service controls, a server-side Java class file that adheres to the Java Control Extension (JCX) standard.

This chapter discusses the Mediator APIs and how to use in Java client applications.

Client application developers will use some combination of the APIs shown in Table 3-2, depending on your application design and specific goals. Data service developers will also use the SDO Update API (specifically, the UpdateOverride interface) to customize data service functionality.

Table 3-2 Data Service Mediator APIs Package Reference

	SDO Mediator APIs	SDO Update APIs
Package	`com.bea.dsp.dsmediator.client`	`com.bea.ld.dsmediator.update`
Description	DataServiceFactory and other classes. DataService, StreamingDataService, and PreparedExpression interfaces.	DataServiceMediatorContext, DataServiceToUpdate, KeyPair, DataServiceMediator, and UpdatePlan classes. UpdateOverride interface.
Usage note	Instantiate remote interface to a data service.	Submit changed data objects to data service. Override default update processing for a particular data service.
Location	*<bea_home>*`\weblogic81\liquiddata\lib\ld-client.jar`	Same as for SDO Mediator APIs.
Javadoc	Data Services API Javadoc.	Same as for SDO Mediator APIs.
Javadoc location	`http://e-docs.bea.com/aldsp/docs21/Javadoc/index.html`	Same as for SDO Mediator APIs.

Generating a Static Mediator API JAR File

Client applications can access the classes representing a static data service interface using the JAR (Java Archive) file generated from the DSP project. Client application developers must obtain this JAR file (typically, from the data services architect or the Data Services Platform administrator) and add the JAR file in the classpath of their development environment.

The naming convention for the generated, static Mediator client JAR file is:

<AppName>`-ld-client.jar`

Building the Client JAR

Once the data service application has been built into an EAR file, the client version of the data service — the *<AppName>*`-ld-client.jar` file — can be generated from the EAR. The client version includes wrapper classes that allow the client to call the data service functions through a dynamic API.

The necessary JAR file can be generated in either of two ways:

- From WebLogic Workshop, with the top folder of the application selected, right-click and select Build SDO Mediator Client from the pop-up menu. (This menu option is available from the root folder of the application only.)
- From the command prompt of the data service development machine by using the Ant script, as follows:

  a. At a command prompt, navigate to the directory.

  b. Execute the shell or command file script to set the environment for your machine. For Windows use `setWLSEnv.cmd`; for UNIX use `setWLSEnv.sh`.

     These scripts can be found in the following location:

     `<beahome>/weblogic81/server/bin`

  c. Run the Ant script, passing in the name of a temporary directory as one of the parameters as shown below:

```
ant -Doutdir=<output-directory> -Darchive=<archive>
-Dtmpdir=\tmp\clientbld -fld_clientapi.xml
```

Table 3-3 Arguments for Generating a Mediator Static Client JAR from Data Services EAR

Argument	Description
`<archive>`	Fully qualified name of the generated EAR file. The generated name is derived from the name of the application.
`<outdir>`	Directory in which to generate the client JAR file. Optional parameter; if unspecified, the current directory is used.
`<tmpdir>`	Directory in which to produce the temporary, expanded EAR file contents. Although this parameter is optional, BEA recommends that you always create and specify a temporary directory, since all contents will be deleted at the end of the process. If <tmpdir> is not specified, the current directory will be used.

For example:

```
ant -Doutdir="c:\myApp"
=Darchive=C:\bea\user_projects\applications\myApp.ear -Dtmpdir=c:\temp
-fld_clientapi.xml
```

Executing the command as shown in this example produces the client JAR file, as follows:

```
C:\myApp-ld-client.jar
```

Using the Data Service Mediator API

To use the Data Service Mediator API to invoke data service functions and procedures, create a Java class as follows:

1. Import the `com.bea.dsp.dsmediator.client` package.
2. Create a JNDI context for the WebLogic Server that hosts the DSP application.

   Note: For more information, see "Obtaining a WebLogic JNDI Context for Data Services Platform" on page 3-7. For complete information about WebLogic Server contexts, see:

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

3. Instantiate remote interfaces for the data service. You can use either a static or dynamic mediator API interface. The dynamic interface is generic; the data service name is passed as an argument. For example:

   ```
   DataService ds = DataServiceFactory.newDataService(
             JndiCntxt, "RTLApp", "ld:DataServices/RTLServices/Customer");
   ```

   Here is the same operation using a static interface:

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

4. Invoke a function or procedure on the data service.

   The following is the operation using the dynamic interface to invoke a read function on a data service:

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

   Here is the same operation using a static interface:

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

Note:

Obtaining a WebLogic JNDI Context for Data Services Platform

Java client applications use JNDI to access named objects, such as data services, on a WebLogic Server. A single JNDI call is made to obtain an initial context, which is then passed to the data services factory class. 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>);
 InitialContext jndiCtxt = new InitialContext(h);
```

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

Invoking Functions and DSP Procedures

The Dynamic Mediator API provides two different methods (see Table 3-4) for invoking functions and procedures, respectively:

- **invoke( ).** Dynamically invokes read and navigate functions on a data service. When a read or navigate function is invoked (getCustomerByCustID( ), for example), the function returns an array of data objects.
- **invokeProcedure( ).** Use invokeProcedure() to invoke procedures that have been registered with a data service. You can use the dynamic Mediator API to invoke a DSP procedure as in the following example, passing in the name of the procedure:

  ```
  ds.invokeProcedure("updateCustomerAddress", params);
  ```

Table 3-4 Data Service Mediator API (Client Mediator API)

Method Signature	Description
`invoke(String method, Object[] args)`	Method to invoke a data service's read and navigate functions. Using invoke() with a DSP procedure raises an exception.
`invokeProcedure(String method, Object[] args)`	Method to invoke a data service's procedures (stored procedures, Web services, and Java code that have side effects). Using invokeProcedure() with a read or navigation function raises an exception.
`submit(DataObject sdo)`	Method to submit changes to the Mediator service. Assumes that a change summary exists as part of the DataObject.

In your code, you must use the appropriate method call — invoke() or invokeProcedure() — for the functions and procedures, respectively, to avoid raising exceptions, as noted in Table 3-4.

For more information, see information on Data Services API Javadocs at "DSP Mediator API Javadoc" on page 1-13.

When using static mediator APIs, the distinction between invoking a DSP function and a DSP procedure is hidden. Read and navigate functions, as well as procedures, are named based on the function name, with no indication as to whether or not they are side-effecting procedures.

Static and Dynamic Mediator APIs

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 static 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.

Using a Static Data Service Mediator API

To use the static data service interface, you must have the SDO Mediator Client JAR file that was generated from the specific 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 Java class file that will be the basis for your client application.

For example, to use a data service named Customer in a DSP project named customerApp, use the following import statement:

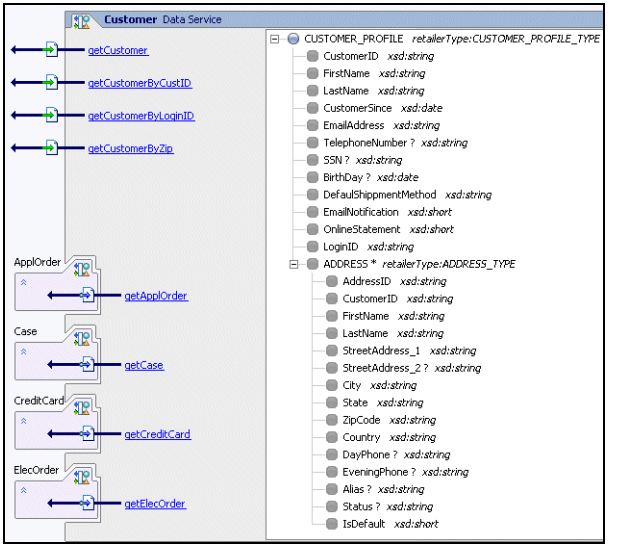
```
import customerapp.Customer;
```

With the imported factory classes and interfaces available in your Java application, you can instantiate the interface to the specific data service by invoking the getInstance() method with 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 and procedures on the data service. For example, consider the data service shown in Figure 3-5.

Figure 3-5 Customer Data Service



Based on the data service shown in Figure 3-5, the generated artifacts for a typed client interface would include static methods for both dynamic data APIs and the static Mediator APIs (see Listing 3-1). As shown in Listing 3-1, each read and navigate function from the data service results in a static data API method, such as getCustomer( ) and getApplOrder( ).

Listing 3-1 Generated Dynamic Methods for the Customer DataService Class

```
getCustomer()
getCustomerByCustID(String)
getCustomerByCustIDToFile(String, String)
getCustomerByZip(String)
getCustomerByZipToFile(String, String)
getCase(CUSTOMERPROFILEDocument)
getCreditCard(CUSTOMERPROFILEDocument)
getApplOrder(CUSTOMERPROFILEDocument)
getElecOrder(CUSTOMERPROFILEDocument)
getCustomerByLoginID(String)
```

See "Static Data API" on page 2-5 for more information about generated SDO data API methods, such as those listed above (getCustomer() and get CustomerByLoginID(), for example).

There are several DataService methods that are part of the dynamic API which are inherited by all static DataService classes. These include:

- **Submit( ) method.** The submit( ) method takes a DataObject as its parameter. (The static submit () would take Customer.) In either style, though, a submit() method is used to save changes to the data objects served by the data service.
- **The prepareExpression( ) method**. The prepareExpression() method lets you create ad hoc queries against the data service.

Listing 3-2 shows a small but complete example of using the static interface.

Listing 3-2 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.CUSTOMERPROFILEDocument;

public class MyTypedCust
{
  public static void main(String[] args) throws Exception {
    //Get access to DSP data service
    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 typed Mediator API
    Customer customerDS = Customer.getInstance(context, "RTLApp");
    CUSTOMERPROFILEDocument[] myCust =
        customerDS.getCustomerByCustID("CUSTOMER2");
    System.out.println(" CUST" + myCustomer);
  }
}
```

Using a Dynamic Mediator API

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.

With the dynamic interface, names of specific data services are passed as parameters in the generic get() and set() method calls. For example:

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

A data object returned by the dynamic interface can be downcast to a static object, as follows:

```
DataService ds =
    DataServiceFactory.newDataService(
           context, "RTLApp", "ld:DataServices/Customer");
 Object[] params = {"CUSTOMER2"};
 CUSTOMERDocument myCustomer =
```

```
                    (CUSTOMERDocument) ds.invoke("getCustomer", params);
System.out.println(myCustomer.getCUSTOMER().getCUSTOMERNAME() );
```

Note: This code example only works if the generated static SDO mediator JAR is on the classpath at compile time and at runtime.

For a 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 3-3 shows a full example.

Listing 3-3 Mediator Client Sample Using the Dynamic Mediator API Data Service Interface

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

```
    }
}
```

Static and Dynamic SDO APIs

You can invoke data service functions using either static or dynamic SDO APIs. The dynamic API is often called generic SDO, since you do not need to materialize the SDO object on the client side through a JAR file. Instead, you simply invoke the appropriate set( ) or get( ) method based on your knowledge of underlying schema of the data service.

Each approach has its advantages and disadvantages, as described in the next table:

Table 3-6 Static vs. Dynamic Mediator APIs

Method	Advantages	Disadvantages
Static (typed)	• Runtime type validation • Code completion in most IDEs	• Requires generation of *[App]*`-ld-client` JAR file
Dynamic (untyped), using generic SDO	• Easily adapt to schema changes • Unnecessary to compile Java classes from their schema • Less overhead	• No runtime type checking

The static and dynamic SDO API options are described in detail in:

Using a Static Data Service Mediator API

Using a Dynamic Mediator API

Using the Static SDO API

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 static data objects. (Alternatively, the generic SDO dynamic interface lets you use data services specified at runtime. It is described under the topic "Using a Dynamic Mediator API" on page 3-12.)

A 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 specific DSP-enabled application that contains the query functions you want to use with your client 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 Java class file that will be the basis for your client application.

For example, to use a data service named Customer in a DSP project named customerApp, use the following import statement:

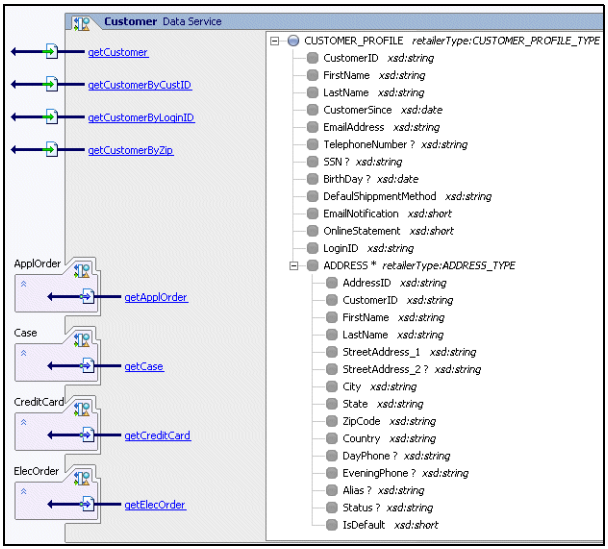
```
import customerapp.Customer;
```

With the imported factory classes and interfaces available in your Java application, you can instantiate the interface to the specific data service by invoking the getInstance( ) method with 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 and procedures on the data service. For example, consider the data service shown in Figure 3-5.

Figure 3-7 Sample Customer Data Service



The generated artifacts for a static client interface would include typed methods for both dynamic data APIs and the static Mediator APIs. As shown in Listing 3-1, each read and navigate function from the data service results in a static data API method, such as getCustomer( ) and getApplOrder( ).

Listing 3-4 Generated Dynamic Methods for the Customer DataService Class

```
getCustomer()
getCustomerByCustID(String)
getCustomerByCustIDToFile(String, String)
getCustomerByZip(String)
getCustomerByZipToFile(String, String)
getCase(CUSTOMERPROFILEDocument)
getCreditCard(CUSTOMERPROFILEDocument)
getApplOrder(CUSTOMERPROFILEDocument)
getElecOrder(CUSTOMERPROFILEDocument)
getCustomerByLoginID(String)
```

See "Static Data API" on page 2-5 for more information about generated SDO data API methods, such as those listed above.

There are several DataService methods that are part of the dynamic API which are inherited by all static DataService classes including the following methods:

- **Submit( ).** The submit( ) method takes a DataObject as its parameter. (The static submit() would take Customer.) In either style, though, a submit() method is used to save changes to the data objects served by the data service.
- **prepareExpression( ).** The prepareExpression( ) method lets you create ad hoc queries against the data service.

Listing 3-2 shows a small but complete example using a static interface.

Listing 3-5 Mediator Client Sample Using a Static Interface to a Data Service

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import dataservices.rtlservices.Customer;  //
import retailerType.CUSTOMERPROFILEDocument;

public class MyTypedCust
{
  public static void main(String[] args) throws Exception {
    //Get access to DSP data service
    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 typed Mediator API
    Customer customerDS = Customer.getInstance(context, "RTLApp");
    CUSTOMERPROFILEDocument[] myCust =
        customerDS.getCustomerByCustID("CUSTOMER2");
    System.out.println(" CUST" + myCustomer);
  }
}
```

Using the Dynamic SDO API

The dynamic data service interface — or *generic SDO* — is ideal for programming with data services that are unknown or do not exist at development time.

With generic SDO, DataObjects depend on the XML schema to determine:

- Data types
- Default values
- Data structure of input XML data

The generic SDO correctly supports SDO APIs through get( ) and setType( ) on DataObject.

Such a SDO definition consists of a single generic DataGraph and a number of DataObject classes.

Generic SDOs are created using a createRootDataObject( ) method:

```
DataObject /* root SDO document */ createRootDataObject()
```

The code fragment in Listing 3-6 illustrates these familiar operations using generic SDO:

- Creating the root data object
- Using navigation functions
- Updating data on the back end
- Submitting a changed SDO to the server
- Deleting a data object
- Creating a data object on the client side

Listing 3-6 Common Generic SDO Operations

```
DataService custDS =
DataServiceFactory.newDataService(context,"RTLApp","ld:DataServices/CustomerDB
/CUSTOMER");
DataObject root =(DataObject)custDS.invoke("getCustomerByCustID", new
Object[]{"CUSTOMER1"})[0];
DataObject myCustomer = root.getDataObject(0);
String name = myCustomer.getString("name");

//use navigation function
DataObject order = (DataObject)custDS.invoke("getApplOrder", new
Object[]{root})[0];

// update
myCustomer.setString("Street","Lake Drive");
((DataObject)myCustomer.getList("ADDRESS").get(0)).setString("City",
"Hayward");
```

```
// submit the changed SDO to server
custDS.submit( root );

// Delete a DataObject
((DataObject)myCustomer.getList("ADDRESS").get(0)).delete();
custDS.submit( root );

// create new SDO object on client side
DataService custDS =
DataServiceFactory.newDataService(context,"RTLApp","ld:DataServices/CustomerDB
/CUSTOMER");

DataObject root = custDS.createRootDataObject ();
root.createDataObject("CUSTOMER_PROFILE").setString("FirstName","helloW");
custDS.submit( root );
```

How XML Schemas Are Made Available to Generic SDO Operations

XML schemas are not available at client side, the dynamic mediator must download schemas from the server. The internal mean of downloading schemas varies depending on whether you have an EJB client or a Web service client.

- **Downloading schemas through an EJB metadata API.** As in the first code fragment in Listing 3-6, the client-side query contains a Qname and arity as an identifier for the function. A generic SDO-capable mediator uses the identifier to locate the primary schema from the metadata EJB, then recursively resolves its dependent schemas, loading them from server to client upon request. Once all schemas are prepared on the client side, the schemas are processed (compiled) and made available to the calling routine.
- **Downloading schemas through a Web service client.** A WSDL file generated from WebLogic Workshop contains all the schema definitions in the specified data service. In this case they are simply processed and made available to provide typing services to the calling routine.

Schema Type Caching

Generating the SchemaTypeSystem can be a costly operation. For this reason schema caching functionality is provided that allows for schema reuse and lifecycle management through flush and clear APIs.

If no cache is passed in to get a data service instance on the client side, then an internal cache is created. The default lifetime of the internal schema cache is the same as the lifetime of the data service instance.

You can create a cache object per data service or for multiple data services. All caches are thread-safe. (As there is no differentiator across multiple DSP applications, caches should not be shared across multiple applications.)

The following schema caching APIs are available:

- **SchemaTypeObject cache.** The SchemaTypeObject is composed of a key:value pair.

  key: Qname composed of root element name and target namespace
  value: compiled schema type object

- **Cache debugging APIs.** The following APIs are more fully described in Javadoc:

```
SchemaTypeCache.dump( String dsName ) //dump contents for specified DS
SchemaTypeCache.dump();                //dump entire contents
```

- **Flush schema cache APIs.** The following APIs are more fully described in Javadoc:

```
public void flush()
public void clear( String dsName )
```

Schema Cache Management Scenarios

The following represents several schema cache management scenarios:

- **Using multiple caches.** In this case each data service has its own schema type.

```
// Note: each DS will have its own schema type cache.

SchemaTypeCache custSchemaTypes = new SchemaTypeCache();
SchemaTypeCache addrSchemaTypes = new SchemaTypeCache();
DataService custDS = DataServiceFactory.newDataService( context,
   "RTLApp",
"ld:DataServices/Customer",
custSchemaTypes );
 DataService addrDS = DataServiceFactory.newDataService( context,
       "RTLApp",
"ld:DataServices/Address",
addrSchemaTypes );
```

- **Using a single cache for multiple** data service**s.** In this case a single cache is used across multiple data services.

```
// Note: User manages cache across multiple DS's.

  SchemaTypeCache schemaTypes = new SchemaTypeCache();
 DataServicecustDS=DataServiceFactory.newDataService(context,
       "RTLApp",
"ld:DataServices/Customer",
schemaTypes );
```

```
DataServiceaddrDS=DataServiceFactory.newDataService(context,
      "RTLApp",
"ld:DataServices/Address",
schemaTypes);
```

- **No schema cache specified.** In such a case a cache is created implicitly and stored on the data service object automatically. There is no API available to inspect, flush, or edit cache entries.

```
DataService custDS = DataServiceFactory.newDataService( context, "RTLApp",
"ld:DataServices/Customer" );
DataService addrDS = DataServiceFactory.newDataService( context, "RTLApp",
"ld:DataServices/Address" );
```

Bypassing the Cache When Using the Mediator API

Data retrieved by data service functions can be cached for quick access. (See "Configuring the Query Results Cache", in the DSP *Administration Guide* for details.) Assuming the data changes infrequently, it's likely that you'll want to use the cache capability. However, you can bypass the cache and obtain data directly from the data sources by passing the GET_CURRENT_DATA attribute within a function call, as shown in Listing 3-7. As a by-product, the cache is also refreshed.

Listing 3-7 Cache Bypass Example When Using Mediator API

```
DataService ds = DataServiceFactory.newDataService(
import com.bea.dsp.RequestConfig;
getInitialContext(),                                    // Initial Context
"Evaluation",                                           // Application Name
"ld:DataServices/CustomerManagement/CustomerProfile" // Data Service URI
);
Object[] params = {"CUSTOMER3"};
RequestConfig config = new RequestConfig();
attr.enableFeature(RequestConfig.GET_CURRENT_DATA);

CustomerProfileDocument doc = (CustomerProfileDocument)
ds.invoke("getCustomerProfile",params.config);
```

Step-by-Step: A Java Client Programming Example

This section describes common Java client application programming tasks:

- Step 1. Instantiating and Populating Data Objects

- Step 2: Accessing Data Object Properties
- Step 3: Modifying, Adding, and Deleting Data Objects and Properties
- Step 4: Submitting Changes to the Data Service

Client application development encompasses the SDO data APIs; client Mediator APIs (which are used to instantiate a local proxy to the remote server); and possibly the Update SDO API (to submit changed data objects to the data service). Thus, the steps in this section include calls using the Mediator APIs—getInstance() and submit(), for example, as well as SDOs.

Step 1. Instantiating and Populating Data Objects

Working with SDO data objects from a client application starts by obtaining an interface (either static or dynamic) to the data service. Depending upon the approach you take, you must import the generated static data type interfaces or dynamic data interfaces, as well as the data service interfaces.

- **Static data service imports**. To instantiate a data object using a static data service instance, you must import the packages that contain the generated typed interfaces. These are contained in the `<appname>-ld-client.jar` file generated from WebLogic Workshop (or by using DSP's Ant or Java generation tools). Using the static SDO API is a two-step process:
  - Place the JAR file (`<appname>-ld-client.jar`) in the classpath of your development environment.
  - Import the data types that you will be using in your code into your Java class file. For example:

    ```
    import dataservices.myservice.MyCustomer;
    ```

    Note: Static data services packages are always lowercase.
- **Dynamic data service imports**. To instantiate a data object using a dynamic API, you must import the DataServiceFactory class and invoke the newDataService method (see Table 3-8).

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

Table 3-8 Static and Dynamic Mediator API Interfaces

Static Mediator API	Dynamic Mediator API
`Customer cust = Customer.getInstance( context, "MyApp");`	`DataService ds = DataServiceFactory.newDataService( context, "MyApp", "ld:DataServices/CustomerDB/CUSTOMER");`

Instantiating a local interface for an static mediator API is done by passing the context, the application name, and the data service name to the DataServiceFactory class. For the static mediator API, the local interface is instantiated using the getInstance() method (after establishing a JNDI context).

Once the local interface is constructed, you can invoke data service functions to obtain a data object.

As discussed in "Data Services Platform and Service Data Objects (SDOs)" on page 2-2, the returned data object is associated with a data graph. The data graph also provides a handle to the root data object of the data graph.

Table 3-9 shows both a static and dynamic approach to populating data objects. The static data API example shows how to instantiate the root node of the data graph, in this case, using the data that comprises a logical data service function (getCustomerView( )). The example is selecting information about Customer3.

In the dynamic example, the root node of a data graph is being populated with an array of all customers available through the data service.

Table 3-9 Static and Dynamic Mediator APIs to Instantiate Data Objects (SDOs)

Static Mediator API	Dynamic Mediator API
`CUSTOMERDocument[] custDoc = ds.getCustomerView("CUSTOMER3");`	`CUSTOMERDocument [] custDoc = (CUSTOMERDocument[]) ds.invoke("CUSTOMER", null);`

Step 2: Accessing Data Object Properties

After obtaining a data object, you can access its properties using either its generated static data API or the dynamic data API. Table 3-10 shows side-by-side comparisons of using the static and dynamic

methods to access properties. The static interface returns a single CUSTOMER object, while the dynamic interface returns a generic data object.

Table 3-10 Static and Dynamic Mediator API Property Acquisition Examples

Static Mediator API	Dynamic Mediator API
`CUSTOMERDocument.CUSTOMER cust = custDoc[0].getCUSTOMER(); String lastName = cust.getLASTNAME();`	`CUSTOMERDocument.CUSTOMER cust = (CUSTOMERDocument.CUSTOMER)cust-Doc[0].get("CUSTOMER"); String lastName = (String) cust.get("LAST_NAME");`

With the static interface, the type name (as a string) is passed as a parameter to the dynamic get( ) method. The returned object can be then cast to the necessary type.

If the return type is unbounded, you need to cast the returned object to a List. To traverse all objects in an unbounded type you must use an iterator, as shown in Listing 3-8.

Listing 3-8 Using an Iterator to Traverse a List of Returned Data Objects

```
List addressList = (List) cust.get("ADDRESS");
Iterator iterator = addressList.iterator();
while ( iterator.hasNext() ){
       CUSTOMERDocument.CUSTOMER.ADDRESS address =
          (CUSTOMERDOcument.CUSTOMER.ADDRESS) iterator.next();
       }
}
```

You can identify properties in SDO accessor arguments by element name. Accessor methods can 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 element identifiers have identical values, all elements are returned.

For example, the ADDRESS 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 unbounded objects, you must use a List.)

Note: For specifying index position, note that SDO supports regular XPath notation (one-based) and Java-style (zero-based). See "XPath Support in the Dynamic Data API" on page 2-11 for more information.

You can get a data object's containing parent data object by using the get() method with XPath notation:

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

You can get the root containing the data object by using the get() method with XPath notation:

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

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

Step 3: Modifying, Adding, and Deleting Data Objects and Properties

By default, change tracking on the data graph is enabled so that any changes made to object values are recorded in the *change summary*.

Modifying Data Object Properties

You can modify data object property values using either dynamic or static set() methods.

Table 3-11 Examples of Static and Dynamic Mediator API Setting of Properties

Static Mediator API	Dynamic Mediator API
`cust.setLASTNAME("Smith");`	`cust.set("LAST_NAME", "Smith");`

Both approaches take string arguments for the new property values; both approaches result in changing the customer object's last name to Smith. The static mediator API example assumes that you have instantiated the static interface on the data service.

Adding New Data Objects

You can create new a data object by using an addNew() method (a static data API). A new data object can be added to a root data object or, more commonly, as a new element in a data object array. (New arrays can also be added to data objects.) When adding an object to an array, you must be sure to set any and all required fields for the new object, as specified by its XML schema, before calling submit().

Listing 3-9 shows how to add a data object to an array of objects.

Listing 3-9 Adding a New Data Object to an Array

```
CUSTOMERDocument.CUSTOMER newCust = custDoc[0].addNewCUSTOMER();
    int idNo = custDoc.length;
    newCust.setCUSTOMERID("CUSTOMER" + String.valueOf(idNo));
    newCust.setFIRSTNAME("Clark");
    newCust.setLASTNAME("Kent");
    newCust.setCUSTOMERSINCE(java.util.Calendar.getInstance());
    newCust.setEMAILADDRESS("kent@dailyplanet.com");
    newCust.setTELEPHONENUMBER("555-555-5555");
    newCust.setSSN("509-00-3683");
    newCust.setDEFAULTSHIPMETHOD("Air");
```

If the data source associated with the object being added is an RDBMS, note these additional considerations:

- 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 often use 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.

  If added objects 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.

  See "Primary-Foreign Key Relationships Mapped Using a KeyPair" on page 2-20 for more information.

Deleting Data Objects

To delete a data object, you must delete it from the data graph that contains it. For example, Listing 3-10 searches a CUSTOMER array for a specific customer's name and deletes that customer.

Listing 3-10 Deleting a Data Object

```
CUSTOMERDocument.CUSTOMER[] custs =
         custDoc[0].getCUSTOMERArray();
             for (int i=0; i < custs.length; i++){
                   if (custs[i].getFIRSTNAME().equals("Clark") &&
                        custs[i].getLASTNAME().equals("Kent"))
          {
            custs[i].delete();
            custDS.submit(custDoc);
         }
    }
```

When you remove an object from its container, only the reference to the object is deleted, not the values; values are deleted later, during Java garbage collection.

The data object interface (*DataObject* in the `commonj.sdo` package) provides the delete() method for deleting objects.

Deleting an object is a cascade-style operation; that is, children of the deleted object are deleted as well. However, note that the deleted object only—not its children—is tracked in the change summary as having been deleted.

Step 4: Submitting Changes to the Data Service

To submit data changes, call the submit() method on the data service bound to an object, passing the root changed object as in:

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

A basic example of a submit operation is shown in Listing 3-11.

Listing 3-11 Static Interface

```
CUSTOMER custDS = CUSTOMER.getInstance(ctx, "RTLApp");
CUSTOMERDocument[] custDoc = (
   CUSTOMERDocument[]) custDS.CUSTOMER();
custDoc[0].getCUSTOMER().setLASTNAME("Nimble");
custDS.submit(CustDoc);
```

Listing 3-12 demonstrates making changes to a data object using the dynamic interface.

Listing 3-12 Dynamic Interface

```
DataService ds = DataServiceFactory.newDataService(new InitialContext(),
"RTLApp");
DataObject[] custDoc = (DataObject[])custDS.invoke("CUSTOMER", null);
custDoc[0].getCustomer().set("LastName", "Nimble");
custDS.submit(myCust,"ld:DataServices/CustomerDB/CUSTOMER");
```

Examining a Java Client Application

Listing 3-13 shows a complete example that recaps many of the steps described above. The example SDO client application shows how the static mediator API is used to create a handle to the CUSTOMER data service.

The client application extracts information about a customer, modifies the information, and then submits the changes. In addition to demonstrating some of the basics of SDO client programming, Listing 3-13 also shows how the Mediator API is used to obtain a handle to the data service, and how the Update Mediator API is used to submit the changes to the data service.

Listing 3-13 Sample Client Application

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

      // create a handle to the Customer data service
     CUSTOMER custDS = CUSTOMER.getInstance(context, "RTLApp");
     // use dynamic data API to instantiate an SDO (shaped as a "Customer")
     CUSTOMERDocument[] myCustomer =
            (CUSTOMERDocument[]) custDS.invoke("CUSTOMER", null);

     // get and show customer name
     String existingFName =
              myCustomer[0].getCUSTOMER().getFIRSTNAME();
     String existingLName =
            myCustomer[0].getCUSTOMER().getLASTNAME();

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

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

     // re-query and print new name

     myCustomer = (CUSTOMERDocument[]) custDS.invoke("CUSTOMER",null);
     String newFName =
              myCustomer[0].getCUSTOMER().getFIRSTNAME();
     String newLName =
            myCustomer[0].getCUSTOMER().getLASTNAME();

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

Listing 3-13 highlights how to use the SDO data APIs and the Mediator API, as follows:

1. The application instantiates the remote interface to the Customer data service, passing a JNDI context that identifies the WebLogic Server where DSP is deployed. The static Mediator API is

used in this call to instantiate the actual Customer data service interface (rather than the generic DataService interface):

```
CUSTOMER custDS = CUSTOMER.getInstance(context, "RTLApp");
```

The custDS serves as a handle for the CUSTOMER data service that is executing on the RTLApp WebLogic Server application.

2. The program uses the Mediator API to invoke a read function on the Customer data service, pouring the results into an array of CUSTOMERDocument objects:

```
CustomerDocument[] myCustomer =
               ( CustomerDocument[]) ds.invoke("CUSTOMER", null);
```

3. Once the data object is created, its properties can be accessed using SDO's static data API (the static interface), which returns the actual type of that node:

```
myCustomer[0].getCUSTOMER().getFIRSTNAME();
```

4. New values for the FIRSTNAME and LASTNAME property of the CUSTOMER are set using the static data API:

```
myCustomer[0].getCUSTOMER().setFIRSTNAME("J.B.");
myCustomer[0].getCUSTOMER().setLASTNAME("Kwik");
```

5. The change is submitted to the data service (by using the Client Mediator API's submit() method) for propagation to the back-end data sources:

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

6. The Mediator API's invoke() method is executed once more, and the results (now showing the changed data) are printed to output.

Note: The invoke() method is for read and navigation functions only. For data service procedures, use the invokeProcedure() method available in the DataService interface. For details on the Mediator API see DSP Javadoc, described under "DSP Mediator API Javadoc" on page 1-13.

See "Invoking Functions and DSP Procedures" on page 3-8 for more information about procedures.

Although code for handling exceptions is not shown in the example, an SDO runtime error throws an `SDOMediatorException`. The `SDOMediatorException` class is also used to wrap data source exceptions.

CHAPTER 4

Web Services and DSP-Enabled Applications

Web services provide an industry-standard way to develop SOA (service-oriented architecture) applications—loosely coupled, distributed units of programming logic that can be re-configured easily to deliver new application functionality, both intra- and extra-enterprise. By wrapping your data services as Web services, you enhance the Web services model with a data services layer that can abstract a wide variety of data sources, including relational database management systems, workflow applications, portals, and other Web services.

In short, using Web services and BEA AquaLogic Data Services Platform (DSP) together lets you leverage all your data assets. This chapter shows you how to expose data services as standard Web services, and how to create client applications that can obtain the benefits of both Web services and SDOs. It covers these topics:

- Overview of Web Services and DSP
- Server-side DSP-Enabled Web Service Development
- Client-side DSP-Enabled Web Service Development

Overview of Web Services and DSP

Exposing data services as Web services makes your information assets accessible to a wide variety of client types, including other Java Web service clients, Microsoft ADO.NET and other non-Java applications, and other Web services (see Figure 4-1).

Figure 4-1 Web Services Enables Access to DSP-Enabled Applications from a Variety of Clients

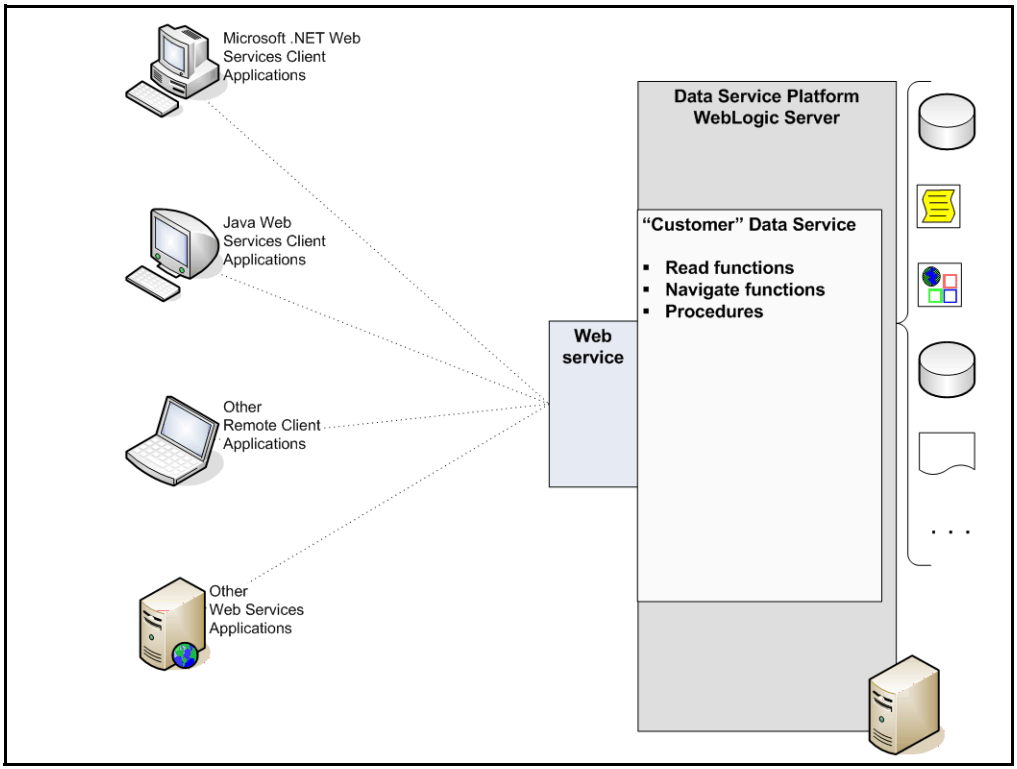


Figure 4-1 illustrates the various approaches that client application developers can take to integrating data services and Web services.

Note: This chapter focuses on leveraging DSP-enabled applications as Web services, and on accessing such DSP-enabled Web services from Java client applications. For information about ADO.NET-enabled Web services and client applications, see "Supporting ADO.NET Clients" on page 6-1.

Different Styles of Web Services Integration for DSP

DSP-enabled applications can be integrated with Web services in one of two general ways:

- **As a standard Web service**. A standard Web service can be invoked from other Web services, or by .NET clients or any other type of client, Java and non-Java alike. On the server side, at runtime, the Web service simply passes the results obtained from the data service function back

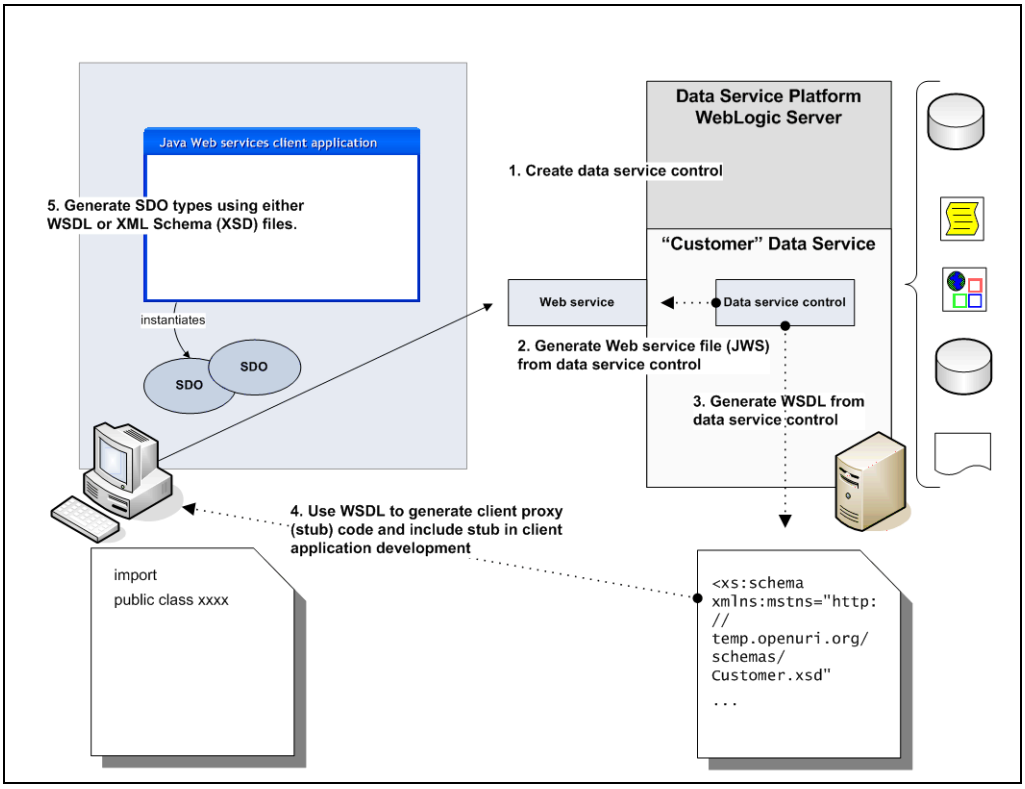
to the client as a standard SOAP message. This approach is best for simple query only applications that do not need to modify or add data to back-end data sources behind the Web service facade.

- **As an SDO-Enabled Web service**. An SDO-enabled Web service can support updates to back-end data sources. You can use either static SDOs client-side proxy code (as detailed "Generating SDO-Enabled Web Services Clients" on page 4-18), or use dynamic SDOs (also known as *generic SDO*), which are automatically downloaded to the client at runtime.

  Note: For details on working with static and dynamic SDO see "Static and Dynamic SDO APIs" on page 3-14.

This chapter covers SDO-enabled Web service client applications, starting with the server-side development tasks required to expose DSP-enabled applications as Web services. Figure 4-2 shows the end-to-end process—both the server-side and client-side tasks—that expose a DSP-enabled application as a Web service and implement a client application that invokes operations on that service.

Figure 4-2 Java Clients Supported via Web Services



Server-side DSP-Enabled Web Service Development

There are two ways to easily integrate data services with Web services:

- Adding a Data Service Control to a Web Service
- Generating a Web Service from a Data Service Control

Both approaches, covered in the next two sections, rely on Data Service controls as the component-based integration mechanism.

Adding a Data Service Control to a Web Service

You can easily add one or more Data Service controls to a Web service using WebLogic Workshop. You must first create a folder for the controls inside the Web service's project folder, and then create the

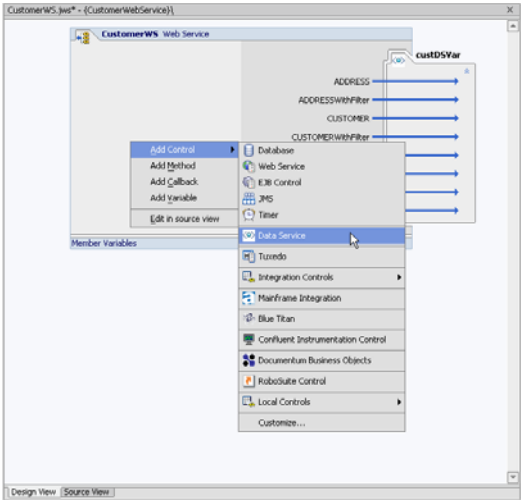
Data Service controls. The controls must be placed inside the controls folder so they will be available to add to the Web service, as instructed in this section.

Note: You can also create controls during the process of adding them to the Web service, but for simplicity's sake, the instructions in this section assume that you have created the Data Service controls in advance. (See "Creating Data Service Controls" on page 5-8 for more information about creating Data Service controls.)

a. In WebLogic Workshop, open the existing Web service file (JWS) by double-clicking on its name in the Application pane.

b. Click the Design View tab on the Web service to open the graphical representation of the Web service (as shown in Figure 4-3).

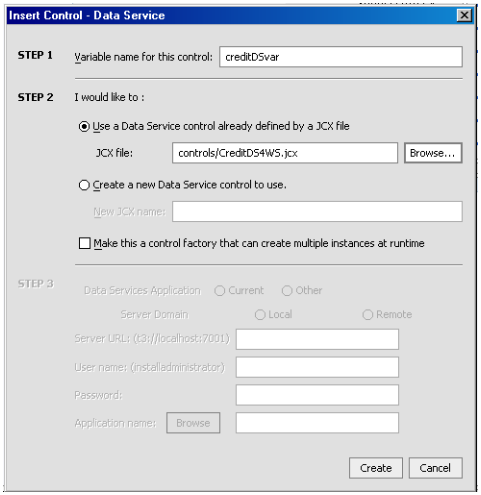
Figure 4-3 Adding a Data Service Control to a Web Service



c. Right-click and select Add Control →Data Service from the popup menu. The Insert Control – Data Service wizard launches, showing the multi-step dialog page shown in Figure 4-4.

d. In the STEP 1 field of the dialog, enter a meaningful variable name for the Data Service control that is unique in the context of the Web service.

Figure 4-4 Insert Control – Data Services Wizard



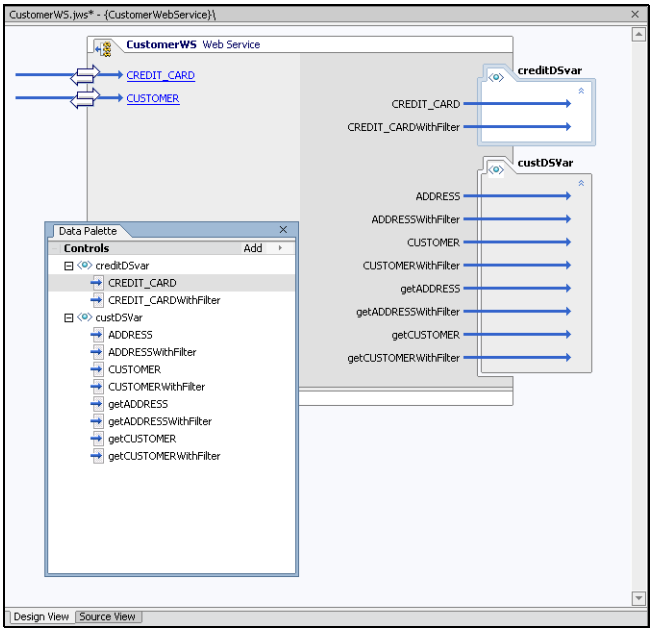
e. In the STEP 2 field, click Browse... to navigate to the controls folder, then select the Data Service control you want to add to the Web service. (Alternatively, click Create a New Data Service Control button to launch the Data Service control wizard to create and configure a new control.)

   Leave the Make This a Control Factory checkbox de-selected: This checkbox will cause the Data Service control to be instantiated at runtime using the factory pattern, rather than as a singleton. To use the control in a Web service, it must be a singleton.

f. In the STEP 3 section of the dialog (which will be active only if your Data Service control is associated with a remote DSP instance, that is, a DSP instance running on a separate domain from WebLogic Workshop), provide the user name, password, server URL, and domain information associated with the remote Data Service control to complete the link between the Web service and the control.

g. Click the Create button on the Insert Control – Data Service dialog. The `LiquidDataControl.jar` file is copied into the Libraries directory of the application, and the variable you created in STEP 1 of the dialog displays as a node in the Data Palette, with its functions and procedures listed under the node. It is these functions and procedures that you can now expose to client applications, by adding them to the Web service's callable interface (shown as the left-hand portion of the Web service's Design View in WebLogic Workshop — see Figure 4-5), as described in the next step.

Figure 4-5 Adding Functions from a Data Service Control to a Web Service



h. Select the function or procedure from under the variable name listed in the Data Palette by clicking on the node, and then drag and drop the function onto the left side of the Web service in Design View.

When you are finished, you can test the Web service as described in "Testing a Web Service in WebLogic Workshop" on page 4-9. After testing, you can deploy to your production WebLogic Server and use it as you would any other Web service. For more information about Web services, see:

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

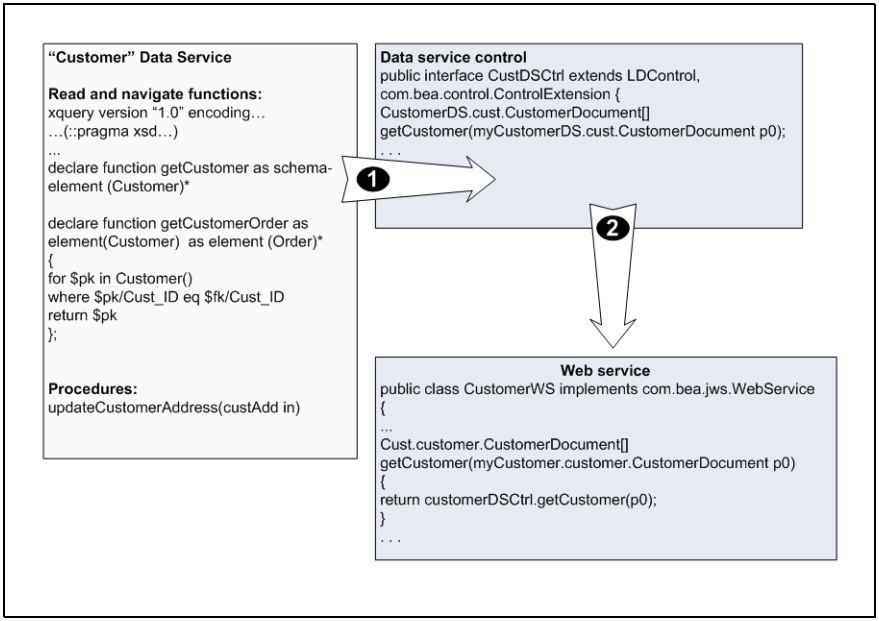
For information about developing Java-based Web service clients, see "Client-side DSP-Enabled Web Service Development" on page 4-11.

Generating a Web Service from a Data Service Control

Using WebLogic Workshop you can generate stateful or stateless (conversational) Web services from Data Service controls. The generated Web services include method calls (referred to as operations) for each of the functions and procedures that the Data Service control comprises, as well as two operations specifically for testing the Web service.

Follow the instructions in this section to generate and test a stateless Web service. The instructions assume that you have already created the Data Service control and that WebLogic Workshop is open.

Figure 4-6 Stateless Web Services Are Generated from Data Service Controls



1. From WebLogic Workshop's Application pane, select the Data Service control that you want to use as the basis for your Web service by clicking on its name. While the control is selected, right-mouse-click to display the pop-up menu; select Generate Test JWS File (Stateless) from the menu. WebLogic Workshop generates the JWS Java Web service file for your Data Service control.

Note: Note that although WebLogic Workshop by default generates Web services that have the word "Test" embedded in the file names, these are deployable Web services. You can rename the generated Web service to eliminate the word "Test" from its name.

2. Click on your Web service project to select it, right-click, and select Build Project. WebLogic Workshop builds a Web service project.

3. When the build process completes, double-click on the `.jws` file to open it. Click the Design View tab if necessary to display the generated Web service in the Design View.

   You will see methods (operations) for each of the functions and procedures contained in the Data Service control, as well as two additional operations, startTestDrive() and finishTestDrive(). You can use these two operations to quickly test the Web service (using

WebLogic Workshop's runtime server), as described in Testing a Web Service in WebLogic Workshop. Before testing, however, you should modify any submit() operations in your generated Java Web service, as described in the next section.

Modifying Submit Operations and Generating a WSDL File

If the Web service must support submits from Java Web service clients, you must modify the JWS file before generating the WSDL, as follows:

1. Modify submit operations in your Java Web service (JWS) implementation control file to accept a DatagraphDocument object as a parameter. The signature for the submit() method should be similar to:

   ```
   java.util.Properties[] submitCustomerProfile(DatagraphDocument
    rootDataObject)
   ```

2. Modify the body of the submit operation to instantiate and initialize the document from a DatagraphDocument object being passed as a parameter; for example:

   ```
   CustomerProfileDocument doc = (CustomerProfileDocument) new
        DataGraphImpl(rootDataObject).getRootObject();
   return customerData.submitCustomerProfile(doc);
   ```

The following code shows the context of a method declaration.

```
public java.util.Properties[] submitCustomerProfile(DatagraphDocument
rootDataObject) throws Exception  {
CustomerProfileDocument doc = (CustomerProfileDocument) new DataGraphImpl(
                                  rootDataObject).getRootObject();
return customerData.submitCustomerProfile(doc);
}
```

3. Generate a Web Service Definition Language (WSDL) file from the JWS file by right-clicking on on the file name and selecting the Generate WSDL file option.

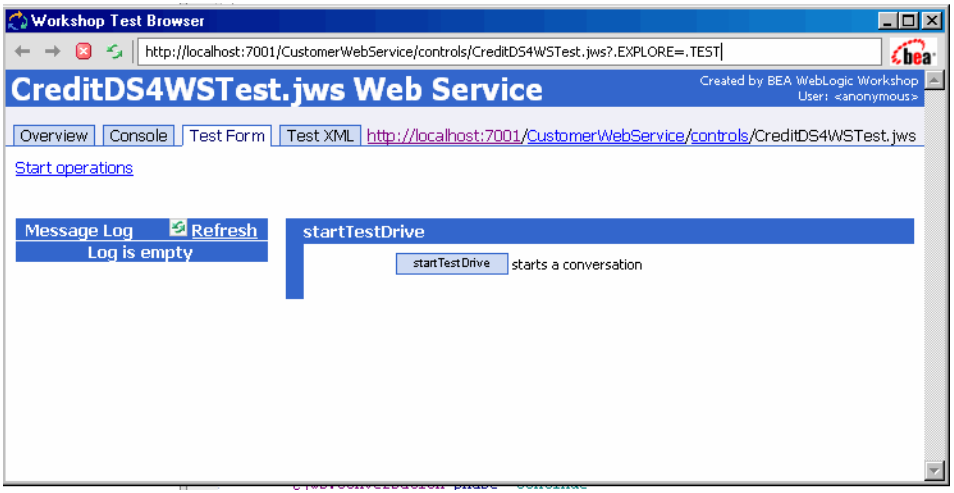
After you have created the WSDL file, provide it to client application developers, so they can generate the Web services client interfaces and proxy code necessary (as discussed in "Client-side DSP-Enabled Web Service Development" on page 4-11).

Testing a Web Service in WebLogic Workshop

By default, WebLogic Workshop creates two operations in its generated Web services that can be used for testing purposes.

1. Click the Start icon (or select Debug →Start from the WebLogic Workshop menu) to deploy and run the Web service using the local runtime. An informational message briefly appears, notifying you that the Web service is running. Shortly, the WebLogic Workshop Test Browser launches, displaying the Test Form, as shown in Figure 4-7.

Figure 4-7 WebLogic Workshop Test Browser



2. Click the startTestDrive button to start the conversation for the Web service.

3. Click the Continue this Conversation link (in the left corner of the test page). The available operations display as buttons on the page, along with informational messages.

4. 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 returns results to the test browser.

5. 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.

6. To end the Web service conversation, click the Continue this Conversation link (to redisplay the Test Form page) and then click the finishTestDrive button.

Continue developing the functionality of the Web service as required, testing as you go along. Once the Web service is complete, you can create the artifacts necessary for client application development, as described in the next section, "Client-side DSP-Enabled Web Service Development."

Note: For more information about Web service client applications and WebLogic Server in general, see "Invoking Web Services" in *Programming WebLogic Web Services* in the WebLogic Server documentation.

Client-side DSP-Enabled Web Service Development

There are many different approaches to developing Web service client applications, but the choices in any given instance are limited by the type of Web service itself. For example, if the Web service is an ADO.NET-Enabled Web service hosting data service functions, the assumption is that a Microsoft ADO.NET client Web service application will be using the Web service—not a Java client.

Another consideration is whether the client application will use the static or dynamic approach to Web services, as follows:

- **Static Web service client**. Uses typed client classes (derived from Stub base class) to invoke Web service operations on local proxy. DSP includes utilities (Java classes and Ant tasks) to generate the following code:
  - SDO client classes (for typed clients)
  - Web service client proxy (stub) that can instantiate and manipulate SDOs
- **Dynamic Web service client**. Uses Call interface from a client to dynamically invoke Web service operations. In this context *dynamic* means late-binding.

Either approach can be used with DSP-enabled Web service applications. Listing 4-1 shows an example of a Java Web service client application that invokes several operations on a DSP-enabled Web service. The example implements the static Web service client model, and demonstrates how to marshal data from Java for a SOAP request, by serializing an SDO DataGraph:

```
wssoap.submitCustomer(((DataGraphImpl)doc.getDataGraph()).getSerializedDocumen
t());
```

At runtime, DSP uses a codec (DataGraphCodec, an encoder-decoder class that extends AbstractCodec) behind the scenes to:

- serialize SDO DataGraphs when marshalling data for SOAP requests (Java-to-SOAP)
- de-serialize SDO DataGraphs when SOAP messages are un-marshaled (SOAP-to-Java)

Listing 4-1 Sample Java Client

```
public class ClientTest {
```

```
  public static void main(String[] args) throws Exception {
        SimpleCtrlTest wstest = new SimpleCtrlTest_Impl();
        SimpleCtrlTestSoap wssoap = wstest.getSimpleCtrlTestSoap();
        CUSTOMERDocument doc = wssoap.getCustomer(987654);

        doc.getCUSTOMER().setCUSTOMERNAME("323777");
        String result = doc.getDataGraph().toString();
        System.out.println(result);
wssoap.submitCustomer(((DataGraphImpl)doc.getDataGraph()).ge
tSerializedDocument());
}
```

Data Services Platform automatically downloads the typed client-side artifacts upon first invocation of a Web service operation, as required.

Client-side Artifact Generation Utilities

Data Services Platform provides both Ant tasks and Java classes for generating the various artifacts required for DSP-enabled Web service client development (see Table 4-8). The Ant tasks can be thought of as wrappers around the Java classes to which they refer, and provide a simple way to incorporate the Java classes into an Ant build script.

Table 4-8 DSP's Web Service Client Utilities Summary

Utility	Description	Classname
sdogen	Ant task that compiles client SDO classes[1] from XSD or WSDL files.	`com.bea.sdo.impl.SDOGenTask`
sdoclientgen	Ant task that generates Web service-specific client proxy (stub classes) from a DSP-enabled Web service's WSDL.	`com.bea.sdo.impl.WSClientGenTask`

Utility	Description	Classname
SDOGen	Java class that generates typed client SDO classes.	`com.bea.sdo.impl.SDOGen`
WSClientGen	Java class that generates typed Web service client proxy (stub classes).	`com.bea.sdo.impl.WSClientGen`

1. <*appname*>-ld-client.jar

Generating SDO Client Classes

Developing a Web services client application that can access and update SDOs hosted on a Web service requires one of the following:

- Access to the Web service (through its URL, over the network; through UDDI, if the Web service is registered in a public or private UDDI registry; or by access to the physical WSDL file, located locally on the development machine.)
- Access to the XML schema definition (XSD) files that comprise the data type definitions for the data service.

DSP provides an Ant task and a Java application, each of which can use either the WSDL or XSD to generate the client side artifacts.

Setting the Environment for the Utilities

1. At a command prompt, navigate to the directory where the build script and Ant configuration file are located:

   ```
   <bea_home>\weblogic81\samples\domains\ldplatform
   ```

2. Execute the shell or command file script to set the environment for your machine (Windows or Unix):

   ```
   setDomainEnv.cmd
   ```

   ```
   setDomainEnv.sh
   ```

Generating SDO Classes Using Ant

The SDOGen Ant task creates an SDO client JAR file that contains the typed classes for working with SDOs. It can use either the XSDs from the data service or the WSDL (assuming the DSP-enabled

application has been exposed as a Web service) to generate the SDO classes and compile them into the client JAR file.

The SDOGen Ant task lets you build the necessary SDO client JAR which you can then use in your client application code. You can also run the SDOGen task so that it generates the Java and XML (XMLBeans) source code that comprises the SDO type system specified by the schema files.

Environmental Settings

To generate the classes, make sure your classpath includes:

- `wlsdo.jar`
- `xbean.jar`

Syntax

To create a JAR comprising the client classes, execute sdogen at the command prompt as follows:

1. Add the sdogen taskdef to the build script. For example:

```
<taskdef name="sdogen" classname="com.bea.sdo.impl.SDOGenTask"
classpath="path/to/wlsdo.jar:path/to/xbean.jar"/>
```

   This task implicitly defines an Ant FileSet, and supports all FileSet attributes (for example, dir becomes basedir) as well as the nested attributes and elements.

Table 4-9 summarizes the attributes used by the sdogen Ant task.

Table 4-9 Attributes Available for DSP's SDO Generation (sdogen) Ant Task

Attribute	Description	Required?	Default Value
`schema`	A file that points to either an individual schema file or a directory of files. Not a path reference. If multiple schema files need to be built together, use a nested fileset instead of setting schema.	Yes	None
`destfile`	Creates a non-default name for the JAR file. For instance, myXMLBean.jar will output the results of this task into a JAR named myXMLBean.	No	xmltypes.jar
`classgendir`	Directory in which to generate .class files.	No	Current directory

Attribute	Description	Required?	Default Value
`classpath`	Specify the classpath if Java files are in the schema fileset, or if the fileset imports include compiled XMLBeans JAR files. Also supports a nested classpath.	No	
`classpathref`	Adds a classpath, given as reference to a path defined elsewhere.	No	
`debug`	Indicates whether source should be compiled with debug information. If set to false (off), -g:none will be passed on the command line for compilers that support it (for other compilers, no command line argument will be used). If set to true, the value of the debuglevel attribute determines the command line argument.	No	False (off)
`fork`	Flag that indicates whether the JDK compiler (javac) should be executed externally.	No	Yes
`memoryInitialSize`	The initial size of the memory for the underlying VM, if javac is run externally; ignored otherwise. Defaults to the standard VM memory setting.	No	Configured VM memory setting for the machine. For example: 83886080, 81920k, or 80m.
`memoryMaximumSize`	The maximum size of the memory for the underlying VM, if javac is run externally; ignored otherwise. Defaults to the standard VM memory setting.	No	Configured VM memory setting for the machine. For example: 83886080, 81920k, or 80m.
`verbose`	Controls the amount of build message output.	No	True

To build all XML schema definition (XSD) files in the schemas directory and create a JAR named `Schemas.jar`, your Ant script would include the following:

```
<sdogen schema="MyTestWS.WSDL" destfile="Schemas.jar"
classpath="path/to/wlsdo.jar:path/to/xbean.jar"/>
```

Generating SDO Classes Using Java

Rather than using the SDOGen Ant task, you can use the SDOGen Java class at the command-line to generate SDO client classes.

from XML schema definition (XSD) files or WSDL files based on data services.

SDOGen is a Java class that extends the XMLBean schema compiler class.

Optionally, by using the -srconly parameter, the utility can provide you with generated Java source files that define the classes, prior to compiling. Using the -srconly parameter, for example, you can generate the sources and then from the sources, generate Javadoc. In this way you can examine the class hierarchy and methods of the XML data type handlers.

See Table 4-12 for other command-line options for the SDOGen utility.

Table 4-10 Command-line Options for the Java SDO Class Generation Utility

Option	Description	Default Value
`-cp [a;b;c]`	Classpath	
`-d [dir]`	Target directory for binary `.class` and `.xsb` files.	
`-src [dir]`	Target directory for generated Java source files.	
`-srconly`	Flag to prevent compiling Java source files and archiving into JAR file.	
`-out [result.jar]`	Name of the output JAR file.	`xmltype.jar`
`-dl`	Enables network downloads for imports and includes.	Off (not enabled).
`-noupa`	Do not enforce the unique particle attribution rule.	
`-nopvr`	Do not enforce the particle valid (restriction) rule	
`-compiler`	Path to external Java compiler.	

Option	Description	Default Value
`-jar`	Path to JAR (Java Archive) utility	
`-ms`	Initial memory for external Java compiler	8 Megabyte
`-mx`	Maximum memory for external Java compiler	256 Megabyte
`-debug`	Compile with debug symbols.	
`-quiet`	Print minimal informational messages to Java console.	
`-verbose`	Print maximum amount of informational messages to Java console.	
`-license`	Prints license information.	
`-allowmdef "[ns] [ns] [ns]"`	Ignores multiple defs in given namespaces.	

Environmental Settings

To execute the utility, make sure your classpath includes:

- `wlsdo.jar`
- `xbean.jar`

Syntax

To create a JAR comprising the client classes, execute SDOGen at the command prompt as follows:

```
java com.bea.sdo.impl.SDOGen [options] xmlschema
```

XMLSchema can be:

- the URL of a WSDL
- an XSD or WSDL file
- a directory containing numerous XSD or WSDL files

Usage Examples

The following are examples of using SDOGen with various options (see Table 4-10) to obtain different results:

- To create a file named xmltype.jar (the default) based on the WSDL associated with Web service named MyApp running locally:

```
java com.bea.sdo.impl.SDOGen
http://localhost:7001/WebApp/DSCtrls/MyApp.jws?WSDL
```

- To create a file named xmltype.jar (the default) based on the WSDL associated with a publicly available Web service. (The -dl option permits downloading.):

```
java com.bea.sdo.impl.SDOGen -dl
http://198.68.125.17:7001/WebApp/DSCtrls/MyApp.jws?WSDL
```

- To create a file named xmltype.jar using XML schema definition (XSD files) located in the `\myApps\xsd_dir` directory on the local machine:

```
java com.bea.sdo.impl.SDOGen C:\myApps\xsd_dir
```

- To create the MySDOClasses.jar file in the `c:\test\xsd_dir` directory:

```
java com.bea.sdo.impl.SDOGen -out MySDOClasses.jar C:\test\xsd_dir
```

Generating SDO-Enabled Web Services Clients

SDO Web service client generation can be done using the Ant task (SDO Web Service Client Gen) or the Java class (WSClientGen).

Generating SDO Web Services Clients Using Ant

The SDO Web Service Client Gen utility is an Ant task generates an SDO-enabled Web services client JAR file that client applications can use to consume JWS generated from a Data Service control. The generated client JAR file includes:

- Client interface and stub files (conforming to the JAX-RPC specification) used to invoke a Web service in static mode.
- Optional serialization class for converting SDO classes between its XML and Java representation.
- Optional client-side copy of the Web service WSDL file.

Although you could use the SDO Ant task to generate a client JAR file from the WSDL file of any existing Web service (not necessarily running on WebLogic Server), the SDO Client Gen utility typically is used to generate the JAR file from an existing WSDL file of an SDO-enabled JWS.

The WebLogic Server distribution includes a client runtime JAR file (webserviceclient.jar) that contains the client side classes needed to support the WebLogic Web services runtime component.

Environmental Settings

Environmental settings must include the following JAR files from `weblogic/server/lib`:

- `xbean.jar`
- `wlxbean.jar`
- `xqrl.jar`
- `webservices.jar`

You also must include `wlsdo.jar` from the `liquiddata/lib` folder.

Syntax

Define your Ant task for SDOClientGen as follows:

```
<taskdef name="sdoclientgen" classname="com.bea.sdo.impl.WSClientGenTask"
classpath="path/to/SDOclasses:path/to/wlsdo.jar:path/to/xbean.jar:path/to/
wlxbean.jar:path/to/xqrl.jar:path/to/webservices.jar"/>
```

Usage Examples

```
<sdoclientgen wsdl="http://example.com/myapp/myservice.wsdl"
packageName="sdoclient" clientJar="myapps/mySDO_WSclient.jar"
classpathref="all the JAR files listed in the task"/>
```

Table 4-11 Attributes Available for DSP's Web-Services Client Proxy Code Generation Ant Task

Attribute	Description	Required?
`packageName`	Package name for the generated JAX-RPC client interfaces and stub files.	Yes
`wsdl`	Full path name or URL of the WSDL that describes a Web service (either WebLogic or non-WebLogic) for which a client JAR file should be generated. The generated stub factory classes in the client JAR file use the value of this attribute in the default constructor.	Yes

Attribute	Description	Required?
`clientJar`	Name of a JAR file or exploded directory into which the clientgen task puts the generated client interface classes, stub classes, optional serialization class, and so on. To create or update a JAR file, speechify the fullname, including the JAR extension (`myclient.jar`); otherwise, the clientgen task interprets the name as a directory. If the specified JAR or directory does not exist, the clientgen task creates a new JAR file or directory.	No
`classpath`	Must include the path to the SDO classes generated from the XSD or WSDL by the SDOGen Ant task.	No
`classpathref`	Adds a classpath, given as reference to a path defined elsewhere.	No

Generating SDO Web Services Clients using Java

The Web Services Client Generation utility is a Java class (WSClientGen) that developers can use to generate Web services client interfaces and stub classes from a WSDL that uses typed SDO classes for argument and return types. Use this utility to create the artifacts necessary for a client application to invoke DSP functions or submit SDOs.

Environmental Settings

Environmental settings must include the following JAR files from `weblogic/server/lib`:

- `xbean.jar`
- `wlxbean.jar`
- `xqrl.jar`
- `webservices.jar`

You also must include `wlsdo.jar` from the `liquiddata/lib` folder.

Syntax

Define your Ant task for SDOClientGen as follows:

```
<taskdef name="sdoclientgen" classname="com.bea.sdo.impl.WSClientGenTask"
classpath="path/to/SDOclasses:path/to/wlsdo.jar:path/to/xbean.jar:path/to/
wlxbean.jar:path/to/xqrl.jar:path/to/webservices.jar"/>
```

The following should also be kept in mind:

- Make sure the Web service is running if you want to obtain the WSDL by using the Web service's URL.
- Use the SDOGen utility to first generate the JAR file (of typed SDO classes) from the DSP-enabled Web service's WSDL.
- Execute the Java utility as follows:

  ```
  java com.bea.sdo.impl.WSClientGen [options] wsdl
  ```

  The WSDL can be the URL of the WSDL (available over the network), or the actual, physical WSDL file located on your machine. Command-line options that you can pass to the utility are shown in Table 4-12.

Usage Examples

Here are some examples of using the utility:

- To generate an SDO client JAR file from a publicly available WSDL, pass the URL to the Web service as an argument on the command line:

  ```
  java com.bea.sdo.impl.WSClientGen
  http://localhost:7001/WebApp/DSCtrls/MyApp.jws?WSDL
  ```

- To create a JAR named MyClient.jar (rather than the default), pass the filename with the -clientJar parameter at the command line:

  ```
  java com.bea.sdo.impl.WSClientGen -clientJar MyClient.jar
  http://localhost:7001/WebApp/DSCtrls/MyApp.jws?WSDL
  ```

Table 4-12 WSClientGen Utility Options

Option	Description	Default Value
`-version`	Print version information to the Java console.	
`-verbose`	Print the maximum amount of informational messages to the Java console.	
`-clientJar`	Specify the name for the generated JAR file.	`SDOClient.jar`
`-packageName`	Package name of the generated JAX-RPC client interfaces and stub.	sdoclient

Option	Description	Default Value
`-overwrite`	Boolean that specifies if existing files should be overwritten by newly generated code.	True
`-keepGenerated`	Boolean that specifies if generated Java source code should be deleted (False) or kept (True).	False

Using the SDO Web Service Client Gen Utility

The SDO Web Service Client Gen utility is an Ant build script that you can invoked from a command line to build SDO objects for the client. The script (including its pathname) is:

```
<bea_home>\weblogic81\liquiddata\bin\sdo_wsclientgen.xml
```

The WSDL file you created in the procedure described in "Modifying Submit Operations and Generating a WSDL File" on page 4-9 is passed to the utility as a parameter, and the SDO objects generated by `sdo_wsclientgen.xml` are based on that file.

The configuration parameters for the Ant build script `sdo_wsclientgen.xml` are:

- *clientJar* . The name of the JAR file that will be created.
- *wsdl*. The full path to the WSDL file needed to create the SDO objects.
- *classgendir*. The path to location of the generated JAR file.

Before using the Ant script to build SDO classes, make sure you set your environment by calling the setWLSEnv.cmd in the command prompt window. This command file is located in the directory $bea_home\weblogic81\server\bin.

You must include the following packages in your client's CLASSPATH to work with SDO objects:

- `wlsdo.jar`
- `webserviceclient.jar`
- `xbean.jar`
- `wlxbean.jar`
- `xqrl.jar`
- Generated SDO WS client JAR file. The generated SDO WS client JAR file is the file you produced in step 3 of the procedure described in "Modifying Submit Operations and Generating a WSDL File."

The specific steps you need to perform with the Ant utility in order to build SDO classes are:

1. Set the domain environment. For example:

   ```
   \bea\weblogic81\samples\domains\ldplatform\setDomainEnv.cmd
   ```

2. Set the WebLogic environment. For example:

   ```
   \bea\weblogic81\server\bin\setWLSEnv.cmd
   ```

3. Add the sdotemp directory to the classpath:

   ```
   set CLASSPATH=%CLASSPATH%;sdotemp
   ```

4. Call Ant from the command line:

   ```
   ant -buildfile ./sdo_wsclientgen.xml
   ```

After using the Ant utility (that is by issuing the command, `ant sdo_wsclientget.xml`), a JAR file is created; among other generated artifacts, the JAR file contains the typed SDO classes. You can distribute the JAR files to all clients that will consume operations from this Web service.

After running the Ant utility, you can call the modified submit operation that you created in step 2 of the procedure described in "Modifying Submit Operations and Generating a WSDL File" on page 4-9. For example, your client code would be as follows, based on the submitCustomerProfile( ) method shown in step 2:

Listing 4-2 Example of Invoking the Submit Method

```
CustomerDataTestSoap wssoap = new
CustomerDataTest_Impl().getCustomerDataTestSoap();
CustomerProfileDocument doc = wssoap.getCustomerProfile(customer_id);
doc.getCustomerProfile().getCustomerArray(0).setLastName("Test");
DataGraphImpl dg = (DataGraphImpl) doc.getDataGraph();
wssoap.submitCustomerProfile(dg.getSerializedDocument());
```

Post-Generation Development Tasks

BEA AquaLogic Data Services Platform includes a sample Web service project — SampleWS — which is used in this section to demonstrate how to update a data service that has been wrapped as a Web service. The assumption is that the client-side programming will use the Web service's WSDL. Using the SampleWS as a starting point, you can

The procedure below adds:

- An SDO client JAR file to the library of the project
- An additional Java file that demonstrates the implementation of a modified submit method as described in "How to Update a Data Service Exposed as a Web Service."

To work with SampleWS and run the example code, do the following:

1. Import the project files for SampleWS into WebLogic Workshop by right-clicking on your Evaluation application and importing SampleWS as a Web service project. SampleWS is located in the directory:

   `$bea_home\weblogic81\samples\LiquidData\EvalGuide`

2. Build your SampleWS project.
3. Right click on your application library folder and select Add Library....
4. Navigate to the following directory:

   `$bea_home\weblogic81\samples\LiquidData\EvalGuidedirectory`

5. Select the `SDOClient.jar` file to be added. Click the Open button.
6. Verify that `SDOClient.jar` is imported into your library folder.
7. Right-click on your application and import ConsumeWS as a Java project. ConsumeWS is located in the directory:

   `$bea_home\weblogic81\samples\LiquidData\EvalGuide`

8. Open the `ConsumeWS.java` file located in the ConsumeWS project and execute it. After executing `ConsumeWS. java`, you should see results similar to the following.

Sample build.xml File

Listing 4-3 shows a complete example of using the Ant tasks as part of a build process.

Listing 4-3 Sample build.xml File for the SDOGen and WSClientGen Ant Tasks

```
- <project name="samplesdogen" default="build" basedir=".">
    <property name="output.jar" value="MyTestClient.jar" />
    <property name="wsdl.file" value="../SimpleCtrlTest.wsdl" />
    <property name="local.build.dir" value="build" />
    <property name="external.resource.dir" value="../DSP/external" />
    <mkdir dir="${local.build.dir}" />
```

```
- <path id="compile.classpath">
    <pathelement path="${java.class.path}" />
    <pathelement path="${local.build.dir}" />
    <pathelement location="${external.resource.dir}/weblogic.jar" />
    <pathelement location="${external.resource.dir}/xbean.jar" />
    <pathelement location="${external.resource.dir}/wlxbean.jar" />
    <pathelement location="${external.resource.dir}/xqrl.jar" />
    <pathelement location="${external.resource.dir}/webservices.jar" />
    <pathelement location="${external.resource.dir}
                           /../src/ld-core/sdoUpdate/dist/wlsdo.jar" />
    </path>
<taskdef name="sdogen" classname="com.bea.sdo.impl.SDOGenTask"
classpathref="compile.classpath" />
<taskdef name="sdoclientgen" classname="com.bea.sdo.impl.WSClientGenTask"
classpathref="compile.classpath" />
- <target name="sdo" depends="clean">
  <sdogen classgendir="${local.build.dir}" schema="${wsdl.file}"
classpath="${external.resource.dir}/../src/ld-core/sdoUpdate/dist/wlsdo.ja
r:${external.resource.dir}/xbean.jar" memoryInitialSize="8m"
memoryMaximumSize="256m" fork="true" failonerror="true" />
  </target>
- <target name="build" depends="sdo">
  <sdoclientgen wsdl="${wsdl.file}" packageName="sdoclient"
clientJar="${local.build.dir}/${output.jar}"
classpathref="compile.classpath" />
  - <jar jarfile="${local.build.dir}/${output.jar}" update="yes">
       - <fileset dir="${local.build.dir}">
          <exclude name="${output.jar}" />
          </fileset>
       </jar>
     </target>
   - <target name="clean">
        <delete dir="${local.build.dir}" />
        <mkdir dir="${local.build.dir}" />
     </target>
</project>
```

CHAPTER 5

Accessing Data Services from WebLogic Workshop Applications

BEA AquaLogic Data Services Platform

This chapter describes how you can use Data Service controls in WebLogic Workshop to develop client applications for Data Services Platform. The following topics are included:

WebLogic Workshop and Data Services Platform

Data Service controls provide WebLogic Workshop applications an easy way to use 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, "DSP's Data Programming Model and Update Framework."

In addition to the functionality discussed in this chapter, Data Service controls also provide many of the same features available through the Mediator API, including:

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

For more information on these features, see Chapter 10, "Advanced Topics."

Data Service Controls

A Data Service control is a wizard-generated Java file that can be used to add data service functions and procedures to WebLogic Workshop applications. Functions and procedures can be added to Data Service controls from data services deployed on any accessible WebLogic Server, both local or remote. The Data Service control wizard retrieves all available data service functions and procedures 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 other Java controls available in WebLogic Workshop applications, 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 a Data Service 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

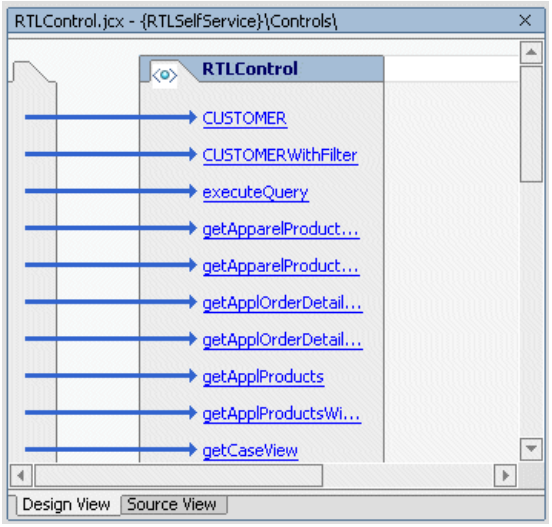
When you create a Data Service control, WebLogic Workshop generates a Java Control Extension (`.jcx`) file that contains methods based on the data service's functions, and a commented method that can be uncommented and used to pass any XQuery statements (called *ad hoc queries)* to the server. This section describes the Data Service control in detail and includes the following sections:

- Design View
- Source View
- Using Data Service Controls for Ad Hoc Queries

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 or procedure associated with the 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 Data Service control (a Java Control Extension, or JCX file. It includes annotations defining the data service function names 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 "Using Data Service Controls for Ad Hoc Queries" on page 5-7.

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/ApplOrderDetailView.ds"
functionName="submitApplOrderDetailView"
     */
    java.util.Properties[]
submitApplOrderDetailView(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/ApplOrderDetailView"
```

```
functionName="getApplOrderDetailView"
     */
    retailer.ORDERDETAILDocument getApplOrderDetailView(java.lang.String p0);
     .
     .
     .
    /**
     *
     * @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 differnt 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);

 }
```

Using Data Service Controls for Ad Hoc Queries

Client applications 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);
   ```

   (Replace the function name with one that is meaningful for your application. Be default, the ad hoc query returns an XMLObject, 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 ldcontrol.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 for 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 the application. You can create a Data Service control in most types of WebLogic Workshop projects; most commonly, you will create them in:

- 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 hosts 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 WebLogic Workshop).

Step 3: Create a Folder in a Project

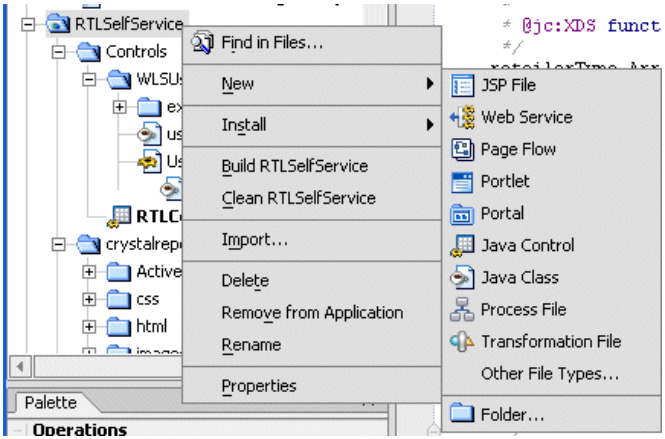
Create a folder in the project to hold the Data Service control by selecting a folder and right-clicking on that folder. You can also create other controls (database controls, for example) in the same folder

as needed. WebLogic 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

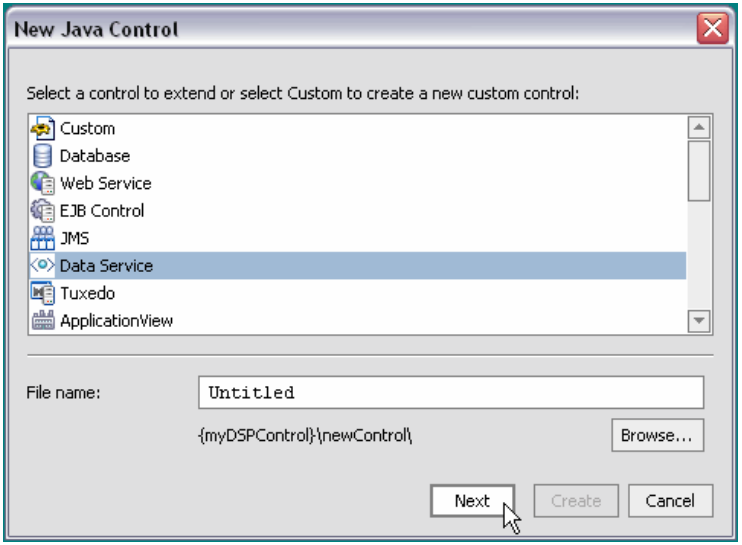
To create a Data Service control, start the Java Control Wizard by right-clicking on the new folder in your project and choosing New →Java Control as shown in Figure 5-2. (You can also create a control using the File →New →Java Control menu item.)

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 for WebLogic Server

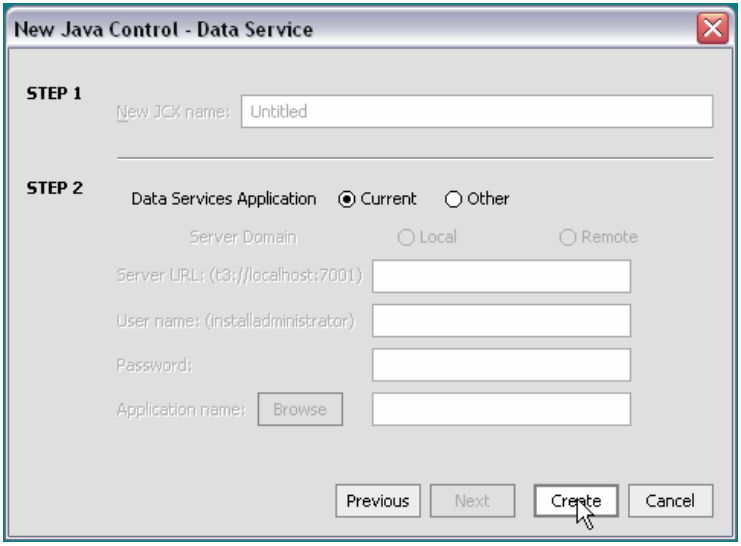
The New Java Control - DSP dialog (Figure 5-4) allows you to enter connection information for the WebLogic Server that hosts your Data Services Platform application or project. If the server is local, a Data Service control uses the connection information stored in the application properties. (To view these settings, access the Tools →Application Properties menu item in WebLogic 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 used for creating the JCX file only, not for testing queries through the control. For more details, see "Security Considerations When Using Data Service Controls" on page 5-27.

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

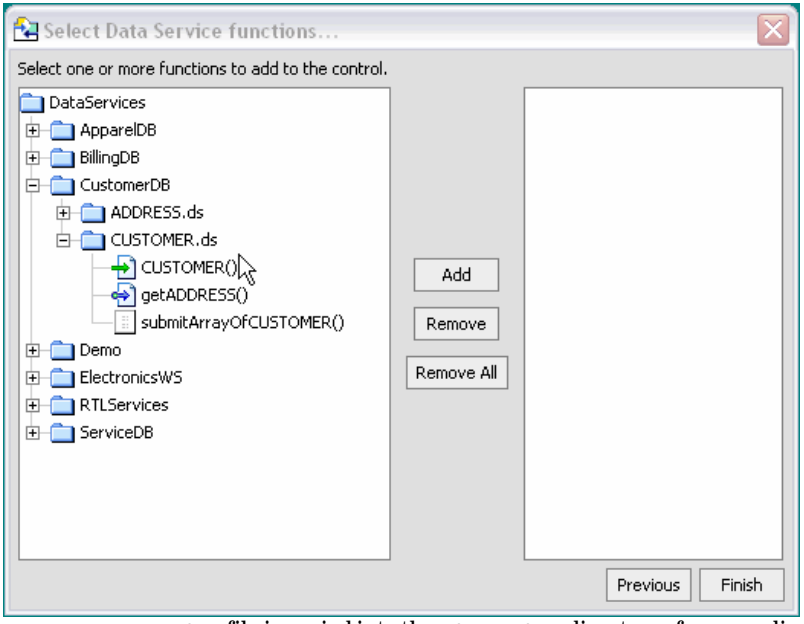
Figure 5-4 Data Service Control Wizard: Connection Information



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

In the Select Data Service functions... page, 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 "Filtering, Sorting, and Fine-tuning Query Results" in Chapter 10, "Advanced Topics."

After you have added all the queries you need in the wizard, click Finish. WebLogic 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 WebLogic Workshop Application.

Note: If you get a timeout error when attempting to create a Data Service control, you 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:

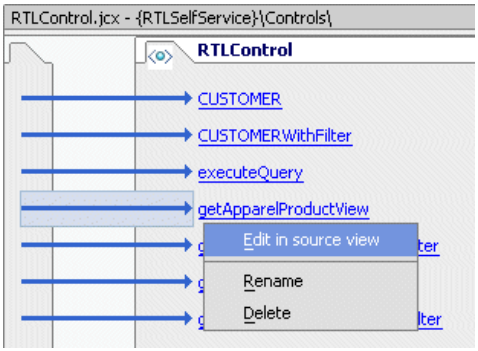
- Changing a Method Used by a Control
- Adding a New Method to a Control
- Updating an Existing Control When Schemas Change

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.

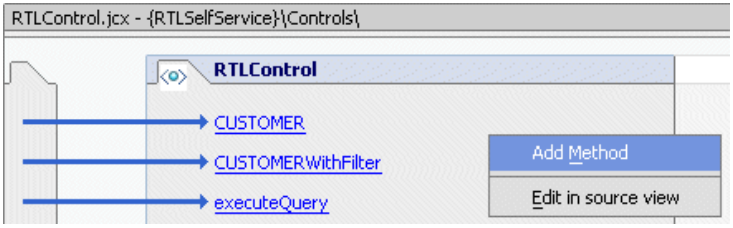
When you save your changes, the SDO classes based on the control are automatically recompiled.

Adding a New Method to a Control

To add a new method to an existing Data Service control, perform the following steps:

1. In WebLogic 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 When Schemas Change

If any of the schemas corresponding to any methods in a Data Service control change, you must clean and re-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.

Note: If you developed a client application using a static client API and you modify any schemas, you must also recompile and redeploy the application to your user community, using the re-generated classes.

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

Note: For details on working with static and dynamic SDO see "Static and Dynamic SDO APIs" on page 3-14.

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

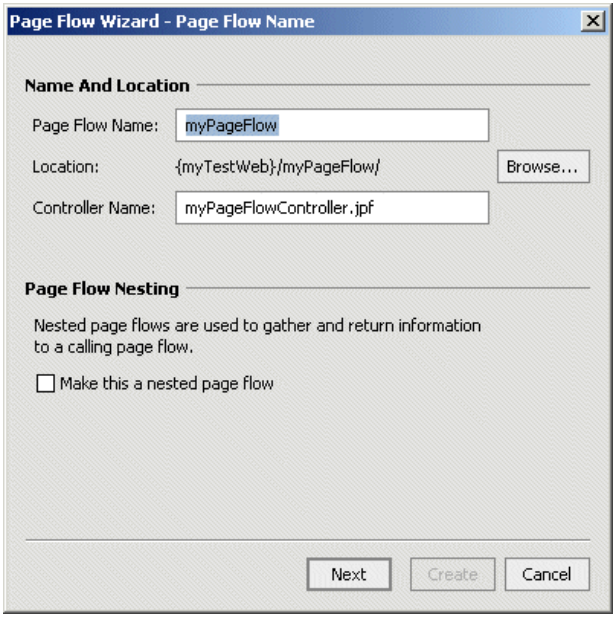
When you use WebLogic Workshop to generate a page flow, WebLogic Workshop creates the page flow, a start page (`index.jsp`), and a JSP file and action for each method you specify in the Page Flow wizard.

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.

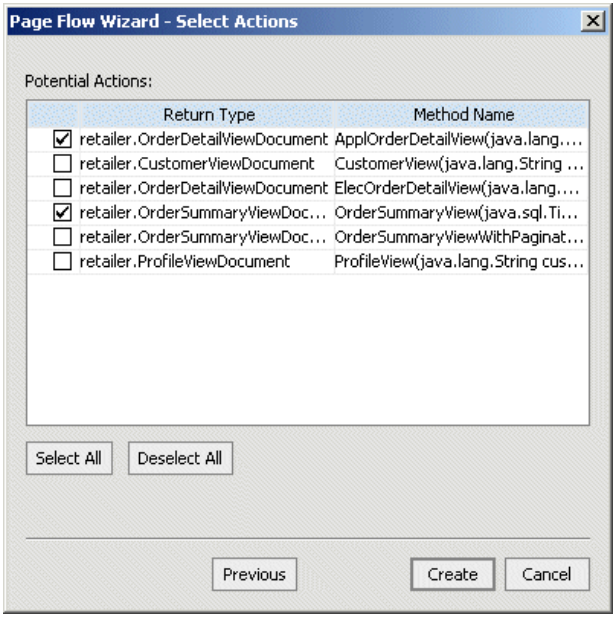
1. Select a Data Services Platform control JCX file from the application file browser, right-click, and select Generate Page Flow.
2. In the Page Flow Wizard (see Figure 5-8), enter a name for your Page Flow and click Next.

Figure 5-8 Enter a Name for the Page Flow



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.

   WebLogic Workshop generates the Java Page Flow (JPF 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 5-18.

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 5-22.

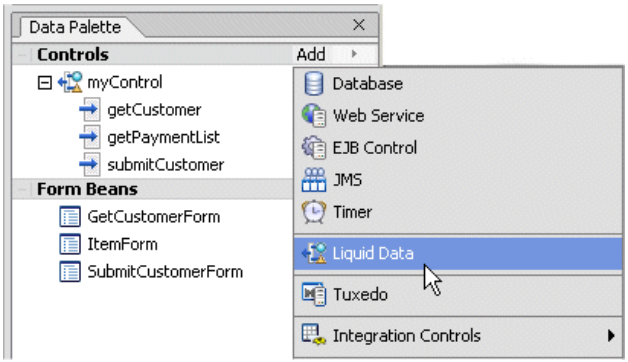
7. Build and test the application in WebLogic Workshop.

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" in Chapter 4, "Web Services and DSP-Enabled Applications." 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 JPF 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, "DSP's Data Programming Model and Update Framework."

Defining a variable in the page flow JPF 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.

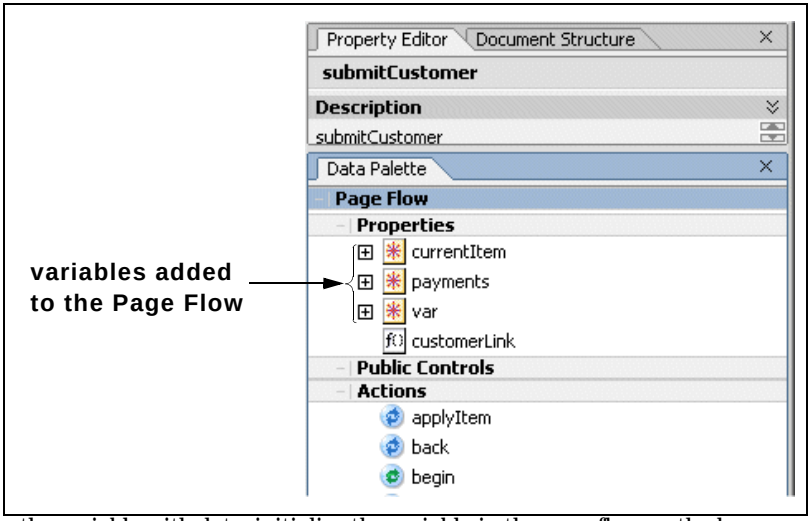
Define each variable with a type corresponding to an SDO object. Define the variables in the source view of the page flow controller class. The variables should be declared public. In the following example, the bold-typed variable declarations show an example of user variable declarations:

```
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 5-20.

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 JPF file in WebLogic 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 JPF file in WebLogic 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 5-15).

```
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.p1);
        ...
        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.

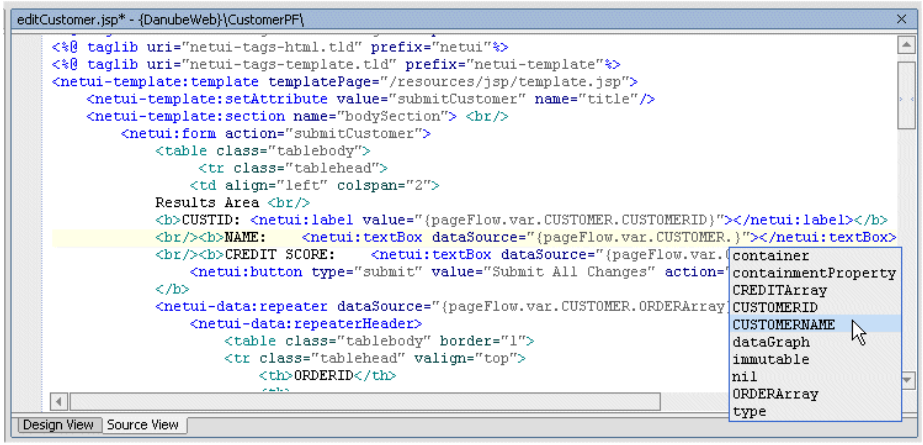
The elements appear in dot-delimited chain format, such as:

```
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, WebLogic 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, "DSP's Data Programming Model and Update Framework."

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).

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

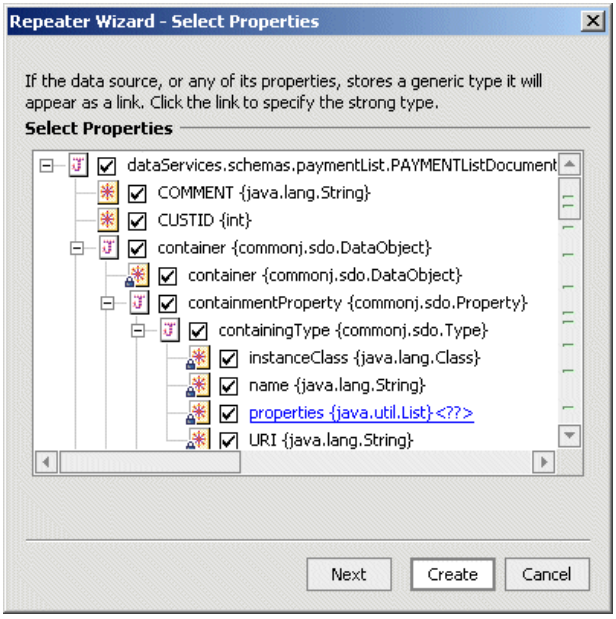
To add a NetUI repeater tag (used to display the data from a Data Services Platform query) to a JSP file, perform the following steps:

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 5-20.

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. WebLogic Workshop displays the repeater wizard as shown in Figure 5-13.

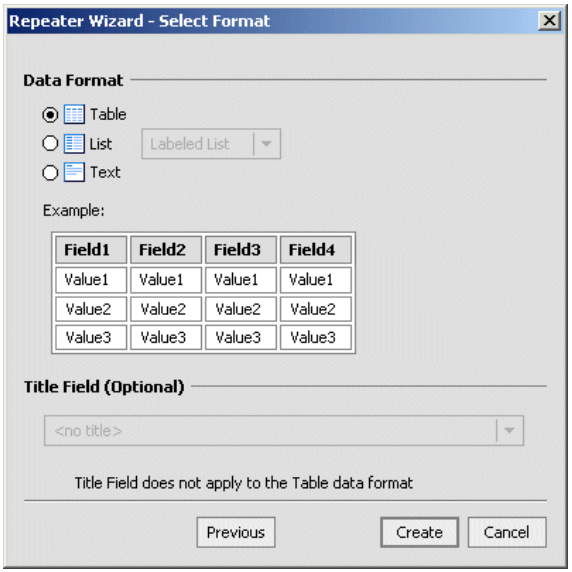
Figure 5-13 Repeater Wizard



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



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 5-22.
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 5-22.
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 5-20.

```
<%
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.

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 Data Service 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 Data Service 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 by-product, 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 WebLogic 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 WebLogic 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.

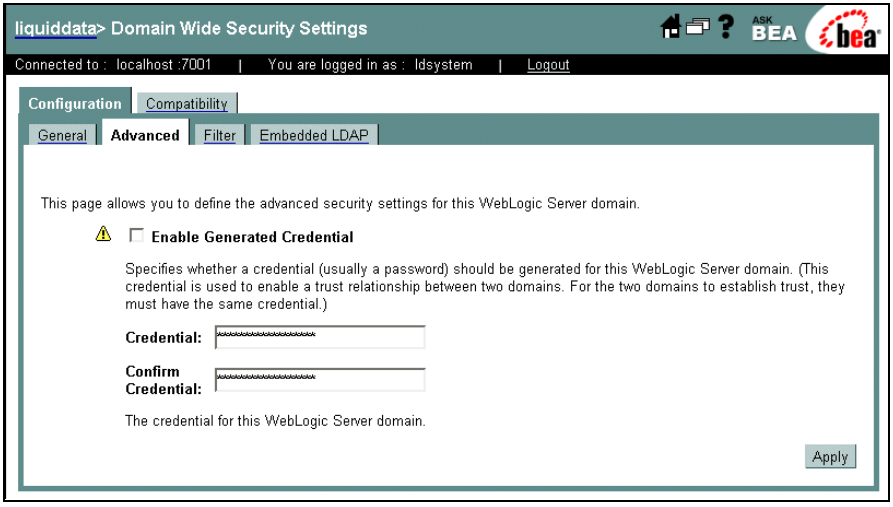
Note: After configuring domains as trusted, you must restart the domains before the trusted configuration takes effect.

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 5-15.)

Figure 5-15 Setting up Trusted Domains



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 DSP *Administration Guide*.

CHAPTER 6

Supporting ADO.NET Clients

This chapter describes how to enable interoperability between BEA AquaLogic Data Services Platform (DSP) data services and ADO.NET client applications. With support for ADO.NET client applications, Microsoft Visual Basic and C# developers who are familiar with Microsoft's disconnected data model can leverage DSP data services as if they were ADO.NET Web services.

From the Microsoft ADO.NET developers' perspective, support is transparent: you need do nothing extraordinary to invoke functions on a DSP data service—all the work is done on the server-side. ADO.NET-client-application developers need only incorporate the DSP-generated WSDL into their programming environments, as you would when creating any Web service client application.

General information about how DSP achieves ADO.NET integration is provided in this chapter, as are the server-side operations required to enable it. The chapter includes the following sections:

- Overview of ADO.NET Integration in Data Services Platform
- Enabling DSP Support for ADO.NET Clients
- Adapting DSP XML Types (Schemas) for ADO.NET Clients
- Generated Artifacts Reference

Note: The details of ADO.NET development are described on Microsoft's MSDN Web site (http://msdn.microsoft.com). See that site for information about developing ADO.NET-enabled applications.

Overview of ADO.NET Integration in Data Services Platform

Functionally similar to service data objects (SDO), ADO.NET is data object technology for Microsoft ADO.NET client applications. ADO.NET provides a robust, hierarchical, data access component that enables client applications to work with data while disconnected from the data source. Developers creating data-centric client applications use C#, Visual Basic.NET, or other Microsoft .NET programming languages to instantiate local objects based on schema definitions.

These local objects, called DataSets, are used by the client application to add, change, or delete data before submitting to the server. Thus, ADO.NET client applications sort, search, filter, store pending changes, and navigate through hierarchical data using DataSets, in much the same way that SDOs are used by DSP client applications.

See "Role of the Mediator and SDOs" on page 2-14 for more information about working with SDOs in a Java client application. Developing client applications to use ADO.NET DataSets is roughly analogous to the process of working with SDOs.

Although functionally similar on the surface, as you might expect with two dissimilar platforms (Java and .NET), the ADO.NET and SDO data models are not inherently interoperable. To meet this need, Data Services Platform provides ADO.NET-compliant DataSets so that ADO.NET client developers can leverage data services provided by Data Services Platform, just as they would any ADO.NET-specific data sources.

Enabling a Data Services Platform data service to support ADO.NET involves three key steps:

- Creating an ADO.NET-Enabled Data Service Control (Note that ADO.NET-Enabled Data Service controls are intended exclusively to provide support to ADO.NET clients via a Web service interface, as described in this chapter: such controls cannot be used in Page Flows, Portals, or other development scenarios.)
- Generating a Web Service for ADO.NET Clients
- Generating an ADO.NET-Enabled WSDL

These steps are described in "Enabling DSP Support for ADO.NET Clients" on page 6-7.

Understanding ADO.NET

ADO.NET is a set of libraries included in the Microsoft .NET Framework that help developers communicate from ADO.NET client applications to various data stores. The Microsoft ADO.NET libraries include classes for connecting to a data source, submitting queries, and processing results. The DataSet also includes several features that bridge the gap between traditional data access and

XML development. Developers can work with XML data through traditional data access interfaces, and vice-versa.

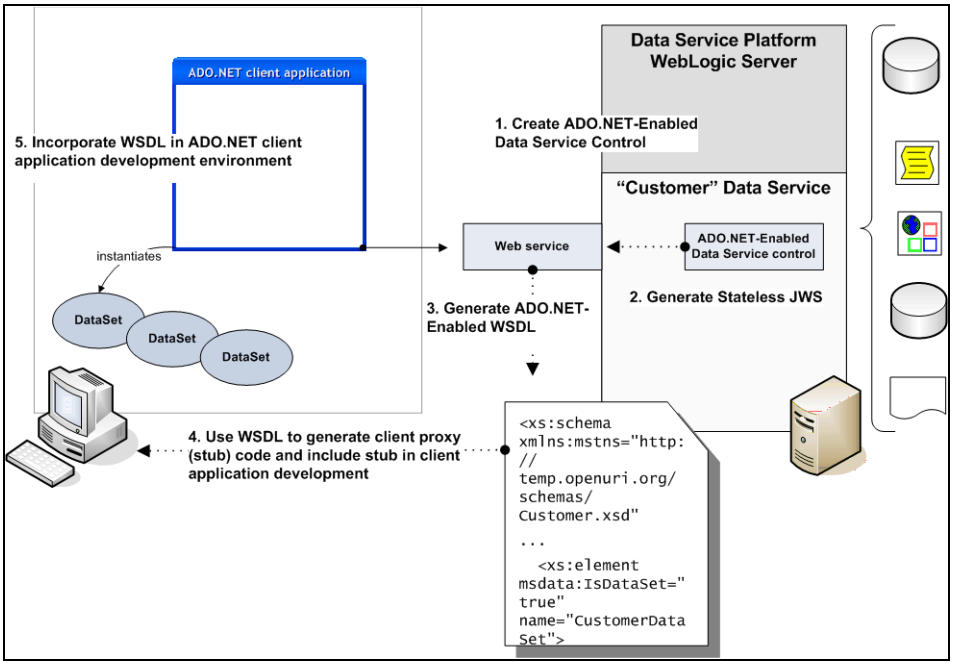
Note: See Microsoft's MSDN site (`http://msdn.microsoft.com/`) for more information about ADO.NET and client application development.

Although ADO.NET supports both connected (direct) and disconnected models, in Data Services Platform only the disconnected model is supported.

ADO.NET Client Application Development Tools

ADO.NET client applications are typically created using Microsoft Windows Forms, Web Forms, C#, or Visual Basic. Microsoft Windows Forms is a collection of classes used by client application developers to create graphical user interfaces for the Windows .NET managed environment. Web Forms provides similar client application infrastructure for creating Web based client applications. Any of these client tools can be used by developers to create applications that leverage ADO.NET for data sources.

Figure 6-1 ADO.NET Clients Supported via Web Services



Support for ADO.NET clients is provided via Web services, so before you can use your Microsoft tools of choice, you must perform the two basic tasks required for web-service client development, just as you normally would for any Microsoft Web services client application (see Figure 6-1):

- Obtain the WSDL for the DSP Web service application.
- Generate the client side artifacts from the WSDL as required for the client application development tool you are using.

Once the client-side artifacts have been incorporated into your development environment, you can invoke functions on the data service and manipulate the DataSet objects in your code as you normally would.

Note: The process of generating the WSDL and server-side artifacts is described in "Generating a Web Service for ADO.NET Clients" on page 6-10.

Understanding How DSP Supports ADO.NET Clients

BEA AquaLogic Data Services Platform supports ADO.NET at the data object level. That is, Data Services Platform maps inbound ADO.NET DataSet objects to SDO DataObjects, and maps outbound SDOs to DataSets. The mapping is performed transparently on the server, and is bidirectional.

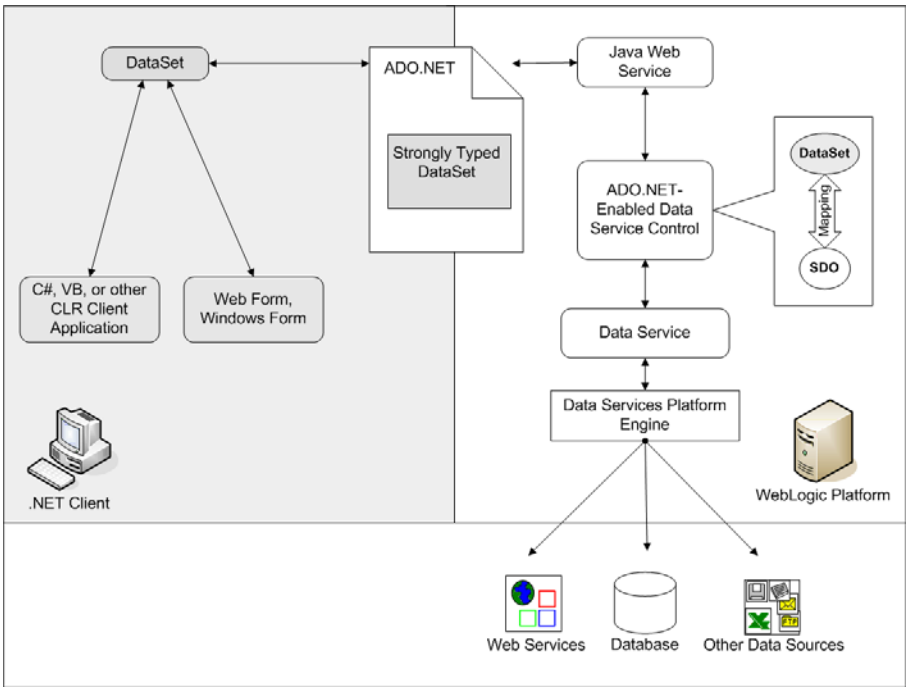
Table 6-2 ADO.NET and SDO Data Objects Compared

ADO.NET	SDO	Description
DataSet	DataObject	Disconnected data models. Queries return results conforming to this data model.
DiffGram	ChangeSummary	Mechanisms for tracking changes made to data objects by a client application.

As shown in Figure 6-3, the ADO.NET typed DataSet is submitted to and returned by DSP. At runtime, when a Microsoft-.NET client application makes a SOAP invocation to the ADO.NET-enabled Web service, the Web service intercepts the object and passes it to the Data Service control.

The ADO.NET-enabled Data Service control is the linchpin of the interoperability between the two platforms. It comprises several wrapper classes—one for each typed DataSet—that are used to provide bidirectional mapping.

Figure 6-3 Data Services Platform and .NET Integration



The required wrapper classes are created automatically, during the process of creating the ADO.NET-enabled Data Service control, as described later in this chapter. The wrapper classes are based on the XML schema file that gets generated during Data Service control creation.

At runtime, the ADO.NET-enabled Data Service control uses the wrapper classes to provide the ADO.NET client with the appropriate objects. The specifics vary, depending on the type of function or procedure:

- **Functions**. The Data Service control wraps a query result using the typed DataSet schema, adds the DataSet schema type to the result, and returns to the client.
- **Procedures**. A DSP procedure can return an SDO; another data type; or nothing (void). The Data Service control uses the wrapper classes as required, but only if required.
- **Submitting changes.** The Data Service control transforms an ADO.NET DataSet DiffGram to an SDO ChangeSummary, and then submits it to SDO Mediator. All submit methods take the corresponding wrapper classes as arguments.

As mentioned previously, mapping, transformation, and packaging processes are transparent to client application developers and data services developers. Only the items listed in Table 6-4 are exposed to data service developers.

Table 6-4 Data Services Platform—Java and ADO.NET-Enabled Artifacts

Name	Example	Description
Data Service	`Customer.ds`	An XQuery file that instantiates read functions, navigation functions, procedures, and update functionality at runtime.
Data Service Schema	`Customer.xsd`	The schema associated with the return type of the original data service.
DataSet Schema	`CustomerDataSet.xsd`	The typed DataSet schema that conforms to Microsoft requirements for ADO.NET data objects.
Data Service Control	`Customer.jcx`	An ADO.NET-enabled data service control.
Web Service Source	`Customer.jws`	A Java Web service that can intercept ADO.NET data objects and pass them to an ADO.NET-enabled Data Service control.
***<DSControlName>*_schema**	`Customer_schema`	An automatically created folder for containing generated typed DataSet XSDs.
Web Service Definition	`CustomerNET.wsdl`	Generated WSDL that conforms to the ADO.NET typed DataSet schema.

Supporting Java Clients

The WSDL generated by the WebLogic Server from an ADO.NET-enabled Data Service control is specific for use by Microsoft ADO.NET clients. Exposing data services as Web services that are usable by Java clients is generally the same, although the actual steps (and the generated artifacts) are specific to Java. The steps are summarized in Table 6-5.

Table 6-5 Summary of Steps for Supporting Regular Clients

Task	For more information...
Generate Data Service Control (regular, not ADO.NET-enabled)	"Creating Data Service Controls" on page 5-8
Generate Web service file (JWS)	"Server-side DSP-Enabled Web Service Development" on page 4-4
Generate WSDL	"Server-side DSP-Enabled Web Service Development" on page 4-4

Enabling DSP Support for ADO.NET Clients

The process of providing ADO.NET clients with access to data services is a server-side operation that takes place in the context of an application and WebLogic Workshop.

The instructions in this section assume that you have created a data service application and that you want to provide access to the functions of the service to ADO.NET client applications. (For information about designing and developing data services, see the *Data Services Developer's Guide*.)

Enabling a DSP application to support ADO.NET clients is generally a three-step process:

- Creating an ADO.NET-Enabled Data Service Control
- Generating a Web Service for ADO.NET Clients
- Generating an ADO.NET-Enabled WSDL

The tasks described in the remaining sections assume that a data services application is open in WebLogic Workshop.

Creating a New Web Service Project

Since the ADO.NET support is accomplished through the use of Data Service controls, and since the Data Service controls require being exposed as Web services in order to make them network accessible, the first step is to create a Web service project and the folder structure necessary to hold generated components.

In the data service application that you want to ADO.NET-enable, create a new Web service project specifically for the ADO.NET-enabling components of the application (see Figure 6-6).

Note: Be sure to give the Web service project a meaningful name; the name will be used during the generation of several artifacts, including the name of the Data Service control.

Figure 6-6 Folder Structure for ADO.NET-Enabled Project Components

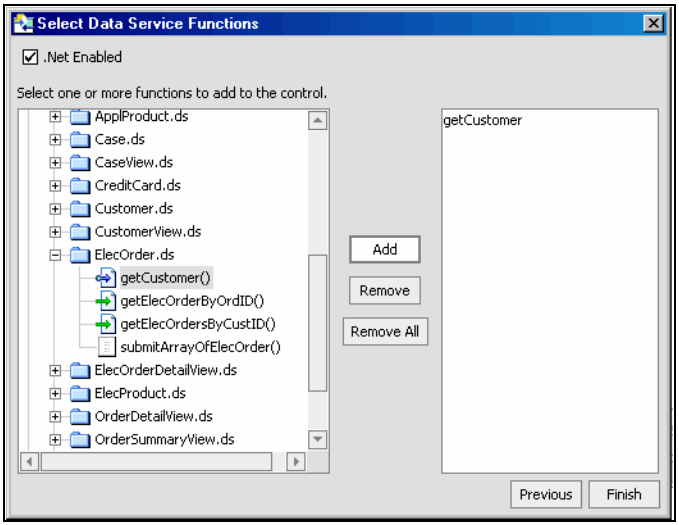
Creating an ADO.NET-Enabled Data Service Control

Data Service controls can be ADO.NET-enabled simply by selecting the appropriate checkbox during the creation process. The ADO.NET-Enabled Data Service controls created (as described in this section) are designed exclusively to support ADO.NET clients through a Web services interface: such controls cannot be used in Page Flows, Portals, or other development scenarios.

Starting from the Web service project folder, here are the general steps:

1. Create a folder in your project for the Data Service control by selecting a folder and right-clicking on that folder. (Java controls must be contained inside a folder within a project—they cannot reside at the top level of the project.)
2. Right-click on the folder in the project to display the popup menu, and then select New →Java Control. The New Java Control dialog displays.
3. Select Data Services Platform from the New Java Control dialog. Enter a filename for the control (JCX) file and click Next. The New Java Control - Data Service dialog displays.
4. Enter the connection information for the WebLogic Server that hosts the Data Services Platform application.
   - For a local server, the Data Service control uses the connection information stored in the application properties.
   - For a remote server, you must select Remote and then provide the server URL, user name, and password.
5. Click Create to continue. The Select Data Service Functions dialog displays. Note the ADO.NET-Enable checkbox in the upper-left-hand corner of the dialog, shown in Figure 6-7.

Figure 6-7 Select Data Service Functions Dialog



6. Click the ADO.NET-enabled box and then select one or more functions or procedures to use in the ADO.NET-enabled data service control.

   Note: Due to a Microsoft limitation, the functions and procedures that you add to your Data Service control must belong to the same namespace.

7. Click Next to continue. A Control generation detailed configuration page displays, showing the functions select on the previous page. On this page, you can select the functions (if any) that should include a filter or an attribute.

   – Add a filter to the JCX method (For more information about filters, see "Filtering, Sorting, and Fine-tuning Query Results" on page 10-5.)

   – Add an attribute to the method.

8. Click Finish to complete the process.

As the ADO.NET-enabled Data Service control file is being generated, a folder is also created inside the controls folder, and a Microsoft-style XML schema definition file (XSD) is generated and placed inside the folder. The generated folder follows this simple naming convention:

```
<Data Service control name>_schema
```

The schema file created in the *<Data Service control name>*_schema folder is a combination of the Data Service control name and "DataSet;" for example, CustomerDataSet.xsd. (See Table 6-4 for other

relevant naming conventions.) The XML schema file contains method calls for all selected functions and procedures.

As the XSD is created, you may see a Message box display briefly in WebLogic Workshop, notifying you that you have added one or more XSD files to a non-Schema project. Such a message can be disregarded; it is raised because the Microsoft ADO.NET style XSD is not the same as other data service XSD files.

Note: For more information about Data Service controls, see "Creating Data Service Controls" on page 5-8.

Java controls are not network-addressable unless wrapped as Web services. Invoking a Java Control of any kind, including a Data Service control from outside the application, requires that it be exposed as a Web service or as another Web-based application, such as a JSP (JavaServer Page).

Generating a Web Service for ADO.NET Clients

After the ADO.NET-enabled Data Service control has been generated, it is used as the basis for generating a Java Web service file (JWS), as follows:

1. Right-click on the Data Service control.
2. Select Generate Test JWS File (Stateless) from the pop-up menu. (ADO.NET client support is limited to stateless Web services.)

Shortly, the JWS is generated; you will see it displayed as a node under the Data Service control. From this JWS you can now generate the companion WSDL (Web Services Description Language) file that will be used by Web service client-application developers.

Note: After the Java Web service (JWS) file has been generated, it can be deployed in the usual manner. See the Web services page on BEA's documentation site for more information:

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

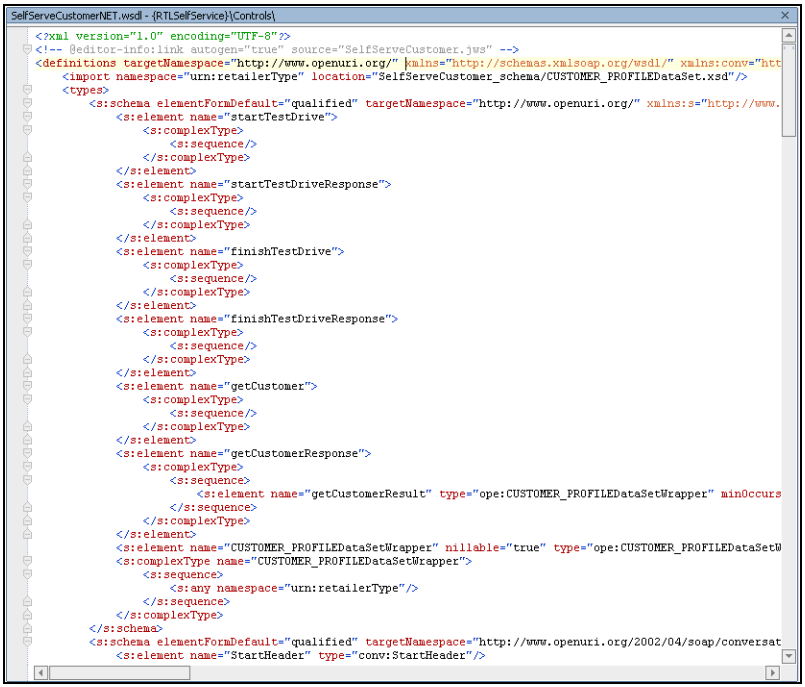
Generating an ADO.NET-Enabled WSDL

To generate the companion WSDL (Web Services Description Language) file from the JWS that can be used by Web service clients to invoke operations on the ADO.NET-enabled Web service:

1. Right-click on the JWS file created in "Generating a Web Service for ADO.NET Clients."
2. Select Generate ADO.NET Enabled WSDL File from the pop-up menu.

In a moment, the WSDL is generated; you will see it displayed as a node under the JWS file.

Figure 6-8 Generated WSDL in WebLogic Workshop



- See "Web Services Description Language (WSDL) File for Microsoft ADO.NET Clients" on page 6-16 for information about the format of the WSDL.

The WSDL should be made available to ADO.NET developers directly (for example, by sending the physical file to them). Developers can also obtain the WSDL from the Web service, from BEA WebLogic Server's home page.

Adapting DSP XML Types (Schemas) for ADO.NET Clients

Fundamentally, Microsoft's ADO.NET DataSet is designed to provide data access to a data source that is — or appears very much like — a database table (columns and rows). Although, later adapted for consumption of Web services, ADO.NET imposes many design restrictions on the Web service data source schemas.

Due to these restrictions, Data Services Platform XML types (also called schemas or XSD files) that work fine with data services may not be acceptable to ADO.NET's DataSet.

This section explains how you can prepare XML types for consumption by ADO.NET clients. It covers both read and update from the ADO.NET client side to the DSP server, specifically explaining how to:

- Read a DSP query result as a ADO.NET DataSet via SDO (since query results are presented as SDO DataObjects within DSP).
- Update DSP data sources using an ADO.NET DataSet's diffgram that is mapped to a SDO ChangeSummary.

Note: See the Data Services Developer's Guide for detailed information related to creating and working with XML types.

Approaches to Adapting XML Types for ADO.NET

There are several approaches to adapting XML types for use with an ADO.NET DataSet:

- **Develop ADO.NET-compatible data services above the physical data service layer.** You can develop data services on top of physical data sources that are specifically intended to be consumed by ADO.NET clients. (Details are described in "XML Type Requirements for Working With ADO.NET DataSets.")

  Note: Any ADO.NET-compatible data service XML types can be consumed by non-ADO.NET clients.

- **Develop ADO.NET-compatible data services above a logical data service layer.** If existing logical data services that are not ADO.NET-compatible must be reused, you can build an additional layer of ADO.NET-compatible data services on top of the logical data services.

  Note: This approach may increase the likelihood of having to work with inverse functions and custom updates. (The usage of inverse functions is described in "Best Practices and Advanced Topics", Data Services Developer's Guide.)

XML Type Requirements for Working With ADO.NET DataSets

The following guidelines are provided to help you develop ADO.NET DataSet-compatible XML types (schemas) by providing pattern requirements for various data service artifacts.

Requirements for Complex Types

Requirements for supporting a complex type in an ADO.NET DataSet include:

- Defining the entire XML type in a single schema definition file. This means not using include, import, or redefine statements.
- Define one global element in the XML type and all other complex types as anonymous complex types within that element. Define one global element in the schema and define all other complex types as anonymous complex types within the element. Do not define any of the following:

  - global attribute
  - global attributeGroup
  - global simple type

- Be sure that the name of an element in the anonymous complex type is unique within the entire schema definition.

  Note: The name of an element of simple type need not be unique, unless the occurrence of the element is unbounded.

Requirements for Recurring References

Since ADO.NET does not support true recurring references among complex types, the requirements noted in Requirements for Complex Types should be followed when simulating schema definitions utilizing such constructs as:

- Nested complex types
- Recurring references among complex types
- Multiple references from different complex type to a single complex type

As an example, if an address complex type has been referenced by both Company and Department, there should be two element definitions, CompanyAddress and DepartmentAddress, each with an anonymous complex type. The following code illustrates this:

```
<xsd:schema targetNamespace="urn:company.xsd"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element name="Company">
  <xsd:complexType>
    <xsd:sequence>
    <xsd:element name="Name" type="xsd:string"/>
    <xsd:element name="CompanyAddress">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="City" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
      <xsd:element name="Department">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="Name" type="xsd:string"/>
            <xsd:element name="DepartmentAddress">
              <xsd:complexType>
```

```
            <xsd:sequence>
              <xsd:element name="City" type="xsd:string"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
 </xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

Requirements for Simple Types

Requirements for supporting simple types in an ADO.NET DataSet include:

- Use xs:dateTime type in the XML type rather than xs:date, or xs:time, or any gXXX type, such as gMonth, etc. (If a physical date source uses gXXX type, you should rely on the use of an inverse function to handle the type for update. For gXXX types, you should rely on the use of a DSP update override function to handle the update.)

  Note: The usage of inverse functions is described in "Best Practices and Advanced Topics", Data Services Developer's Guide.

- Base64Binary type should be used, rather than hexBinary type.
- Avoid using List or Union type.
- Avoid using xs:token type.
- Avoid defining default values in your XML type.
- The length constraining facet for 'String' should not be used.

Requirements for Target Namespace and Namespace Qualification

Requirements for using target namespaces and namespace qualification include:

- Your XML type must have a target namespace defined. Everything in the type should be under a single namespace.
- Set the elementFormDefault and attributeFormDefault to unqualified for the entire XML type. (As these are the default setting of a schema document, you can generally leave these two attributes of xs:schema unspecified.)

References

Further information regarding XML schemas can be found at:

```
http://www.w3.org/TR/xmlschema-0
```

Generated Artifacts Reference

The process of creating a ADO.NET-enabled Data Service control and Web service generates two ADO.NET-specific artifacts:

- XML Schema Definition for ADO.NET Typed DataSet
- Web Services Description Language (WSDL) File for Microsoft ADO.NET Clients

Technical specifications for these artifacts are included in this section.

XML Schema Definition for ADO.NET Typed DataSet

During the process of creating a ADO.NET-enabled data service control, WebLogic Workshop generates a special schema file that conforms to Microsoft's specifications for typed DataSet objects. A schema is generated for each data service query that has been selected for inclusion in the ADO.NET-enabled data service control. These schema files take the name of the source schema's root element.

In the generated schema, the root element has the IsDataSet attribute (qualified with the Microsoft namespace alias, msdata) set to True, as in:

```
msdata:IsDataSet="true"
```

In keeping with Microsoft's requirements for ADO.NET artifacts, the generated target schema of the data service and all schemas on which it depends are contained in the same file as the schema of the typed DataSet. As you select functions to add to the control, WebLogic Workshop obtains the associated schemas and copies the content into the schema file.

In addition, the generated schema includes:

- A reference to the Microsoft-specific namespace definition, as follows:
  `xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"`
- Namespace declaration for the original target schema (the schema associated with the DSP data service)

Listing 6-1 shows an excerpt of a schema—`CustomerDS.xsd`—for a typed DataSet generated from a DSP Customer schema.

Listing 6-1 Example of a Typed DataSet (ADO.NET) Schema

```
<xs:schema xmlns:mstns="http://temp.openuri.org/schemas/Customer.xsd"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
xmlns="http://temp.openuri.org/schemas/Customer.xsd"
targetNamespace="http://temp.openuri.org/schemas/Customer.xsd"
id="CustomerDS" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element msdata:IsDataSet="true" name="CustomerDS">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element ref="CUSTOMER"/>
      </xs:choice>
    </xs:complexType>
</xs:element>
<xs:element name="CUSTOMER">
. . .
</xs:element>
</xs:schema>
```

Web Services Description Language (WSDL) File for Microsoft ADO.NET Clients

The process of generating the Java Web service produces a WSDL for the client-side application development. The WSDL file contains import statements that correspond to each typed DataSet. Each of the import statements is qualified with the namespace of its associated DataSet schema, as in this example:

```
<import namespace="http://temp.openuri.org/schemas/Customer.xsd"
location="LDTest1NET/CustomerDataSet.xsd"/>
```

In addition, the WSDL includes the ADO.NET compliant wrapper type definitions. The wrappers' type definitions comprise complex types that contain sequences of any type element from the same namespace as the typed DataSet, as in:

```
<s:complexType name="CustomerDataSetWrapper">
        <s:sequence>
         <s:any namespace="http://temp.openuri.org/schemas/Customer.xsd"/>
        </s:sequence>
 </s:complexType>
```

CHAPTER 7

Using Workflow with DSP-Enabled Applications

BEA's WebLogic Integration server provides WebLogic Platform components with business-process management (BPM) capabilities. A business process coordinates interaction among various resources to perform a complete set of specific tasks. WebLogic Integration business processes are designed using visual components available, such as Process controls, in WebLogic Workshop.

By bringing WebLogic Integration and BEA AquaLogic Data Services Platform together, developers can achieve sophisticated programming scenarios that might otherwise be difficult, at best.

For example, a WebLogic Integration process (JPD) can be defined that encompasses multiple DSP data services, and that uses the JPD to enforce distributed transactional semantics without using XA and to reduce the number of locks held on disparate data sources (such as Web services or other non-XA-compliant data sources) that might not otherwise be able to participate in the same transaction. In other words JPD is used to achieve atomicity over disparate data sources (see Figure 7-1).

This chapter provides information about such topics as these, and includes information about how to develop server-side workflow-and-DSP-enabled applications. It includes these topics:

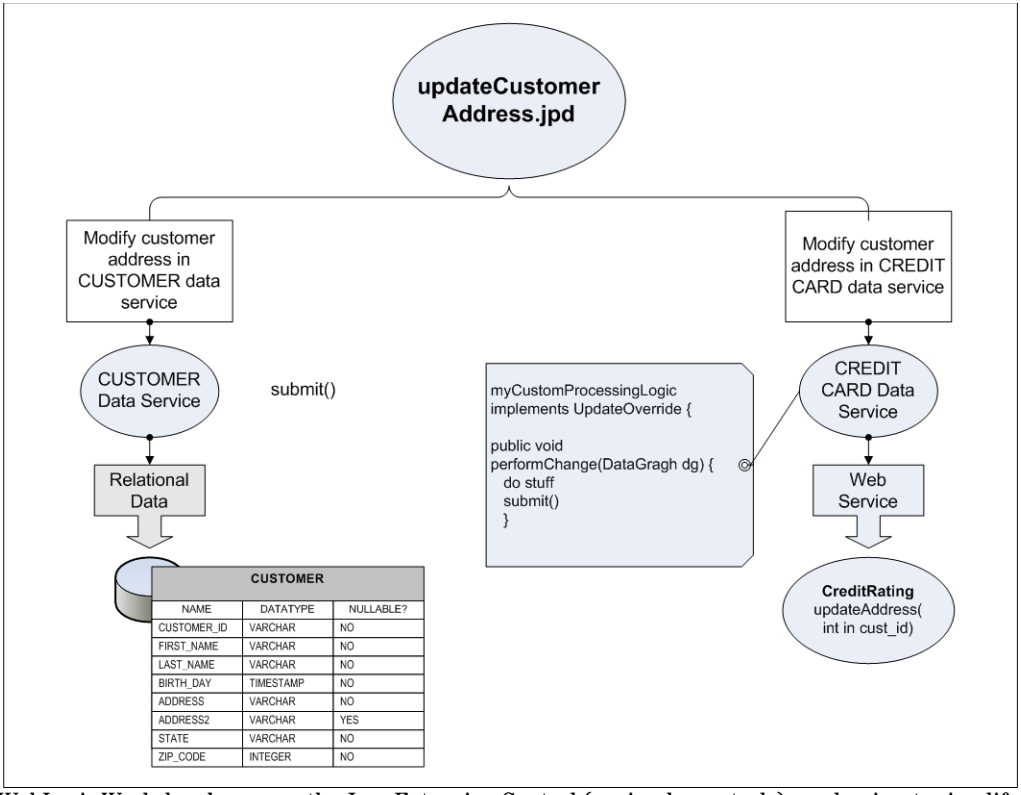
- Adding a Data Service Control to a Process
- Invoking JPDs from Data Services Platform

Brief Overview of WebLogic Integration JPDs

Much of the underlying Java code for the Process (defined in a Java class, as a Java Process Definition, or JPD) is generated or created automatically. Processes coordinate interactions among resources by means of Java controls (Java Control Extensions, or JCX) that are specific to these process definitions.

Using WebLogic Workshop, developers can add various components, including Data Service controls, and customize behavior in the business process, as needed, to accomplish the specifics of the workflow.

Figure 7-1 Using WLI JPD with DSP to Provide Distributed, Two-Phase Commit Capability to Data Service



WebLogic Workshop leverages the Java Extension Control (or simply, controls) mechanism to simplify working with J2EE resources.

A Java Control is an abstraction layer that simplifies working with J2EE resources in WebLogic Workshop.

Controls provide a runtime behavior for accessing functionality and resources using Java classes. WebLogic Workshop provides Controls for numerous WebLogic and AquaLogic components, including Data Service controls for DSP and Process controls for WebLogic Integration.

WLI Process controls enable Web services, business processes, or pageflows to send requests to, and receive callbacks from, a business process (JPD).

See "Accessing Data Services from WebLogic Workshop Applications" on page 5-1 for more information about Data Service controls.

For more information about WebLogic Integration, process controls, and business-process management in general, see the WebLogic Integration documentation page at:

```
http://e-docs.bea.com/wli/docs85/index.html
```

DSP and JPD can be integrated in two different ways:

- By adding Data Service controls to JPD projects you can leverage DSP-enabled application information as part of a workflow.
- By invoking JPDs from DSP-enabled applications. (See "Invoking JPDs from Data Services Platform" on page 7-7.)

Once the JPD is created, it can be called from a data service instance using the JpdService API, a server-side Mediator API that can be invoked in an update override. See "Invoking JPDs from Data Services Platform" on page 7-7 for details.

How SDO's Handling of XMLObjects Differs from JPD

By default, a JPD converts XML objects to an XML proxy class; the class implements the ProcessXML interface. The ProcessXML interface does not know how to handle SDO objects, such as change summaries.

You must override the default behavior in the JPD by editing the source code.

Adding a Data Service Control to a Process

You can use Data Services Platform in WebLogic Integration (WLI) business process applications through a Data Service control. For example, you might add DSP-based information to decision-making logic in the business process.

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

- Creating a Data Service Control
- Adding a Data Service Control to a JPD File
- Setting Up the Data Service Control in the Business Process

Creating a Data Service Control

Before you can execute a Data Services Platform query from 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.

See "Accessing Data Services from WebLogic Workshop Applications" on page 5-1 for more information about creating Data Service controls.

In WebLogic Workshop:

1. Create a Process application.
2. Create a Data Services project in the Process application. In the Data Services project, import the existing Data Service projects that you want to incorporate into the JPD.
3. Create a Data Service control, adding the functions you want to use from the data services to the control.
4. When the process is defined, you can then generate a Process control from the JPD, from within WebLogic Workshop (right-mouse click on the Design view of the JPD and select Generate Process control from the popup menu).
5. The control is generated.

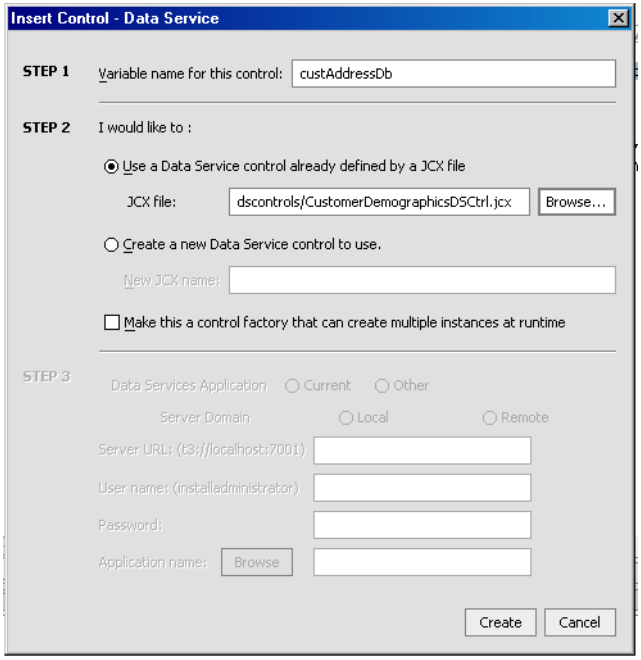
For complete details, see "Data Service Controls" on page 5-2.

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 WebLogic Workshop Data Palette.

The Data Service controls must be created in the same project as the JPD.

Figure 7-2 Creating a Data Service Control



Setting Up the Data Service Control in the Business Process

Once the Data Service control has been added to the business process, its functions are available. As shown in Figure 7-3, 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 7-3 Specifying in the Business Process Input and Output Parameters for a Data Service Control

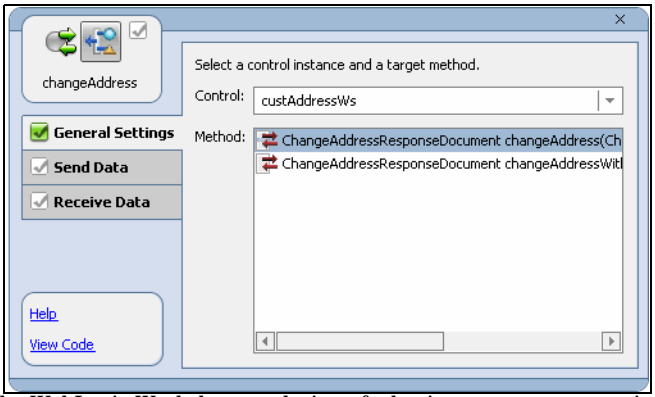
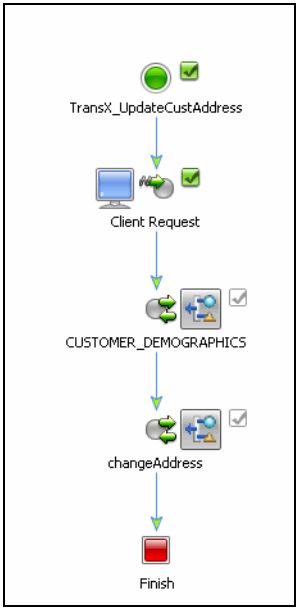


Figure 7-4 shows the WebLogic Workshop rendering of a business process accessing a Data Service Control.

Figure 7-4 WebLogic Integration Business Process Accessing a Data Service Control



Submitting Changes from a Business Process

By default, a business process (Java process definition, or JPD) converts XML objects to an XML proxy class by implementing the ProcessXML interface. 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 XMLBean serialization on the XML object factory by calling the XmlObjectVariableFactory.setXBean().
2. In the JPD you need to disable the XMLBean serialization and turn on the default ProcessXML deserialization on the XML object by calling XmlObjectVariableFactory.unset().
3. Invoke the Data Service control.

Invoking JPDs from Data Services Platform

Data architects writing Java custom update classes can create a JPD workflow to handle the updates to different data services. Data Services Platform developers can then write server-side Java code that initiates synchronous or asynchronous JPDs using the JpdService interface.

As with other types of DSP server-side custom functionality, the update override interface facilitates the implementation.

Update overrides are user-defined Java classes that implement DSP's UpdateOverride interface (from the sdoupdate package). Update overrides are registered in DSP and invoked by the Update Data Mediator when an SDO is submitted for an update. As its name implies, an update override implements custom processing on the server, for data updates. Update overrides are required to update non-relational data sources. See Chapter 9, "Customizing Data Service Update Behavior," for more information.

The JPD and the data service containing the Java update override can be running in the same WebLogic Server domain or in different WebLogic Server domains.

Invoking a JPD from an Update Override

An update override can use a JPD to process requests. The JpdService is invoked with the name of the JPD, the start method of the JPD, the service URI, and the server location and credentials for the JPD, as shown in this example:

```
JpdService jpd = JpdService.getInstance("CustOrderV1",
    "clientRequestwithReturn", env);
```

JPD provides a public interface (as a JAR file containing the compiled class file for the JPD public contract or interface). Transparently to developers, the JpdService object uses the standard Java reflection API to find the JPD class that implements the JPD public contract.

The server-side update overrides Java code and then passes the DataGraph as an argument to the invoke method:

```
Object jpd.invoke( DataGraph sdoDataGraph );
```

The returned object is dependent on the JPD being invoked and may be null. Typically, if any top level SDO is being inserted and its primary key is autogenerated, then this should be returned from the JPD (see Listing 7-1).

Any keys for the top-level DataObject in the serialized UpdatePlan are returned to the calling function as a Properties object (comprising a byte array). Thus, the return value from the workflow must be a serialized byte array, as in:

```
Properties [] jpd.invoke( byte[] serializedUpdatePlan );
```

The array returned is a Properties object array representing any keys for the top-level DataObject in the UpdatePlan that was serialized and sent to the workflow.

Invoking a JPD by Using the JpdService API in an Update Override

Support for JPDs from DSP is provided through two server-side APIs that can be invoked from within an UpdateOverride implementation (see Table 7-5).

Table 7-5 JpdService API)

Data Type	Signature
JpdService	`JpdServicegetInstance(String jpdClass, String jpdStartMethod, Environment context)`
	`JpdService.getInstance(String jpdClass, String jpdStartMethod, String serviceUri, Environment context);`

Listing 7-1 shows how to invoke a JPD from an UpdateOverride. The code sample assumes that a JPD exists comprising a series of data services configured as part of a workflow.

Listing 7-1 Sample Code Listing—Invoking a JPD from a DSP UpdateOverride

```
public boolean performChange( DataGraph ) {
  ChangeSummary changeSum = dataGraph.getChangeSummary();
  //Size of 0 means no changes so there's nothing to do
```

```
  if (changeSum.getChangedDataObjects().size()==0) {
      return true;
   }
   Environment env = new Environment();
   env.setProviderUrl( "t3://localhost:7001" );
   env.setSecurityPrincipal( "weblogic" );
   env.setSecurityCredentials( "weblogic" );
   try {
      JpdService jpd = JpdService.getInstance(
                       "CustOrderV1",
                        "clientRequestwithReturn",
                        env);
      UpdatePlan updatePlan = DataServiceMediatorContext.
                    currentContext().getCurrentUpdatePlan( dataGraph );
      byte[] bytePlan =  UpdatePlan.getSerializedBytes( updatePlan );
      Properties (Properties) returnProps = jpd.invoke( bytePlan );
     }
     catch( Exception e )
     {
        e.printStackTrace();
        throw e;
     }
     return false;
   }
}
```

Synchronous and Asynchronous Behavior

Data Services Platform supports JPD invocations both synchronously and asynchronously; both styles of invocation are handled the same way in the DSP update override code. Invoke the JPD and get the response back as a byte array, as shown in Listing 7-1 above.

Error Handling

You must write your own error-handling code with the JPD. Calling a non-existent JPD raises the standard Java "ClassNotFoundException."

Using callbacks in your JPD is not supported. Business processes that include client callbacks will fail at runtime since the callback is sent to the JPD Proxy rather than the originating client that started the JPD.

CHAPTER 8

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:

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 8-1 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_PROPERTY1,"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.executeQuery();
       return resultSet;
}
```

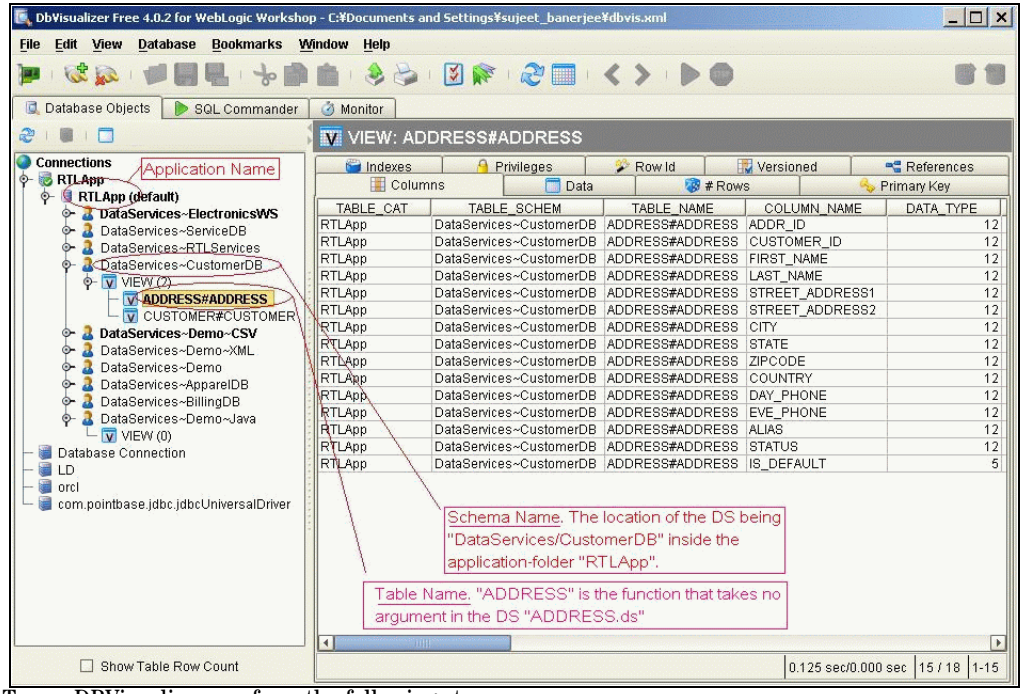
Connecting to the JDBC Driver from a Java Application

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 8-2 shows a sample application as viewed from DbVisualizer for WebLogic Workshop.

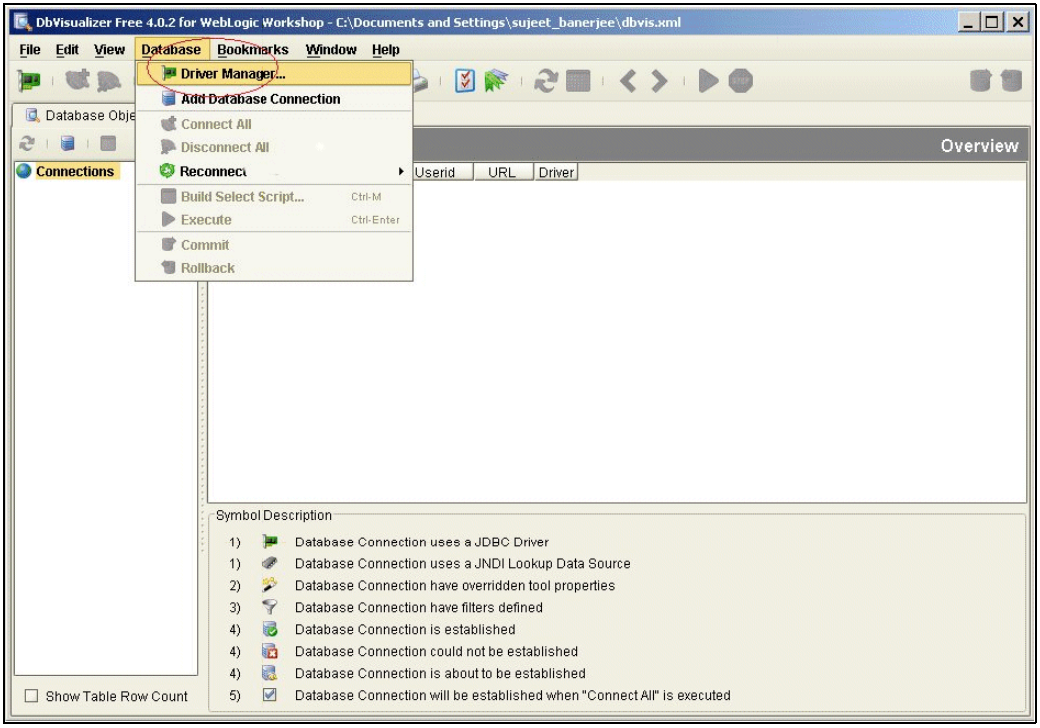
Figure 8-2 DbVisualizer View of DSP



To use DBVisualizer, perform the following steps:

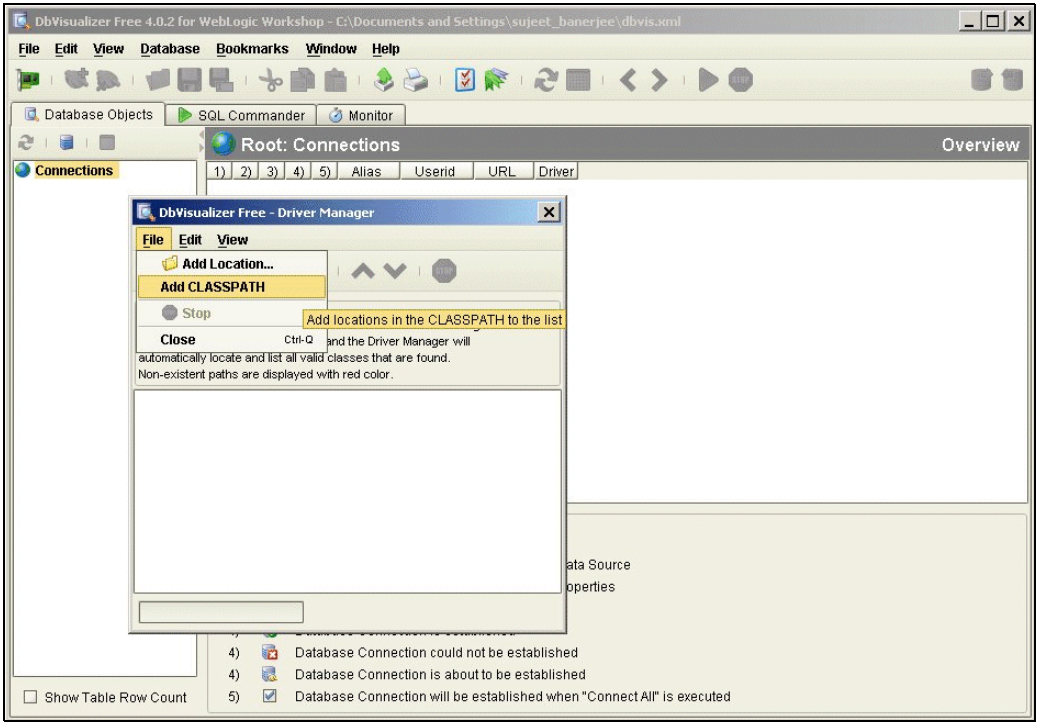
1. Configure DBVisualizer.

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



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.

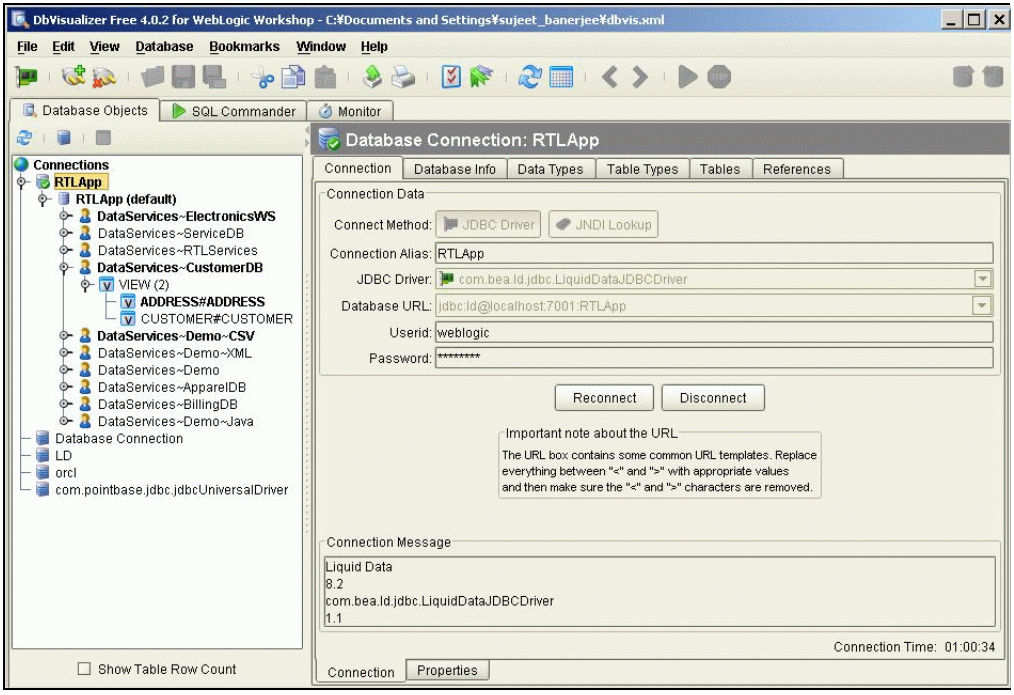
2. Add connection parameters by performing the following steps:

   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"`

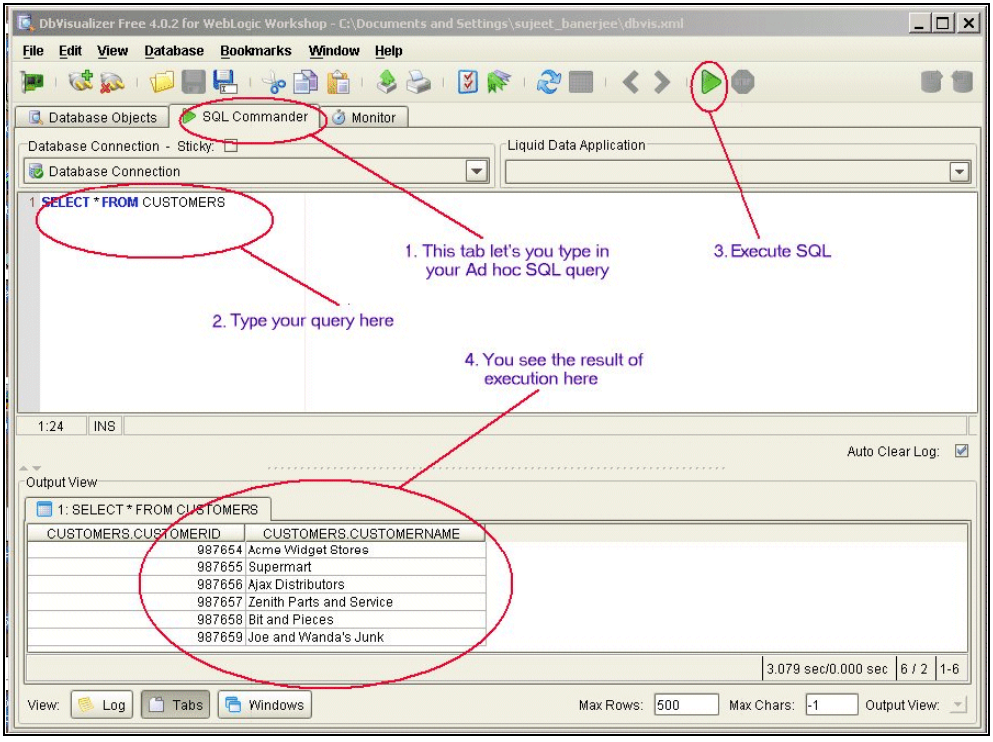
   c. Provide the username and password for connecting to the DSP application.

3. Click connect. On completion of a successful connection, you should see the following:



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.


DbVisualizer Free 4.0.2 for WebLogic Workshop - C:\Documents and Settings\sujeet_banerjee\dbvis.xml
File Edit View Database Bookmarks Window Help
Database Objects
SQL Commander
Monitor
Database Connection - Sticky:
Database Connection
Liquid Data Application
1 SELECT * FROM CUSTOMERS
1. This tab let's you type in your Ad hoc SQL query
2. Type your query here
3. Execute SQL
4. You see the result of execution here
1:24
INS
Auto Clear Log:
Output View
1: SELECT * FROM CUSTOMERS
CUSTOMERS.CUSTOMERID
CUSTOMERS.CUSTOMERNAME
987654 Acme Widget Stores
987655 Supermart
987656 Ajax Distributors
987657 Zenith Parts and Service
987658 Bit and Pieces
987659 Joe and Wanda's Junk
3.079 sec/0.000 sec
6 / 2
1-6
View: Log Tabs Windows
Max Rows: 500
Max Chars: -1
Output View:

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.

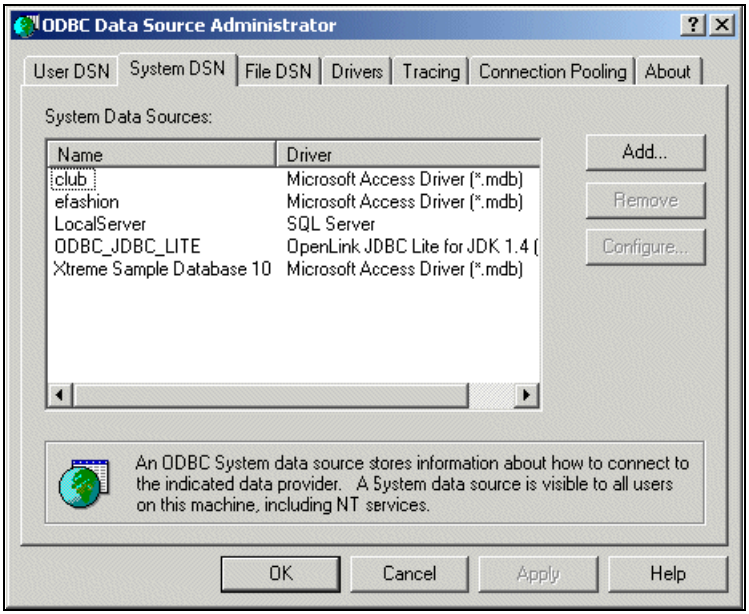
To use the EasySoft bridge, perform the following steps:

1. Install the EasySoft ODBC-JDBC bridge. Go to the EasySoft site for information about installation:

   `http://www.easysoft.com`

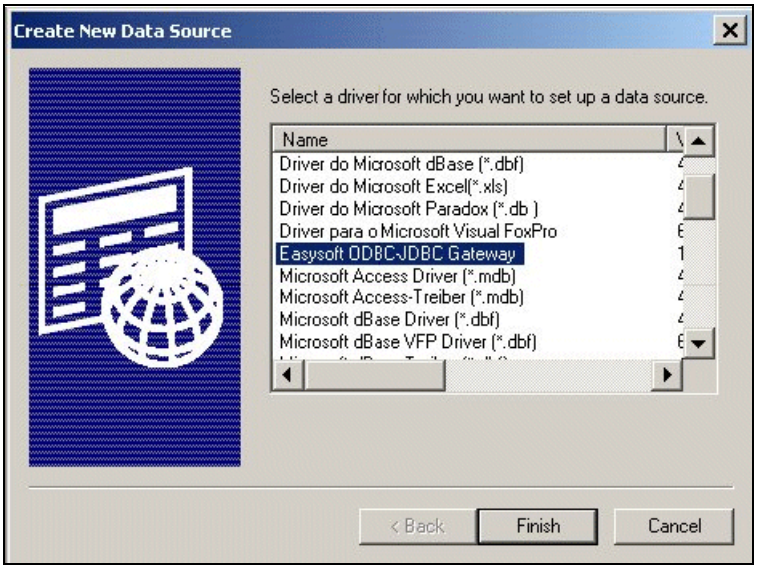
2. Creating a system DSN and configuring it with respect to DSP by performing the following steps:

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.



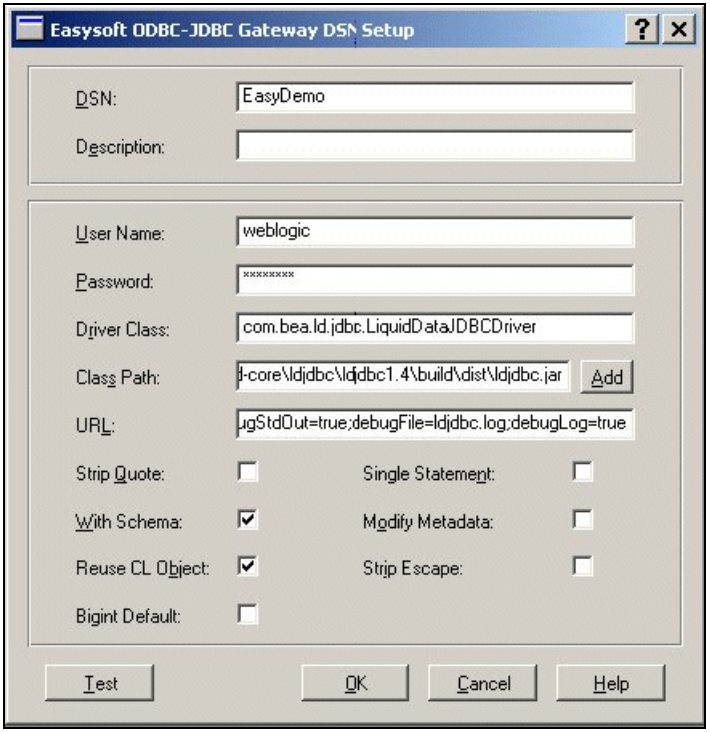
d. On the next screen, fill in the fields as follows:

- For Class Path, enter the absolute path to the ldjdbc.jar
- For URL, enter:

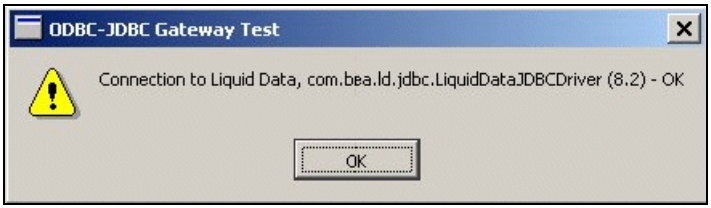
```
jdbc:ld@<machine_name>:<port>:<app_name>
```

- 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.

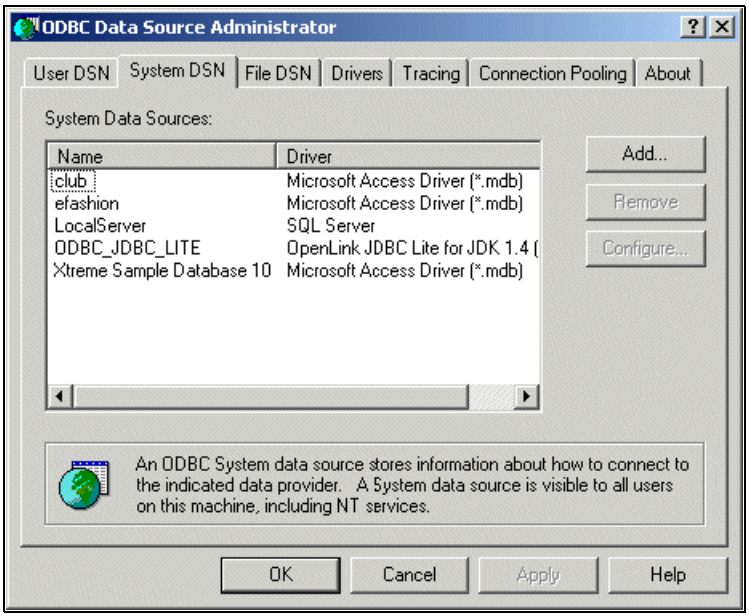
2. Create a system DSN and configure it for your DSP application by performing the following steps:

   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;
   ```

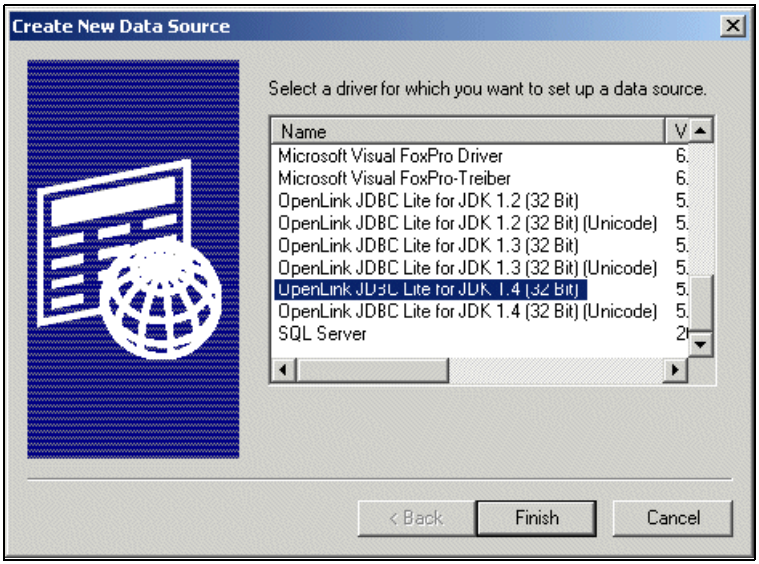
   b. Update your system path to point to the `jvm.dll`, which should be under your `%javaroot%/jre/bin/server` directory.

   c. Open Administrative tools Data Sources (ODBC). You should see the following:

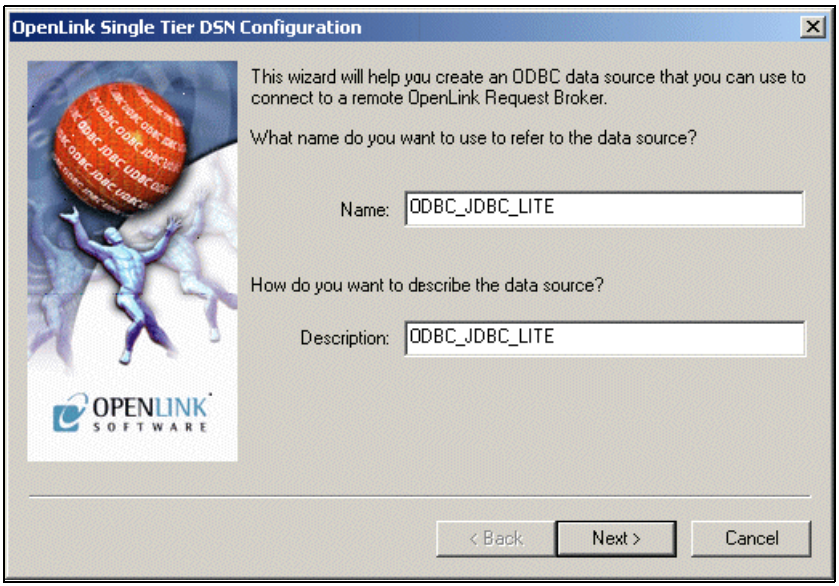


d. Go to the System DSN tab and click Add.

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:

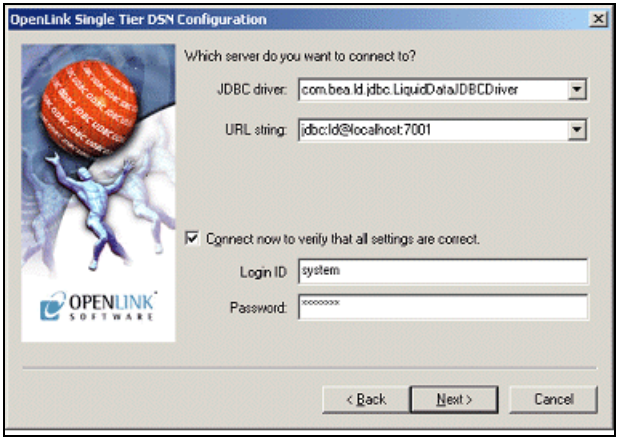


g. Click Next. Then on the next screen, enter the following in the JDBC driver field:

```
com.bea.ld.jdbc.LiquidDataJDBCDriver.
```

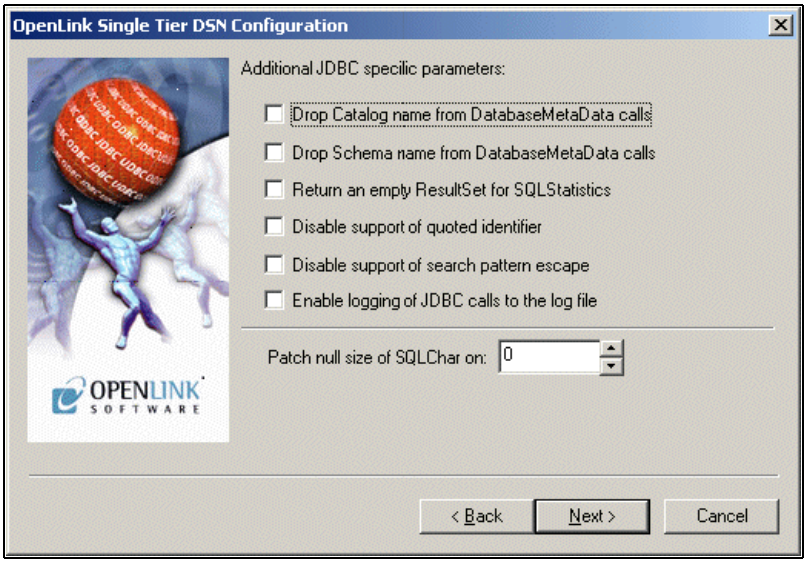
Enter the following in the URL string field:

```
jdbc:ld@<machine_name>:<port>:<app_name>
```

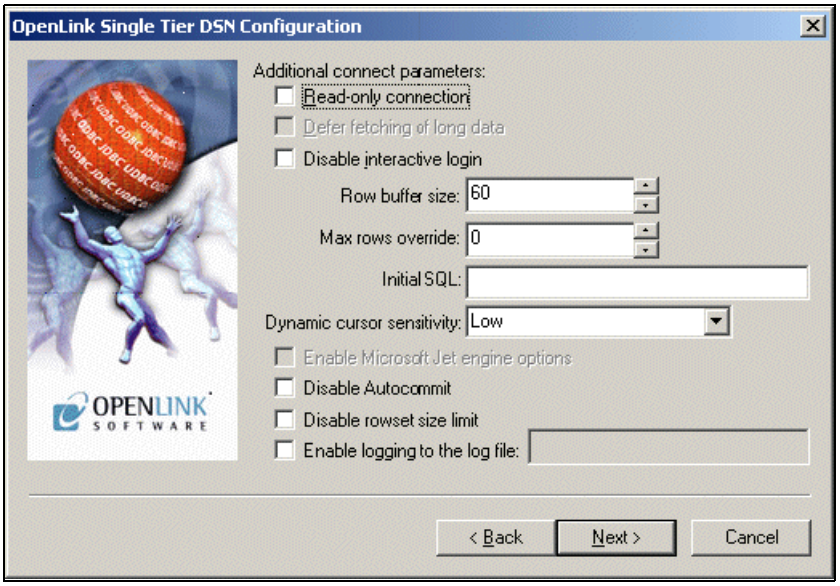


h. Check the Connect now to verify that all settings are correct checkbox. Provide the login and password to connect to the Data Services Platform WebLogic Server.

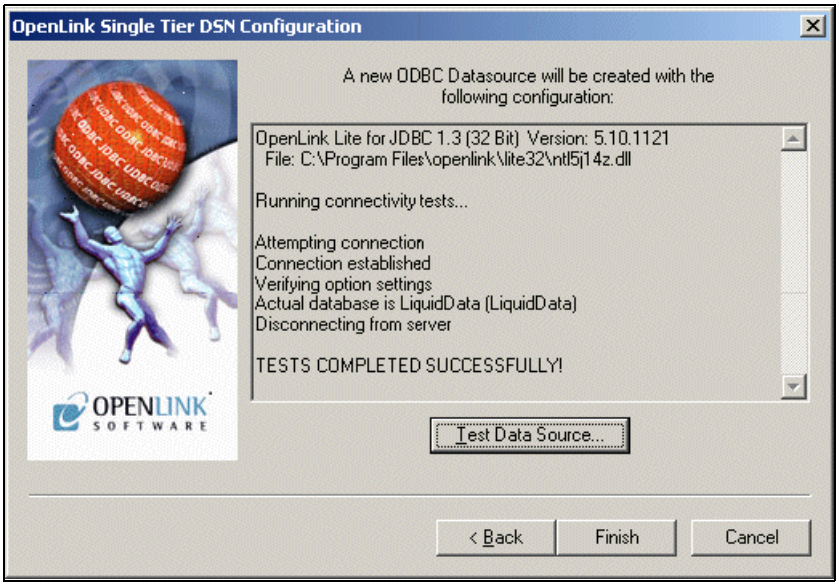
i. Click **Next**. The screen shown below will display:



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

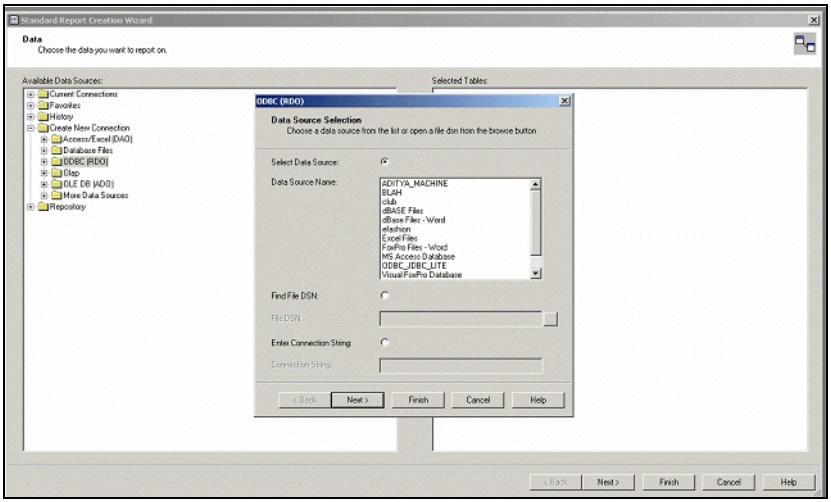
Note: Some reporting tools issue multiple SQL statement executions to emulate a scrollable cursor if the ODBC-JDBC bridge does not implement one. Some drivers do not implement a scrollable cursor, so the reporting tool issues multiple SQL statements. This can affect performance.

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:

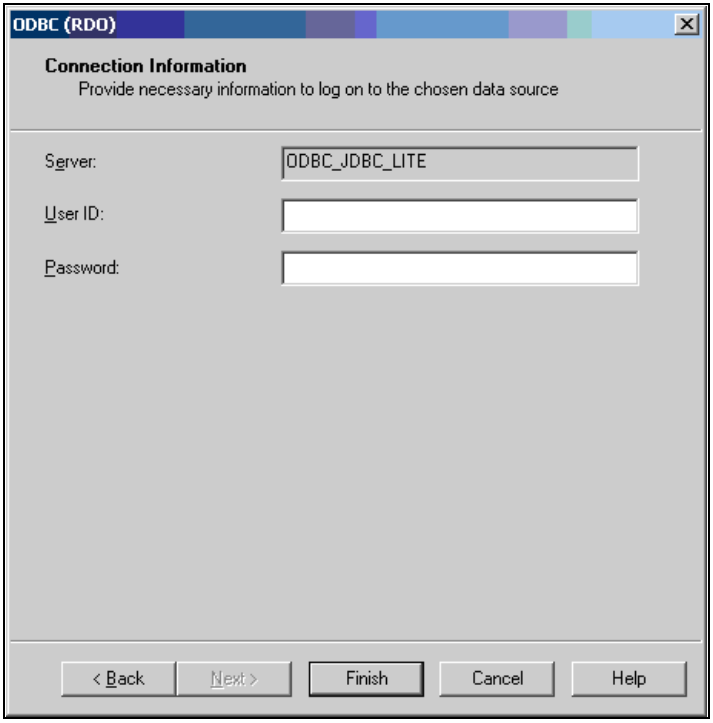
1. In Crystal Reports 10, you need to create a new Connection on ODBC RDO. You can do this by clicking on the New Report wizard button, which will prompt you immediately for a data source. Select the ODBC (RDO) option in the left-hand window as shown in the Figure 8-3.

Figure 8-3 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.

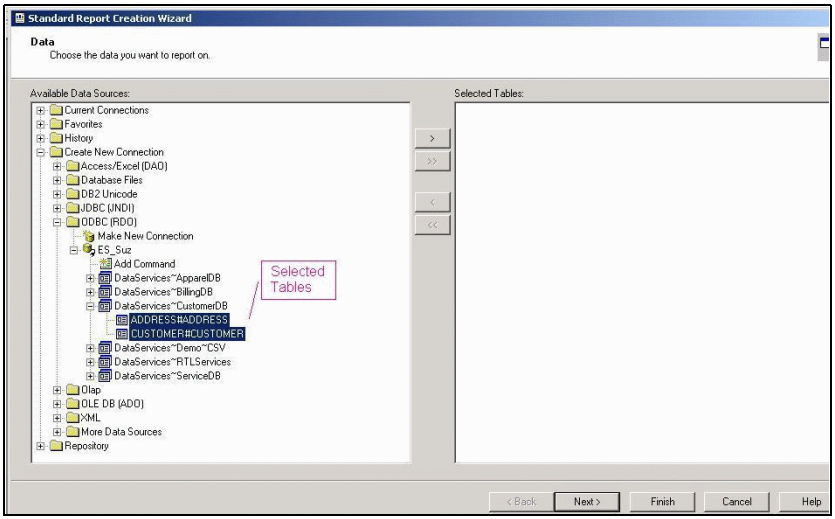
Selecting ODBC_JDBC_LITE, prompts the following dialog:



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".)

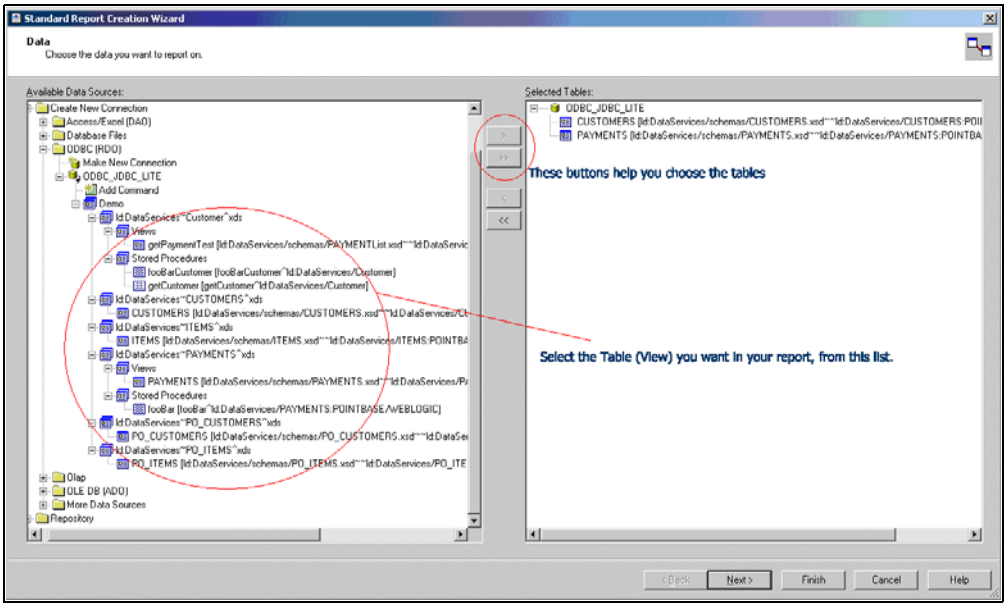
   Once authenticated, Crystal Reports will show you a view of the DSP application on the server as shown in Figure 8-4.

Figure 8-4 Available Data Sources



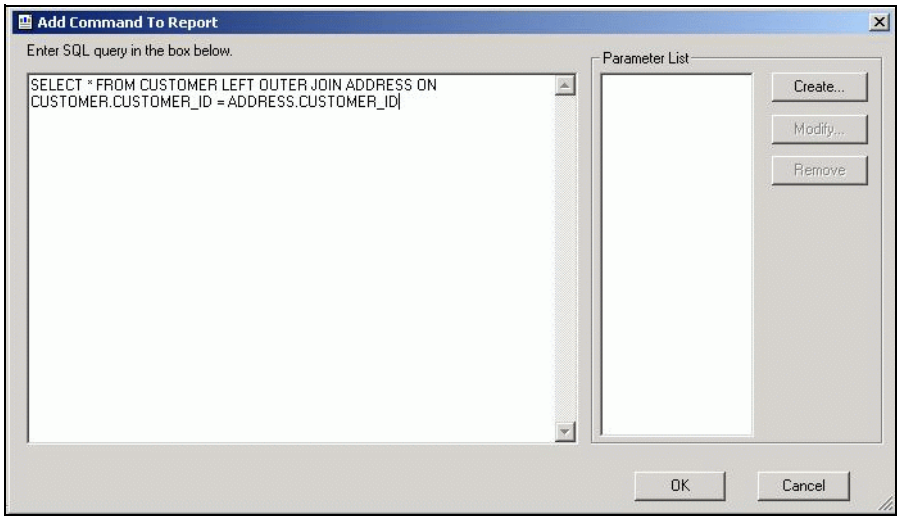
3. Generate a report using the Add command or by dragging the metadata to the right. In this example we will be using both options. You can choose the tables you want to use in the report as shown in Figure 8-5.

Figure 8-5 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 8-6.

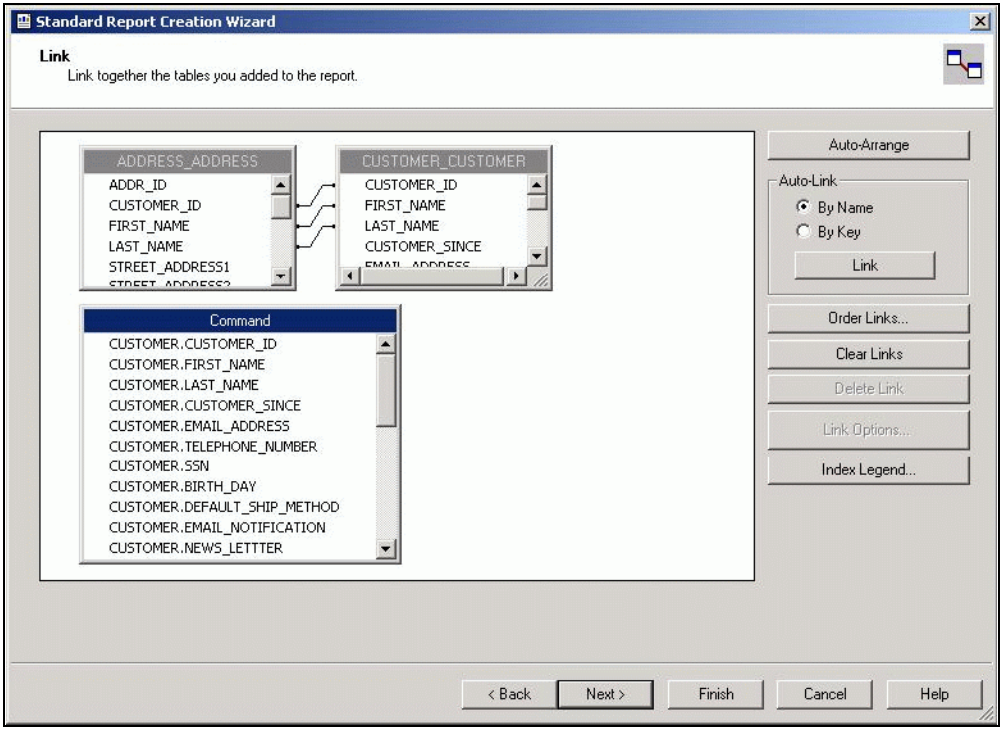
Figure 8-6 Add Command



4. Click the Ok Button to see the Command added to the Right hand side of the window.

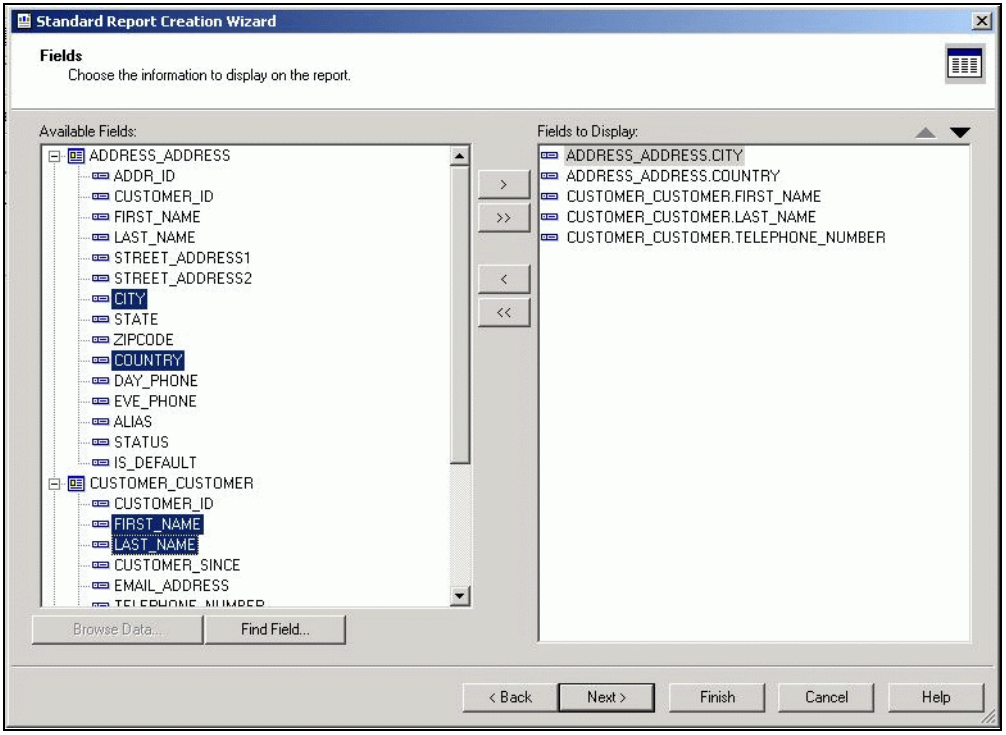
   Clicking Next in the wizard shows you all the available views for this Report generation, as shown in Figure 8-7.

Figure 8-7 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 8-8.

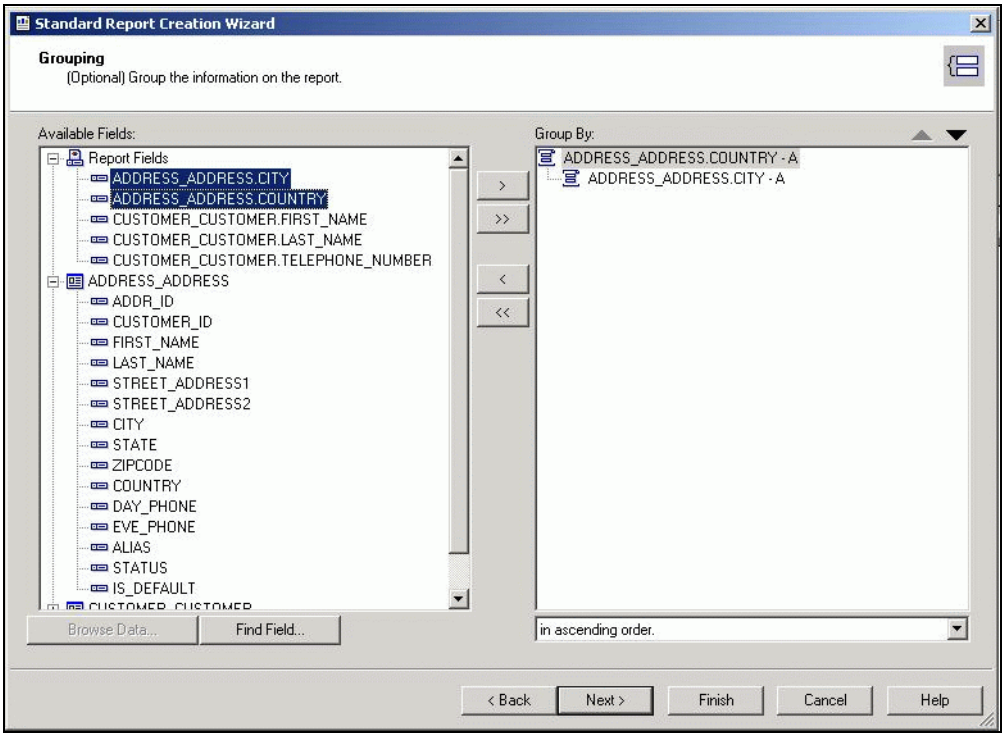
Figure 8-8 Column Chooser



Note: This example chooses columns from the user-generated Command and the view CUSTOMER.

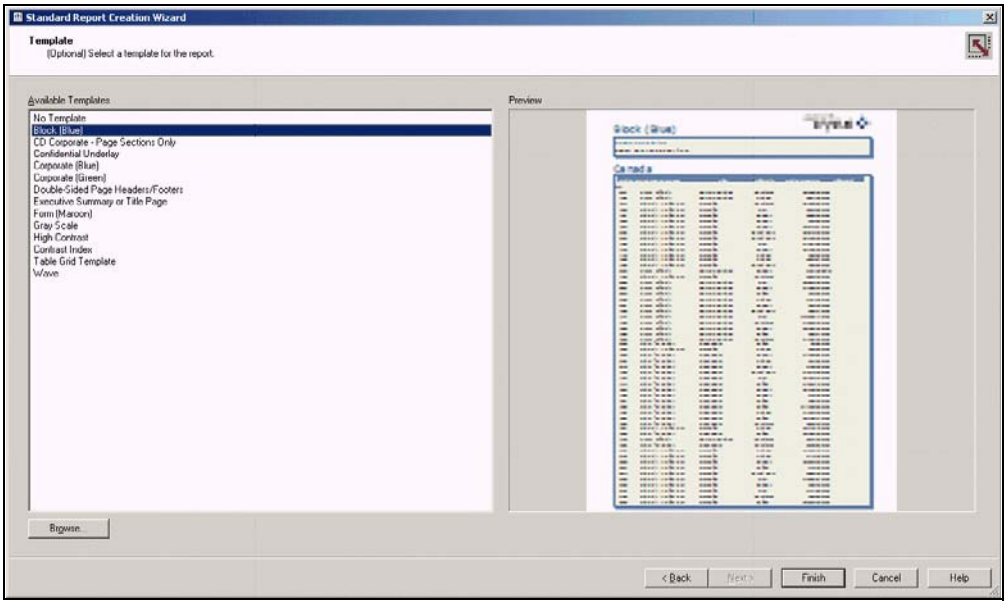
Clicking on Next again takes us to the Group by screen (as shown in Figure 8-9), which allows you to choose a column to group by. (This is grouping is performed by Crystal Reports. The Group-by information is not passed on to the JDBC driver.)

Figure 8-9 Group-by Screen



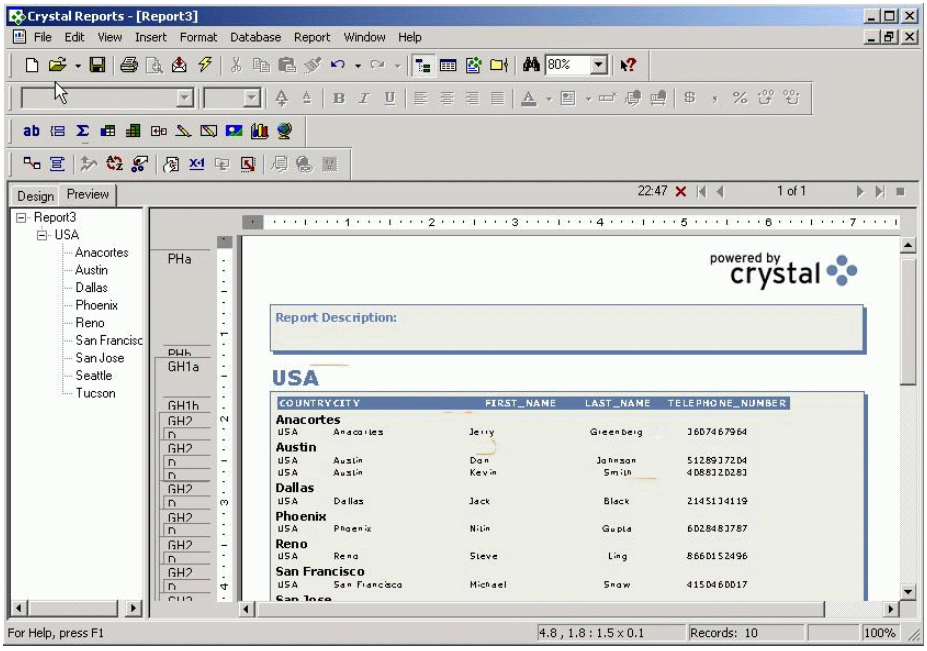
5. Skip the next few screens for now, clicking Next till you reach the Template Chooser Screen Figure 8-10. Choose any appropriate Template. In this example, the user has chosen the Block (Blue) Template.

Figure 8-10 Template Chooser Screen



6. Click Finish. A Report similar to that shown in Figure 8-11 is generated.

Figure 8-11 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 8-12.

Figure 8-12 Connection Dialog Box

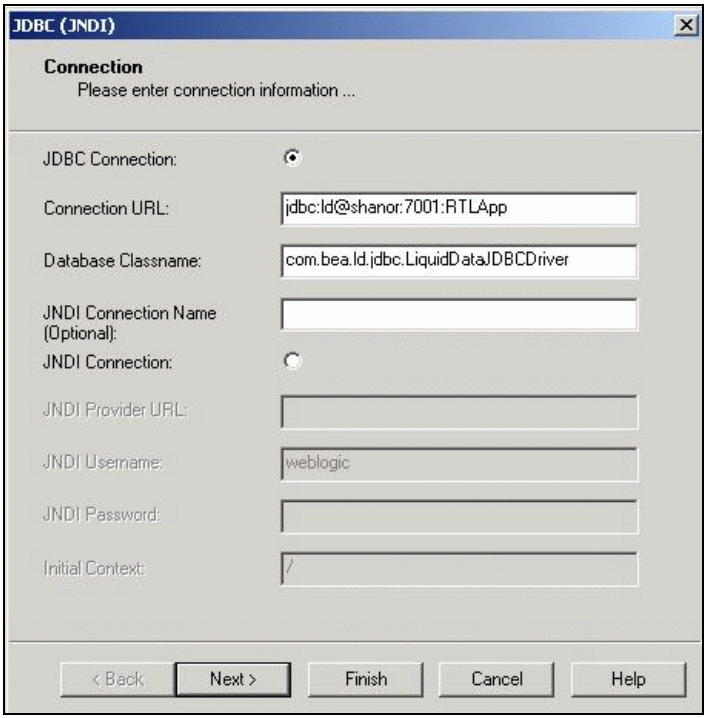
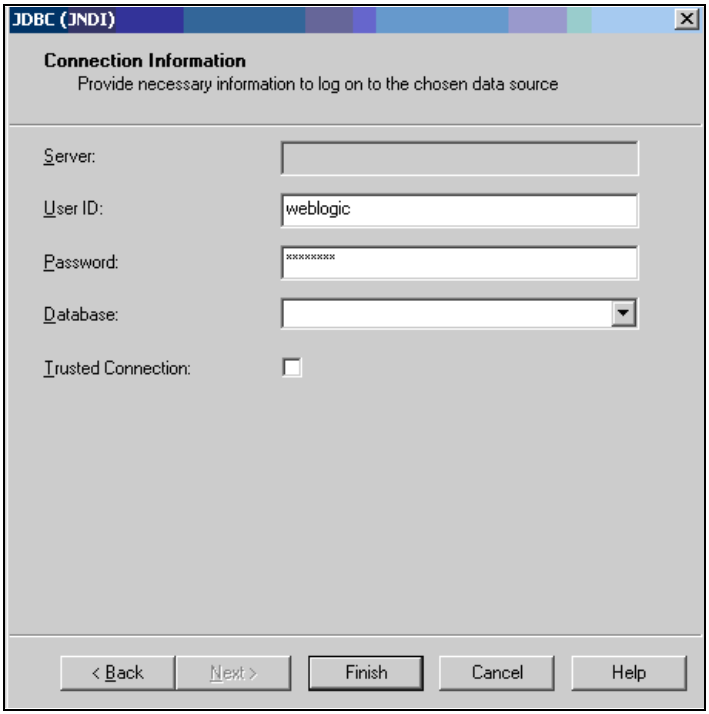


Figure 8-13 shows screen that requests the connection parameters for the JDBC Interface of Crystal Reports.

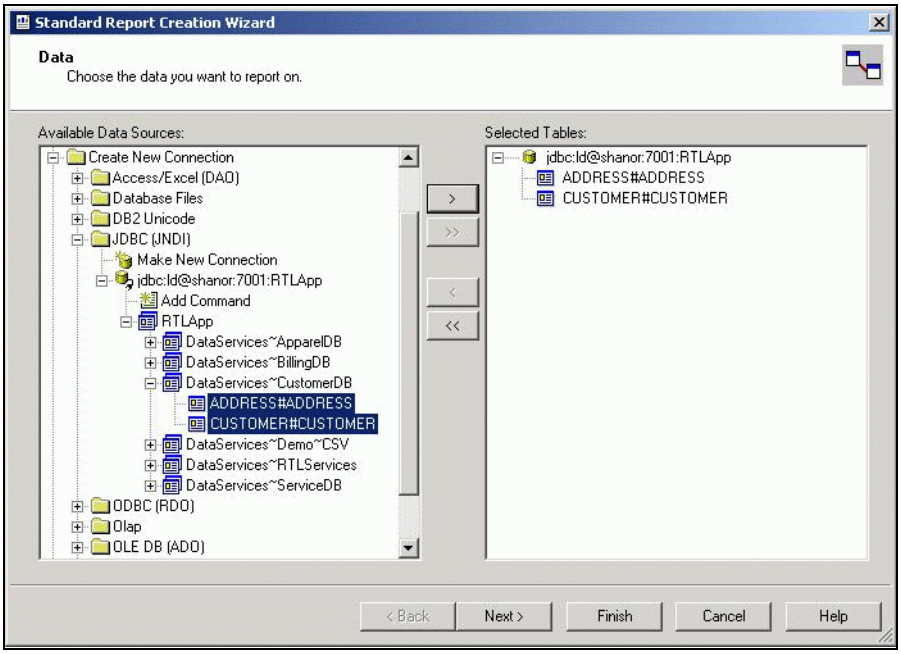
Figure 8-13 Connection Information Dialog Box



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 8-13.

Clicking the Finish button on the previous screen. This takes you the metadata browser shown in Figure 8-14. The rest of the process is similar to the procedure described in the section "Crystal Reports 10 - ODBC."

Figure 8-14 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.

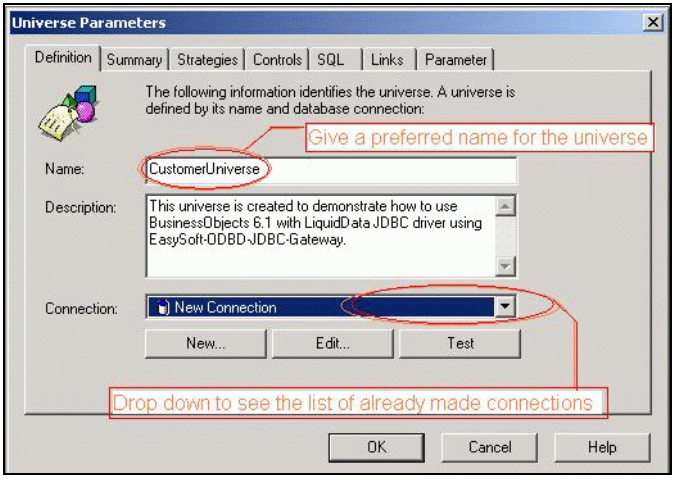
To generate a report, perform the following steps:

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.

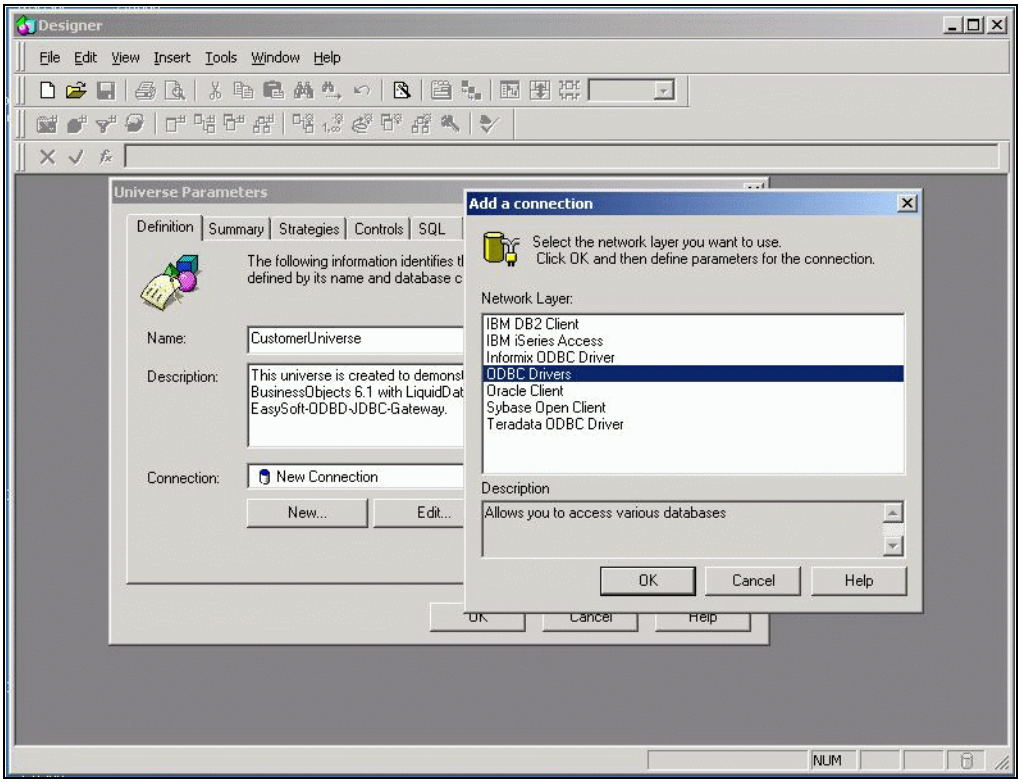
   b. Fill in a name for your Universe and select the appropriate DSN connection from the drop-down list, as shown in Figure 8-15.

Figure 8-15 Selecting the DSN Connection



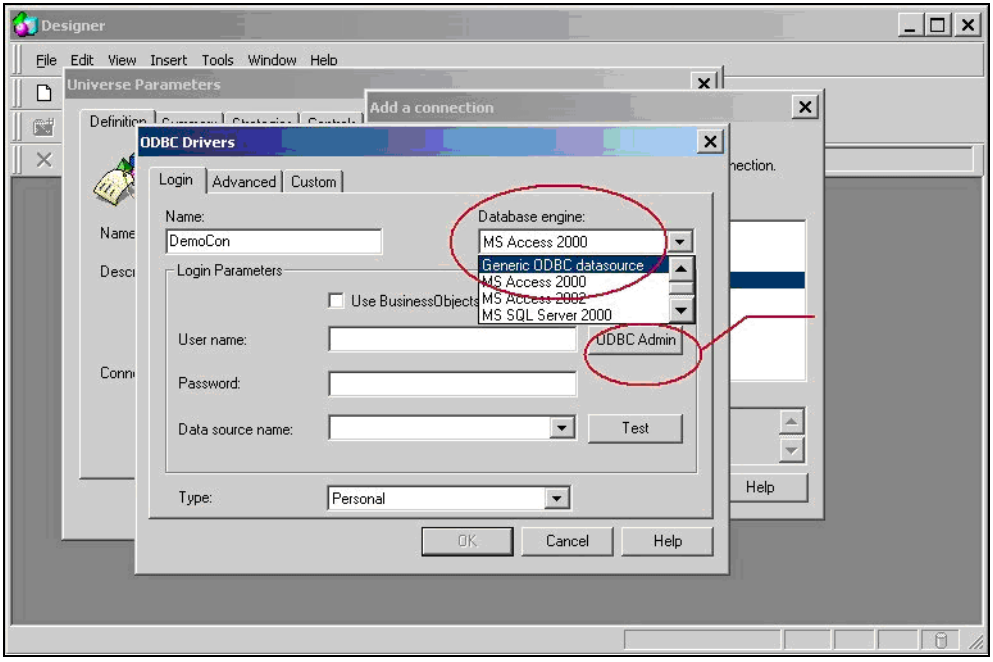
c. If the DSN you wish doesn't appear in the list (this happens if you are using the application for the first time), use New to create a new connection. Select ODBC Drivers, as shown in Figure 8-16, and click OK.

Figure 8-16 Selecting the ODBC Drivers



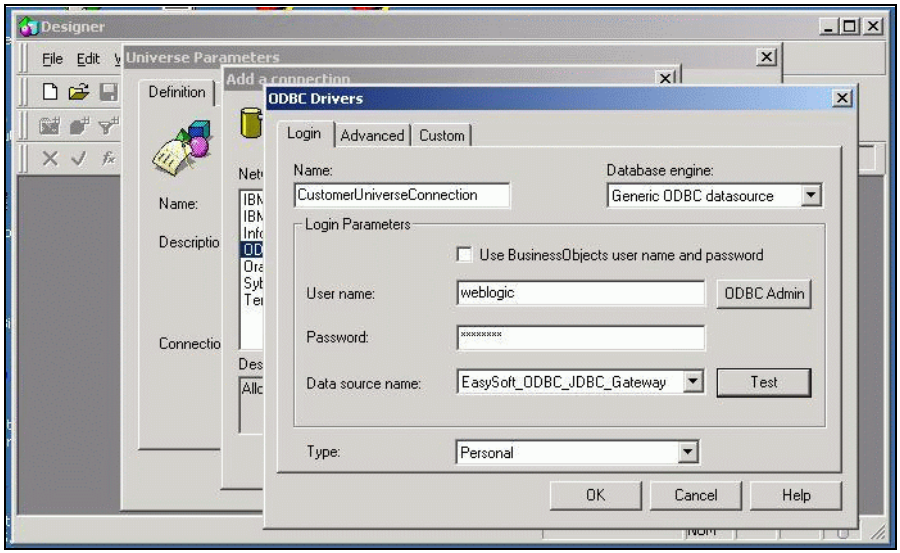
d. Now select the database engine as a Generic ODBC data source, as shown in Figure 8-17. Use the ODBC Admin button to check if the DSN you wish is already created. For any help creating a DSN using OpenLink or EasySoft please refer to the section ODBC-JDBC bridge of this document.

Figure 8-17 Selecting the Database Engine



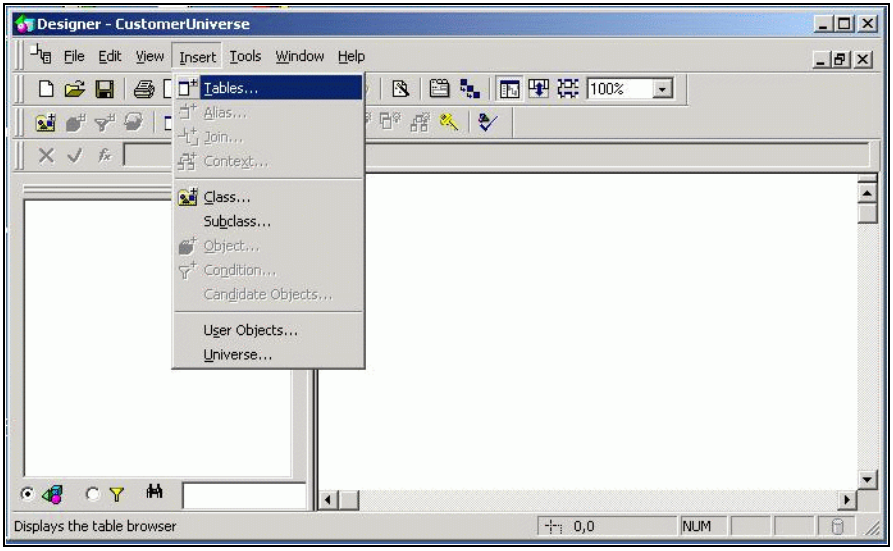
e. Now select the data source name as shown in Figure 8-18. This would be the name of DSN you wish to connect to. Refer to the picture below. Click OK to get back to the Universe creation window.

Figure 8-18 Selecting the Data Source Name



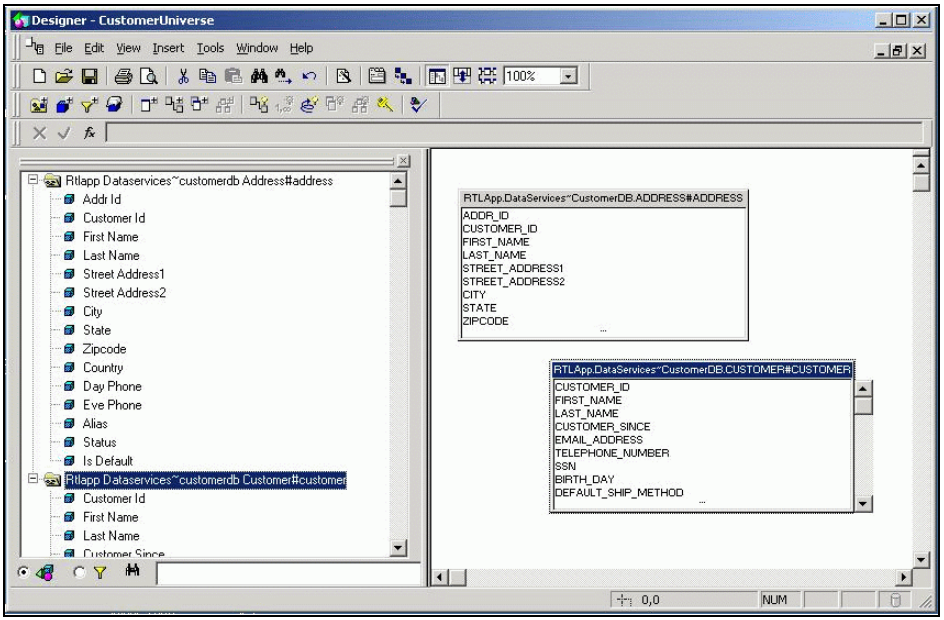
f. Fill in the other details and click Test to see if the connection is successful. Click OK. You should see a new blank panel, as shown in Figure 8-19.

Figure 8-19 Designer UI Screen



g. From the Insert menu select Table, as shown in Figure 8-19. 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 8-20.

Figure 8-20 Table Browser

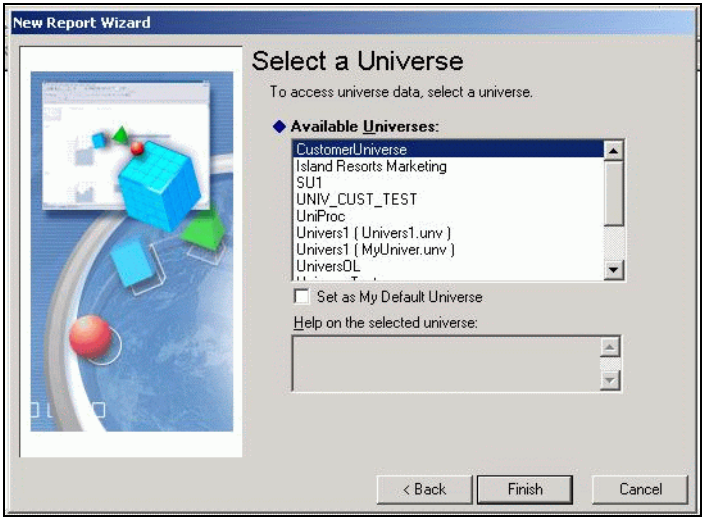


h. Save the Universe and exit.

2. Creating a report using the New Report wizard. To create a new report, follow these steps:

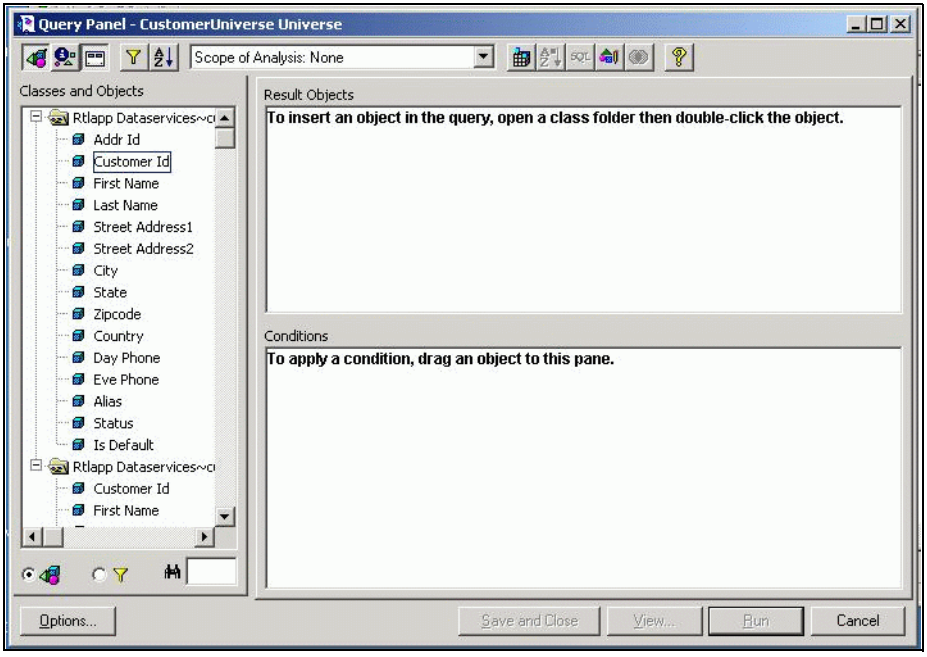
   a. Run the Business Objects application. Click New to open the New Report Wizard. Choose Specify to access data and click Begin. You should see the dialog-box shown in Figure 8-21.

Figure 8-21 Available Universe Dialog Box



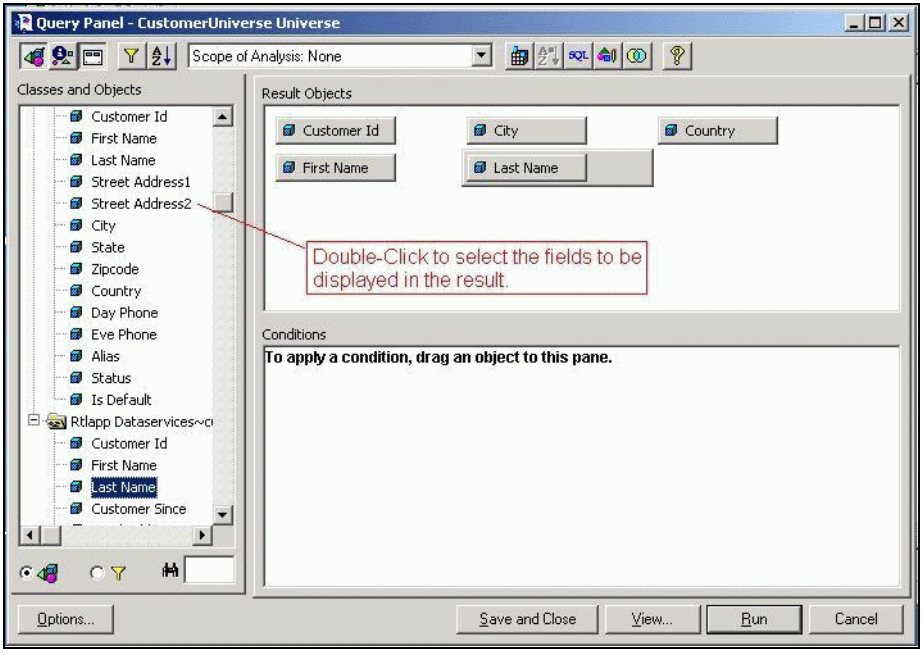
b. Choose a Universe. Click Next. On the left pane, you should see the tables and their fields (columns) on expansion, as shown in Figure 8-22.

Figure 8-22 Query Panel



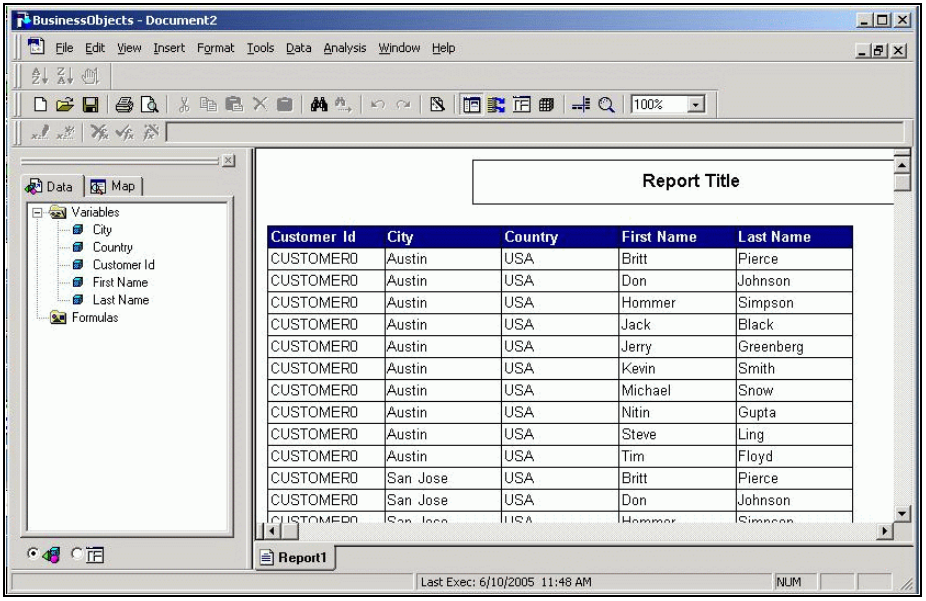
c. Select the Universe of your choice and click Finish. Double-click a column (table-field) in the left pane to select it in the result, as shown in Figure 8-23.

Figure 8-23 Selecting the Object



d. Click Run to execute the query. The result is seen as shown in Figure 8-24.

Figure 8-24 Business Objects Panel



3. You can execute the pass-through queries as follows:

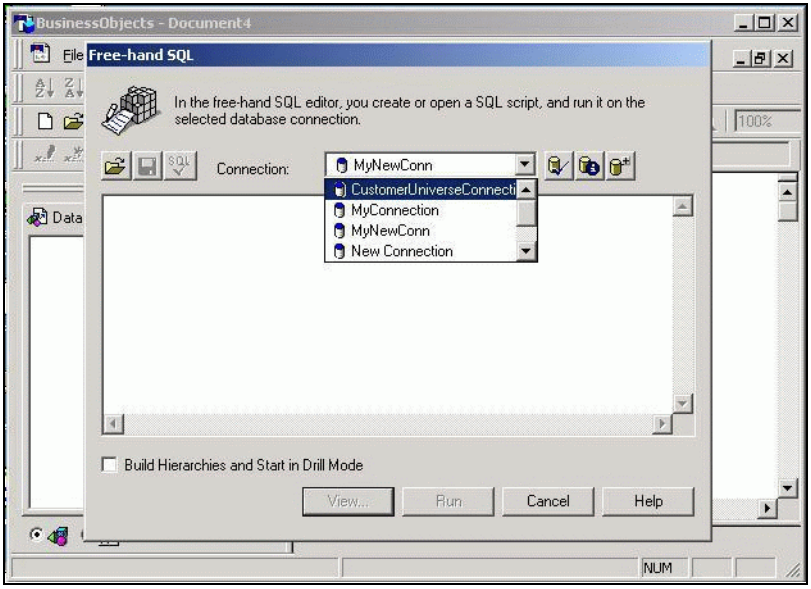
   a. In the Business Object application, click New to create a new report.

   b. In the New Report Wizard choose Others instead of Universe as shown in Figure 8-25.

   c. Choose Free-hand SQL and click Finish.

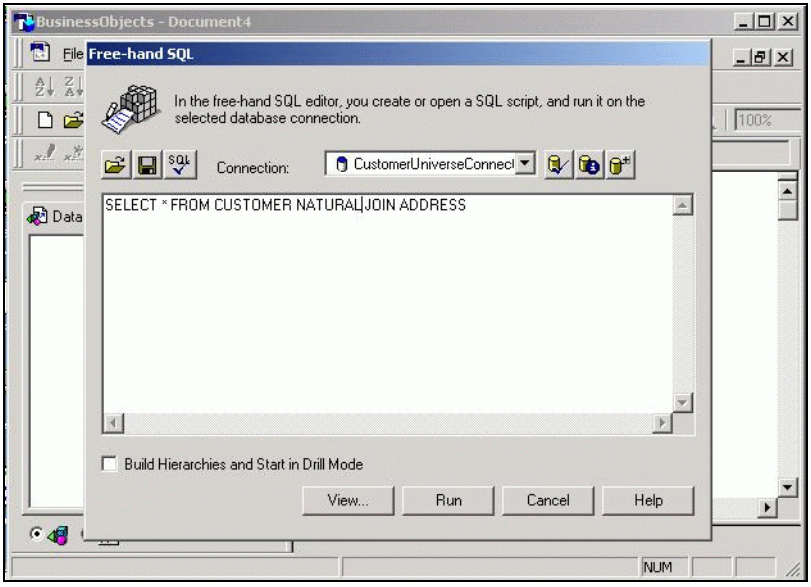
d. Select the connection you made using Designer 6.1, as shown in Figure 8-25.

Figure 8-25 Free Hand SQL Menu



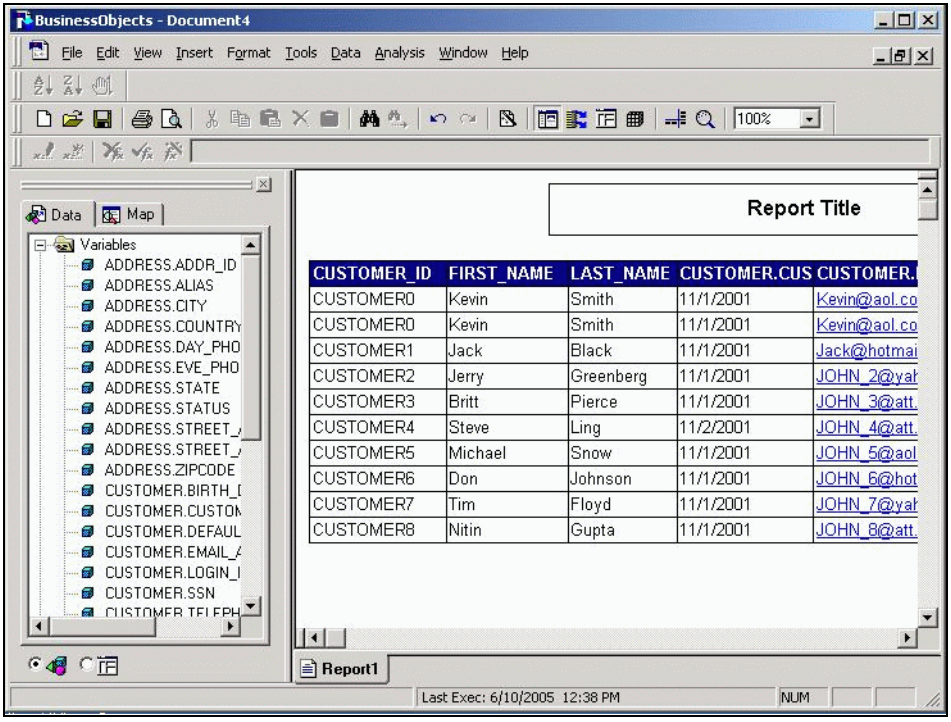
e. Type in your SQL query and click Run to generate the report, as shown in Figure 8-26.

Figure 8-26 Specifying the SQL Query



f. Click Run. You should see the report shown in Figure 8-27.

Figure 8-27 Business Objects Report



Microsoft Access 2000 - ODBC

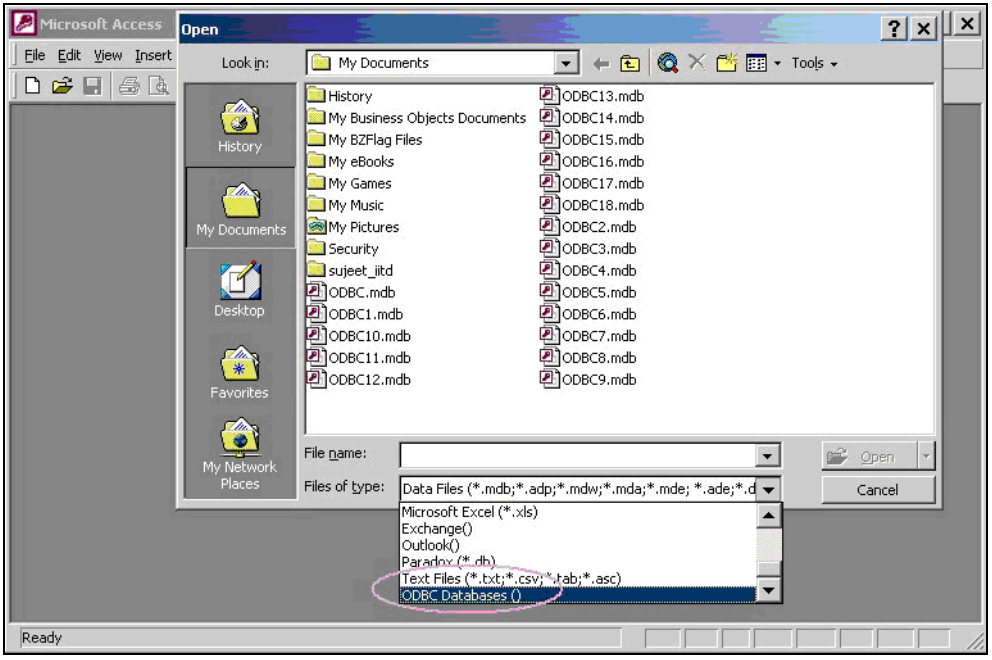
This section describes the procedure for connecting Microsoft Access 2000 to DSP through an ODJB-JDBC bridge.

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.

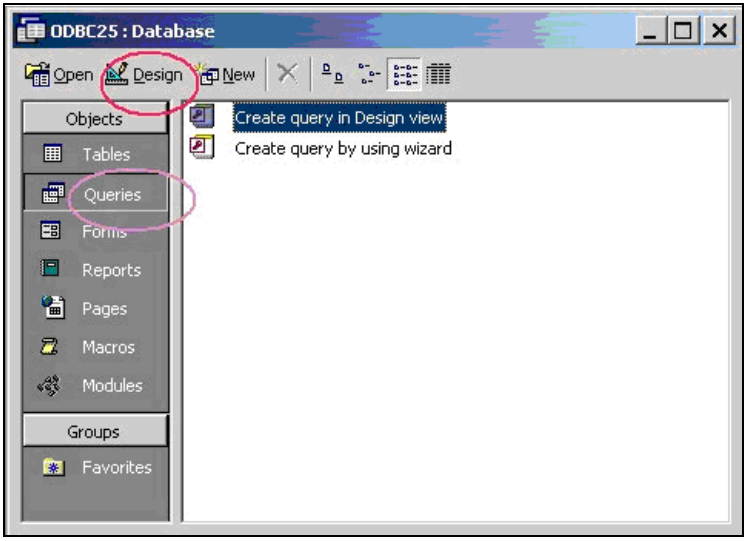
1. Run MS Access, click File Open, then select ODBC Databases as the file type as shown in the Figure 8-28.

Figure 8-28 Selecting the ODBC Database in Access



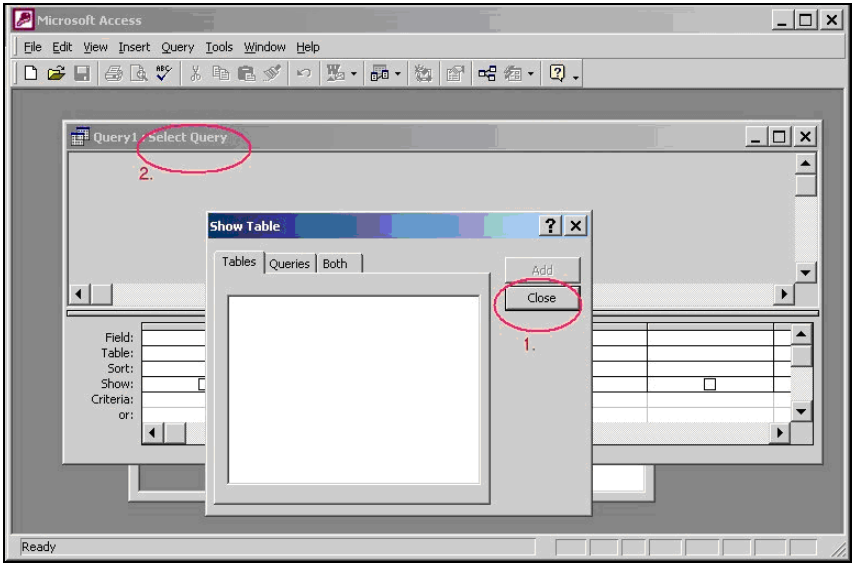
2. Once the dialog Select Data Source pops up, click Cancel to close it. You should see the window shown in Figure 8-29.

Figure 8-29 OBDC23: Database Screen



3. Click Queries, then Design as indicated in Figure 8-29. You should see a screen shown similar to that shown in Figure 8-30.

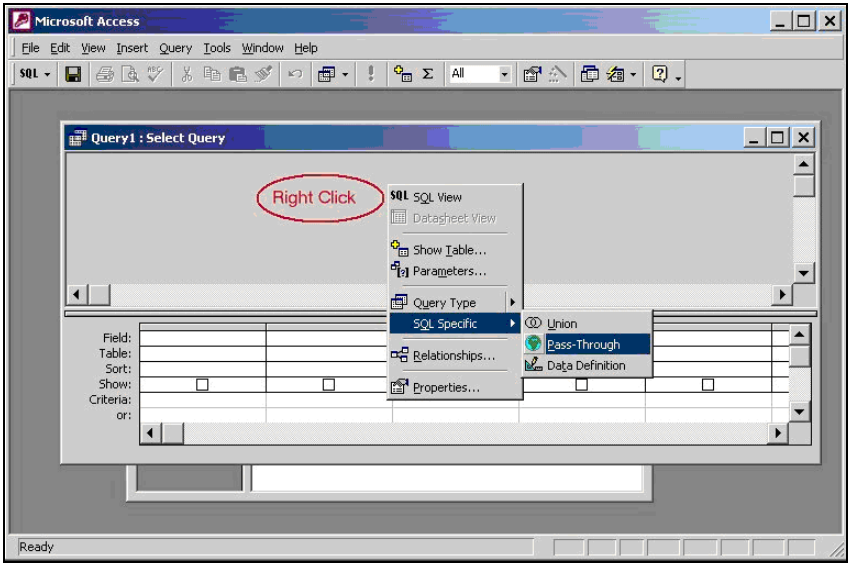
Figure 8-30 Select Query and Show Table Screens



4. Close the Show Table dialog box. You should now be able to see the Select Query dialog.

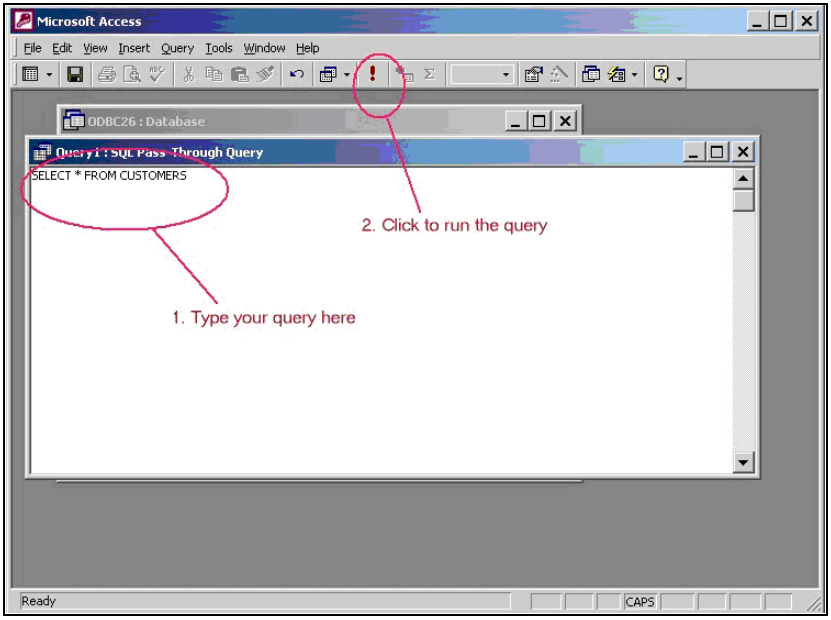
5. Right click in the upper pane and select SQL Specific →Pass-Through as indicated in Figure 8-31. This will open an editor.

Figure 8-31 Selecting SQL Specific and Pass Through



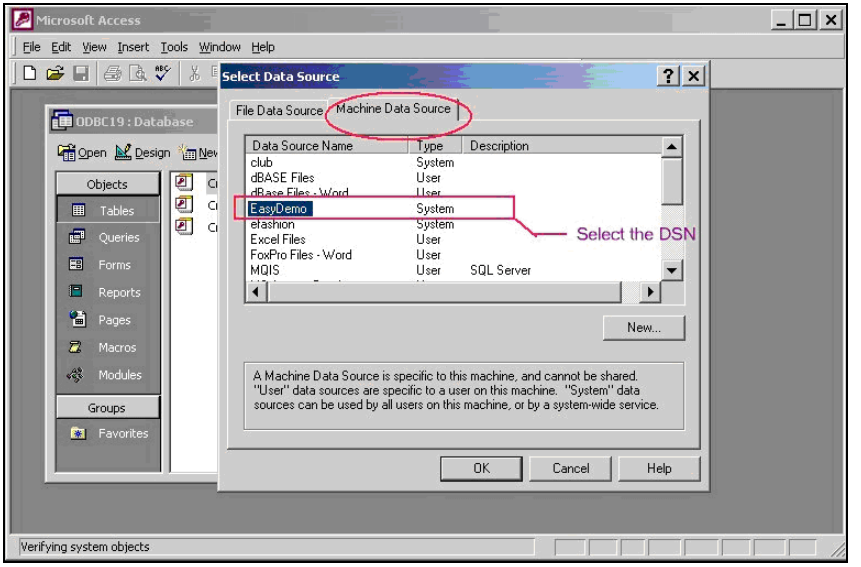
6. Type in your SQL query and click Run, as shown in the Figure 8-32.

Figure 8-32 Running the SQL Query



7. In the dialog that pops up (as shown in Figure 8-33), move to the tab Machine Data Source and select the appropriate DSN for the database connectivity.

Figure 8-33 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 8-34.

The XML types are defined by xmlns:xs="http://www.w3.org/2001/XMLSchema". The Java types are defined by java.sql.Types.

Table 8-34 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`
`xs:decimal`	`Types.DECIMAL`

Table 8-34 XML to SQL Type Mapping

XML Type	SQL Types
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 table_name can have an alias, but you cannot specify the `colmn_name_alias_list` 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)	The SQL-92 specified default column ordering in the set operations is supported. Both the table-expressions (the operands of the set-operator) must conform to the same relational schema.	(SELECT NAME, CITY FROM CUSTOMER1) UNION CORRESPONDING BY (CITY, NAME) (SELECT CITY, NAME FROM CUSTOMER2) The supported query is: (SELECT NAME, CITY FROM CUSTOMER1) UNION (SELECT NAME, CITY FROM CUSTOMER2)

Unsupported Feature	Comments	Example
"...table1 UNION table2..."	Not supported. Also not supported are set operations between tables in a FROM clause, except through a sub-query. The TABLE keyword is not supported.	SELECT * FROM TABLE CUSTOMER1 UNION TABLE CUSTOMER2 Where TABLE is a keyword not supported by the LDJDBC SQL interface. The supported version is: SELECT * FROM (SELECT * FROM CUSTOMER1 UNION SELECT * FROM CUSTOMER2) T1 Other supported UNION constructs: SELECT * FROM CUSTOMER1 UNION SELECT * FROM CUSTOMER2 SELECT * FROM CUSTOMER1 UNION (SELECT * FROM CUSTOMER2 UNION SELECT * FROM CUSTOMER3)
SELECT-query within the SELECT clause	Not supported.	SELECT A, (SELECT B FROM C) FROM... WHERE...

CHAPTER 9

Customizing Data Service Update Behavior

BEA AquaLogic Data Services Platform handles updates to relational data sources automatically. However, for any non-relational data sources, including Web services, you must provide the update logic by writing an update override class and associating it with the data service. In addition, there are times when you may want (or need) to provide custom update logic for relational data sources as well.

Any data service, logical or physical, can have an associated update override class to perform a variety of customizations.

This chapter explains how to create an update override class (the class comprising the update behavior) and when you may want to do so for relational data sources. It includes the following topics:

What is an Update Override?

An update override provides you with a mechanism for customizing or completely replacing the default update process (as discussed in "How It Works: The Decomposition Process" on page 2-16). With an update override associated with your data service, you can:

- Invoke data service functions or procedures.

- Execute externally defined JPDs (Java process definition) to perform workflow operations from a data service. For example, you can initiate a workflow that ties together numerous data services to accomplish distributed transactional semantics across data services that comprise non-XA-compliant data sources (such as Web services).
- Validate changes before submitting them, checking or modifying the values in some way.
- Invoke other resources, for example, by passing modified values to a workflow or Web service.
- Execute SQL statements directly within the update plan.
- Log changes to an external log file.
- Perform virtually any other customization required.

An Update Override is a Java Class

In programming terms, an update override is a Java class; it is a compiled Java source code file that implements the UpdateOverride interface (<UpdateOverride>, one of the DSP APIs located in the `com.bea.ld.dsmediator.update` package). The UpdateOverride interface has as its sole method an empty performChange( ) method (see Listing 9-1).

As shown in Listing 9-1, the performChange() method takes a DataGraph object (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 (as discussed in "Data Services Platform and Service Data Objects (SDOs)" on page 2-2.)

Listing 9-1 UpdateOverride Interface

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

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

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

    }
}
```

As you can see from the performChange( ) method signature (Listing 9-1), it returns a Boolean value. This value serves as something of a flag to the Mediator, as follows:

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

How an Update Override Affects Update Processing

The performChange( ) method will be executed whenever a submit is issued for objects bound to the overridden data service.

If the object being passed in the submit() is an array of DataService 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.)

In your code, you should verify that the root data object for the data graph being passed at runtime is an instance of the singleton data object bound to the data service (configured with the update override).

When Are Update Overrides Required?

You must create custom update classes to update any non-relational data sources—Web services, XML files, flat-files, and DSP procedures, for example, and for these types of scenarios:

- Initiate a workflow (business process or JPD) from a DSP application.
- Compute your own primary key value when adding a data object as a new record to an RDBMS.
- Handle circular dependencies that arise when modifying or adding objects with mutual dependencies.

  For example, your client application code is adding both a *department* and a manager; however, manager is also a required field of department. How can you set the department's manager field before the manager exists? As follows:

  – 1) Add department with manager set to a temporary value
  – 2) Add the employee manager
  – 3) Reset the department manager to the new employee.

Once you have written and compiled the Java code that comprises the update override class, you must register the class with the data service. Update overrides can be registered on physical or logical data services: Each data service has an Override Class property that can be associated with a specific Java class file that comprises the implementation of the UpdateOverride.

At runtime, the data service executes the UpdateOverride class when it identifies it as available during the decomposition process (see "Logical Data Service Update Process" on page 2-18).

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, as discussed in When Are Update Overrides Required for Relational Data Sources?

When Are Update Overrides Required for Relational Data Sources?

DSP automatically updates relational data sources. However, in some cases, such as those listed Table 9-1, DSP cannot automatically update relational data sources, and requires that you provide an update override to handle update processing.

Table 9-1 Issues that Can Interfere with Automatic Relational Data Source Updates

Issue	Description, example, or recommendation
Ambiguous data lineage	The data service decomposition function cannot contain "if-then-else" constructs that provide alternate composition from lower level data services.
Transformation issue	The lineage involves a transformation other than data() or rename. For example, the following would not be supported by automatic updates: `<ACCOUNT> { sum(data($C/ACCOUNT)) }; </ACCOUNT>`
Multiple lineage for a composed property	An example of a property with more than one lineage, or data source, for a property: `<customerName>{ cat(data($C/FNAME), " ", data($WS/LAST_NAME)) }; </customerName>`

Issue	Description, example, or recommendation
Nested matching logic issue	Typically, nested containment is expressed in XQuery using a where clause. If the query does not use a where clause to implement nesting, DSP cannot determine the foreign key-primary key association. (Nested matching logic should be expressed in a where predicate clause.) 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.
Ambiguous tuple identity	Distinct-values or group-by would lead to an arbitrary tuple remaining from a set of duplicate tuples.
Redundant instance values	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.
Repeating complex type values issue	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 that provide the data object values. 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.
Typematch issue	If the query plan of the decomposition function has a “typematch” node, the decomposition will stop at that point for the SDO.

Developing the UpdateOverride Class

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

1. Create a new Java class file in the DSP project. (If you do not add the Java class file to the project, it must be in the classpath.) You can put the class anywhere in the application folder. For basic projects, you can simply add the class to the same directory as your data services. For larger projects, you might want to keep update classes in their own folder.

   g. Import the appropriate DSP API and SDO DataGraph packages into the class in which you are implementing the UpdateOverride interface:

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

   h. Your Java class declaration must implement the UpdateOverride interface, as in:

   ```
   public class SpecialOrders implements UpdateOverride
   ```

   i. 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)
   ```

   j. In the body of the performChange( ) method, implement your processing logic. Your processing logic can access the changed object; instantiate new data objects;modify and submit them, or access the Mediator context's update plan and decomposition map. You can also invoke a data service procedure from within this method, or invoke a JPD.

2. Compile the Java source code to create the class file.

3. Associate the class file with a specific data service by embedding the appropriate text in the data service source code (the .ds file) or by setting the Update Override property on the data service. WebLogic Workshop is used for either approach, albeit from within two different view tabs, as follows:

   a. Add the name of the update override class (classname only, without the .class extension) as an attribute of an empty `javaUpdateExit` element tag (in the pragma statement of the data service). For example:

   ```
   <javaUpdateExit className="SpecialOrderUpdate"/>
   ```

   b. Alternatively, open the Property entering the class name in the Update Override property WebLogic Workshopas the update override class Update Override for specific data service by referring to it from the data service by placing a javaUpdate element in the pragma statement of the data service.

Listing 9-2 is an example of an update override implementation.

Listing 9-2 Update Override Sample

```
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 9-2, an OrderDetailUpdate class implements the UpdateOverride class, and, as required by the interface, defines a performChange( ) method. Listing 9-2 demonstrates a common coding pattern for update overrides:

- The submitted data graph (as changed by the client application) is passed to the performChange( ) method.
- The data graph's root data object is obtained and then cast to an `ORDERDETAILDocument` using the variable name orderDocument.

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

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

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

- A processing loop iterates through the objects in the array of line items and calculates sub-totals and sales tax for each order item, adding the amounts to the order object.
- Finally, the method returns true and the Mediator continues with the normal course of update processing (using the modified update plan).

Note: See "Update Override Programming Patterns" on page 9-14 for some other common programming patterns.

Invoking Data Service Procedures from an UpdateOverride

Listing 9-3 shows an example of an update override class that invokes a data service procedure. Since UpdateOverrides are invoked locally, within the DSP server, the sample uses the typed Mediator API. As shown in Listing 9-3, several Web services operations (to create, delete, and modify a customers address) have been registered with a Data Service.

Listing 9-3 Invoking a Procedure from an UpdateOverride

```
public class CustomerAddressUpdate implements UpdateOverride {
   public boolean performChange(DataGraph graph) {
     bool status = true;  // assume the best
     ChangeSummary changeSum = datagraph.getChangeSummary();
     // If no changes, do nothing.
     if (changeSum.getChangedDataObjects().size()==0) {
       return true;
     }
// Get the DataGraph's root DataObject and cast to customer object to
```

```
// enable getting DataGraph constituents
     CUSTOMERDocument custDoc = (CUSTOMERDocument) graph.getRootObject();
       ADDRESS[] addr = custDoc.ADDRESS().getADDRESSArray();
        int i;
        try {
          CUSTOMER custDS = CUSTOMER.getInstance(
               new InitialContext(), "RTLApp" );
// For each address in the Customer's address array, call the Web Service's
// update, delete, or create procedure as appropriate
         for( i = 0; i < addr.length; i++ ) {
              if ( changeSum.isModified( addr[ i ] ) ) {
                    custDS.invokeProcedure("modifyCustomerAddress",
                    new Object [] {addr[ i ]} );
                }
              else if ( changeSum.isDeleted( addr[ i ] ) ) {
                    custDS.invokeProcedure("deleteCustomerAddress",
                    new Object [] {addr[ i ]});
                }
              else if ( changeSum.isCreated( addr[ i ] ) ) {
                    custDS.invokeProcedure("createCustomerAddress",
                    new Object [] {addr[ i ]} );
                }
                else {
                    // throw an exception for IllegalState
                    }
                }
            } // end for
        }
        catch( Exception ex ) {
            System.err.println( ex.printStackTrace() );
            throw ex;
        }
        return status;
    }
}
```

The example in Listing 9-3 is for a Web service running locally on the WebLogic Server instance, so it does not include basic setup code to obtain context and location. (If the Web service is not local to the WebLogic Server instance, your code must obtain an InitialContext and providing appropriate location and security properties. See "Obtaining a WebLogic JNDI Context for Data Services Platform" on page 3-7 for more information about InitialContext.)

Listing 9-4 shows an update override alters the update plan in order to enforce referential integrity by removing product information from the middle of a list and adds it back at the end.

Listing 9-4 Update Override Example That Enforces Referential Integrity

```
        // delete order, item, product, due to RI between ITEM and Product
        // product has to be deleted after items
    public boolean performChange(DataGraph graph)
    {
    DataServiceMediatorContext context =
DataServiceMediatorContext.currentContext();
       UpdatePlan up =context.getCurrentUpdatePlan( graph, false );
       Collection dsCollection = up.getDataServiceList();
       DataServiceToUpdate ds2u = null;
       for (Iterator it=dsCollection.iterator();it.hasNext();)
       {
           ds2u = (DataServiceToUpdate)it.next();
           if
(ds2u.getDataServiceName().compareTo("ld:DataServices/PRODUCT.ds") == 0 ) {
// remove product from the mid of list and add it back at the end
                    up.removeContainedDataService( ds2u.getDataGraph() );
                    up.addDataService(ds2u.getDataGraph(), ds2u );
           };
        }
       context.executeUpdatePlan( up );
       return false;
    }
}
```

Testing Submit Results

Data service updates should always be tested to ensure that changes occur as expected. You can test submits using the Test View in BEA WebLogic Workshop.

The results in Test View depend on the type of changes being made, specifically, whether you are testing read and navigate functions or DSP procedures. For functions, the submit() returns the data.

For procedures, the Test View displays:

```
"Side effect function executed successfully."
```

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 the UpdateOverride Class" on page 9-6 for information about override classes.

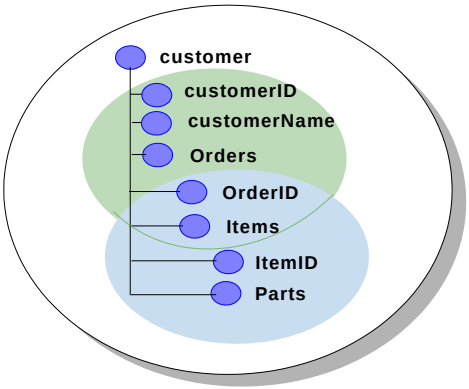
Update Override Context

Although an update override class can programmatically access several update framework artifacts, including the update plan, decomposition map, and the tree of modified data objects, the content available at any time depends on the data service context, as follows:

- **Top-level logical data service object**. The update override class has access to the entire tree of changed data objects.
- **Any lower-level or physical data service**. Only the objects in the change tree bound to the data service are available, along with the contents of the immediate container object—the performChange( ) method cannot access objects at any layer above it.

Figure 9-2 illustrates the context visibility within an update override.

Figure 9-2 Context Visibility in Update Override

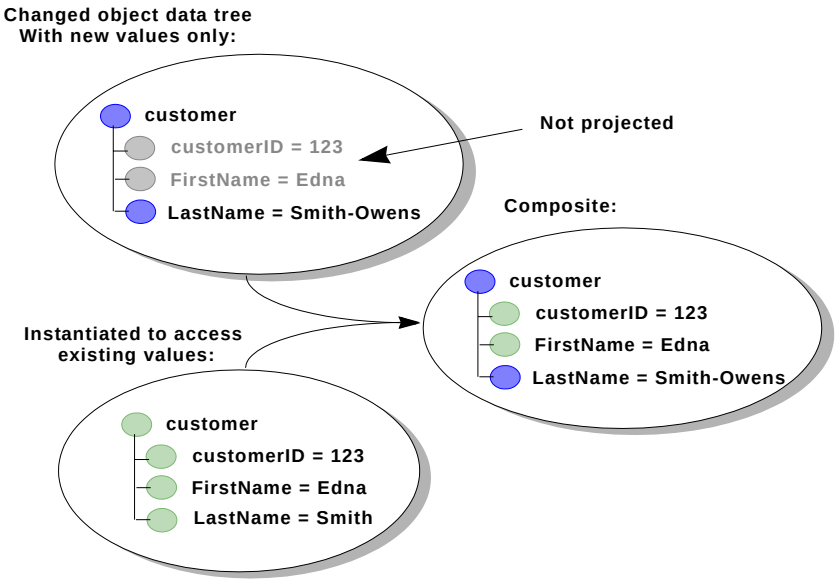


Update Overrides and Physical Data Services

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.
- A modified SDO that is passed to the physical level update override can see only those data object properties projected in the higher level data service. (See Figure 9-3.) To access the unprojected values as well, the update override must re-instantiate the data object.

Figure 9-3 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 raises an error and cancels the update request.

For physical non-relational data services, your performChange( ) method must:

- Provide an 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 data 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 Override Programming Patterns

In an update override, you can modify the server-side update process as much or as little as you like, at any step of the way, to accomplish your goal. This section provides some code samples that illustrate common update override programming patterns, including:

- Overriding the Entire Decomposition and Update Process
- Augmenting Data Object Content
- Customizing an Update Plan
- Executing an Update Plan
- Retrieving the Container of the Current Data Object
- Invoking Other Data Service Functions and Procedures
- Capturing Runtime Data about Overrides in the Server Log
- Default Optimistic Locking Policy: What it Means, How to Change

Remember that an Update Override class is simply a Java class that implements the UpdateOverride interface. You can give the class any valid Java filename, but should use a meaningful name for common-sense reasons. After writing the class, you must register it with the data service, by setting the name of the class in the data service's Update Override Property field.

The class must include an implementation of the performChange( ) method; it is inside this method that you provide all custom code required for the programming task at hand. The performChange( ) method returns a boolean value that either continues or aborts processing by the Mediator, as discussed in "How It Works: The Decomposition Process" on page 2-16. The level of customization that you provide in your performChange( ) method determines whether you should return true or false, as noted in each of the sections below.

Overriding the Entire Decomposition and Update Process

To customize the entire decomposition and update process, the performChange( ) method can implement the following types of routines:

- Instantiating lower level data objects and submit them for update.

- Calling a Web service passing the appropriate data.
- Using JDBC to execute SQL statements.

If your performChange( ) method does take over all processing, it should return false so that the Mediator does not proceed with automated decomposition.

Augmenting Data Object Content

The performChange( ) method can include code to inspect changed data object values and raise DataServiceException to signal errors, rolling back the transaction in such cases.

Return true to have the Mediator proceed with update propagation using the objects as changed.

Accessing the Data Service Mediator Context

To access the change plan and decomposition map for an update, you first must get the data service's Mediator 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, you can access the decomposition map as follows:

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

Once you have a decomposition map, you can use its toString() method to obtain the string rendering of the XML that map, as shown in Listing 9-5. (Note that although you can access the default decomposition map, you should not modify it.)

In addition to accessing the decomposition map, you can access the update plan 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.

Listing 9-5 Decomposition Map Example as XML String Fragment

```
<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>
          ...
    </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:

- Setting values on decomposed data objects.
- Adding, removing, or rearranging data objects in the update tree.
- Passing the modified update plan executeUpdatePlan() method, as in:

```
DataServiceMediatorContext.executeUpdatePlan()
```

After executing the update plan, the performChange( ) method 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 programmatically walk the update plan to view its contents by using your own method, similar to the navigateUpdatePlan(). As shown in Listing 9-6, navigateUpdatePlan() method takes a Collection object and uses an iterator to recursively walk the plan.

Listing 9-6 Walking an Update Plan

```
public boolean performChange(DataGraph datagraph){

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

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, Listing 9-7 illustrates an update plan in which item 1001 will be deleted from Order 100, and then the Order is deleted.

Listing 9-7 Example of Deleting a Line Item and Then Its Container

```
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, the contained SDOToUpdates routines are visited and processed. So the PO_ITEMS is deleted first and then PO_CUSTOMERS is deleted.

If the contents of the Update Plan are changed the new plan can then be executed. 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 Mediator to propagate changes 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.

Invoking Other Data Service Functions and Procedures

Other data services may be accessed and updated from an update override. The data service the 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.

Capturing Runtime Data about Overrides in the Server Log

DSP uses the underlying WebLogic Server for logging. WebLogic logging is based on the JDK 1.4 logging APIs (available in the java.util.logging package). You can contribute to the log (from an update override) by acquiring a DataServiceMediatorContext instance, and then 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 specific log 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 values listed in Table 9-4, from lowest to highest impact:

Table 9-4 WebLogic Server Log Level Definitions

Level	Description
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.

Level	Description
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:

Table 9-5 DSP Log Levels

Level	Information provided for...	Information captured
Notice or summary	Each submit from a client	• Fully qualified data service name • Invocation time • Total execution time • Invocation by user/group
Information or Detail	Each submit on a data service at any level	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
	Each update override invocation	• Name of data service being overridden • Number of times called • Total execution time

Listing 9-8 shows a sample log entry.

Listing 9-8 Sample Log Entry

```
<Nov 4, 2004 11:50:10 AM PST> <Notice> <LiquidData> <000000> <Demo - begin
client sumbitted 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
clientsumbitted ds: ld:DataServices/Customer.ds Overall execution time:
381>
```

Default Optimistic Locking Policy: What it Means, How to Change

Locking mechanisms are used in numerous types of multi-user systems for concurrency control—to ensure that data is consistent, across transactions and regardless of the number of users acting on the system at the same time. Optimistic locking mechanisms are so-called because they typically only lock data at the time it is being updated (written to), not when it is being read.

DSP uses optimistic locking as its concurrency control policy, locking data only when updates are being attempted. When DSP receives submitted data graph, it compares the values of the data used to instantiate the original data objects with the original values in the data graph to ensure that the data was not changed by another user process during the time the data objects were being modified by a client application.

The Mediator compares fields from the original and the source; by default, Projected is used as the point of comparison (see Table 9-6).

You can specify the fields 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. Table 9-6 describes the options.

Table 9-6 Optimistic Locking Update Policy Options

Optimistic Locking Update Policy	Effect
Projected	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. 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 to be verified.
Update	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: If DSP cannot read data from a database table because another application has a lock on the table, queries issued by DSP are queued until the application releases the lock. You can prevent this by setting transaction isolation (on your WebLogic Server's JDBC connection pool) to read uncommitted. See "Setting the Transaction Isolation Level" in the *Administration Guide* for details on how to set the transaction isolation level.

CHAPTER 10

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:

- Using Catalog Services to Obtain Data Services' Metadata
- Filtering, Sorting, and Fine-tuning Query Results
- Handling Large Result Sets with Streaming APIs
- Providing Role-based Access to DSP Relational Sources

Using Catalog Services to Obtain Data Services' Metadata

BEA AquaLogic Data Services Platform (DSP) maintains metadata about all data services through a system catalog-type data service, known as *Catalog Services*. Catalog Services are available to client application developers to use in the same way they use any other data service in DSP.

Catalog services provide a convenient way for client-application developers to programmatically obtain information about the data services that are running on the server. The primary benefit of Catalog Services to developers is that they can create dynamic applications based on the metadata underlying the data service applications that have been deployed. Enterprise, third-party, and other developers who want to build dynamic, metadata driven query-by-form (QBF) applications can leverage DSP's Catalog Services to do just that. In addition, Catalog Services enables interoperability with other metadata repositories.

By querying Catalog Services, developers can obtain all the information they need about data services. For example, you can obtain information about:

- Applications
- Folders
- DataServices
- DataServiceRefs
- Functions
- Relationships
- Schemas
- SchemaRefs

To develop a metadata-driven application, developers can use the client Mediator API and invoke the Catalog Service's methods (see Listing 10-2) as needed populate the page they present to users of their application, for example.

Since the Catalog Services are data services, just as with any other data service you can also view these data services in three other ways, specifically through the:

- DSP Console
- DSP Palette
- Data Service controls

However, given the typical use case for the catalog services—metadata driven QBF applications—it is far more likely that application developers will invoke Catalog Services methods by using the Mediator API.

Generally speaking, to create a QBF application your code can leverage Catalog Services as follows:

1) Call Folder.getFolder()

2) Select a data service

3) Call DataService.getDataServiceById(dsId)

4) [optional] call Schema.getSchemaByDataServiceRef(ds.getRef( ))

5) Select a function from the selected dataservice

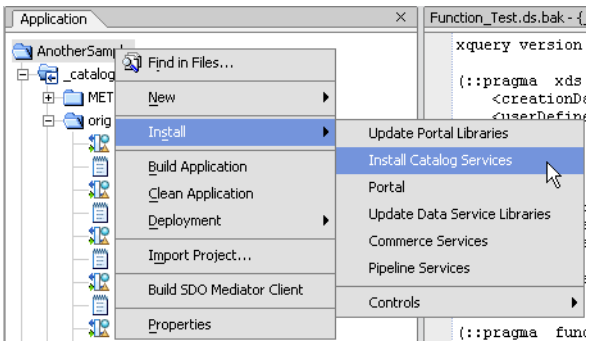
6) Provide a form to enter the arguments. The more complex the arguments, the more complex the code you must write.

Obtain a schema for parameters using the Catalog I think you can get the schema for parameters from Catalog Services – see Test View code on how to generate a 'template'

Installing Catalog Services

The ability to build Catalog Services within any DSP-enabled application is available by default when you install the product. DSP Catalog Services are installed easily on a per-application basis, by selecting Install Catalog Services from the WebLogic Workshop main menu.

Figure 10-1 Installing Catalog Services



Installing the Catalog Services creates a `_metadata.jar` file in the application's Library folder, and creates the various CLASS files that provide the typed accessors (see Table 10-2).

After installing Catalog Services, you will have access to all application metadata. For any DSP-enabled application that you want to leverage in this way, simply install the Catalog Services into the application.

Table 10-2 Catalog Services Accessor Methods

Data Service Name	Return Type (Java Class)	Accessor
DataService	DataService	getDataServiceByRef(DataServiceRef)
DataService	DataService	getDataServiceById(string)
DataService	DataServiceRef	getDataServiceDependencyRefs(DataService)
DataService	DataServiceRef	getDataServiceDependentRefs(DataService)
DataService	Function	getFunctionsByDataService(DataService)

Data Service Name	Return Type (Java Class)	Accessor
DataService	Relationship	getRelationshipsByDataService(DataService)
DataService	SchemaRef	getSchemaRefsByDataService(DataService)
DataServiceRef	DataServiceRef	getDataServiceRefs()
Folder	Folder	getFolder(String)
Function	Function	getFunctionById(FunctionId)
Function	DataService	getDataServiceByFunction(Function)
Function	Function	getFunctionDependenciesByFunction(Function)
Function	Function	getFunctionDepdendentsByFunction(Function)
Function	Relationship	getRelationshipsByFunction(Function)
Function	SchemaRef	getSchemaRefsByFunction(Function)
Relationship	Relationship	getRelationshipsByDataService(DataService)
Relationship	Relationship	getRelationshipsByDataServiceId(String)
Relationship	Relationship	getRelationshipById(String)
Schema	Schema	getSchemaById (String)
Schema	SchemaRef	getSchemaDependencyRefsBySchema (Schema)
SchemaRef	SchemaRef	getSchemaDependencyRefsBySchemaRef (SchemaRef)
SchemaRef	Schema	getSchemaByRef(SchemaRef)

In addition to the methods shown in Table 10-2, you will also see several extraneous methods that define relationships among data services. However, the methods shown in Table 10-2 are the only methods you need to develop a metadata-driven client application.

Creating a Query-by-Form (QBF) Application Using Catalog Services

You can create a Query-by-Form (QBF) application using DSP's Catalog Services and the Mediator APIs. Your application can leverage the Catalog Service in the same way you might leverage any data service using the Mediator APIs.

Note: For more information about using the Mediator API, see Chapter 3, "Accessing Data Services from Java Clients."

Filtering, Sorting, and Fine-tuning Query 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.

Only objects in the function return set that meet the condition are returned to the client. (The evaluation occurs at the server, so objects that are filtered are not passed over the network. Often, objects that are filtered out 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 apply multiple filter conditions using AND and OR operators. Other operators that be applied to filter conditions are listed in Table 10-3.

Table 10-3 Filter Operators

Operator	Usage note or example
LESS_THAN	Can also use "<". For example: myFilter.addFilter("CUST/CUST_ORDER/ORDER", "CUST/CUST_ORDER/ORDER/ORDER_AMOUNT", ">", "1000"); myFilter.addFilter("CUST/CUST_ORDER/ORDER", "CUST/CUST_ORDER/ORDER/ORDER_AMOUNT", FilterXQuery.GREATER_THAN, "1000");
GREATER_THAN	Can also use ">".
LESS_THAN_EQUAL	Can also use "<=".
GREATER_THAN_EQUAL	Can also use ">=".
EQUAL	Can also use "=".
NOT_EQUAL	Can also use "!=".
matches	Tests for string equality.
sql-like	Tests whether a string contains a specified pattern.
OR	Compound operator that can apply to more than one filter.
NOT	Compound operator that can apply to more than one filter.
AND	Compound operator that can apply to more than one filter.

Note: Filter API Javadoc, as well as other Data Services Platform APIs, is described at "DSP Mediator API Javadoc" on page 1-13.

Using Filters

Filtering capabilities are available to Mediator and Data Service control client applications. You use filter conditions to specify the 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 condition to the filter object using the addFilter() method. With this method 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 several signatures with different parameters, including the following:

   ```
   public void addFilter(java.lang.String appliesTo,
                         java.lang.String field,
                         java.lang.String operator,
                         java.lang.String value,
                          java.lang.Boolean everyChild)
   ```

   This version of 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`. See Table 10-3, “Filter Operators,” on page 10-6 for information about 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");
```

3. 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);
```

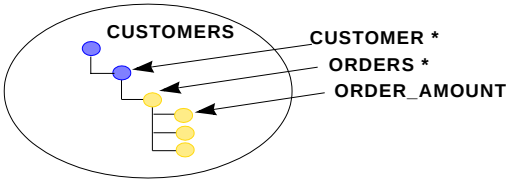
4. Invoke the data service function. (For more information on invoking data service functions, see Chapter 3, "Accessing Data Services from Java Clients.")

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 10-1. 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 10-1 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. Listing 10-1 uses the AND operator to apply a combination of filters to a result set, given a data service instance customerDS.

Listing 10-1 Example of Combining Filters by Using Logical Operators

```
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

Another type of filter you can use in client application code is an ordering condition—you specify the order (descending, ascending) in which results should be returned from the data service. The method (addOrderBy(), in the FilterXQuery class), takes a property name as the criterion upon which the ascending or descending decision is based. Listing 10-2 provides an example of creating a filter that will return customer profiles in ascending order, based on the date each person became a customer.

Listing 10-2 Example of Applying an Ordering Filter

```
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.)

Listing 10-3 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`.

Listing 10-3 Example of Applying a Filter that Truncates (Limits) Results

```
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);
```

Using Ad Hoc Queries to Fine-tune Results from the Client

An ad hoc query is an XQuery function that is not defined as part of a data service, but is instead defined in the context of a client application. Ad hoc queries are typically used in client applications to invoke data service functions and refine the results in some way. You can use an ad hoc query to execute any valid XQuery expression against a data service. The expression can target the actual data sources that underlie the data service, or can use the functions and procedures hosted by the data service.

To execute an XQuery expression, use the PreparedExpression interface, available in the Mediator API. Similar to JDBC's PreparedStatement interface, the PreparedExpression interface takes the XQuery expression as a string in its constructor, along with the JNDI server context and application name. After constructing the prepared expression object in this way, you can call the executeQuery() method on it. If the ad hoc query invokes data service functions or procedures, the data service's

namespace must be imported into query string before you can reference the methods in your ad hoc query. Listing 10-4 shows a complete example; the code returns the results of a data service function named getCustomers(), which is in the namespace:

```
ld:DataServices/RTLServices/Customer
```

Listing 10-4 Invoking Data Service Functions using an Ad Hoc Query

```
String queryStr =
      "declare namespace ns0=\"ld: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 data type. Once you have the XMLObject, you can downcast to the data type of the deployed XML schema. Since XMLObject has only a single root type, if the data service function returns an array, your ad hoc query should include a root element as a container for the array.

For example, the ad hoc query shown in Listing 10-4 specifies a <Results> container object to hold the array of CUSTOMER_PROFILE elements that will be returned by the getCustomer() data service function.

Security policies defined for a data service apply to the data service calls in an ad hoc query as well. If an ad hoc query uses secured resources, the appropriate credentials must be passed when creating the JNDI initial context. (For more information, see "Obtaining a WebLogic JNDI Context for Data Services Platform" on page 3-7.)

As with the PreparedStatement interface of JDBC, the PreparedExpression interface supports dynamically binding variables in ad hoc query expressions. PreparedExpression provides several methods (bind*Type*() methods; see Table 10-4), for binding values of various data types.

Table 10-4 PreparedExpression Methods for Bind Variables

To bind data type of...	Use bind method...
Binary	bindBinary(javax.xml.namespace.QName qname, byte[] abyte0)
BinaryXML	bindBinaryXML(javax.xml.namespace.QName qname, byte[] abyte0)
Boolean	bindBoolean(javax.xml.namespace.QName qname, boolean flag)
Byte	bindByte(javax.xml.namespace.QName qname, byte byte0)
Date	bindDate(javax.xml.namespace.QName qname, java.sql.Date date)
Calendar	bindDateTime(javax.xml.namespace.QName qname, java.util.Calendar calendar)
DateTime	bindDateTime(javax.xml.namespace.QName qname, java.util.Date date)
DateTime	bindDateTime(javax.xml.namespace.QName qname, java.sql.Timestamp timestamp)
BigDecimal	bindDecimal(javax.xml.namespace.QName qname, java.math.BigDecimal bigdecimal)
double	bindDouble(javax.xml.namespace.QName qname, double d)
Element	bindElement(javax.xml.namespace.QName qname, org.w3c.dom.Element element)
Object	bindElement(javax.xml.namespace.QName qname, java.lang.String s)
float	bindFloat(javax.xml.namespace.QName qname, float f)
int	bindInt(javax.xml.namespace.QName qname, int i)
long	bindLong(javax.xml.namespace.QName qname, long l)
Object	bindObject(javax.xml.namespace.QName qname, java.lang.Object obj)
short	bindShort(javax.xml.namespace.QName qname, short word0)
String	bindString(javax.xml.namespace.QName qname, java.lang.String s)
Time	bindTime(javax.xml.namespace.QName qname, java.sql.Time time)
URI	bindURI(javax.xml.namespace.QName qname, java.net.URI uri)

To use the bind*Type* methods, pass the variable name as an XML qualified name (*QName*) along with its value; for example:

```
adHocQuery.bindInt(new QName("i"),94133);
```

Listing 10-5 shows an example of using a bindInt() method in the context of an ad hoc query.

Listing 10-5 Binding a Variable to a QName (Qualified Name) for use in an Ad Hoc Query

```
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();
```

Note: For more information on QNames, see:

http://www.w3.org/TR/xmlschema-2/#QName

Listing 10-6 shows a complete ad hoc query example, using the PreparedExpression interface and QNames to pass values in bind methods.

Listing 10-6 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();
        }
}
```

Handling Large Result Sets with Streaming APIs

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 a 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";
StreamingPreparedExression 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 a Dynamic Mediator API" in Chapter 3, "Accessing Data Services from Java Clients."

Javadoc for the StreamingDataService interface and other Data Services Platform APIs is described at: "DSP Mediator API Javadoc" on page 1-13.

Listing 10-7 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 10-7 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 10-16).

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.

Providing Role-based Access to DSP Relational Sources

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 intercepter 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, cgDataSourc2, 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 10-8 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 10-8 Interceptor Class Example

```
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;
     }
}
```