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/docs25/index.html

Version: 2.5
Document Date: June 2005
Revised: September 2006

Copyright

Restricted Rights Legend

Trademarks and Service Marks

Contents

1. Data Programming Model and Update Framework

2. Introducing Data Services for Client Applications

3. Accessing Data Services from Java Clients

4. Enabling AquaLogic Data Services Applications for Web Service Clients

5. Using SQL to Access Data Services

6. Using Excel to Access Data Services

7. Accessing Data Services from WebLogic Workshop Applications

10.Using Workflow with AquaLogic Data Services Platform-Based Applications

11.Advanced Topics

CHAPTER 1

Data Programming Model and Update Framework

BEA AquaLogic Data Services Platform 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 through Mediator APIs or through Data Service controls. (See "Introducing Service Data Objects (SDO)" on page 2-8.)

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 2-9, 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 its update framework.

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 1-1, 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 1-1 Client Applications and Data Service Mediator Send and Receive Data Graphs

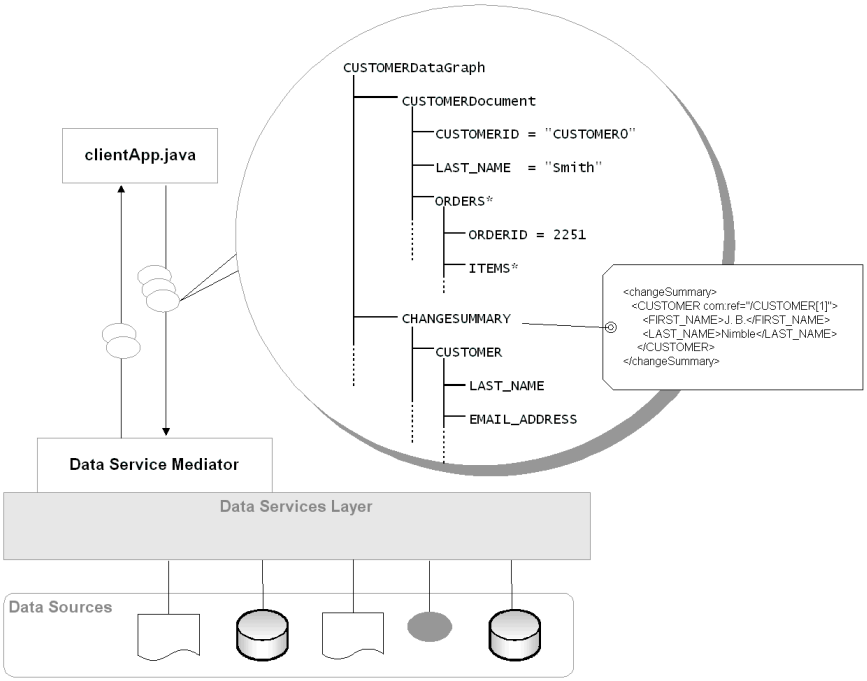


Table 1-2 summarizes the various SDO data programming artifacts and lists an example of each (as shown in Figure 1-1).

Table 1-2 Data Graph Artifacts and Examples

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

Table 1-2 Data Graph Artifacts and Examples

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: getCustomer( ) and setCustomer( ). For examples see Table 1-5, “Static (Typed) Data API Getters and Setters,” on page 1-7.
- **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 1-3 summarizes the advantages of each approach.

Table 1-3 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 "Developing Static Web Service Clients" on page 4-11 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 1-4), with getters and setters for each of the properties that comprise the complex type.

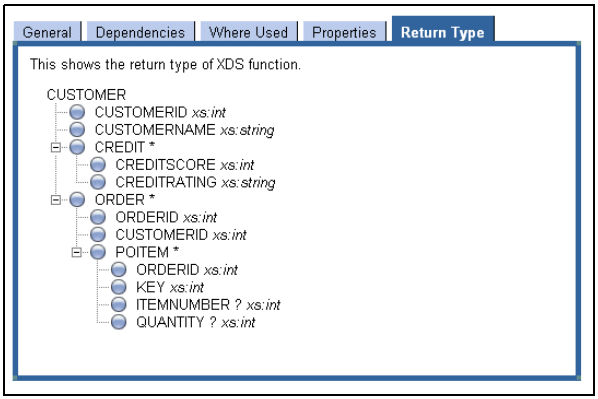
In addition, for properties that may have multiple occurrences, getters and setters are also generated for manipulating arrays and array elements. A multiple-occurring property is an XML schema element that has its maxOccurs attribute set to either unbounded or greater than one. In the DSP Console Metadata Browser, such elements are flagged with an asterisk—for example, ORDER* and POITEM* (see Figure 1-4) 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 1-4, generating typed client interfaces results in:

- CUSTOMER, CUSTOMERDocument, CREDIT, ORDER, and POITEM interfaces, each of which includes getters, setters, and factory classes (for instantiating static 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 1-4 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 10, "Using Workflow with AquaLogic Data Services Platform-Based 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 1-5 lists static data API getters and setters.

Table 1-5 Static (Typed) Data API Getters and Setters

Static Data API (Generated)	Description	Examples
`Type get`*`PropertyName`*`()`	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[] get`*`PropertyName`*`Array()`	For multiple occurrence elements, returns all *PropertyName* elements.	`getCREDITArray()`
`Type get`*`PropertyName`*`Array( int PropertyIndex)`	Returns the *PropertyName* child element at the specified index.	`getCREDITArray(int), setCREDITSCORE(int)`
`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()`

Table 1-5 Static (Typed) Data API Getters and Setters

Static Data API (Generated)	Description	Examples
`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()
`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 1-4, "CUSTOMER Return Type Displayed in DSP Console's Metadata Browser," on page 1-6). 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 1-6.

Table 1-6 XML Schema to Java Data Type Mapping

XML Schema Type	SDO Java Type	XML Schema Type	SDO Java Type
xs:anyType	commonj.sdo.DataObject	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
xs:duration	String	xs:normalizedString	String
xs:ENTITIES	String	xs:NOTATION	String

Table 1-6 XML Schema to Java Data Type Mapping

XML Schema Type	SDO Java Type	XML Schema Type	SDO Java Type
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 1-7 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 ("AquaLogic Data Services Platform Mediator API Javadoc" on page 2-13). 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 1-7 lists dynamic data API getters and setters.

Table 1-7 Dynamic (Untyped) 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)`

Table 1-7 Dynamic (Untyped) Data API Getters and Setters

Dynamic Data API	Description	Example
`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)`
`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 1-8

compares the XPath standard and SDO augmented notations to refer to the same element, the first ORDER child node under CUSTOMER (Table 1-8).

Table 1-8 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 3-26.

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 1-9) and the Property interface (see Table 1-10) 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 1-1.

Listing 1-1 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 1-9 lists other useful methods in the Type interface.

Table 1-9 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.
`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 1-10 lists the methods of the Property interface.

Table 1-10 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.

Table 1-10 Property Interface Methods

Method	Description
`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 in the *Building Queries and Data Views* Handling Updates Through Data Services chapter.

CHAPTER 2

Introducing Data Services for Client Applications

BEA AquaLogic Data Services Platform brings a service-oriented architecture (SOA) approach to data access. AquaLogic 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, AquaLogic Data Services Platform 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 AquaLogic Data Services Platform for client application developers. It includes the following topics:

Note: AquaLogic 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 AquaLogic Data Services Platform (AquaLogic Data Services Platform) 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 AquaLogic Data Services Platform, 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.

AquaLogic Data Services Platform 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 AquaLogic Data Services Platform. 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 AquaLogic Data Services Platform-aware client applications. It explains the various client access mechanisms that AquaLogic Data Services Platform 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 AquaLogic Data Services Platform update framework.

This guide provides information about how to leverage data services in your applications.

- For information about server-side aspects of creating and managing data services see the *Data Services Developer's Guide*.
- For information on administering data services, including metadata, cache, and security management see the AquaLogic Data Services Platform *Administration Guide*.

What is a Data Services 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 AquaLogic Data Services Platform, 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.

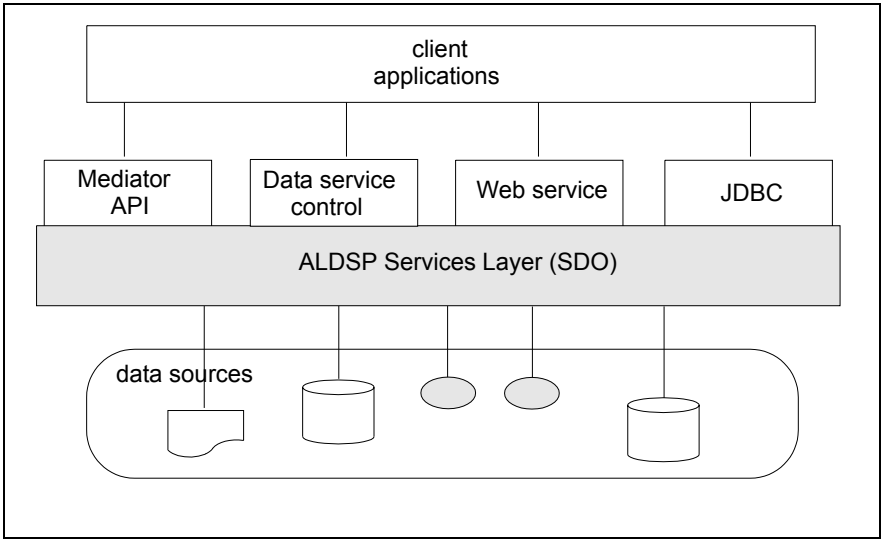
Data Your Way

An AquaLogic Data Services Platform 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. These can all be thought of as *client APIs*.

For client application developers needing to leverage such data assets, AquaLogic Data Services Platform supports multiple access methods (see Figure 2-1):

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

Figure 2-1 Accessing AquaLogic Data Services Platform Services



Whatever the client type, AquaLogic Data Services Platform 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 server-side, in WebLogic Workshop. Workshop is BEA's integrated development environment for building and deploying many types of applications: portals, Web services, and integration applications, for example.

What Is a Data Service?

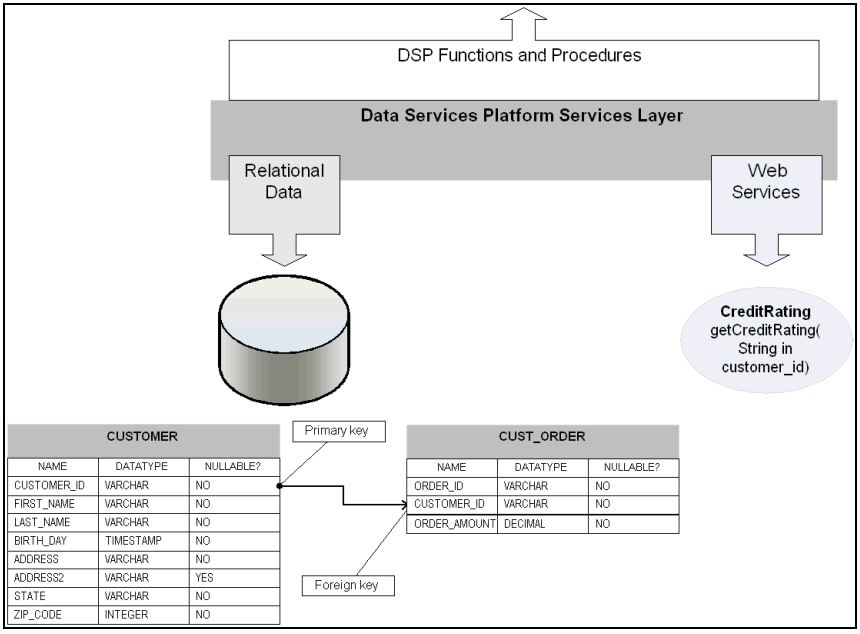
From a high-level perspective, a data service defines a distinct *business entity* such as a report that describes a customer and customer orders. The data service defines a unified view by aggregating data from any number of sources — relational database management systems (RDBMS), Web services, enterprise 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 AquaLogic 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 2-2 Data Services Layer Exposes Functions and Procedures to Client Application Developers



What is a AquaLogic Data Services Platform Client Application?

A AquaLogic Data Services Platform 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 AquaLogic Data Services Platform-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 AquaLogic Data Services Platform 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 AquaLogic Data Services Platform JDBC driver provides JDBC and ODBC clients, such as reporting tools, with SQL-based access to AquaLogic Data Services Platform data.

Regardless of the client type, AquaLogic Data Services Platform 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 AquaLogic Data Services Platform 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

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

Access policies for 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 client API models for accessing AquaLogic Data Services Platform services. The model chosen will depend on the access mechanism you decide to used. The access methods are:

- Data Mediator API

- Data Service control
- Web Services
- JDBC/ODBC

Each access method has its own advantages and use. Table 2-3 provides a description of each of these access methods and summarizes the advantages of the various models for accessing AquaLogic Data Services Platform services.

Table 2-3 AquaLogic 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 1, "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 AquaLogic Data Services Platform resources. For more information, see Chapter 7, "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 2-3 AquaLogic 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, "Enabling AquaLogic Data Services Applications for Web Service Clients."	• 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 AquaLogic Data Services Platform services using SQL queries. The AquaLogic Data Services Platform JDBC driver supports SQL-92. For more information, see Chapter 5, "Using SQL to Access Data Services."	• 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. AquaLogic Data Services Platform 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 1, "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. AquaLogic Data Services Platform provides a Data Service Mediator, a server-side component of AquaLogic Data Services Platform's XQuery processing engine that serves as the intermediary between data services and client applications or processes.

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

- **Relational data sources**. Database management systems (RDBMS) such as IBM, Oracle, Microsoft SQL Server, and any other SQL-92 compliant RDBMS are handled automatically.

  Note: There are times when it may be convenient or necessary to override the default update processing for a relational source. This is done through an update override class associated with a data service. See the topic "Creating Update Overrides for Relational

Data Sources" in the chapter Handling Updates Through Data Services in the *Building Queries and Data Views*.

- **Non-relational data sources**. This includes such non-relational sources 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 the topic "Developing an UpdateOverride Class" in the chapter Handling Updates Through Data Services in the *Building Queries and Data Views*.

Typical Client Application Development Process

Developing a AquaLogic Data Services Platform-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 AquaLogic Data Services Platform 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 2-3, "AquaLogic Data Services Platform Access Models," on page 2-7 describes the advantages of the different access mechanisms.) The approach you choose also depends on precisely how the data service has been deployed.

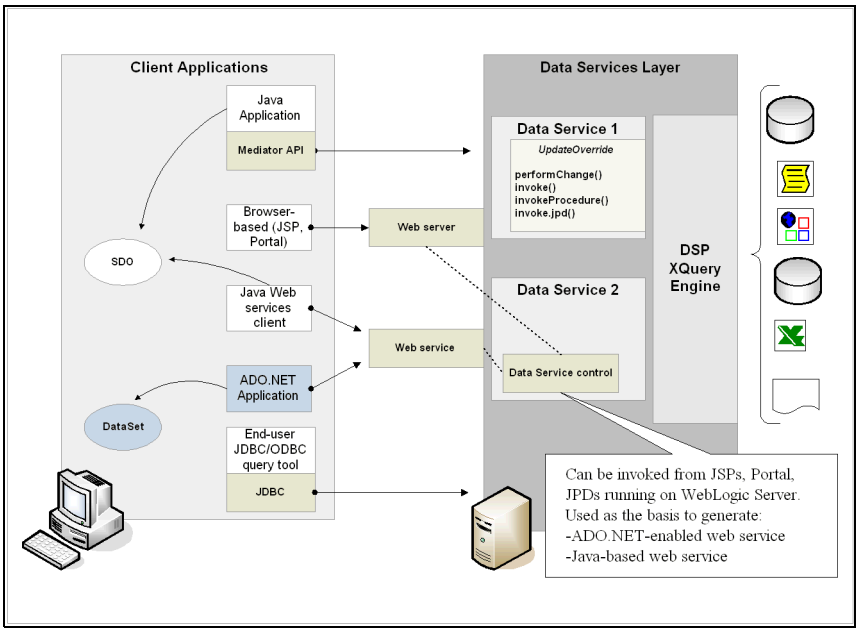
   For example, if the data service is hosted as a Web service, you can develop a Web service client application using Java in conjunction with the service's WSDL file.

   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 2-5, "AquaLogic Data Services Platform Java Archive Files," on page 2-12). To use the typed data service and SDO interfaces, obtain the AquaLogic Data Services Platform application's generated Mediator client JAR file from your AquaLogic Data Services Platform 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 2-4 provides a conceptual overview of the various approaches, highlighting some of the relationships among various sub-systems and components. See the *Data Services Developer's Guide* for complete information about such server-side data service development.

Figure 2-4 Types of AquaLogic 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 "AquaLogic Data Services Platform Mediator API Javadoc" on page 2-13).

Runtime Client JAR Files

AquaLogic Data Services Platform APIs are contained in the packages listed in Table 2-5.

Table 2-5 AquaLogic 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 prefixed by the name of the AquaLogic Data Services Platform application from which the static interfaces are generated. Such an application-specific JAR file is not required if the only interface to AquaLogic Data Services Platform routines is through an untyped interface using generic SDO.	(Provided by your AquaLogic 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 AquaLogic 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.

AquaLogic Data Services Platform Mediator API Javadoc

The AquaLogic Data Services Platform Mediator API describes the routines needed by AquaLogic Data Services Platform client applications to invoke various AquaLogic Data Services Platform 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 AquaLogic Data Services Platform 2.5 and 2.1 Mediator API at:

`http://edocs.bea.com/aldsp/docs25/javadoc/index.html`

Client applications built on earlier versions of AquaLogic 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://edocs.bea.com/aldsp/docs25/javadoc-dsp201/index.html`

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

`http://edocs.bea.com/aldsp/docs25/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 AquaLogic Data Services Platform'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 AquaLogic Data Services Platform audit capabilities see AquaLogic Data Services Platform *Administration Guide*.

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 AquaLogic Data Services Platform Mediator API
- Generating a Static Mediator API JAR File
- Accessing Data Services Via WebLogic Server 9.2 Clients
- Step-by-Step: A Java Client Programming Example
- Examining a Java Client Application

Overview of the AquaLogic Data Services Platform Mediator API

The BEA AquaLogic Data Services Platform 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 11-8.
- **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 11-19.
- **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 11-14.

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 AquaLogic 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 2-5, "AquaLogic Data Services Platform Java Archive Files," on page 2-12.

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 dynamic 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:

- Use Mediator APIs, which encompass the SDO Update and SDO Meditator API listed in Table 3-2.
- Use 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. Data Service controls are discussed in Chapter 7, "Accessing Data Services from WebLogic Workshop 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	• Mediator API Javadoc for AquaLogic Data Services Platform versions 2.5 and 2.1 • Mediator API Javadoc for AquaLogic Data Services Platform version 2.0.1 Both these documents can be accessed from the AquaLogic Data Services Platform edocs home page: `http://edocs.bea.com/aldsp/docs25/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 AquaLogic 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 or a static 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 a parameters, as shown below:

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

  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
  ```

Table 3-3 describes the available arguments that can be used to generate a static client JAR mediator.

Table 3-3 Generating a Static Client JAR Mediator from a 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.

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 AquaLogic Data Services Platform" on page 3-8. 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 in that 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");
```

Obtaining a WebLogic JNDI Context for AquaLogic 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);
  ```

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.

Table 3-4 Method Signature of the 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.

For more information, see information on Data Services API Javadocs at "AquaLogic Data Services Platform Mediator API Javadoc" on page 2-13.

When using static mediator APIs, the distinction between invoking DSP functions and procedures 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:

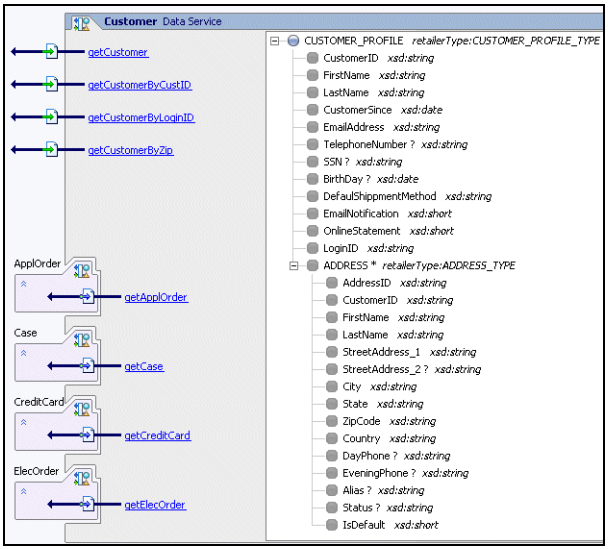
```
import customerapp.Customer;
```

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 Static Methods of 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 1-5 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 Table 3-6, below:

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 Mediator 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 dynamic SDO interface lets you use data services specified at runtime. It is described under the topic "Using a Dynamic Mediator API" on page 3-13.)

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:

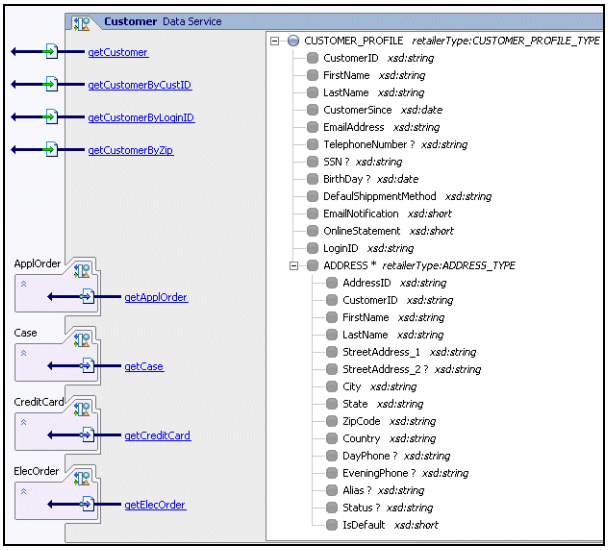
```
import customerapp.Customer;
```

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 1-5 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 dynamic SDO, DataObjects depend on the XML schema to determine:

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

The dynamic 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.

Dynamic SDOs are created using a createRootDataObject( ) method:

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

The code fragment in Listing 3-6 illustrates these familiar operations using dynamic 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 Dynamic 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 Dynamic 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 dynamic 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 Data Cache When Using the Mediator API

Data retrieved by data service functions can be cached for quick access. This is known as a *data caching*. (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 data cache by passing the GET_CURRENT_DATA attribute within a function call, as shown in Listing 3-7. GET_CURRENT_DATA returns a Boolean value. 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);
```

Client Management of the Data Cache

When invoking a DSP query you can more precisely control the behavior of the data cache by using the REFRESH_CACHE_EARLY attribute in conjunction with GET_CURRENT_DATA attribute.

Setting the GET_CURRENT_DATA Attribute

When the GET_CURRENT_DATA attribute is set to True:

- All data cache access is bypassed in favor of the physical data source. Function values are recalculated based on the underlying data and the cache is refreshed. If a call involves access to several cacheable functions, all will be refreshed with current data.
- The audit property:

  `evaluation/cache/data/forcedrefresh`

  indicates that a GET_CURRENT_DATA operation has been invoked.
- The REFRESH_CACHE_EARLY attribute property setting is ignored.

SETTING the REFRESH_CACHE_EARLY Attribute

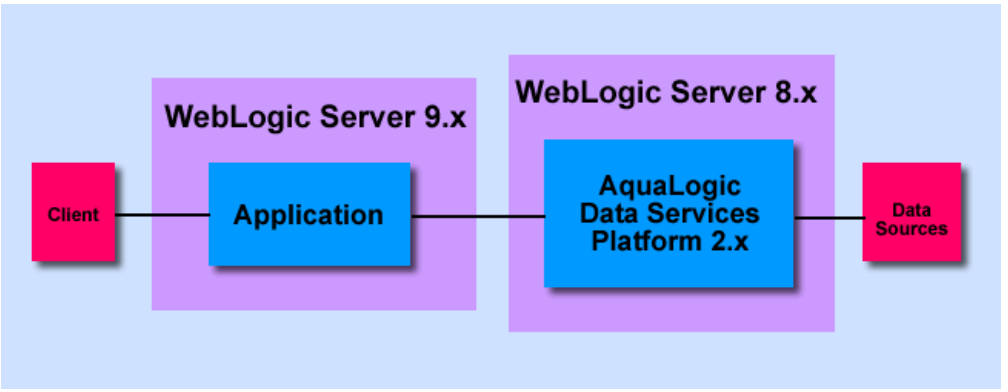
If the GET_CURRENT_DATA property is set to False or is not invoked, you can use the REFRESH_CACHE_EARLY to control whether cached data is used based on the remaining TTL (time-to-live) available for the function’s data cache.

The REFRESH_CACHE_EARLY attribute is of type integer. It is set by invoking the setIntegerAttribute( ) method. The setting of REFRESH_CACHE_EARLY to a particular value requires that a cached record must have at least *n* seconds of remaining TTL before it can be used. If the record is set to expire in less than *n* seconds, it will not be retrieved. Instead its value is recalculated based on the underlying data and the data cache associated with that function is refreshed. The same REFRESH_CACHE_EARLY value applies to all cache operations during a query evaluation.

Note: The supplied integer value of REFRESH_CACHE_EARLY should always be positive. Negative values are ignored.

Accessing Data Services Via WebLogic Server 9.2 Clients

This section describes how to make AquaLogic Data Services Platform 2.x and WebLogic Server 9.2 interoperable. Once interoperability is set up, a WebLogic Server 9.2 client can access AquaLogic Data Services Platform 2.*x* data services.



Notes: AquaLogic Data Services Platform 2.x runs under WebLogic Server 8.x.

Follow these steps to enable interoperability between a WebLogic Server 9.2 client and AquaLogic Data Services Platform 2.*x*:

Interoperability Steps

1. Put `${WLS_HOME}/liquiddata/lib/wls90interop.jar` in `PRE_CLASSPATH` on the AquaLogic Data Services Platform 2.*x* server by editing `setDomainEnv.{cmd|sh}`

   Note: After this change, all the non JDK 1.5 based clients of the AquaLogic Data Services Platform 2.*x* server will need to have `wls90interop.ear` in their class path ahead of all other classes.

2. Put `${WL_HOME}/liquiddata/lib/wls90interop.jar` in the classpath of WebLogic Workshop by modifying the value for the `-cp` key in `Workshop.cfg`

3. Build the application inside Workshop.

4. Build the client-side JAR using `${WL_HOME}/liquiddata/bin/ld_clientapi.xml` with the following arguments on the command line:

   ```
   -Dapproot=<application directory>
   ```

   ```
   -Dxbeanjarpath=${WEBLOGIC9x}/server/lib/xbean.jar
   ```

5. Copy `${WL_HOME}/liquiddata/lib/wlsdo90interop.jar`, `ld-client.jar` and `<appName>-ld-client.jar` into `WEB-INF/lib` or classpath on the 9.2 server side.

   Note: If you are using the WebLogic 9.0 release, you will need to upgrade the XMLBeans V2 shipped with WLS 9.0. You will need to use XMLBeans from WLS 9.1 or above.

Step-by-Step: A Java Client Programming Example

This section describes common Java client application programming tasks:

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 also Table 3-10).

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

Table 3-8 lists static and dynamic mediator API interfaces.

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 1-1, 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 3-11 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 shows the static and dynamic mediator APIs available to instantiate SDOs.

Table 3-9 Static and Dynamic Mediator APIs

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)custDoc[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 1-12 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( ).

Quantifying Return Types

Return types from data service functions can be quantified based on the semantics shown in Table 3-11.

Table 3-11 Quantifying Return Type Symbols and Their Definition

Quantifier Symbol	Definition	Comments
+	Same semantics as star (*)	Returns all.
?	Same semantics as unqualified except that there is additional built-in logic to handle the possibility of empty results. When the results are empty, the return value of the static mediator method will be either: • **Null.** Null is returned if the Java return type is non-primitive (such as typed subclasses of DataObject, BigDecimal, String, and so forth). • **Best value.** Best value is returned if the Java return value is a primitive (such as int, float, and so forth).	"Best value" is the same as is returned when RequestConfig prevents results from being sent, for instance in the OUTPUT_FILENAME case. For integer numeric types, best value is the corresponding MIN_VALUE; for floating point numeric types, best value is NaN (not a number); for Boolean, best value is False.

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

Table 3-12 provides examples showing how you can modify data object property values using either dynamic or static set( ) methods.

Table 3-12 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 the topic "Primary-Foreign Key Relationships Mapped Using a KeyPair" in the Handling Updates Through Data Services chapter of the *Building Queries and Data Views*.)

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

```
import com.bea.dsp.dsmediator.client.DataService;
import com.bea.dsp.dsmediator.client.DataServiceFactory;
import commonj.sdo.DataObject;

        DataService custDS =
DataServiceFactory.newDataService(getInitialContext(), "MyDSPApp",
"ld:MyDSPAppDataServices/CUSTOMER");
        DataObject[] custDocs = (DataObject[])custDS.invoke("CUSTOMER",
null);
        DataObject custDoc=custDocs[0];
        DataObject customer=(DataObject)custDoc.get("CUSTOMER");
        customer.set("LAST_NAME","Nimble");
        custDS.submit(custDoc);
```

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 dynamic 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 "AquaLogic Data Services Platform Mediator API Javadoc" on page 2-13.

See "Invoking Functions and DSP Procedures" on page 3-9 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

Enabling AquaLogic Data Services Applications for Web Service Clients

Web services provide an industry-standard way to develop SOA (service-oriented architecture) applications. Such services can be thought of as loosely coupled, distributed units of programming logic that can be re-configured easily to deliver new application functionality, both intra- and extra-enterprise.

Using Web services and BEA AquaLogic Data Services Platform allow your applications to better leverage enterprise data assets.

This chapter explains 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

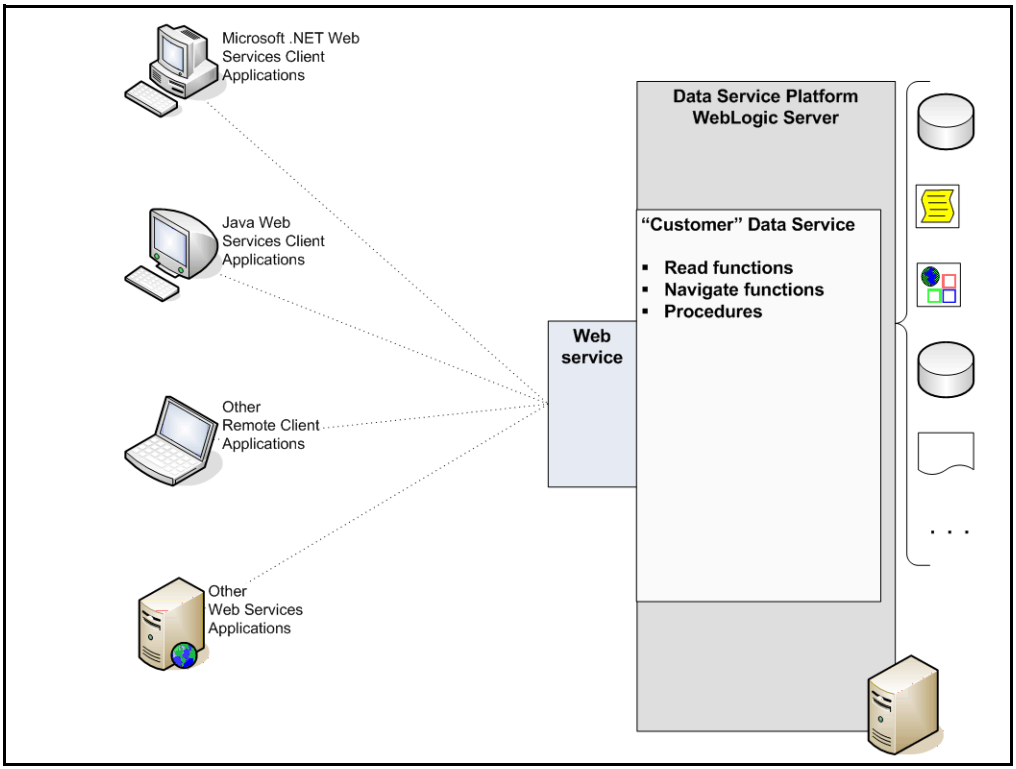
For more information about Web services, see:

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

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

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



Note: For information about ADO.NET-enabled Web services and client applications, see "Supporting ADO.NET Clients" on page 9-1.

Different Styles of Web Services Integration for DSP

Data services can be integrated with Web services in one of two general ways:

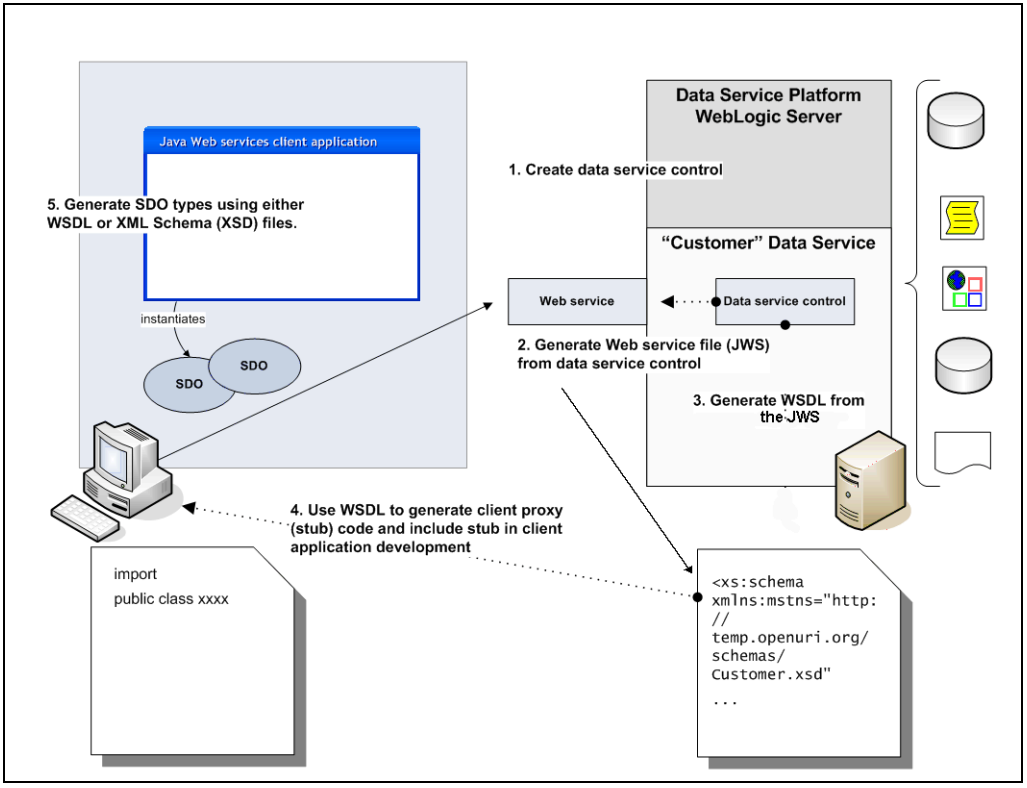
- **As a read-only** Web service. A standard Web service can be invoked from other Web services, or by .NET clients or any other type of client, Java and non-Java alike. On the server side, at runtime, the Web service simply passes the results obtained from the data service function back to the client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

Service Clients" on page 4-11), or dynamic SDOs (also known as *generic SDOs*). See "Developing Dynamic Web Service Clients" on page 4-17.

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

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 Data Service Java Clients Supported Through Web Services



Server-Side DSP-Enabled Web Service Development

DSP-enabled Web service development depends on whether you are working with read-only Web services or Web services which support read-write functionality.

Developing DSP-Enabled Read-Only Web Services

There are two ways to create Web services from data services. By:

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

Both approaches rely on Data Service controls as the component-based integration mechanism.

Adding a Data Service Control to a Web Service

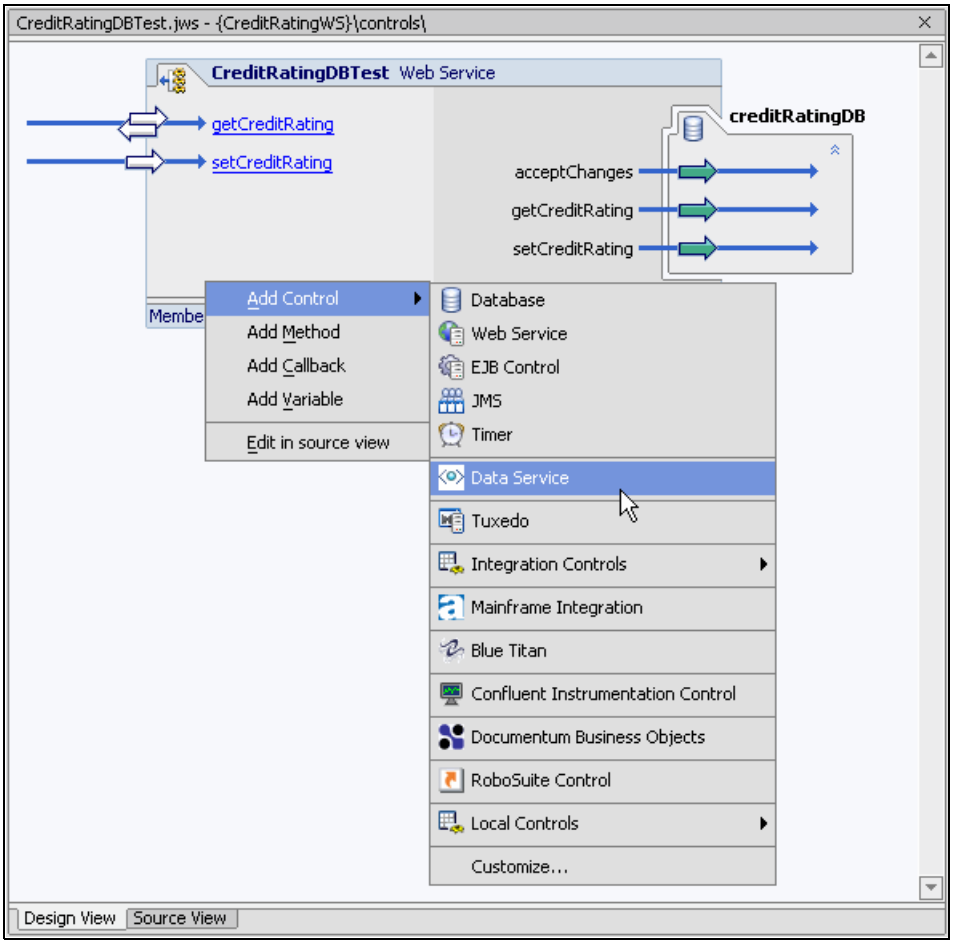
You can easily add one or more Data Service controls to a Web service using WebLogic Workshop. Firstcreate a folder for the controls inside the Web service's project folder, and then create the Data Service controls.

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 7-7 for more information on creating Data Service controls.)

Here are the steps involved:

1. In WebLogic Workshop, open an existing Web service file (JWS) by double-clicking on its name in the Application pane.
2. Click the Design View tab on the Web service to open the graphical representation of the Web service (as shown in Figure 4-3).

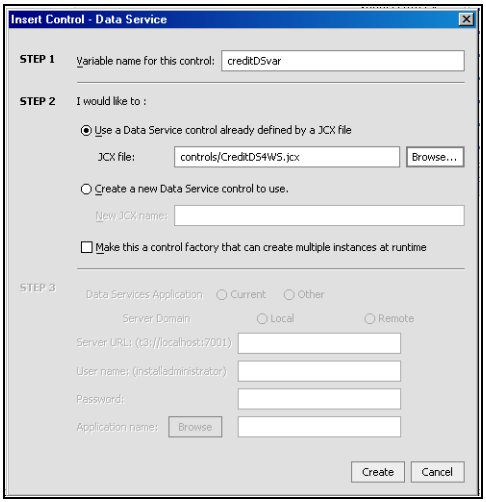
Figure 4-3 Adding a Data Service Control to a Web Service



3. Right-click and select Add Control → Data Service from the popup menu. The Insert Control – Data Service wizard launches (Figure 4-4).

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

Figure 4-4 Insert Control – Data Services Wizard



5. 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.)

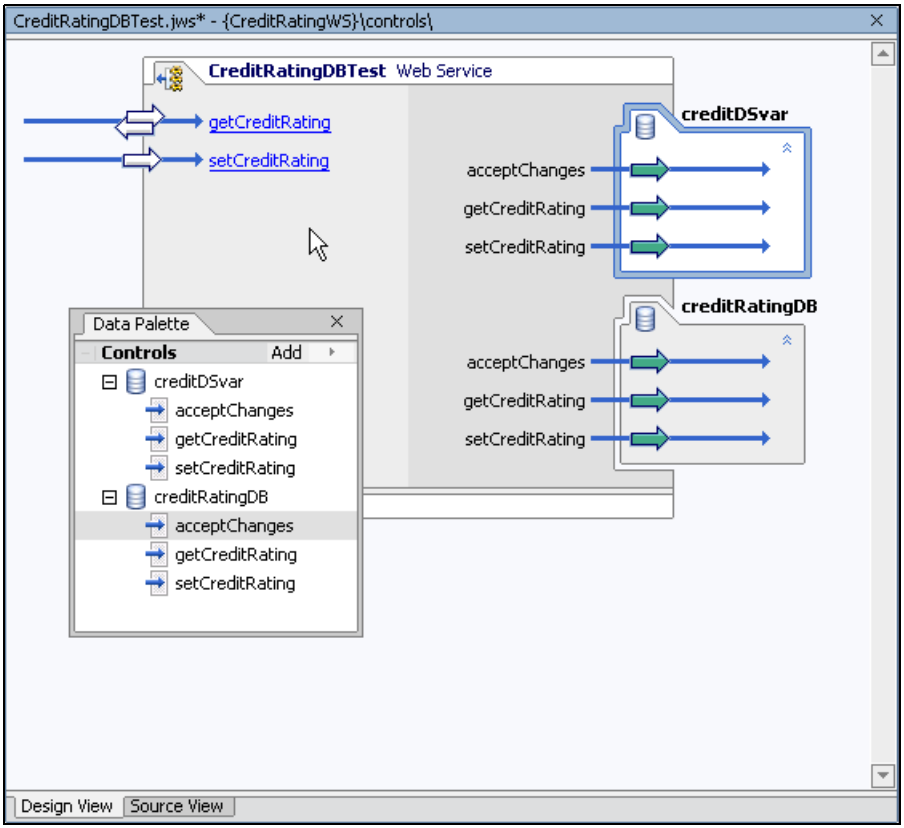
   Leave the checkbox labeled Make This a Control Factory unselected. (This checkbox would cause the Data Service control to be instantiated at runtime using the factory pattern, rather than as a singleton. To use the control in a Web service, it must be a singleton.)

6. STEP 3 is 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. The dialog provides for entry of a user name, password, server URL, and domain information associated with the remote Data Service control. This information is needed to complete the link between the Web service and the control.

7. Click the Create button on the Insert Control – Data Service dialog. The `LiquidDataControl.jar` file is copied into the Libraries directory of the application. 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 Data Service Control Functions to a Web Service



8. Select the variable's function or procedure listed in the Data Palette by clicking on the node, and then dragging the function onto the left side of the Web service in Design View.

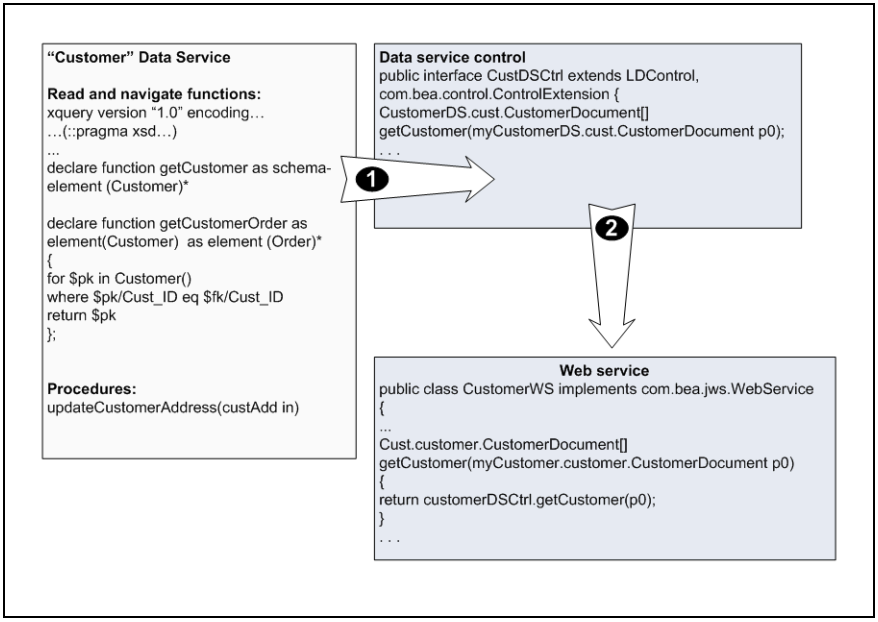
You can test your Web service as described in "Testing a Web Service in WebLogic Workshop" on page 4-9. After testing, you can deploy your Web service to a production WebLogic Server and use it as you would any other Web service. For information about developing Java-based Web service clients, see "Client-Side DSP-Enabled Web Service Development" on page 4-10.

Generating a Web Service from a Data Service Control

Another way to create a DSP-enabled Web service is by generating stateless Web services from Data Service controls. The generated Web services automatically include operations (method calls) for each of the functions and procedures that the Data Service control comprises.

Follow the instructions in this section to generate and test a stateless Web service. (These 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



Here are the steps involved:

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: 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".

2. Click on your Web service project to select it, then 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. If necessary, click the Design View tab 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.

Developing DSP-Enabled Read-Write Web Services

If your Web service must support submits from Java Web service clients, you first need to 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.

   For instance, if the original signature of the submit() method of the generated JWS looks appears as:

```
java.util.Properties[] submitCustomerProfile(CustomerProfileDocument doc);
```

   It should be modified to the following:

```
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); //customerData is the DSP
control
```

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

After you have created the WSDL file, provide it to your 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 4-10).

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.

2. Click the Test button to run the Web service and obtain your result.

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 WebLogic Server documentation.

Client-Side DSP-Enabled Web Service Development

Your client application uses either a static or a dynamic approach to Web services. Both approaches are discussed in this section. The following topics provide brief summaries of the appropriate uses of static and dynamic clients.

Static Web Service Clients

A static Web service client requires:

- A Web service client proxy that invokes Web service operations.
- The static SDO API (such as customer.getName( )) to read or modify data returned by the Web service.

DSP includes the necessary utilities (Java classes and Ant tasks) to generate the following classes for your static Web service client:

- SDO classes (such as CustomerDocument)
- Web service client proxy

  A typical static Web service client can use the following code to retrieve a customer's record:

  ```
  CUSTOMERDocument doc = wssoap.getCustomer("987654");
  ```

  Note: The wssoap class is an instance of the generated Web service client proxy class; the doc object is an instance of the generated Static SDO class CUSTOMERDocument.

Dynamic Web Service Clients

A dynamic Web service client requires:

- JAX-RPC API to dynamically invoke Web service operations
- Dynamic SDO API to read or modify data returned by the Web service

Neither the generated SDO classes, nor the Web service client proxy classes are needed for the Dynamic Web service client.

A typical dynamic Web service client can retrieve a customer's record with the following:

```
XmlObject param = XmlObject.Factory.parse( "<msg:getCustomer
xmlns:msg='http://www.openuri.org/'><CustomerID>987654</CustomerID></msg:getCu
stomer>");

DataObject doc = (DataObject)call.invoke(new Object[]{param});
```

Note: The call class is an instance of the Call interface of the JAX-RPC API. The doc object is an instance of the DataObject interface of the Dynamic SDO API.

Developing Static Web Service Clients

The following general steps are involved in developing a static Web service client for DSP:

- Generate the DSP Web services proxy that includes the SDO classes.
- Set up the Web service client environment.
- Develop the client.

Generating the DSP Web Service Proxy

You can generate an SDO Web service client using an Ant task (SDO Web Service Client Gen) or through a Java class (WSClientGen). Each approach is described in this section.

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 XSD files 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.

Environmental Settings

To generate the necessary 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"/>
```

   where *path/to* is replaced with the location of your JAR files.

   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 4-7 summarizes the attributes used by the sdogen Ant task.

Table 4-7 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
`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	

Table 4-7 Attributes Available for DSP's SDO Generation (sdogen) Ant Task

Attribute	Description	Required?	Default Value
`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. 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 a WSDL or XML schema definition (XSD) files in the schemas directory and create a JAR named `Schemas.jar`, your Ant script would need to include the following:

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

where *path/to* represents the actual location of your JAR files.

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. See Table 4-8 for other command-line options for the SDOGen utility.

Table 4-8 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.	
`-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.	

Table 4-8 Command-line Options for the Java SDO Class Generation Utility

Option	Description	Default Value
`-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
```

The XMLSchema can be:

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

SDOGen Usage Examples

Here are some examples showing use of the SDOGen Web service client generation utility:

The following are examples of using SDOGen with various options (Table 4-8) 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 you can use:

```
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, use:

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

  The `-dl` option permits downloading.

- To create a file named `xmltype.jar` using an XML schema definition (a XSD file) located in the following directory on your local machine:

  ```
  \myApps\xsd_dir
  ```

  You can use:

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

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

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

How To Set Up a Web Service Client Environment for DSP

The following instructions enable you to set up your Web service client environment for DSP.

After generating the SDO Web service client classes in a JAR file (`SDOClient.jar`), set up the classpath for the Web service client using the following JAR files in the following order:

- `wlsdo.jar`
- `webserviceclient.jar`
- `xbean.jar`
- `wlxbean.jar`
- `xqrl.jar`
- `SDOClient.jar` (the generated SDO Web service client JAR file)

Caution: The order of files shown above must be maintained.

Steps Involved in Developing Your Web Service Client

If you are not already familiar with the concept of using a Web service client proxy or JAX-RPC API to invoke Web services, see the following document:

`http://edocs/wls/docs81/webserv/client.html#1069703`

Then, in developing your Web service client, follow these steps:

1. Invoke the Web service method (e.g. getCustomer) to get the strongly typed root SDO data object (e.g. CUSTOMERDocument). At this point, a SDO datagraph has already been created and attached to the root data object (i.e. CUSTOMERDocument). Change tracking is also turned on by default.
2. Use the Static SDO API to read the data (e.g. getCustomerName). Alternative you can use the static API to modify the data (e.g. setCustomerName("J D")).

   Alternatively, you can also use the dynamic SDO API to read or modify the data.

   See Chapter 1, "Data Programming Model and Update Framework" for details on handling insertions and deletions using the static and dynamic SDO APIs.
3. Invoke the Web service proxy method to submit the changed SDO datagraph to your server to update your data sources. Here is an example of such an invocation:

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

Sample Java Static Web Service Client

The following code shows a sample Java static Web services client for DSP:

```
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("J D");


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

Developing Dynamic Web Service Clients

Developing your dynamic Web services involves the following:

- Setting Up a Dynamic Web Service Environment
- "Developing the Dynamic Web Service Client

Setting Up a Dynamic Web Service Environment

Set up the classpath for your Web service client using the following JAR files in the following order:

- wlsdo.jar
- webserviceclient.jar
- xbean.jar
- wlxbean.jar
- xqrl.jar

Note: The order of files shown above must be maintained.

Developing the Dynamic Web Service Client

There are three aspects to developing a dynamic Web service client. First the client must be created using standard development procedures. Then there are several DSP specific steps.

Initiating Dynamic Web Service Client Development

Follow the JAX-RPC instructions in JAX-RPC documentation (http://java.sun.com/webservices/jaxrpc/docs.html) to create the framework for a dynamic Web services client. Essentially this work involves:

- Creating your service factory instance.
- Creating your service using the URL to WSDL and service name.

Steps Specific to DSP

To enable the dynamic Web service for Data Services Platform you then need to:

1. Create a DataGraphCodec instance using the URL to WSDL.
2. Create a TypeMappingRegistry.
3. Create a TypeMapping and register the DataGraphCodec instance to be used to serialize/de-serialize the SOAPElement for both the request and response message.

Completing Dynamic Web Service Client Development

Finally complete development by:

1. Creating an instance of the JAX-RPC call interface for your read method (such as getCustomer( )).
2. Invoke your Web service.

3. Read or modify the response data using the SDO "Dynamic Data API" on page 1-10.
4. Create a call instance for the submit( ) method (such as submitArrayOfCustomer( )).
5. Wrap the serialized SDO datagraph with the SOAP message for the submit( ) method.
6. Invoke the submit( ) method to update your data sources.

Sample Java Dynamic Web Service Client

The following code (with comments emphasized) shows a complete Java dynamic Web services client for DSP, including import statements.

Listing 4-1 Sample Java Dynamic Web Service Client

```
import com.bea.sdo.impl.DataGraphCodec;
import com.bea.xml.XmlObject;
import commonj.sdo.DataObject;
import java.io.File;
import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.rpc.Call;
import javax.xml.rpc.Service;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.encoding.TypeMapping;
import javax.xml.rpc.encoding.TypeMappingRegistry;
import javax.xml.soap.SOAPConstants;
import javax.xml.soap.SOAPElement;

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

        System.setProperty("javax.xml.soap.MessageFactory",
"weblogic.webservice.core.soap.MessageFactoryImpl");

        // Setup the global JAX-RPC service factory
        System.setProperty( "javax.xml.rpc.ServiceFactory",
"weblogic.webservice.core.rpc.ServiceFactoryImpl");
```

```
        // create service factory
        ServiceFactory factory = ServiceFactory.newInstance();

        // define qnames
        String targetNamespace = "http://www.openuri.org/";

        QName serviceName = new QName(targetNamespace, "org3Test");

        QName portName = new QName(targetNamespace, "org3TestSoap");

        URL wsdlLocation = new
URL("http://localhost:7001/ElecWS/controls/org3Test.jws?WSDL");

        // create service
       Service service = factory.createService(wsdlLocation, serviceName);

        // create Codec
        DataGraphCodec dgCodec = new DataGraphCodec(wsdlLocation);

      TypeMappingRegistry registry = service.getTypeMappingRegistry();

        TypeMapping mapping = registry.getTypeMapping(
SOAPConstants.URI_NS_SOAP_ENCODING );

        mapping.register( SOAPElement.class,
                          new QName(targetNamespace, "getCustomer"),
                          dgCodec,
                          dgCodec );
        mapping.register( SOAPElement.class,
                       new QName( targetNamespace, "getCustomerResponse" ),
                          dgCodec,
                          dgCodec );
        mapping.register( SOAPElement.class,
                     new QName( targetNamespace, "submitArrayOfCustomer" ),
                          dgCodec,
                          dgCodec );
        mapping.register( SOAPElement.class,
```

```
                        new QName( targetNamespace,
"submitArrayOfCustomerResponse" ),
                        dgCodec,
                        dgCodec );

        // create call for read
       Call call = service.createCall(portName, new QName(targetNamespace,
"getCustomer"));

        XmlObject reqdoc = XmlObject.Factory.parse( "<getCustomer
xmlns='http://www.openuri.org/'/>");

        DataObject[] customerdocs = (DataObject[]) call.invoke(new
Object[]{reqdoc});


        // user can modify the DataObject here
        DataObject customer = customerdocs[0].getDataObject(0);
        customer.setString("EmailAddress", "BEAarray@BEA.com");

        String dgstring = customer.getDataGraph().toString();
        System.out.println(dgstring);

        // create call for submit
        call = service.createCall(portName, new QName(targetNamespace,
"submitArrayOfCustomer"));

        XmlObject submitdoc = XmlObject.Factory.parse(
   "<sub:submitArrayOfCustomer
   xmlns:sub='http://www.openuri.org/'><sub:docs>" +
        dgstring + "</sub:docs></sub:submitArrayOfCustomer>");

         Object obj = call.invoke(new Object[]{submitdoc});

        System.out.println(obj);

    }
}
```

CHAPTER 5

Using SQL to Access Data Services

Applications can access data services through SQL. This is necessary in the case of many reporting tools such as Crystal Reports, Hyperion, and Business Objects. But the ability to handle SQL is also useful in other contexts.

For example, it is useful to be able to run ad hoc SQL queries against data services using tools such as DbVisualizer. Application developers also can use a standalone Query Plan Viewer utility which supports both XQuery and SQL.

BEA AquaLogic Data Services Platform supports table parameters, an extension to SQL-92.

SQL access is provided through the AquaLogic Data Services Platform JDBC driver. The driver implements the java.sql.* interface in JDK 1.4x to provide access to an AquaLogic Data Services Platform server through the JDBC interface. You can use the JDBC driver to execute SQL92 SELECT queries, or stored procedures over AquaLogic Data Services Platform applications.

This chapter explains how to use SQL to access data services as well as how to set up and use the AquaLogic Data Services Platform JDBC driver. It covers the following topics:

- Publishing Data Service Functions As SQL
- SQL Support in AquaLogic Data Services Platform
- Accessing Data Services Functions Through JDBC
- Using the Query Plan Viewer Utility

Note: For data source and configuration pool information, refer to the WebLogic *Administration Guide*:

http://edocs.bea.com/platform/docs81/admin/index.html

Publishing Data Service Functions As SQL

To access data services through SQL, data service functions first need to be published as SQL objects through a JDBC interface. These SQL objects include tables, stored procedures, and functions.

Note: SQL objects published through AquaLogic Data Services Platform need to be enclosed in double quotes when used in an SQL query, if the object name contains a hyphen.

To publish data service functions as SQL Objects, the following tasks need to be performed:

1. Publish data service functions to a special schema that models them as SQL objects.
2. Build and deploy your AquaLogic Data Services Platform application.

Once deployed, the newly created SQL objects are available to your application through standard JDBC.

Techniques for publishing data services as SQL are described in "Publishing Data Services Functions for SQL Use" in the *Data Services Developer's Guide*.

Note: For details on accessing the AquaLogic Data Services Platform JDBC driver, and information on the relationship between data services artifacts and JDBC, see "Introducing AquaLogic Data Services Platform JDBC Driver" on page 5-11.

Using Custom Database Functions through AquaLogic Data Services Platform

Built-in or custom functions in your database can be made available through data services once the function has been registered with AquaLogic Data Services Platform through a library. For more information about using database functions with AquaLogic Data Services Platform, refer to Creating and Working with XQuery Function Libraries in *Data Services Developer's Guide*.

SQL Support in AquaLogic Data Services Platform

This section outlines the SQL-92 support in the AquaLogic Data Services Platform JDBC driver.

Supported Features

The AquaLogic Data Services Platform JDBC driver provides SQL-92 support for the SELECT statement. INSERT, UPDATE, and DELETE statements are not supported. DDL (Data Definition Language) statements are also not supported.

The AquaLogic 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`
- `java.sql.Blob`

Additional Details

The following limitations are known to exist in the AquaLogic Data Services Platform JDBC driver:

- Each connection points to only one AquaLogic Data Services Platform application.

The following table (Table 5-1) notes additional limitations that apply to SQL language features.

Table 5-1 Additional AquaLogic Data Services Platform JDBC Limitations Applying to SQL Language Features

Unsupported Feature	Comments	Example
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)`

Table Parameter Support

Table parameters extend SQL-92 by providing the ability to add parameters to SQL FROM clauses. For example, in SQL you can encounter a situation where it is necessary to code an exact number of parameters (highlighted) into a query.

In the following query test.rtlall.CUSTOMER is the entire customer table.

```
SELECT cust.CUSTOMER_ID, cust.FIRST_NAME, cust.LAST_NAME
FROM test.rtlall.CUSTOMER cust
where cust.CUSTOMER_ID in (?, ?, ?, ...)
and cust.LAST_NAME in (?, ?, ?, ... )
```

If a large number of parameters are involved, data entry can be slow and setting up the SQL statement tedious.

Table parameters provide an alternative. The following query uses table parameters (highlighted):

```
SELECT cust.CUSTOMER_ID, cust.FIRST_NAME, cust.LAST_NAME
FROM ? as id_cust(id_cust_num), test.rtlall.CUSTOMER as cust ,
? as id_cust(last_name)
WHERE id_cust.id_cust_num = cust.CUSTOMER_ID
and cust.LAST_NAME = id_cust.last_name
```

The table parameter is specified through the same mechanism as a a parameter; a question mark ("?") is used in place of the appropriate table name.

Note: In the current implementation only a single table column can be passed as a table parameter. If more than one column is specified, an exception is thrown.

Use Case for Table Parameters

A scenario: a data service contains consolidated information on all recent customer orders. A sales manager has a consolidated list of all government customers in European countries. The goal is to use a data service to obtain order information for that specific set of customers.

Stepping back from the example it is easy to see the that the scenario is a common one: a join between the manager's customer list and order information. However, if the manager's customer list is long and not already available through a database, it would be convenient to be able to pass in a list of values as if it were a column in a table.

In the SQL cited above a list of customers is passed in as a table with a single column. The clause:

```
? as id_cust(id_cust_num)
```

provides a virtual table value (id_cust) and a virtual column name (id_cust_num).

Although aliasing is not mandatory, it is generally recommended since default parameter column names follow numerical sequence (0, 1, and so forth) and as such are subject to unexpected name conflicts.

The one case where defaults are appropriate is when wildcards are employed such as:

```
select * from ?
```

and other simple cases involving wildcards.

Setting Table Parameters Using JDBC

Table parameters are passed to data services through the AquaLogic Data Services Platform JDBC driver, specifically through its TableParameter class. The class (shown in its entirety in Listing 5-1) represents an entire table parameter as well as the rows it represents.

Listing 5-1 Table Parameter Interface

```
public class TableParameter implements Serializable {
    /**
     * Constructor
     *
     * @schema      the schema for the table
     */
```

```
public TableParameter(ValueType[] schema);

/**
 * Creates a new a row and adds it to the list of rows in this table
 */
public Row createRow();
/**
 * Gets the rows of this table
 */
public List/*Row*/ getRows();
/**
 * Gets the schema of this table
 */
public ValueType[] getSchema();
/**
 * Represents a row in the table
 */
public class Row implements Serializable {
    /**
     * Sets a value to a particular column
     * @param colIdx    the index of the column to set
     * @param val       the value for the column
     * @exception       if index is out of bounds
     */
    public void setObject(int colIdx,Object val) throws SQLException;
    Object getObject(int colIdx);
}
```

Creating Table Parameters

The following steps show how to create a TableParameter class:

1. Instantiate TableParameter with the schema of your table.

   Note: At present only one column is supported.

2. Call the createRow( ) method on TableParameter to create a new Row object representing a tuple in the table.

3. Fill in the row object using the setObject(colIdx,val) call until all columns are set.
4. Call createRow( ) again to create as many rows as the table requires.

JDBC Usage

TableParameters are passed through JDBC just like any other parameter, through a PreparedStatement.

For example, you would first create a PreparedStatement with the query:

```
SELECT cust.CUSTOMER_ID, cust.FIRST_NAME, cust.LAST_NAME
FROM ? as id_cust(id_cust_num), test.rtlall.CUSTOMER as cust ,
? as id_cust(last_name)
WHERE id_cust.id_cust_num = cust.CUSTOMER_ID
and cust.LAST_NAME = id_cust.last_name
and cust.ORDER_AMT > ?
```

- Set the property on the PreparedStatement using the following call:

  ```
  Float orderAmt = new Float(75f)
  setObject(3,orderAmt)
  ```

  This sets the value for the third parameter (the parameter in the WHERE clause).

- Set the properties on the PreparedStatement for the two table parameters using the following calls:

  ```
  setObject(2,y)
  setObject(1,z)
  ```

  to set the value for the table parameter. The "y" and “z” values should be of the type TableParameter.

Table Parameter Example

The following simplified example illustrates the use of a table parameter. An in-memory list contains three customers: CUST_1, CUST_2, and CUST_3.

```
SELECT cust.cust_num, order.item, order.price
FROM ? as cust(cust_num), Order as order
WHERE cust.cust_num = order.cust_id
```

The supporting JDBC code is:

```
//first create the table parameter
```

```
ValueType[] schemaList = new ValueType[1];
schemaList[0] = ValueType.REPEATING_VARCHAR_TYPE;
TableParameter tableParam = new TableParameter(schemaList);

// then create the rows in your virtual table
// (in practice you would read data in from a file
// or some other data stream)

TableParameter.Row row1 = tableParam.createRow();
row1.setObject(1," CUSTOMER _1");
TableParameter.Row row2 = tableParam.createRow();
Row2.setObject(1," CUSTOMER _2");

// repeat for second table parameter
//     (...)

// execute the query
PreparedStatement objPreparedStatement = objConnection.prepareStatement(
  "SELECT cust.CUSTOMER_ID, cust.FIRST_NAME, cust.LAST_NAME
FROM ? as id_cust(id_cust_num), test.rtlall.CUSTOMER as cust , ? as
id_cust(last_name)
WHERE id_cust.id_cust_num = cust.CUSTOMER_ID
and cust.LAST_NAME = id_cust.last_name
and cust.ORDER_AMT > ? ");

// and set table parameters
objPreparedStatement.setObject(1,tableParam);
objPreparedStatement.setObject(2,tableParam2);
objResultSet = objPreparedStatement.executeQuery();
```

XML 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 XML types are defined by:

```
xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

The Java types are defined by:

```
java.sql.Types
```

XML types that can be mapped to SQL Type Mappings are shown in Table 5-2.

Table 5-2 XML and SQL Type Mapping

XML Type	SQL Types
xdt:dayTimeDuration	Types.OTHER
xdt:yearMonthDuration	Types.OTHER
xs:boolean	Types.BOOLEAN.
xs:byte	Types.SMALLINT
xs:dateTime	Types.TIMESTAMP
xs:date	Types.DATE
xs:decimal	Types.DECIMAL
xs:double	Types.DOUBLE
xs:duration	Types.OTHER
xs:float	Types.REAL
xs:hexBinary	Types.BLOB
xs:int	Types.INTEGER
xs.integer	Types.DECIMAL
xs:long	Types.BIGINT
xs:negativeInteger	Types.DECIMAL
xs:nonNegativeInteger	Types.DECIMAL
xs:nonPositiveInteger	Types.DECIMAL
xs:positiveInteger	Types.DECIMAL
xs:short	Types.SMALLINT
xs:string	Types.VARCHAR
xs:time	Types.TIME
xs:unsignedByte	Types.SMALLINT

Table 5-2 XML and SQL Type Mapping

XML Type	SQL Types
xs:unsignedInt	Types.BIGINT
xs:unsignedLong	Types.DECIMAL
xs:unsignedShort	Types.INTEGER

Accessing Data Services Functions Through JDBC

The AquaLogic Data Services Platform JDBC driver enables JDBC and ODBC clients to access information available from data services using SQL. The JDBC driver increases the flexibility of the AquaLogic Data Services Platform integration layer by enabling access from database visualization and reporting tools, such as DbVisualizer, Crystal Reports, Hyperion, and Business Objects. For the client, the AquaLogic Data Services Platform integration layer appears as a relational database, with each data service function comprising a table. Internally, AquaLogic Data Services Platform translates SQL queries into XQuery.

Some constraints associated with the AquaLogic Data Services Platform JDBC driver include the following:

- The AquaLogic Data Services Platform JDBC driver can only be used to access data through data services that have a flat data shape, which means that the data service type cannot have nesting. This is because SQL provides a traditional, two-dimensional approach to data access as opposed to the multi-level, hierarchical approach defined by XML.
- The AquaLogic Data Services Platform JDBC driver only exposes non-parameterized flat data service functions as tables because SQL tables do not have parameters. Parameterized flat data services are exposed as SQL stored procedures.

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

This section discusses the SQL Name Mapping technique used to map SQL functions to AquaLogic Data Services Platform functions, along with the steps to configure the JDBC Driver connection using Java and non-Java applications. It includes the following topics:

- Introducing AquaLogic Data Services Platform JDBC Driver
- Supported Functions
- Configuring the AquaLogic Data Services Platform JDBC Driver

- Accessing AquaLogic Data Services Platform JDBC Driver Using a Java Application
- Accessing Data Service Functions from DbVisualizer
- Connecting to AquaLogic Data Services Platform Client Using ODBC-JDBC Bridge from Non-Java Applications
- Accessing Data Services Data from Reporting Tools

Introducing AquaLogic Data Services Platform JDBC Driver

The AquaLogic Data Services Platform JDBC driver has the following features:

- Supports SQL-92 SELECT statements
- Implements JDBC 3.0 API
- Supports AquaLogic Data Services Platform with JDK 1.4
- Usable from both Java and ODBC clients
- Allows metadata access control at the JDBC driver level

Using AquaLogic Data Services Platform JDBC Driver, you can control the metadata accessed through SQL based on the access rights set at the JDBC driver level. This ensures that authorized users can view only those tables and procedures, which they are authorized to access. However, to be able to use this feature, the AquaLogic Data Services Platform console configuration should be set to check access control. For more information, refer to the "Securing Data Services Platform Resources" section in the *Administration Guide*.

Note:

- The AquaLogic 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 AquaLogic Data Services Platform product libraries: `wlclient.jar`, `ld-client.jar`, `Schemas_UNIFIED_Annotation.jar`, `jsr173_api.jar`, and `xbean.jar`.s

Data Service Functions and Corresponding JDBC Artifacts

AquaLogic Data Services Platform views data retrieved from a database in the form of data sources and functions. The following table (Table 5-3) shows the equivalent terminology.

Table 5-3 AquaLogic Data Services Platform and JDBC Driver Artifacts

AquaLogic Data Services Platform	JDBC
AquaLogic Data Services Platform Project	Database Catalog Name
Folder under the DSP project	Virtual name to maintain consistency between the database structure (`Catalog.Schema.Table`) and AquaLogic Data Services Platform
Function with parameters	Stored procedure
Function without parameters	Table

For example, if you have a project `TestDataServices` and `CUSTOMERS.ds` with a function `getCustomers()` under the schema `MySchema`, then you can map `getCustomers` as an SQL object as follows:

```
TestDataServices.MySchema.getCustomer
```

where `TestDataServices` is the catalog and `MySchema` is the name of the schema folder. This mapping is based on mapping the AquaLogic Data Services Platform functions to SQL objects. For more information about mapping AquaLogic Data Services Platform functions as SQL objects, refer to Publishing Data Services Functions for SQL Use in the *Data Services Developer's Guide*.

Supported Functions

AquaLogic Data Services Platform supports many functions that can be used to access data services through various reporting tools. In the following tables functions are divided into the following types:

- Numeric Functions
- String Functions
- Datetime Functions
- Aggregate Functions

Numeric Functions

The following numeric operation functions are provided:

Table 5-4 Numeric Functions

Function	Signature	Comment
ABS	`numeric ABS (numeric n)`	ABS returns the absolute value of *n*. If *n* is NULL, the return value is NULL.
CEIL	`numeric CEIL(numeric n)`	CEIL returns the smallest integer greater than or equal to *n*. If *n* is NULL, the return value is NULL.
FLOOR	`numeric FLOOR(numeric n)`	FLOOR returns largest integer equal to or less than *n*. If *n* is NULL, the return value is NULL.
ROUND	`numeric ROUND (numeric n)`	ROUND returns *n* rounded to 0 decimal places. If *n* is NULL, the return value is NULL.

String Functions

The following string management functions are provided:

Table 5-5 String Functions

Function	Signature	Comment
CONCAT	`varchar CONCAT(varchar s1, varchar s2)`	CONCAT returns *s1* concatenated with *s2*. If any argument is NULL, it is considered to be equivalent to the empty string.
LENGTH	`numeric LENGTH(varchar s)`	LENGTH returns the length of *s*. The function returns 0 if *s* is NULL.
LOWER	`varchar LOWER(varchar s)`	LOWER returns *s*, with all letters lowercase. If *s* is NULL, the function returns an empty string.

Table 5-5 String Functions

Function	Signature	Comment
LTRIM	`varchar LTRIM(varchar s)`	LTRIM trims leading blanks from *s*. If *s* is NULL, the function returns NULL.
RTRIM	`varchar RTRIM(varchar s)`	RTRIM trims trailing blanks from *s*. If *s* is NULL, the function returns NULL.
SUBSTR	`varchar SUBSTR(varchar s, numeric start)`	SUBSTR with two arguments returns substring of *s* starting at start, inclusive. The first character in *s* is located at index 1. If *s* is NULL, the function returns an empty string.
TRIM	`varchar TRIM(varchar s)`	TRIM trims leading and trailing blanks from *s*. If *s* is NULL, TRIM returns NULL.
UPPER	`varchar UPPER(varchar s)`	UPPER returns *s*, with all letters uppercase. If *s* is NULL, UPPER returns the empty string.

Datetime Functions

The following datetime functions are provided:

Table 5-6 Datetime Functions

Function	Signature	Comment
DAYS	`numeric DAYS(T value)`	DAYS returns the days component from *value*. *T* can be a date, timestamp, or duration. If *value* is NULL, the result is NULL.
HOUR	`numeric HOUR(T value)`	HOUR returns the hour component from *value*. *T* can be one of time, timestamp, or duration. If *value* is NULL, the result is NULL.
MINUTE	`numeric MINUTE(T value)`	MINUTE returns the minute component from *value*. *T* can be a time, timestamp, or duration. If *value* is NULL, the result is NULL.
MONTH	`numeric MONTH(T value)`	MONTH returns the month component from *value*. *T* can be one of date, timestamp, or duration. If *value* is NULL, the result is NULL.
SECOND	`numeric SECOND(T value)`	SECOND returns the seconds component from *value*. *T* can be a time, timestamp, or duration. If *value* is NULL, the result is NULL.
YEAR	`numeric YEAR(T value)`	YEAR returns the year component from value. *T* can be one of date, timestamp, or duration. If *value* is NULL, the result is NULL.

Aggregate Functions

The following aggregation functions are provided:

Table 5-7 Aggregate Functions

Function	Signature	Comment
COUNT	`numeric COUNT(ROWS r)`	COUNT returns the number of rows in *r*.
AVG	`T AVG(T r)`	AVG returns the average values of all values in *r*. *T* can be a numeric or duration type.
SUM	`T SUM(T r)`	SUM returns the sum of all values in *r*. *T* can be a numeric or duration type.
MAX	`T MAX(T r)`	MAX returns a value from *r* that is greater than or equal to every other value in *r*. T can be a numeric, varchar, date, timestamp, or duration type.
MIN	`T MIN(T r)`	MIN returns a value from *r* that is less than or equal to every other value in *r*. *T* can be a numeric, varchar, date, timestamp, or duration type.

Configuring the AquaLogic Data Services Platform JDBC Driver

AquaLogic Data Services Platform JDBC driver is located in an archive file named `ldjdbc.jar`, which is available in the following directory after you install BEA AquaLogic Data Services Platform:

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

To use the AquaLogic Data Services Platform JDBC driver on a client computer, you need to configure the classpath, class name, and the URL for the JDBC driver. To configure the driver on a client computer, perform the following steps:

1. Copy the `ldjdbc.jar` and `weblogic.jar` to the client computer.
2. Add `ldjdbc.jar` and `weblogic.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.dsp.jdbc.driver.DSPJDBCDriver
      ```

   b. Set the driver URL to:

      ```
      jdbc:dsp@<DSPServerName>:<DSPServerPortNumber>/<DSPApplicationName>]
      ```

      For example:

      ```
      jdbc:dsp@localhost:7001/RTLApp
      ```

   c. You can also set the default catalog name and schema name in the URL while connecting to the JDBC driver using the following syntax:

      ```
      jdbc:dsp@<DSPServerName>:<DSPServerPortNumber>/<DSPApplicationName>/<ca
      talogname>/<schemaname>
      ```

      Note: If you do not specify the CatalogName and SchemaName in the JDBC driver URL, then you need to specify the three-part name for all queries. For example:

      ```
      select * from <catalogname>.<schemaname>.CUSTOMER
      ```

   d. Debugging can be enabled by using the `logFile` property. To log debugging information, use the following JDBC driver URL syntax:

      ```
      jdbc:dsp@localhost:7001/test;logFile=c:\output.txt
      ```

      In this case, the log file will be created in `c:\output.txt`. You can also specify the debug property separately instead of specifying it with the URL.

      Note: If you build a SQL query using a reporting tool, the unqualified JDBC function name is used in the generated SQL. Consequently, when an application developer invokes an XFL database function, the default catalog and schema name must be defined in the JDBC connection URL. It is also a requirement that any JDBC connection utilize those functions available from a single SQL catalog:schema pair location.

      The following is an example URL defining a default catalog and schema for a JDBC connection:

      ```
      jdbc:dsp@localhost:7001/myApplication/myCatalog/mySchema
      ```

5. To configure the connection object for the AquaLogic Data Services Platform application, you can specify configuration parameters as a Properties object or as a part of the JDBC URL.

Note: If `application`, `default_catalog`, or `default_schema` appears in both the connection properties and the URL, the one in the URL takes precedence.

Configuring the Connection Using the Properties Object:

You can configure the JDBC driver connection using the properties object as follows:

```
props.put("user", "weblogic");

props.put("password", " weblogic ");

props.put("application", "RTLApp");

Connection objConnection =
DriverManager.getConnection("jdbc:dsp@localhost:7001", props);
```

Note: You can specify the default schema and catalog name using the `default_catalog` and `default_schema` property fields in case you do not specify it in the properties.

Alternatively, you can specify the AquaLogic Data Services Platform application name, RTLApp, in the connection object itself, as shown in the following snippet:

```
props.put("user", " weblogic");

props.put("password", " weblogic ");

Connection objConnection =
DriverManager.getConnection("jdbc:dsp@localhost:7001/RTLApp",props);
```

Configuring the Connection in the JDBC URL:

You can also configure the JDBC driver connection without creating a properties object, as shown in the following code:

```
Connection objConnection =

DriverManager.getConnection("jdbc:dsp@localhost:7001/RTLApp;logFile=c:\out
put.txt; ", <username>, <password>);
```

Accessing AquaLogic Data Services Platform JDBC Driver Using a Java Application

The steps to connect an application to AquaLogic Data Services Platform as a JDBC/SQL data source are substantially the same as connecting to any JDBC/SQL data source directly. In the database URL, use the AquaLogic Data Services Platform application name as the database identifier with "dsp" as the sub-protocol, in the form:

```
jdbc:dsp@<WLServerAddress>:<WLServerPort>/<DSPApplicationName>
```

For example:

```
jdbc:dsp@localhost:7001/RTLApp
```

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

```
com.bea.dsp.jdbc.driver.DSPJDBCDriver
```

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

Obtaining a Connection

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

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

For example:

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

// Load the driver
Class.forName("com.bea.dsp.jdbc.driver.DSPJDBCDriver");


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

Using the preparedStatement Interface

The `storedQueryWithParameters` method explained in this section, demonstrates how to use the preparedStatement interface using a connection object (`con`). It is a valid connection obtained through the `java.sql.Connection` interface to the WebLogic Server, which hosts AquaLogic Data Services Platform.

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

In the method, `CUSTOMER` refers to `CUSTOMER.ds`.

```
public ResultSet storedQueryWithParameters() throws java.sql.SQLException
```

```
{
       PreparedStatement preStmt =
              con.prepareStatement (
                  "SELECT * FROM DataServices.MySchema.CUSTOMER WHERE
CUSTOMER.LAST_NAME=?");
       preStmt.setString(1,"SMITH");
       ResultSet res = preStmt.executeQuery();
       return res;
}
```

In the SELECT query, `DataServices` is the catalog name and `MySchema` is the name of the schema folder.

Note: To use the CUSTOMER table in the SELECT query, you must first map it as an SQL Object. For details, refer to "Publishing Data Service Functions As SQL" on page 5-2.

Using the CallableStatement Interface

Once a connection is established to a server where AquaLogic Data Services Platform 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 AquaLogic 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
       con.prepareCall("call  DataServices.MySchema.getCustomerById(?) ");
       call.setString(1, "CUSTOMER2");
       ResultSet resultSet = call.executeQuery();
       return resultSet;
```

Note: You can also use the prepareCall method as follows:

```
con.prepareCall(" { call DataServices.MySchema.getCustomerById(?)}");
```

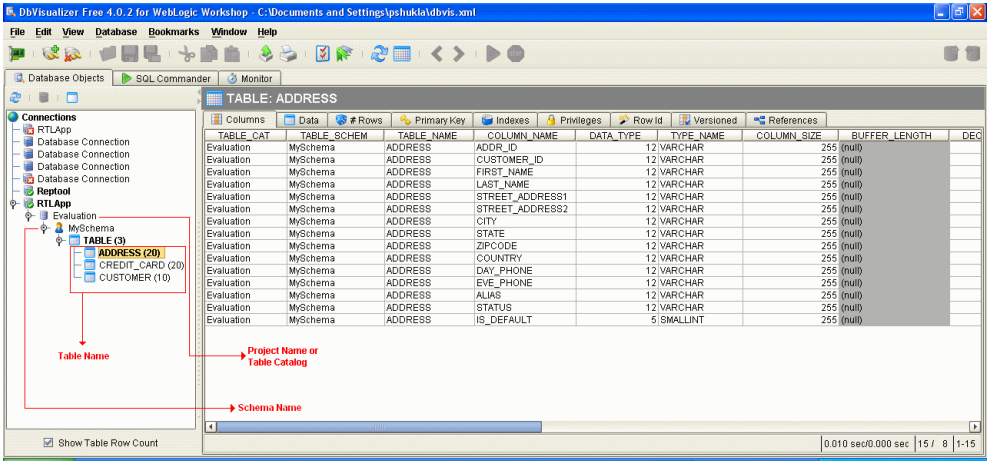
Accessing Data Service Functions from DbVisualizer

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

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

This section describes how to connect to the driver from DBVisualizer. Figure 5-1 shows a sample application as viewed from DbVisualizer.

Figure 5-1 DbVisualizer View of DSP

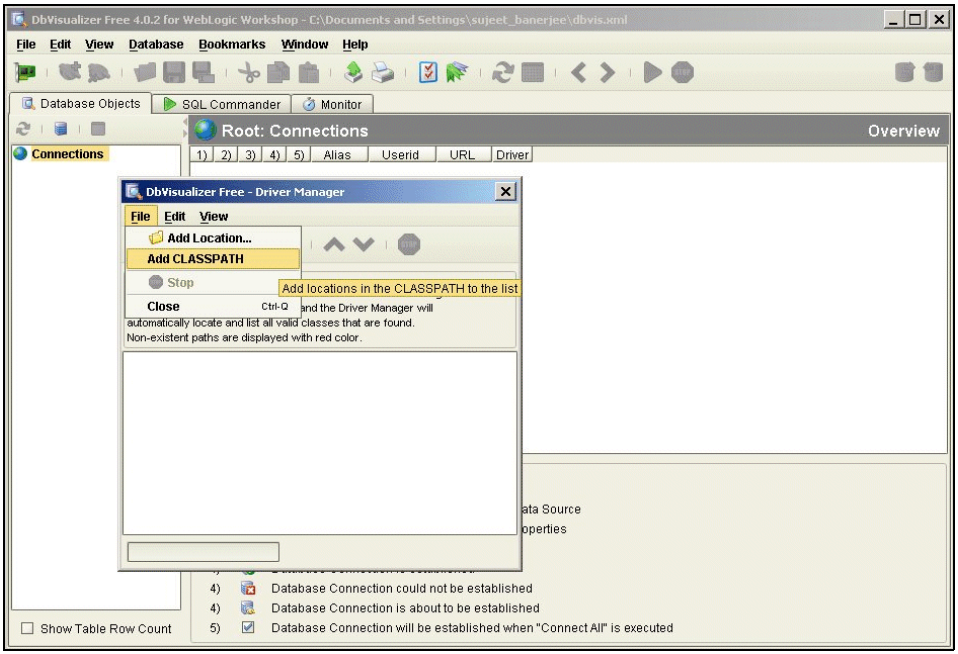


To use DBVisualizer, perform the following steps:

1. Click Start→All Programs→BEA WebLogic Platform 8.1→Other Development Tools→DbVisualizer.
2. Configure DBVisualizer.
   a. Ensure that `ldjdbc.jar` exists in your CLASSPATH.
   b. To establish a connection with AquaLogic Data Services Platform select Driver Manager from the Database menu.

c. Select Add CLASSPATH from the File menu of the driver manager dialog. You should see the `ldjdbc.jar` listed.

d. Select `ldjdbc.jar` from the list and then select Find Drivers from the Edit menu of the driver manager. The Driver Manager will detect the `com.bea.dsp.jdbc.driver.DSPJDBCDriver` JDBC driver and display it in the list box.



e. Close the driver manager.

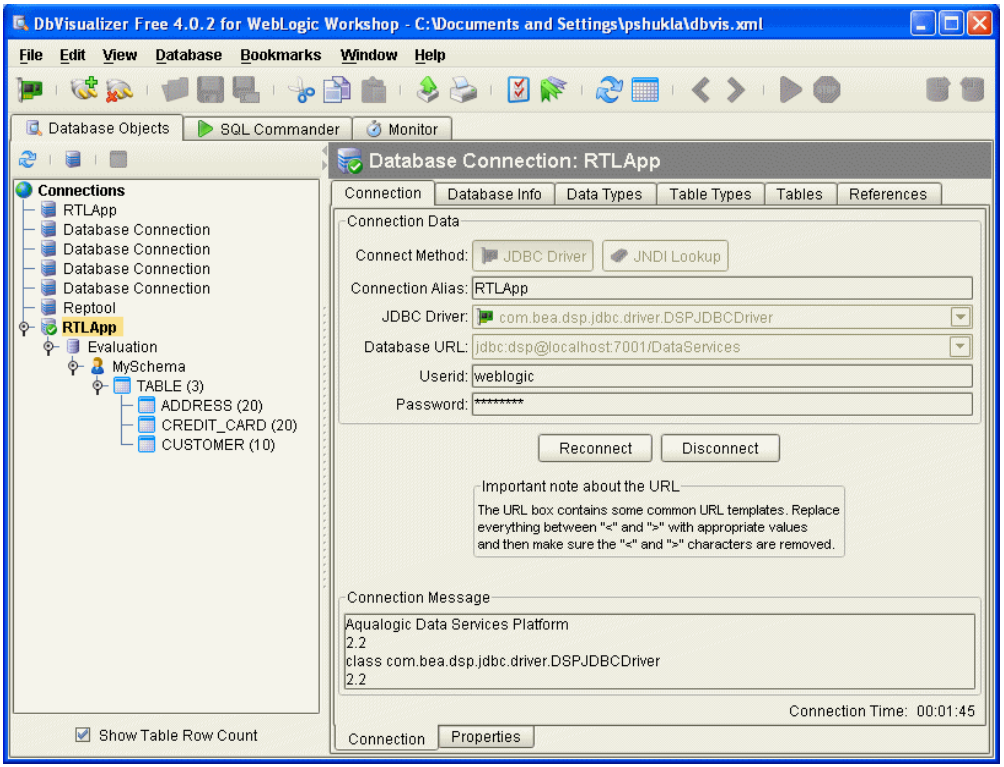
3. Add connection parameters by performing the following steps:

a. On the right pane, select the JDBC Driver as `com.bea.dsp.jdbc.driver.DSPJDBCDriver`, from drop down list.

b. For the Database URL, enter `jdbc:dsp@<machine_name>:<port>/<app_name>`. For example `"jdbc:dsp@localhost:7001/RTLApp"`

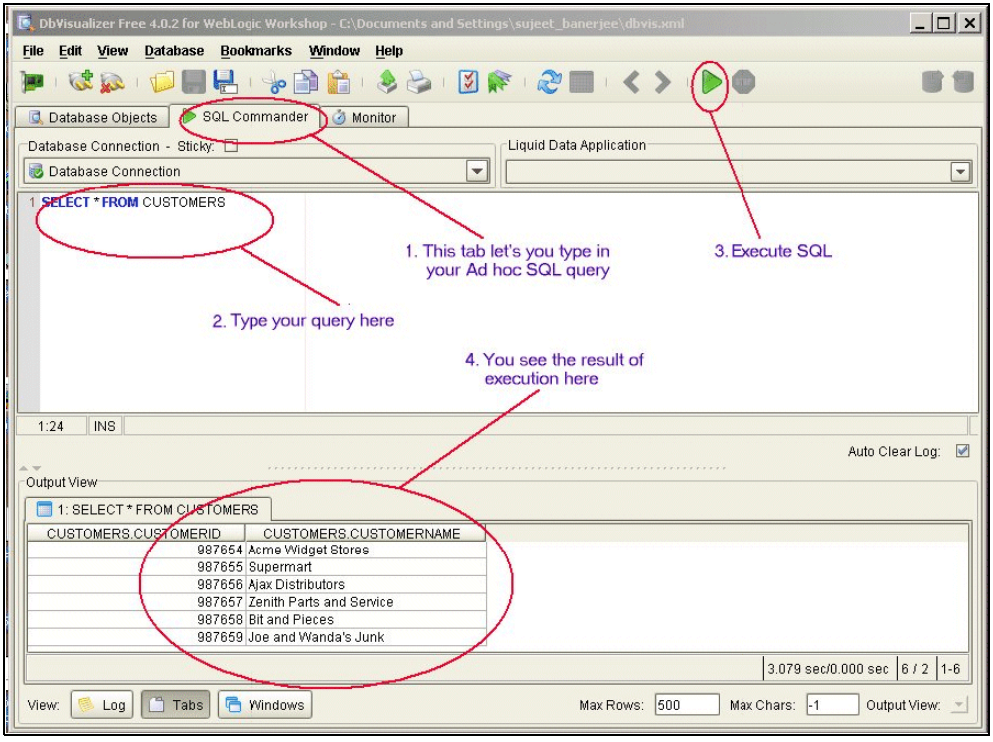
c. Provide the user name and password for connecting to the AquaLogic Data Services Platform application.

4. Click **Connect**. On completion of a successful connection, you should see the following:



5. On the right pane of the window (see preceding figure), 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.

6. Execute ad hoc queries by activating the SQL Commander tab as shown in the following figure. Enter the SQL query and click the execute icon.



Once you have configured your ODBC-JDBC Bridge, you can use your application to access the data source presented by AquaLogic Data Services Platform. The usual reason for doing so is to connect AquaLogic Data Services Platform to the reporting tool that you need to use.

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

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

You can use an ODBC-JDBC bridge to connect to AquaLogic 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 AquaLogic Data Services Platform JDBC driver.

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

Using OpenLink ODBC-JDBC Bridge

The Openlink ODBC-JDBC driver can be used to interface with the AquaLogic Data Services Platform JDBC driver to query AquaLogic Data Services Platform applications with client applications, such as Crystal Reports 10, Business Objects 6.1, and MS Access 2003.

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 installing OpenLink ODBC-JDBC-Lite, refer to:

   `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 may overwrite it.

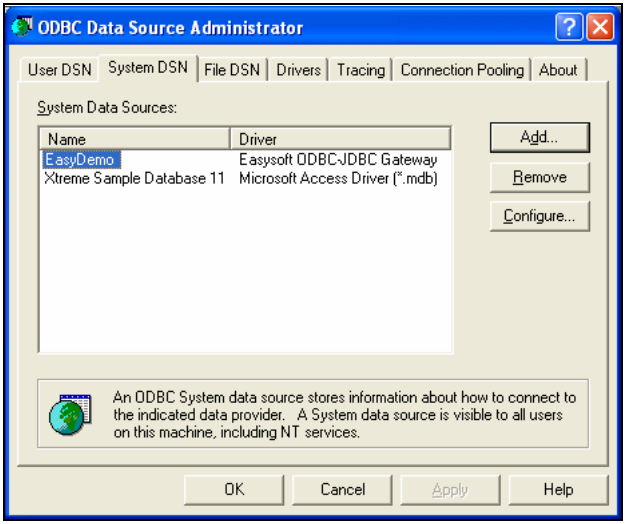
2. Create a system DSN and configure it for your AquaLogic Data Services Platform application by performing the following steps:

   a. Ensure that the CLASSPATH contains the following jars required by ODBC-JDBC-Lite, as well as `ldjdbc.jar` and `weblogic.jar`. A typical CLASSPATH might look like:

   ```
   D:\lddriver\ldjdbc.jar; D:\bea\weblogic81\server\lib\weblogic.jar;
   D:\odbc-odbc\openlink\jdk1.4\opljdbc3.jar;
   D:\odbc-jdbc\openlink\jdk1.4\megathin3.jar;
   ```

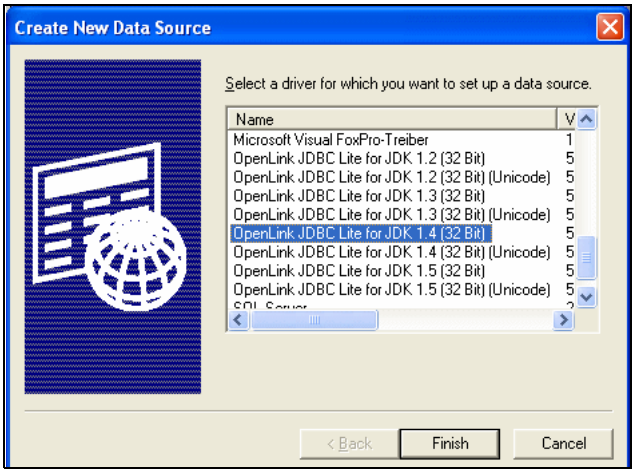
   b. Update your system path to point to the `jvm.dll`, which should be under your `%javaroot%\jre\bin\server` directory.

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

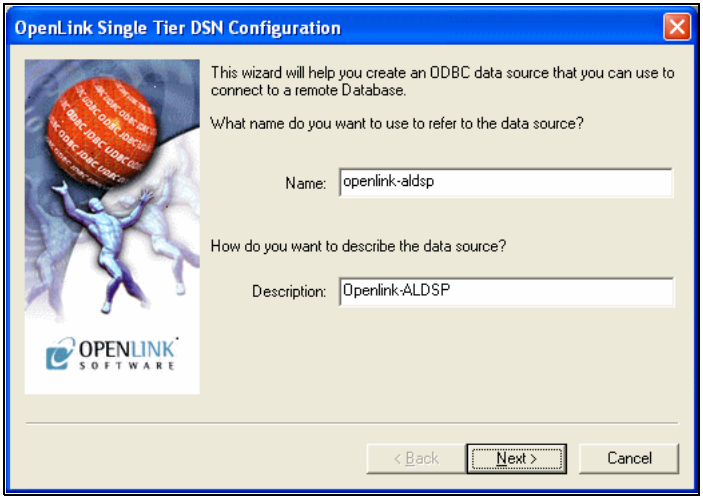


d. Click **System DSN** tab and then click **Add**.

e. Select JDBC Lite for JDK 1.4 (32 bit) and click Finish.



f. Specify the DSN name. For example, openlink-aldsp, as shown in the following figure:

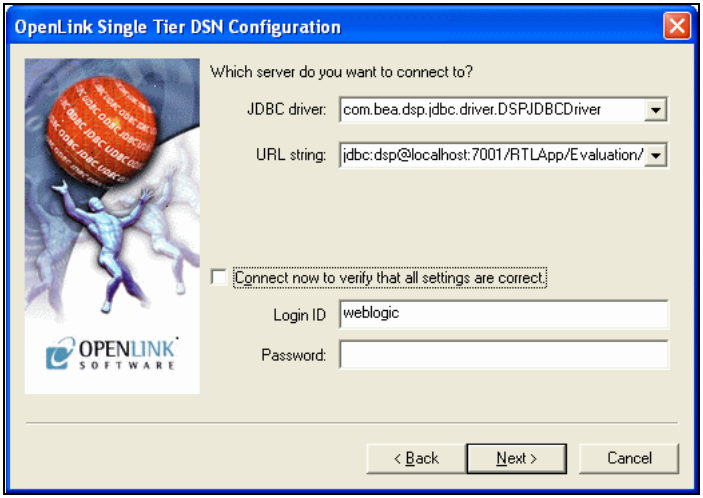


g. Click Next. Then on the next screen, enter the following in the JDBC driver field:

```
com.bea.dsp.jdbc.driver.DSPJDBCDriver.
```

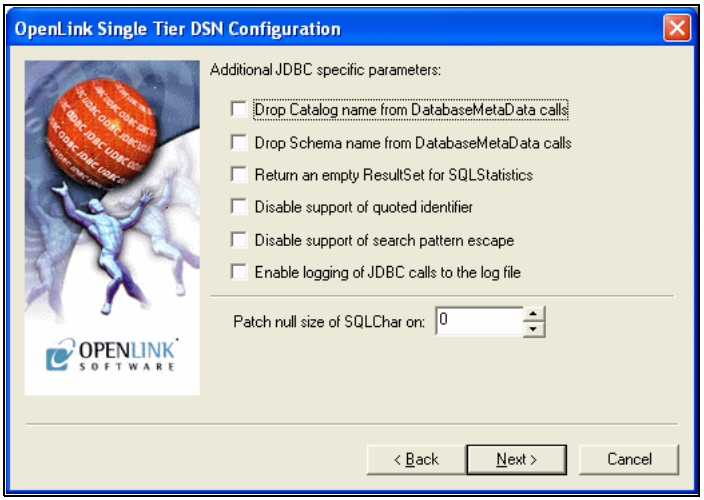
Enter the following in the URL string field:

```
jdbc:dsp@<machine_name>:<port>/<app_name>/<catalogname>/<schemaname>
```

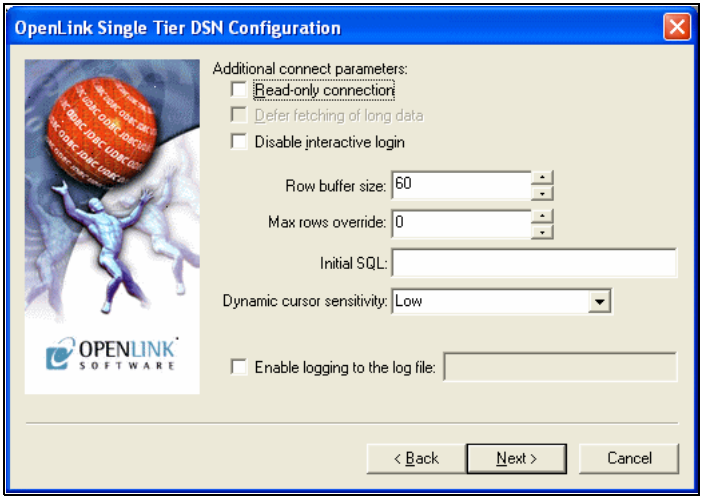


h. Select the Connect now to verify that all settings are correct checkbox. Provide the login and password to connect to the AquaLogic Data Services Platform WebLogic Server.

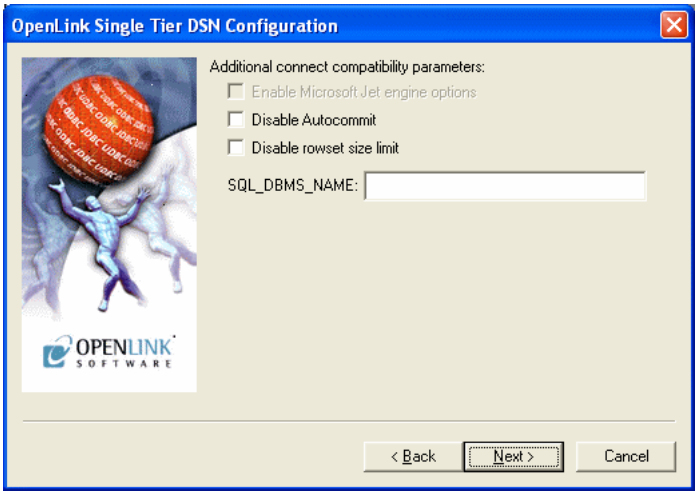
i. Click **Next**. The screen shown below will display:



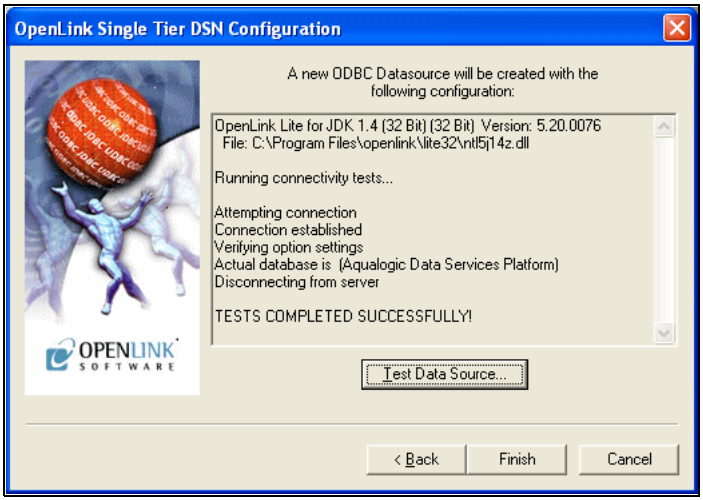
j. Click **Next**. The following screen is displayed.



k. Click **Next** and specify the connection compatibility parameters as displayed in the following figure.



l. Click Next and then click Test Data Source. This screen will verify that the setup is successful.



m. Click Finish.

Using the EasySoft ODBC-JDBC Bridge

Applications can also communicate with the AquaLogic 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.

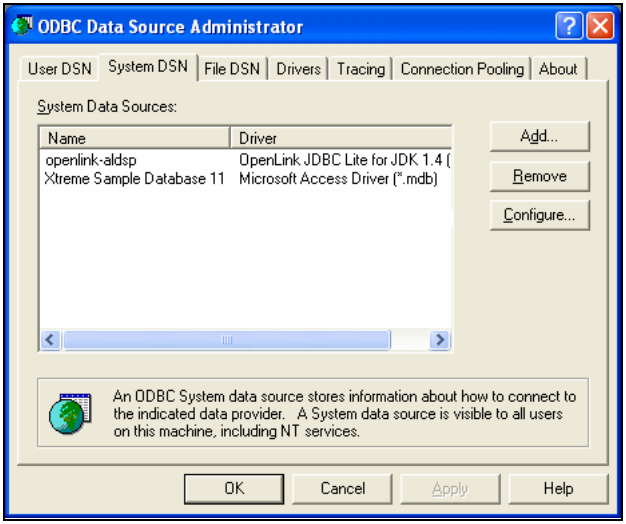
To use the EasySoft bridge, perform the following steps:

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

   http://www.easysoft.com

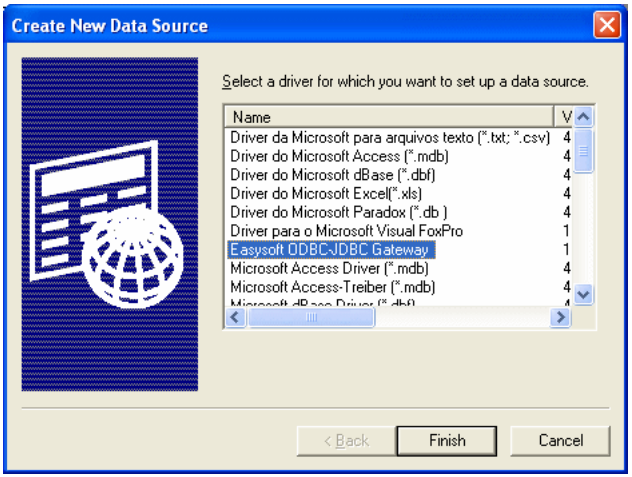
2. Create a system DSN and configure it for AquaLogic Data Services Platform by performing the following steps:

   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 following figure 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 the URL enter:

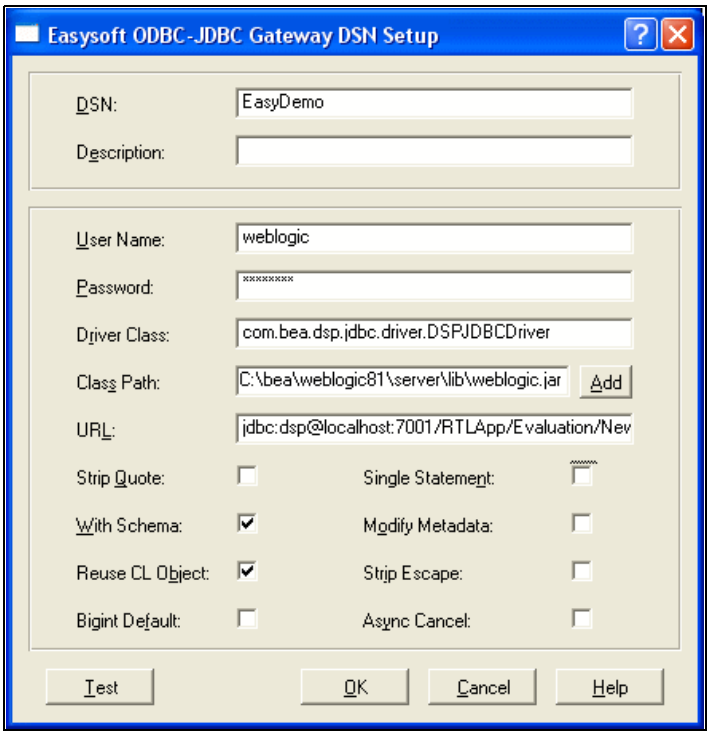
   ```
   jdbc:dsp@<machine_name>:<port>/<app_name>/<catalogname>/<schemaname>
   ```

   For example:

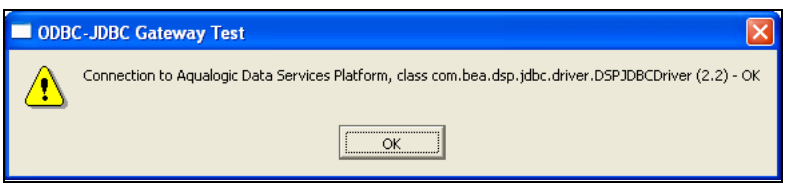
   ```
   jdbc:dsp@localhost:7001/DataServices/Evaluation/NewSchema;logFile=c:
   \jdbc.log
   ```

For the Driver class, enter:

```
com.bea.dsp.jdbc.driver.DSPJDBCDriver
```



e. Click Test. The following screen will display, indicating the connection has completed successfully.



f. Click OK to complete the set-up sequence.

Accessing Data Services Data from Reporting Tools

This section describes how to configure the following reporting tools to use the AquaLogic Data Services Platform ODBC-JDBC driver:

- Crystal Reports XI

- Business Objects XI-Release 2 (ODBC)
- Hyperion-ODBC
- Microsoft Access 2003-ODBC
- Microsoft Excel 2003-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 XI

This section describes the steps to connect Crystal Reports to the AquaLogic Data Services Platform JDBC driver along with information about standard configuration files that are available with AquaLogic Data Services Platform installation. It also describes the limitations of using Crystal Reports with AquaLogic Data Services Platform. It includes the following topics:

- Crystal Reports Configuration File Support
- Limitations
- Connecting to Crystal Reports Using JDBC

Crystal Reports Configuration File Support

Before you start using Crystal Reports with AquaLogic Data Services Platform, you must modify the default Crystal Reports configuration file, `CRConfig.xml`, to verify that Crystal Reports is able to access data services through JDBC.

The sample AquaLogic Data Services Platform `CRConfig.xml` file is provided with the standard installation of AquaLogic Data Services Platform. The sections of the configuration file that need to be modified contain the string “ALDSP”. This file is located at:

```
<weblogic81>/LiquidData/resources/ReportingToolConfigs/CrystalReports
```

You cannot use the sample `CRConfig.xml` directly unless Crystal Reports is installed in the same directory as the path specified in the sample `CRConfig.xml` and AquaLogic Data Services Platform is installed in the same directory as the path specified in the sample `CRConfig.xml`. Therefore, you need to modify the default `CRConfig.xml` file available with Crystal Reports according to the sample `CRConfig.xml` file available with AquaLogic Data Services Platform.

Note: Although you can modify the sample `CRConfig.xml` file available with AquaLogic Data Services Platform, it is recommended that you modify the default Crystal Reports `CRConfig.xml` file based on the sample file available with AquaLogic Data Services Platform.

Table 5-8 identifies some restrictions and specifies configuration changes you need to make to your Crystal Reports configuration file when accessing data using the AquaLogic Data Services Platform JDBC driver.

Table 5-8 Crystal Reports Configuration File Support for AquaLogic Data Services Platform

Configuration File	Discussion
`CRConfig.xml`	• Add the path to `weblogic.jar` and `ldjdbc.jar` to the beginning of CLASSPATH environment variable. • Replace the entire `<JDBC>` element with the sample <JDBC> element. • Modify the `<JDBCURL>` and `<JDBCUserName>` elements based on the JDBC driver URL and JDBC user name that you want to use to establish the connection with AquaLogic Data Services Platform JDBC driver.

The following code snippet is a sample of the `<JDBC>` element with the JDBC driver URL and user name configuration settings:

```
<!-- ALDSP2.5 : The following <JDBC configuration is specific to
ALDSP. -->
<JDBC>
<CacheRowSetSize>100</CacheRowSetSize>
<!-- ALDSP: Please replace <JDBCURL> with your URL. -->
<JDBCURL>jdbc:dsp@localhost:7001/YOUR_APP</JDBCURL>
<JDBCClassName>com.bea.dsp.jdbc.driver.DSPJDBCDriver</JDBCClassName>
<!-- ALDSP: Please replace <JDBCUserName> with your user name. -->
<JDBCUserName>ENTER_USER_NAME_HERE</JDBCUserName>
<JNDIURL></JNDIURL>
<JNDIConnectionFactory></JNDIConnectionFactory>
<JNDIInitContext>/</JNDIInitContext>
<JNDIUserName></JNDIUserName>
<GenericJDBCDriver>
   <Option>Yes</Option>
```

```
    <DatabaseStructure>catalogs,schemas,tables</DatabaseStructure>
    <StoredProcType>Standard</StoredProcType>
    <LogonStyle>Standard</LogonStyle>
  </GenericJDBCDriver>
  </JDBC>
```

Limitations

Before you use Crystal Reports to access data services, ensure that you consider the following facts:

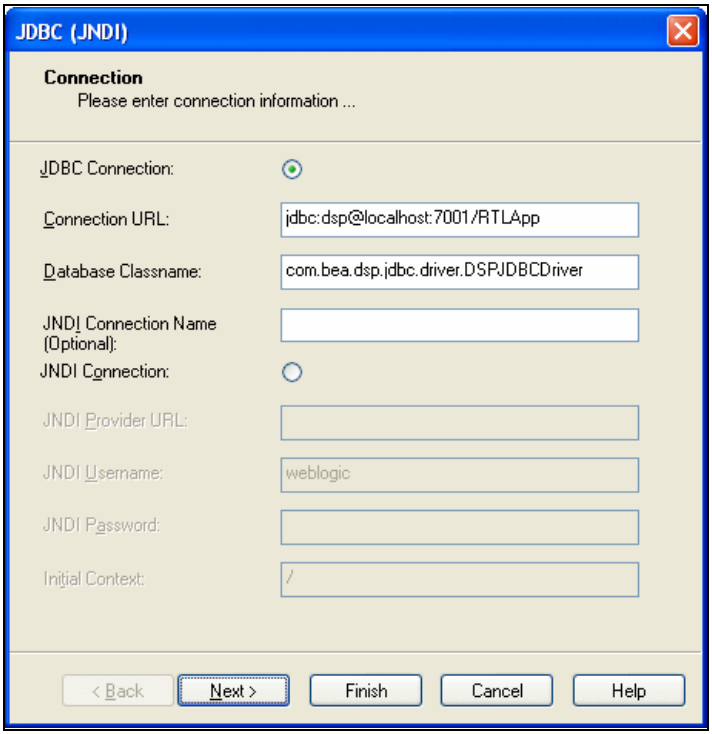
- Crystal Reports is not able to invoke the stored procedure with parameters for any AquaLogic Data Services Platform XQuery function, which has the parameters defined using the built-in data type keyword such as $integer. To resolve this issue, change the parameter name in the XQuery function.
- Crystal Reports supports all XML types that are supported by AquaLogic Data Services Platform JDBC driver except the following:
  - `yearMonthDuration`
  - `dayTimeDuration`
- Some of the JDBC functions that are used by Crystal Reports are not supported by AquaLogic Data Services Platform. Refer to the "Supported Functions" on page 5-12 for a list of supported functions.

Connecting to Crystal Reports Using JDBC

To connect Crystal Reports to the JDBC driver and access data services to generate reports, perform the following steps:

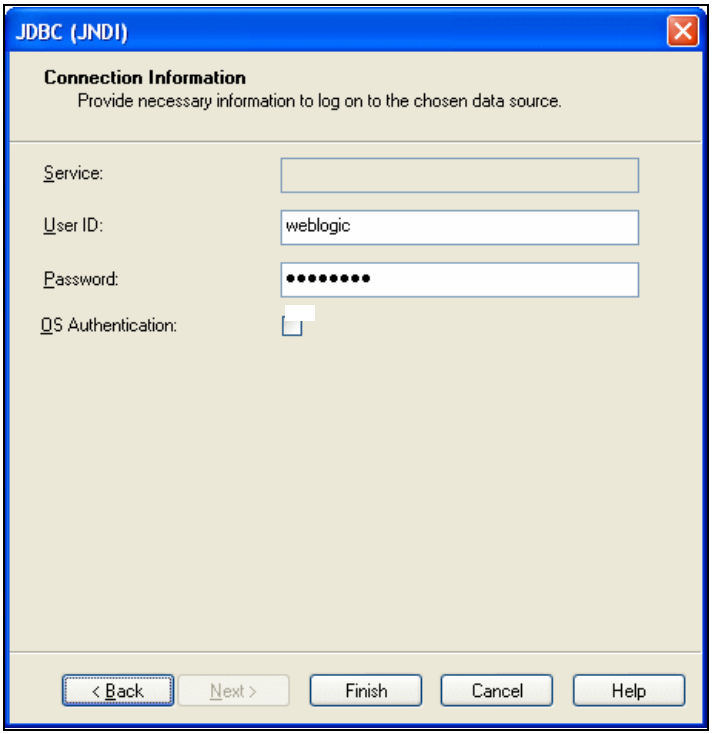
1. Crystal Reports 11.0 comes with a direct JDBC interface that can be used to interact directly with the AquaLogic Data Services Platform JDBC driver. You need to create a new connection for JDBC by selecting JDBC (JNDI) connection from the Standard Report Creation Wizard. This displays the JDBC (JNDI) Connection dialog box, as shown in Figure 5-2.

Figure 5-2 Connection Dialog Box



2. Specify the connection parameters for the JDBC interface of Crystal Reports as shown in Figure 5-3.

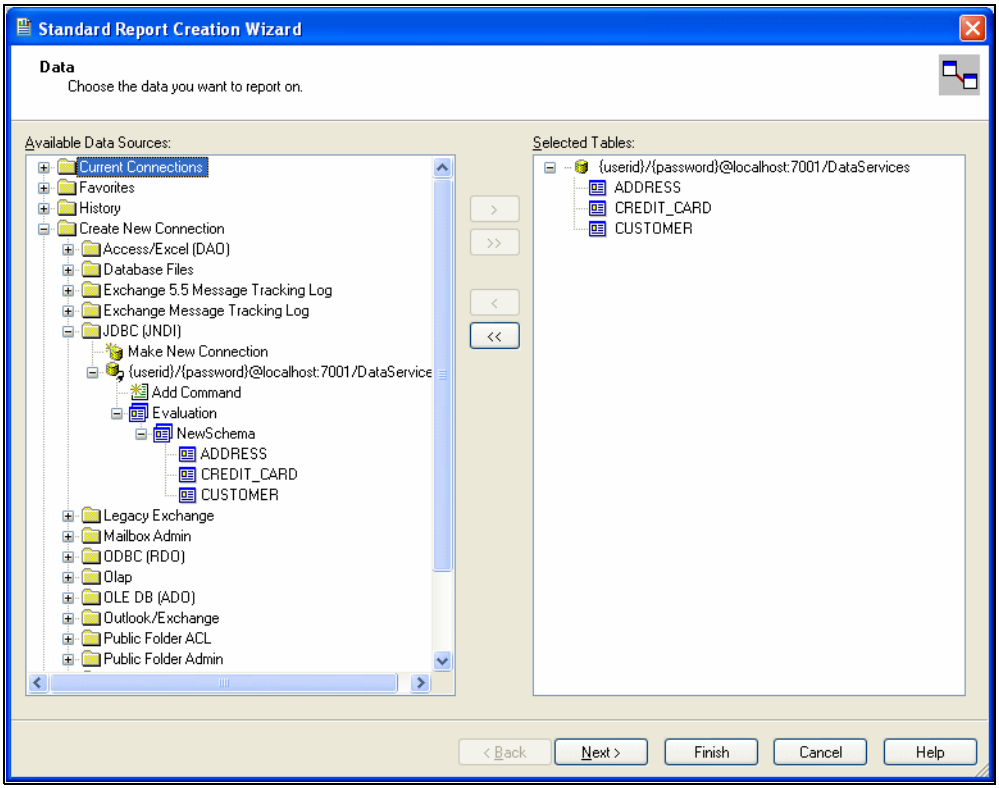
Figure 5-3 Connection Information Dialog Box



Note: The Database drop down box is populated with the available catalogs (AquaLogic Data Services Platform applications) once you have specified the correct parameters for User ID and Password as shown in Figure 5-3.

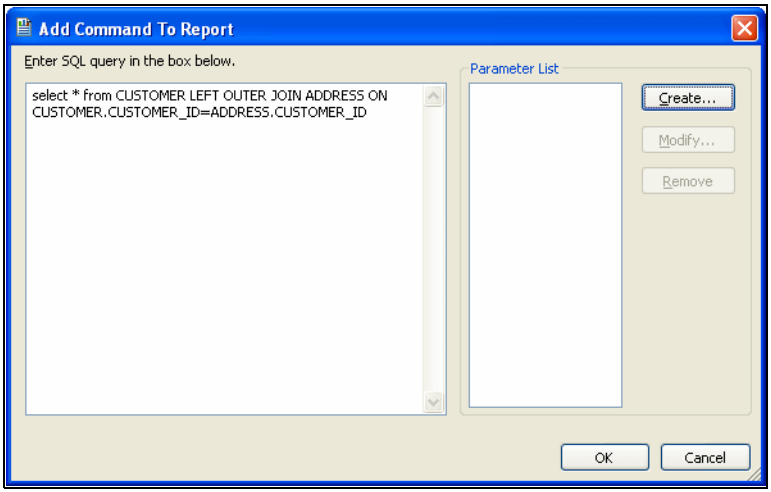
3. Click Finish to go back to the Standard Report Creation Wizard.
4. Drag the tables for which you want to generate the report to the right side as shown in Figure 5-4.

Figure 5-4 Metadata Browser Window



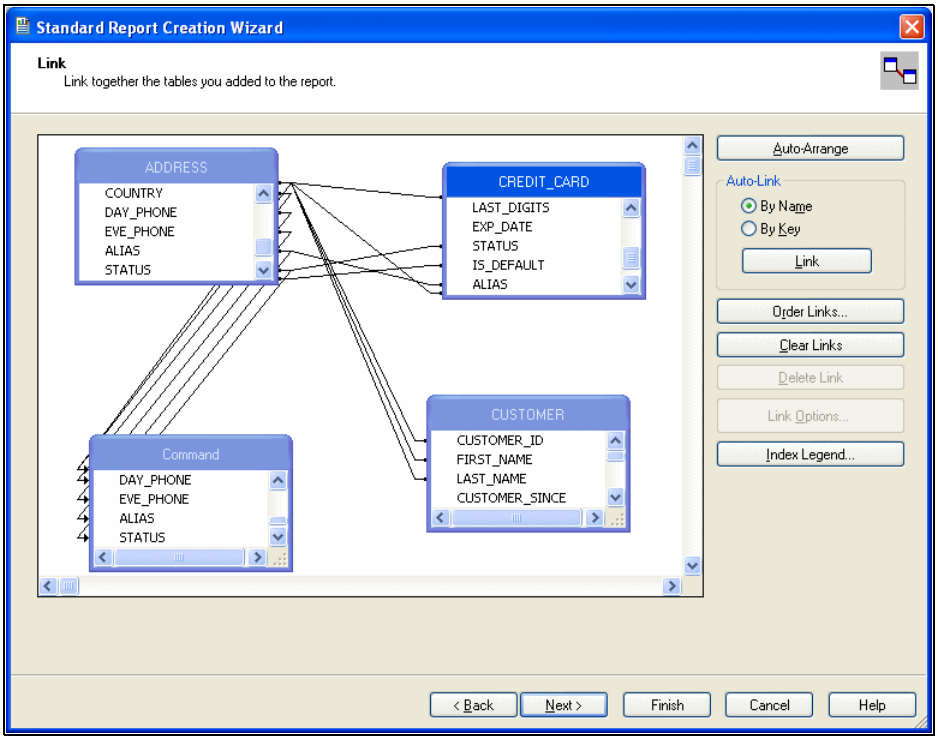
5. Alternatively, you can choose the Add Command option to type an SQL query directly, which displays a window as shown in Figure 5-5.

Figure 5-5 Add Command



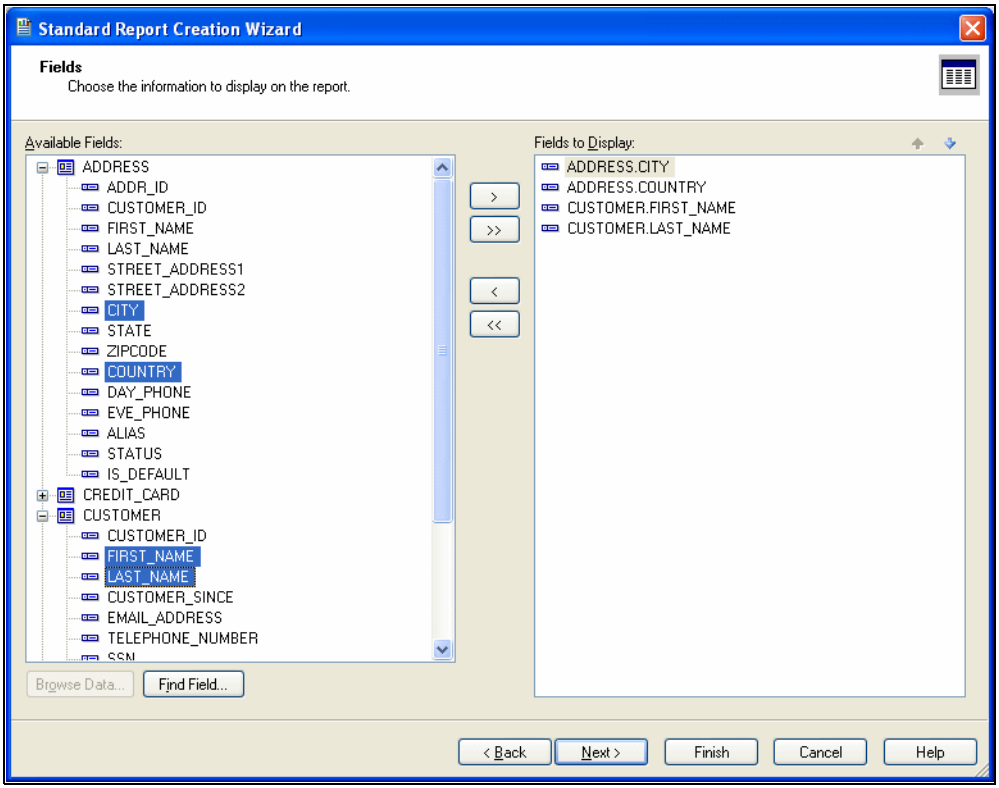
6. Click OK and the Command is added to the right side of the window.
7. Clicking Next in the wizard shows you all the available views for generating the report, as shown in Figure 5-6.

Figure 5-6 Link Screen



8. Click Next to go back to the Column chooser window as shown in Figure 5-7. This window allows you to select the columns you want to see in the final report.

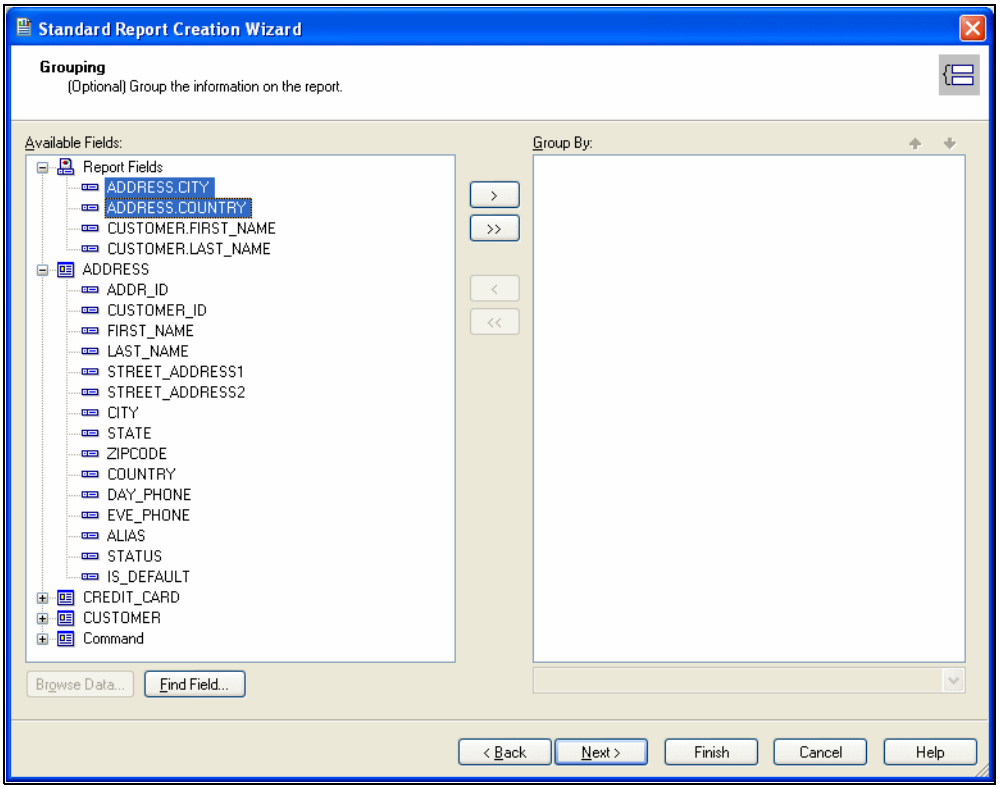
Figure 5-7 Column Chooser



Note: This example chooses columns from the user-generated Command and the view CUSTOMER.

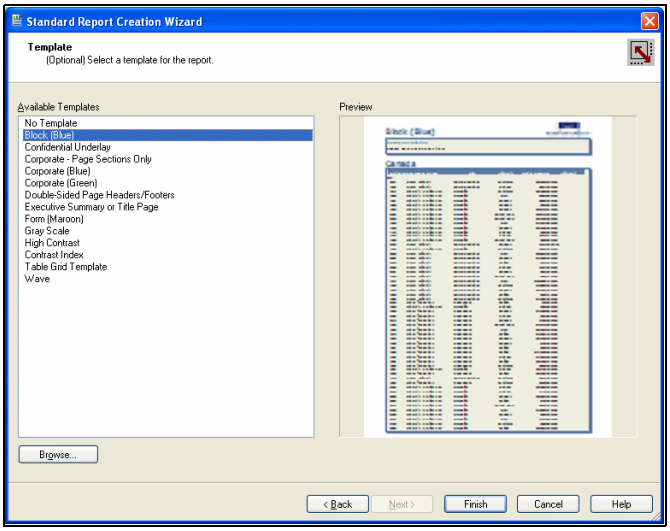
9. Click Next and the Grouping screen is displayed (as shown in Figure 5-8), which allows you to choose a column to group by. (This grouping is performed by Crystal Reports. The Group-by information is not passed on to the JDBC driver.)

Figure 5-8 Group-by Screen



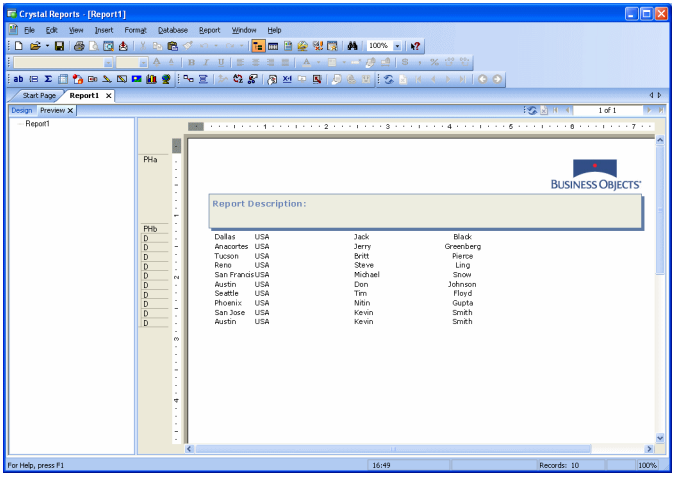
10. Skip the next few screens for now. Click Next till you reach the Template Chooser Screen Figure 5-9. Choose any appropriate Template. In this example, the user has chosen the Block (Blue) Template.

Figure 5-9 Template Chooser Screen



11. Click Finish. A report similar to that shown in Figure 5-10 is displayed.

Figure 5-10 Generated Report



Business Objects XI-Release 2 (ODBC)

Business Objects allows you to create a Universe and 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.

Note: If you need to specify a four part name in a SELECT list (such as, `<catalogname>.<schemaname>.<tablename>.<columnname>`), define a table alias using the `FROM` clause, and then use only two parts `<tablealias>.<columnname>` in the `SELECT` list. AquaLogic Data Services Platform JDBC driver extracts only the last two parts from the `SELECT` list item, and ignores the rest.

For example,

```
SELECT C.Name FROM DataServices.MySchema.CUSTOMER C
```

where,

`DataServices` is the catalog name

`MySchema` is the schema name

`CUSTOMER` is the table name

`Name` is the column name

`C` is the table alias for CUSTOMER

This section provides information on configuring Business Objects to access the AquaLogic Data Services Platform JDBC driver. It includes the following topics:

- Business Objects Configuration File Support
- Prerequisites and Limitations
- Generating a Business Objects Report

Business Objects Configuration File Support

There are two BusinessObjects configuration files, `odbc.prm` and `odbc.sbo`, available with the standard BusinessObjects installation, which need to be replaced with the `odbc.prm` and `odbc.sbo` configuration files available with AquaLogic Data Services Platform, to access data services using BusinessObjects.

When you install BusinessObjects, these files are copied to the following location:

```
<Business Objects Home >\BusinessObjects Enterprise
11.5\win32_x86\dataAccess\connectionServer\odbc
```

With the AquaLogic Data Services Platform installation, these configuration files are available at the following location:

```
<weblogic81>/LiquidData/resources/ReportingToolConfigs/BusinessObjects
```

After installing AquaLogic Data Services Platform, save the original configuration files available with BusinessObjects at a different location and then replace them with the files packaged with AquaLogic Data Services Platform.

The BusinessObjects configuration files provided with AquaLogic Data Services Platform should be reviewed for comments. (Relevant comments contain the string "ALDSP").

Tip: When first getting started using BusinessObjects with AquaLogic Data Services Platform, it is recommended that the included configuration file be used to verify your ability to access data services through JDBC.

Table 5-9 identifies some restrictions and specifies configuration changes you may want to make to your BusinessObjects configuration files when accessing data using the AquaLogic Data Services Platform JDBC driver.

Table 5-9 Business Objects Configuration File Support for AquaLogic Data Services Platform

Configuration File	Discussion
ODBC.PRM	Specifically supported: • `EXT_JOIN` (outer join) • `QUALIFIER` (table prefix) • `DISTINCT` • `ANSI_92` Not supported: • `INTERSECT` • `INTERSECT_IN_SUBQUERY` • `MINUS` • `MINUS_IN_SUBQUERY`
ODBC.SBO	Set Transactional Available option to YES

Prerequisites and Limitations

Before you start using Business Objects to access data services, ensure that you consider the following facts:

- To generate a report using Business Objects, all the data sources need to be from the same catalog. This implies that in WebLogic Workshop the same project should be used to publish the data services for SQL use. This catalog needs to be defined as the default catalog while connecting to AquaLogic Data Services Platform. However, the catalog can contain arbitrary number of schemas.
- When you connect Business Objects to AquaLogic Data Services Platform using OpenLink, then you need to specify the default catalog name in the AquaLogic Data Services Platform JDBC driver URL, while configuring OpenLink. If you do not specify the default catalog name, then an error similar to the following will be generated:

```
com.bea.ld.sql.compiler.ast.SQLTypeCheckingException: Invalid table
reference. No such table null.Xtreme.CUSTOMER found.

at

com.bea.ld.sql.compiler.ast.TableRSN.inferSchemaAndCheck(Lcom/bea/ld
/

sql/context/SQLContext;Lcom/bea/ld/sql/types/SchemaIndex;)V(TableRSN
.java:149)
```

For details about configuring OpenLink, refer to "Using OpenLink ODBC-JDBC Bridge" on page 5-25.

- BusinessObjects supports all XML types that are supported by AquaLogic Data Services Platform JDBC driver except the following:
  - `yearMonthDuration`
  - `dayTimeDuration`
  - `xs:boolean`
  - `xs:hexBinary`

Generating a Business Objects Report

To generate a report, perform the following steps:

1. Create a Universe:
   a. Run the Business Objects 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.
   c. If the DSN you want to use does not appear in the list (this happens if you are using the application for the first time), then click New to create a new connection.
   d. In the Define a New Connection wizard, select Generic ODBC3 Datasource as the middleware.

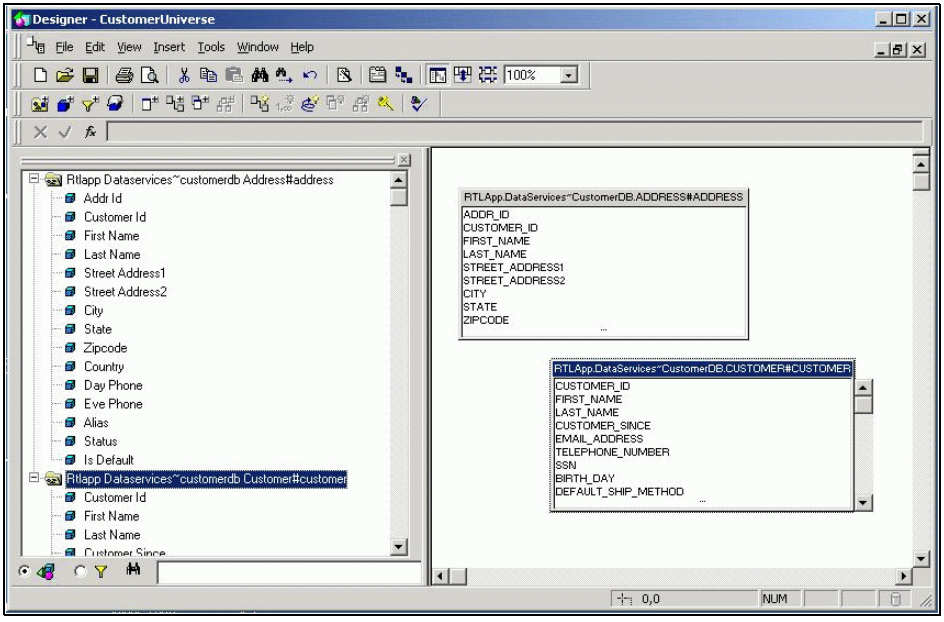
e. Specify the user name and password to connect to WebLogic Server and select openlink-aldsp as the DSN. For details about configuring the OpenLink ODBC-JDBC bridge, refer to "Using OpenLink ODBC-JDBC Bridge" on page 5-25.

f. Click Next and test if the connection with the server is successful. Follow the instructions in the wizard to complete creating the connection.

g. After creating the connection, specify this connection in the Universe and click OK. A new blank panel is displayed.

h. From the Insert menu select Table. Once the list of tables is shown in the Table Browser, double click the tables you wish to put in the Universe. You should see a screen similar to that shown in Figure 5-11.

Figure 5-11 Table Browser



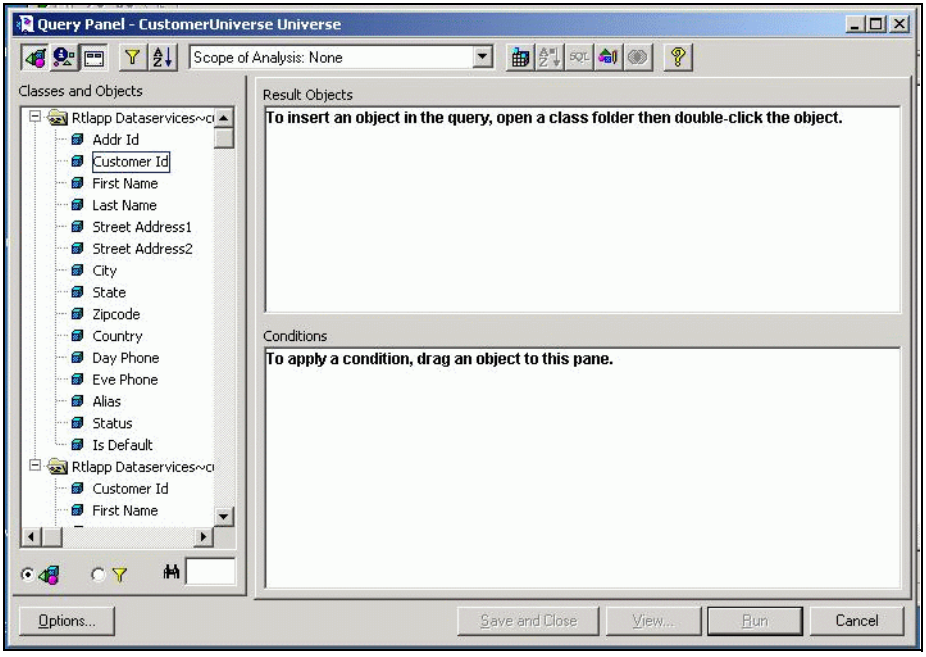
i. Save the Universe and exit.

2. To create a new report:

a. Run the Desktop Intelligence application. Click New to open the New Report Wizard. Choose Specify to access data and click Begin.

b. Choose a Universe and click Next. On the left pane, you should see the tables and their fields (columns) on expansion, as shown in Figure 5-12.

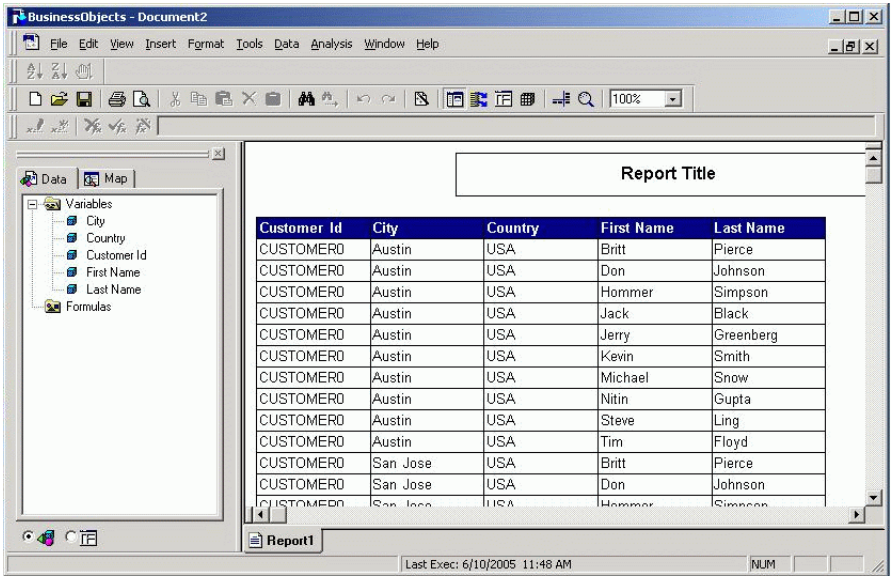
Figure 5-12 Query Panel



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.

d. Click Run to execute the query. The result is displayed as shown in Figure 5-13.

Figure 5-13 Business Objects Panel



3. You can execute the pass-through queries as follows:

   a. In the Desktop Intelligence application, click New to create a new report.

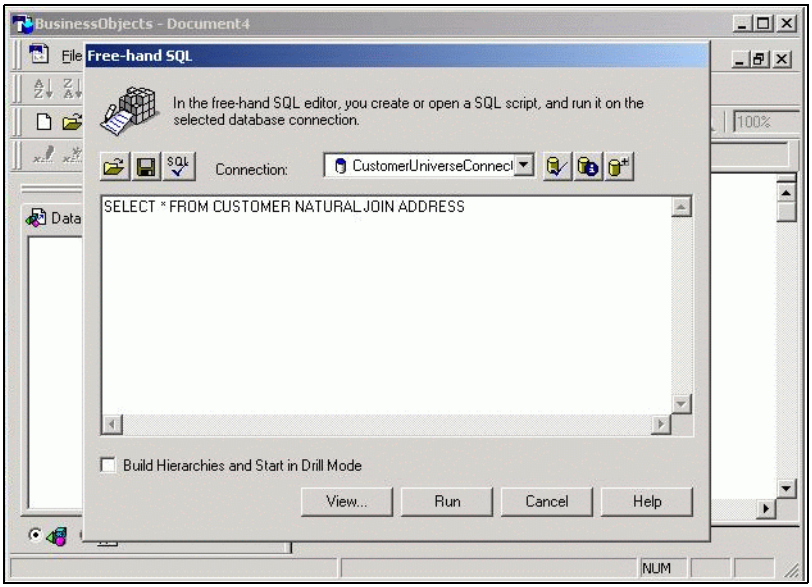
   b. In the New Report Wizard choose Others instead of Universe.

   c. Choose Free-hand SQL and click Finish.

   d. Select the connection you made using Designer.

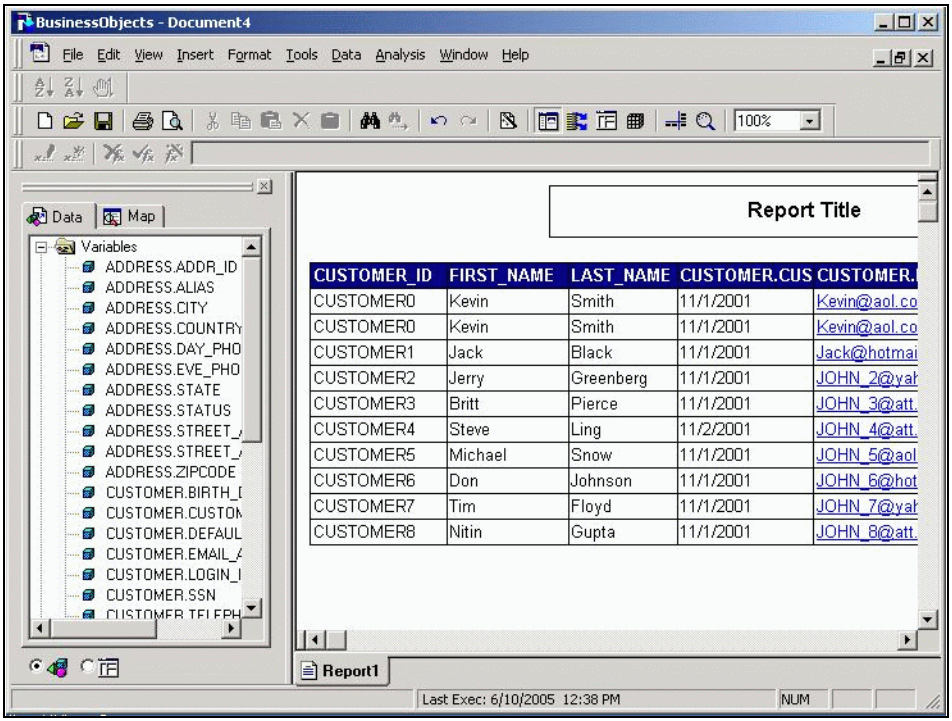
   e. Type in your SQL query and click Run to generate the report, as shown in Figure 5-14.

Figure 5-14 Specifying the SQL Query



f. Click Run. You should see the report shown in Figure 5-15.

Figure 5-15 Business Objects Report



Hyperion-ODBC

Hyperion allows you to generate interactive and production reports using its Interactive Reporting Studio and Production Reporting Studio. This section describes the steps to access AquaLogic Data Services Platform data sources using an ODBC-JDBC bridge and generate interactive and production reports.

It includes the following topics:

- Limitations
- Using Hyperion Production Reporting Studio
- Using Hyperion Interactive Reporting Studio

Limitations

Before you start using Hyperion to access data services, ensure that you consider the following facts:

- Hyperion supports all XML types that are supported by AquaLogic Data Services Platform JDBC driver except the following:

XML Type	Behavior
`xs:hexBinary`	Hyperion displays the metadata information, however, you cannot include this binary type in the `SELECT` list.
`xs:yearMonthDuration`	• Hyperion displays the metadata information but Item Type property in the metadata is empty. • The `SELECT` query using this XML type throws the "`Program type out of range`" exception.
`xs:dayTimeduration`	• The metadata information for the Item Type property is not retrieved. • SELECT list using this XML type throws the "`Program type out of range`" exception.

- Hyperion Interactive Reporting Studio does not support boolean datatype with OpenLink. The Item Type property in the metadata information for the boolean type shows the type as byte for boolean datatype, and any query issued with the datatype generates the "`Program type out of range`" error.

Using Hyperion Production Reporting Studio

To establish the connection and view results using the Hyperion Production Reporting Studio, perform the following steps:

1. Open Production Reporting Studio and select the tabular report from the Create New Report wizard.
2. Select the bridge you want to use to connect to AquaLogic Data Services Platform from the Data Connection box and click OK.

   Note: To create reports using Hyperion Production Reporting Suite, you need to use lower case names for tables published in AquaLogic Data Services Platform. See "Publishing Data Services Functions for SQL Use" in *Data Services Developer's Guide* for further details on how tables are published in AquaLogic Data Services Platform.

3. If you want to create a new connection, click New and follow the instructions in the Create Data Connection wizard. Select the provider as ODBC. For more information about configuring OpenLink or EasySoft as your preferred ODBC-JDBC bridge, refer to "Using OpenLink ODBC-JDBC Bridge" on page 5-25 or "Using the EasySoft ODBC-JDBC Bridge" on page 5-29.

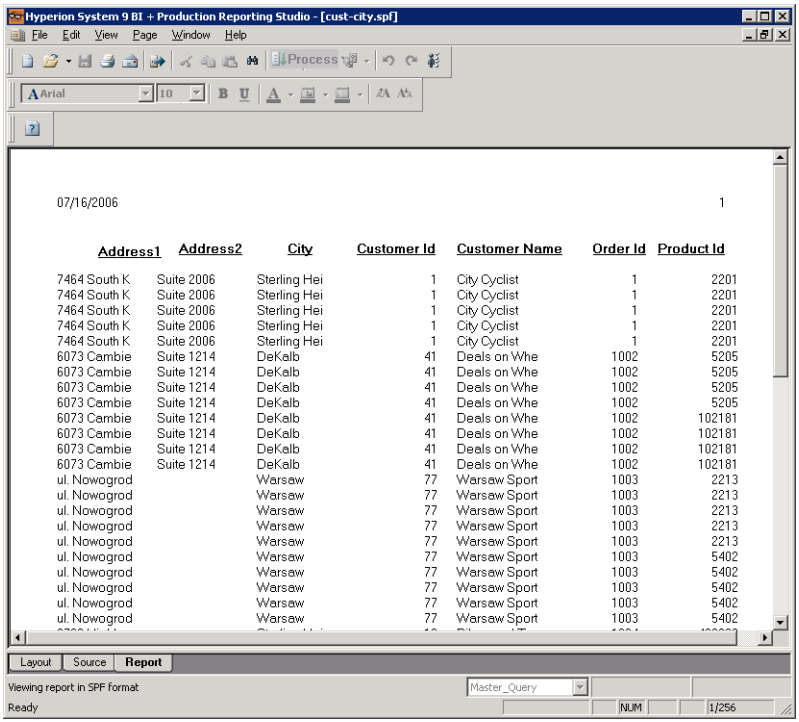
4. Follow the instructions to create the new data connection and select ODBC and the SQR database. Specify the user name and password for authentication.

5. From the Query Builder - Tables dialog box, select the tables that you want to use to generate the report and click Next.

6. Select the query fields, which you want to use to generate the report and follow the instructions in the Query Builder configuration.

7. Click Finish and a layout of the report is displayed. Now, run the report by clicking Process and save the report. The report is displayed as shown in Figure 5-16.

Figure 5-16 Report in Hyperion Production Reporting Studio



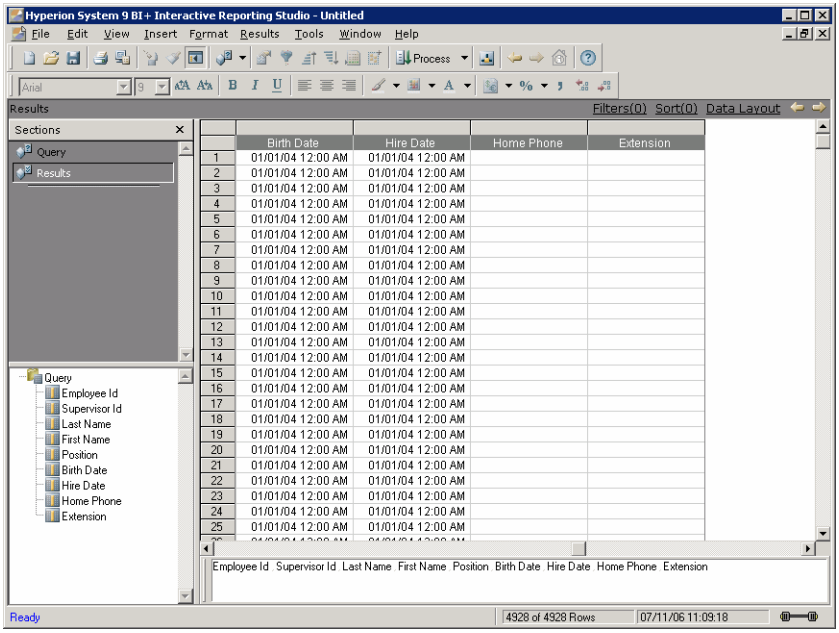
Using Hyperion Interactive Reporting Studio

To generate reports using Interactive Reporting Studio, perform the following steps:

1. Open Interactive Reporting Studio and select to create a new database connection. Specify ODBC as the type of connection and the database.

2. Select Easysoft or Openlink as the bridge and specify the credentials to connect to the data source using the Database Connection Wizard. The rest of the steps to create a new connection are the same as followed in production reports. After you create the connection, a blank layout is displayed.
3. Add the tables to the query area and then drag and drop the columns for which you want to retrieve the data in the Requests field. You can also set the filter for the query using the Filter field.
4. Run the report by clicking Process Current as shown in Figure 5-17, and then save the report.

Figure 5-17 Report in Hyperion Interactive Reporting Studio



Microsoft Access 2003-ODBC

This section describes the procedure to connect Microsoft Access 2003 to AquaLogic Data Services Platform through an ODBC-JDBC bridge. It includes the following topics:

- Limitations and Usage Notes
- Generating Reports Using MS Access

Limitations and Usage Notes

- Microsoft Jet database engine, shipped with MS-Access, maps `SQL_DECIMAL` and `SQL_NUMERIC` fields to the closest Jet numeric data type depending upon the precision and scale of the ODBC field. In certain cases this results in mapping to a non-exact (floating point) numeric Jet data type, such as `Double` or a Text field. For details, refer to the following Microsoft article:

  `http://support.microsoft.com/kb/214854/en-us`

  This implicit type conversion by MS Access causes some errors when retrieving data from AquaLogic Data Services Platform using MS Access.

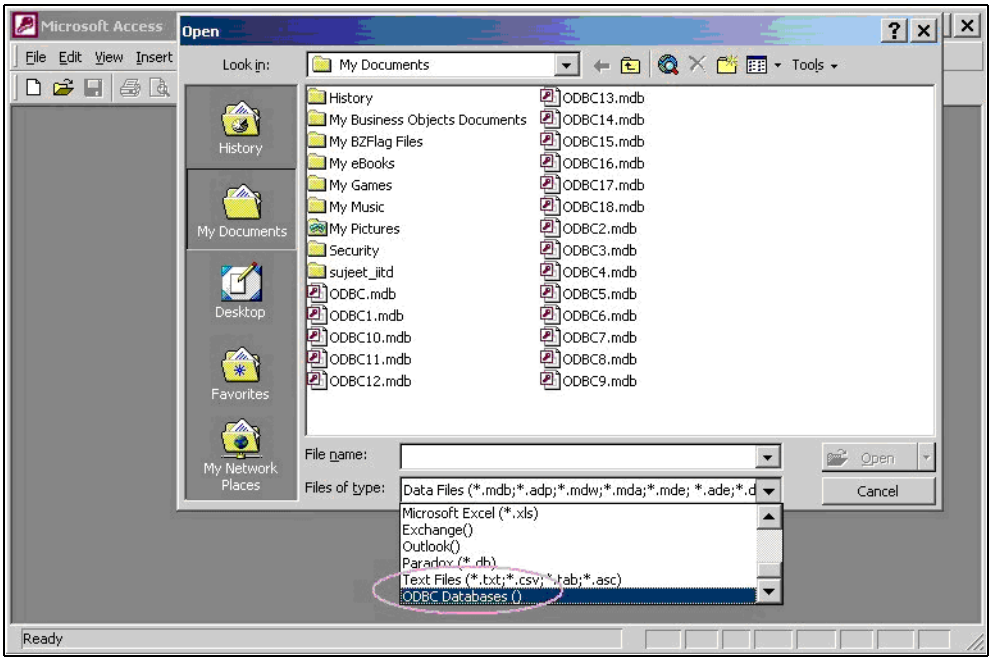
- In MS Access, to sort data retrieved from AquaLogic Data Services Platform, you need to select a Unique Record Identifier when you link tables imported from AquaLogic Data Services Platform with MS Access. If you do not select the Unique Record Identifier then an exception will occur when you try to sort data.

Generating Reports Using MS Access

To connect MS Access to the bridge, perform the following steps.

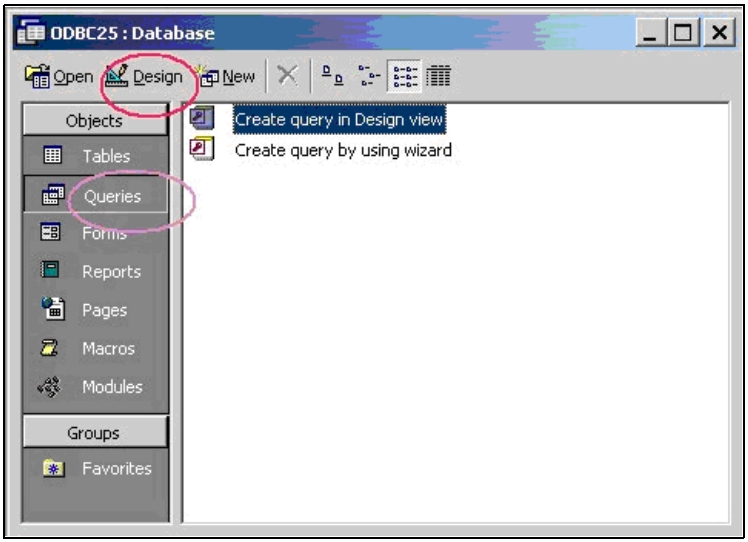
1. Run MS Access, click File→Open, then select ODBC Databases as the file type as shown in the Figure 5-18.

Figure 5-18 Selecting the ODBC Database in Access



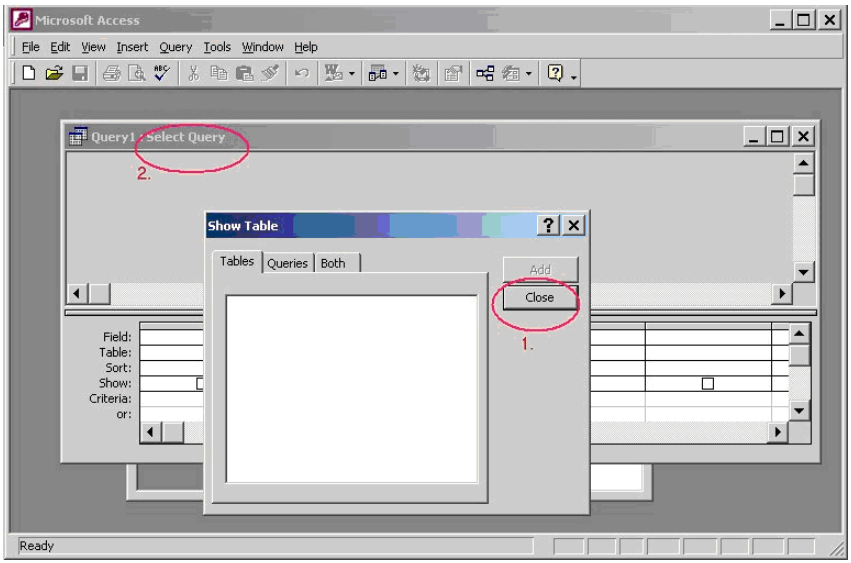
2. Once the dialog Select Data Source pops up, click Cancel to close it. You should see the window shown in Figure 5-19.

Figure 5-19 OBDC23: Database Screen



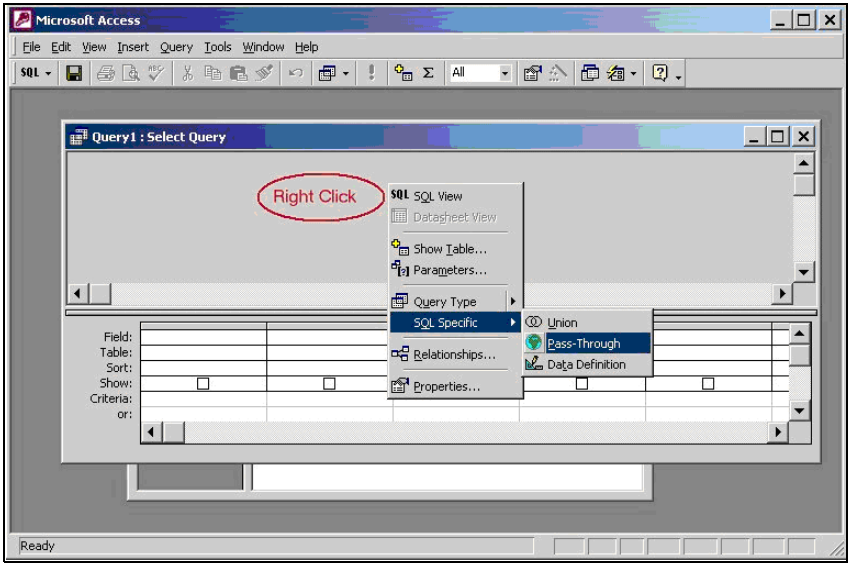
3. Click Queries, then Design as indicated in Figure 5-19. You should see a screen as shown in Figure 5-20.

Figure 5-20 Select Query and Show Table Screens



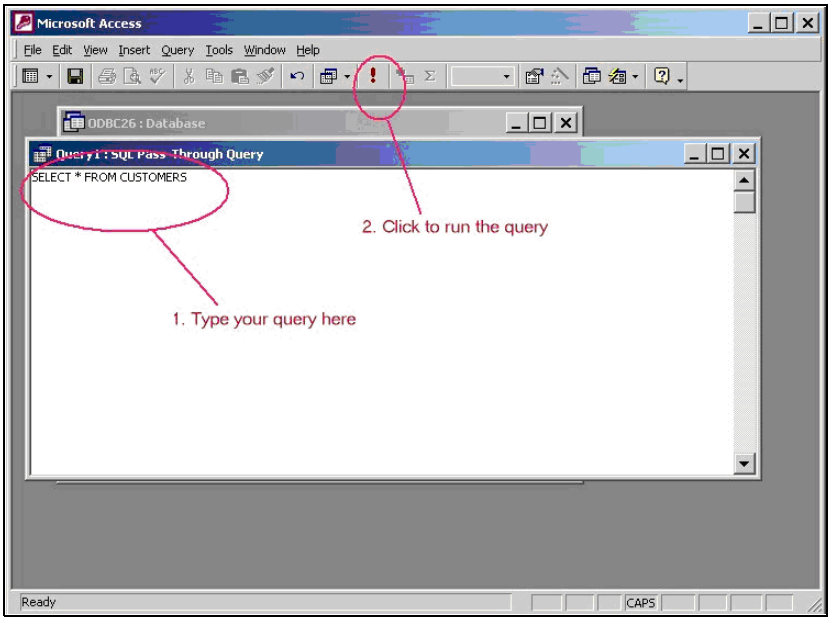
4. Close the Show Table dialog box. You should now be able to see the Select Query window.
5. Right-click in the window and select SQL Specific→Pass-Through as shown in Figure 5-21.

Figure 5-21 Selecting SQL Specific and Pass Through



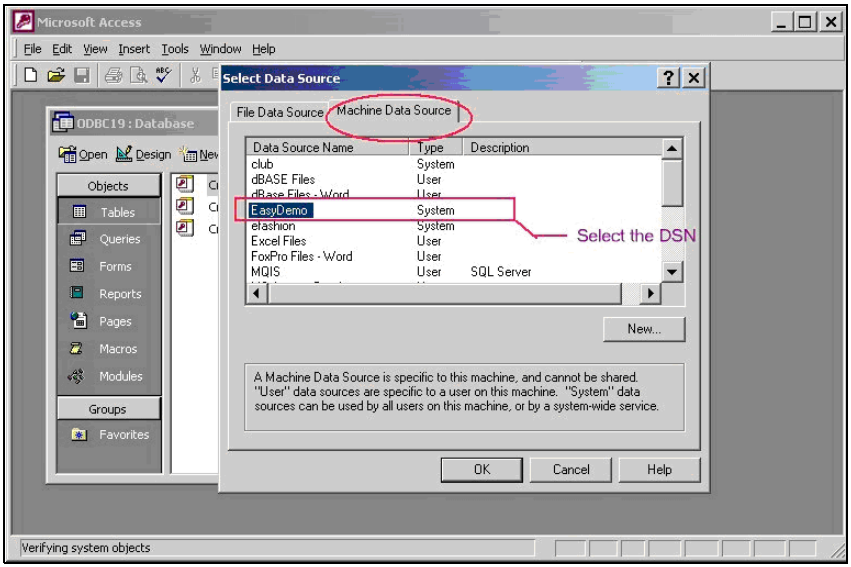
6. Type in your SQL query and click Run, as shown in the Figure 5-22.

Figure 5-22 Running the SQL Query



7. In the dialog box that is displayed (as shown in Figure 5-23), move to the Machine Data Source tab and select **openlink-aldsp** to connect to AquaLogic Data Services Platform JDBC driver and generate the report.

Figure 5-23 Selecting the DSN for the Database



Microsoft Excel 2003-ODBC

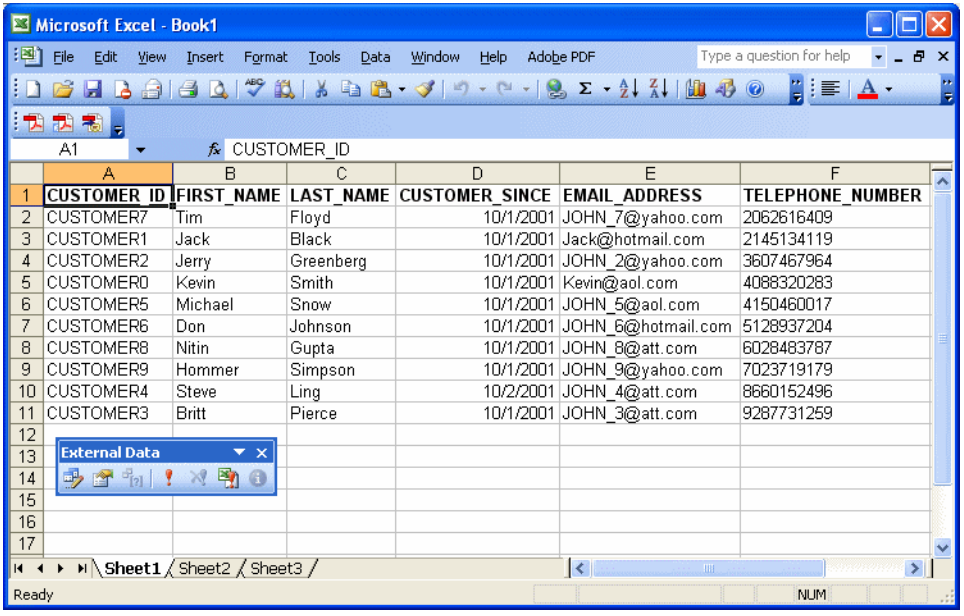
This section describes the procedure for connecting Microsoft Excel 2003 to AquaLogic Data Services Platform through an ODJB-JDBC bridge using EasySoft.

To connect MS Excel to AquaLogic Data Services Platform, perform the following steps:

1. Start Workshop for WebLogic and then start WebLogic Server.
2. Build and deploy the AquaLogic Data Services Platform application.
3. Start Microsoft Excel and open a new worksheet.
4. Click Data→Import External Data→New Database Query. The Choose Data Source dialog box is displayed.
5. Select EasyDemo from the list of data sources and then click OK. The Query Wizard - Choose Columns dialog box is displayed. For details on configuring the JDBC driver using EasySoft, refer to "Using the EasySoft ODBC-JDBC Bridge" on page 5-29.
6. Select the tables for which you want to generate the report and click Next.
7. Follow the Query Wizard instructions and in the Query Wizard - Finish dialog box, select Return Data to Microsoft Office Excel.

8. Click Finish and import the data in a new MS Excel spreadsheet. The query results will be displayed in the spreadsheet as shown in Figure 5-24.

Figure 5-24 Query Results Displayed in MS Excel



	A	B	C	D	E	F
1	CUSTOMER_ID	FIRST_NAME	LAST_NAME	CUSTOMER_SINCE	EMAIL_ADDRESS	TELEPHONE_NUMBER
2	CUSTOMER7	Tim	Floyd	10/1/2001	JOHN_7@yahoo.com	2062616409
3	CUSTOMER1	Jack	Black	10/1/2001	Jack@hotmail.com	2145134119
4	CUSTOMER2	Jerry	Greenberg	10/1/2001	JOHN_2@yahoo.com	3607467964
5	CUSTOMER0	Kevin	Smith	10/1/2001	Kevin@aol.com	4088320283
6	CUSTOMER5	Michael	Snow	10/1/2001	JOHN_5@aol.com	4150460017
7	CUSTOMER6	Don	Johnson	10/1/2001	JOHN_6@hotmail.com	5128937204
8	CUSTOMER8	Nitin	Gupta	10/1/2001	JOHN_8@att.com	6028483787
9	CUSTOMER9	Hommer	Simpson	10/1/2001	JOHN_9@yahoo.com	7023719179
10	CUSTOMER4	Steve	Ling	10/2/2001	JOHN_4@att.com	8660152496
11	CUSTOMER3	Britt	Pierce	10/1/2001	JOHN_3@att.com	9287731259

Limitations

When passing a generated SQL string to Excel, there are situations where Excel inserts single quotes around an alias, resulting in an exception from the AquaLogic Data Services Platform JDBC driver. Here is an example:

```
SELECT Sum(CREDIT.AMOUNT) AS 'Sum of AMOUNT' FROM Xtreme.CREDIT CREDIT
```

Although you can edit your query post-generation, another option is to install a patch from Microsoft which is designed to address the problem. The current URL for accessing information on this problem and patch is:

http://support.microsoft.com/kb/298955/en-us

Using the Query Plan Viewer Utility

You can review the plan for the execution of a SQL query or XQuery using a standalone Query Plan Viewer utility. The functionality is similar to that described in "Using Query Plan View" in the Testing Query Plan Functions and Viewing Query Plans chapter of *Data Services Developer's Guide*.

Note: In order to use this utility some components must be installed and an authorized log-in to an AquaLogic Data Services Platform-enabled application provided.

Installing Query Plan Utility Components

In the absence of a full installation of AquaLogic Data Services Platform, a specific set of component files must be installed to run the Query Plan Viewer utility. Default locations for these are assumed, but these default can be easily modified.

Table 5-10 Default Location of Query Plan Utility Component Files

File	Default location
• `aldspqpv.cmd` (windows) • `aldspqpv.sh` (unix/linux)	`/liquiddata/bin`
• `aldspqpv.jar` • `ldjdbc.jar`	*<weblogic_home>*`/liquiddata/lib`
• `wlclient.jar`	*<weblogic_home>*`/server/lib`

You can adjust the default location settings by editing the `aldspqpv.cmd` (Windows) or `aldspqpv.sh` (unix/linux) files.

Command Line Syntax

The command line syntax for the Query Plan Viewer utility is:

```
Java -classpath
[path1/]wlclient.jar;[path2/]ldjdbc.jar;[path3/]aldspqpv.jar

   com.bea.dsp.client.ui.shell.Shell

   [Server host name][Server port number][Application name][User
id][Password]
```

Here is an example using default settings:

```
Java -classpath
../../server/lib/wlclient.jar;../lib/ldjdbc.jar;../lib/aldspqpv.jar


   com.bea.dsp.client.ui.shell.Shell

   localhost 7001 RTLApp weblogic weblogic
```

Invoking the Query Plan Viewer Utility

Once your classpath is properly set you can invoke the Query Plan Viewer utility from the command-line using either:

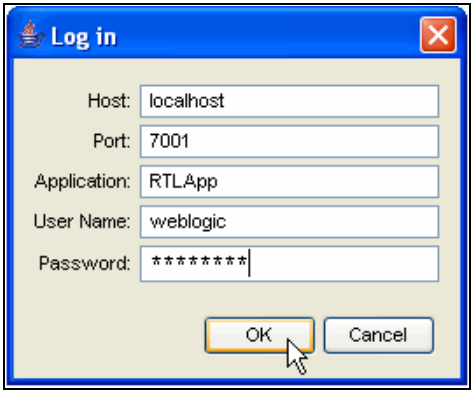
- `aldspqpv.cmd` (windows)
- `aldspqpv.sh` (unix/linux)

On Windows systems you can easily associate the invocation command by associating a start menu option with `aldspqpv.cmd`.

Query Plan Viewer Login Dialog

Before you can view query plans, you need to log in to an AquaLogic Data Services Platform-enabled server (Figure 5-25).

Figure 5-25 Query Plan Viewer Login Dialog



In the dialog enter the following information:

- **Host.** This is the server host name. Localhost is only appropriate if the Query Plan Viewer is running on your local server. Otherwise enter your server's URI.
- **Port.** This is the server port number. For localhost the default port number is 7001.

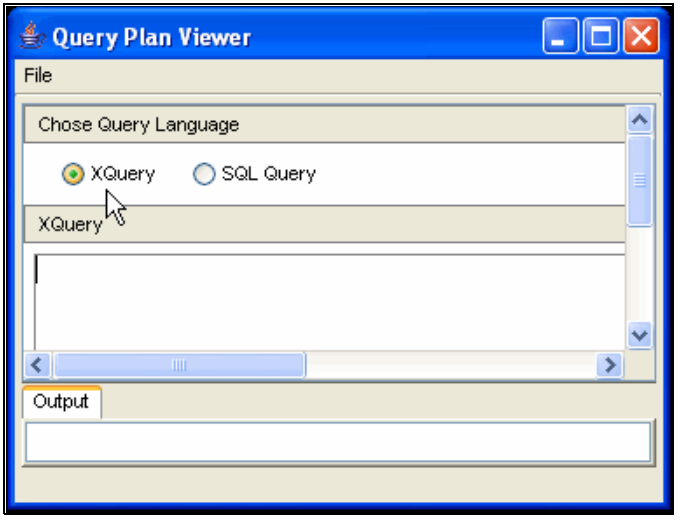
- **Application name.** This is the case-sensitive name of your AquaLogic Data Services Platform-enabled application.
- **User name.** This is the name of the authorized user.
- **Password.** This is the password of the authorized user.

Once you have logged into the Query Plan Viewer, all information but the password is saved on your system.

Entering an SQL Query or XQuery

Once you login to your server successfully, you are ready to enter a query by selecting whether to enter either an SQL query or an XQuery.

Figure 5-26 Selecting a Query Type



Note: Functionally choosing XQuery or SQL Query is equivalent to selecting Ad hoc XQuery or Ad hoc SQL Query in Query Plan View in the data service development environment.

For details on entering an XQuery or SQL Query and working with the resulting query plan, see Creating Ad Hoc Queries in the *Data Services Developer's Guide*.

Additional Query Plan Viewer Utility Options

The utility offers several options, available from its File menu. The options are described in Table 5-11.

Table 5-11 Query Plan Viewer Utility Options

Option	Usage
Open Query Plan from File	This option opens a dialog where you can enter the name of a query plan that has been previously saved.
Connect	Opens or reopens the Query Plan Viewer connection dialog box.
Disconnect	Disconnect the current server session.
Print	Opens a standard dialog box that allows you to print the current query plan.
Save to File	Opens a standard dialog box that allows you to save the current query to an XML format file.
Exit	Quits the Query Plan Viewer utility.

CHAPTER 6

Using Excel to Access Data Services

Using the AquaLogic Data Services Platform™ Excel Add-in you can access data from data services in Microsoft® Excel® spreadsheets. This has many advantages:

- Data service integration of real-time data can be quickly rendered in the familiar Microsoft Excel format.
- Inclusion of Web service results in worksheets without programming; just drag-and-drop.
- Flexibility and scalability through the BEA AquaLogic Service Registry.
- Extensible development platform with access to the Excel Add-in API.

This chapter provides a brief overview of the AquaLogic Data Services Platform Excel Add-in, focusing on:

- Installing the Excel Add-in
- Generating WSDL Files for the Excel Add-in
- Accessing the Excel Add-in Documentation

Installing the Excel Add-in

This section describes installation of the Excel Add-in.

System Requirements

The following system requirements have been identified:

- Windows XP (tested under Service Pack 2).
- Disk space: 43MB (6 MB if Microsoft .NET Framework with SP1 is already installed).
- RAM: 256K MB or more recommended.
- Microsoft Excel 2002 (SP2) or Excel 2003 (SP1).

Installation Instructions

The AquaLogic Data Services Platform Excel Add-in is included in with AquaLogic Data Services Platform installation as a separate installation executable. You can either directly install the executable or save the executable and install it from your desktop. The program installs on your local machine and also is accessible from Excel directly as a menu item.

Preparing To Install

Prior to installing the Excel Add-in, please be sure to uninstall any previous versions of the program. Refer to Uninstalling ALDSP Excel Add-in for instructions on how to complete this task.

Note: Microsoft .NET with SP1 is required for the ALDSP Excel Add-in installation If Microsoft If .NET Framework is not installed on your system, or you have .NET 1.1 or earlier without SP1, the Excel Add-in prompts you to confirm its installation from the Microsoft Web site. Once the .NET install is complete, the system proceeds with the ALDSP Excel Add-in installation. (The installation of .NET could take up to 10 minutes for download and configuration.)

To install AquaLogic Data Services Platform Excel Add-in:

1. Locate the Excel plug-in installation file. In can be found in following directory:

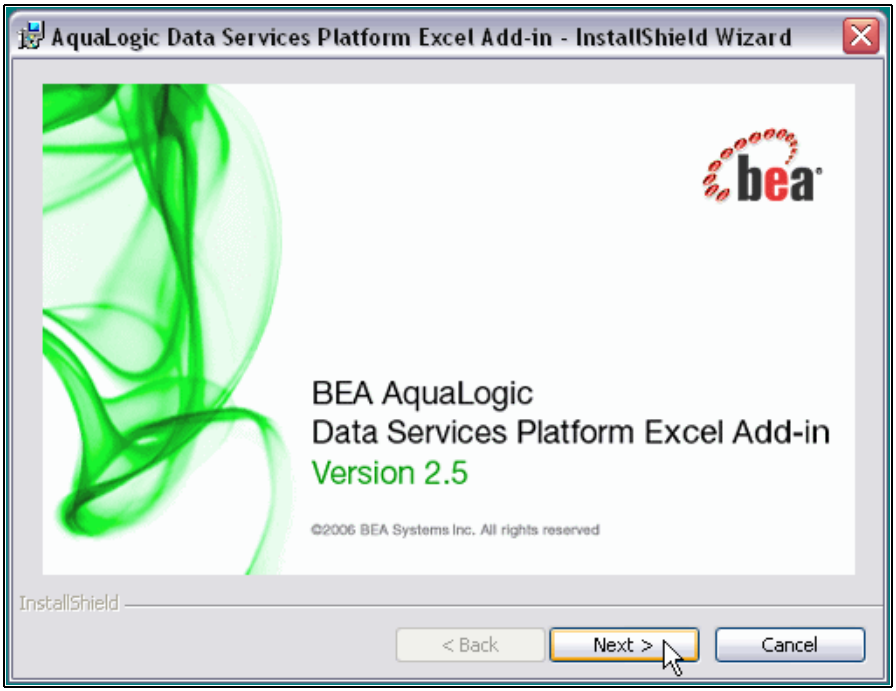
   `<bea_home>/weblogic81/liquiddata/add-in`

   where <bea_home> is the BEA installation on your system.

2. Double-click on the installation file:

   `aldsp_excel_addin_250_win32.exe`

Figure 6-1 Excel Add-in Installation

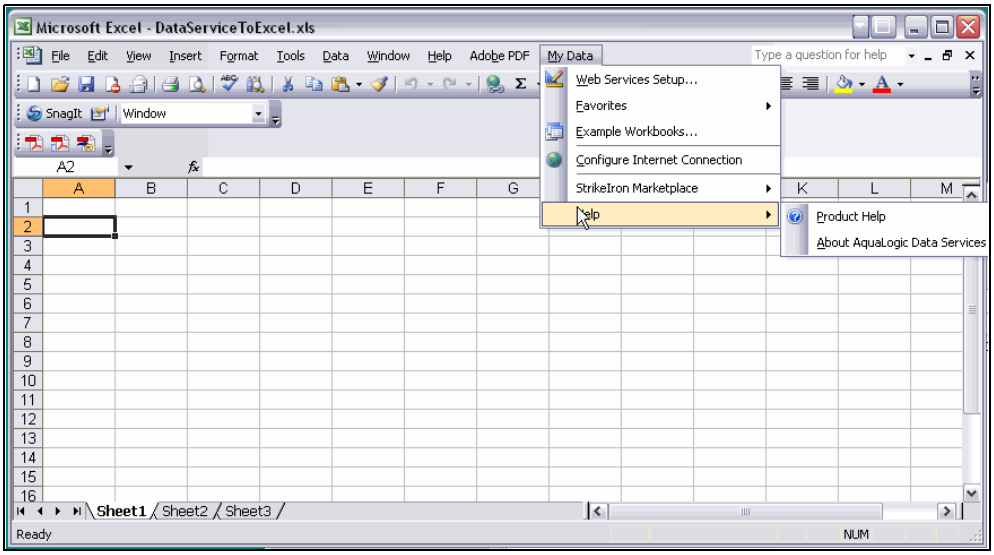


3. Progress through the installation program using standard Next buttons.
4. If you do not already have Microsoft .NET Framework 1.1 installed on your system, you will need to install it. This can be done through the Excel Add-in installation dialog.
5. Determine the user of the application. Typically anyone with access to your system would be able to use the Add-in.
6. Determine the location of the Add-in. By default the Add-in is installed in the following directory:

   ```
   C:\Program Files\BEA\AquaLogic Data Services Platform Excel Add-in
   ```

7. Complete the installation, optionally launching Excel at the end of the process.

Figure 6-2 Post-Installation of the Excel Add-in



Accessing the Excel Add-in Documentation

Once you have completed the installation, you will be able to obtain the Excel Add-in documentation:

```
AquaLogic Data Excel User Guide v2.pdf
```

by default located in the following directory:

```
C:\Program Files\BEA\AquaLogic Data Services Platform Excel Add-in\Documentation
```

Select Run to install directly to your local machine or

a. Select Save to save the installation package to your local machine and install from that location.

The documentation includes the following major topics:

- Installation and uninstallation of the Excel add-in
- Using the Add-in
- Managing Web Services
- Refreshing Web Service Data in Excel
- Troubleshooting

Tip: Information on using the Add-in is also available post-installation from the Excel menu option (Figure 6-2).

Generating WSDL Files for the Excel Add-in

Before you access data services through the Excel Add-in, you need to generate a Web service WSDL file. The process is simple, but there are a few important things to keep in mind. The basic steps are:

- Create a Data Service control containing the functions you want to access from Excel.
- Create a stateless Java Web service (.jws) from the control.
- Generate a WSDL file. This file contains the necessary URL to allow access to your data service.

Creating a WSDL File from a Data Service

Information on generating Web services, including WSDL files, from data services can be found in several locations:

- Chapter 4, "Enabling AquaLogic Data Services Applications for Web Service Clients."
- Samples Tutorial Part I, available from the AquaLogic Data Services Platform edocs page:

  `http://edocs.bea.com/aldsp/docs25/index.html`

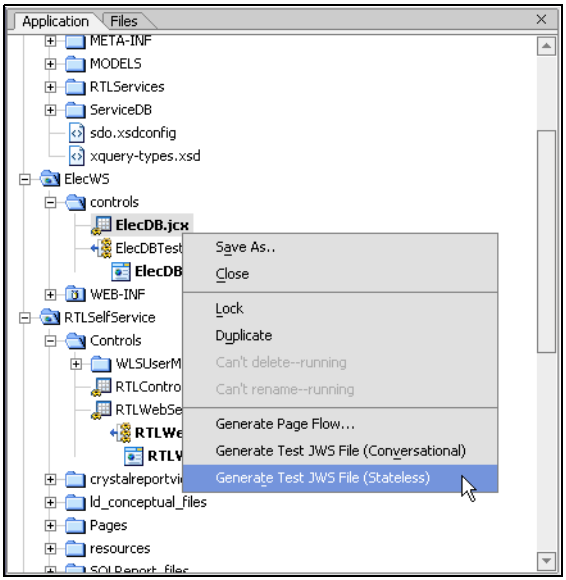
  contains a tutorial entitled, "Accessing Data in Web Services."

Note: Be sure and review this material so that you can properly generate a data service-compatible Data service control, Java Web service, and WSDL file for the Excel Add-in.

The RTLApp contains a Web service called ElecWS that you can use to test the process. Once the control is created as explained in "Enabling AquaLogic Data Services Applications for Web Service Clients," you can quickly generated a compliant WSDL file by following these steps:

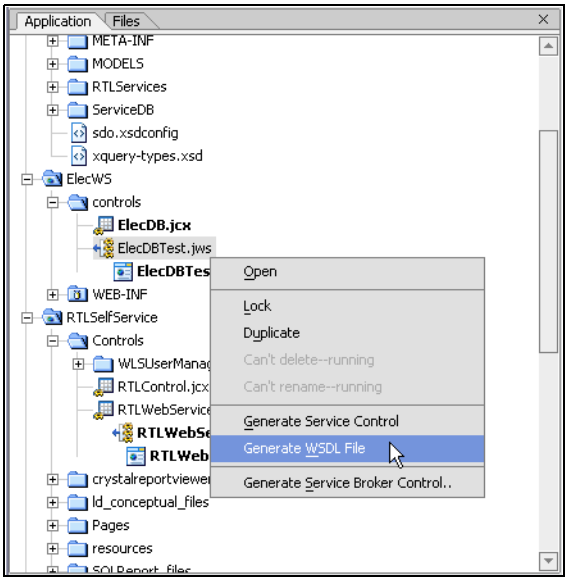
1. Right-click on the JCX file, selecting Generate Test JWS File (Stateless), as shown in Figure 6-3.

Figure 6-3 Generating a JWS File in the RTLApp



2. Right-click on your newly generated JWS file, selecting the Generate WSDL File option (Figure 6-4).

Figure 6-4 Generating a WSDL File



Note: It is recommended that you test and verify your JWS file before attempting to use the WSDL address in the Excel Add-in.

Obtaining a Valid WSDL URL for Use with the Excel Add-in

The URL used to access your data service must:

- be based on a stateless JWS file.
- be generated using the Generate WSDL File menu option, as previously described.
- point directly to the generated WSDL file

An easy way to obtain the first portion address of your generated Web service is to run your JWS file in the WebLogic Workshop Test Browser. Simply open the Web service and click the Run icon or choose Start from the Debug menu.

Figure 6-5 RTLApp ElecDBTest Web Service in Test Browser

In the above case, the URL is composed of the following elements:

- Hostname. In the case of the RTLApp sample, the host name is:

  `localhost`

- Port. In the case of the RTLApp, the port is:

  `7001`

- Path to the location of the WSDL file. In the case of the ElecWS Web service, the path is:

  `ElecWS/controls/ElecDBTest`

Then, you need to substitute the full name of your WSDL file for the rest of the address. In this example you would substitute:

```
ElecDBTestContract.wsdl
```

for:

```
ElecDBTest.jws?.EXPLORE=.TEST
```

Thus the full path to the WSDL in your sample that you can use in the Excel Add-in should be:

```
http://localhost:7001/ElecWS/controls/ElecDBTestContract.wsdl
```

Using the Excel Add-in with a Remote or Deployed Server

If you plan to access your data through the Excel Add-in using a server other than your local development machine you need to do the following:

1. Take note of the current hostname and port settings.
2. Set the hostname and port in the Weblogic Workshop Tools -> WebLogic Server -> Properties to reflect the address of the server you intend to use.
3. Regenerate your WSDL file as described in Chapter 4, "Enabling AquaLogic Data Services Applications for Web Service Clients."
4. Create an EAR file for deployment. This will establish the address of your WSDL to the currently set hostname and port settings. For additional information on creating EAR files for deployment see Deploying AquaLogic Data Service Platform Applications in the *Administration Guide*.

CHAPTER 7

Accessing Data Services from WebLogic Workshop Applications

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:

- Introduction to Data Service Controls
- Creating Data Service Controls
- Modifying Existing Data Service Controls
- Caching Considerations When Using Data Service Controls
- Security Considerations When Using Data Service Controls
- Using Data Services Platform with NetUI
- Using Data Service Control 9.2

Introduction to Data Service Controls

Data Service controls provide WebLogic Workshop applications with an easy way to access data services. When you use a Data Service control to invoke data service functions, 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 1, "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 SDO Mediator API, including:

- Function result filtering
- Ad hoc query capability
- APIs for result ordering, sorting, and truncating

For more information on these features, see Chapter 11, "Advanced Topics."

Data Service Controls Defined

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. AquaLogic Data Services Platform generates interface files for the target schemas associated with the queries and the Data Service control (`.jcx`) file.

For AquaLogic Data Services Platform 2.5 release, the Data Service control is available with both WebLogic Workshop 8.*x* and Workshop for WebLogic Platform 9.2. The Workshop for WebLogic Platform 9.2 Data Service control allows you to access data services from the WebLogic Server 8.1 environment. For more information about installing and using the Data Service control 9.2, refer to "Using Data Service Control 9.2" on page 7-28.

Note: All client APIs, including the Data Service control, support calling data service functions without optional parameters. Data service functions with optional parameters can be called within other data service functions or from an ad hoc query, but such functions cannot be invoked from through a Data Service control itself.

Page Flow, Web Services, Portals, Business Processes

Like Java controls, 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 query functions in the control in your application.

This chapter describes 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.

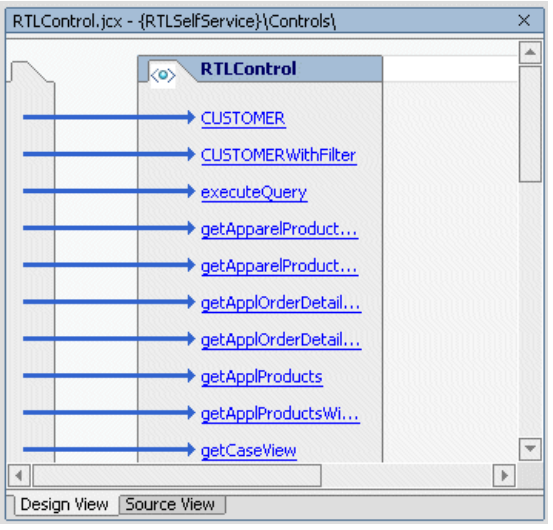
Description of the Data Service Control (JCX) File

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.

Design View

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

Figure 7-1 Design View of a JCX File



Using the right-click menu, you can add or edit 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. 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 7-6.

Listing 7-1 shows portions of a generated Data Service control (`.jcx`) file. It shows the package declaration, import statements, and data service URI used with the queries.

Listing 7-1 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.PROFILEDocument[]
rootDataObject) throws Exception;

    /**
     *
     * @jc:XDS 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="CUSTOMER"
     */
    dataServices.customerDB.customer.CUSTOMERDocument[] CUSTOMER();

/**

< section of code removed >

*/

    /**
     *
     * @jc:XDS functionURI="ld:DataServices/RTLServices/ProfileView"
functionName="getProfileView" schemaURI="urn:retailer"
schemaRootElement="PROFILE"
     */
    retailer.PROFILEDocument[] 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);


    void newMethod1();
 }
```

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 creating an ad hoc query. To create 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, an 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 involved in 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
- 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

Make sure that the WebLogic Server that hosts the AquaLogic Data Services Platform-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).

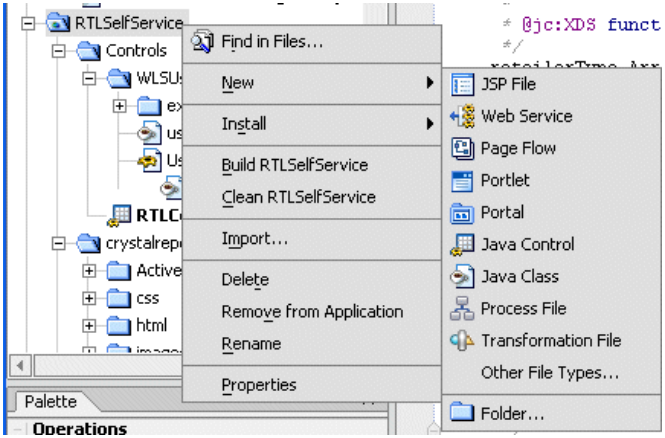
Step 3: Create a Folder in a Project

Create and name 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.)

Step 4: Create the Data Service Control

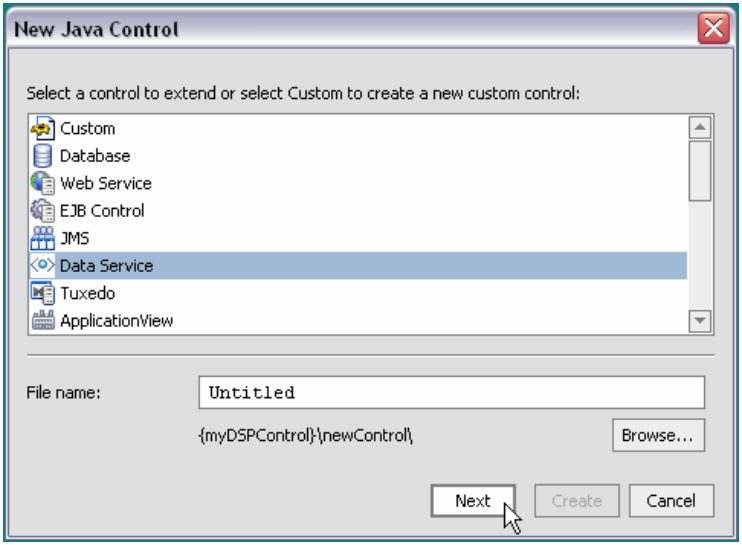
To create a Data Service control, start the Java Control Wizard by right-clicking on the new folder in your project and choosing New → Java Control as shown in Figure 7-2. (You can also create a control using the File → New → Java Control menu item.)

Figure 7-2 Create a New Data Service Control



Next, select Data Services Platform from the New Java Control dialog as shown in Figure 7-3. Enter a filename for the control (`.jcx`) file and click Next.

Figure 7-3 New Java Control Dialog



Step 5: Enter Connection Information for WebLogic Server

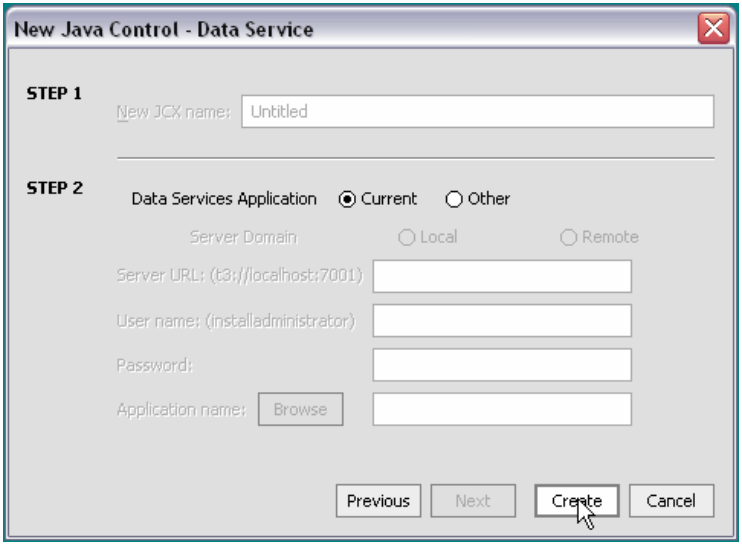
The New Java Control - AquaLogic Data Services Platform dialog (Figure 7-4) allows you to enter connection information for the WebLogic Server that hosts your Data Services Platform application or project. If the server is local, a Data Service control uses the connection information stored in the application 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 7-15.

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

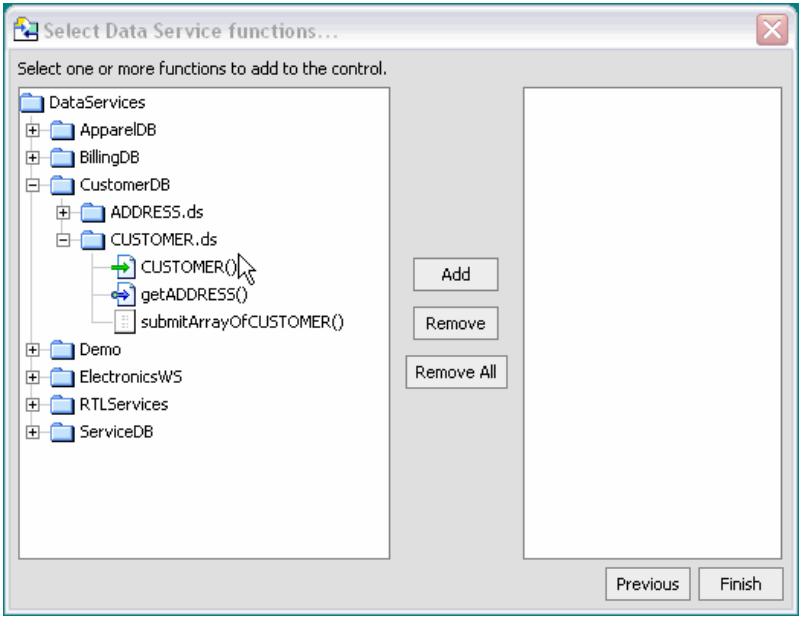
Figure 7-4 Data Service Control Wizard: Connection Information



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 (Figure 7-5). When done, click Finish. At that point, your Data Service control JCX file is generated, containing a call for each selected function.

Figure 7-5 Control Wizard: Select Data Service Functions Dialog Box



The control appears with the functions you chose.

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.

If not already present, the `LiquidDataControl.jar` file is copied into the Libraries directory of your application when you create your Data Service control.

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:

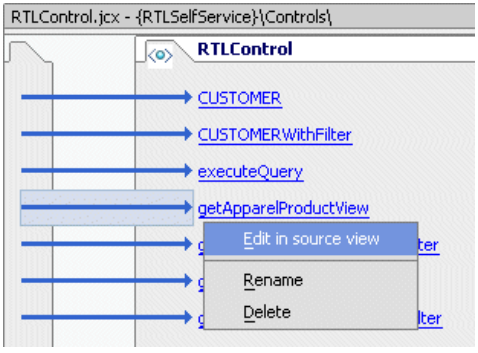
- Changing a Method Used by a Control
- Adding a New Method to a Control
- Updating an Existing Control When Schemas Change

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 (Figure 7-6.).

Figure 7-6 Changing a Function in a Data Service Control



3. In Source View, change the comment for the function in the following ways:
   - 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.
   - Change the return type, parameters, and name of the function, as needed.

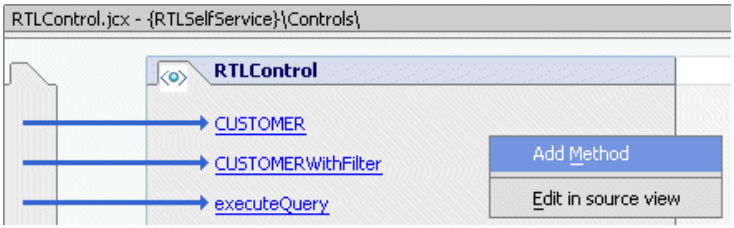
When you save your changes, the SDO classes based on the control are automatically recompiled.

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 7-7.

Figure 7-7 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 AquaLogic Data Services Platform data service folders to regenerate the SDO classes. 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 (mediator API or control) and you modify any schemas, you must recompile and redeploy the application, 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.

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 7-2 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 7-2 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 7-2, 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 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 7-4).

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 AquaLogic Data Services Platform 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

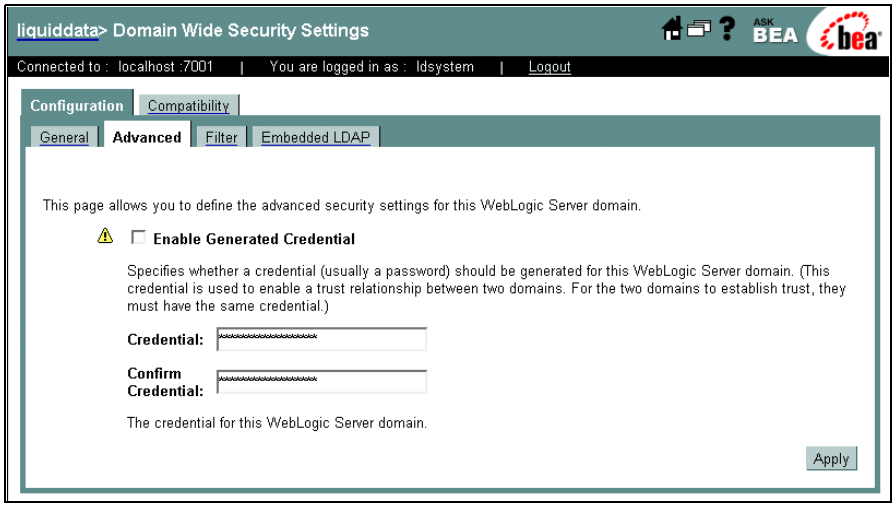
To configure domains as a trusted user, perform the following steps:

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 7-8.)

Figure 7-8 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 security, see:

- "Securing AquaLogic Data Services Platform Resources" in the *Administration Guide*.

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

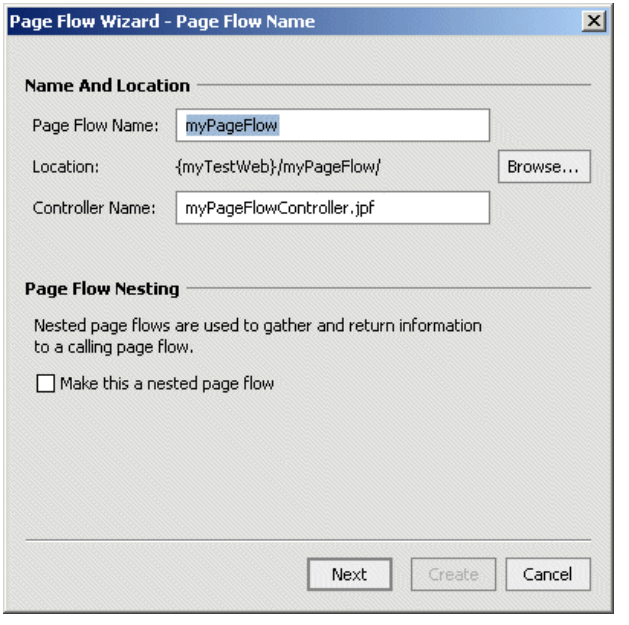
When you use WebLogic Workshop to generate a page flow, WebLogic Workshop creates the page flow, a start page (`index.jsp`), and a JSP file and action for each method you specify in the Page Flow wizard.

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.

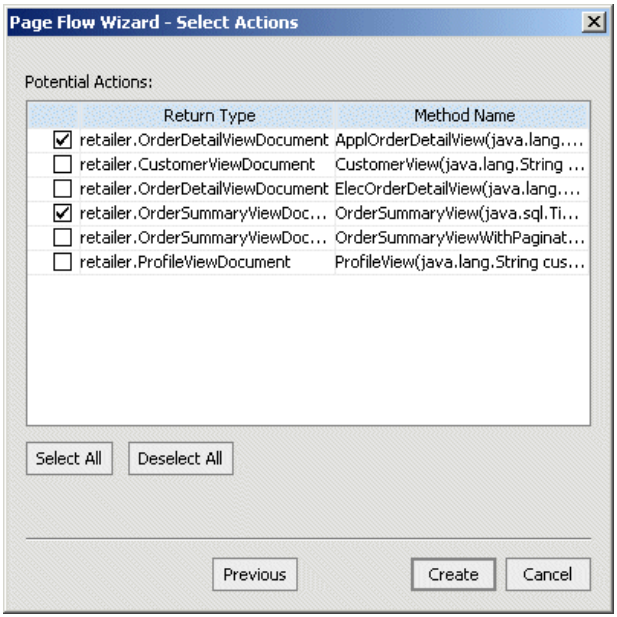
1. Select a Data Services Platform control JCX file from the application file browser, right-click, and select Generate Page Flow.
2. In the Page Flow Wizard (see Figure 7-9), enter a name for your Page Flow and click Next.

Figure 7-9 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 7-10.)

Figure 7-10 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 7-20.

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 7-24.

7. Build and test the application in WebLogic Workshop.

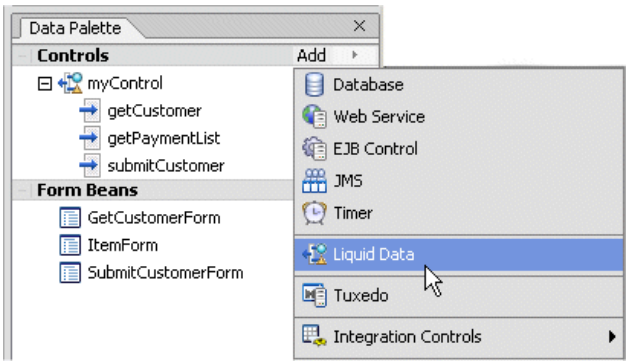
Adding a Data Service Control to an Existing Page Flow

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, "Enabling AquaLogic Data Services Applications for Web

Service Clients." 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 7-11.

Figure 7-11 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 1, "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.

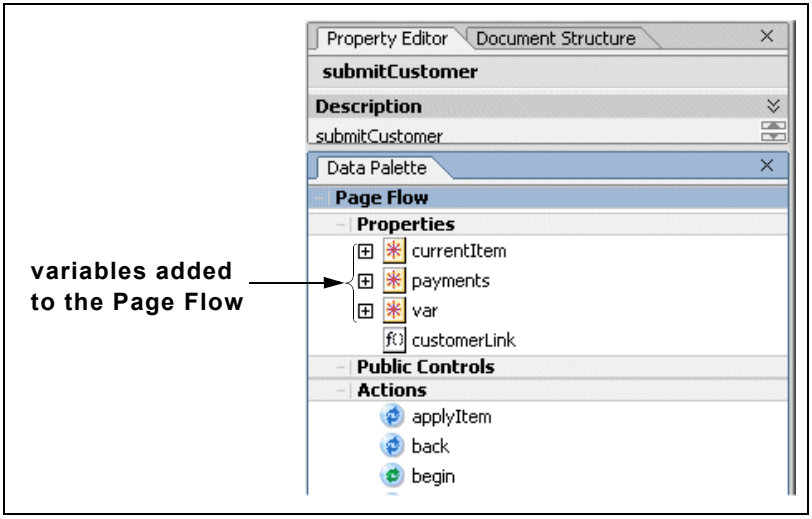
Define each variable with a type corresponding to an SDO object. Define the variables in the source view of the page flow controller class. The variables should be declared public. In the following example, the bold-typed variable declarations show an example of user variable declarations:

```
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 7-12.)

Figure 7-12 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 "Initializing a Variable in the Page Flow" on page 7-22.

Adding 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.

Initializing a Variable in the Page Flow

You can initialize a 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 7-18).

```
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.

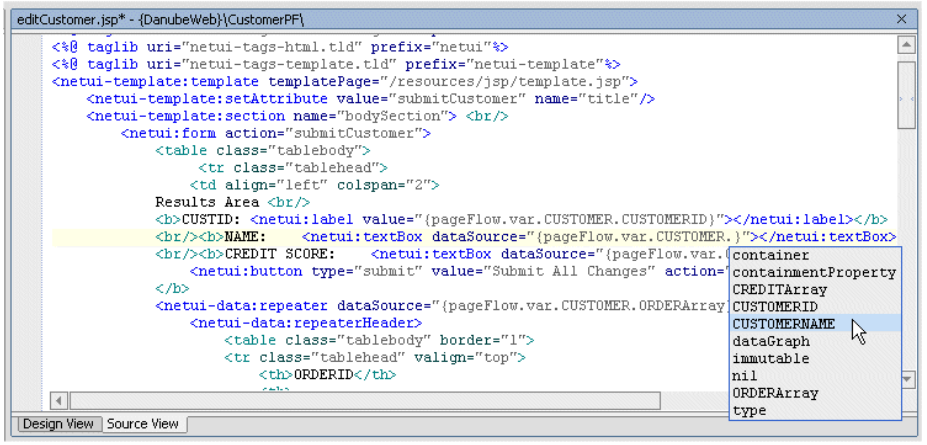
The elements appear in dot-delimited chain format, such as:

```
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 7-13.

Figure 7-13 DataObject Method Name Completion



Note: For more information on programming with AquaLogic Data Services Platform data objects, see Chapter 1, "Data Programming Model and Update Framework."

Displaying Array Values in a Table or List

AquaLogic Data Services Platform 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:

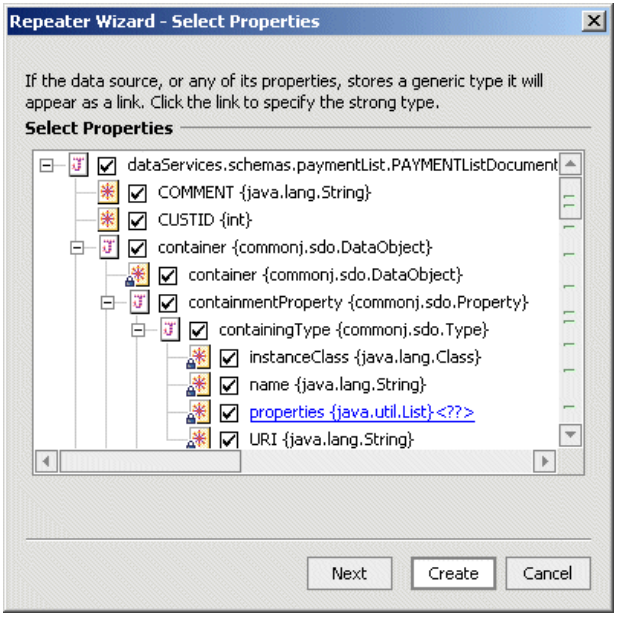
1. Open a JSP file in your Page Flow project where you want to display data. This should be the page corresponding to the action in which the variable is initialized.

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 "Initializing a Variable in the Page Flow" on page 7-22.

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 7-14.

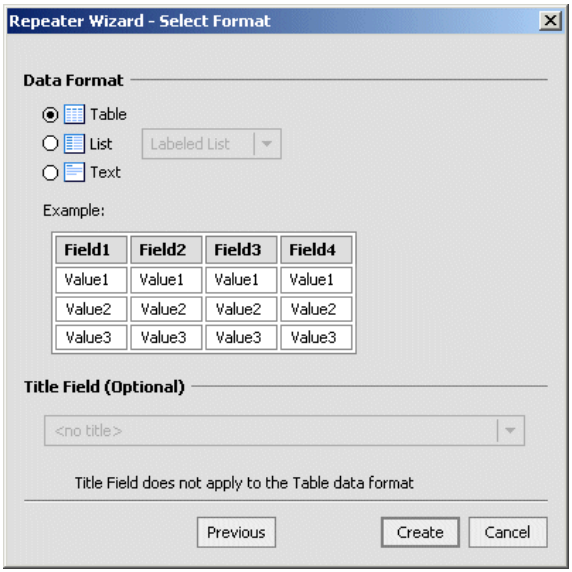
Figure 7-14 Repeater Wizard



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 7-15.

Figure 7-15 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 7-24.
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 7-24.
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 "Initializing a Variable in the Page Flow" on page 7-22.

```
<%
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 your application.

Using Data Service Control 9.2

Data Service control 9.2 is a custom Java control accessible through BEA Workshop for WebLogic Platform 9.2, which allows easy access to data services deployed on WebLogic Server 8.1, from other components of Workshop for WebLogic Platform 9.2 application, such as Java Web Service (JWS).

For AquaLogic Data Services Platform 2.5 release, the control acts as a bridge between the 9.2 based Workshop for WebLogic Platform client and 8.*x* AquaLogic Data Services Platform server.

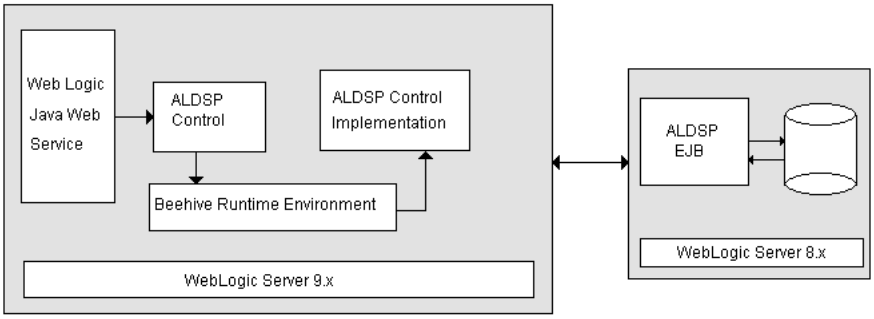
Note: With the 9.2 release, WebLogic Workshop has been renamed to Workshop for WebLogic Platform. In this section, the term Workshop for WebLogic Platform is used to refer to the 9.2 environment.

The Data Service control is based on the open source *beehive control architecture*. Beehive control is an Apache Software Foundation project that provides a programming model for designing business functionality that makes it easy to use lightweight JavaBeans and declarative configuration through JDK 1.5 annotations. For more information about beehive controls, refer to:

http://beehive.apache.org/docs/1.0/controls/programming.html

Figure 7-16 illustrates the Data Service control 9.2 architecture, which is based on the beehive control architecture. The Data Service control implementation files in 9.2 use the beehive runtime environment to interface with WebLogic Server 8.1.

Figure 7-16 AquaLogic Data Services Platform 9.2 Control Architecture



Data Service control 9.2 consists of the core control infrastructure classes and the Wizard classes, which are used to create the Data Service control. This section describes the features of Data Service control 9.2 and provides steps to use it for accessing data services deployed on WebLogic Server 8.1. It includes the following topics:

- Differences Between the 9.2 and 8.1 Data Service Control
- Installing the Data Service Control 9.2 Plug-In
- Using Data Service Control 9.2 from Workshop for WebLogic Platform

Differences Between the 9.2 and 8.1 Data Service Control

The data service control is created in Workshop for WebLogic Platform and is similar to the control available with the AquaLogic Data Services Platform 8.1 environment. However, there are some differences in functionality, which include:

- The Workshop for WebLogic Platform control does not support the .NET type controls.
- The 9.2 control does not create a new schema project for the application, instead it creates a new schema directory in the default directory path, which is set using the XMLBeans Builder preferences. The schemas for the method return types are downloaded from WebLogic Server 8.1 into this directory.
- The control in 9.2 uses the `.java` extension whereas the 8.1 control has the `.jcx` extension.
- The Design View feature is not available with the 9.2 control.
- Local control is not supported.

- The method signature of the general and ad hoc queries in the 9.2 environment is different from the 8.*x* method signature. In the 9.2 environment, the JDK 1.5 annotations are used for general and ad hoc queries. In addition, the return type for ad hoc queries is different in the 9.2 environment.

  Listing 7-3 and Listing 7-4 show sample method signatures used in general and ad hoc queries in the 9.2 environment, respectively.

Listing 7-3 Annotations Used in a General Query in 9.2

```
@DSPControlMethodAnnotation(functionURI =
"ld:DataServices/Demo/Java/Physical/ShipSource1", functionName
="getShipSource1", schemaURI = "http://customJava/ShipSource1",
schemaRootElement = "ShipSource1", locator  =
"ld:DataServices/Demo/Java/Physical/ShipSource1.ds", hasSideEffect =
false)


public customJava.shipSource1.ShipSource1Document getShipSource1() ;
```

Listing 7-4 Annotations Used in an Ad Hoc Query in 9.2

```
@DSPControlAdhocQueryAnnotation(body=<Your Xquery>)

java.lang.Object[] executeQuery(String xquery, ExternalVariables params,
RequestConfig config);
```

Installing the Data Service Control 9.2 Plug-In

Data Service control 9.2 is available as a plug-in with Workshop for WebLogic Platform 9.2. The control is packaged in the `com.bea.dsp.control.wizard.zip` file, which you need to unzip into the following location:

`<BEA_HOME>/workshop92/workshop4WP/eclipse`

After you unzip the file, the `com.bea.dsp.control.wizard_1.0.0` directory is created in the following location:

```
<BEA_HOME>/workshop92/workshop4WP/eclipse/plugins/
```

Setting Up WebLogic Server 8.1 to Use Data Service Control 9.2

For using the Data Service control 9.2, both the 8.1 and 9.2 WebLogic Servers should be running. In addition, you need to configure the classpath of WebLogic Server 8.1 to include `wls90interop.jar`, as follows:

1. Go to `<BEA_HOME>\weblogic81\samples\domains\ldplatform`.
2. Open the `setDomainEnv.cmd` file for editing and search for the `PRE_CLASSPATH` variable.
3. Add the path to the `wls90interop.jar` file to the `PRE_CLASSPATH` variable and save the file. This file is located in the `<BEA_HOME>\liquiddata\lib` directory.

Note: WebLogic Server8.1 is the server where the AquaLogic Data Services Platform application is deployed.

Using Data Service Control 9.2 from Workshop for WebLogic Platform

After installing the control plug-in, you can create the control in Workshop for WebLogic Platform and use it to access data from AquaLogic Data Services Platform. This section provides the steps to create, edit, consume, delete, and uninstall the control. It includes the following topics:

- Creating and Using the Data Service Control
- Modifying and Uninstalling the Control

Creating and Using the Data Service Control

After the control plug-in is installed in the 9.2 environment, it is accessible through Workshop for WebLogic Platform 9.2 environment.

To use the Data Service control, you need to:

1. Create a Web service project.
2. Include a Java package in the project, which in turn contains the Data Service control file.
3. Create a JWS, which will use the Data Service control 9.2 to access data services from WebLogic Server 8.1.

This section describes in more detail the steps needed to create the Web service project and then use the control through JWS.

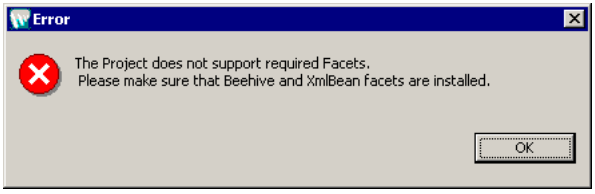
Creating a Web Service Project

The Web service project requires some facets installed for the Data Service control to work, which are:

- XMLBeans
- Beehive Control
- XMLBeans Builder

If these facets are not installed with the Web Service project, then an error message is displayed as shown in Figure 7-17:

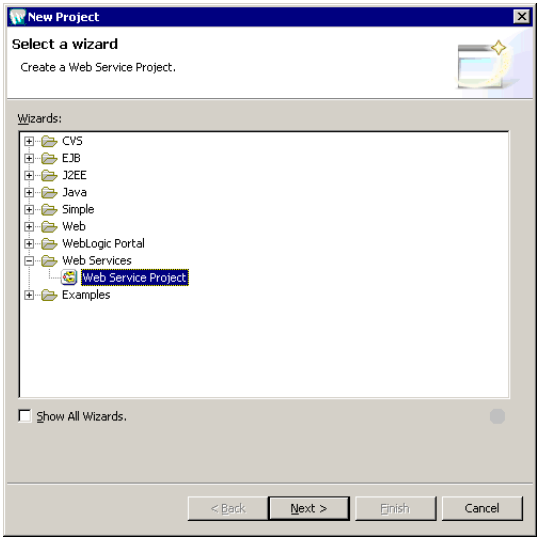
Figure 7-17 Error Message



To create the Web Service project, perform the following steps:

1. Click Start→All Programs→BEA Products (BEA HOME)→Workshop for WebLogic Platform.
2. In Workshop for WebLogic Platform, click File → New→ Project and select Web Service Project as shown in Figure 7-18.

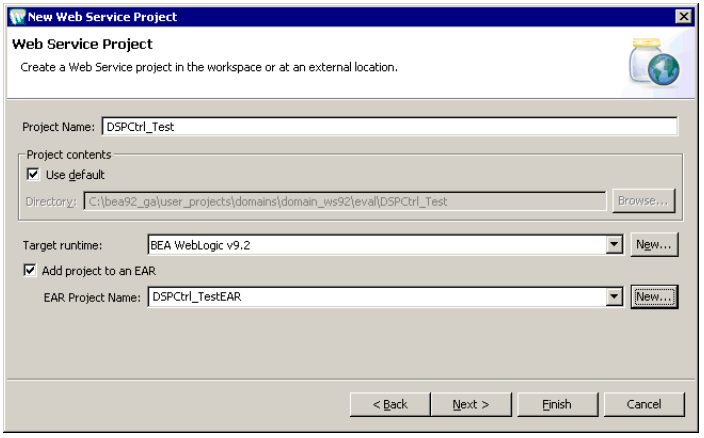
Figure 7-18 New Project: Select a Wizard Dialog Box



3. Click Next and specify the project name in the Project Name box of the New Web Service Project dialog box as shown in Figure 7-19.

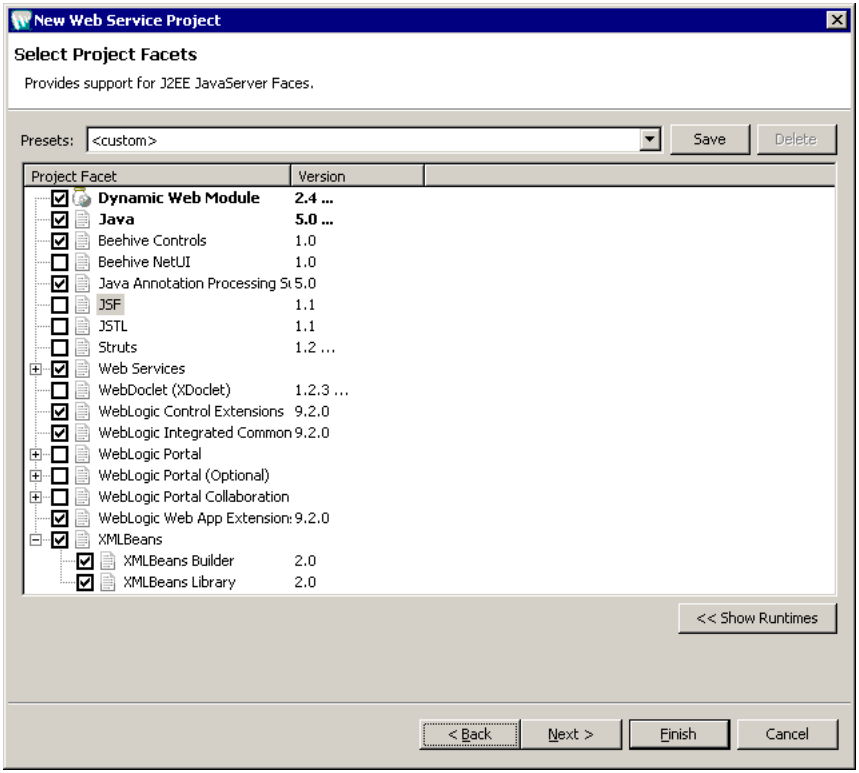
4. Select the Add Project to an EAR check box to create the EAR for the Web service project. The EAR represents the AquaLogic Data Services Platform application, which is deployed on the WebLogic Server 8.1.

Figure 7-19 New Web Service Project Dialog Box



5. Click Next and in the Select Project Facets dialog box, the Beehive Controls option is selected by default. You also need to select XMLBeans Builder as shown in Figure 7-20.

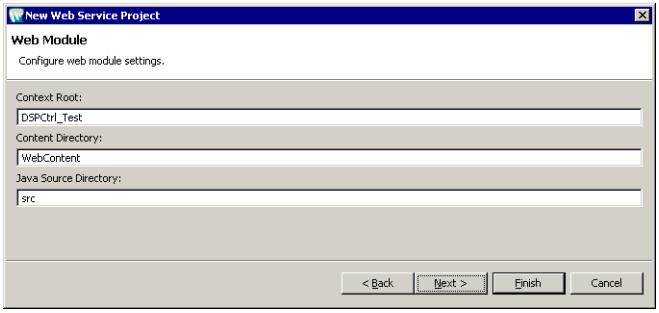
Figure 7-20 New Web Service Project: Select Project Facets



6. Click Next and specify the name of the Web module as shown in Figure 7-21. This module is created in the default WebContent folder that is created with the Web Service project.

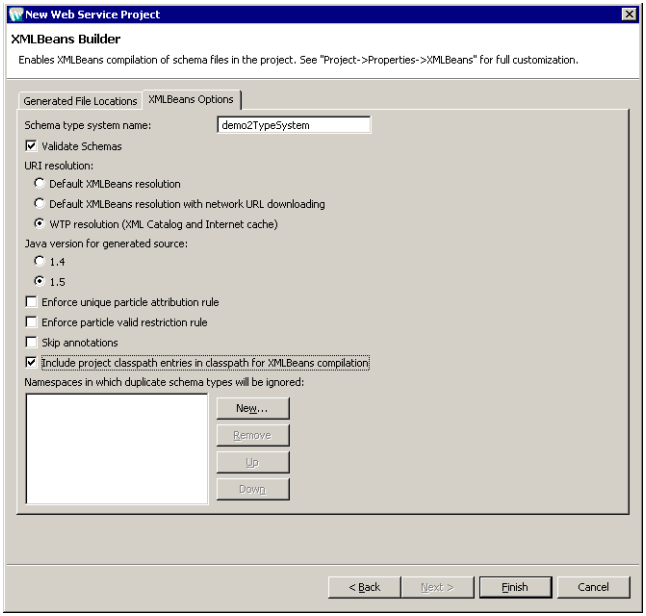
   Note: If a folder with the same name exists, you are prompted to provide a different name.

Figure 7-21 New Web Service Project: Web Module Dialog Box



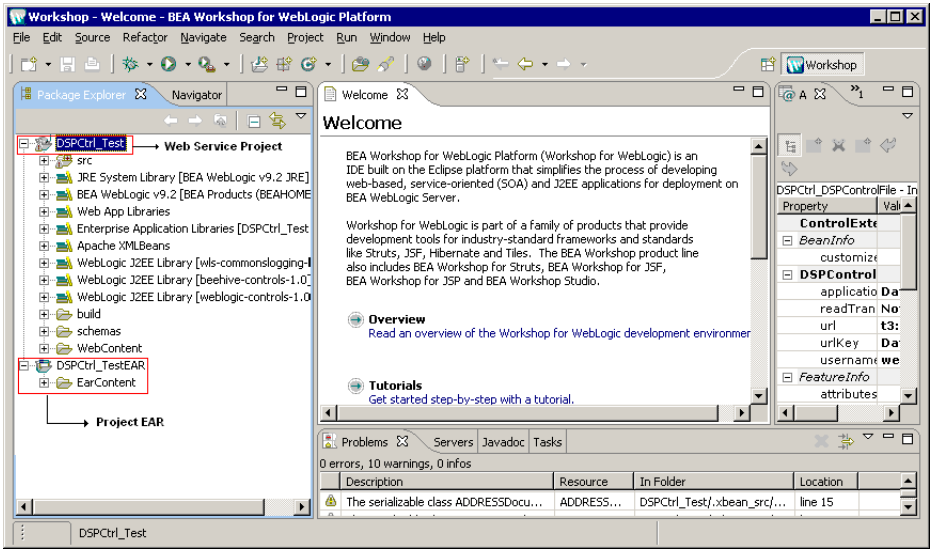
7. In the Web Module dialog box, retain the default selection in this dialog box, then click Next.
8. In the XMLBeans Builder dialog box, click the XMLBeans Option tab and select the **Include project classpath entries in classpath for XMLBeans compilation** check box, as shown in Figure 7-22. Click Finish.

Figure 7-22 New Web Service Project: XMLBeans Builder Dialog Box



The new Web service project (`DSPCtrl_Test`) and EAR (`DSPCtrl_EAR`) are created in Workshop for WebLogic Platform 9.2, as displayed in Figure 7-23. The EAR is the same as the application in AquaLogic Data Services Platform 8.1 environment.

Figure 7-23 New Web Service Project and EAR



9. After creating the Web service project, create the package in which the Data Service control will be wrapped, as follows:

   a. Right-click the `src` folder in the Web service project and select New→Package to display the New Java Package dialog box.

   b. Enter the name of the Java package, such as `com.bea.dspctrltest`, in the Name field and click Finish.

      The package is created under the `<WebServiceProject>/src/` directory in Workshop for WebLogic Platform.

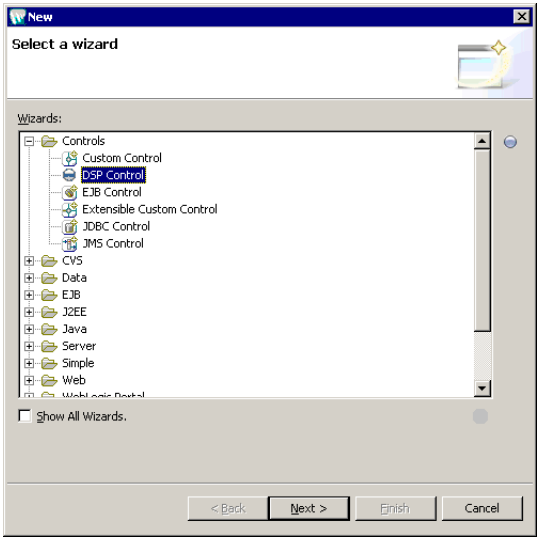
Creating Data Service Control

To create the Data Service control in Workshop for WebLogic Platform:

1. Right-click the package `com.bea.dspctrltest` you created and select New→ Others→ Controls.

2. Select DSP Controls from the Select a wizard dialog box and click Next, as shown in Figure 7-24.

Figure 7-24 Select a Wizard Dialog Box

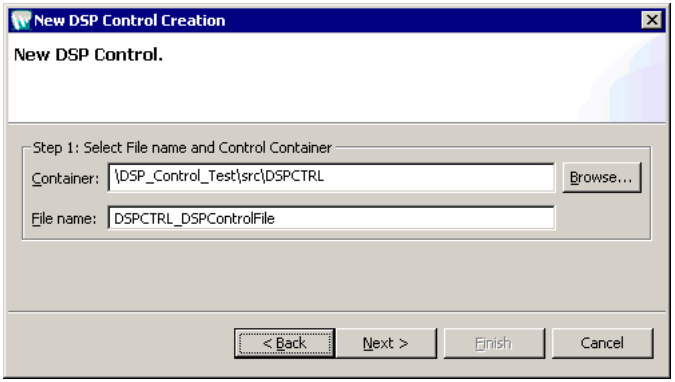


3. In the New DSP Control Creation dialog box displayed in Figure 7-25

   a. Enter the location where you want to create the control in the Container field.

   b. Enter the name of the control in the File name field.

Figure 7-25 New DSP Control Creation Dialog Box



Note: A Data Service control can be created only in a Web service project.

4. In the Select Control Attributes dialog box (Figure 7-26), specify the URL of the WebLogic Server where the AquaLogic Data Services Platform application is deployed, along with the user name and password for connecting to the server. The URL should be in the following format:

   ```
   t3://<ALDSPServerName>:<PortNum>
   ```

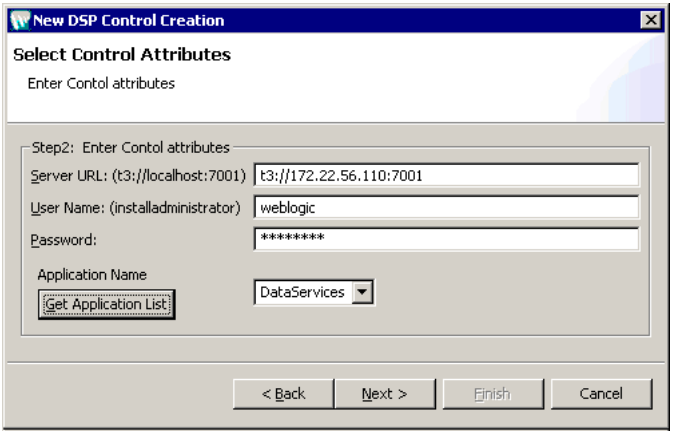
   where,

   `<ALDSPServerName>` is the WebLogic Server 8.1 where the AquaLogic Data Services Platform application is deployed.

   `<PortNum>` is the port number of the WebLogic Server 8.1.

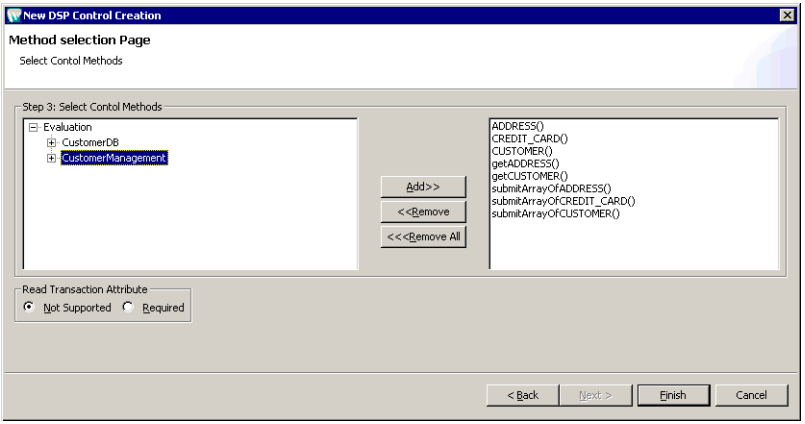
5. Click Get Application List for a list of data services deployed on the server and click Next.

Figure 7-26 Select Control Attribute



6. In the Method Selection Page dialog box (Figure 7-27), select the data services, which you want to access and click Add. These data services will now be accessible through the WebLogic 9.2 environment.

Figure 7-27 Method Selection Page Dialog Box

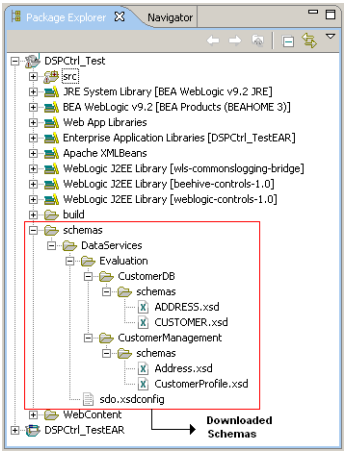


7. Now, click Finish. This completes the task of creating the Data Service control in Workshop for WebLogic Platform.

After you create the Data Service control 9.2, several operations automatically occur:

- The `DSP_Control.jar` file is copied to the `Webcontent/WEB_INF/lib` directory
- The schemas are copied to the default directory, which is configured for use by XMLBeans Builder (Figure 7-28). The hierarchy of the schemas is the same as the hierarchy in the 8.*x* environment.

Figure 7-28 Downloaded Schemas



- The `sdo.xsdconfig` file is copied to the following location:

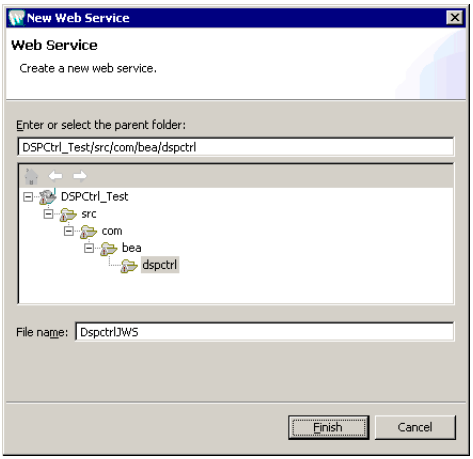
  `<XML_BEAN_BUILDER_SRC_DIR>/<ApplicationName>`

Creating the JWS and Using the Data Service Control

You can now use Data Service control 9.2 by creating a JWS. To create the JWS:

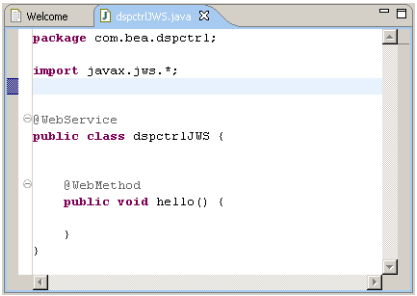
1. Right-click the `com.bea.dspctrltest` Java package that you created earlier and select New→WebLogic Web Service. Specify the name of the JWS project in the File Name field as displayed in Figure 7-29.

Figure 7-29 New Java Web Service



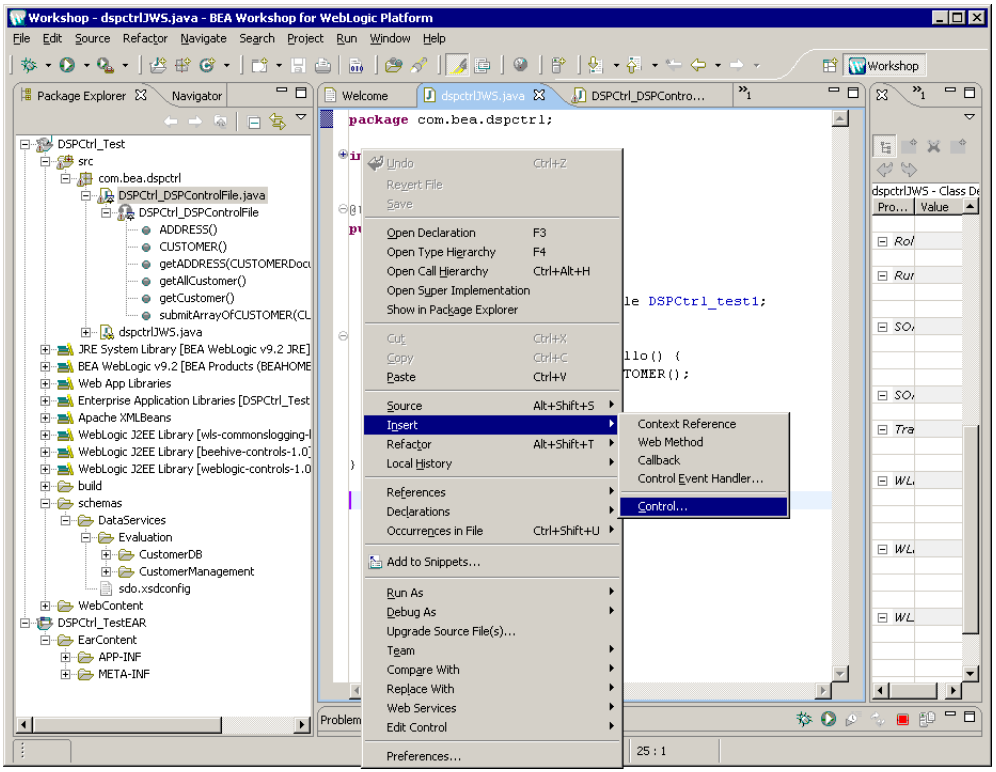
2. Click Finish. This creates the Java Web service as shown in Figure 7-30, which will use Data Service control to access data services.

Figure 7-30 Java Web Service



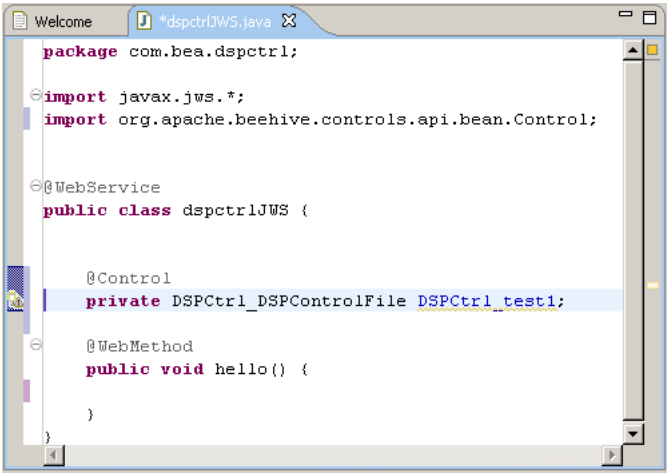
3. Right-click in the JWS file tab (`dspctrlJWS.java`) and select Insert→Control (Figure 7-31) to display the Select Control dialog box.

Figure 7-31 Adding Data Service Control to JWS



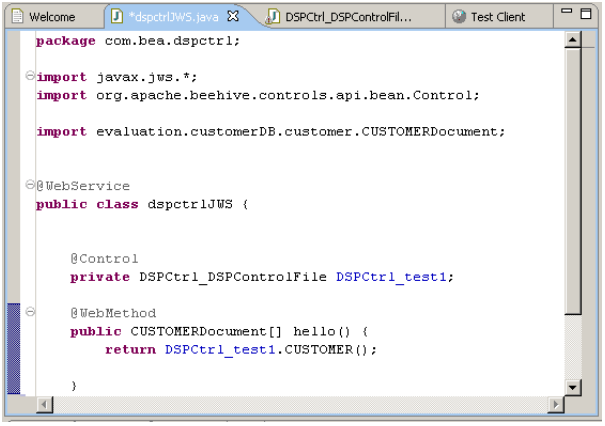
4. The `DSPCtrl_Test` control that you created earlier is displayed in the Existing Project Controls list. Select this control and click OK. This creates the Control annotation (`@Control`) in the WebServices annotation and declares the control variable, as shown in Figure 7-32.

Figure 7-32 Control Annotation



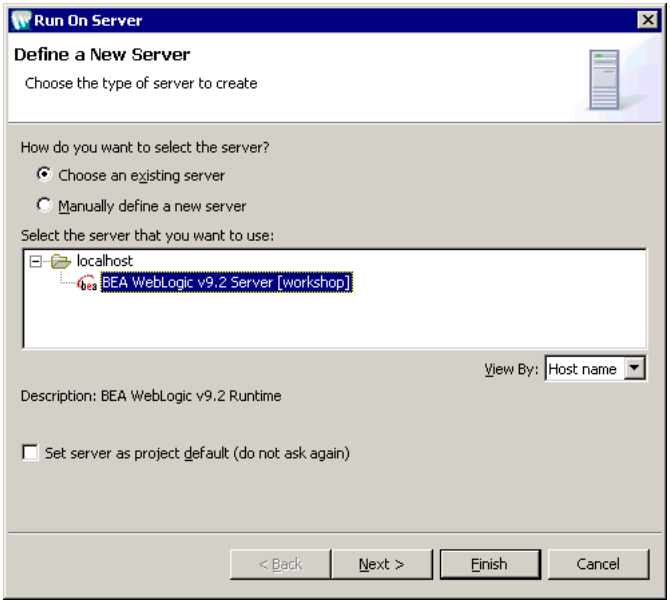
5. You can now invoke any method for the data services that you selected for the control. In this case, the `CUSTOMER()` has been invoked, as shown in Figure 7-33.

Figure 7-33 Invoking a Method through Data Service Control



6. To run the method on the server, right-click in the window and select Run As→Run on Server to display the Run on Server dialog box (Figure 7-34).

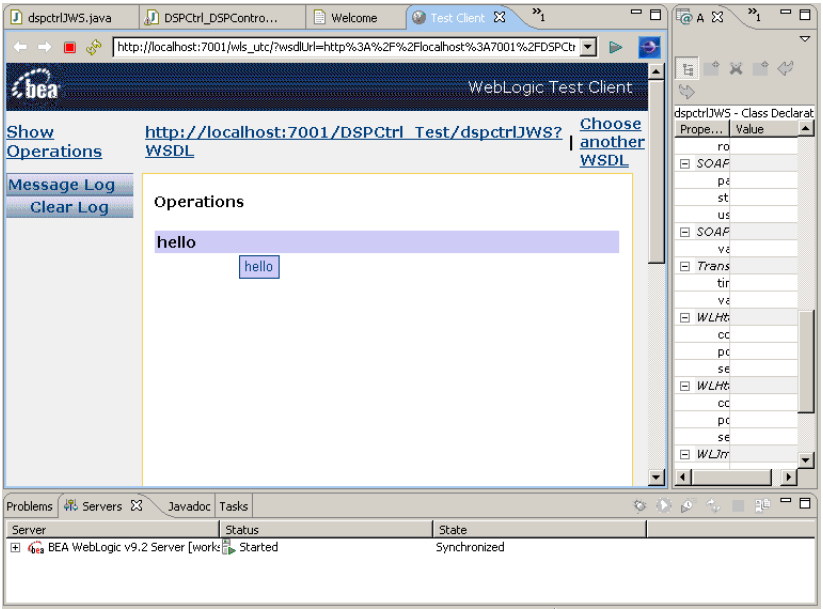
Figure 7-34 Run on Server Dialog Box



7. Select the 9.2 WebLogic Server and click Next. In the Add and Remove Projects dialog box, the project EAR contains the control. If there are any other EARs, which you do not want to run on the server, you should remove them from the list, and then click Finish.

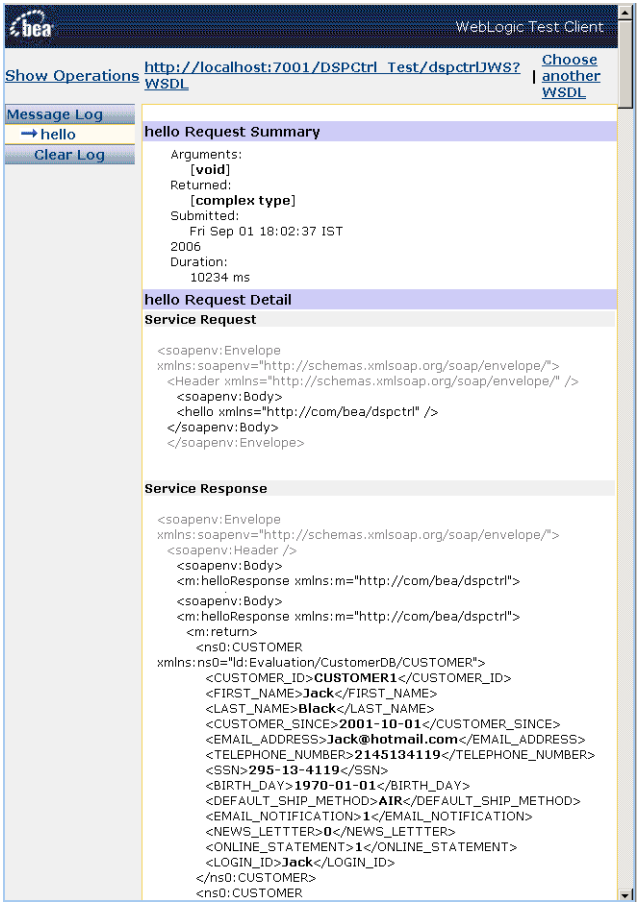
   The WebLogic Server 9.2 is started and the application opens up in the Workshop Test Browser, as shown in Figure 7-35.

Figure 7-35 Workshop Test Browser



8. Now, you can execute the `hello()` method, by clicking **hello** in the test browser. The server response is displayed, as shown in Figure 7-36.

Figure 7-36 Response for the Hello Method



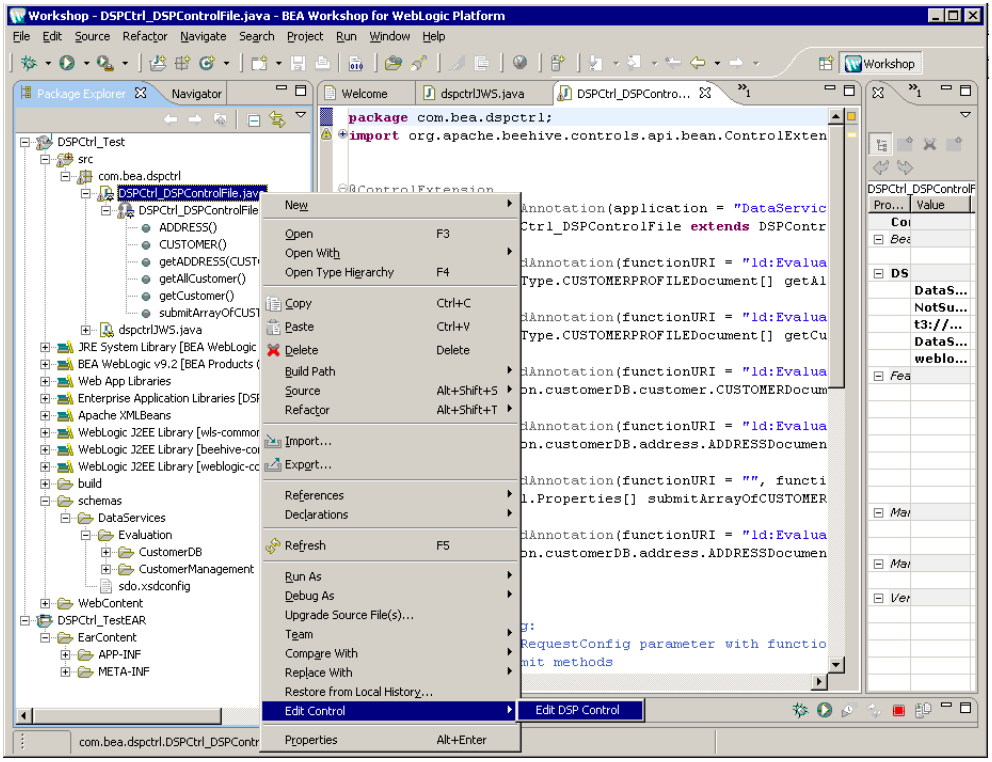
Modifying and Uninstalling the Control

This section provides the steps to edit, delete, and uninstall the Data Service control.

Editing and Deleting the Data Service Control

To edit the Data Service control, right-click the data service control file (`DSPCtrl_DSPControlFile.java`), and select Edit Control→Edit DSP Control, as shown in Figure 7-37.

Figure 7-37 Editing the Data Service Control



To delete the Data Service control 9.2, right-click the control file and select Delete.

Uninstalling the Control

To uninstall the control:

1. Go to the directory where you installed the control, which is set to:

   `<BEA_HOME>/workshop92/workshop4WP/eclipse/plugins/`

2. Delete the control plug-in directory, `com.bea.dsp.control.wizard_1.0.0`, from this location.

CHAPTER 8

Accessing Data Services Through AquaLogic Service Bus

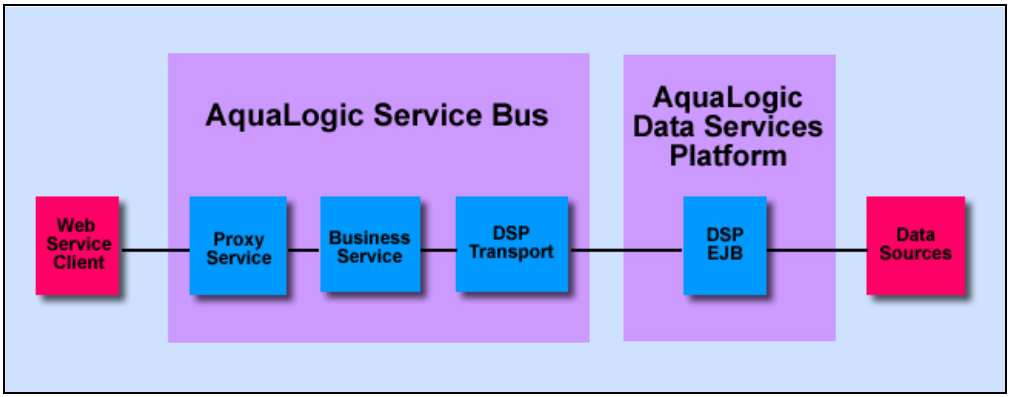
AquaLogic Data Services Platform can be accessed from AquaLogic Service Bus. Thus an AquaLogic Service Bus client make full use of data services (Figure 8-1). This allows a more efficient and flexible approach to accessing data services than exposing them as web services via WebLogic Workshop and Java Web Services (JWS).

To make an AquaLogic Data Service Platform data service available to an AquaLogic Service Bus client, you need to do the following:

- Deploy the AquaLogic Data Service Platform transport in the AquaLogic Service Bus as an application;
- Generate the WSDL file for the data service of interest and import it into the AquaLogic Service Bus;
- Generate a business service based on the WSDL and generate a proxy service based on the business service.

Your client is then able to access the data service as an AquaLogic Service Bus client.

Figure 8-1 AquaLogic Data Service Platform and AquaLogic Service Bus Interoperability Architecture



The following section provides the details.

Accessing AquaLogic Data Services Platform from AquaLogic Service Bus

Note: The following assumes that you are running AquaLogic Service Bus 2.5 under WebLogic Server 9.2 and AquaLogic Data Service Platform under WebLogic Server 8.1.

Perform the following steps to access AquaLogic Data Services Platform from AquaLogic Service Bus:

Step 1: Start Your Servers

1. Start the WebLogic Server 9.2 for the AquaLogic Service Bus application needing access to your WebLogic Server 8.1 data service.
2. Start the WebLogic Server 8.1 for the data service.

For example, suppose that the MortgageBroker application sample in AquaLogic Service Bus needs access to a RTLApp data service that comes with AquaLogic Data Service Platform.

You then need to start the server for the AquaLogic Service Bus Mortgage Broker examples (on Windows you can do this by selecting:

Start → All Programs → BEA Products → Examples → AquaLogic Service Bus → Start Examples Server

You also need to start the server for the RTL demo (on Window you can do this by selecting:

Start → All Programs → BEA WebLogic Platform 8.1 → BEA AquaLogic Data ServicesPlatform2.5 → Examples → RTL Demo → Launch RTL Demo Server

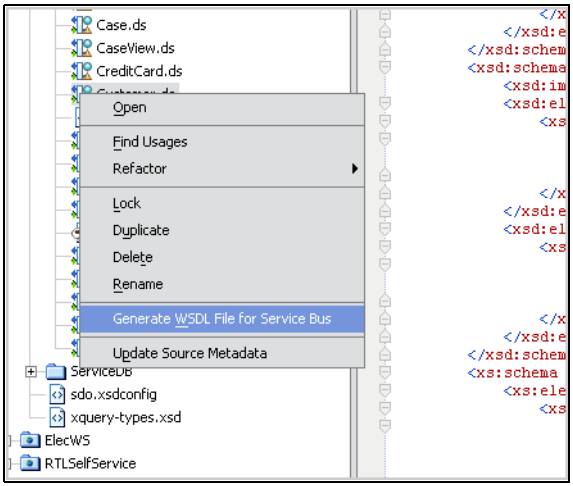
Step 2: Generate the WSDL for the Data Service

You can obtain the WSDL for the Data Service in two different ways. Each is described below.

Generate the WSDL Through WebLogic Workshop 8.1

1. Launch WebLogic Workshop 8.1.
2. Navigate in the Application panel (on the left) to your data service (`.ds` file) that you want to be available from AquaLogic Service Bus and select it.
3. Right-click to select Generate WSDL File for Service Bus (Figure 8-2). A WSDL file for the data service will be generated in the same directory where the data service is located.

Figure 8-2 Generate WSDL for Service Bus Dialog



For example, if you had navigated to the `Customer.ds` file in the RTLApp, `Customer.wsdl` will be generated.

Export the WSDL with the AquaLogic Data Service Console

1. Launch the AquaLogic Data Services Console.

   On Windows you can do this by selecting:

Start → All Programs → BEA WebLogic Platform 8.1 → BEA AquaLogic Data Services Platform 2.5 → Examples → RTL Demo → AquaLogic Data Services Console

You can also do this by typing `http://localhost:7001/ldconsole` in your web browser.

2. In the project navigator on the left, select `ldplatform`, then your application underneath it.
3. Next navigate to `Data Services` and to the data service for your application.

   For instance, if your application were the RTLApp example application, then you would navigate from `ldplatform` to `RTLApp` to `Data Services` to `RTLServices`.
4. Pick the particular service that you want to export.

   If you were exporting a service from the RTLApp example, you might pick `Customer`.
5. Click Export WSDL in the far right column of the data service you want to export. You will be given the opportunity to save it to any location.

Step 3: Deploy the Data Services Transport

1. Launch the AquaLogic Service Bus console, then select the WebLogic Server Console.

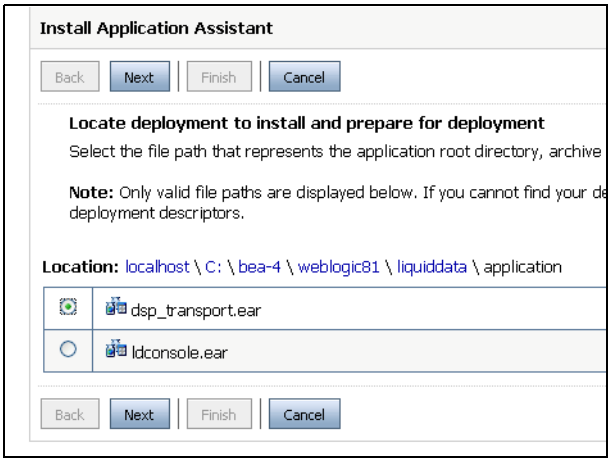
   The WebLogic Server Console will appear in a new window. Click on Deployments under Your Deployed Resources.
2. When the Summary of Deployments panel appears, click Lock & Edit under Change Center on the left.
3. The Install button then becomes active. Click it.
4. In the Install Assistant, navigate to the deployment EAR file, `dsp_transport.ear`, in WebLogic Server 8.1.

   In Windows, this is located at:

   `<bea_home>\weblogic81\liquiddata\application\`
5. Then select `dsp_transport.ear` and click Next.

   `dsp_transport` will appear under Deployments.

Figure 8-3 Install Application Assistant Dialog



6. Click Activate, then check the box in front of `dsp_transport.ear` and select Servicing All Requests from the Start drop-down menu. This will complete activation.

Step 4: Import the WSDL for the Data Service

1. Return to the service bus console and select Project Explorer.
2. Navigate to your project folder and click either Create or Edit.

   Note: There will be either a Create or an Edit button in the Change Center, depending on whether you are creating a new session or editing an existing one.

3. Select the project folder or an existing subfolder, or create a new one. For the latter, type a name in Enter New Folder Name under Folders and click Add. The new folder will be added to the tree structure.
4. Next import the WSDL file that you generated in Step 2. Do this by selecting WSDL under Interface in the Create Resource drop-down menu.
5. On the panel that appears next, give the resource a name and a description (optional), then click Browse to locate the WSDL file that you generated in Step 2. Select it and click Save. It will then appear as a resource in the Resources table.

For more information, see Adding a WSDL in "Using the AquaLogic Service Bus Console."

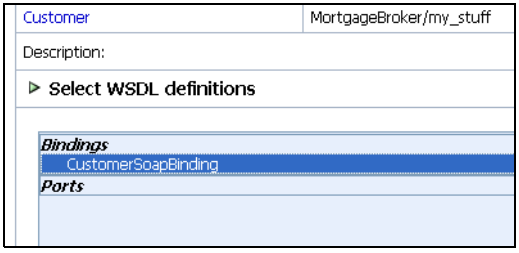
Step 5: Create the Business Service

1. Next, in Create Resources under Resources, select Business Service under Service. In the panel that appears, enter a name for the service in Service Name. For instance, if you had created `Customer.wsdl` for the sample RTLApp, then you might want to name your business service CustomerBS. Enter an optional description of the service.

2. Below in the same panel, select WSDL Web Service and click Browse to locate the WSDL file that you imported in Step 4. Click on it. A Select WSDL Definitions panel will appear. Select a binding under Bindings.

   If you were running the sample application and had imported `Customer.wsdl` as described above, you would select CustomerSoapBinding (Figure 8-4).

Figure 8-4 Setting SOAP Binding for WSDL



3. Click Submit. Then click Next in the main panel.

4. Under Protocol, select `dsp` from the drop-down menu. Then enter the Endpoint URI. For Endpoint URI, enter:

   `t3://<host:port>/<application_name>.`

   If you were running the RTLApp as described above in the default setup, you would enter:

   `t3://localhost:7001/RTLApp`

5. Click Add to add the Endpoint URI, then click Next.

6. Click Next again to accept the defaults. Then on the Create a Business Service - Summary screen that appears, click Save.

   The business service should then appear under the table of Resources. If you were creating `CustomerBS`, it will appear in the table.

For more information, see Adding a Business Service in "Using the AquaLogic Service Bus Console."

Step 6: Create the Proxy Service

The procedure for creating the proxy service is similar to that for creating the business service.

1. Under Create Resources, select Proxy Service under Service in the drop-down menu.
2. In the next panel that appears, give your proxy service a name. If you were working with the example, you might want to name it CustomerPS. Provide a description (optional) for the service in Description. Then select Business Service under Create from Existing Service and click Browse. In the screen that appears, select the name of the business service that you created in the previous step. If you had created `CustomerBS`, as in the example, select CustomerBS. Click Submit, then click Next.
3. In the next panel, select http in the drop-down menu for Protocol and type in a name for the Endpoint URI. For Endpoint URI you can give it any name you want. Click Next.
4. In the next two screens click Next to accept defaults, or fill in other values or select other choices.
5. You will then see a summary screen allowing you to edit your entries. When you are satisfied with your entries, click Save. The proxy service should then appear in the table of resources. If you were working with the example, you should see CustomerPS in the table.
6. Click Activate, then click Submit.

For more information, see Adding a Proxy Service in "Using the AquaLogic Service Bus Console."

Step 7: Test Your Setup

1. Reselect the folder where your resources are located, then, under Resources, locate the proxy service you created.
2. Select Launch Test Console under Actions. The test console should appear. Pick a method under Available Options in the drop-down menu and click Execute. For the RTLApp example, you might select the `getCustomer` method. In Response Document, you should see a list of customers.

For more information, see Test Console in "Using the AquaLogic Service Bus Console."

Additional Information

For more information, see the following:

- BEA WebLogic Workshop 8.1 documentation

  `http://edocs.bea.com/workshop/docs81/index.html`

- Using the AquaLogic Service Bus Console

  `http://edocs.bea.com/alsb/docs25/consolehelp/index.html`

- Using the AquaLogic Data Services Platform Console

  `http://edocs.bea.com/aldsp/docs25/admin/ldconsole.html`

CHAPTER 9

Supporting ADO.NET Clients

This chapter describes how to enable interoperability between BEA AquaLogic Data Services Platform 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 AquaLogic Data Services Platform 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 AquaLogic Data Services Platform data service—all the work is done on the server-side. ADO.NET-client-application developers need only incorporate the AquaLogic Data Services Platform-generated WSDL into their programming environments, as you would when creating any Web service client application.

General information about how AquaLogic Data Services Platform 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 AquaLogic Data Services Platform Support for ADO.NET Clients
- Adapting AquaLogic Data Services Platform 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 AquaLogic Data Services Platform client applications.

See "Role of the Mediator and SDOs" on page 1-16 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

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.

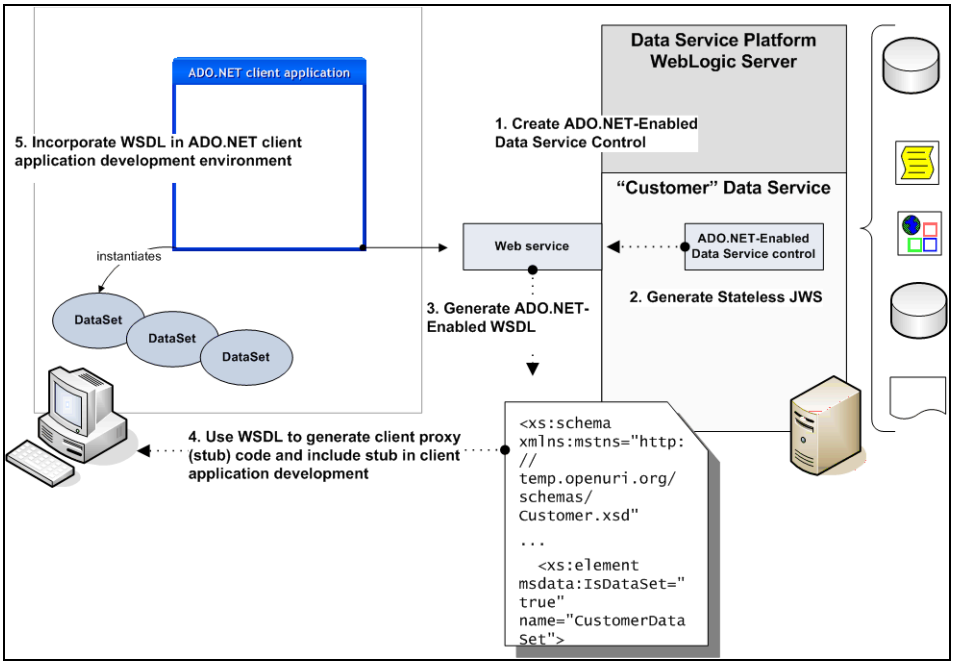
Note: See Microsoft's MSDN site (`http://msdn.microsoft.com/`) for more information about ADO.NET and client application development.

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 9-1 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 9-1):

- Obtain the WSDL for the AquaLogic Data Services Platform 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 9-10.

Understanding How AquaLogic Data Services Platform 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.

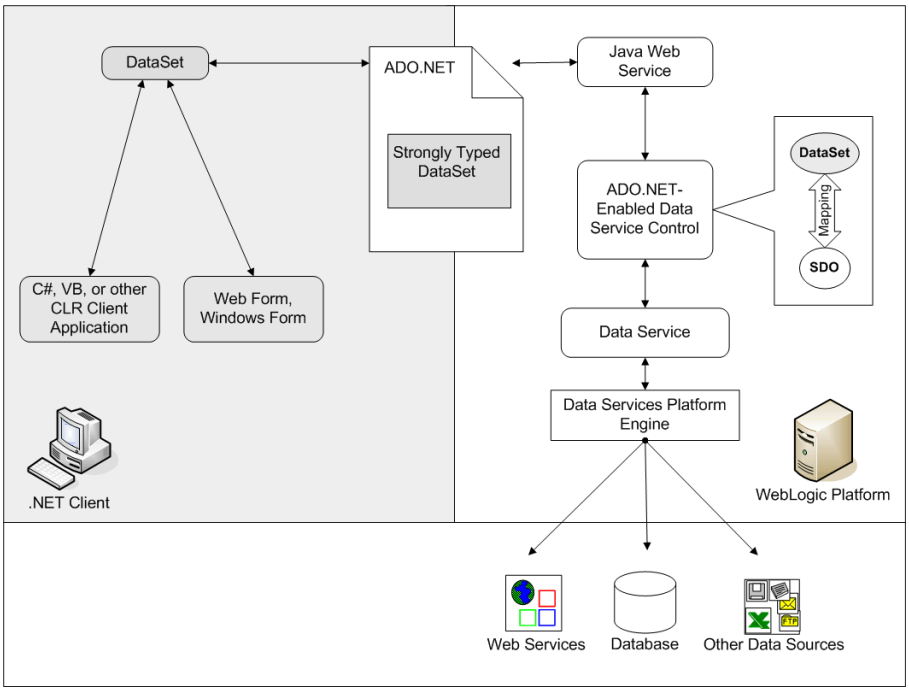
Table 9-2 ADO.NET and SDO Data Objects Compared

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 9-3, the ADO.NET typed DataSet is submitted to and returned by AquaLogic Data Services Platform. 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 9-3 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 AquaLogic Data Services Platform 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 9-4 are exposed to data service developers.

Table 9-4 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 9-5.

Table 9-5 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 7-7
Generate Web service file (JWS)	"Server-Side DSP-Enabled Web Service Development" on page 4-3
Generate WSDL	"Server-Side DSP-Enabled Web Service Development" on page 4-3

Enabling AquaLogic Data Services Platform 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 AquaLogic Data Services Platform 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 9-6).

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 9-6 Folder Structure for ADO.NET-Enabled Project Components

Creating an ADO.NET-Enabled Data Service Control

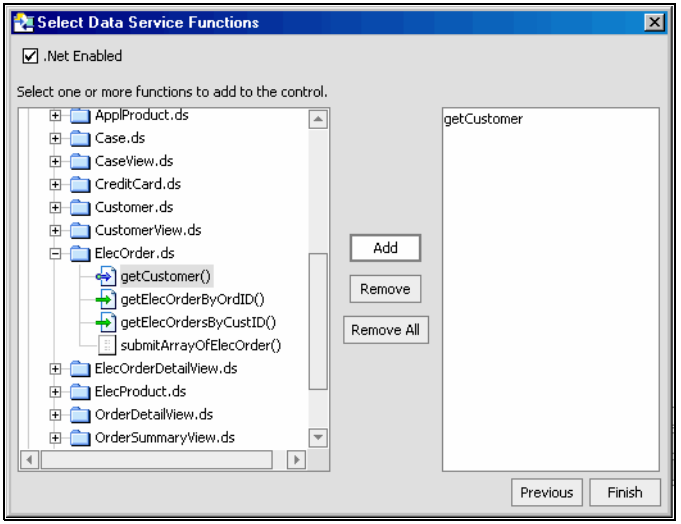
Data Service controls can be ADO.NET-enabled simply by selecting the appropriate checkbox during the creation process. The ADO.NET-Enabled Data Service controls created (as described in this section) are designed exclusively to support ADO.NET clients through a Web services interface: such controls cannot 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 9-7.

Figure 9-7 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 11-8.)

   – 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 9-4 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 7-7.

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).

Note: Deleting a JCX does not cause the deletion of any associated schema (XSD) files. Instead if you need to remove these files from your system, do so manually.

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`

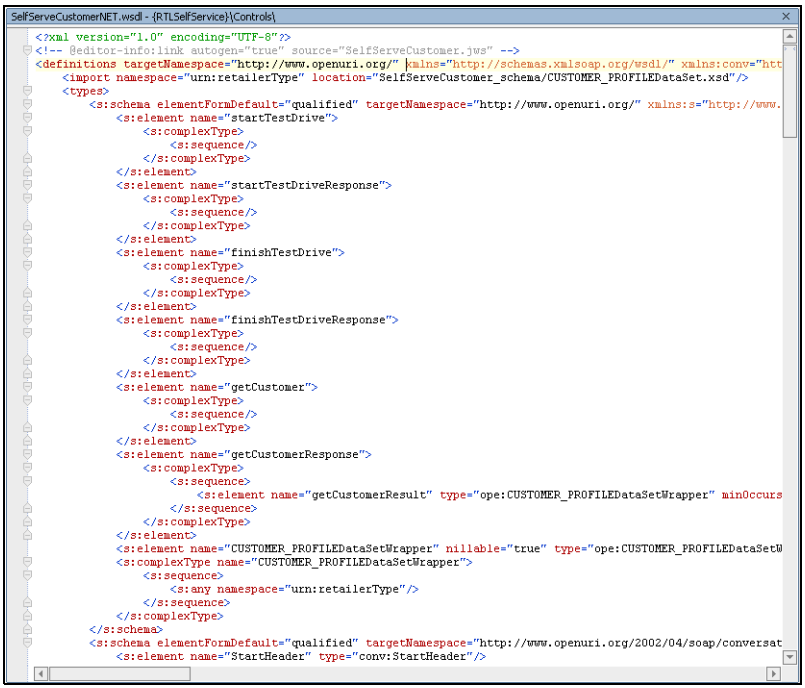
Generating an ADO.NET-Enabled WSDL

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 9-8 Generated WSDL in WebLogic Workshop



- See "Web Services Description Language (WSDL) File for Microsoft ADO.NET Clients" on page 9-17 for information about the format of the WSDL.

Note: The building of RPC-style Web services on top of AquaLogic Data Services Platform controls is not supported. For this reason RPC-style Web services built on cannot be created from ADO.NET clients utilizing AquaLogic Data Services Platform.

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 BEA WebLogic Server's home page.

Adapting AquaLogic Data Services Platform 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 AquaLogic Data Services Platform server, specifically explaining how to:

- Read a AquaLogic Data Services Platform query result as a ADO.NET DataSet via SDO (since query results are presented as SDO DataObjects within AquaLogic Data Services Platform).
- Update AquaLogic Data Services Platform 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 service**s 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" on page 9-13)

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

- **Develop ADO.NET-compatible** data service**s 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 the Best Practices and Advanced Topics chapter of the *Building Queries and Data Views*.)

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 AquaLogic Data Services Platform update override function to handle the update.)

  Note: The usage of inverse functions is described in the Best Practices and Advanced Topics chapter of the *Building Queries and Data Views*.

- 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 AquaLogic Data Services Platform data service)

Listing 9-1 shows an excerpt of a schema—`CustomerDS.xsd`—for a typed DataSet generated from a AquaLogic Data Services Platform Customer schema.

Listing 9-1 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 10

Using Workflow with AquaLogic Data Services Platform-Based 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.

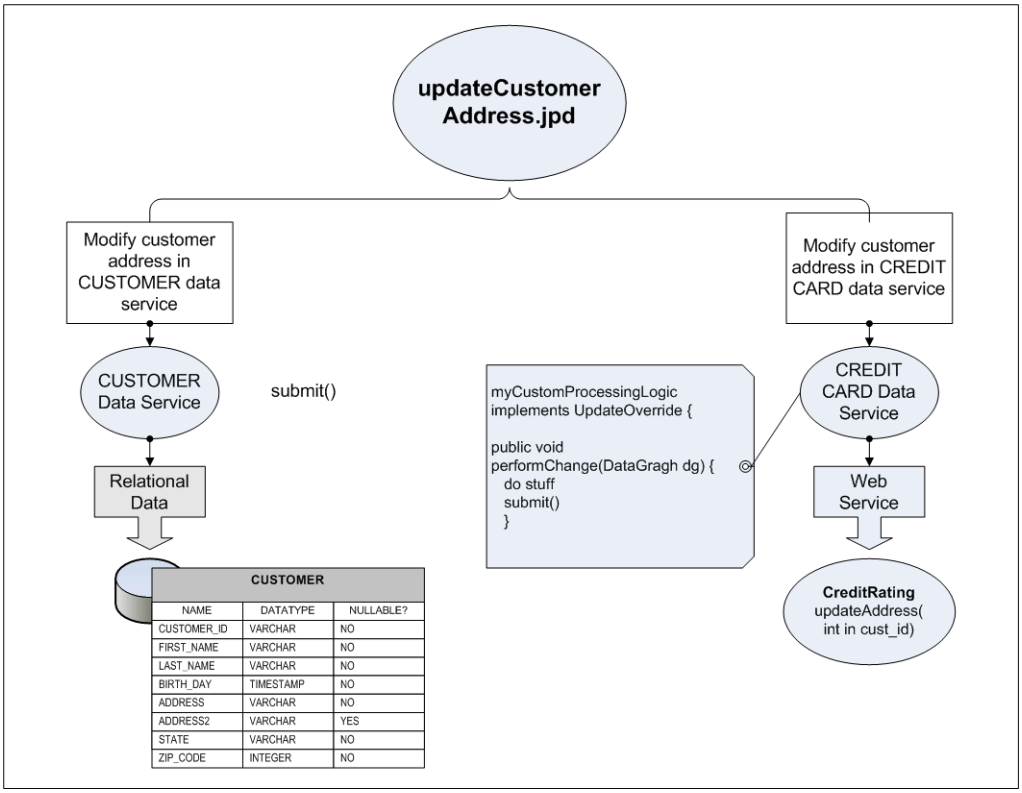
For example, a WebLogic Integration process (JPD) can be defined that encompasses multiple AquaLogic Data Services Platform data services, and that uses the JPD to enforce distributed transactional semantics without using XA and to reduce the number of locks held on disparate data sources (such as Web 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 10-1).

Note: The invocation of JPDs from Data Services Platform is described in the Handling Updates Through Data Services chapter of the *Building Queries and Data Views*.

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 10-1 Using WLI JPD with AquaLogic Data Services Platform 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 AquaLogic Data Services Platform 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 7-1 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
```

AquaLogic Data Services Platform and JPD can be integrated in two different ways:

- By adding Data Service controls to JPD projects you can leverage AquaLogic Data Services Platform-enabled application information as part of a workflow.
- By invoking JPDs from AquaLogic Data Services Platform-enabled applications. (See the Handling Updates Through Data Services chapter of the *Building Queries and Data Views* for details.)

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.

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 AquaLogic Data Services Platform-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 7-1 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.

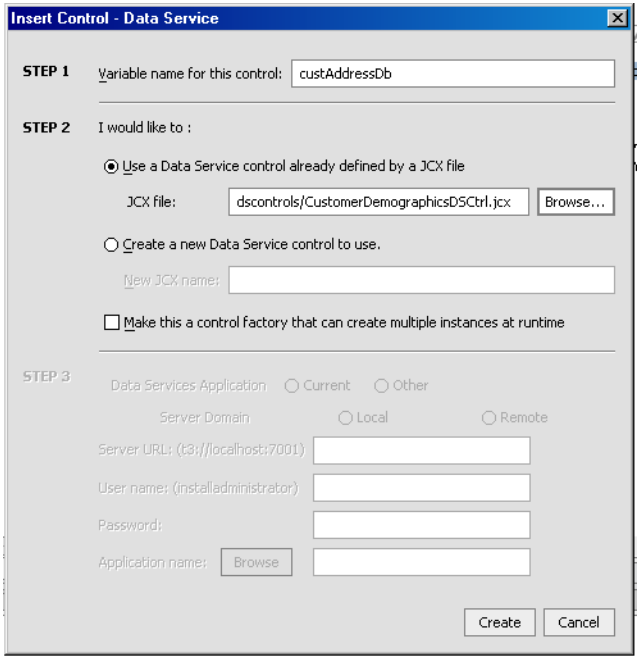
For complete details, see "Data Service Controls Defined" on page 7-2.

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 10-2 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 10-3, 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 10-3 Specifying in the Business Process Input and Output Parameters for a Data Service Control

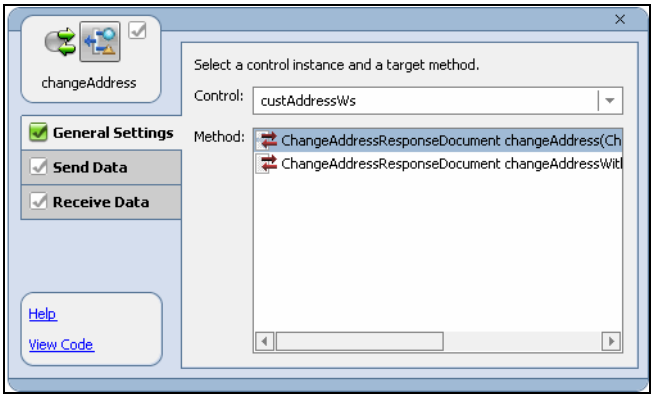
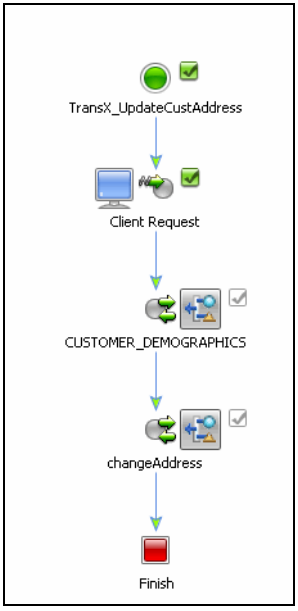


Figure 10-4 shows the WebLogic Workshop rendering of a business process accessing a Data Service Control.

Figure 10-4 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.

Updating Multiple Data Services Using Workflows

Once created, custom update classes can be used to create workflows that manage updates to multiple data services. For information on invoking a JPD from an update override class see "Invoking JPDs from Data Services Platform" in the Handling Updates Through Data Services chapter of *Data Services Developer's Guide*.

CHAPTER 11

Advanced Topics

This chapter provides information on miscellaneous topics related to client programming with BEA AquaLogic Data Services Platform. It covers the following topics:

- Using Catalog Services to Obtain Metadata on Data Services
- Filtering, Sorting, and Fine-tuning Query Results
- Handling Large Result Sets with Streaming APIs

Using Catalog Services to Obtain Metadata on Data Services

BEA AquaLogic Data Services Platform 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 AquaLogic Data Services Platform.

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 AquaLogic Data Services Platform'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 11-3) 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:

- AquaLogic Data Services Console
- AquaLogic Data Services Platform 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. A sample is shown in Listing 11-1.

Listing 11-1 Example of Client Code Utilizing Catalog Services

```
import com.bea.dsp.RequestConfig;
import com.bea.ld.DSPAuditRecord;
import com.bea.ld.DataServiceAudit;
import com.bea.ld.metadata.DataServiceDocument;
import com.bea.ld.metadata.DataServiceDocument.DataService;
import com.bea.ld.metadata.DataServiceRefDocument;
```

```
import com.bea.ld.metadata.DataServiceRefDocument.DataServiceRef;
import com.bea.ld.metadata.FunctionDocument;
import com.bea.ld.metadata.FunctionDocument.Function;
import com.bea.ld.metadata.RelationshipDocument;
import com.bea.ld.metadata.RelationshipDocument.Relationship;
import com.bea.ld.server.audit.DataServiceAuditImpl;
import com.bea.dsp.dsmediator.client.exception.SDOMediatorException;
import com.bea.ld.filter.FilterXQuery;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import weblogic.jndi.Environment;

public class Client
{
    public static void main(String[] args){


        try{
            DataServiceAudit dsAudit=null;
            RequestConfig reqConfig = new  RequestConfig();
         reqConfig.enableFeature(RequestConfig.RETURN_DATA_SERVICE_AUDIT);

reqConfig.setStringArrayAttribute(RequestConfig.RETURN_AUDIT_PROPERTIES,at
tributes);

            String appName="CatalogServicesClient";
            Context ctx=getInitialContext();
            _catalogservices.DataServiceRef
csDSR=_catalogservices.DataServiceRef.getInstance(ctx, appName);

            DataServiceRefDocument[]
dataServiceRefDocs=csDSR.getDataServiceRefs(reqConfig);
            dsAudit=reqConfig.retrieveDataServiceAudit();
```

```
            List l=dsAudit.getAllRecords();
            for(Iterator it=l.iterator(); it.hasNext();  ){
                DSPAuditRecord auditRec = (DSPAuditRecord)it.next();
                Map propMap = auditRec.getAuditProperties();
              for(Iterator pit=propMap.keySet().iterator(); pit.hasNext();
){
                    String key=(String)pit.next();
                    System.out.println("Audit Information: "+key+" =
"+propMap.get(key));
                }
            }
            for(int i=0; i< dataServiceRefDocs.length; i++){
                DataServiceRefDocument
dataServiceRefDoc=dataServiceRefDocs[i];
                DataServiceRef
dataServiceRef=dataServiceRefDoc.getDataServiceRef();
            System.out.println("DataServiceRef: "+dataServiceRef.getId());
                /* this will work */ //DataServiceDocument
dataServiceDoc=csDSR.getDataService(dataServiceRefDoc);
                _catalogservices.DataService
csDS=_catalogservices.DataService.getInstance(ctx,appName);
                DataServiceDocument
dataServiceDoc=csDS.getDataService(dataServiceRefDoc);
                //DataServiceDocument
dataServiceDoc=csDSR.getDataService(dataServiceRefDoc);
                DataService dataService=dataServiceDoc.getDataService();
                System.out.println("DataService: "+dataService);
                System.out.println("Functions: ");
                FunctionDocument[]
functionDocs=csDSR.getFunctions(dataServiceRefDoc);
              for(int j=0; functionDocs != null && j<functionDocs.length;
j++){
                    FunctionDocument functionDoc=functionDocs[j];
                    Function function=functionDoc.getFunction();
                    System.out.println("Function: "+function);
                }
                System.out.println("Relationships: ");
                    RelationshipDocument[]
```

```
relationshipDocs=csDSR.getRelationships(dataServiceRefDoc);
                    for(int j=0;  relationshipDocs != null &&
j<relationshipDocs.length; j++){
                        RelationshipDocument
relationshipDoc=relationshipDocs[j];
                        Relationship
relationship=relationshipDoc.getRelationship();
                        System.out.println("Relationship: "+relationship);
                    }


            }
        } catch(Exception e){
            e.printStackTrace();
        }
    }

        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());
    }

       static protected String[] attributes = new String[]{
                                        "common",
                                   "query"
                                        };

}
```

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

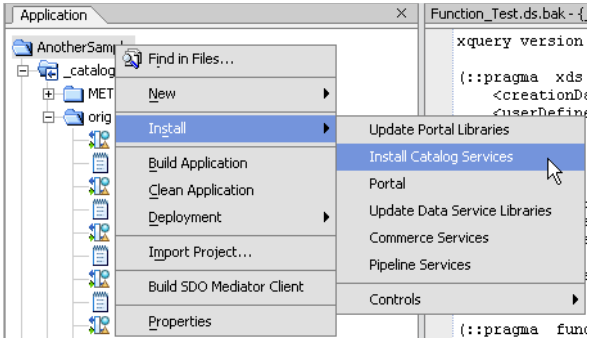
6) Provide a form to enter the arguments. The more complex the arguments, the more complex the code you must write.

You can obtain a schema for parameters using the Catalog. You can get the schema for parameters from Catalog services.

Installing Catalog Services

The ability to build Catalog Services within any AquaLogic Data Services Platform-enabled application is available by default when you install the product. AquaLogic Data Services Platform Catalog Services are installed easily on a per-application basis, by selecting Install Catalog Services from the WebLogic Workshop main menu.

Figure 11-1 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 11-2).

After installing Catalog Services, you will have access to all application metadata. For any AquaLogic Data Services Platform-enabled application that you want to leverage in this way, simply install the Catalog Services into the application.

Table 11-2 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)
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)

Table 11-2 Catalog Services Accessor Methods

Data Service Name	Return Type (Java Class)	Accessor
SchemaRef	SchemaRef	getSchemaDependencyRefsBySchemaRef (SchemaRef)
SchemaRef	Schema	getSchemaByRef(SchemaRef)

In addition to the methods shown in Table 11-2, you will also see several extraneous methods that define relationships among data services. However, the methods shown 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 AquaLogic Data Services Platform'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 11-3.

Table 11-3 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 AquaLogic Data Services Platform APIs, is described at "AquaLogic Data Services Platform Mediator API Javadoc" on page 2-13.

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 11-3, “Filter Operators,” on page 11-9 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);
```

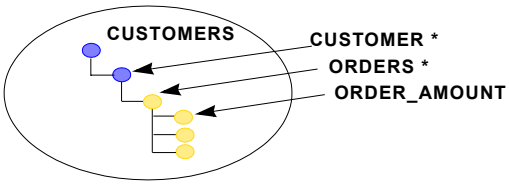
4. Invoke the data service function. (For more information on invoking data service functions, see Chapter 3, "Accessing Data Services from Java Clients.")

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 11-1. 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 11-4 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,

```
FilterXQuery myFilter = new FilterXQuery();
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:

```
FilterXQuery myFilter = new FilterXQuery();
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:

```
FilterXQuery myFilter = new FilterXQuery();
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:

```
FilterXQuery myFilter = new FilterXQuery();
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 11-2 uses the AND operator to apply a combination of filters to a result set, given a data service instance customerDS.

Listing 11-2 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 11-3 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 11-3 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 11-4 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 11-4 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 11-5 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 11-5 Invoking Data Service Functions using an Ad Hoc Query

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

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();
```

AquaLogic Data Services Platform 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 11-5 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 AquaLogic Data Services Platform" on page 3-8.)

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 11-5), for binding values of various data types.

Table 11-5 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)`

Table 11-5 PreparedExpression Methods for Bind Variables

To bind data type of...	Use bind method...
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)

Table 11-5 PreparedExpression Methods for Bind Variables

To bind data type of...	Use bind method...
`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 11-6 shows an example of using a bindInt() method in the context of an ad hoc query.

Listing 11-6 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 11-7 shows a complete ad hoc query example, using the PreparedExpression interface and QNames to pass values in bind methods.

Listing 11-7 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 AquaLogic Data Services Platform. For such cases, AquaLogic Data Services Platform can serve information as an output stream.

AquaLogic Data Services Platform 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: The streaming API is intended to be used when large result sets are needed and cannot be easily materialized in memory. It may be advisable to run your data service queries on different servers in order to simultaneously support both real-time queries and large, batch-oriented queries.

However, if two servers are not possible, then consider running the streaming API queries during off-peak times.

In AquaLogic Data Services Platform only server-side steaming is supported. Thus a JSP, which is hosted on the server, can leverage the streaming API. However, an external Java application could not.

You can get AquaLogic Data Services Platform information as a stream by using either an ad hoc or an untyped data service interface.

Note: Streaming is not supported through static interfaces. 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.

The streaming interface can be found in the following 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 = DataServiceFactory.newStreamingDataService(
                              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 AquaLogic Data Services Platform APIs is described at: "AquaLogic Data Services Platform Mediator API Javadoc" on page 2-13.

Listing 11-8 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 11-8 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 11-19).

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.