Contents

1. Introducing Data Services for Client Applications
   Introduction ........................................................................................................... 1-2
   What Is a Data Service? ......................................................................................... 1-2
   What is an ALDSP Client Application? ............................................................... 1-3
   Choosing a Client Programming Model ............................................................. 1-5
   Introducing Service Data Objects (SDO) ............................................................ 1-6
   Introducing The Data Service Mediator API ...................................................... 1-7
   Typical Client Application Development Process ............................................. 1-8
   Security Considerations in Client Applications ................................................. 1-8
   Performance Considerations ............................................................................... 1-9
   Client Classpath Settings .................................................................................... 1-9
      Java Mediator API Clients ................................................................................ 1-10
      Web Services Clients ....................................................................................... 1-10
      JMX Mbean Management API Client Classpath .............................................. 1-11
      ALDSP JDBC API Client Classpath ................................................................ 1-11
   Backward Compatibility ...................................................................................... 1-12

2. Data Programming Model and Update Framework
   Introduction ........................................................................................................... 2-1
   ALDSP and SDO ................................................................................................. 2-2
3. Invoking Data Services from Java Clients

Introducing the Mediator API .................................................. 3-2
What is SDO? ................................................................. 3-2
What is the Mediator API? ............................................. 3-3
Dynamic and Static Mediator APIs ........................................ 3-3
API Overview ................................................................. 3-4
Summary ................................................................. 3-5
Getting Started ............................................................... 3-5
Basic Steps ................................................................. 3-6
Setting the CLASSPATH .................................................. 3-7
  Static Java Mediator API Client CLASSPATH ................ 3-7
  Dynamic Java Mediator API Client CLASSPATH .............. 3-7
Specifying the Class Loader Directly .................................. 3-8
Running the Sample Applications ..................................... 3-9
Sample Static Mediator Application ................................ 3-9
Setting Up the Sample Data Service ................................ 3-10
Generating the Mediator Client JAR File ......................... 3-15
Setting Up the Java Project ............................................. 3-16
Running and Testing the Code ........................................... 3-19
Examining the Sample Code ............................................. 3-20
  Importing Packages .................................................. 3-20
  Obtaining a Data Access Service Handle .................... 3-21
Retrieving Data from the Service .................................................. 3-21
Obtaining a DataObject from the Result ........................................ 3-22
Disposing the Result Object ......................................................... 3-22
Modifying the DataObject ............................................................. 3-22
Returning Changes to the Server ................................................... 3-23
Sample Dynamic Mediator Application ......................................... 3-23
Setting Up and Running the Sample Code ..................................... 3-23
Sample Java Client Code (Dynamic Mediator API) ............................ 3-24
Examining the Sample Code ............................................................ 3-25
Importing Classes ......................................................................... 3-26
Obtaining a DataAccessService Handle ........................................... 3-26
Retrieving Data from the Service ................................................... 3-27
Obtaining a DataObject from the Result ........................................ 3-27
Disposing the Result Object ........................................................... 3-27
Modifying the DataObject .............................................................. 3-27
Returning Changes to the Server ................................................... 3-27
Creating New DataObjects ............................................................ 3-28
Creating a New DataObject with the Static API ............................... 3-28
Setting Up and Running the Sample .............................................. 3-28
Importing Packages ........................................................................ 3-31
Obtaining a Data Access Service Handle ........................................ 3-31
Creating a DataFactory ................................................................. 3-32
Create and Name the DataObject ................................................... 3-32
Modifying the DataObject .............................................................. 3-32
Returning New DataObject to the Server ....................................... 3-33
Returning the New DataObject Key ............................................... 3-33
Creating a New DataObject with the Dynamic API ......................... 3-33
Running the Sample ........................................................................ 3-33
4. Invoking Data Services Through Web Services

Overview ................................................................. 4-1
Before You Begin .................................................... 4-3
Getting Started ....................................................... 4-3
Basic Steps ............................................................... 4-4
Setting the CLASSPATH ................................................ 4-5
Running the Sample Applications ................................. 4-5
Sample Static Mediator Application ............................... 4-6
Setting Up the Sample Data Service ............................... 4-6
5. Using SQL to Access Data Services

Introducing SQL Access to Data Services
Features of the Data Services Platform JDBC Driver ........................................... 5-5
Exploring Data Services Platform and JDBC Artifacts ........................................ 5-5
JDBC and SQL Support in Data Services Platform ............................................. 5-6
JDBC Support ..................................................................................................... 5-6
SQL Support ....................................................................................................... 5-13
  Supported SQL Statements ........................................................................... 5-13
  Supported SQL Functions ............................................................................. 5-13
  Table Parameter Support ............................................................................. 5-18
Additional Details and Limitations ................................................................. 5-20
Preparing to Use SQL to Access Data Services .............................................. 5-21
Publishing Data Service Operations ............................................................... 5-21
Configuring the Data Services Platform JDBC Driver ..................................... 5-21
Accessing Data Services Using SQL From a Java Application ....................... 5-23
  Obtaining a Connection ............................................................................... 5-24
  Using the PreparedStatement Interface ...................................................... 5-25
  Using the CallableStatement Interface ...................................................... 5-26
Advanced Features ............................................................................................ 5-26
  Using Table Parameters .............................................................................. 5-26
  When to Use Table Parameters .................................................................. 5-26
  Setting Table Parameters Using JDBC ....................................................... 5-27
Accessing Custom Database Functions Using JDBC ....................................... 5-32
Accessing Data Services Using SQL-Based Applications ............................. 5-34
Accessing Data Services Using SQL Explorer ............................................... 5-34
Connecting to the Data Services Platform Client Using OpenLink ODBC-JDBC Bridge .............................................................. 5-38
Using OpenLink with Reporting Tools ........................................................... 5-39
  Crystal Reports XI ....................................................................................... 5-41
  Business Objects XI-Release 2 (ODBC) ...................................................... 5-43
  Microsoft Access 2003-ODBC ................................................................... 5-47
6. Using Excel to Access Data Services
   Introducing the Excel Add-in ................................................................. 6-2
   Installing the Excel Add-in ................................................................. 6-2
   System Requirements ........................................................................ 6-2
   Installation Instructions .................................................................... 6-2
   Accessing Excel Add-in Documentation ........................................... 6-6
   Preparing to Use the Excel Add-in ...................................................... 6-6
   Accessing the Data Service Using Excel .......................................... 6-8

7. Accessing Data Services Through a WebLogic Workshop Control
   Introduction to Data Service Controls ............................................. 7-2
   Data Service Controls Defined .......................................................... 7-2
   Description of the Data Service Control File ..................................... 7-3
   Design View ...................................................................................... 7-3
   Source View ..................................................................................... 7-4
   Creating Data Service Controls ......................................................... 7-6
   Prerequisites ...................................................................................... 7-7
   Step 1: Create a Project ................................................................. 7-7
   Step 2: Start WebLogic Server ....................................................... 7-7
   Step 3: Create a Package under src Folder ..................................... 7-7
   Step 4: Create the Data Service Control ........................................ 7-7
   Using ALDSP Control for Ad Hoc Queries ........................................ 7-12
   ALDSP Control Backward Compatibility ......................................... 7-13
   Modifying Existing ALDSP Control ................................................. 7-14
   Adding and Removing Operations Used by a Control ........................ 7-14
8. Supporting ADO.NET Clients

Overview of ADO.NET Integration in Data Services Platform

Understanding ADO.NET

Understanding How ALDSP Supports ADO.NET Clients

Supporting Java Clients

Enabling ALDSP Support for ADO.NET Clients

Generating an ALDSP Web Services Mapper

Creating a Web Reference in ADO.NET Client by Providing the ALDSP WSDL URL

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

XML Type Requirements for Working With ADO.NET DataSets

References

Creating a Data Service Based on an RPC-Style Web Service

Generated Artifacts Reference

XML Schema Definition for ADO.NET Types DataSet

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

ALDSP Control Source Upgrade ................................. 9-1
Accessing Metadata Using Catalog Services .................. 9-2
Installing Catalog Services .................................... 9-3
Using Catalog Services ......................................... 9-4
  Application (application.ds) .................................. 9-4
  DataService (DataService.ds) ................................. 9-5
  DataServiceRef (DataServiceRef.ds) ......................... 9-6
  Folder (folder.ds) ............................................. 9-8
  Function (Function.ds) ....................................... 9-9
  Relationship (Relationship.ds) ............................. 9-12
  Schema (Schema.ds) .......................................... 9-15
Filtering, Sorting, and Fine-tuning Query Results .......... 9-17
  Introducing the Filter API .................................. 9-19
    addFilter() .................................................. 9-19
    createFilter() ............................................. 9-20
    addOrderBy() ................................................ 9-20
    setLimit() .................................................. 9-21
  Exploring the Filter Operators ............................. 9-21
  Using Filters ............................................... 9-22
  Filtering Examples .......................................... 9-23
    Specifying a Compound Filter ............................. 9-24
    Ordering and Truncating Data Service Results .......... 9-25
  Using Ad Hoc Queries to Fine-tune Results from the Client .. 9-26
CHAPTER 1

Introducing Data Services for Client Applications

This chapter provides an overview of ALDSP for client application developers. It includes the following topics:

- Introduction
- What Is a Data Service?
- What is an ALDSP Client Application?
- Choosing a Client Programming Model
- Introducing Service Data Objects (SDO)
- Introducing The Data Service Mediator API
- Typical Client Application Development Process
- Security Considerations in Client Applications
- Client Classpath Settings
- Performance Considerations
Introduction

BEA AquaLogic Data Services Platform (ALDSP) brings data access into the world of service-oriented architecture (SOA). ALDSP enables organizations to consolidate, integrate, transform, and service-enable disparate data sources scattered throughout their enterprise, making enterprise data available as an easy-to-access, reusable commodity: a data service.

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, assembling it from multiple sources and transforming 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 ALDSP. 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.

With ALDSP, 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. For client developers, ALDSP 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.

This document describes how to create ALDSP-aware client applications. It explains the various client access mechanisms that ALDSP 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 ALDSP update framework.

- 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 ALDSP Administration Guide.

What Is a Data Service?

From a high-level perspective, a data service defines a distinct business entity such as a customer and the customer’s orders. The data service defines a unified view of the business entity 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 as a client, you need know only a few details, such as:

- The name of the data service.
- The functions and procedures exposed by the data service.
- The data types associated with the data service.

Data service client applications can use data services in the same way that a web service client application invokes the operations of a Web service.

For detailed information on developing data services, see the *Data Services Developer’s Guide*.

**What is an ALDSP Client Application?**

An ALDSP 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, JDBC/ODBC, or web-service based applications in any programming language.

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

- The ALDSP JDBC driver provides JDBC clients, such as reporting tools, with SQL-based read access to ALDSP data. (ODBC clients can use a JDBC bridge to connect to the data.)
- Other web-service based applications can access data service operations through ALDSP’s web services API.
Figure 1-1 provides an overview of these multiple access methods.

Regardless of the client type, ALDSP provides a uniform, service-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, a client simply invokes the data service routine; in turn, ALDSP:

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

The ALDSP data objects conform to the Service Data Object (SDO 2.1) specification, a Java-based API for data programming that is the result of joint effort by BEA, IBM, Oracle, SAP, and others.
Choosing a Client Programming Model

Application developers can choose from among several client API models for accessing ALDSP services. The model chosen will depend on the desired access mechanism. Each access method has its own advantages and uses. **Table 1-2** provides a description of each of these access methods and summarizes the advantages of the various programming models for accessing ALDSP data services.

**Table 1-2  Summary of Techniques for Exposing Data Services to Clients**

<table>
<thead>
<tr>
<th>Data Access Technique</th>
<th>Description</th>
<th>Advantages</th>
<th>Other Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Data Service Mediator</td>
<td>Instantiate a remote data service interface and invoke public methods on the interface. See Chapter 3, “Invoking Data Services from Java Clients.”</td>
<td>Full read/write access to data.</td>
<td>Requires adequate Java programming skills.</td>
</tr>
<tr>
<td>Web services</td>
<td>Data services can be directly mapped to web services. Clients have access to data through SOAP messages and/or SDOs. See Chapter 4, “Invoking Data Services Through Web Services.”</td>
<td>Read/write access to data. Industry standard.</td>
<td>N/A</td>
</tr>
<tr>
<td>SQL</td>
<td>Data service functions first need to be published as SQL objects. These SQL objects are then available to your application through JDBC. See Chapter 5, “Using SQL to Access Data Services.”</td>
<td>Accepted by commonly used reporting tools.</td>
<td>Read-only, and for use SQL-based clients only.</td>
</tr>
</tbody>
</table>
Introducing Data Services for Client Applications

Introducing Service Data Objects (SDO)

Service Data Objects (SDO), a specification proposed jointly by BEA, IBM, Oracle, SAP, and others, is a Java-based API for data programming. SDO simplifies data programming against data sources of different types. 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. Under this architecture, a client gets a copy of externally persisted data in an SDO data object or data graph, which is a structure for holding data. The client operates on the data remotely; that is, while disconnected from the data source.

If the client makes data changes that need to be saved to the data source, a connection to the source is re-acquired later. Keeping connections active for the minimum time possible maximizes scalability and performance of web and service-oriented applications.

To SDO clients, the data has a uniform appearance no matter where it originated or what its underlying source format is. Enabling this unified view of data in the SDO model is the concept of a data 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. ALDSP implements such an SDO mediator.

For details on SDO, see Chapter 2, “Data Programming Model and Update Framework.”

---

Table 1-2  Summary of Techniques for Exposing Data Services to Clients

<table>
<thead>
<tr>
<th>Technique</th>
<th>Description</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Excel</td>
<td>The ALDSP Excel add-in lets you invoke data service operations from Microsoft® Excel®. Real-time data can be quickly rendered in the familiar Microsoft Excel format.</td>
<td>Requires Data Services Studio</td>
</tr>
<tr>
<td>ADO.NET</td>
<td>Allows interoperability between ALDSP data services and ADO.NET. Enables ALDSP data services to be used in Microsoft ADO.NET client applications.</td>
<td>Specific to ADO.NET applications.</td>
</tr>
</tbody>
</table>

See Chapter 6, “Using Excel to Access Data Services.”

See Chapter 8, “Supporting ADO.NET Clients.”

Excel The ALDSP Excel add-in lets you invoke data service operations from Microsoft® Excel®.

See Chapter 6, “Using Excel to Access Data Services.”

Real-time data can be quickly rendered in the familiar Microsoft Excel format.

Requires Data Services Studio

ALDSP implements such an SDO mediator.

Requires Data Services Studio

Real-time data can be quickly rendered in the familiar Microsoft Excel format.

Requires Data Services Studio

ALDSP implements such an SDO mediator.

Requires Data Services Studio
Introducing The Data Service Mediator API

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

The Data Service Mediator facilitates access and updates to the various data sources that comprise any data service. The Mediator is also the core mechanism for the data service update framework. For details on using the Mediator API for web services clients and for Java clients, see:

- Chapter 3, “Invoking Data Services from Java Clients”
- Chapter 4, “Invoking Data Services Through Web Services”
Typical Client Application Development Process

Developing an ALDSP-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 ALDSP architecture; it shows available data services, including the specific functions and procedures that each data service provides.

2. Choose the data access approach that best suits your needs. (Table 1-2, “Summary of Techniques for Exposing Data Services to Clients,” on page 1-5 describes the advantages of the different access mechanisms.) The approach you choose also depends on how the data service has been deployed.

   For example, if the data service has been mapped out 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, using a control.

3. Obtain the required JAR files. (See specific chapters in this guide for JAR file requirements.)

Security Considerations in Client Applications

ALDSP administrators can control access to deployed ALDSP resources through role-based security policies. ALDSP leverages and extends the security features of the underlying WebLogic platform. Roles can be set up in the WebLogic Administration Console. (See the ALDSP Administration Guide for detailed 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 information on ALDSP security, see “Securing AquaLogic Data Services Platform Resources,” in the ALDSP Administration Guide. For complete information on WebLogic security, see Programming WebLogic Security on e-docs.
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, types, and capabilities of data sources, complexity of logical data source aggregation, and other data service design considerations can affect performance.

- **Number of clients accessing the data service.** The number of simultaneous clients can affect performance.

- **Performance of the underlying data sources.** Since data services access underlying data, the performance and availability of those systems can affect performance.

- **Hardware resources.** The number of servers, processing power, memory, network structure, 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 be aware of the performance of each underlying data source and benchmark the performance of the data services as you develop them. Use load-testing tools to determine the maximum number of clients that your deployed data services can support.

You can use ALDSP’s auditing capabilities to obtain performance profile information that you can use to identify and resolve performance problems if they occur. For detailed information on ALDSP audit capabilities see *ALDSP Administration Guide*.

Client Classpath Settings

The following tables provide classpath requirements for:

- Java mediator API client (dynamic and static)
- Web Service client (dynamic and static)
- JMX Mbean Management API client
- JDBC API Client
Java Mediator API Clients

Client applications using the ALDSP Mediator API need one of the following classpath settings:

Listing 1-1  Static Java Mediator API Client Classpath

```
CLASSPATH=
<app-static-client>.jar <= this is generated static client jar
<ALDSP_HOME>/lib/sdo.jar
<ALDSP_HOME>/lib/ld-client.jar
<WL_HOME>/common/lib/apache_xbean.jar
<WL_HOME>/server/lib/weblogic.jar
```

Listing 1-2  Dynamic Mediator API Classpath

```
CLASSPATH=
<ALDSP_HOME>/lib/sdo.jar
<ALDSP_HOME>/lib/ld-client.jar
<WL_HOME>/common/lib/apache_xbean.jar
<WL_HOME>/server/lib/weblogic.jar
```

Web Services Clients

Client applications using the ALDSP Native Web Services feature need one of the following classpath settings:

Listing 1-3  Static Web Service Client Classpath

```
CLASSPATH=
<app-static-client>.jar <= this is generated static client jar
<ALDSP_HOME>/lib/sdo.jar
<ALDSP_HOME>/lib/ld-client.jar
<WL_HOME>/common/lib/apache_xbean.jar
<WL_HOME>/server/lib/weblogic.jar
```
Listing 1-4  Dynamic Web Service Clients

CLASSPATH=
  <ALDSP_HOME>/lib/sdo.jar
  <ALDSP_HOME>/lib/ld-client.jar
  <WL_HOME>/common/lib/apache_xbean.jar
  <WL_HOME>/server/lib/weblogic.jar

JMX Mbean Management API Client Classpath

The JMX Mbean Management API needs the following classpath settings:

Listing 1-5  JMX Mbean Management API Client Classpath

CLASSPATH=
  <ALDSP_HOME>/lib/ld-server-core.jar
  <ALDSP_HOME>/lib/ld-client.jar
  <WL_HOME>/common/lib/apache_xbean.jar
  <WL_HOME>/server/lib/weblogic.jar

ALDSP JDBC API Client Classpath

The ALDSP JDBC API client needs the following classpath settings:

Listing 1-6  ALDSP JDBC API Client Classpath

CLASSPATH=
  <ALDSP_HOME>/lib/ldjdbc.jar
  <ALDSP_HOME>/lib/ld-client.jar
  <WL_HOME>/server/lib/weblogic.jar
Backward Compatibility

ALDSP 2.5 provided an interoperability JAR file (wls90interop.jar) that allowed WebLogic Server 9.2 clients to access ALDSP 2.x data services. After upgrading to ALDSP 3.0, you need to remove all uses of wls90interop.jar from both the WLS 9.2 based client CLASSPATH and the ALDSP 2.5 client CLASSPATH.
BEA AquaLogic Data Services Platform (ALDSP) implements Service Data Objects (SDO) as its data client-application programming model. This chapter discusses SDO concepts and APIs that are of interest to ALDSP client application developers.

- Introduction
- ALDSP and SDO
- Role of the Mediator API and SDO

**Introduction**

SDO is an architecture and set of APIs for working with data objects while disconnected from their source. In ALDSP, SDO-compliant data objects—whether typed or untyped data objects—are obtained from data services through Mediator APIs or through Data Service controls. (See also “Introducing Service Data Objects (SDO)” on page 1-6.)

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 data objects; 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 “Introducing The Data Service Mediator API” on page 1-7, ALDSP’s Data Service 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 ALDSP’s implementation of the SDO data programming model, as well as its update framework.

**ALDSP and SDO**

When you invoke a data service’s read operation through the Data Service Mediator API, a data object is returned. Data objects are the fundamental artifacts of the SDO data programming model.

---

**Tip:** For information on the Mediator API, see Chapter 3, “Invoking Data Services from Java Clients.”

---

Data objects represent the contents of a complex type. A data object contains properties, which represent elements and attributes. The properties can be of simple or complex types. In SDO, a simple type property is called a datatype property, while a complex type property contains a data object (which in turn has properties).

Data objects can be defined to contain a special kind of property called a change summary. A change summary is used to track changes to the data object. As changes are made to the properties (or properties of nested descendant data objects), the changes are captured in the change summary.

The change summary is used by the Mediator 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 update function succeeds, so it can support rollbacks when necessary.

A datagraph is a built-in data object type that is defined to have a change summary property. Thus it is convenient to use a datagraph to encapsulate change tracking. The datagraph has one immediate data object child, and a change summary that can track changes to this data object. **Figure 2-1** shows the structure of a datagraph.
Static and Dynamic Data Object APIs

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

- **Static.** The static data object 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 2-4, “Static (Typed) Data Object API Getters and Setters,” on page 2-5.

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

Table 2-2 summarizes the advantages of each approach.

<table>
<thead>
<tr>
<th>Data Model</th>
<th>Advantages...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Data Object API</td>
<td>• Easy-to-implement interface; code is easy to read and maintain.</td>
</tr>
<tr>
<td></td>
<td>• Compile-time type checking.</td>
</tr>
<tr>
<td></td>
<td>• Enables code-completion in Workshop for WebLogic Source View.</td>
</tr>
<tr>
<td>Dynamic Data Object API</td>
<td>• Dynamic; allows discovery.</td>
</tr>
<tr>
<td></td>
<td>• Runtime type checking.</td>
</tr>
<tr>
<td></td>
<td>• Allows for a general-purpose coding style.</td>
</tr>
</tbody>
</table>

**Static Data Object API**

SDO’s static data object 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 data service developer using Workshop for WebLogic, or by using one of the provided tools.

The generated interfaces extend the commonj.sdo.DataObject interface and provide typed getters and setters for all properties of the XML datatype.

An interface is also generated for each complex property (such as CREDIT and ORDER shown in Figure 2-3), with getters and setters for each of the properties that comprise the complex type.

For many-valued properties, a get method is generated that returns a java.util.List object. A many-valued property corresponds to an XML schema element that has minOccurs greater than one. The List returned by a get method for a many-valued property is “live.” This means that if you modify the List object, the changes are reflected directly and immediately in the containing data object.

As an example of how static data object APIs are generated, given the CUSTOMER data type shown in Figure 2-3, generating typed client interfaces results in CUSTOMER, CREDIT, ORDER, and POITEM interfaces, each of which includes getters, setters, and factory classes (for instantiating static data objects and their properties).
When you develop Java client applications that use SDO’s static data object APIs, you will import these typed interfaces into your Java client code. For example:

```java
import appDataServices.AddressDocument;
```

Table 2-4 lists static data accessor and related API methods. These methods are generated using names that match the schema names with the first letter in the name forced to be upper-case. The generated names cannot conflict with standard Java naming rules.

<table>
<thead>
<tr>
<th>Static Data Object API (Generated)</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type getProperty()</td>
<td>Returns the value of the property. Generated for boolean-valued properties.</td>
<td>String name = getLast_NAME()</td>
</tr>
<tr>
<td>List&lt;Type&gt; getProperty()</td>
<td>For multiple occurrence elements, returns all PropertyName elements.</td>
<td>List&lt;ORDER&gt; orders = getORDER()</td>
</tr>
<tr>
<td>void setName(Property new)</td>
<td>Sets the value of the property to the new Value.</td>
<td>setLAST_NAME(&quot;Smith&quot;)</td>
</tr>
<tr>
<td>boolean isChecked()</td>
<td>Determines whether the PropertyName element or attribute exists in the data object.</td>
<td>isSPECIAL_DELIVERY()</td>
</tr>
</tbody>
</table>
ALDSP client application developers can use the Data Services Platform Console to view the XML schema types associated with data services (see Figure 2-3, “CUSTOMER Return Type Displayed in DSP Console’s Metadata Browser,” on page 2-5). 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 corresponding Java types using the data type mappings shown in Table 2-5.

**Note:** The following XQuery types are discussed by the SDO specification and are listed in Table 2-5 but are not supported for input or output from the Mediator API: xs:ENTITIES, xs:ENTITY, xs:ID, xs:IDREF, xs:IDREFS, xs:language, xs:Name, xs:NCName, xs:NMTOKEN, xs:NMTOKENS, xs:NOTATION. The Mediator API is discussed in Chapter 3, “Invoking Data Services from Java Clients.”

Table 2-4 Static (Typed) Data Object API Getters and Setters

<table>
<thead>
<tr>
<th>Static Data Object API (Generated)</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>void createPropertyName()</td>
<td>Generated only for non-datatype properties. Creates a data object for the specified property. The (created) data object is initialized with no values in its properties.</td>
<td>createORDER()</td>
</tr>
<tr>
<td>boolean isSetPropertyName()</td>
<td>Determines whether the property is set to some value.</td>
<td>isSetLAST_NAME()</td>
</tr>
<tr>
<td>void unsetPropertyName()</td>
<td>Unsets the property. The property is then considered not to be set.</td>
<td>unsetLAST_NAME()</td>
</tr>
</tbody>
</table>

### XML Schema-to-Java Type Mapping Reference

ALDSP client application developers can use the Data Services Platform Console to view the XML schema types associated with data services (see Figure 2-3, “CUSTOMER Return Type Displayed in DSP Console’s Metadata Browser,” on page 2-5). 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 corresponding Java types using the data type mappings shown in Table 2-5.

**Note:** The following XQuery types are discussed by the SDO specification and are listed in Table 2-5 but are not supported for input or output from the Mediator API: xs:ENTITIES, xs:ENTITY, xs:ID, xs:IDREF, xs:IDREFS, xs:language, xs:Name, xs:NCName, xs:NMTOKEN, xs:NMTOKENS, xs:NOTATION. The Mediator API is discussed in Chapter 3, “Invoking Data Services from Java Clients.”

Table 2-5 XML Schema to Java Data Type Mapping

<table>
<thead>
<tr>
<th>XML Schema Type</th>
<th>SDO Java Type</th>
<th>XML Schema Type</th>
<th>SDO Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs:anyType</td>
<td>commonsj.sdo.DataObject</td>
<td>xs:integer</td>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>xs:anySimpleType</td>
<td>java.lang.Object</td>
<td>xs:language</td>
<td>String</td>
</tr>
<tr>
<td>xs:anyURI</td>
<td>String</td>
<td>xs:long</td>
<td>long or java.lang.Long</td>
</tr>
<tr>
<td>xs:base64Binary</td>
<td>byte[]</td>
<td>xs:Name</td>
<td>String</td>
</tr>
</tbody>
</table>
Dynamic Data Object API

Every static (typed) data object implements the Data Object interface; therefore, you can use the DataObject (dynamic) methods as well as the static API. This API provides generic property
getters and setters for specific Java data types (String, Date, List, BigInteger, and BigDecimal, for example). Table 2-6 lists representative APIs from SDO’s dynamic Data Object 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 Object 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.

As an example, assuming that you have a reference to a CUSTOMER data object, you can use the dynamic Data Object API to get the LAST_NAME property as follows:

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

The SDO APIs are standard implementations. You can read the full SDO specification, “SDO for Java Specification V2.1” here:

http://www.osoa.org/display/Main/Service+Data+Objects+Specifications

See also Service Data Objects on dev2dev.

Table 2-6 lists dynamic Data Object API getters and setters.

<table>
<thead>
<tr>
<th>Dynamic Data Object API</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>get(int PropertyIndex)</td>
<td>Returns the PropertyName child element at the specified index.</td>
<td>get(5)</td>
</tr>
<tr>
<td>set(int PropertyIndex, Object newValue)</td>
<td>Sets the value of the property to the newValue.</td>
<td>set(5, CUSTOMER3)</td>
</tr>
<tr>
<td>set(String PropertyName, Object newValue)</td>
<td>Sets the value of the PropertyName to the newValue.</td>
<td>set(&quot;LAST_NAME&quot;, &quot;Nimble&quot;)</td>
</tr>
<tr>
<td>set(commonj.sdo.Property property, Object newValue)</td>
<td>Sets the value of Property object to the newValue.</td>
<td>set(LASTNAME, &quot;Nimble&quot;)</td>
</tr>
<tr>
<td>getType(String PropertyName)</td>
<td>Returns the value of the PropertyName. Type indicates the specific data type to obtain.</td>
<td>getBigDecimal(&quot;CreditScore&quot;)</td>
</tr>
<tr>
<td>unset(int PropertyIndex)</td>
<td>Unsets the property. The property is then considered not to be set.</td>
<td>unset(5)</td>
</tr>
</tbody>
</table>
XPath Expressions in the Dynamic Data Object API

ALDSP supports a limited subset of XPath expressions called SDO path expressions. SDO path expressions offer flexibility in how you locate data objects and attributes in the dynamic Data Object 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 path implementation augments XPath 1.0 support by adding zero-based array index notation ("index_from_0") to XPath's standard bracketed notation ([n]). As an example, Table 2-7 compares the XPath standard and SDO augmented notations to refer to the same element, the first ORDER child node under CUSTOMER (Table 2-7).
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.

ALDSP fully supports both the traditional index notation and the augmented notation. Keep in mind these other points regarding ALDSP’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 ALDSP SDO implementation.

- In SDO Path, "@" has no significance and is ignored. Elements and attributes in XML Schema both map to properties in SDO, and ",@", the notation for denoting an attribute, can be used with any property; however, the ",@" will be ignored. Moreover, attributes can be referenced in SDO Path simply with the attribute name, without an "@". For example, the ID attribute of the following element:

  <ORDER ID="3434">

  is accessed with the following path:

  ORDER/@ID, or with

  Order/ID

See also “Specifying XPath Expressions as Arguments” on page 3-56.

<table>
<thead>
<tr>
<th>XPath Standard Notation</th>
<th>SDO Augmented Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>get(&quot;CUSTOMER/ORDER[1]&quot;);</td>
<td>get(&quot;CUSTOMER/ORDER.0&quot;);</td>
</tr>
</tbody>
</table>
Obtaining Type Information about Data Objects

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

According to the SDO specification, the Type interface (see Table 2-8) and the Property interface (see Table 2-9) 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:

```java
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 `getProperties()` method, as shown in Listing 2-1.

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

```java
public void printDataObject(DataObject dataObject, int indent) {  
    Type type = dataObject.getType();  
    List properties = type.getProperties();  
    for (int p=0, size=properties.size(); p < size; p++) {  
        if (dataObject.isSet(p)) {  
            Property property = (Property) properties.get(p);  
            // For many-valued properties, process a list of values  
            if (property.isMany()) {  
                List values = dataObject.getList(p);  
                for (int v=0; count=values.size(); v < count; v++) {  
                    printValue(values.get(v), property, indent);  
                }  
            } else { // For single-valued properties, print out the value  
                printValue(dataObject.get(p), property, indent);  
            }  
        }  
    }
}
```
Table 2-8 lists other useful methods in the Type interface.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java.lang.Class getInstanceClass()</code></td>
<td>Returns the Java class that this type represents.</td>
</tr>
<tr>
<td><code>java.lang.String getName()</code></td>
<td>Returns the name of the type.</td>
</tr>
<tr>
<td><code>java.lang.List getProperties</code></td>
<td>Returns a list of the properties of this type.</td>
</tr>
<tr>
<td><code>Property getProperty(java.lang.String propertyName)</code></td>
<td>Returns from among all Property objects of the specified type the one with the specified name. For example, <code>dataObject.get(&quot;name&quot;)</code> or <code>dataObject.getDataObject().getProperty(&quot;name&quot;)</code></td>
</tr>
<tr>
<td><code>java.lang.String getURI()</code></td>
<td>Returns the namespace URI of the type.</td>
</tr>
<tr>
<td><code>boolean isInstance(java.lang.Object object)</code></td>
<td>Returns True if the specified object is an instance of this type; otherwise, returns false.</td>
</tr>
</tbody>
</table>

Table 2-9 lists the methods of the Property interface.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type <code>getContainingType()</code></td>
<td>Returns the containing type of this property.</td>
</tr>
<tr>
<td><code>java.lang.Object getDefault()</code></td>
<td>Returns the default value this property will have in a data object where the property has not been set.</td>
</tr>
<tr>
<td><code>java.lang.String getName()</code></td>
<td>Returns the name of the property.</td>
</tr>
<tr>
<td>Type <code>getType()</code></td>
<td>Returns the type of the property.</td>
</tr>
<tr>
<td><code>boolean isContainment()</code></td>
<td>Returns True if the property represents by-value composition.</td>
</tr>
<tr>
<td><code>boolean isMany()</code></td>
<td>Returns True if the property is many-valued.</td>
</tr>
</tbody>
</table>
Role of the Mediator API and SDO

In ALDSP, data objects are passed between data services and client applications: when a client application invokes a read function on a data service, for example, a data object 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 object to the data service. The Data Service Mediator is an API 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 data objects (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 ALDSP’s update framework, including the Mediator, and how the default update process works, are described in more detail in “Managing Update Maps” in the Data Services Development Guide.

For detailed information on using the Mediator APIs for web services clients and Java clients, see:

- Invoking Data Services from Java Clients
- Invoking Data Services Through Web Services
This chapter discusses the Data Services Mediator API, a Java API for invoking data service operations from Java applications. This chapter explains in detail how to use the Mediator API and includes working sample applications to help you get started.

Using the Mediator API is one of several techniques for invoking data services from client applications. See Chapter 1, “Introducing Data Services for Client Applications” for a summary of these techniques.

This chapter includes these topics:

- Introducing the Mediator API
- Getting Started
- Sample Static Mediator Application
- Sample Dynamic Mediator Application
- Creating New DataObjects
- Mediator API Basics
- Mapping Data Service Types to Java Types
- Web Services Support
- Advanced Topics
Introducing the Mediator API

The Mediator API is the Java API for retrieving Service Data Object (SDO) artifacts from a data service and returning them to their source. In your Java client, you call Mediator API methods to connect to a data service, invoke data service methods, and send updated data objects back to the server. You use SDO API methods to manipulate the data objects within your Java client.

For example, you might call a Mediator API method getAllCustomers() to retrieve a collection of customer data objects from the data service. Then, you could call an SDO method such as setCustomerName() to modify a customer object. Finally, you might call another Mediator API method, such as updateCustomers() to return the modified data object to the data source on the server.

Topics in this section include:
- What is SDO?
- What is the Mediator API?
- Dynamic and Static Mediator APIs
- Summary

What is SDO?

The Java programming model provided by ALDSP for invoking data service operations is based on Service Data Objects (SDO). SDO, a specification proposed jointly by BEA, IBM, Oracle, SAP, and others, is a Java-based architecture and API for data programming. ALDSP lets programmers uniformly access data objects from heterogeneous data sources, including relational databases, XML data sources, web services, and enterprise information systems.

Tip: See “Introducing Service Data Objects (SDO)” on page 1-6 for a general overview of SDO. For a more in-depth discussion of SDO, see “Data Programming Model and Update Framework” on page 2-1. Finally, see the dev2dev article Service Data Objects, which provides links to the SDO specifications and Javadoc.
What is the Mediator API?

While the SDO specification does not specify a mechanism for updating data objects, it does discuss the need for update services, called mediator services. The Mediator API is an ALDSP implementation of a mediator service. The Mediator API lets you gain access to SDO-compliant objects, called DataObjects, and return them to their source data store.

The important points to remember are that the Mediator API lets you connect to a data service and invoke data service operations. Results are returned as SDO-compliant data objects. Using methods of the SDO API, you can then change or manipulate the data objects. Finally, you use the Mediator API to perform the update.

See “ALDSP and SDO” on page 2-2 for a general overview of SDO data objects and other artifacts.

Dynamic and Static Mediator APIs

The ALDSP Mediator API comprises two main interfaces: dynamic and static. As an application developer, you need to choose one of these approaches.

- The Dynamic Mediator API is useful for programming with data services that are unknown or do not exist at development time. This API is useful, for example, for developing tools and user interfaces that work across data services. The Dynamic Mediator API lets you invoke data service operations directly by name. Clients that use the Dynamic Mediator API are not bound to use specific data services: a dynamic client can use any available data service. See “Sample Dynamic Mediator Application” on page 3-23.

- The Static Mediator extends the Dynamic Mediator with pre-generated Java classes that have type-safe named methods for accessing data service operations. A Static Mediator API must be explicitly generated using either the IDE or a command line utility. The generated API classes are placed in a JAR file that must be accessible by your client application. See “Sample Static Mediator Application” on page 3-9.

**Tip:** For most use cases, the Static Mediator API is your best choice. The Static Mediator inherits from the Dynamic Mediator and therefore includes all of the functionality of the Dynamic Mediator API. In addition, the static API is type-safe at compile-time. Generally speaking, the static API is simpler and more convenient to use than the Dynamic Mediator API.
**API Overview**

The Dynamic Mediator API consists of the classes and interfaces listed in Table 3-1. Refer to the Javadoc on e-docs for more information on these classes and interfaces.

<table>
<thead>
<tr>
<th>Interface or Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DataAccessService</strong></td>
<td>The interface for interacting with a data service. The invoke() method of this interface is used to call data service operations. If a data service operation returns a result, the invoke() method returns a DASResult object—a collection of SDO data objects or simple types. (Package: com.bea.dsp.das)</td>
</tr>
<tr>
<td><strong>DASResult</strong></td>
<td>The Mediator APIs that return data sets return an object called DASResult (Data Access Service Result). DASResult is similar to a Java Iterator. See “Understanding DASResult” on page 3-41. (Package: com.bea.dsp.das)</td>
</tr>
<tr>
<td><strong>PreparedExpression</strong></td>
<td>The 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. See “Making Ad Hoc Queries” on page 3-57. (Package: com.bea.dsp)</td>
</tr>
<tr>
<td><strong>DataAccessServiceFactory</strong></td>
<td>The factory class for creating local interfaces to data services. Can be used for dynamic data service instantiation and ad hoc queries. (Package: com.bea.dsp.das)</td>
</tr>
<tr>
<td><strong>HelperContextCache</strong></td>
<td>ALDSP maintains a global cache of SDO HelperContext objects. These objects can be used, for instance, to create new data objects. This class contains methods that let you query and manipulate this cache. See “Creating New DataObjects” on page 3-28. (Package: com.bea.dsp.das)</td>
</tr>
<tr>
<td><strong>RequestConfig</strong></td>
<td>This class encapsulates a collection of attributes that control how a data service method is to be invoked from a client. This class also serves as a way to return arbitrary information to the client. (Package: com.bea.dsp)</td>
</tr>
<tr>
<td><strong>SDOUtil</strong></td>
<td>This utility class contains methods for manipulating SDO data objects in the context of ALDSP. While not part of the Mediator API, this utility class is commonly used in programs that use the Mediator API. (Package: com.bea.dsp.sdo)</td>
</tr>
</tbody>
</table>
Sample of both Static and Dynamic Mediator clients applications are provided. See “Sample Static Mediator Application” on page 3-9 and “Sample Dynamic Mediator Application” on page 3-23.

**Summary**

It may be confusing at first discussing SDO and the mediator APIs together. You can think of SDO as the standard enabling technology that allows client applications to access and update data through ALDSP data services. SDO has a Java API for handling DataObjects and collections of DataObjects. SDO DataObjects can be either dynamic or static.

The SDO APIs are standard implementations. You can read the full SDO specification, “SDO for Java Specification V2.1” here:

http://www.osoa.org/display/Main/Service+Data+Objects+Specifications

The mediator APIs, on the other hand, are ALDSP-specific implementations. The mediator APIs are designed to let you access SDO DataObjects and return them to the server. For more information on the how ALDSP uses SDO, see Chapter 2, “Data Programming Model and Update Framework.”

**Getting Started**

This section lists the basic steps to get started writing a Java client application that interacts with a data service.

Topics in this section include:

- Basic Steps
- Setting the CLASSPATH
- Running the Sample Applications
Basic Steps

These are the basic steps to follow when developing a Java client that uses the Mediator APIs.

1. The first thing you need is a data service to call. To use a data service, you need to know its name and the names and signatures of its operations. The mediator API method signatures will be the same as the signatures for the data service operations.

   **Tip:** You can discover data services that are available to you by using the ALDSP Console. See Viewing Metadata Using the Service Explorer in the ALDSP Administration Guide.

2. Decide whether to use the Static or Dynamic Mediator API to interact with the data service from your Java client. See “Dynamic and Static Mediator APIs” on page 3-3 for a summary of each API. To use the Static Mediator API, you need to generate or obtain the Static Mediator Client JAR file. For instructions on generating a Static Mediator Client JAR, see the Data Services Developer’s Guide.

   **Tip:** The Static Mediator API is generally recommended for most use cases. The static API is type safe and generally easier to use than the Dynamic Mediator API.

3. Set up your Java build environment. You need certain JAR files in your CLASSPATH. See “Setting the CLASSPATH” on page 3-7 for details.

4. Write and test your client application. This document provides working sample applications that demonstrate both the Static and Dynamic Mediator API. See “Running the Sample Applications” on page 3-9.
Setting the CLASSPATH

CLASSPATH settings depend on whether you are using the Static or Dynamic Mediator API.

Static Java Mediator API Client CLASSPATH

The following JARs must be in the CLASSPATH of your Java application if you are using the Static Mediator API.

- `<Static Mediator Client File>.jar`
  
  **Note:** This first entry is only required if you are using the Static Mediator API. This entry must precede the other entries listed below. For instructions on generating a Static Mediator Client JAR, see the *Data Services Developer’s Guide*. Typically, this file is generated by a data services developer.

- `<ALDSP_HOME>/lib/sdo.jar`
- `<ALDSP_HOME>/lib/ld-client.jar`
- `<WEBLOGIC_HOME>/common/lib/apache_xbean.jar`
- `<WEBLOGIC_HOME>/server/lib/weblogic.jar`

Dynamic Java Mediator API Client CLASSPATH

The following JARs must be in the CLASSPATH of your Java application if you are using the Dynamic Mediator API.

- `<ALDSP_HOME>/lib/sdo.jar`
- `<ALDSP_HOME>/lib/ld-client.jar`
- `<WEBLOGIC_HOME>/common/lib/apache_xbean.jar`
- `<WEBLOGIC_HOME>/server/lib/weblogic.jar`
Specifying the Class Loader Directly

Certain application contexts, such as web applications, employ their own class loaders. In these cases, you must take steps to ensure that the static mediator classes use the correct class loader. If you do not take these steps, class cast exceptions can occur.

To ensure that your static mediator classes resolve properly in such contexts, you can pass the appropriate class loader object to com.bea.dsp.das.HelperContextCache.setClassLoader() before creating DataAccessService or PreparedExpression objects in your code. The example code in Listing 3-1 shows one possible variation on this approach, where the class loader is obtained from the current thread object. This variation works well for web applications deployed on WebLogic Server.

Listing 3-1  Example Code: Getting and Setting the Class Loader

```java
import com.bea.dsp.das.HelperContextCache;
...
ClassLoader sdoCompiledSchemaLoader =
    Thread.currentThread().getContextClassLoader();
HelperContextCache.setClassLoader(dataSpaceName, sdoCompiledSchemaLoader);
...
DataAccessService das = DataAccessServiceFactory.newDataAccessService(ctx,
    dataSpaceName, dsUri);
...
```

Note: In Listing 3-1, you could use the following code to obtain a PreparedExpression object:

```java
PreparedExpression pe = DataAccessServiceFactory.prepareExpression(ctx,
    dspDataSpace, adhoc);
```

Other possible approaches to obtaining the class loader object include:

- `this.getClass().getClassLoader()` – This option is typically used when invoking code from JPD, JPF, and JWS applications.
- `classname.class.getClassLoader()` – This option can be used in place of the previous option inside a static method (where you cannot use the `this` keyword to refer to the current object).
Sample Static Mediator Application

- `fully-qualified-name-of-compiled-sdo-class-or-interface.class.getClassLoader()` – This is the most general-purpose option.

- `external-Java-function-class.class.getClassLoader()` – Use this option for external Java functions.

Running the Sample Applications

A good way to get started is to run the sample application code that is provided in this chapter. Samples that use both the Static and the Dynamic Mediator APIs are included. The samples illustrate simple but common use cases: retrieving data, modifying it, and updating it. See “Sample Static Mediator Application” on page 3-9 and “Sample Dynamic Mediator Application” on page 3-23.

Sample Static Mediator Application

This section presents a simple Java program that you can copy, compile, and run. The program uses the Static Mediator API to perform these basic tasks: authenticating the client, retrieving data, modifying data, and updating data on the server. For a basic overview of the Static Mediator API, see “Dynamic and Static Mediator APIs” on page 3-3. See also “Mediator API Basics” on page 3-39 and “Advanced Topics” on page 3-51.

Topics include:

- Setting Up the Sample Data Service
- Generating the Mediator Client JAR File
- Setting Up the Java Project
- Running and Testing the Code
- Examining the Sample Code
Setting Up the Sample Data Service

Before you can build and test the sample Java application, you need to set up an ALDSP data service. The instructions assume that you are familiar with the ALDSP perspective in the Eclipse IDE, as described in the Data Services Developer’s Guide.

Note: The sample Java client that is presented in this section calls operations in this sample data service. The sample Java code is designed to work with this specific data service.

1. Install ALDSP.
2. In the Eclipse IDE, create a server that uses the ALDSP samples domain. The samples domain is: <ALDSP_HOME>/samples/domain/aldsp.
3. Start the server.
4. Create an ALDSP Dataspace called MediatorSamples.
5. Copy the sample data service (Listing 3-2, “CUSTOMER.ds,” on page 3-11) into a file called MediatorSamples/Retail/CUSTOMER.ds.
6. Copy the schema file (Listing 3-3, “CUSTOMER_KEY.xsd,” on page 3-14) into a file called MediatorSamples/Retail/schemas/CUSTOMER_KEY.xsd.
7. Copy the schema file (Listing 3-4, “CUSTOMER.xsd,” on page 3-14) into a file called MediatorSamples/Retail/schemas/CUSTOMER.xsd.

Figure 3-2 shows the resulting Dataspace configuration:
Note: Listing 3-2 is a simple data service file, containing the XQuery code that defines the service and its operations. Listing 3-3 and Listing 3-4 are schema files that are required by the data service. The Mediator API lets you invoke the data service operations from a Java client. For more information on data services, see Data Services Developer’s Guide.

Listing 3-2 CUSTOMER.ds

```xml
<xquery version "1.0" encoding "UTF-8">

<::pragma xds <x:xds targetType="t:CUSTOMER"
xmlns:x="urn:annotations.ld.bea.com" xmlns:t="ld:Retail/CUSTOMER">
<creationDate>2007-11-08T17:13:51</creationDate>
<relationalDB name="dspSamplesDataSource" providerId="Pointbase"/>
<field xpath="CUSTOMER_ID" type="xs:short">
    <extension nativeXpath="CUSTOMER_ID" nativeTypeCode="5"
        nativeType="SMALLINT" nativeSize="5" nativeFractionalDigits="0"
        nativeKey="true">
        <autoNumber type="identity"/>
    </extension>
    <properties nullable="false"/>
</field>
<field xpath="FIRST_NAME" type="xs:string">
    <extension nativeXpath="FIRST_NAME" nativeTypeCode="12"
        nativeType="VARCHAR" nativeSize="64" nativeFractionalDigits="0"/>
    <properties nullable="false"/>
</field>
<field xpath="LAST_NAME" type="xs:string">
    <extension nativeXpath="LAST_NAME" nativeTypeCode="12"
        nativeType="VARCHAR" nativeSize="64" nativeFractionalDigits="0"/>
    <properties nullable="false"/>
</field>
<field xpath="CUSTOMER_SINCE" type="xs:date">
    <extension nativeXpath="CUSTOMER_SINCE" nativeTypeCode="91"
        nativeType="DATE" nativeSize="10" nativeFractionalDigits="0"/>
    <properties nullable="false"/>
</field>
<field xpath="EMAIL_ADDRESS" type="xs:string">
    <extension nativeXpath="EMAIL_ADDRESS" nativeTypeCode="12"
        nativeType="VARCHAR" nativeSize="32" nativeFractionalDigits="0"/>
    <properties nullable="false"/>
```
Invoking Data Services from Java Clients

</field>
<field xpath="TELEPHONE_NUMBER" type="xs:string">
  <extension nativeXpath="TELEPHONE_NUMBER" nativeTypeCode="12"
    nativeType="VARCHAR" nativeSize="32" nativeFractionalDigits="0"/>
  <properties nullable="false"/>
</field>
<field xpath="SSN" type="xs:string">
  <extension nativeXpath="SSN" nativeTypeCode="12" nativeType="VARCHAR"
    nativeSize="16" nativeFractionalDigits="0"/>
  <properties nullable="true"/>
</field>
<field xpath="BIRTH_DAY" type="xs:date">
  <extension nativeXpath="BIRTH_DAY" nativeTypeCode="91" nativeType="DATE"
    nativeSize="10" nativeFractionalDigits="0"/>
  <properties nullable="true"/>
</field>
<field xpath="DEFAULT_SHIP_METHOD" type="xs:string">
  <extension nativeXpath="DEFAULT_SHIP_METHOD" nativeTypeCode="12"
    nativeType="VARCHAR" nativeSize="16" nativeFractionalDigits="0"/>
  <properties nullable="true"/>
</field>
<field xpath="EMAIL_NOTIFICATION" type="xs:short">
  <extension nativeXpath="EMAIL_NOTIFICATION" nativeTypeCode="5"
    nativeType="SMALLINT" nativeSize="5" nativeFractionalDigits="0"/>
  <properties nullable="true"/>
</field>
<field xpath="NEWS_LETTER" type="xs:short">
  <extension nativeXpath="NEWS_LETTER" nativeTypeCode="5"
    nativeType="SMALLINT" nativeSize="5" nativeFractionalDigits="0"/>
  <properties nullable="true"/>
</field>
<field xpath="ONLINE_STATEMENT" type="xs:short">
  <extension nativeXpath="ONLINE_STATEMENT" nativeTypeCode="5"
    nativeType="SMALLINT" nativeSize="5" nativeFractionalDigits="0"/>
  <properties nullable="true"/>
</field>
<field xpath="LOGIN_ID" type="xs:string">
  <extension nativeXpath="LOGIN_ID" nativeTypeCode="12" nativeType="VARCHAR"
    nativeSize="50" nativeFractionalDigits="0"/>
  <properties nullable="true"/>
</field>
declare namespace f1 = "ld:Retail/CUSTOMER";

import schema namespace t1 = "ld:Retail/CUSTOMER" at "ld:Retail/schemas/CUSTOMER.xsd";

import schema "ld:Retail/CUSTOMER" at "ld:Retail/schemas/CUSTOMER_KEY.xsd";

(pragma function <f:function xmlns:f="urn:annotations.ld.bea.com"
   visibility="public" kind="read" isPrimary="false" nativeName="CUSTOMER"
   nativeLevel2Container="SAMPLECUSTOMER" style="table">
   <nonCacheable/>   </f:function>::)

declare function f1:CUSTOMER() as schema-element(t1:CUSTOMER)* external;

(pragma function <f:function xmlns:f="urn:annotations.ld.bea.com"
   visibility="public" kind="create" isPrimary="true" nativeName="CUSTOMER"
   nativeLevel2Container="SAMPLECUSTOMER" style="table">
   <nonCacheable/>   </f:function>::)

declare procedure f1:createCUSTOMER($p as element(t1:CUSTOMER)*)as
   schema-element(t1:CUSTOMER_KEY)* external;

(pragma function <f:function xmlns:f="urn:annotations.ld.bea.com"
   visibility="public" kind="update" isPrimary="true" nativeName="CUSTOMER"
   nativeLevel2Container="SAMPLECUSTOMER" style="table">
   <nonCacheable/>   </f:function>::)

declare procedure f1:updateCUSTOMER($p as changed-element(t1:CUSTOMER)*) as
   empty()  external;

(pragma function <f:function xmlns:f="urn:annotations.ld.bea.com"
   visibility="public" kind="delete" isPrimary="true" nativeName="CUSTOMER"
   nativeLevel2Container="SAMPLECUSTOMER" style="table">
   <nonCacheable/>   </f:function>::)
declare procedure f1:deletecustomerId($p as element(t1:CUSTOMER]*) as empty()
    external;

---

### Listing 3-3  CUSTOMER_KEY.xsd

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema targetNamespace="ld:Retail/CUSTOMER"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="CUSTOMER_KEY">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="CUSTOMER_ID" type="xs:short"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

### Listing 3-4  CUSTOMER.xsd

```xml
<xs:schema targetNamespace="ld:Retail/CUSTOMER"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="CUSTOMER">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="CUSTOMER_ID" type="xs:short" minOccurs="0"/>
                <xs:element name="FIRST_NAME" type="xs:string"/>
                <xs:element name="LAST_NAME" type="xs:string"/>
                <xs:element name="CUSTOMER_SINCE" type="xs:date"/>
                <xs:element name="EMAIL_ADDRESS" type="xs:string"/>
                <xs:element name="TELEPHONE_NUMBER" type="xs:string"/>
                <xs:element name="SSN" type="xs:string" minOccurs="0"/>
                <xs:element name="BIRTH_DAY" type="xs:date" minOccurs="0"/>
                <xs:element name="DEFAULT_SHIP_METHOD" type="xs:string" minOccurs="0"/>
                <xs:element name="EMAIL_NOTIFICATION" type="xs:short" minOccurs="0"/>
                <xs:element name="NEWSLETTER" type="xs:short" minOccurs="0"/>
                <xs:element name="ONLINE_STATEMENT" type="xs:short" minOccurs="0"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```
Generating the Mediator Client JAR File

The sample Java application listed later in this section requires that you first generate a Mediator Client JAR from the data service. The classes in this JAR contain type-safe methods that call the data service functions and procedures. The generated Java methods have the same names as their corresponding data service functions and procedures.

Tip: You can generate a Mediator Client JAR file using the IDE, the ALDSP Console, or an Ant script. These methods are described in detail in the Data Services Developer’s Guide. For this example, we will use the IDE.

To generate a mediator client JAR file using the IDE:

1. Select File > Export.
2. In the Select dialog, select AquaLogic Data Services Platform > Mediator Client JAR File and click Next.
3. Complete the Mediator Client JAR File dialog as follows:
   - Select the Dataspace project to export. For this example, the Dataspace project is called MediatorSamples.
   - Specify a directory in which to place the exported JAR file. You can select any location on your system. By default, the exported JAR will be named: MediatorSamples-dsp-client.jar.
4. Click Finish.

Tip: For detailed information on how generated class names in the JAR file are derived, see “Naming Conventions for Generated Classes” on page 3-41.
Setting Up the Java Project

Listing 3-6 lists the sample Java program that uses the Static Mediator API. The application simply retrieves a DataObject from a data store, modifies the object, and returns it to the data store. This example assumes you are using the Eclipse IDE, but you can use the IDE or build environment of your choice. For this example, we set up an Eclipse Java project called MediatorClient.

To set up the project:

1. Create a Java project called MediatorClient.

2. Set up your Java Build Path to include the JAR files listed in “Setting the CLASSPATH” on page 3-7. To do this, select Project > Properties > Java Build Path. Be sure to include the Mediator Client JAR file, as discussed in “Generating the Mediator Client JAR File” on page 3-15.

3. Create a package called com.bea.dsp.sample in your Java project. To do this, right-click the Java project in the Package Explorer and select New > Package.

4. Create a Java class called StaticSampleApp.java in the package. To do this, right-click the package in the Package Explorer and select New > Class.

5. Delete the default contents of the new source file and copy the entire file listed in Listing 3-5 into the source file.

6. Save the file. Figure 3-3 shows the completed project configuration in the Eclipse IDE.
Note: The imported classes CUSTOMERDAS and CUSTOMER (see Listing 3-5) are located in the Static Mediator Client JAR file, which must be in the CLASSPATH.

Listing 3-5 StaticSampleApp.java

```java
package com.bea.dsp.sample;

import das.ejb.retail.CUSTOMERDAS;
import retail.customer.CUSTOMER;

import com.bea.dsp.das.DASResult;
import com.bea.dsp.sdo.SDOUtil;

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

public class StaticSampleApp {
    public static void main(String[] args) throws Exception {
        // Create InitialContext for mediator
        Hashtable<String, String> hash = new Hashtable<String, String>();
```
hash.put(Context.INITIAL_CONTEXT_FACTORY,
          "weblogic.jndi.WLInitialContextFactory");
hash.put(Context.PROVIDER_URL,"t3://localhost:7001");
hash.put(Context.SECURITY_PRINCIPAL,"weblogic");
hash.put(Context.SECURITY_CREDENTIALS,"weblogic");
Context ctx = new InitialContext(hash);

// Create DataAccessService handle with Context and dataspace name
CUSTOMERDAS das = CUSTOMERDAS.getInstance(ctx, "MediatorSamples");

// Invoke the basic 'get all customers' function
DASResult<CUSTOMER> result = das.CUSTOMER();

// Obtain the first CUSTOMER DataObject - also be sure to
// always dispose() any DASResults
try {
    CUSTOMER customer = result.next();

    // Enable change-tracking for that CUSTOMER
    SDOUtil.enableChanges(customer);

    // Modify customer
    customer.setFIRST_NAME("New First Name");
    customer.setEMAIL_ADDRESS("first_name@example.com");

    // Send changes back to DSP - update function takes an array
    // of CUSTOMERs
    das.updateCUSTOMER(new CUSTOMER[] { customer });
}
finally {
    result.dispose();
}
Running and Testing the Code

To test the application:

1. Start the server.
2. Run the Java client as a Java application. In Eclipse, this is commonly done by right-clicking the Java file and selecting Run As > Java Application.

To verify that the Java client worked, simply test the data service:

1. Open the data service in the Data Service editor.
2. Click the Test tab (see Figure 3-4).
3. Select an operation from the drop down menu. For this example, select the CUSTOMER() operation.
4. Click Run (see Figure 3-4).
5. Inspect the first row of the data table. The client application changes the first customer’s name and email address to “New First Name” and “first_name@example.com” as shown in Figure 3-4.

Figure 3-4 Testing the Client

![Figure 3-4 Testing the Client](image)
Examining the Sample Code

This section examines the parts of the Java sample in Listing 3-5. This section discusses:

- Importing Packages
- Obtaining a Data Access Service Handle
- Retrieving Data from the Service
- Obtaining a DataObject from the Result
- Disposing the Result Object
- Modifying the DataObject
- Returning Changes to the Server

Importing Packages

The first two classes are located in the generated Mediator Client JAR file, which must be in your build path. The CUSTOMERDAS class is the generated DataAccessService class for the data service. This class contains type-safe methods that map to the actual data service operations. The CUSTOMER class provides the SDO interface for manipulating DataObjects returned from the data service.

```java
import das.ejb.retail.CUSTOMERDAS;
import retail.customer.CUSTOMER;
import com.bea.dsp.das.DASResult;
import com.bea.dsp.sdo.SDOUtil;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
```
Obtaining a Data Access Service Handle

A DataAccessService object lets you call methods on a data service. See the Javadoc for more information on this class. For the Static Mediator API, DataAccessService (DAS) classes have a factory method named getInstance() to return the handle.

The getInstance() method requires two parameters to return the handle:

- A WebLogic JNDI Context object. The Context object allows the Java client to connect to the data service running through WebLogic Server. See “Obtaining the WebLogic JNDI Context for ALDSP” on page 3-44. For more information on WebLogic JNDI context objects, see Programming WebLogic JNDI on e-docs.

- The name of the Dataspace project in which the data service is deployed. In this sample, the project is called MediatorSamples.

```java
Hashtable<String, String> hash = new Hashtable<String, String>();
hash.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContextFactory");
hash.put(Context.PROVIDER_URL,"t3://localhost:7001");
hash.put(Context.SECURITY_PRINCIPAL,"weblogic");
hash.put(Context.SECURITY_CREDENTIALS,"weblogic");
Context ctx = new InitialContext(hash);
CUSTOMERDAS das = CUSTOMERDAS.getInstance(ctx, "MediatorSamples");
```

Retrieving Data from the Service

The generated DataAccessService method CUSTOMER() retrieves the result set from the data service. This method returns all customer objects from the data service. The return type is a DASResult object, which works like an iterator. For more information on this return type, see “Understanding DASResult” on page 3-41.

```java
DASResult<CUSTOMER> result = das.CUSTOMER();
```

Note: The method CUSTOMER() is mapped directly from the original no-argument data service operation of the same name. The operation definition as specified in the data service file looks like this:

```xml
<function xmlns:f="urn:annotations.ld.bea.com"
  visibility="public" kind="read" isPrimary="false" nativeName="CUSTOMER"
  nativeLevel2Container="SAMPLECUSTOMER" style="table">
  <nonCacheable/>
</function>
```

```
declare function f1:CUSTOMER() as schema-element(t1:CUSTOMER)* external;
```

The entire data service file is shown in Listing 3-2.
Obtaining a DataObject from the Result

The DASResult.next() method works very much like the Java method Iterator.next(). It returns the next CUSTOMER, which is an SDO DataObject. SDO is a Java-based data programming model (API) and architecture for accessing and updating data. For details on SDO, see Using Service Data Objects (SDO) in the ALDSP Concepts Guide.

CUSTOMER customer = result.next();

Disposing the Result Object

You must call DASResult.dispose() whenever you are finished iterating through a result object. For more information on dispose(), see “Disposing of DASResult Objects” on page 3-42.

result.dispose();

Tip: Placing the dispose() call in a try/finally block is a recommended best practice.

Modifying the DataObject

After you obtain a DataObject, you can modify it; however, if you intend to submit these changes back to the ALDSP server, you must enable change-tracking on the DataObject before making any modifications. The SDOUtil.enableChanges() method lets you enable change-tracking for a single DataObject or an array of DataObjects. For more information on this method, see “Working with Data Objects” on page 3-45. After the customer object has change-tracking enabled, the generated setters are called to modify certain values in the customer object.

Tip: Note that the set method below is called on an SDO DataObject. Technically, such methods are part of the SDO API, not the Mediator API. See Chapter 2, “Data Programming Model and Update Framework” for information on SDO.

SDOUtil.enableChanges(customer);

// Modify customer
customer.setFIRST_NAME("New First Name");
customer.setEMAIL_ADDRESS("first_name@example.com");
Returning Changes to the Server

Finally, the generated DataAccessService.updateCUSTOMER() method is called with a single parameter: an array of CUSTOMER objects. The method calls its equivalent data service operation to update the database with the newly modified row of data.

das.updateCUSTOMER(new CUSTOMER[] { customer });

Tip: In this example, the update method generated by ALDSP accepts an array of DataObjects. It accepts an array because the data service operation (created by the data service developer) accepts an array of data objects. If the data service developer had created an additional update method that accepted a single CUSTOMER, it would not be necessary to put the customer DataObject into an array.

Sample Dynamic Mediator Application

This section presents a simple example that you can copy, compile, and run. This example uses the Dynamic Mediator API to perform these basic tasks: authenticating the client, retrieving data, modifying data, and updating data on the server.

The topics in this section include:

- Setting Up and Running the Sample Code
- Sample Java Client Code (Dynamic Mediator API)
- Examining the Sample Code

Setting Up and Running the Sample Code

To set up and run this sample code, follow the basic instructions in “Sample Static Mediator Application” on page 3-9. The procedures for creating a sample data service, setting up the Java project, and running the program are the same as the Static Mediator sample; however, when using the Dynamic Mediator API, you do not need to generate or reference the Static Mediator Client JAR file. Use the sample Java code shown in Listing 3-6 in your project.
package com.bea.dsp.sample;

import com.bea.dsp.das.DataAccessServiceFactory;
import com.bea.dsp.das.DataAccessService;
import com.bea.dsp.das.DASResult;
import com.bea.dsp.sdo.SDOUtil;
import commonj.sdo.DataObject;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;

public class DynamicSampleApp {
    public static void main(String[] args) throws Exception {
        // Create InitialContext for mediator
        Hashtable<String, String> hash = new Hashtable<String, String>();
        hash.put(Context.INITIAL_CONTEXT_FACTORY,
            "weblogic.jndi.WLInitialContextFactory");
        hash.put(Context.PROVIDER_URL,"t3://localhost:7001");
        hash.put(Context.SECURITY_PRINCIPAL,"weblogic");
        hash.put(Context.SECURITY_CREDENTIALS,"weblogic");
        Context ctx = new InitialContext(hash);

        // Create DataAccessService handle with Context, dataspace
        // name, and data service URI
        DataAccessService das = DataAccessServiceFactory.newDataAccessService(
            ctx, "MediatorSamples", "ld:Retail/CUSTOMER");

        // Invoke the basic 'get all customers' function, which takes
        // no arguments
        DASResult<Object> result = das.invoke("CUSTOMER", new Object[0]);

        // Obtain the first CUSTOMER DataObject - also be sure to
        // always dispose() any DASResults
```java
try {
    DataObject customer = (DataObject) result.next();

    // Enable change-tracking for that CUSTOMER
    SDOUtil.enableChanges(customer);

    // Modify customer
    customer.set("FIRST_NAME", "DynamicClient");
    customer.set("EMAIL_ADDRESS", "dynamic@example.com");

    // Send changes back to DSP - update function takes an array
    // of CUSTOMERS
    das.invoke("updateCUSTOMER", new Object[]{ customer });
}
finally {
    result.dispose();
}
```

### Examining the Sample Code

This section examines the parts of the Java sample in Listing 3-6. This section discusses:

- Importing Classes
- Obtaining a DataAccessService Handle
- Retrieving Data from the Service
- Obtaining a DataObject from the Result
- Disposing the Result Object
- Modifying the DataObject
- Returning Changes to the Server
Importing Classes

These classes are required by the sample. For detailed information on the classes, refer to the Javadoc on e-docs.

```java
import com.bea.dsp.das.DataAccessServiceFactory;
import com.bea.dsp.das.DataAccessService;
import com.bea.dsp.das.DASResult;
import com.bea.dsp.sdo.SDOUtil;
import commonj.sdo.DataObject;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
```

Obtaining a DataAccessService Handle

A DataAccessService object lets you call methods on a data service. See the Javadoc for more information on this class. The DataAccessServiceFactory class requires three parameters to return the handle:

- A WebLogic JNDI Context object. The Context object allows the Java client to connect to the data service running through WebLogic Server and provides security attributes. See “Obtaining the WebLogic JNDI Context for ALDSP” on page 3-44. For more information on WebLogic JNDI context objects, see Programming WebLogic JNDI on e-docs.

- The name of the Dataspace project in which the data service is deployed.

- The name of the data service as based on its location in the Dataspace’s folder hierarchy.

Here is the code:

```java
Hashtable<String, String> hash = new Hashtable<String, String>();
hash.put(Context.INITIAL_CONTEXT_FACTORY, 
"weblogic.jndi.WLInitialContextFactory");
hash.put(Context.PROVIDER_URL,"t3://localhost:7001");
hash.put(Context.SECURITY_PRINCIPAL,"weblogic");
hash.put(Context.SECURITY_CREDENTIALS,"weblogic");
Context ctx = new InitialContext(hash);

DataAccessService das = DataAccessServiceFactory.newDataAccessService
(ctx, "MediatorSamples", "ld:Retail/CUSTOMER");
```
Retrieving Data from the Service

In this example, the invoke() method calls the data service CUSTOMER operation. This operation returns all customer objects from the data service. The invoke() method returns a DASResult object, which works like an iterator. For more information on this return type, see “Understanding DASResult” on page 3-41. Note that the CUSTOMER operation takes no arguments.

```java
DASResult<Object> result = das.invoke("CUSTOMER", new Object[0])
```

**Note:** The generic type parameter for DASResult is <Object> because data of any type can be returned by the invoke() method of the Dynamic Mediator API.

Obtaining a DataObject from the Result

The DASResult.next() method works very much like the Java method Iterator.next(). It returns the next object in the result set. Because the CUSTOMER data service method returns SDO-compliant DataObjects, you can cast the return value to DataObject. SDO is a Java-based data programming model (API) and architecture for accessing and updating data. For details on SDO, see Using Service Data Objects (SDO) in the ALDSP Concepts Guide. See also “What is SDO?” on page 3-2.

```java
DataObject customer = (DataObject) result.next();
```

Disposing the Result Object

You must call DASResult.dispose() whenever you are finished iterating through a result object. For more information on dispose(), see “Disposing of DASResult Objects” on page 3-42.

```java
result.dispose();
```

**Tip:** Placing the dispose() call in a try/finally block is a recommended best practice.

Modifying the DataObject

After you obtain a DataObject, you can modify it; however, if you intend to submit these changes back to the ALDSP server, you must enable change-tracking on the DataObject before making any modifications. The SDOUtil.enableChanges() method lets you enable change-tracking for a single DataObject or an array of DataObjects. For more information on this method, see “Working with Data Objects” on page 3-45. After the customer object has change-tracking enabled, the Dynamic SDO set() method is called to modify certain values in the customer object.
For more information on SDO methods, see Chapter 2, “Data Programming Model and Update Framework.”

```java
SDOUtil.enableChanges(customer);

customer.set("FIRST_NAME", "DynamicClient");
customer.set("EMAIL_ADDRESS", "dynamic@example.com");
```

### Returning Changes to the Server

Finally, the DataAccessService method `invoke()` calls the `update` method on the data service with a single parameter: an array of CUSTOMER objects. The data service operation updates the database with the newly modified row of data.

```java
das.invoke("updateCUSTOMER", new Object[] { customer });
```

**Tip:** In this example, the update method accepts an array of DataObjects. It accepts an array because the data service operation (created by the data service developer) accepts an array of data objects. If the data service developer had created an additional update method that accepted a single CUSTOMER, it would not be necessary to put the customer DataObject into an array.

---

### Creating New DataObjects

This section explains how to use the Data Services Mediator and SDO APIs to create new data objects and submit them to the ALDSP server. As with previous examples, both the static and dynamic APIs are illustrated.

#### Creating a New DataObject with the Static API

The Java program in Listing 3-7 creates a new DataObject, modifies it, and updates it on the ALDSP server.

#### Setting Up and Running the Sample

The sample code in Listing 3-7 is designed to work in the same Java project and with the same data service project that are described in “Sample Static Mediator Application” on page 3-9. You can run the sample code presented here by following the setup instructions in that section. When you test the data service, you will see a new row has been added to the table.
Listing 3-7  StaticCreateSample.java

package com.bea dsp.sample;

import das.ejb.retail.CUSTOMERDAS;
import retail.customer.CUSTOMER;
import retail.customer.CUSTOMER_KEY;

import com.bea dsp.das.DASResult;
import com.bea dsp.sdo.SDOUtil;

import commonj.sdo.helper.HelperContext;
import commonj.sdo.helper.DataFactory;

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

public class StaticCreateSample {
  public static void main(String[] args) throws Exception {
    // Create InitialContext for mediator
    Hashtable<String, String> hash = new Hashtable<String, String>();
    hash.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContextFactory");
    hash.put(Context.PROVIDER_URL, "t3://localhost:7001");
    hash.put(Context.SECURITY_PRINCIPAL, "weblogic");
    hash.put(Context.SECURITY_CREDENTIALS, "weblogic");
    Context ctx = new InitialContext(hash);

    // Create DataAccessService handle with Context and dataspace name
    CUSTOMERDAS das = CUSTOMERDAS.getInstance(ctx, "MediatorSamples");

    // Obtain the SDO HelperContext for this dataspace
    HelperContext hctx = das.getHelperContext();
    // Could also use:
    // HelperContext hctx = HelperContextCache.get("MediatorSamples");

    // Get DataFactory from HelperContext
    DataFactory factory = hctx.getDataFactory();

    // Create an "empty" CUSTOMER DataObject by naming the XML
// schema "type". For schema global elements that do not
// explicitly specify a type, their type name will be the same
// as the element name.
CUSTOMER customer = (CUSTOMER) factory.create
         ("ld:Retail/CUSTOMER", "CUSTOMER");

// Have to provide this DataObject with its own name. Note
// that this is the XML schema "name", not the "type" -
// although as noted, when the global element does not
// explicitly specify a type, the type that is provided for it
// has the same name as the element.
SDOUtil.setElementName(customer, "ld:Retail/CUSTOMER", "CUSTOMER");

// Note that you must NOT enable change-tracking for this
// DataObject using enableChanges(). Change-tracking is only
// for tracking changes to data originally received from the
// DSP server.

// Set fields on new DataObject. Don't set auto-generated
// fields, such as CUSTOMER_ID. May omit optional fields or
// those with default values
customer.setFIRST_NAME("New First Name");
customer.setLAST_NAME("New Last Name");
customer.setCUSTOMER_SINCE("2007-10-18");
customer.setEMAIL_ADDRESS("first_name@example.com");
customer.setTELEPHONE_NUMBER("867-5309");

// Send new DataObject to DSP - create function takes an array
// of CUSTOMERS, and returns CUSTOMER_KEYS
DASResult<CUSTOMER_KEY> result =
         das.createCUSTOMER(new CUSTOMER[] { customer });

// Can obtain new customer ID from the returned key - also be
// sure to always dispose() any DASResults.
try {
      CUSTOMER_KEY key = result.next();
      System.out.println("New customer key: " + key.getCUSTOMER_ID());
}
finally {
      result.dispose();
}
Creating New DataObjects

// Note that the created DataObject is NOT automatically
// updated based on the generated key values. If you want to
// get a DataObject populated with the new CUSTOMER_ID, you
// need to re-read. This is easier if the data service
// architect provides a getByID() function on the data
// service.

Importing Packages

Two SDO classes are required by this program. A HelperContext provides access to a consistent
set of instances of SDO helpers. It represents a helper execution context. The set of helpers
returned by the methods in this interface have visibility to the same SDO metadata, that is, they
execute in the same “scope.” A DataFactory is a helper for the creation of DataObjects. The
created DataObjects are not connected to any other DataObjects. Only Types with DataType false
and abstract false may be created.

Obtaining a Data Access Service Handle

A DataAccessService object lets you call methods on a data service. See the Javadoc for more
information on this class. For the Static Mediator API, DataAccessService (DAS) classes have a
factory method named getInstance() to return the handle.

The getInstance() method requires two parameters to return the handle:

- A WebLogic JNDI Context object. The Context object allows the Java client to connect to
  the data service running through WebLogic Server. See “Obtaining the WebLogic JNDI
  Context for ALDSP” on page 3-44. For more information on WebLogic JNDI context
  objects, see Programming WebLogic JNDI on e-docs.

- The name of the Dataspace project in which the data service is deployed. In this sample,
  the project is called MediatorSamples.

```java
Hashtable<String, String> hash = new Hashtable<String, String>();
hash.put(Context.INITIAL_CONTEXT_FACTORY,
         "weblogic.jndi.WLInitialContextFactory");
hash.put(Context.PROVIDER_URL,"t3://localhost:7001");
hash.put(Context.SECURITY_PRINCIPAL,"weblogic");
hash.put(Context.SECURITY_CREDENTIALS,"weblogic");
Context ctx = new InitialContext(hash);
```
CUSTOMERDAS das = CUSTOMERDAS.getInstance(ctx, "MediatorSamples");

Creating a DataFactory
To create a DataFactory, you need to first obtain a HelperContext object for the Dataspace.

    HelperContext hctx = das.getHelperContext();
    DataFactory factory = hctx.getDataFactory();

You could also use this call to return the HelperContext:

    HelperContext hctx = HelperContextCache.get("MediatorSamples");

Create and Name the DataObject
After you create a DataObject, you must explicitly name it. The factory.create() method takes two
String parameters. The first is a URI, the location of the data service in the Dataspace project. The
second parameter is the XML schema type of the DataObject you are creating. For schema global
elements that do not explicitly specify a type, their type name will be the same as the element
name.

    CUSTOMER customer = (CUSTOMER) factory.create("ld:Retail/CUSTOMER", "CUSTOMER");

Next, you must provide the new DataObject with a name. The SDOUtil.setElementName()
method takes these parameters: the DataObject, the namespace URI of the element QName, and
the local part of the element QName.

    SDOUtil.setElementName(customer, "ld:Retail/CUSTOMER", "CUSTOMER");

Modifying the DataObject
After you create a new DataObject, you can modify it before submitting it to the server.

    customer.setFIRST_NAME("New First Name");
    customer.setLAST_NAME("New Last Name");
    customer.setEMAIL_ADDRESS("first_name@example.com");
    customer.setTELEPHONE_NUMBER("867-5309");
Tip: You can omit optional fields or fields with default values.

Returning New DataObject to the Server

After the new object is created, the data service operation createCUSTOMER is called from the Static Mediator API. The data service create operation takes an array of objects as input. The Mediator API method returns a CUSTOMER_KEY objects in a DASResult.

```java
DASResult<CUSTOMER_KEY> result = das.createCUSTOMER(new CUSTOMER[] { customer });
```

Returning the New DataObject Key

To return the CUSTOMER_KEY for the new CUSTOMER object, call the next() method on the DASResult object. Be sure to dispose the DASResult object (result) after it is returned. Placing dispose() in a try/finally block is a recommended best practice.

```java
try {
    CUSTOMER_KEY key = result.next();
    System.out.println("New customer key: " + key.getCUSTOMER_ID());
} finally {
    result.dispose();
}
```

Tip: The newly created local copy of the DataObject (customer, in this example) is not automatically updated with generated keys such as CUSTOMER_ID. If you want to obtain a DataObject populated with CUSTOMER_ID, you need to retrieve the new DataObject from the server by invoking the data service’s read operation. This is easier if the data service developer provides a getbyID() operation on the data service.

Creating a New DataObject with the Dynamic API

The Java program in Listing 3-7 creates a new DataObject, modifies it, and updates it on the ALDSP server.

Running the Sample

The sample code in Listing 3-8 is designed to work in the same Java project and with the same data service project that are described in “Sample Dynamic Mediator Application” on page 3-23. You can run the sample code presented here by following the setup instructions in that section.
Listing 3-8 DynamicCreateSample.java

```java
package com.bea.dsp.sample;

import com.bea.dsp.das.DataAccessService;
import com.bea.dsp.das.DataAccessServiceFactory;
import com.bea.dsp.das.DASResult;
import com.bea.dsp.das.HelperContextCache;
import com.bea.dsp.sdo.SDOUtil;
import commonj.sdo.helper.HelperContext;
import commonj.sdo.helper.DataFactory;
import commonj.sdo.DataObject;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;

public class DynamicCreateSample {
    public static void main(String[] args) throws Exception {
        // Create InitialContext for mediator
        Hashtable<String, String> hash = new Hashtable<String, String>();
        hash.put(Context.INITIAL_CONTEXT_FACTORY,
            "weblogic.jndi.WLInitialContextFactory");
        hash.put(Context.PROVIDER_URL,"t3://localhost:7001");
        hash.put(Context.SECURITY_PRINCIPAL,"weblogic");
        hash.put(Context.SECURITY_CREDENTIALS,"weblogic");
        Context ctx = new InitialContext(hash);

        // Obtain the SDO HelperContext for this dataspace. As with
        // StaticCreateSample, I could obtain this from the
        // DataAccessService. However, here I'm demonstrating how to
        // create an all-new DataObject prior to creating any
        // DataAccessService instance. In this case, since I'm using
        // the dynamic mediator, I need to first ensure that the
        // global HelperContext cache is populated with the schemas
        // for my data service.
        HelperContextCache.loadSchemasForDataspace(ctx, "MediatorSamples", "ld:Retail/CUSTOMER");

        // Now that the schemas are loaded, I can get the
    }
}
```
// HelperContext for the dataspace
HelperContext hctx = HelperContextCache.get("MediatorSamples");

// Get DataFactory from HelperContext
DataFactory factory = hctx.getDataFactory();

// Create an "empty" CUSTOMER DataObject by naming the XML
// schema *type*. For schema global elements that do not
// explicitly specify a type, their type name will be the same
// as the element name.
DataObject customer = factory.create("ld:Retail/CUSTOMER", "CUSTOMER");

// Have to provide this DataObject with its own name. Note
// that this is the XML schema *name*, not the *type* -
// although as noted, when the global element does not
// explicitly specify a type, the type that is provided for it
// has the same name as the element.
SDOUtil.setElementName(customer, "ld:Retail/CUSTOMER", "CUSTOMER");

// Note that you must NOT enable change-tracking for this
// DataObject using enableChanges(). Change-tracking is only
// for tracking changes to data originally received from the
// DSP server.

// Set fields on new DataObject. Don't set auto-generated
// fields, such as CUSTOMER_ID. May omit optional fields or
// those with default values
customer.set("FIRST_NAME", "Dynamic");
customer.set("LAST_NAME", "Mediator");
customer.set("CUSTOMER_SINCE", "2007-10-18");
customer.set("EMAIL_ADDRESS", "dynamic@example.com");
customer.set("TELEPHONE_NUMBER", "867-5309");

// Create DataAccessService handle with Context and dataspace name
DataAccessService das = DataAccessServiceFactory.newDataAccessService(ctx, "MediatorSamples", "ld:Retail/CUSTOMER");

// Send new DataObject to DSP - create function takes an array
// of CUSTOMERS, and returns CUSTOMER_KEYS
DASResult<Object> result =
    das.invoke("createCUSTOMER", new Object[] { customer });
// Can obtain new customer ID from the returned key. Always be
// sure to dispose() any DASResults you get.
try {
    DataObject key = (DataObject) result.next();
    System.out.println("New customer key: " + key.get("CUSTOMER_ID"));
}
finally {
    result.dispose();
}

// Note that the created DataObject is NOT automatically
// updated based on the generated key values. If you want to
// get a DataObject populated with the new CUSTOMER_ID, you
// need to re-read. This is easier if the data service
// architect provides a getByID() function on the data
// service.

---

**Importing Packages**

Three SDO classes are required by this program. A DataObject is a representation of some structured data. It is the fundamental component in the SDO (Service Data Objects) package. A HelperContext provides access to a consistent set of instances of SDO helpers. It represents a helper execution context. The set of helpers returned by the methods in this interface have visibility to the same SDO metadata, that is, they execute in the same “scope.” A DataFactory is a helper for the creation of DataObjects. The created DataObjects are not connected to any other DataObjects. Only Types with DataType false and abstract false may be created.

This example also uses the com.bea dsp.das.HelperContextCache class, which provides access to the global cache of SDO HelperContext objects maintained by ALDSP. The use of HelperContextCache is described in the next section “Creating a DataFactory” on page 3-36.

**Creating a DataFactory**

As with the static mediator example discussed previously, we need to obtain the SDO HelperContext for this Dataspace. In the static example, we created the HelperContext from the DataAccessService. However, in this example, we create a new DataObject before we create the
Creating New DataObjects

DataAccessService instance. To do this, you need to ensure that the global HelperContext cache is populated with the data service schemas.

```
HelperContextCache.loadSchemasForDataspace
    (ctx, "MediatorSamples", "ld:Retail/CUSTOMER");
    // Now that the schemas are loaded, get the HelperContext for the dataspace.
    HelperContext hctx = HelperContextCache.get("MediatorSamples");

    DataFactory factory = hctx.getDataFactory();
```

**Tip:** For detailed information on HelperContextCache, refer to the Javadoc.

Create and Name the DataObject

When you create a DataObject, you must explicitly name it. The factory.create() method takes two String parameters. The first is a URI, the location of the data service in the Dataspace project. The second parameter is the XML schema type of the DataObject you are creating. For schema global elements that do not explicitly specify a type, their type name will be the same as the element name.

```
DataObject customer = factory.create("ld:Retail/CUSTOMER", "CUSTOMER");
```

Next, you must provide the new DataObject with a name. The SDOUtil.setElementName() method takes these parameters: the DataObject, the namespace URI of the element QName, and the local part of the element QName.

```
SDOUtil.setElementName(customer, "ld:Retail/CUSTOMER", "CUSTOMER");
```

Modifying the DataObject

After you create a new DataObject, you can modify it before submitting it to the server.

**Note:** You must not enable change-tracking in this new DataObject using the SDOUtil.enableChanges() method. Change-tracking is only used for tracking changes to data that was originally received from the ALDSP server.

```
customer.set("FIRST_NAME", "New First Name");
customer.set("LAST_NAME", "New Last Name");
customer.set("EMAIL_ADDRESS", "first_name@example.com");
customer.set("TELEPHONE_NUMBER", "867-5309");
```
**Tip:** You can omit optional fields or fields with default values.

**Returning New DataObject to the Server**

You need a DataAccessService handle to call methods on the data service. A DataAccessService object lets you call methods on a data service. See the Javadoc for more information on this class. The DataAccessServiceFactory class requires three parameters to return the handle:

- A WebLogic JNDI Context object. The Context object allows the Java client to connect to the data service running through WebLogic Server and provides security attributes. See “Obtaining the WebLogic JNDI Context for ALDSP” on page 3-44. For more information on WebLogic JNDI context objects, see Programming WebLogic JNDI on e-docs.
- The name of the Dataspacet project in which the data service is deployed.
- The name of the data service as based on its location in the Dataspaces’s folder hierarchy.

The DataAccessServiceFactory returns the handle.

```java
// Create DataAccessService handle with Context and dataspace name
DataAccessService das = DataAccessServiceFactory.newDataAccessService(ctx, "MediatorSamples", "ld:Retail/CUSTOMER");
```

The `DataAccessService.invoke()` method is used to call the `createCUSTOMER` data service operation on the server. Note that the `createCUSTOMER` data service operation is designed to take an array of objects as input. The function returns a `CUSTOMER_KEYS` objects in the DASResult object.

```java
DASResult<Object> result = das.invoke("createCUSTOMER", new Object[] { customer });
```

**Returning the New DataObject Key**

To return the key for the new CUSTOMER object, call the `next()` method on the DASResult object. Be sure to dispose the DASResult object (result) after it is returned. Placing `dispose()` in a try/finally block is a recommended best practice.

```java
try {
    DataObject key = (DataObject) result.next();
    System.out.println("New customer key: " + key.get("CUSTOMER_ID"));
} finally {
    result.dispose();
}
```
Tip: The newly created local copy of the DataObject (customer, in this example) is not automatically updated with generated keys such as CUSTOMER_ID. If you want to obtain a DataObject populated with CUSTOMER_ID, you need to retrieve the new DataObject from the server by invoking the data service’s read operation. This is easier if the data service developer provides a getByPrimaryKey() operation on the data service.

Mediator API Basics

This section discusses various Mediator API topics.

- Beyond the Sample Applications
- More on the Static Mediator API
- More on the Dynamic Mediator API
- Naming Conventions for Generated Classes
- Understanding DASResult
- Obtaining the WebLogic JNDI Context for ALDSP
- Working with Data Objects

Beyond the Sample Applications

It is recommended that you review and run the sample applications provided in this chapter:

- “Sample Static Mediator Application” on page 3-9
- “Sample Dynamic Mediator Application” on page 3-23

Although the sample code is very basic, it demonstrates common use cases of retrieving, modifying, and updating data. The samples also include details to help you understand the code. The rest of this chapter discusses additional features of the APIs as well as advanced topics and important reference material.
More on the Static Mediator API

When called through the Static Mediator API, data service operations that return empty() or that return a single item do not return DASResult; instead, they return void or the single item. See also “Understanding DASResult” on page 3-41.

More on the Dynamic Mediator API

This section provides additional information on the Dynamic Mediator API.

- Invoking Data Service Operations
- Getters and Setters

Invoking Data Service Operations

The invoke(String method, Object[] args) method dynamically invokes data service operations. When an operation is invoked (getCustomerByCustID(), for example), it returns a DASResult object. All data service functions return a DASResult when called through the Dynamic Mediator API. See also “Understanding DASResult” on page 3-41.

You can see the invoke() method in use in the Dynamic Mediator API sample in Listing 3-6, “DynamicSampleApp.java,” on page 3-24:

```java
DASResult<Object> result = das.invoke("updateCustomer", new Object[0]);
```

More information on the invoke() method is available in Javadoc on e-docs.

Getters and Setters

SDO provides generic getters and setters for working with data objects. The SDO API can be used with data types that have not yet been deployed at development time. XPath expressions are passed as arguments to the generic methods. For example:

```java
customer.set("EMAIL_ADDRESS", "first_name@example.com");
```

or

```java
String name = customer.get("EMAIL_ADDRESS");
```

See also “Specifying XPath Expressions as Arguments” on page 3-56 and Chapter 2, “Data Programming Model and Update Framework.”
Naming Conventions for Generated Classes

When you generate a Mediator Client JAR file or a Web Services Mediator Client JAR file, the generated DataAccessService subclasses and packages are named according to the following conventions:

**Mediator Client JAR Naming Convention**

Generated DataAccessService subclasses are named `<Data_Service_Name>DAS.class`. For example, if you generate a Mediator Client JAR file from a data service called Customer.ds, a class called CustomerDAS.class is generated in the JAR file. Package names contain das.ejb.

**Web Services Mediator Client JAR Naming Convention**

Generated DataAccessService subclasses are named `<Data_Service_Name>DAS.class`. For example, if you generate a Web Services Mediator Client JAR from a web service map file called Customer.ws, a class called CustomerDAS.class is generated in the JAR file. Package names contain das.ws.

Understanding DASResult

The mediator APIs that return data sets return an object called DASResult (Data Access Service Result). DASResult is similar to a Java Iterator.

This section includes these topics:

- Overview of DASResult
- Disposing of DASResult Objects
- Dynamic Mediator APIs and DASResult
- Static Mediator APIs and DASResult
- Retrieving an Array of Objects
Overview of DASResult

By default, data is returned to the Mediator from the ALDSP server in small blocks. This “streaming” behavior means that large result sets are never held in memory all at once on either the ALDSP server or the client application, which optimizes memory utilization. However, this requires that resources on the server be held open until all results have been returned to the client. See “Support for Stateless Operations” on page 3-54.

For example, the signature for the invoke() method is:

```java
DASResult<Object> invoke(String operation, Object[] args) throws DASException;
```

Like an Iterator object, DASResult is forward-only; there is no way to return to a previous item nor to restart the iteration.

DASResult includes these standard Java Iterator methods:

- Object next() throws DASException;
- boolean hasNext();

**Note:** The Java Iterator remove() method is not included because DASResult is a read-only object.

All complex XML items in the DASResult object are represented as DataObjects. All simple items (which can be returned from library data service operations) in the result are represented by a corresponding Java object, such as Integer, Long, and so on. For information on how types are mapped to schema types, see “Mapping Data Service Types to Java Types” on page 3-46. See also “Making Ad Hoc Queries” on page 3-57.

**Tip:** All mediator methods that return DASResult never return NULL; if the data service function returns no results, then the DASResult iterates through zero items. That is, hasNext() immediately returns false and next() returns NULL.

Disposing of DASResult Objects

The server is required to hold open resources such as database handles until the code finishes iterating through all of the results. Therefore, you are required to dispose of returned DASResult objects to tell the server to release the resources.
DASResult includes the following methods:

void dispose() – Disposes of resources required by DASResult on the ALDSP server.

boolean isDisposed() – Returns whether the connection to the ALDSP server has been closed.

**Note:** You must call dispose() when you are finished iterating through a DASResult. If you call dispose() on a result object that has already been disposed, nothing happens, and no error is generated.

The dispose() method is automatically called for you in the following two cases:

- You use the FETCH_ALL_IMMEDIATELY feature of the RequestConfig object. For more information this feature, see “Support for Stateless Operations” on page 3-54.
- You use the DASResult.getItems() method. For more information, see “Retrieving an Array of Objects” on page 3-44.

**Dynamic Mediator APIs and DASResult**

All Dynamic Mediator API methods that return XQuery results return a DASResult object. These methods include:

- The basic invoke() method
- The form of invoke() that takes a RequestConfig object. See “API Overview” on page 3-4 for information on RequestConfig.
- All forms of PreparedExpression.executeQuery() for ad-hoc queries. For details on forming ad-hoc queries, see “Using Ad Hoc Queries to Fine-tune Results from the Client” on page 9-26.

**Static Mediator APIs and DASResult**

In the Static Mediator API, all generated methods for data service operations that have plural results are declared to return a DASResult<T>. Plural results are results of data service operations whose XQuery return type is type* (zero or more instances of the type) or type+ (one or more instances of the type). T is the class of DataObjects that are returned by the DASResult.next().

Generated methods for data service operations that have a maximum of one return value (that is, data service operations whose XQuery return type is type or type?) will be declared to return the corresponding Java type directly, rather than a DASResult object. In addition, a data service operation whose return type is empty() will generate a static mediator method with a return type of void. See “Mapping Data Service Types to Java Types” on page 3-46 for more information.
Retrieving an Array of Objects

DASResult includes a method `T[] getItems()`. This method returns the results as an array. This method immediately disposes the DASResult. See “Disposing of DASResult Objects” on page 3-42.

- You must call getItems() before any calls to DASResult.next(). If you call getItems() after any calls to next() an IllegalStateException is thrown.
- If you call next() after calling getItems(), an IllegalStateException is thrown.
- When you call getItems(), all results are materialized in memory on the client at once.

Obtaining the WebLogic JNDI Context for ALDSP

Java client applications use JNDI to access named objects, such as data services, on a WebLogic Server. To use any of the Mediator APIs, you need to obtain the WebLogic Server JNDI context for ALDSP. This context allows the mediator APIs to call data service operations and acquire information from data services. For more information on WebLogic JNDI context objects, see Programming WebLogic JNDI on e-docs.

Use the following call to obtain the JNDI context. The hashtable parameter is explained below.

```
InitialContext jndiCtxt = new InitialContext(hashtable);
```

Table 3-5 lists the keys and values that you can insert into the hashtable parameter.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context.INITIAL_CONTEXT_FACTORY</td>
<td>weblogic.jndi.WLInitialContextFactory</td>
</tr>
<tr>
<td>Context.PROVIDER_URL</td>
<td>URL of the WebLogic Server hosting ALDSP. For example: t3://localhost:7001.</td>
</tr>
<tr>
<td>Context.SECURITY_PRINCIPAL</td>
<td>(optional) A username</td>
</tr>
<tr>
<td>Context.SECURITY_CREDENTIALS</td>
<td>(optional) A password</td>
</tr>
</tbody>
</table>

Listing 3-9 shows example code for obtaining the JNDI context.
Listing 3-9  Obtaining the JNDI Context

```java
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);
```

Working with Data Objects

When you invoke a data service operation using the Mediator API, a collection of SDO-compliant data objects is returned in a DASResult object. (See also “Understanding DASResult” on page 3-41.)

This section discusses working with DataObjects within the context of a Java client. For more details on SDO data objects and the SDO API, see Chapter 2, “Data Programming Model and Update Framework.”

Enabling Data Objects for Change Tracking

Before you make any changes to a DataObject, you must enable it for change tracking. To do this, pass the DataObject to the com.bea dsp.sdo.SDOUtil.enableChanges() method. For example:

```java
SDOUtil.enableChanges(customer);
```

There are two forms of enableChanges(). One takes a DataObject and the other takes an array of DataObjects:

```java
com.bea.dsp.sdo.SDOUtil.enableChanges(DataObject);
com.bea.dsp.sdo.SDOUtil.enableChanges(DataObject[]);
```

**Tip:** When a DataObject that is enabled for changes is returned to the server, it contains its original data and its changed data. The mechanics of handling changed data is somewhat complex; therefore, the SDOUtil.enableChanges() utility method was created to handle those details.
Modifying Data Object Properties

After you pass a DataObject to enableChanges(), you can make any allowable modifications to the DataObject using the standard SDO APIs. See Chapter 2, “Data Programming Model and Update Framework” for detailed information on the SDO API interfaces.

SDO provides static (typed) and dynamic (untyped) interfaces. For details, see “Static Data Object API” on page 2-4 and “Dynamic Data Object API” on page 2-7.

Example static (typed) method call:

```java
customer.setFIRST_NAME("New First Name");
```

Example dynamic (untyped) method call:

```java
customer.set("FIRST_NAME", "New First Name");
```

After an SDO object is enabled for change and modified, it can be passed as an argument to an update method. ALDSP then handles the details of performing the update. For example, from Listing 3-5, “StaticSampleApp.java,” on page 3-17:

```java
das.updateCUSTOMER(new CUSTOMER[] { customer });
```

Creating a New Data Object

You can use the API to create a completely new data object. In RDBMS terms this would be considered creating a new record. Data object creation is an advanced topic. For detailed information, see “Creating New DataObjects” on page 3-28.

Mapping Data Service Types to Java Types

This section explains how types in data services are mapped to Java types by the Mediator API. For example, the Static Mediator API generator makes these type conversions when creating a Mediator Client JAR file. This section also helps you understand how argument types passed to Mediator API methods are mapped to corresponding XQuery types.
Topics in this section include:

- Conversion of Simple Types
- Conversion of Date/Time Types
- Passing Empty Sequence Arguments
- Quantified Return Types
- What is Autoboxing?

Note: ALDSP 3.2 enables you to employ user-derived simple types as parameters or return types in data service operations. See “Support for Derived Simple Types” in the ALDSP 3.2 New Features Supplement.

Conversion of Simple Types

Table 3-6 specifies how simple XQuery types are converted to Java types by the Mediator API. For example, a data service operation that returns xs:int produces a Java method that returns a Java Integer object. An operation that returns xs:int* (zero or more ints) returns a DASResult<Integer> object. (See also “Static Mediator APIs and DASResult” on page 3-43.)

Note: Simple types in are mapped to Java Objects when returned from the mediator in a manner that is identical to the SDO for Java Specification V2.1. You can find this specification online at: www.osoa.org/display/Main/Service+Data+Objects+Specifications.

Note: The following XQuery types are discussed by the SDO specification but are not supported for input or output from the Mediator API: xs:ENTITIES, xs:ENTITY, xs:ID, xs:IDREF, xs:IDREFS, xs:language, xs:Name, xs:NCName, xs:NMTOKEN, xs:NMTOKENS, xs:NOTATION.

### Table 3-6 Simple XQuery to Java Type Conversion

<table>
<thead>
<tr>
<th>XQuery Type</th>
<th>Mediator Accepts</th>
<th>Mediator Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs:boolean</td>
<td>Boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>xs:byte</td>
<td>Byte</td>
<td>Byte</td>
</tr>
<tr>
<td>xs:short</td>
<td>Short</td>
<td>Short</td>
</tr>
<tr>
<td>xs:int</td>
<td>Integer</td>
<td>Integer</td>
</tr>
<tr>
<td>xs</td>
<td>Java Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>xs:long</td>
<td>Long</td>
<td>Long</td>
</tr>
<tr>
<td>xs:integer</td>
<td>BigInteger</td>
<td>BigInteger</td>
</tr>
<tr>
<td>xs:negativeInteger</td>
<td>BigInteger</td>
<td>BigInteger</td>
</tr>
<tr>
<td>xs:positiveInteger</td>
<td>BigInteger</td>
<td>BigInteger</td>
</tr>
<tr>
<td>xs:nonNegativeInteger</td>
<td>BigInteger</td>
<td>BigInteger</td>
</tr>
<tr>
<td>xs:nonPositiveInteger</td>
<td>BigInteger</td>
<td>BigInteger</td>
</tr>
<tr>
<td>xs:unsignedByte</td>
<td>Short</td>
<td>Short</td>
</tr>
<tr>
<td>xs:unsignedShort</td>
<td>Integer</td>
<td>Integer</td>
</tr>
<tr>
<td>xs:unsignedInt</td>
<td>Long</td>
<td>Long</td>
</tr>
<tr>
<td>xs:unsignedLong</td>
<td>BigInteger</td>
<td>BigInteger</td>
</tr>
<tr>
<td>xs:float</td>
<td>Float</td>
<td>Float</td>
</tr>
<tr>
<td>xs:double</td>
<td>Double</td>
<td>Double</td>
</tr>
<tr>
<td>xs:decimal</td>
<td>BigDecimal</td>
<td>BigDecimal</td>
</tr>
<tr>
<td>xs:string</td>
<td>String</td>
<td>String</td>
</tr>
<tr>
<td>xs:anyURI</td>
<td>String, java.net.URI</td>
<td>String</td>
</tr>
<tr>
<td>xs:base64Binary</td>
<td>byte[]</td>
<td>byte[]</td>
</tr>
<tr>
<td>xs:hexBinary</td>
<td>byte[]</td>
<td>byte[]</td>
</tr>
<tr>
<td>xs:QName</td>
<td>javax.xml.namespace.QName</td>
<td>String</td>
</tr>
</tbody>
</table>

This string is formed by concatenating a URI, a # symbol, and the local name of the QName. Input can be a string of that form or a QName object.
Conversion of Date/Time Types

The mediator APIs handle date/time conversions in a manner that is consistent with the SDO specification. In the SDO for Java Specification V2.1, all date/time values are mapped to Java Strings. You can find this specification online at: www.osoa.org/display/Main/Service+Data+Objects+Specifications.

Note: The form of these Strings is the same as the canonical lexical representation of the corresponding schema type according to the XML Schema specification.

Table 3-7 XQuery Date/Time Types to Java Conversions

<table>
<thead>
<tr>
<th>XQuery Type</th>
<th>Mediator Accepts</th>
<th>Mediator Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs:date</td>
<td>String, java.sql.Date</td>
<td>String</td>
</tr>
<tr>
<td>xs:time</td>
<td>String, java.sql.Time</td>
<td>String</td>
</tr>
<tr>
<td>xs:dateTime</td>
<td>String, java.sql.Timestamp,</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>java.util.Date, java.util.Calendar</td>
<td></td>
</tr>
<tr>
<td>xs:duration, xs:gDay,</td>
<td>String</td>
<td>String</td>
</tr>
<tr>
<td>xs:gMonth, xs:gMonthDay,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xs:gYear, xs:gYearMonth</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Passing Empty Sequence Arguments

If a data service operation takes an optional argument (for example CUSTOMER?), you can pass a NULL parameter to a Static or Dynamic Mediator method. You can pass NULL in these situations:

- As an entry in the Object[] of arguments passed to methods such as DataAccessService.invoke() or as one of the arguments to the Static Mediator method.
- As a value to PreparedExpression.bindObject(QName variable, Object value).
### Quantified Return Types

In the Static Mediator API, quantified return types from data service operations are generated based on the following rules:

Any data service parameters that are quantified with `*` or `+` (for example, `xs:int*`) have static mediator methods that are declared to return `DASResult<type>`.

For example, a data service operation with the following signature:

```xml
declare function t1:someFunc($a as xs:int*, $b as xs:double+) as xs:int*
external;
```

is converted to a method like this:

```java
DataObject<java.lang.Integer> someFunc(Integer[] a, Double[] b);
```

Data service operations that return unquantified or `?`-quantified types have Static Mediator methods that are declared to return the type directly. In the case of `?`, it is possible that the result of the operation will be 0 instances of the type, in which case the static mediator method returns `NULL`.

### What is Autoboxing?

Autoboxing is a Java 1.5 language feature that can help simplify your code when working with Java primitive types and their object wrappers. Autoboxing automatically casts between object wrappers such as `Integer` and their primitive countertypes. With autoboxing, for instance, you can use an `Integer` object returned from a Mediator API method in a mathematical expression and you can pass an `int` to a Mediator API method that takes an `Integer` object. For detailed information on autoboxing, refer to the Sun’s Java documentation.

In Listing 3-10, an `Integer` object is retrieved from a `DASResult` is auto-cast to an `int`.

**Note:** If you use an `Integer` returned from a static mediator method as an `int`, but the static mediator method actually returns `null`, you will get a `NullPointerException`. This can only occur from a data service operation that is declared to return `xs:int?` – that is, 0 or 1 integers. (This is not unique to `int`, but to any use of autoboxing.)
Listing 3-10  Autoboxing Example

CustomerDAS custdas = CustomerDAS.getInstance(..);
// Invoke a 0-argument procedure that returns xs:int*
DASResult<Integer> result = custdas.getIDs();
while (result.hasNext()) {
    int cust = result.next(); // Note use of autoboxing.
}
result.dispose();

Web Services Support

The ALDSP native web services feature lets you map data services to web services directly. Client applications access data through web services using the Mediator API. Both the Dynamic and Static Mediator APIs support native web services. See Chapter 4, “Invoking Data Services Through Web Services” for detailed information on the native web services feature.

Advanced Topics

This section includes these topics:

- Schema Management
- Support for Stateless Operations
- Cache Management
- Specifying XPath Expressions as Arguments
- Making Ad Hoc Queries
Schema Management

SDO provides a series of APIs that assist with schema management. These APIs include:

- XMLHelper – Creates DataObjects from XML.
- DataFactory – Creates DataObjects from scratch.
- XSDHelper – Loads the SDO type system with schemas.
- HelperContext – Obtains instances of all the various helpers.

Schema Scope

A HelperContext object represents SDO’s concept of scope; all schemas loaded into a particular XSDHelper are available and used when creating DataObjects from the XMLHelper or DataFactory of the same HelperContext.

A Dataspace represents the basic unit of scope for schemas. A Dataspace will not contain any schemas with conflicting type declarations.

For the web services-based mediator, the scope is defined by the WSDL. All schemas necessary for all operations in a WSDL are included in that WSDL, so the WSDL itself forms a reasonable scope for schemas. See Chapter 4, “Invoking Data Services Through Web Services” for more information.

The Mediator automatically keeps a global cache of HelperContexts, and the key to that cache will be either the Dataspace name or, for the web services case, the WSDL URL. The Mediator will automatically use the HelperContext for the appropriate Dataspace/WSDL when creating new DataObjects for the return values of operations. You can obtain the HelperContext for a given Dataspace/WSDL and use this HelperContext to create your own DataObjects, query the type system, and so on. See “Schema Cache Management” on page 3-53 for more information on the HelperContext API.
Schema Download

This section describes the process of downloading schemas for DataAccessService objects instantiated from data services and WSDLs (web services).

**Note:** Schemas are only downloaded when creating a dynamic DataAccessService. Creating an instance of the Static Mediator API never downloads schemas, because the schemas are already compiled into the Static Mediator Client JAR file.

- **Data Services Case** – When a DataAccessService is instantiated for a data service, a HelperContext object is automatically populated with all schemas imported by the data service (and all schemas imported or included by those schemas, recursively).

- **WSDL Case** – When a DataAccessService is instantiated for a WSDL, the mediator automatically populates the HelperContext for that WSDL with all schemas in that WSDL. Schemas are only loaded if they have not previously been loaded. You can set an optional boolean flag on the newDataAccessService() method that requests the mediator not to download schemas. Use this flag if you intend to download schemas manually using methods described in the next section, or if you plan to load them manually using SDO XSDHelper methods.

**Tip:** The schema download feature ensures that you do not need to worry about schemas. The default behavior ensures that schemas are available to clients at the appropriate times.

Schema Cache Management

Use the following methods for querying and manipulating the mediator cache of HelperContexts. These are static methods on the class com.bea.dsp.das.HelperContextCache.

- **HelperContext get(String key)** – Returns (or creates, if necessary) the HelperContext for the given Dataspace name/WSDL URL. You may wish to obtain this object to load schemas that are only available on the client. For example, this technique is useful if you have a schema for the return type of an ad-hoc query that you intend to execute.

- **void flush(String key)** – Instructs the mediator to remove the HelperContext for the given Dataspace name/WSDL URL from its cache All DataAccessServices created in the future will use a new HelperContext. You can use this technique if you know the state of schemas has changed on the ALDSP server.

**Note:** It is not possible in SDO 2.1 to “unload” schemas from a HelperContext; therefore the only way to change schema information is to create an entirely new HelperContext.
Invoking Data Services from Java Clients

- boolean loadSchemasForDataspace(Context ctx, String dataspace, String dsname) –
  Instructs the Mediator to download all appropriate schemas for the given Dataspace and
data service. This download mechanism is similar to the one used by the Mediator when
creating a DataAccessService instance. This method allows you to “pre-load” schemas, for
instance, so you do not have to create a DataAccessService instance just to obtain the
schemas to create DataObjects. (See also “Creating New DataObjects” on page 3-28.) This
method is also the only way to achieve server schema download when you intend to use
PreparedExpression for executing an ad-hoc query. Finally, if you pass NULL for the data
service name, this method automatically downloads schemas for all data services in the
Dataspace. This may be slow, but is useful for certain ad-hoc query circumstances.

- boolean loadSchemasForWSDL(String wsdl) – Instructs the mediator to load all schemas
  from the WSDL into the corresponding HelperContext. This download mechanism is
  similar to the one used by the Mediator when creating a DataAccessService instance. As
  ad-hoc queries are not feasible over the web service transport, this method is typically not
  used.

Note: Both loadSchemasForDataspace() and loadSchemasForWSDL() return a boolean
indicating whether they actually loaded any schemas; they return false if the schemas for
the requested data service or WSDL were previously loaded.

Support for Stateless Operations

By default, the Mediator API holds resources on the ALDSP server open while data is being
returned. As discussed in “Understanding DASResult” on page 3-41, data objects are returned
through DASResult one object at a time. ALDSP refers to this strategy as stateful.

Generally, stateful operations are desirable. Stateful behavior allows both the client and the server
to minimize memory consumption. The ALDSP server will only hold open resources as long as
is absolutely necessary, and the client will not use more network round trips than are necessary
to transfer data to the server.

However, in some cases, you may want to guarantee that the client uses exactly one network
round trip. For instance, if your network connection to the ALDSP server is highly latent or
potentially unreliable, using one round trip minimizes response time and ensures that there is no
possibility of resources being left open on the server longer than necessary.

To override the default stateful behavior and return results immediately, use the RequestConfig
flag FETCH_ALL_IMMEDIATELY. When this flag is specified, the Mediator uses exactly one
network round trip to retrieve all the results at once. In addition, server resources are closed
immediately.
**Note:** The client must have enough memory to materialize the entire result set immediately. In addition, the ALDSP server will need to fully materialize the result set in memory when this flag is specified. Therefore, it is very important to use this flag when there is any possibility that the result set cannot be held comfortably in memory on both the client and the server.

**Note:** WebLogic Server specifies a maximum amount of data which can be sent in a single network operation. By default, this amount is 10 MB. If you use FETCH_ALL_IMMEDIATELY and the results are larger than this block size, you may receive a weblogic.socket(MaxMessageSizeExceeded)Exception. You can change this 10 MB limit in the WebLogic Server Console by selecting:

Environment > Servers > (server) > Protocols > General > Maximum Message Size

When this flag is enabled, all methods that return a DASResult will return one which is already disposed, as explained in “Disposing the Result Object” on page 3-22.

If you use FETCH_ALL_IMMEDIATELY, you can still use the normal iterator methods of DASResult, or use getItems() to read all the results at once.

**Cache Management**

This section discusses API features that let you manage data caching through the Mediator APIs.

**Forcing Data Cache Read-through and Update**

Data retrieved by data service operations can be cached for quick access. This is known as a data caching. (See “Configuring the Query Results Cache”, in the ALDSP Administration Guide for details.) Assuming the data changes infrequently, it’s likely that you’ll want to use the cache capability.

When the RequestConfig.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

You can control the data cache using the RequestConfig.REFRESH_CACHE_EARLY attribute. If the RequestConfig.GET_CURRENT_DATA property is not enabled, you can use the RequestConfig.REFRESH_CACHE_EARLY property 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 RequestConfig.setIntAttribute() 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.

Specifying XPath Expressions as Arguments

ALDSP supports a limited subset of XPath expressions called SDO path expressions. SDO path expressions offer flexibility in how you locate data objects and attributes in the dynamic data API’s accessors.

SDO path uses only SDO property names (which can be different from the element/attribute name from schema/xml) in the selector. If there are alias names assigned, those are also used to match. Each step of the path before the last must return a single DataObject.

For example:

```java
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 multiple values, all elements are returned.

The get() method returns an Object. If the result of the expression is a property that isMany, the method returns a List of DataObjects.

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

```java
myCustomer.get("..")
```
You can get the root containing the data object by using the get() method with XPath notation:

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

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

ALDSP fully supports both the traditional index notation and the augmented notation. See “XPath Expressions in the Dynamic Data Object API” on page 2-9 for details.

## Making Ad Hoc Queries

The `DataAccessServiceFactory.prepareExpression()` method lets you create ad hoc queries against the data service. Listing 3-11 shows an example of the `prepareExpression()` method. For more information on ad hoc queries, see “Using Ad Hoc Queries to Fine-tune Results from the Client” on page 9-26.

### Listing 3-11  Example of preparedExpression Method

```java
PreparedExpression pe = DataAccessServiceFactory.prepareExpression
    (ctx, appname, "18 + 25");
DASResult<Object> result = pe.executeQuery();
int answer = (int) result.next();
result.dispose();
```

## Understanding Transaction Behavior

This section discusses the transaction behavior of read/write and read-only operations and queries.

### Transaction Behavior for Read/Write Operations

The ALDSP server always creates a new transaction if necessary when executing read/write operations, such as create, update, or delete.
Transaction Behavior for Read-Only Operations

By default, read operations make use of a transaction if one is currently active on the client; however, if no transaction is open, one will not be created. You can change this default by setting an attribute on RequestConfig and passing RequestConfig as a parameter to invoke(). (For detailed information on invoke() see the Javadoc.) The attribute RequestConfig.ReadTransactionMode lets you set one of the following values to configure transaction behavior of read-only operations.

- **SUPPORTS** – Read operations will make use of a transaction if one is active on the client. If no transaction is open, however, one will not be created. This is the default setting.
- **REQUIRED** – Read operations will make use of a transaction if one is active on the client. If no transaction is open on the client, one will be created on the server for the duration of the read operation.
- **NOT_SUPPORTED** – Read operations will not use a transaction if one is active on the client, and no transaction will be created.

Use the RequestConfig.setEnumAttribute() method to set the ReadTransactionMode attribute. For example, the following code sets the ReadTransactionMode mode to SUPPORTS.

```java
RequestConfig config = new RequestConfig();
config.setEnumAttribute(RequestConfig.ReadTransactionMode.SUPPORTS);
```
Invoking Data Services Through Web Services

This chapter explains how to expose data services as industry-standard Web services and how to create client applications that invoke data services through those web services.

This chapter includes these topics:

- Overview
- Before You Begin
- Getting Started
- Sample Static Mediator Application
- Sample Dynamic Mediator Application
- Transaction Behavior and Web Services
- Securing Your Web Services Application

Overview

The ALDSP Native Web Services feature lets you map data services to web services directly. Client applications access data through web services with the Data Services Mediator API.

Tip: Chapter 3, “Invoking Data Services from Java Clients,” discusses the Mediator API in detail. We recommend that you review that chapter before you develop web service-enabled applications.
When you expose data services as Web services, your information assets become accessible to a wide variety of client types, including other Java Web service clients, Microsoft ADO.NET, 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. Web service WSDL operations map directly to data service operations on the server.

**Tip:** For detailed information on creating data service operations (read, create, update, delete, libraryFunction, and libraryProcedure) see the *Data Services Developer’s Guide*.

---

**Figure 4-1 Web Services Enable Access to ALDSP-Enabled Applications from a Variety of Clients**
Before You Begin

This chapter is intended for Java developers who wish to write client applications that invoke data services through web services.

We recommend that:

- You have a basic understanding of web service technology and terms such as WSDL (Web Service Description Language) and SOAP (Simple Object Access Protocol).

  **Tip:** If you are unfamiliar with web services, you can refer to the WebLogic Server document “WebLogic Web Services: Getting Started” on e-docs. This document provides a thorough introduction to web services as well as detailed information on developing web services for WebLogic Server.

- You are a Java programmer (the Data Service Mediator APIs you will use are Java APIs).

- You review Chapter 3, “Invoking Data Services from Java Clients.” This chapter discusses the Data Services Mediator API and related topics in detail.

- You review the basics of the SDO (Service Data Object) standard. SDO provides APIs for manipulating data objects and is central to the client programming model adopted by ALDSP. See Chapter 2, “Data Programming Model and Update Framework,” for information on SDO.

- You are familiar with XML.

Getting Started

This section lists the basic steps to get started writing a Java client application that interacts with a data service.

- **Basic Steps**

- **Setting the CLASSPATH**

- **Running the Sample Applications**
Basic Steps

These are the basic steps to follow when developing a Java client that invokes data service functions through web service operations.

1. The first thing you need is a data service to call. Someone (typically a data service developer) creates the data service.

2. A Web Service Map file must then be generated from the data service. The map file is typically generated by a data service developer. The procedure for generating a Web Service Map file, see the Data Services Developer’s Guide.

3. Deploy and test the web service.

4. Decide whether to use the Static or Dynamic Mediator API to interact with the web service from your Java client. See “Dynamic and Static Mediator APIs” on page 3-3 for a summary of each API. To use the Static Mediator API, you need to generate or obtain the Web Services Mediator Client JAR file. For instructions on generating a Web Services Mediator Client JAR, see the Data Services Developer’s Guide.

   **Tip:** The Static Mediator API is generally recommended for most use cases. The Static Mediator API is type safe and generally easier to use than the Dynamic Mediator API.

5. Set up your Java build environment. You need certain JAR files in your CLASSPATH. See “Setting the CLASSPATH” on page 4-5 for details.

6. Learn the WSDL operations that are available to you for accessing data service functions. The operations have the same names and parameters as the data service from which they were generated.

7. Write and test your client application. See the sample applications provided in this document: “Sample Static Mediator Application” on page 4-6 and “Sample Dynamic Mediator Application” on page 4-15.
**Setting the CLASSPATH**

CLASSPATH settings depend on whether you are using the Static or Dynamic Mediator API.

**Static Java Mediator API Client CLASSPATH**

The following JARs must be in the CLASSPATH of your Java application if you are using the Static Mediator API.

- `<Web Services Mediator Client File>.jar`
  
  **Note:** This first entry is only required if you are using the Static Mediator API. This entry must precede the other entries listed below. For instructions on generating a Web Services Mediator Client JAR, see the *Data Services Developer’s Guide*. Typically, this file is generated by a data services developer.

- `<ALDSP_HOME>/lib/sdo.jar`
- `<WEBLOGIC_HOME>/common/lib/apache_xbean.jar`
- `<WEBLOGIC_HOME>/server/lib/weblogic.jar`

**Dynamic Java Mediator API Client CLASSPATH**

The following JARs must be in the CLASSPATH of your Java application if you are using the Dynamic Mediator API.

- `<ALDSP_HOME>/lib/sdo.jar`
- `<ALDSP_HOME>/lib/ld-client.jar`
- `<WEBLOGIC_HOME>/common/lib/apache_xbean.jar`
- `<WEBLOGIC_HOME>/server/lib/weblogic.jar`

**Running the Sample Applications**

A good way to get started is to run the sample application code that is provided in this chapter. Samples that use the Static and the Dynamic Mediator APIs are included. The samples illustrate simple but common use cases: retrieving data, modifying it, and updating it. See “Sample Static Mediator Application” on page 4-6 and “Sample Dynamic Mediator Application” on page 4-15.
Sample Static Mediator Application

This section presents a simple Java program that you can copy, compile, and run. The program uses the Static Mediator API to invoke WSDL operations to perform the following basic tasks: authenticating the client, retrieving data, modifying data, and updating data on the server. For an overview of the Static Mediator API, see “Dynamic and Static Mediator APIs” on page 3-3.

Topics include:

- Setting Up the Sample Data Service
- Creating a Web Service Map File
- Generating the Web Services Mediator Client JAR File
- Setting Up the Java Project
- Running and Testing the Code
- Examining the Sample Code

Setting Up the Sample Data Service

The sample application presented here is designed to work with a sample data service. You need to create this data service and configure a sever before continuing. For detailed instructions on creating the data service that is required by this sample application, see “Setting Up the Sample Data Service” on page 3-10.

Creating a Web Service Map File

To run the example, you need to generate a Web Service Map file. You can do this easily using Data Services Studio. See the Data Services Developer’s Guide for detailed instructions.

To create the file using Data Services Studio:

1. Right-click the data service project and select New > Web Service Map.
2. Follow the wizard to create the map file. For this example, name the file PhysicalCUSTOMER.ws.
3. Drag the data service file, PhysicalCUSTOMER.ds onto the Web Service Mapper editor.
4. Save the file.
5. Deploy and test the web service. To do this, right-click the PhysicalCUSTOMER.ds file and select Test Web Service.

**Tip:** You can generate a Web Services Map file using Data Services Studio, the ALDSP Console, or an Ant script. These methods are described in detail in the *Data Services Developer’s Guide*.

### Generating the Web Services Mediator Client JAR File

The sample Java application listed later in this section requires that you first generate a Web Services Mediator Client JAR file. The classes in this JAR contain type-safe methods that call WSDL operations.

**Tip:** You can generate a Web Services Mediator Client JAR file using Data Services Studio, the ALDSP Console, or an Ant script. These methods are described in detail in the *Data Services Developer’s Guide*. For this example, we will use Data Services Studio.

To generate a Web Services Mediator Client JAR file using Data Services Studio:

1. Generate the Web Service Map file, as discussed previously in “Creating a Web Service Map File” on page 4-6.
2. Select File > Export.
3. In the Select dialog, select AquaLogic Data Services Platform > Web Services Mediator Client JAR File and click Next.
4. Complete the Web Services Mediator Client JAR File dialog as follows:
   - In the left panel, select the Dataspace project that contains the .ws file(s) to export. For this example, the Dataspace project is called MediatorTest.
   - In the right panel, select the Web Service Map file to export. You can select one or more .ws files. For this example, be sure PhysicalCUSTOMER.ws is selected.
   - Specify a directory in which to place the exported JAR file. You can select any location on your system. By default, the exported JAR will be named: MediatorTest-ws-client.jar.
5. Click Finish.

6. After you generate the JAR file, you must place it in the CLASSPATH for your Java build environment. See “Setting the CLASSPATH” on page 4-5 for more information.

**Tip:** Generated classes in the JAR file are named according to the conventions described in “Naming Conventions for Generated Classes” on page 3-41.

### Setting Up the Java Project

**Listing 4-1** lists the sample Java client that uses the Static Mediator API to call WSDL operations. The application simply retrieves a DataObject from a data store, modifies the object, and returns it to the data store. This example assumes you are using Data Services Studio, but you can use the IDE or build environment of your choice. For this example, we set up a Java project called MediatorWSClient.

To run the sample:

1. Create a Java project called MediatorWSClient.

2. Set up your Java Build Path to include the JAR files listed in “Setting the CLASSPATH” on page 4-5. To do this, select Project > Properties > Java Build Path. Be sure to add the generated Web Services Mediator Client JAR file.

3. Create a package called com.bea.ws.sample in your Java project. To do this, right-click the Java project in the Package Explorer and select New > Package.

4. Create a Java class called StaticWSSampleApp.java in the package. To do this, right-click the package in the Package Explorer and select New > Class.

5. Delete the default contents of the new source file and copy the entire file listed in Listing 4-1 into the source file.

6. Save the file. Figure 4-2 shows the completed project configuration.
Figure 4-2 Completed Project Configuration

Note: The imported classes PhysicalCUSTOMERDAS and PhysicalCUSTOMER are taken from the Web Services Mediator Client JAR file, which must be in the CLASSPATH.

Listing 4-1 StaticWSSampleApp.java

    package com.bea.ws.sample;

    import das.ws.ld.PhysicalCUSTOMERDAS;
    import physicaldss.physicalcustomer.PhysicalCUSTOMER;

    import com.bea.dsp.das.DASResult;
    import com.bea.dsp.sdo.SDOUtil;

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

    public class StaticWSSampleApp {
        public static void main(String[] args) throws Exception {
            // Code goes here
        }
    }
// Create InitialContext for mediator
Hashtable<String, String> hash = new Hashtable<String, String>();
hash.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContextFactory");
hash.put(Context.SECURITY_PRINCIPAL, "weblogic");
hash.put(Context.SECURITY_CREDENTIALS, "weblogic");
Context ctx = new InitialContext(hash);

String wsdlURL = "http://localhost:7001/MediatorTest/PhysicalCUSTOMER.ws?WSDL";

// Create DataAccessService handle with Context and dataspace name
PhysicalCUSTOMERDAS das = PhysicalCUSTOMERDAS.getInstance(ctx, wsdlURL);

// Invoke the basic 'get all customers' function
DASResult<PhysicalCUSTOMER> result = das.PhysicalCUSTOMER();

// Obtain the first PhysicalCUSTOMER DataObject
PhysicalCUSTOMER customer = result.next();

// When finished iterating through results, always call dispose().
result.dispose();

// Enable change-tracking for that PhysicalCUSTOMER
SDOUtil.enableChanges(customer);

// Modify customer
customer.setFIRST_NAME("StaticWSMediator");
customer.setEMAIL_ADDRESS("staticwsmediator@example.com");

// Send changes back to DSP - update function takes an array
// of PhysicalCUSTOMERS
das.updatePhysicalCUSTOMER(new PhysicalCUSTOMER[] { customer });
Running and Testing the Code

To test the application, simply start the server and run the Java client as a Java application. In Data Services Studio, this is commonly done by right-clicking the Java file and selecting Run As > Java Application.

To verify that the Java client worked, re-test the data service:

1. Open the data service in the Data Service editor.
2. Click the Test tab (see Figure 4-3).
3. Select an operation from the drop down menu. For this example, select the PhysicalCUSTOMER() function.
4. Click Run (see Figure 4-3).
5. Inspect the first row of the data table. The client application changes the first customer’s name and email address to “StaticWSMediator” and “staticwsmediator@example.com” as shown in Figure 4-3.

Figure 4-3 Testing the Client
Examining the Sample Code

This section discusses the parts of the Java sample in Listing 4-1. This section examines the following components of the sample code:

- Importing Packages
- Obtaining a Data Access Service Handle
- Retrieving Data from the Service
- Obtaining a DataObject from the Result
- Modifying the DataObject
- Returning Changes to the Server
- Disposing the Result Object

Importing Packages

Note that the first two imported classes come from the generated Web Services Mediator Client JAR file.

The PhysicalCUSTOMERDAS class is the generated DataAccessService class that contains Java methods to call the WSDL operations. Method names are the same as their corresponding WSDL operations. This class contains type-safe methods that map to the actual WSDL operations. The PhysicalCUSTOMER class provides the SDO interface for manipulating DataObjects returned from the data service.

```java
import das.ws.PhysicalCUSTOMERDAS;
import physicaldss.physicalcustomer.PhysicalCUSTOMER;
import com.bea.dsp.das.DASResult;
import com.bea.dsp.sdo.SDOUtil;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
```
Obtaining a Data Access Service Handle

A DataAccessService object lets you call methods on a data service. See the Javadoc for more information on this class. For the Static Mediator API, DataAccessService (DAS) classes have a factory method named getInstance() to return the handle.

The getInstance() method requires two parameters to return the handle:

- A WebLogic JNDI Context object. The Context object holds properties, such as certain security attributes. See “Obtaining the WebLogic JNDI Context for ALDSP” on page 3-44. For more information on WebLogic JNDI context objects, see Programming WebLogic JNDI on e-docs. For information on security settings, see “Securing Your Web Services Application” on page 4-21.

- A WSDL URL that specifies the address of the web service to access.

```java
Hashtable<String, String> hash = new Hashtable<String, String>(){
    hash.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContextFactory");
    hash.put(Context.SECURITY_PRINCIPAL, "weblogic");
    hash.put(Context.SECURITY_CREDENTIALS, "weblogic");
    Context ctx = new InitialContext(hash);

    String wsdlURL =
        "http://localhost:7001/MediatorTest/PhysicalCUSTOMER.ws?WSDL";

    // Create DataAccessService handle with Context and dataspace name
    PhysicalCUSTOMERDAS das = PhysicalCUSTOMERDAS.getInstance(ctx, wsdlURL);
```

**Note:** Both the Static and Dynamic Mediator APIs accept either a file or a WSDL URL. For example:

```java
file:///C:/RTLApp/DataServices/RTLServices/Customer.wsdl
```

where Customer.wsdl is the WSDL file located on the local hard drive.

Retrieving Data from the Service

The generated DataAccessService method PhysicalCUSTOMER() retrieves the result set from the data service. This method returns all customer objects from the data service. The return type is a DASResult object, which works like an iterator. For more information on this return type, see “Understanding DASResult” on page 3-41.

```java
DASResult<PhysicalCUSTOMER> result = das.PhysicalCUSTOMER();
```
Obtaining a DataObject from the Result

The DASResult.next() method works very much like the Java method Iterator.next(). It returns the next PhysicalCUSTOMER, which is an SDO DataObject. SDO is a Java-based data programming model (API) and architecture for accessing and updating data. For details on SDO, see Using Service Data Objects (SDO) in the ALDSP 2.5 Concepts Guide.

```java
PhysicalCUSTOMER customer = result.next();
```

Disposing the Result Object

You must call DASResult.dispose() whenever you are finished iterating through a result object. For more information on dispose(), see “Disposing of DASResult Objects” on page 3-42.

```java
result.dispose();
```

Modifying the DataObject

After you obtain a DataObject, you can modify it; however, if you intend to submit these changes back to the ALDSP server, you must enable change-tracking on the DataObject before making any modifications. The SDOUtil.enableChanges() method lets you enable change-tracking for a single DataObject or an array of DataObjects. For more information on this method, see “Working with Data Objects” on page 3-45. After the customer object has change-tracking enabled, the generated setters are called to modify certain values in the customer object.

```java
SDOUtil.enableChanges(customer);

customer.setFIRST_NAME("StaticWSMediator");
customer.setEMAIL_ADDRESS("staticwsmediator@example.com");
```

Returning Changes to the Server

Finally, the generated DataAccessService.updatePhysicalCUSTOMER() method is called with a single parameter: an array of PhysicalCUSTOMER objects. The method calls its equivalent data service function to update the database with the newly modified row of data.

```java
das.updatePhysicalCUSTOMER(new PhysicalCUSTOMER[] { customer });
```
Sample Dynamic Mediator Application

This section presents a simple example that you can copy, compile, and run. This example uses the Dynamic Mediator API to perform these basic tasks: authenticating the client, retrieving data, modifying data, and updating data on the server.

- Setting Up and Running the Sample Code
- Sample Java Client Code (Dynamic Mediator API)
- Examining the Sample Code

Setting Up and Running the Sample Code

To run this sample code, follow the basic setup instructions in “Sample Static Mediator Application” on page 4-6. The procedures for creating a sample data service, setting the CLASSPATH, and running the program are the same as the Static Mediator sample; however, when using the Dynamic Mediator API, you do not need to generate or reference the Web Services Mediator Client JAR file.

Sample Java Client Code (Dynamic Mediator API)

Listing 4-2  DynamicWSSampleApp.java

```java
package com.bea.ws.sample;

import com.bea.dsp.das.DataAccessServiceFactory;
import com.bea.dsp.das.DataAccessService;
import com.bea.dsp.das.DASResult;
import com.bea.dsp.sdo.SDOUtil;
import commonj.sdo.DataObject;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;

public class DynamicWSSampleApp {
    public static void main(String[] args) throws Exception {
        // Create InitialContext for mediator
```
Hashtable<String, String> hash = new Hashtable<String, String>();
hash.put(Context.INITIAL_CONTEXT_FACTORY,
        "weblogic.jndi.WLInitialContextFactory");
hash.put(Context.SECURITY_PRINCIPAL, "weblogic");
hash.put(Context.SECURITY_CREDENTIALS, "weblogic");
Context ctx = new InitialContext(hash);

String wsdlURL =
"http://localhost:7001/MediatorTest/PhysicalCUSTOMER.ws?WSDL";

// Create DataAccessService handle with Context, dataspace
// name, and data service URI
DataAccessService das = DataAccessServiceFactory.newDataAccessService
    (ctx, wsdlURL);

// Invoke the basic 'get all customers' function, which takes
// no arguments
DASResult<Object> result = das.invoke("PhysicalCUSTOMER", new Object[0]);

// Obtain the first PhysicalCUSTOMER DataObject
DataObject customer = (DataObject) result.next();

// When finished iterating through results, always call dispose().
result.dispose();

// Enable change-tracking for that PhysicalCUSTOMER
SDOUtil.enableChanges(customer);

// Modify customer
customer.set("FIRST_NAME", "DynamicWSMediator");
customer.set("EMAIL_ADDRESS", "dynamicwsmediator@bea.com");

das.invoke("updatePhysicalCUSTOMER", new Object[] { customer });
result.dispose();

}
Examining the Sample Code

This section discusses the parts of the Java sample in Listing 4-2. This section examines the following components of the sample code:

- Importing Classes
- Obtaining a DataAccessService Handle
- Retrieving Data from the Service
- Obtaining a DataObject from the Result
- Disposing the Result Object
- Modifying the DataObject
- Returning Changes to the Server

Importing Classes

These classes are required by the sample. For detailed information on the classes, refer to the Javadoc on e-docs.

```java
import com.bea.dsp.das.DataAccessServiceFactory;
import com.bea.dsp.das.DataAccessService;
import com.bea.dsp.das.DASResult;
import com.bea.dsp.sdo.SDOUtil;
import commonj.sdo.DataObject;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
```
Obtaining a DataAccessService Handle

A DataAccessService object lets you call methods on and submit changes to a data service. The DataAccessServiceFactory requires two parameters to return the handle.

- A WebLogic JNDI Context object. The Context object holds properties, such as certain security attributes. See “Obtaining the WebLogic JNDI Context for ALDSP” on page 3-44. For more information on WebLogic JNDI context objects, see Programming WebLogic JNDI on e-docs. For more information on security, see “Securing Your Web Services Application” on page 4-21.

- A WSDL URL that specifies the address of the web service to access.

```java
Hashtable<String, String> hash = new Hashtable<String, String>();
hash.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContextFactory");
hash.put(Context.SECURITY_PRINCIPAL, "weblogic");
hash.put(Context.SECURITY_CREDENTIALS, "weblogic");
Context ctx = new InitialContext(hash);
String wsdlURL = "http://localhost:7001/MediatorTest/PhysicalCUSTOMER.ws?WSDL";

// Create DataAccessService handle with Context, dataspace
// name, and data service URI
DataAccessService das = DataAccessServiceFactory newDataAccessService
(ctx, wsdlURL);
```

**Note:** Both the Static and Dynamic Mediator APIs accept either a file or a WSDL URL. For example:

```
file:///C:/RTLApp/DataServices/RTLServices/Customer.wsdl
```

where Customer.wsdl is the WSDL file located on the local hard drive.

Retrieving Data from the Service

In this example, the invoke() method calls the WSDL operation PhysicalCUSTOMER. This method returns all customer objects from the data service. The method returns a DASResult object, which works like an iterator. For more information on this return type, see “Understanding DASResult” on page 3-41. Note that the PhysicalCUSTOMER operation takes no arguments. This signature corresponds to the data service function that the WSDL operation calls.

```
DASResult<Object> result = das.invoke("PhysicalCUSTOMER", new Object[0])
```
Obtaining a DataObject from the Result

The DASResult.next() method works very much like the Java method Iterator.next(). It returns the next object in the result set. Because the PhysicalCUSTOMER data service method returns SDO-compliant DataObjects, you can cast the return value to DataObject. SDO is a Java-based data programming model (API) and architecture for accessing and updating data. For details on SDO, see Using Service Data Objects (SDO) in the ALDSP Concepts Guide. See also “What is SDO?” on page 3-2.

```java
DataObject customer = (DataObject) result.next();
```

Disposing the Result Object

You must call DASResult.dispose() whenever you are finished iterating through a result object. For more information on dispose(), see “Disposing of DASResult Objects” on page 3-42.

```java
result.dispose();
```

Modifying the DataObject

After you obtain a DataObject, you can modify it; however, if you intend to submit these changes back to the ALDSP server, you must enable change-tracking on the DataObject before making any modifications. The SDOUtil.enableChanges() method lets you enable change-tracking for a single DataObject or an array of DataObjects. For more information on this method, see “Working with Data Objects” on page 3-45. After the customer object has change-tracking enabled, the set method is called to modify certain values in the customer object.

```java
// Enable change-tracking for that PhysicalCUSTOMER
SDOUtil.enableChanges(customer);
// Modify customer
customer.set("FIRST_NAME", "DynamicWSMediator");
customer.set("EMAIL_ADDRESS", "dynamicwsmediator@bea.com");
```

Returning Changes to the Server

Finally, the DataAccessService method invoke() is called with the update WSDL operation. The operation takes a single parameter: an array of PhysicalCUSTOMER objects. The data service function updates the database with the newly modified row of data.

```java
das.invoke("updatePhysicalCUSTOMER", new Object[] { customer });
```
Transaction Behavior and Web Services

Transactions are not propagated from the client to the server through web services, because there is no way for a client transaction to be sent through the web services interface. If a failure occurs and there is a transaction on the client side, the transaction will be rolled back, depending on how the client handles the failure; however, the transaction is not propagated to the server.

You can configure how transactions are handled on the server by setting attributes on the static com.bea.dsp.RequestConfig.ReadTransactionMode object:

- If set to REQUIRED and you invoke a read operation, a transaction is started on the server.
- If set to SUPPORTS (the default), a currently running transaction on the server will continue. If there is no currently running transaction, a new one is not created.

A third attribute, NOT_SUPPORTED, is not supported for web service operations. For a detailed discussion of transaction behavior with the Mediator API, see “Understanding Transaction Behavior” on page 3-57.

Listing 4-3 shows how to set the com.bea.dsp.RequestConfig.ReadTransactionMode attribute.

Listing 4-3 Setting the ReadTransactionMode Attribute

```java
RequestConfig config = new com.bea.dsp.RequestConfig();
RequestConfig requestConfig = request.getConfig();

if(readTransactionRequired) {
    config.setEnumAttribute(RequestConfig.ReadTransactionMode.REQUIRED);
}
```
Securing Your Web Services Application

ALDSP Native Web Services supports the following security features:

- Basic authentication (Web Application Security)
- Transport level security (HTTPS)
- Message level security (Web Services Security)

**Tip:** For detailed information on configuring these security options, see “Configure Security for Web Services Applications” in the *Data Services Developer’s Guide*.

Typically, security configuration is performed on the server side by an administrator or data services developer. As a client developer, you need to pass the required values to the server to satisfy the required authentication. See Listing 4-4 for one example.

**Note:** The Native Web Services feature supports only the HTTP and HTTPS transport protocols. If you wish to use another transport protocol, you must use ALDSP's AquaLogic Service Bus transport. This transport allows ALDSP data services to be exposed through ALSB.

If basic authentication is enabled, you must pass the following properties through the Mediator API using the context object. For more information on WebLogic JNDI context objects, see Programming WebLogic JNDI on e-docs.

- **DSPWebServicesProperties.USER_NAME**
- **DSPWebServicesProperties.PASSWORD**

Listing 4-4 illustrates one possible way to set up a web service security for a data access service. In this case, a client-side BinarySecurityToken credential provider is created that uses the public key infrastructure standard X.509 for identity. The credential provider is then set as a property in the Context object, which is used to create the data access service. The credential provider, security token, username token, and trust manager are standard web service security properties.

For more information refer to the WebLogic Service documentation on web services security. See also see “Configure Security for Web Services Applications” in the *Data Services Developer’s Guide*.
Listing 4-4  Example X.509 Certificate Token Profile Setup

Hashtable h = new Hashtable();
h.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContextFactory");

// Create empty list of credential providers.
List credProviders = new ArrayList();

// Create a client-side BinarySecurityToken credential provider that uses
// X.509 for identity, based on certificate and keys parameters.
CredentialProvider cp = new ClientBSTCredentialProvider(cert, key);
credProviders.add(cp);

String userid = "weblogic";
String password = "weblogic";
// Create a client-side UsernameToken credential provider based on username
// and password parameters.
cp = new ClientUNTCredentialProvider(userid.getBytes(), password.getBytes());
credProviders.add(cp);

h.put(WSSecurityContext.CREDENTIAL_PROVIDER_LIST, credProviders);
h.put(WSSecurityContext.TRUST_MANAGER, userTrustMgrImpl);

Context context = new InitialContext(h);
Using SQL to Access Data Services

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

- Introducing SQL Access to Data Services
- JDBC and SQL Support in Data Services Platform
- Preparing to Use SQL to Access Data Services
- Accessing Data Services Using SQL From a Java Application
- Advanced Features
- Accessing Data Services Using SQL-Based Applications
Introducing SQL Access to Data Services

Many reporting tools, such as Crystal Reports, Business Objects, Microsoft Access, and Microsoft Excel, can access data using SQL. SQL can also be useful in other contexts. Java applications, for example, can access data using SQL. You can also run ad hoc SQL queries using development tools such as Data Tools Platform (DTP) or SQL Explorer.

The Data Services Platform JDBC driver enables JDBC and ODBC clients to access information from data services using SQL. The Data Services Platform JDBC driver thereby increases the flexibility of the ALDSP integration layer by enabling access from a range of database applications and reporting tools.

For the client, the ALDSP integration layer appears as a relational database, with each data service operation comprising a table or a stored procedure. Internally, ALDSP translates SQL queries into XQuery. Figure 5-1 illustrates SQL access to data using the Data Services Platform JDBC driver.

As Figure 5-1 shows, source data can be consolidated, integrated, and transformed using Data Services Platform data services. The source data itself can come from disparate sources throughout the enterprise, including relational databases and Web services, among others.

You can then, in turn, expose the data service operations as a relational data source accessible using SQL queries. This enables JDBC clients to access data consolidated through Data Services Platform.

Note that the Data Services Platform JDBC driver does impose the following constraints on data services:

- You can use the Data Services Platform JDBC driver to access data only through data services that have a flat data shape, which means that the data service type cannot have nesting. SQL provides a traditional, two-dimensional approach to data access, as opposed to the multi-level, hierarchical approach defined by XML.

- The Data Services Platform JDBC driver exposes non-parameterized flat data service operations as tables because SQL tables do not have parameters. Parameterized flat data services are exposed as SQL stored procedures.
Figure 5-1 SQL Access to Data Services
Using SQL to Access Data Services

**Logical Data Services**

```xml
<empinfo empid=...>
  <name></name>
  <salary></salary>
  <options></options>
</empinfo>

<contact empid=...>
  <address>
    <street></street>
    <city></city>
    <state></state>
    <zip></zip>
  </address>
  <phone>
    <work></work>
    <home></home>
  </phone>
</contact>

GetBySalary(minSal)
```

**Physical Data Sources**

- **RDBMS Tables**
  - **EMPLINFO**
    - EMPID EMPNAME CURSALARY
  - **ADDRPHONE**
    - EMPLID STREET CITY STATE ZIP WORK HOME

Source data consolidated as data services

**SQL View**

- **Tables**
  - empinfo
    - empid name salary options
  - contact
    - street city state zip work home

**Published Data**

- **Stored Procedures**
  - GETBYSALARY(in minSal)

**Clients**

- Java JDBC Applications
- Reporting Applications
- Query Tools

**ALDSP JDBC Driver**

**JDBC Access**

**Data service operations published for SQL access**

stock:GetOptionsValue(empid)
Features of the Data Services Platform JDBC Driver

The Data Services Platform JDBC driver implements the java.sql.* interface in JDK 1.5x to provide access to a Data Services Platform server through the JDBC interface. The driver has the following features:

- Supports SQL-92 SELECT statements
- Implements the JDBC 3.0 Application Programming Interface (API)
- Supports Data Services Platform with JDK 1.5
- Supports both Java and ODBC bridge software clients
- Supports table parameters, an extension to SQL-92.
- Allows metadata access control at the JDBC driver level

Using the 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 access control ensures that users can view only those tables and procedures that they are authorized to access.

However, to use this feature, the 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.

Exploring Data Services Platform and JDBC Artifacts

The Data Services Platform views data retrieved from a database in the form of data sources and operations. Table 5-2 shows the equivalent terminology.

<table>
<thead>
<tr>
<th>Data Services Platform</th>
<th>JDBC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataspac Project</td>
<td>JDBC connection parameter (Driver URL)</td>
</tr>
<tr>
<td>Operation with parameters</td>
<td>Stored procedure</td>
</tr>
<tr>
<td>Operation without parameters</td>
<td>Table or stored procedure</td>
</tr>
</tbody>
</table>

For example, if you have a project SQLHowTo and a data service EmpInfo.ds with an operation getAll(), you can use SQL Mapper to expose it as JDBCdemo.empData.empinfo. The JDBC driver would then see a table called empinfo with schema empData and catalog JDBCdemo.
Using SQL to Access Data Services

**JDBC and SQL Support in Data Services Platform**

This section describes the JDBC and SQL support in the Data Services Platform JDBC driver.

**JDBC Support**

The Data Services Platform JDBC driver implements the following interfaces from the `java.sql` package as specified in JDK 1.5x:

- `java.sql.Blob`
- `java.sql.CallableStatement`
- `java.sql.Connection`
- `java.sql.DatabaseMetaData`
- `java.sql.ParameterMetaData`
- `java.sql.PreparedStatement`
- `java.sql.ResultSet`
- `java.sql.ResultSetMetaData`
- `java.sql.Statement`
- `java.sql.PreparedStatement`
- `java.sql.ResultSet`
- `java.sql.ResultSetMetaData`
- `java.sql.Statement`

The Data Services Platform JDBC driver supports the following methods:

<table>
<thead>
<tr>
<th>Interface</th>
<th>Supported Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java.sql.Blob</code></td>
<td>• getBinaryStream</td>
</tr>
<tr>
<td></td>
<td>• getBytes</td>
</tr>
<tr>
<td></td>
<td>• length</td>
</tr>
<tr>
<td></td>
<td>• position</td>
</tr>
<tr>
<td></td>
<td>• truncate</td>
</tr>
<tr>
<td><code>java.sql.CallableStatement</code></td>
<td>• clearParameters</td>
</tr>
<tr>
<td></td>
<td>• executeQuery</td>
</tr>
<tr>
<td></td>
<td>• setAsciiStream</td>
</tr>
<tr>
<td></td>
<td>• setBigDecimal</td>
</tr>
<tr>
<td></td>
<td>• setBoolean</td>
</tr>
<tr>
<td></td>
<td>• setByte</td>
</tr>
<tr>
<td></td>
<td>• setBytes</td>
</tr>
<tr>
<td></td>
<td>• setCharacterStream</td>
</tr>
<tr>
<td></td>
<td>• setDate</td>
</tr>
<tr>
<td></td>
<td>• setDouble</td>
</tr>
<tr>
<td></td>
<td>• setFloat</td>
</tr>
<tr>
<td></td>
<td>• setInt</td>
</tr>
<tr>
<td></td>
<td>• setLong</td>
</tr>
<tr>
<td></td>
<td>• setNull</td>
</tr>
<tr>
<td></td>
<td>• setObject</td>
</tr>
<tr>
<td></td>
<td>• setShort</td>
</tr>
<tr>
<td></td>
<td>• setString</td>
</tr>
<tr>
<td></td>
<td>• setTime</td>
</tr>
<tr>
<td></td>
<td>• setTimestamp</td>
</tr>
</tbody>
</table>
## Table 5-3 Data Services Platform JDBC Driver Methods

<table>
<thead>
<tr>
<th>Interface</th>
<th>Supported Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.sql.Connection</td>
<td>clearWarnings, close, createStatement, getAutoCommit, getCatalog, getHoldability, getConnection, getLogPrintWriter, getMetaData, getSchema, getTransaction, getTypeMap, getWarnings, isClosed, isReadOnly, nativeSQL, prepareCall, prepareStatement, setAutoCommit, setCatalog, setHoldability, setReadOnly, nativeSQL, prepareCall, prepareStatement, setAutoCommit, setCatalog, setHoldability, setReadOnly</td>
</tr>
<tr>
<td>java.sql.DatabaseMetaData</td>
<td>allProceduresAreCallable, allTablesAreSelectable, dataDefinitionCausesTransactionCommit, dataDefinitionIgnoredInTransactions, deletesAreDetected, doesMaxRowSizeIncludeBlobs, getAttributes, getBestRowIdentifier, getCatalogs, getCatalogSeparator, getCatalogTerm, getColumnPrivileges, getColumns, getConnection, getCrossReference, getDatabaseMajorVersion, getDatabaseMinorVersion, getDatabaseProductName, getDatabaseProductVersion, getDefaultTransactionIsolation, getDriverName, getDriverVersion, getExportedKeys, getExtraNameCharacters, getIdentifierQuoteString, getImportedKeys, getIndexInfo, getJDBCMajorVersion, getJDBCMinorVersion, getJDBCProductVersion, getJDBCProductName, getJDBCProductVersion, getJDBCMajorVersion, getJDBCMinorVersion, getJDBCMajorVersion, getJDBCMinorVersion</td>
</tr>
</tbody>
</table>
### Table 5-3 Data Services Platform JDBC Driver Methods

<table>
<thead>
<tr>
<th>Interface</th>
<th>Supported Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.sql.DatabaseMetaData</td>
<td>• getMaxBinaryLiteralLength&lt;br&gt;• getMaxCatalogNameLength&lt;br&gt;• getMaxCharLiteralLength&lt;br&gt;• getMaxColumnNameLength&lt;br&gt;• getMaxColumnsInGroupBy&lt;br&gt;• getMaxColumnsInIndex&lt;br&gt;• getMaxColumnsInOrderBy&lt;br&gt;• getMaxColumnsInSelect&lt;br&gt;• getMaxColumnsInTable&lt;br&gt;• getMaxConnections&lt;br&gt;• getMaxCursorNameLength&lt;br&gt;• getMaxIndexLength&lt;br&gt;• getMaxProcedureNameLength&lt;br&gt;• getMaxRowSize&lt;br&gt;• getMaxSchemaNameLength&lt;br&gt;• getMaxStatementLength&lt;br&gt;• getMaxStatements&lt;br&gt;• getMaxTableNameLength&lt;br&gt;• getMaxTablesInSelect&lt;br&gt;• getMaxUserNameLength&lt;br&gt;• getNumericFunctions&lt;br&gt;• getPrimaryKeys&lt;br&gt;• getProcedureColumns&lt;br&gt;• getProcedures&lt;br&gt;• getProcedureTerm&lt;br&gt;• getResultSetHoldability&lt;br&gt;• getSchemas&lt;br&gt;• getSchemaTerm&lt;br&gt;• getSearchStringEscape&lt;br&gt;• getSQLKeywords&lt;br&gt;• getSQLStateType&lt;br&gt;• getStringFunctions&lt;br&gt;• getSuperTables&lt;br&gt;• getSuperTypes&lt;br&gt;• getSystemFunctions&lt;br&gt;• getTablePrivileges&lt;br&gt;• getTables&lt;br&gt;• getTableTypes&lt;br&gt;• getTimeDateFunctions&lt;br&gt;• getTypeInfo&lt;br&gt;• getUDTs&lt;br&gt;• getURL&lt;br&gt;• getUserAgent&lt;br&gt;• getVersionColumns&lt;br&gt;• insertsAreDetected&lt;br&gt;• isCatalogAtStart&lt;br&gt;• isReadOnly&lt;br&gt;• locatorsUpdateCopy&lt;br&gt;• nullsAreSortedAtEnd&lt;br&gt;• nullsAreSortedAtStart&lt;br&gt;• nullsAreSortedHigh&lt;br&gt;• nullsAreSortedLow&lt;br&gt;• othersDeletesAreVisible&lt;br&gt;• othersInsertsAreVisible&lt;br&gt;• othersUpdatesAreVisible</td>
</tr>
</tbody>
</table>
### Table 5-3  Data Services Platform JDBC Driver Methods

<table>
<thead>
<tr>
<th>Interface</th>
<th>Supported Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.sql.DatabaseMetaData (Continued)</td>
<td>• ownDeletesAreVisible</td>
</tr>
<tr>
<td></td>
<td>• ownInsertsAreVisible</td>
</tr>
<tr>
<td></td>
<td>• ownUpdatesAreVisible</td>
</tr>
<tr>
<td></td>
<td>• storesLowerCase</td>
</tr>
<tr>
<td></td>
<td>• storesLowerCaseQuotedIdentifiers</td>
</tr>
<tr>
<td></td>
<td>• storesMixedCaseIdentifiers</td>
</tr>
<tr>
<td></td>
<td>• storesMixedCaseQuotedIdentifiers</td>
</tr>
<tr>
<td></td>
<td>• storesUpperCase</td>
</tr>
<tr>
<td></td>
<td>• storesUpperCaseQuotedIdentifiers</td>
</tr>
<tr>
<td></td>
<td>• supportsAlterTableWithAddColumn</td>
</tr>
<tr>
<td></td>
<td>• supportsAlterTableWithDropColumn</td>
</tr>
<tr>
<td></td>
<td>• supportsANSI92EntryLevelSQL</td>
</tr>
<tr>
<td></td>
<td>• supportsANSI92FullSQL</td>
</tr>
<tr>
<td></td>
<td>• supportsANSI92IntermediateSQL</td>
</tr>
<tr>
<td></td>
<td>• supportsBatchUpdates</td>
</tr>
<tr>
<td></td>
<td>• supportsCatalogsInDataManipulation</td>
</tr>
<tr>
<td></td>
<td>• supportsCatalogsInIndexDefinitions</td>
</tr>
<tr>
<td></td>
<td>• supportsCatalogsInInPrivilegeDefinitions</td>
</tr>
<tr>
<td></td>
<td>• supportsCatalogsInProcedureCalls</td>
</tr>
<tr>
<td></td>
<td>• supportsCatalogsInTableDefinitions</td>
</tr>
<tr>
<td></td>
<td>• supportsColumnAliasing</td>
</tr>
<tr>
<td></td>
<td>• supportsConvert</td>
</tr>
<tr>
<td></td>
<td>• supportsCoreSQLGrammar</td>
</tr>
<tr>
<td></td>
<td>• supportsCorrelatedSubqueries</td>
</tr>
<tr>
<td></td>
<td>• supportsDataDefinitionAndDataManipulationTransactions</td>
</tr>
<tr>
<td></td>
<td>• supportsDataManipulationTransactionsOnly</td>
</tr>
<tr>
<td></td>
<td>• supportsDifferentTableCorrelationNames</td>
</tr>
<tr>
<td></td>
<td>• supportsExpressionsInOrderBy</td>
</tr>
<tr>
<td></td>
<td>• supportsExtendedSQLGrammar</td>
</tr>
<tr>
<td></td>
<td>• supportsFullOuterJoins</td>
</tr>
<tr>
<td></td>
<td>• supportsGetGeneratedKeys</td>
</tr>
<tr>
<td></td>
<td>• supportsGroupBy</td>
</tr>
<tr>
<td></td>
<td>• supportsGroupByBeyondSelect</td>
</tr>
<tr>
<td></td>
<td>• supportsGroupByUnrelated</td>
</tr>
<tr>
<td></td>
<td>• supportsIntegrityEnhancementFacility</td>
</tr>
<tr>
<td></td>
<td>• supportsLikeEscapeClause</td>
</tr>
<tr>
<td></td>
<td>• supportsLimitedOuterJoins</td>
</tr>
</tbody>
</table>
### Table 5-3 Data Services Platform JDBC Driver Methods

<table>
<thead>
<tr>
<th>Interface</th>
<th>Supported Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.sql.DatabaseMetaData (Continued)</td>
<td></td>
</tr>
<tr>
<td>• supportsMinimumSQL Grammar</td>
<td>• supportsResultSet Type</td>
</tr>
<tr>
<td>• supportsMixedCase Identifiers</td>
<td>• supportsSavepoints</td>
</tr>
<tr>
<td>• supportsMixedCase QuotedIdentifiers</td>
<td>• supportsSchemasIn DataManipulation</td>
</tr>
<tr>
<td>• supportsMultipleOpen Results</td>
<td>• supportsSchemasIn IndexDefinitions</td>
</tr>
<tr>
<td>• supportsMultiple ResultSets</td>
<td>• supportsSchemasIn Privilege Definitions</td>
</tr>
<tr>
<td>• supportsMultiple Transactions</td>
<td>• supportsSchemasIn ProcedureCalls</td>
</tr>
<tr>
<td>• supportsNamed Parameters</td>
<td>• supportsSchemasIn TableDefinitions</td>
</tr>
<tr>
<td>• supportsNonNullable Columns</td>
<td>• supportsSelectFor Update</td>
</tr>
<tr>
<td>• supportsOpenCursors AcrossCommit</td>
<td>• supportsStatement Pooling</td>
</tr>
<tr>
<td>• supportsOpenCursors AcrossRollback</td>
<td>• supportsStored Procedures</td>
</tr>
<tr>
<td>• supportsOpen StatementsAcrossCommit</td>
<td>• supportsSubqueriesIn Comparisons</td>
</tr>
<tr>
<td>• supportsOpen StatementsAcrossRollback</td>
<td>• supportsSubqueriesIn Exists</td>
</tr>
<tr>
<td>• supportsOpen StatementsAcrossCommit</td>
<td>• supportsSubqueriesIn InIns</td>
</tr>
<tr>
<td>• supportsOrderBy Unrelated</td>
<td>• supportsSubqueriesIn Quantifieds</td>
</tr>
<tr>
<td>• supportsOuterJoins</td>
<td>• supportsTable CorrelationNames</td>
</tr>
<tr>
<td>• supportsPositioned Delete</td>
<td>• supportsTransaction IsolationLevel</td>
</tr>
<tr>
<td>• supportsPositioned Update</td>
<td>• supportsTransactions</td>
</tr>
<tr>
<td>• supportsResultSet Concurrency</td>
<td>• supportsUnion</td>
</tr>
<tr>
<td>• supportsResultSet Holdability</td>
<td>• supportsUnionAll</td>
</tr>
</tbody>
</table>

Using SQL to Access Data Services
### Table 5-3 Data Services Platform JDBC Driver Methods

<table>
<thead>
<tr>
<th>Interface</th>
<th>Supported Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.sql.DatabaseMetaData</td>
<td>• updatesAreDetected</td>
</tr>
<tr>
<td></td>
<td>• usesLocalFilePerTable</td>
</tr>
<tr>
<td>java.sql.ParameterMetaData</td>
<td>• close</td>
</tr>
<tr>
<td></td>
<td>• getParameterClassName</td>
</tr>
<tr>
<td></td>
<td>• getParameterCount</td>
</tr>
<tr>
<td></td>
<td>• getParameterMode</td>
</tr>
<tr>
<td></td>
<td>• getParameterType</td>
</tr>
<tr>
<td>java.sql.PreparedStatement</td>
<td>• addBatch</td>
</tr>
<tr>
<td></td>
<td>• clearParameters</td>
</tr>
<tr>
<td></td>
<td>• close</td>
</tr>
<tr>
<td></td>
<td>• execute</td>
</tr>
<tr>
<td></td>
<td>• executeQuery</td>
</tr>
<tr>
<td></td>
<td>• getMetaData</td>
</tr>
<tr>
<td></td>
<td>• getParameterMetaData</td>
</tr>
<tr>
<td></td>
<td>• setAsciiStream</td>
</tr>
<tr>
<td></td>
<td>• setBigDecimal</td>
</tr>
<tr>
<td></td>
<td>• setBlob</td>
</tr>
<tr>
<td></td>
<td>• setBoolean</td>
</tr>
<tr>
<td></td>
<td>• setByte</td>
</tr>
<tr>
<td></td>
<td>• setBytes</td>
</tr>
<tr>
<td>java.sql.ResultSet</td>
<td>• clearWarnings</td>
</tr>
<tr>
<td></td>
<td>• close</td>
</tr>
<tr>
<td></td>
<td>• findColumn</td>
</tr>
<tr>
<td></td>
<td>• getAsciiStream</td>
</tr>
<tr>
<td></td>
<td>• getBigDecimal</td>
</tr>
<tr>
<td></td>
<td>• getBlob</td>
</tr>
<tr>
<td></td>
<td>• getBoolean</td>
</tr>
<tr>
<td></td>
<td>• getByte</td>
</tr>
<tr>
<td></td>
<td>• getBytes</td>
</tr>
<tr>
<td></td>
<td>• getCharacterStream</td>
</tr>
<tr>
<td></td>
<td>• getConcurrency</td>
</tr>
<tr>
<td></td>
<td>• getDate</td>
</tr>
</tbody>
</table>

Note: TableMetaData is continued on the next page.
### Table 5-3 Data Services Platform JDBC Driver Methods

<table>
<thead>
<tr>
<th>Interface</th>
<th>Supported Methods</th>
</tr>
</thead>
</table>
| java.sql.ResultSet (Continued) | - getTime  
- getTimestamp  
- getType  
- getWarnings  
- next  
- setFetchDirection  
- setFetchSize  
- setMaxRows  
- wasNull |
| java.sql.ResultSetMetaData | - close  
- getCatalogName  
- getColumnClassName  
- getColumnCount  
- getColumnDisplaySize  
- getColumnLabel  
- getColumnName  
- getColumnType  
- getColumnTypeName  
- getPrecision  
- getScale  
- getSchemaName  
- getTableName  
- isAutoIncrement  
- isCaseSensitive  
- isCurrency  
- isDefinitelyWritable  
- isNullable  
- isReadOnly  
- isSearchable  
- isSigned  
- isWritable |
| java.sql.Statement | - cancel  
- clearWarnings  
- close  
- execute  
- executeQuery  
- getConnection  
- getFetchDirection  
- getFetchSize  
- getGeneratedKeys  
- getLogPrintWriter  
- getMaxFieldSize  
- getMaxRows  
- getMoreResults  
- getQueryTimeout  
- getResultSet  
- getResultSetConcurrency  
- getResultSetHoldability  
- getResultSetType  
- getUpdateCount  
- getWarnings  
- setCursorName  
- setEscapeProcessing  
- setFetchDirection  
- setFetchSize  
- setMaxFieldSize  
- setMaxRows  
- setMaxRows  
- setQueryTimeout |

Using SQL to Access Data Services
SQL Support

This section outlines SQL-92 support in the Data Services Platform JDBC driver, and contains the following sections:

- Supported SQL Statements
- Supported SQL Functions
- Table Parameter Support

Supported SQL Statements

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

Supported SQL Functions

The Data Services Platform JDBC driver supports functions that you can use to access and process data. This section describes the following supported Data Services Platform SQL-92 query language functions:

- Numeric Functions
- String Functions
- Datetime Functions
- Aggregate Functions
- JDBC Metadata Search Patterns
Numeric Functions

The Data Services Platform JDBC driver supports the numeric functions described in Table 5-4.

Table 5-4 Numeric Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>numeric ABS (numeric n)</td>
<td>ABS returns the absolute value of n. If n is NULL, the return value is NULL.</td>
</tr>
<tr>
<td>CEIL</td>
<td>numeric CEIL(numeric n)</td>
<td>CEIL returns the smallest integer greater than or equal to n. If n is NULL, the return value is NULL.</td>
</tr>
<tr>
<td>FLOOR</td>
<td>numeric FLOOR(numeric n)</td>
<td>FLOOR returns largest integer equal to or less than n. If n is NULL, the return value is NULL.</td>
</tr>
<tr>
<td>ROUND</td>
<td>numeric ROUND (numeric n)</td>
<td>ROUND returns n rounded to 0 decimal places. If n is NULL, the return value is NULL.</td>
</tr>
</tbody>
</table>

String Functions

The Data Services Platform JDBC driver supports the string functions described in Table 5-5.

Table 5-5 String Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONCAT</td>
<td>varchar CONCAT(varchar s1, varchar s2)</td>
<td>CONCAT returns s1 concatenated with s2. If any argument is NULL, it is considered to be equivalent to the empty string.</td>
</tr>
<tr>
<td>Left</td>
<td>varchar left (varchar s, numeric n)</td>
<td>Left returns the left n characters of s.</td>
</tr>
<tr>
<td>LENGTH</td>
<td>numeric LENGTH(varchar s)</td>
<td>LENGTH returns the length of s. The function returns 0 if s is NULL.</td>
</tr>
<tr>
<td>LOWER</td>
<td>varchar LOWER(varchar s)</td>
<td>LOWER returns s, with all letters lowercase. If s is NULL, the function returns an empty string.</td>
</tr>
<tr>
<td>LPAD</td>
<td>varchar lpad(varchar v, numeric n, varchar p)</td>
<td>LPAD returns v, with n characters of an infinitely repeating p appended to the left.</td>
</tr>
</tbody>
</table>
Table 5-5  String Functions  (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTRIM</td>
<td>varchar LTRIM(varchar s)</td>
<td>LTRIM trims leading blanks from s. If s is NULL, the function returns NULL.</td>
</tr>
<tr>
<td>Right</td>
<td>varchar right (varchar s, numeric n)</td>
<td>Right returns the right n characters of s.</td>
</tr>
<tr>
<td>RPAD</td>
<td>varchar rpad(varchar v, numeric n, varchar p)</td>
<td>RPAD returns v, with n characters of an infinitely repeating p appended to the right.</td>
</tr>
<tr>
<td>RTRIM</td>
<td>varchar RTRIM(varchar s)</td>
<td>RTRIM trims trailing blanks from s. If s is NULL, the function returns NULL.</td>
</tr>
<tr>
<td>SUBSTR</td>
<td>varchar SUBSTR(varchar s, numeric start)</td>
<td>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.</td>
</tr>
<tr>
<td>TRIM</td>
<td>varchar TRIM(varchar s)</td>
<td>TRIM trims leading and trailing blanks from s. If s is NULL, TRIM returns NULL.</td>
</tr>
<tr>
<td>UPPER</td>
<td>varchar UPPER(varchar s)</td>
<td>UPPER returns s, with all letters uppercase. If s is NULL, UPPER returns the empty string.</td>
</tr>
</tbody>
</table>

Datetime Functions

The Data Services Platform JDBC driver supports the datetime functions described in Table 5-6.

Table 5-6  Datetime Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAYS</td>
<td>numeric DAYS(T value)</td>
<td>DAYS returns the days component from value. T can be a date, timestamp, or duration. If value is NULL, the result is NULL.</td>
</tr>
<tr>
<td>HOUR</td>
<td>numeric HOUR(T value)</td>
<td>HOUR returns the hour component from value. T can be one of time, timestamp, or duration. If value is NULL, the result is NULL.</td>
</tr>
</tbody>
</table>
Aggregate Functions

The Data Services Platform JDBC driver supports the aggregation functions described in Table 5-7.

### Table 5-7 Aggregate Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>numeric COUNT(ROWS r)</td>
<td>COUNT returns the number of rows in r.</td>
</tr>
<tr>
<td>AVG</td>
<td>T AVG(T r)</td>
<td>AVG returns the average values of all values in r. T can be a numeric or duration type.</td>
</tr>
<tr>
<td>SUM</td>
<td>T SUM(T r)</td>
<td>SUM returns the sum of all values in r. T can be a numeric or duration type.</td>
</tr>
</tbody>
</table>
JDBC and SQL Support in Data Services Platform

Table 5-7  Aggregate Functions (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX</td>
<td>$T \text{ MAX}(T \ r)$</td>
<td>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.</td>
</tr>
<tr>
<td>MIN</td>
<td>$T \text{ MIN}(T \ r)$</td>
<td>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.</td>
</tr>
</tbody>
</table>

**JDBC Metadata Search Patterns**

The Data Services Platform JDBC driver supports standard JDBC API search patterns, as shown in Table 5-8.

Table 5-8  JDBC Driver Metadata Search Patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>“string”</td>
<td>Matches the identified string.</td>
</tr>
<tr>
<td>“”</td>
<td>Uses the default catalog/schema.</td>
</tr>
<tr>
<td>“%”</td>
<td>Wildcard; equivalent to * in regular expressions.</td>
</tr>
<tr>
<td>“_”</td>
<td>Matches a single character; equivalent to . (period) in regular expressions.</td>
</tr>
<tr>
<td>null</td>
<td>Wildcard; same as “%”</td>
</tr>
</tbody>
</table>

**Note:** For more information about using the JDBC metadata API, refer to the Java documentation at:

http://java.sun.com/j2se/1.5.0/docs/api/java/sql/
DatabaseMetaData.html
Assuming that the default_catalog is catalog1 and default_schema is schema1, Table 5-9 shows some common matching patterns.

Table 5-9  JDBC Driver Metadata Search Patterns

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matching Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>“AquaLogic”</td>
<td>Matches the identified string, AquaLogic.</td>
</tr>
<tr>
<td>“abc%d”</td>
<td>Matches:</td>
</tr>
<tr>
<td></td>
<td>• abc10d</td>
</tr>
<tr>
<td></td>
<td>• abcd</td>
</tr>
<tr>
<td></td>
<td>• abc_practically anything_d</td>
</tr>
<tr>
<td></td>
<td>But not:</td>
</tr>
<tr>
<td></td>
<td>• abc10e</td>
</tr>
<tr>
<td></td>
<td>• abc10def</td>
</tr>
<tr>
<td>abc%d_</td>
<td>Matches:</td>
</tr>
<tr>
<td></td>
<td>• abc10d</td>
</tr>
<tr>
<td></td>
<td>• abcd</td>
</tr>
<tr>
<td></td>
<td>• abc_practically anything_d</td>
</tr>
<tr>
<td></td>
<td>• abc10dg</td>
</tr>
<tr>
<td></td>
<td>But not:</td>
</tr>
<tr>
<td></td>
<td>• abc10dgh</td>
</tr>
<tr>
<td></td>
<td>• abc10dgPattern</td>
</tr>
<tr>
<td>“” and null</td>
<td>A call to:</td>
</tr>
<tr>
<td></td>
<td>DBDatabaseMetadata.getTables(&quot;&quot;,null,&quot;abc%&quot;)</td>
</tr>
<tr>
<td></td>
<td>would return all tables starting with abc under catalog 1.</td>
</tr>
</tbody>
</table>

Table Parameter Support

The Data Services Platform JDBC driver extends the standard SQL-92 parameter model by providing the ability to add table parameters to SQL FROM clauses. For example, in SQL you might encounter a situation where it is necessary to specify a list of parameters (highlighted) in a query.

In the following query, JDBCdemo.empData.empinfo is the entire customer table.

```
SELECT emp.empid, emp.name, emp.salary
FROM JDBCdemo.empData.empinfo emp
```
WHERE emp.empid in (?, ?, ?, ...)  
or emp.name in (?, ?, ?, ...)

If the number of parameters can vary, you need to specify a query for each case. Table parameters provide an alternative by enabling you to specify that the query accept a list of values (the list can be of variable length). The following query uses table parameters (highlighted):

```
SELECT emp.empid, emp.name, emp.salary  
FROM JDBCdemo.empData.empinfo emp  
WHERE emp.empid in (SELECT * FROM ? as emp(empid))  
or emp.name in (SELECT * FROM ? as emp(empname))
```
The table parameter is specified using the same mechanism as a parameter; a question mark ("?"") is used in place of the appropriate table name.

**Note:** You can only pass a table with a single column as a table parameter. If you specify more than one column, an exception is thrown.

For more information about using table parameters, see Using Table Parameters.

### Additional Details and Limitations

When using the ALSDSP JDBC driver, each connection points to one Data Services Platform dataspace. Table 5-10 notes the Data Services Platform JDBC driver limitations that apply to SQL language features.

**Table 5-10  Data Services Platform JDBC Driver Limitations Applying to SQL Language Features**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Comments</th>
<th>Example</th>
</tr>
</thead>
</table>
| 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) |
Preparing to Use SQL to Access Data Services

This section describes the tasks you need to perform prior to using SQL to access data services, and contains the following topics:

- Publishing Data Service Operations
- Configuring the Data Services Platform JDBC Driver

Publishing Data Service Operations

To access data services using SQL, you first need to publish the data service operations as SQL objects within the AquaLogic Data Services Platform-enabled project. These SQL objects include tables, stored procedures, and functions.

*Note:* SQL objects published through Data Services Platform need to be enclosed in double quotes when used in an SQL query, if the object name contains a hyphen. For example:  
```
SELECT "col-name" FROM "table-name"
```

To publish data service operations as SQL Objects, perform the following steps:

1. Publish the data service operations to a schema that models the operations as SQL objects.
2. Build and deploy the Data Services Platform dataspace.

After the dataspace is deployed, the newly created SQL objects are available to the dataspace through the ALSDSP JDBC driver.

Configuring the Data Services Platform JDBC Driver

The Data Services Platform JDBC driver is located in the `ldjdbc.jar` file, which is available in the `<ALDSP_HOME>/lib` directory after you install BEA AquaLogic Data Services Platform. To use the Data Services Platform JDBC driver on a client computer, you need to configure the classpath, class name, and the URL for the JDBC driver.

*Note:* You will need gateway software to enable connectivity between the JDBC driver and DSP to configure the JDBC driver. For more information, refer to the section entitled Accessing Data Services Using SQL Explorer.
To configure the driver on a client computer, perform the following steps:

1. Copy the ldjdbc.jar and weblogic.jar (in the <ALDSP_HOME>/lib and <WL_HOME>/server/lib directories respectively) to the client computer.

2. Add ldjdbc.jar and weblogic.jar to the classpath on the client computer.

3. Set the appropriate supporting path by adding %JAVA_HOME%/jre/bin to the path on the client computer.

4. To set the JDBC driver, do the following:
   a. Set the driver class name to the following:
      com.bea.dsp.jdbc.driver.DSPJDBCDriver
   b. Set the driver URL to the following:
      jdbc:dsp@<DSPServerName>:<ALDSPServerPortNumber>/<DataspaceName>
      For example the driver URL could be:
      jdbc:dsp@localhost:7001/Test_DataSpace
      Alternatively, set the default catalog name and schema name in the URL while connecting to the JDBC driver using the following syntax:
      jdbc:dsp@<DSPServerName>:<ALDSPServerPortNumber>/<DataspaceName>/<catalogname>/<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
   c. Optionally, enable debugging 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 is created in the c:\output.txt file. You can also specify the debug property separately instead of specifying it with the URL.
      Note: If you build an SQL query using a reporting tool, the unqualified JDBC function name is used in the generated SQL. Consequently, to enable application developers to invoke an 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/myDataspace/myCatalog/mySchema
```

**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. If `dataspace`, `default_catalog`, or `default_schema` appears in both the connection properties and the URL, the variable in the URL takes precedence.

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

For more information, see Configuring the Connection Using the Properties Object or Configuring the Connection in the JDBC URL respectively.

---

**Accessing Data Services Using SQL From a Java Application**

You can have a Java application access information from data services using SQL through the Data Services Platform JDBC driver.

To access the data from a Java application, perform the following steps:

1. Obtain a connection to the Data Services Platform dataspace.
   
   For more information, see Obtaining a Connection.

2. Specify and submit an SQL query to the JDBC datasource.

   You can use either the `PreparedStatement` or `CallableStatement` interface to specify and submit the query to the datasource. For more information, see Using the PreparedStatement Interface and Using the CallableStatement Interface respectively.
Obtaining a Connection

A JDBC client application can connect to a deployed Data Services Platform dataspace by loading the Data Services Platform JDBC driver and then establishing a connection to the dataspace. In the database URL, use the Data Services Platform dataspace name as the database identifier with "dsp" as the sub-protocol, using the following form:

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

For example:

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

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

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

Configuring the Connection Using the Properties Object

You can establish a connection to an Data Services Platform dataspace using the Properties object as follows:

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

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

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

Alternatively, you can specify the Data Services Platform dataspace name, `TestProjectDataSpace`, in the connection object itself, as shown in the following segment:

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

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

// Get the connection
Connection objConnection = DriverManager.getConnection("jdbc:dsp@localhost:7001/TestProjectDataSpace",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 segment:

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

// Get the connection
Connection objConnection = DriverManager.getConnection(
    "jdbc:dsp=localhost:7001/TestProjectDataSpace;logFile=
    c:/output.txt; ", <username>, <password>);
```

Using the PreparedStatement Interface

You can use the `preparedStatementWithParameters` method to specify a query to the JDBC datasource using the connection object (`conn`), obtained earlier. The connection object is obtained through the `java.sql.Connection` interface to the WebLogic Server, which hosts Data Services Platform.

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

In this query, the data service function `getAll()` in the data service `EmpInfo.ds` under the `SQLHowTo` project is mapped using SQL Mapper to `JDBCdemo.empData.empinfo`.

```java
public ResultSet preparedStatementWithParameters(Connection conn) throws java.sql.SQLException {
    PreparedStatement ps = conn.prepareStatement("SELECT *
        FROM JDBCdemo.empData.empinfo emp WHERE emp.salary >= ?");
    ps.setInt(1,275000);
    ResultSet rs = ps.executeQuery();
    return rs;
}
```

In the SELECT query, `JDBCdemo` is the catalog name, `empData` is the schema name, and `empinfo` is the table name.

**Note:** For more information about how to map data service operations to SQL objects, refer to “Publishing Data Service Operations” on page 5-21.
Using the CallableStatement Interface

After you establish a connection to a server where Data Services Platform is deployed, you can call a data service operation to obtain data using a parameterized data service operation call.

The following example shows how to call a stored query with a parameter (where conn is a connection to the Data Services Platform server obtained through the java.sql.Connection interface). In the segment, a stored query named getBySalary is called passing a parameter with a value of 275000.

```java
public ResultSet storedQueryWithParameters(Connection conn) throws java.sql.SQLException {
    CallableStatement ps = conn.prepareCall("call JDBCdemo.empData.getBySalary(?)");
    ps.setInt(1, 275000);
    ResultSet rs = ps.executeQuery();
    return rs;
}
```

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

```java
conn.prepareCall("{call JDBCdemo.empData.getBySalary(?)}");
```

Advanced Features

This section describes advances features and uses of the Data Services Platform JDBC driver and contains the following sections:

- **Using Table Parameters**
- **Accessing Custom Database Functions Using JDBC**

Using Table Parameters

This section describes how to use the Data Services Platform JDBC driver to pass table parameters to data services.

When to Use Table Parameters

Consider the case in which a data service contains consolidated information of all employee contact information. A manager further has a consolidated list of all government employees in European countries. The goal is to use a data service to obtain contact information for that specific subset of employees.
The scenario is a common one involving the need for a join between the manager’s employee list and contact information. However, if the manager’s employee list is long and not already available through a database, it is convenient to pass a list of values as if it were a column in a table.

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

```sql
? as emp(empid)
```

provides a virtual table value (emp) and a virtual column name (empid).

**Note:** You should alias all table parameters since the default table/column names are undefined and may produce unexpected name conflicts.

### Setting Table Parameters Using JDBC

The Data Services Platform JDBC driver passes table parameters to data services through its TableParameter class. The class (shown in its entirety in Listing 5-1) represents an entire table parameter and the rows it represents.

#### Listing 5-1  Table Parameter Interface

```java
public class TableParameter implements Serializable {
    /**
     * Constructor
     *
     * @param schema the schema for the table parameter
     */
    public TableParameter(ValueType[] schema);

    /**
     * Creates a new a row and adds it to the list of rows in this table parameter
     */
    public Row createRow();

    /**
     * Gets the rows of this table parameter
     */
    public List<Row> getRows();

    /**
     * Gets the schema of this table parameter
     */
    public ValueType[] getSchema();

    /**
     * Represents a row in the table parameter
     */
    public class Row implements Serializable {
        /**
         * Constructor
         */
        public Row(ValueType[] values);

        /**
         * Gets the values of this row
         */
        public ValueType[] getValues();
    }
}
```
public class Row implements Serializable {
    /**
     * Sets a value to a particular column
     * @param colIdx    the index of the column to set, always 1
     * @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 instance and populate the instance with data:

1. Instantiate a TableParameter with the schema of your table.
   
   **Note:** At present only one column is supported for table parameters.

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

3. Use the setObject(1,val) call to set the column on the Row object.

4. Call createRow() again to create as many rows as the table requires.

JDBC Usage

You can pass table parameters through JDBC just like any other parameter, using the PreparedStatement interface.

To pass table parameters using the PreparedStatement interface:

1. Create a PreparedStatement with the query, as shown in the following:

   ```java
   PreparedStatement ps = c.prepareStatement("SELECT * " +
   "FROM ? as EMP(empid), JDBCdemo.empData.contact CONTACT " +
   "WHERE CONTACT.empid = EMP.empid AND CONTACT.zip=?");
   ```

2. Set the value of the normal parameter on the PreparedStatement, as shown in the following:

   ```java
   ps.setObject(2,"98765");
   ```

3. Create a table parameter of a specific type, as shown in the following:
ValueType[] tableType = new ValueType[1];
tableType[0] = ValueType.REPEATING_INTEGER_TYPE;
TableParameter p = new TableParameter(tableType);

4. Fill the table parameter by reading rows from a file or other input stream, as shown in the following:

```java
String empidlist = FileUtils.slurpFile("empidlist.txt");
StringTokenizer empids = new StringTokenizer(empidlist,"\n");
while(empids.hasMoreTokens()) {
    TableParameter.Row r = p.createRow();
    r.setObject(1,new Integer(empids.nextToken()));
}
ps.setObject(1,p);
```

5. Set the table parameter as a property of the prepared statement, as shown in the

```java
ps.setObject(1,p);
```

**Table Parameter Example**

The following simplified example illustrates the use of a table parameter. The supporting JDBC code is shown in [Listing 5-2]:

**Listing 5-2  JDBC Code Supporting Table Parameter Example**

```java
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.PreparedStatement;
import java.sql.Connection;
import java.sql.Driver;
import java.util.Properties;
import java.util.StringTokenizer;
import com.bea.ld.sql.types.ValueType;
import com.bea.ld.sql.data.TableParameter;
import weblogic.xml.query.util.FileUtils;

public class TableParameterTest {

    /**
     * Establish a connection to the ALDSP JDBC driver and return it
     */
    protected static Connection connect() throws Exception {
        // Attempt to locate the JDBC driver
        Class.forName("com.bea.dsp.jdbc.driver.DSPJDBCDriver");
```
Driver driver =
    DriverManager.getDriver("jdbc:dsp@localhost:7001");
if(driver == null)
    throw new IllegalStateException("Unable to find driver.");

//Set the connection properties to the driver
Properties props = new Properties();
props.setProperty("user", "weblogic");
props.setProperty("password", "weblogic");
props.setProperty("application", "SQLHowTo");

// Try to connect to the driver using the properties set above
Connection c =
    driver.connect("jdbc:dsp@localhost:7001", props);
if(c == null)
    throw new IllegalStateException("Unable to establish a
    connection.");
return c;

/**
 * Prints a result set to system out
 * @param rs the result set to print
 */
protected static void printResultSet(ResultSet rs)
    throws Exception{
    while(rs.next()) {
        for(int i = 1; i < rs.getMetaData().getColumnCount()+1; i++) {
            rs.getObject(i);
            System.err.print(rs.getObject(i) + "   ");
        }
        System.err.println();
    } 
    System.err.println();
    rs.close();
}

public static void main(String args[]) throws Exception {
    Connection c = connect();

    // Create the query
    PreparedStatement ps = c.prepareStatement("SELECT * "+
        "FROM ? as EMP(empid), JDBCdemo.empData.contact CONTACT "+
        "WHERE CONTACT.empid = EMP.empid AND CONTACT.zip=?");

    // Set the normal parameter
    ps.setObject(2,"98765");

    // Create the table parameter
    ValueType[] tableType = new ValueType[1];
tableType[0] = ValueType.REPEATING_INTEGER_TYPE;
TableParameter p = new TableParameter(tableType);

// Create the rows of the table parameter from values in a file
String empidlist = FileUtilities.slurpFile("empidlist.txt");
StringTokenizer empids = new StringTokenizer(empidlist, "\n");
while (empids.hasMoreTokens()) {
    TableParameter.Row r = p.createRow();
    r.setObject(1, new Integer(empids.nextToken()));
}
ps.setObject(1, p);
// Run the query and print the results
ResultSet rs = ps.executeQuery();
printResultSet(rs);
}

Table Parameter ValueTypes

Table 5-11 lists the table parameter ValueTypes supported by the Data Services Platform JDBC driver.

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Type Value</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ValueType.REPEATING_SMALLINT</td>
<td>16 bit signed integer</td>
<td>Short</td>
</tr>
<tr>
<td>ValueType.REPEATING_INTEGER</td>
<td>32 bit signed integer</td>
<td>Integer</td>
</tr>
<tr>
<td>ValueType.REPEATING_BIGINT</td>
<td>64 bit signed integer</td>
<td>Long</td>
</tr>
<tr>
<td>ValueType.REPEATING_REAL</td>
<td>32 bit floating point</td>
<td>Float</td>
</tr>
<tr>
<td>ValueType.REPEATING_DOUBLE</td>
<td>64 bit floating point</td>
<td>Double</td>
</tr>
<tr>
<td>ValueType.REPEATING_DECIMAL</td>
<td>decimal</td>
<td>BigDecimal</td>
</tr>
<tr>
<td>ValueType.REPEATING_VARCHAR</td>
<td>string</td>
<td>String</td>
</tr>
<tr>
<td>ValueType.REPEATING_DATE</td>
<td>date</td>
<td>java.sql.Date</td>
</tr>
<tr>
<td>ValueType.REPEATING_TIME</td>
<td>time</td>
<td>java.sql.Time</td>
</tr>
<tr>
<td>ValueType.REPEATING_TIMESTAMP</td>
<td>datetime</td>
<td>java.sql.Timestamp</td>
</tr>
<tr>
<td>ValueType.REPEATING_BLOB</td>
<td>byte array</td>
<td>char[]</td>
</tr>
</tbody>
</table>
Several relational database management systems provide mechanisms to extend the library of built-in, standard SQL functions with user-defined, custom functions, defined using another language, such as PL/SQL, that can be directly embedded in SQL statements.

You can make these built-in or custom functions in your database available through data services by registering the function with Data Services Platform through a library. After registering the functions, you can use them in SQL statements submitted to the Data Services Platform JDBC driver. The following example shows the use of the custom function `myLower()` in a SELECT statement:

```
select * from CUSTOMER where ? = myLower( LAST_NAME )
```

Note that the following conditions must be met to enable Data Services Platform to use database-specific or user-defined functions:

- The function must accept at least one argument using the standard syntax `myFunction(arg1, arg2)`. This argument must be from the data source for which the function is defined. Remaining arguments, however, may be constants or arguments from another type of data service, such as a web service.

- ALDSP does not support functions of the form `TRIM( TRAILING ' ' FROM $column)` as custom database functions.

- Data Services Platform does not support special columns such as `SYSDATE`, `ROWNUM`, or similar columns as parameters to custom database functions.

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Type Value</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ValueType.REPEATING_BOOLEAN</td>
<td>Boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>ValueType.REPEATING_YMINTERVAL</td>
<td>year month interval</td>
<td>weblogic.xml.query.datetime.YearMonthDuration</td>
</tr>
<tr>
<td>ValueType.REPEATING_DTINTERVAL</td>
<td>day time interval</td>
<td>weblogic.xml.query.datetime.DayTimeDuration</td>
</tr>
<tr>
<td>ValueType.REPEATING_INTERVAL</td>
<td>both year month &amp; day time interval</td>
<td>weblogic.xml.query.datetime.Duration</td>
</tr>
</tbody>
</table>

**Accessing Custom Database Functions Using JDBC**

Table 5-11  TableParameter ValueTypes (Continued)
• You must explicitly expose Data Services Platform artifacts in the SQL Map for the dataspace.
Accessing Data Services Using SQL-Based Applications

You can access data services using both SQL-based applications and applications that connect to the Data Services Platform JDBC driver through an ODBC-JDBC bridge. This section describes how to configure SQL and ODBC-based applications to access data services, and contains the following sections:

- Accessing Data Services Using SQL Explorer
- Connecting to the Data Services Platform Client Using OpenLink ODBC-JDBC Bridge
- Using OpenLink with Reporting Tools

Note: You can also use the Data Services Platform JDBC driver with the Eclipse Data Tools Platform (DTP) plug-in. To use DTP, download the DTP software using the following link:

http://wiki.eclipse.org/index.php/Getting Started with DTP

Accessing Data Services Using SQL Explorer

You can use the Data Services Platform JDBC driver with Eclipse SQL Explorer to access data services. This section describes how to configure SQL Explorer to use the Data Services Platform JDBC driver and how to specify the connection settings. This section assumes that you have already defined your web server and dataspace project in Eclipse.

Note: SQL Explorer does not support stored procedures and, therefore, data services exposed as stored procedures through the Data Services Platform JDBC driver do not appear in SQL Explorer. For more information, refer to the Eclipse SQL Explorer web site at:

http://eclipsesql.sourceforge.net

To use SQL Explorer, perform the following steps:

1. Download the SQL Explorer software from the following link:
   
   http://sourceforge.net/projects/eclipsesql

2. After you have downloaded the SQL Explorer zip file, extract two folders, Features and Plug-Ins.

3. Copy the SQL Explorer files in the Features folder into the Eclipse Features folder.

4. Copy the SQL Explorer files in the Plug-ins folder into the Eclipse Plug-ins folder.
5. Launch Eclipse in the AquaLogic Perspective. Start the web server within Eclipse and open the dataspace (project).

6. Choose Window → Preferences, expand SQL Explorer in the left margin, and select JDBC Drivers. Click Add and type the driver name, URL, and class name, as follows:
   - Type a name for the JDBC Driver, such as aldsp_jdbc_driver.
   - Set the example URL to:
     `jdbc: dsp@<DSPServerName>:<DSPServerPortNumber>/<DSPDataspaceName>`

   **Figure 5-12 Create New Driver Dialog**

7. Click the Extra Class Path tab and then click Add. Enter the paths for two JAR files, as follows:
   - `<ALDSP_HOME>/weblogic92/aldsp_3.0/lib/ldjdbc.jar`
   - `<ALDSP_HOME>/weblogic92/server/lib/weblogic.jar`

   where `<ALDSP_HOME>` is the home directory of the Data Services Platform installation. Click OK.
8. Set the Driver Class Name to the following:
   com.bea.dsp.jdbc.driver.DSPJDBCDriver
   Click OK twice.

9. Open the SQL Explorer perspective by choosing Open Perspective → Other → SQL Explorer. Click OK.

10. Click on the far left icon under Connections to create a new connection. Enter a name for the new connection and choose Data Services Platform JDBC Driver from the drop-down list. Enter the URL for the JDBC Driver, then enter the user name and password, and click OK.

   ![Create New Connection Profile Dialog]

   **Figure 5-13 Create New Connection Profile Dialog**

11. Right-click the new JDBC Driver connection and choose Edit to ensure that you have the correct connect profile for the JDBC driver.

12. Right-click on JDBC Driver connection and choose Connect. Verify that the connection profile is correct in the Connection dialog, then click OK.
13. If the JDBC Driver window is not open, choose Window → Show View → Other → SQL Explorer Database Structure → OK to display the client data.

14. If you get an exception message, add the catalog name and schema name to the JDBC Connection URL, as follows:

\[jdbc:dsp@<DSPServerName>:<DSPServerPortNumber>/<DSPDataspaceName>/<Your_CatalogName>/<Your_SchemaName>\]

The data displays in the Database Structure and Database Detail window.
Connecting to the Data Services Platform Client Using OpenLink ODBC-JDBC Bridge

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

You can use the Openlink ODBC-JDBC driver to interface with the Data Services Platform JDBC driver to query ALDSP dataspaces with client applications such as Crystal Reports, Business Objects XI, Microsoft Access 2003, and Microsoft Excel 2003.

To use the OpenLink bridge, you need to install the bridge and create a system DSN using the bridge. The following describes the steps to complete these two tasks:

1. Install the OpenLink ODBC-JDBC bridge (called ODBC-JDBC-Lite).

   For information on installing OpenLink ODBC-JDBC-Lite, refer to the OpenLink Software download page for the Single-Tier (Lite Edition) ODBC to JDBC Bridge Driver (Release 6.0) for use on Windows systems. The page can be accessed at:


   **WARNING:** For Windows platforms, be sure to save the value of your CLASSPATH before installation.
2. Create a system DSN and configure it for your ALDSP dataspace. Ensure that the CLASSPATH contains the following JAR files required by ODBC-JDBC-Lite, ldjdbc.jar and weblogic.jar. A typical CLASSPATH might look as follows:

```java
<ALDSP_HOME>/lib/ldjdbc.jar;
<WL_HOME>/server/lib/weblogic.jar;
```

3. Update the system path to include the jvm.dll file, which should be in the

   `<ALDSP_HOME>%javaroot%/jre/bin/server` directory.

**Note:** Do not include the file name jvm.dll in the system path.

4. Launch Control Panel → Administrative Tools → Data Sources (ODBC). The ODBC Data Source Administrator window displays.

5. Click the System DSN tab and then click Add.

6. Select JDBC Lite for JDK 1.5 (6.0) and click Finish.

7. Specify the DSN name, for example, openlink-aldsp.

8. Click Next. Then on the next screen, enter the following next to the JDBC driver:

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

9. Type the following in the URL string field:

   ```java
   jdbc: dsp@<machine_name>:<port>/<dataspace_name>/<catalog_name>/<schema_name>
   ```

10. Select the “Connect now to verify that all settings are correct” checkbox. Type the login ID and password to connect to the Data Services Platform WebLogic Server, and click Next.

11. Select any additional parameters, and click Next.

12. Click Next and specify the connection compatibility parameters.

13. Click Next, and then click Test Data Source. Verify that the setup was successful.

14. Click Finish.

Using OpenLink with Reporting Tools

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

- **Crystal Reports XI**
Using SQL to Access Data Services

- Business Objects XI-Release 2 (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, which can affect performance.
Crystal Reports XI
This section describes the steps to connect Crystal Reports to the Data Services Platform JDBC
driver along with information about standard configuration files that are available with Data
Services Platform installation. It also describes the limitations of using Crystal Reports with Data
Services Platform and includes the following topics:

- Configuring Crystal Reports
- Limitations
- Connecting to Crystal Reports Using JDBC

Configuring Crystal Reports
Before you start using Crystal Reports with Data Services Platform, you must modify the default
Crystal Reports configuration file, CRConfig.xml to ensure that Crystal Reports is able to access
data services through JDBC. The configuration file is located in the following directory:

<Drive>\Program Files\Business Objects\Common\3.5\java

A modified sample CRConfig.xml file and an associated CR_Readme.txt file are available at:

<ALDSP_HOME>/samples/ReportingTools/config/Crystal Reports

The CR_Readme.txt file contains instructions on how to apply DSP-specific rules in an existing
Crystal Reports installation and how to modify the CRConfig.xml file.

Limitations
Before you use Crystal Reports to access data services, consider the following information:

- Crystal Reports is not able to invoke stored procedures with parameters for any Data
  Services Platform XQuery function that has the parameters defined using a 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 Data Services Platform
  JDBC driver except the following:
    - yearMonthDuration
    - dayTimeDuration

- Certain JDBC functions used by Crystal Reports are not supported by Data Services
  Platform. Refer to the “Supported SQL Functions” on page 5-13 for a list of supported
  functions.
Connecting to Crystal Reports Using JDBC

This section assumes that you have successfully completed the following:

- Installed Crystal Reports 11 using the Update Manager in Eclipse. For more information, refer to the instructions at the following location:
  
  [http://diamond.businessobjects.com/node/432](http://diamond.businessobjects.com/node/432)

- Modified the CRConfig.xml file (see Configuring Crystal Reports),

- Built your Crystal Reports Project

- Started your Weblogic92 server running on Eclipse.

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

1. Using the Crystal Reports perspective in Eclipse, create a new connection to the JDBC driver by clicking on the New Connection icon in the menu bar of the Database Explorer window.

2. Specify the Connection parameters for the JDBC interface of Crystal Reports. The New Connection window displays.

3. Check the box under Connection Identification if you want to use the default naming convention. Otherwise, leave the checkbox and associated field blank.

4. Select the driver from the drop-down list next to the JDBC Driver, or select Database Manager from the drop-down window in the left margin.

5. Type the JDBC Driver Class name, the Class Location, and the URL for the JDBC Driver.

6. Under User Information, type your user ID and password, then click Test Connection. If the connection is successful, click Next. The Filter dialog displays.

7. Select a predicate and type a value. Alternatively, indicate whether to include or exclude a selection of items.

8. If you do not want a filter, check the Disable Filter box and click Finish.

9. The data is displayed in the Database Explorer.
Business Objects XI-Release 2 (ODBC)

Business Objects enables 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.

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

- Configuring Business Objects
- Prerequisites and Limitations
- Generating a Business Objects Report

Configuring Business Objects

There are two Business Objects configuration files, odbc.prm and odbc.sbo, available with the
standard Business Objects installation.

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

```plaintext
<Business_Objects_Home>/BusinessObjects Enterprise 11.5/
win32_x86/dataAccess/connectionServer/odbc
```

An Data Services Platform installation includes samples of these configuration files along with
an associated BO_Readme.txt file, available at the following location:

```plaintext
<ALDSP_HOME>/samples/ReportingTools/config/BusinessObjects
```

You can edit the Business Objects configuration files according to the instructions in the readme
file.

**Tip:** When first getting started using Business Objects with Data Services Platform, use the
included configuration file to verify the ability to access data services through JDBC.
Table 5-16 identifies some restrictions and specifies configuration changes you may want to make to your Business Objects configuration files when accessing data using the Data Services Platform JDBC driver.

### Prerequisites and Limitations

Before you start using Business Objects to access data services, consider the following information:

- To generate a report using Business Objects, all the tables need to be from the same catalog. (This requirement implies that in the ALDSP IDE, the same project should be used to publish data services for SQL use.) This catalog needs to be defined as the default catalog when connecting to ALDSP. However, the catalog can contain an arbitrary number of schemas.

- Avoid publishing the same table name in multiple schemas. Doing so will cause the same column name to appear multiple times (as many times as the table is published) in the Business Objects’ Table Browser user interface.

- Business Objects does not generate qualified names for tables. When you connect Business Objects to Data Services Platform using OpenLink, you need to specify the default catalog and schema names in the Data Services Platform JDBC driver URL when configuring OpenLink. If you do not specify the default catalog and schema names, then an error similar to the following is generated:

<table>
<thead>
<tr>
<th>Configuration File</th>
<th>Discussion</th>
</tr>
</thead>
</table>
| 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 |
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, “Connecting to the Data Services Platform Client Using OpenLink ODBC-JDBC Bridge” on page 5-38. Business Objects 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 by Running the Business Objects Designer application and clicking Begin in the Quick Design Wizard window.

2. Type a name for your Universe and choose the appropriate DSN connection from the drop-down list.

3. If the DSN you want to use does not appear in the list (which happens if you are using the application for the first time), then click New to create a new connection.

4. In the Define a New Connection wizard, select Generic ODBC3 Datasource as the middleware. Click Next.

5. 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 “Connecting to the Data Services Platform Client Using OpenLink ODBC-JDBC Bridge” on page 5-38.

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

7. After creating the connection, specify this connection in the Universe and click OK. A new blank panel is displayed.
8. Choose Table from the Insert menu. After the list of tables is shown in the Table Browser, double click the tables that you want to put in the Universe.

9. Save the Universe and exit.

10. To create a new report, run the Desktop Intelligence application. Click New to open the New Report Wizard.

11. Select the report layout and report data and click Begin. Select a universe and click Next. Highlight the universe you want to use.

   If you want to make the selected universe the default universe, check the box next to Set as my Default universe. Click Finish.

   On the left pane, you should see the tables and their fields (columns) on expansion.

12. Double-click a column (table-field) in the left pane to select it in the result.

13. Click Run to execute the query.

You can also run the pass-through queries using the Desktop Intelligence application.

To run pass-through queries, perform the following steps:

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


3. Choose Free-hand SQL and click Finish.

4. Select the connection you created using Designer.

5. Type in the SQL query and click Run to generate the report.

**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. Data Services Platform JDBC driver extracts only the last two parts from the SELECT list item, and ignores the rest.
For example:

```sql
SELECT E.Name FROM JDBCdemo.empData.empinfo E
where
  JDBCdemo is the catalog name
  empData is the schema name
  empinfo is the table name
  Name is the column name
  E is the table alias for empinfo
```

**Microsoft Access 2003-ODBC**

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

- Generating Reports Using Microsoft Access
- Limitations and Usage Notes

**Generating Reports Using Microsoft Access**

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

1. Run MS Access, choose File → Open, then select ODBC Databases as the file type. The Select Data Source dialog displays.
2. Click Cancel to close the Select Data Source dialog.
3. Click Queries, then Design.
4. Close the Show Table dialog box. The Select Query window should be visible.
5. Right-click in the window and choose SQL Specific → Pass-Through.
6. Type the SQL query and click Run.
7. Click the Machine Data Source tab in the dialog that appears, and select openlink-aldsp to connect to the Data Services Platform JDBC driver and generate the report.
Limitations and Usage Notes

- The 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 mapping results in a map 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 Data Services Platform using MS Access.

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

Microsoft Excel 2003-ODBC

This section describes the procedure for connecting Microsoft Excel 2003 to ALDSP through an ODJB-JDBC bridge using OpenLink.

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

1. Launch Data Services Studio and then start the WebLogic Server.

2. Build and deploy the Data Services Platform dataspace.

3. Start Microsoft Excel and open a new worksheet.

4. Click Data → Import External Data → New Database Query. The Choose Data Source dialog box displays.

5. Select openLink-aldsp from the list of data sources and then click OK.
   
   The Query Wizard - Choose Columns dialog box displays. For details on configuring the JDBC driver using OpenLink, refer to “Connecting to the Data Services Platform Client Using OpenLink ODBC-JDBC Bridge” on page 5-38.

6. Select the tables that you want to use 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 display in the spreadsheet.
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 Data Services Platform JDBC driver. Here is an example:

```
SELECT Sum(EMP.SALARY) AS 'Salary Cost' FROM JDBCdemo.empData.empinfo emp
```

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

```
http://support.microsoft.com/kb/298955/en-us
```
Using Excel to Access Data Services

This chapter introduces the AquaLogic Data Services Platform™ Excel Add-in, and describes how to install and use the add-in to access data service operations.

The chapter covers the following topics:

- Introducing the Excel Add-in
- Installing the Excel Add-in
- Preparing to Use the Excel Add-in
- Accessing the Data Service Using Excel
Introducing the Excel Add-in

You can use the AquaLogic Data Services Platform Excel Add-in to invoke data service operations from Microsoft® Excel®. This has multiple uses, including the following:

- 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.

Installing the Excel Add-in

This section describes the ALDSP Excel Add-in system requirements and how to install the add-in.

System Requirements

The ALDSP Excel Add-in adds the following system requirements:

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

Installation Instructions

The Data Services Platform Excel Add-in is provided with Data Services Platform as a separate installation program. The Excel Add-in installs on your local machine and is accessible from Excel as a menu item.
Preparing To Install

Before installing the Excel Add-in, uninstall any previous versions of the add-in by clicking Start → All Programs → BEA AquaLogic Data Services Platform Excel Add-In 3.0 → Unload Add-in from Excel.

Installing the Add-in

This section describes how to install the AquaLogic Data Services Platform Excel Add-in.

**Note:** Microsoft .NET with SP1 is required for the ALDSP Excel Add-in installation. If Microsoft .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 correct .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 the Excel Add-in, complete the following steps:

1. Launch the Excel plug-in installation program by double-clicking the following file:
   
   `aldsp_excel_addin_300_win32.exe`

   You can find this file in the following directory:

   `<bea_home>/aldsp_3.0/excel-addin`

   where `<bea_home>` is the location of the BEA installation on your system. The installation program displays the welcome screen shown in Figure 6-1.
2. Click Next to continue.

3. If you do not already have Microsoft .NET Framework 1.1 installed on your system, you will need to install it. You can do this using the Excel Add-in installation dialog.

4. Determine the user of the application. (Typically anyone with access to your system would be able to use the Add-in.)

5. 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`

6. Complete the installation, and optionally launch Excel.
   The ALDSP Excel Add-in adds a new item, My Data, to the Excel main menu.
Accessing Excel Add-in Documentation

After you have completed the installation, you can reference the Excel Add-in documentation, available in the following document:

*AquaLogic Data Excel User Guide.pdf*

This file is, by default, located in the following directory:

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

The documentation includes the following 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 from the Excel MyData menu option (see Figure 6-2).
Preparing to Use the Excel Add-in

Before accessing data service operations using Excel, you must determine the WSDL URL for the web service.

To prepare to use the Excel Add-in, complete the following steps:

1. Create a Web Service Map for the dataspace you want to access using Excel, if one does not already exist.

   To create a Web Service Map, do the following:
   a. Launch the ALDSP Data Services Studio.
   b. Right-click the dataspace in the Project Explorer and choose New → Web Service Map.
   c. Type a file name for the map and click Finish.
   d. Drag-and-drop the services you want to include in the newly-created map.

2. Set the listen address of the WebLogic server to the host name or IP address of the machine hosting the web services.

   The Excel Add-in requires that the host or IP address specified in /definitions/service/port/soap:address/@location in the WSDL file match the host name or IP address in the WSDL URL.

   To set the listen address of the server, do the following:
   a. Using ALDSP Data Services Studio, double-click the server and click the Open WebLogic Server Admin Console link.
   b. Log in to the WebLogic Admin Console and click Servers in the Environment section.
   c. Click the server in the list.
   d. Click the Lock & Edit button.
e. Type the host name or IP address of the machine hosting the web services in the Listen Address field.

f. Click Save.

g. Click the Activate Changes button.

h. Restart the server.

3. Using ALDSP Data Services Studio, right-click the Web Service Map in the Project Explorer and choose Copy WSDL URL.

You can paste the WSDL URL into the Web Services Setup dialog, as explained in the section Accessing the Data Service Using Excel.

Alternatively, you can manually set the host name or IP address of the server in the WSDL URL when specifying the value in Microsoft Excel.

To determine the host name or IP address of the server, do the following:

1. Using ALDSP Data Services Studio, right-click the Web Service Map for the dataspace in the Project Explorer and choose View WSDL. The application displays the WSDL file.

2. Right-click in the WSDL file and choose Properties. A dialog appears showing the WSDL URL, among other information.

   Use the host name or IP address that appears in the WSDL URL when specifying the value in Microsoft Excel.
Accessing the Data Service Using Excel

This section describes how to access a data service operation using Excel.

**Note:** This assumes that you have installed the Excel Add-in on a local copy of Excel. For more information about installing the Excel Add-in, see **Installing the Excel Add-in**.

To access a data service using Excel, complete the following steps:

1. Launch Microsoft Excel and choose My Data \Web Services Setup from the menu, as shown in **Figure 6-2**.

**Figure 6-2 Setting Up a Web Service for the Excel Add-in**

2. Click New in the Web Service Definitions dialog.
3. Specify the WSDL information by completing the following:

   a. Type a name (alias) for the WSDL in the top left field.
      The example in Figure 6-3 uses retailWS as the name.

   b. Paste the WSDL URL in the top right field.
      This is the WSDL URL that you copied by right-clicking Web Service Map in the
      ALDSP Data Services Studio Project Explorer and choosing Copy WSDL URL, as
      described in Preparing to Use the Excel Add-in.

   c. Click OK.

**Figure 6-3 Entering a WSDL Location and an Alias Name for the New Web Service**
4. Double-click the alias name of the new service to open the operation editor, as shown in Figure 6-4.

**Figure 6-4 Web Service Operation Editor in Excel Add-in**

5. Choose the operation using the Operation drop-down menu.
   In the example shown in Figure 6-4, the parameterized `getCustomerByCustID` operation (which is also the name of the underlying data service function) is selected.

6. Click the Set Input tab.

7. Expand the operation by clicking the + symbol to the left of the entry.
8. Select the appropriate input parameter.
   In the example shown in Figure 6-5, the custID input parameter is selected.

**Figure 6-5 Selecting an Input Parameter**

9. Drag the input parameter icon to the spreadsheet. A label and input field appears in the spreadsheet. If you mouse over the input field, the full parameter path is displayed.

10. Move the cursor to another field in the spreadsheet.

11. In the Excel Add-in Web service dialog, click the Set Output tab.
12. Expand the layers of the operation by clicking the + symbol to display the data elements that the operation can retrieve, as shown in Figure 6-6.

**Figure 6-6 Selecting a Field for Display in Excel**

13. Drag the required output elements to the spreadsheet.

14. In the spreadsheet, type a valid value in the input cell.
   
   For example, you could type CUSTOMER2 as the customer ID value.

15. Press Enter.

16. Choose My Data → Refresh Web Service Data → <web_service> from the menu (or right-click on any field in the spreadsheet to access the option).
17. View and optionally reformat or rearrange the resulting information

Figure 6-7 shows an example of formatted results.

Figure 6-7 Formatted Results
CHAPTER 7

Accessing Data Services Through a WebLogic Workshop Control

This chapter describes how you can use ALDSP Controls in applications generated by BEA Workshop for WebLogic Platform. The following topics are included:

- Introduction to Data Service Controls
- Creating Data Service Controls
- Modifying Existing ALDSP Control
- Caching Considerations When Using Data Service Controls
- ALDSP Control Security Considerations
Introduction to Data Service Controls

An ALDSP Control enables Workshop for WebLogic Platform applications to easily access data services. When you use an ALDSP 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 2, “Data Programming Model and Update Framework.”

In addition to the functionality discussed in this chapter, ALDSP Controls 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 9, “Advanced Topics.”

Data Service Controls Defined

An ALDSP Control provides access to ALDSP dataspaces within Workshop for WebLogic Platform. It is a wizard-generated Java file that can be used to access data service operations. These operations can be added to an ALDSP Control from data services deployed on any accessible WebLogic Server, both local or remote. The ALDSP Control wizard retrieves all available data service operations on the server. It then lets you choose the ones to include in your control.

**Note:** All client APIs, including ALDSP Control, support calling data service functions without 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 using a Data Service control itself.

Like Java controls, you can use an ALDSP Control in Workshop for WebLogic Platform 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 an ALDSP Control in a web service project application. The steps for using it in other Workshop for WebLogic Platform projects are similar.
Description of the Data Service Control File

In the 3.0 release, ALDSP Control is available with both Workshop for WebLogic Platform 9.2 and Workshop Studio 10.1. However, ALDSP Control cannot be used in the Eclipse WTP environment packaged with ALDSP 3.0.

When you create an ALDSP Control, Workshop for WebLogic Platform generates a Java Control file (.java) that contains methods based on the data services 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 the Java web services displays a graphical view of the data service methods that were selected for inclusion in the control.

Figure 7-1 Design View of a Web Service with ALDSP Control Methods
Source View

The Source View tab shows the source code of the ALDSP 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:

```java
locator = "1{ld:ADDRESS}updateADDRESS", functionKind = "procedure"
```

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 an ALDSP Control reside in the Libraries folder of the project. An interface is generated for each data service.

The only time you should need to manually edit the source code is if you want to add a method to run an ad hoc query, as described in Using ALDSP Control for Ad Hoc Queries.

Listing 7-1 shows portions of a generated ALDSP Control file. It shows the package declaration, import statements, and data service URI used with the queries.

Listing 7-1   ALDSP Control File Sample

```java
package xyzpackage;

import org.apache.beehive.controls.api.bean.ControlExtension;
import com.bea.dsp.control.core.DSPControl;
import com.bea.dsp.control.core.annotation.DSPControlInterfaceAnnotation;
import com.bea.dsp.control.core.annotation.ALDSPControlMethodAnnotation;
import com.bea.dsp.RequestConfig;
import commonj.sdo.DataObject;
import com.bea.sdo.impl.data.DataObjectGeneral;

@ControlExtension
@DSPControlInterfaceAnnotation(version = "3.0", application = "DS2",
urlKey = "DS2.xyzpackage.xyz_DSPControlFile", url = "t3://localhost:7001",
username = "weblogic")
public interface xyz_DSPControlFile extends DSPControl {

@ALDSPControlMethodAnnotation(functionURI = "ld:ADDRESS", functionName = "ADDRESS", schemaURI = "ld:ADDRESS", schemaRootElement = "ADDRESS",
locator = "0{ld:ADDRESS}ADDRESS", functionKind = "function")
```

public address.ADDRESS[] ADDRESS() ;

@ALDSPControlMethodAnnotation(functionURI = "ld:ADDRESS", functionName = "createADDRESS", schemaURI = "ld:ADDRESS", schemaRootElement = "ADDRESS_KEY", locator = "1{ld:ADDRESS}createADDRESS", functionKind = "procedure")
public address.ADDRESS_KEY[] createADDRESS(address.ADDRESS[] p) ;

@ALDSPControlMethodAnnotation(functionURI = "ld:ADDRESS", functionName = "updateADDRESS", schemaURI = "", schemaRootElement = "", locator = "1{ld:ADDRESS}updateADDRESS", functionKind = "procedure")
public void updateADDRESS(address.ADDRESS[] p) ;

@ALDSPControlMethodAnnotation(functionURI = "ld:ADDRESS", functionName = "deleteADDRESS", schemaURI = "", schemaRootElement = "", locator = "1{ld:ADDRESS}deleteADDRESS", functionKind = "procedure")
public void deleteADDRESS(address.ADDRESS[] p) ;

@ALDSPControlMethodAnnotation(functionURI = "ld:ADDRESS", functionName = "getCREDIT_CARD", schemaURI = "ld:CREDIT_CARD", schemaRootElement = "CREDIT_CARD", locator = "1{ld:ADDRESS}getCREDIT_CARD", functionKind = "function")
public credit_card.CREDIT_CARD[] getCREDIT_CARD(address.ADDRESS pk) ;

@ALDSPControlMethodAnnotation(functionURI = "ld:ADDRESS", functionName = "getCUSTOMER", schemaURI = "ld:CUSTOMER", schemaRootElement = "CUSTOMER", locator = "1{ld:ADDRESS}getCUSTOMER", functionKind = "function")
public customer.CUSTOMER[] getCUSTOMER(address.ADDRESS fk) ;

/**
 * Using RequestConfig:
 * In order to use a RequestConfig parameter with functions,
 * procedures and submit methods
 * please edit a method signature as follows:
 * From:
 * Customer getCustomer(java.lang.String p0);
* To:
* Customer getCustomer(java.lang.String p0, RequestConfig
* config);
* The RequestConfig parameter allows you to send FilterXQuery and
* QueryAttributes
* to the server and receive Audit Event. /**/

Note: In the ALDSP 3.0 release, the read operations do not contain the control level flag for
txRequired and txSupported. This flag is available in RequestConfig and is applied on a
per-call basis.

Creating Data Service Controls

You can create an ALDSP Control in Workshop Studio 10.1 or in Workshop for WebLogic
Platform 9.2 environment. The steps to create the control in either of these environments are
almost the same. However, some of the options that you select while creating the control may
differ based on your environment.

This section describes the steps involved in creating an ALDSP Control using the Workshop for
WebLogic Platform 9.2 and using it in a Web Service project. In addition, it provides information
about the options that you need to configure differently if you create the control in Workshop
Studio 10.1.

The general steps to create an ALDSP Control are:

- Prerequisites
- Step 1: Create a Project
- Step 2: Start WebLogic Server
- Step 3: Create a Package under src Folder
- Step 4: Create the Data Service Control

The following sections describe each of these steps in detail.
Prerequisites

Ensure that your Workshop for WebLogic Platform 9.2 is using JDK 1.5.0_07 or higher.

Note: For ALDSP 3.0 and, in some cases, ALDSP 3.2, it is necessary to copy an ALDSP Control configuration file to the Workshop installation. Details can be seen in the Client Configuration Section ("Configuration Instructions for WebLogic Workshop in Conjunction with ALDSP Control") of Release Notes.

Note: If you want to use ALDSP Control with WLI, also see the Client Configuration section of ALDSP 3.0/3.2 Release Notes.

Step 1: Create a Project

Before you can create an ALDSP Control in Workshop for WebLogic Platform, you must create a web-enabled project such as portal or Web Service Project. A Web Service Project allows you to create Java web service projects while you can use a dynamic web project to create portal components like Java page flows and JSPs.

Step 2: Start WebLogic Server

Make sure that the WebLogic Server that hosts the ALDSP dataspace project is running. WebLogic Server can be running locally (on the same domain as Workshop for WebLogic Platform) or remotely (on a different domain from Workshop for WebLogic Platform).

Step 3: Create a Package under src Folder

In the web-enabled project or Web Service Project, an src folder is created. Create a package under this folder as follows:

1. Right-click the src folder and select New > Package.
2. Enter the package name in the New Java Package dialog box.
3. Click Finish.

Step 4: Create the Data Service Control

To create an ALDSP Control:

1. Right-click the package you created previously, and select New > Other > Controls > ALDSP Control.
2. Click Next. This starts the ALDSP Control creation wizard.
3. In the New DSP Control dialog box, select the project and package under which you need to create the control, as shown in Figure 7-2.

**Figure 7-2 New DSP Control Dialog Box**

![New DSP Control Creation dialog box](image)

**Note:** An ALDSP Control can be created only in a project that is enabled with Beehive facets because the Beehive runtime is required for the ALDSP Control runtime. If you select a container that is not Beehive-enabled, you will not be able to proceed.
Creating Data Service Controls

To check for the Beehive Control facets:

a. Right-click your web service project and select Properties.

b. From the Properties dialog box, select Project Facets. This displays the version number of the Beehive Control.

4. Enter the name of the control in the File Name box and click Next.

5. In the Select Control Attributes dialog box, you need to specify the location of the ALDSP dataspace project that the control needs to consume, as shown in Figure 7-3.

Note: A control can only invoke data service functions from a single dataspace project.

Figure 7-3 Select Control Attributes Dialog Box

Note: You can choose to generate an ALDSP Control from dataspace projects in your local workspace instead of a deployed dataspace. However this option is available only if your Workshop platform can support ALDSP, otherwise it is disabled. This option will be enabled in a future release of ALDSP.
Select the From server option and then:

a. Choose a local or remote server.

b. Specify the user name and password to authenticate connection to the WebLogic Server where the dataspace is deployed.

c. Click Get Dataspace projects. This shows a drop-down list of available dataspace projects at the selected location.

6. Click Next to move to the Method selection Page as shown in Figure 7-4.

**Figure 7-4 Method Selection Page**

![Image of Method Selection Page](image)

7. Select the methods that you want to include in ALDSP Control and click Add > Finish. This completes the steps to create a new ALDSP Control.

**Note:** An ALDSP Control can access only public data service functions.

8. Create a WebLogic Web Service:

   a. Right-click the package.

   b. Select New > WebLogic Web Service.

9. Drag-and-drop the ALDSP Control file to the Design View of the web service as shown in Figure 7-5.

![Image of Design View](image)
After creating the control, the following files are created:

- **DSP_Control.jar**: This JAR contains the required runtime components and is available under the WebApp Libraries folder.

- **Schemas**: Required schemas are downloaded to a temporary location, then built to SDO classes, made into JAR and moved into the library location. These JAR files have the following naming convention:

  `aldspschemas_controlctrl_xsd_full_package_name.jar`

  **Note**: ALDSP generates interface files for the target schemas associated with the queries and the ALDSP Control file.
The location of both these JAR files is shown in Figure 7-6.

**Figure 7-6 Location of DSP_Control.jar**

Using ALDSP Control 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.

An ALDSP Control generated from a wizard contains a commented ad hoc query method template that can serve as a starting point for creating an ad hoc query. To create the ad hoc query:

1. Generate an ALDSP Control file using the ALDSP Control wizard, if you do not already have it.

2. Add the following lines of code in the ALDSP Control file:

   ```java
   com.bea.xml.Object executeQuery(String query);
   ```
Replace the function name with a meaningful name for your application. Be default, the ad hoc query returns a typed SDO or an object that matches the return type for the ad hoc query. You can also optionally supply ExternalVariables or QueryAttributes (or both) to an ad hoc query.

**Note:** ExternalVariable is used for binding variable name to value for use in adhoc query in ALDSP Control. In ALDSP Control, an xquery can be passed in as a string. If this query has declared external variables, then the value for these variables are passed in using ExternalVariable.

When invoking this ad hoc query function from an ALDSP Control, the caller needs to pass the query string (and the optional ExternalVariables binding and QueryAttributes). For example, an ad hoc query signature in a ALDSP Control will look like the following:

```java
@DSPControlAdhocQueryAnnotation(body="")
*     Object[] executeQuery(String xquery, ExternalVariables params);
}
```

The code to call this ALDSP Control (from a web service JWS file, for example) would be:

```java
public void adHocAddressQuery()
{
    String adhocQuery =
    "declare namespace f1 = "ld:ldc_producerDataServices/ADDRESS";\n" +
    "declare namespace ns0="ld:ldc_producerDataServices/ADDRESS";\n"+
    "<ns0:ArrayOfADDRESS>\n" +
    "{for $i in f1:ADDRESS()\n" +
    "where $i/STATE = "TX"\n" +
    "return $i}\n" +
    "</ns0:ArrayOfADDRESS>\n";
    Object result = myldcontrol.adHocAddressQuery(adhocQuery);
}
```

## ALDSP Control Backward Compatibility

In the 3.0 release, two types of backward compatibility is provided for ALDSP Control:

- **8.x** Control deployed on 8.x WebLogic Server and connected to ALDSP 3.0

  **Note:** In this case, the data interchange is in the form of XmlBeans.

- **8.x** Control that has been upgraded to support the ALDSP 3.0 server
Note: If you using ALDSP 2.5 or any earlier version, then you need to upgrade ALDSP Control using the source upgrade feature. For more information on upgrading ALDSP Control, refer to “ALDSP Control Source Upgrade” on page 9-1.

Modifying Existing ALDSP Control

This section describes how you can modify an existing ALDSP 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 also.

This section contains the following topics:

- Adding and Removing Operations Used by a Control
- Updating an Existing Control When Schemas Change

Adding and Removing Operations Used by a Control

To change a data service function in an ALDSP Control:

1. In Workshop for WebLogic Platform, open the ALDSP Control file that you need to edit. This opens up the ALDSP Control .java file in Source View.
2. Right-click the file in Source View.
3. Select the Edit ALDSP Control > Edit ALDSP Control option. This displays the Edit DSP Control dialog box as shown in Figure 7-7.
4. Enter the password of the ALDSP-enabled WebLogic Server where your project is deployed and click Next.

5. From the Method Selection Page dialog box, select the functions that you want to add or remove from the control, as shown in Figure 7-8.
6. Click Finish to complete the process.

7. Rebuild the project after the control is edited.

**Updating an Existing Control When Schemas Change**

If any of the schemas corresponding to any operation in an ALDSP Control change, you need to edit the control by removing the operation and restoring it.

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

**Note:** For details on working with static and dynamic SDO see “Mediator API Basics” on page 3-39.
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 from 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 ALDSP 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 ALDSP Control file. This function will take a QueryAttribute object as a parameter.
- Instantiate a QueryAttribute object in your project and call the enableFeature() method, passing the GET_CURRENT_DATA attribute.
- Call the function you defined in your ALDSP 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.
Accessing Data Services Through a WebLogic Workshop Control

Listing 7-2  Cache Bypass Example When Using Data Services Platform Control

```java
if (refreshCache == false) {
    address.ADDRESS[] addresses = control.ADDRESS();
} else {
    QueryAttributes attr = new QueryAttributes();
    attr.enableFeature(QueryAttributes.GET_CURRENT_DATA);
    address.ADDRESS[] addresses = control.ADDRESS(attr);
}
```

An additional function is also needed in the ALDSP Control file. For the code shown in Listing 7-2, you need to add the following definition to the control:

```java
@ALDSPControlMethodAnnotation(functionURI = "ld:ADDRESS", functionName = "ADDRESS", schemaURI = "ld:ADDRESS", schemaRootElement = "ADDRESS", locator = "0{ld:ADDRESS}ADDRESS", functionKind = "function")
public address.ADDRESS[] ADDRESS(QueryAttributes attr) ;
```

ALDSP Control Security Considerations

This section describes security considerations for applications using an ALDSP Control. The following sections are included:

- Security Credentials Used to Create Data Service Controls
- Testing Controls in the JWS File
- Trusted Domains
- Using ALDSP Control in Different Domains
Security Credentials Used to Create Data Service Controls

When you create an ALDSP Control and connect to a local or remote Data Services Platform server (a WebLogic Server on a different domain from Workshop for WebLogic Platform), you specify the connection information in the ALDSP Control wizard as shown in Figure 7-3.

When you create an ALDSP Control, the ALDSP 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.

Testing Controls in the JWS File

To test an ALDSP Control:

1. Right-click the Web Service in Design View.
2. Select Run As > Run on Server. The Run on Server dialog box is displayed.
3. Select the WebLogic Server instance where your dataspace project is deployed and click Next.
4. Click Finish. This generates a WebLogic Test Client where you can test the control as shown in Figure 7-9.

Figure 7-9 WebLogic Test Client for ALDSP Control

When a query is run from a dataspace, it must have a mechanism for getting the security credential. The credential can come from a login screen, it can be hard-coded in the dataspace, or it can be imbedded in a J2EE component (for example, using the run-as property in a JWS Web service file).
Note: You can enable the auditing if you want to monitor or debug ALDSP Control in a test environment. For more information about auditing, refer to the Working With Audit and Log Information chapter in the Administration Guide.

Trusted Domains

If the WebLogic Server that hosts the ALDSP project is on a different domain than Workshop for WebLogic Platform, 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.

For more details on WebLogic Server security, see:

- “Configuring Security for a WebLogic Domain” in the WebLogic Server documentation.

For information on security, see:

- "Securing ALDSP Resources" in the Administration Guide

Using ALDSP Control in Different Domains

If you deploy your application on a server, which is the source for ALDSP Control between different domains, then you must create a dspControlConfig.properties file in the domain directory of the cluster where ALDSP Control is deployed.

This property file should contain a key-value mapping where the key is the string representing the urlKey in the control file and the value is the T3 URL at which ALDSP server is found.

For example, if your ALDSP Control file contains urlKey = "ValidSample.com.LocalCtrl", then your property file should contain ValidSample.com.LocalCtrl=t3://<my-qa-prod-dev-url>.

Ensure that for each ALDSP Control that you deploy, there is a corresponding dspControlConfig.properties file entry.
Supporting ADO.NET Clients

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

Information about how ALDSP achieves ADO.NET integration is provided in this chapter, along with the server-side operations required to enable it. The chapter includes the following sections:

- Overview of ADO.NET Integration in Data Services Platform
- Enabling ALDSP Support for ADO.NET Clients
- Adapting ALDSP XML Types (Schemas) for ADO.NET Clients
- Creating a Data Service Based on an RPC-Style Web Service
- Generated Artifacts Reference

Note: The details of ADO.NET development are described on Microsoft’s MSDN Web site (http://msdn.microsoft.com). See this site for information about developing ADO.NET-enabled applications.
Overview of ADO.NET Integration in Data Services Platform

Functionally similar to the service data object (SDO), ADO.NET (Active Data Object) 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 it 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 as SDOs are used by ALDSP client applications.

See “Role of the Mediator API and SDO” on page 2-13 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 the 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 the following steps:

- Generating an ALDSP Web Services Mapper
- Creating a Web Reference in ADO.NET Client by Providing the ALDSP WSDL URL

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.

**Note:** Although ADO.NET supports both connected (direct) and disconnected models, only the disconnected model is supported in the Data Services Platform.
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 8-1 ADO.NET Clients Supported via Web Services

Once the WSDL URL is available, your client can invoke data service operations and 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 an ALDSSP Web Services Mapper” on page 8-8.
Understanding How ALDSP Supports ADO.NET Clients

BEA AquaLogic Data Services Platform supports ADO.NET at the data object level. That is, the 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 8-2  ADO.NET and SDO Data Objects Compared

<table>
<thead>
<tr>
<th>ADO.NET</th>
<th>SDO</th>
<th>Microsoft .NET Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataSet</td>
<td>DataObject</td>
<td>Disconnected data models. Queries return results conforming to this data model.</td>
</tr>
<tr>
<td>DiffGram</td>
<td>ChangeSummary</td>
<td>Mechanisms for tracking changes made to data objects by a client application.</td>
</tr>
</tbody>
</table>

As shown in Figure 8-3, the ADO.NET typed DataSet is submitted to and returned by ALDSP. 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, converts the .NET Dataset to an SDO Data Object, and passes it to Data Services.
Supporting ADO.NET Clients

Figure 8-3 Data Services Platform and .NET Integration
Mapping, transformation, and packaging processes are transparent to client application developers and data services developers. Only the items listed in Table 8-4 are exposed to data service developers.

Table 8-4 Data Services Platform—Java and ADO.NET-Enabled Artifacts

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Service</td>
<td>Customer.ds</td>
<td>An XQuery file that instantiates operations such as read functions, navigation functions, procedures, and update functionality at runtime.</td>
</tr>
<tr>
<td>Data Service Schema</td>
<td>Customer.xsd</td>
<td>The schema associated with the XML type of the original data service.</td>
</tr>
<tr>
<td>DataSet Schema</td>
<td>CustomerDataSet.xsd</td>
<td>The typed DataSet schema that conforms to Microsoft requirements for ADO.NET data objects. Note that dataset xsd is not physically generated into a dataspace project. It is dynamically generated at WSDL generation time when WSDL and its imported schema files are accessed in .NET client.</td>
</tr>
<tr>
<td>Web Service Map</td>
<td>CustomerNET.ws</td>
<td>Web Services mapper file that maps data service operations to web service operations.</td>
</tr>
</tbody>
</table>

Supporting Java Clients

The WSDL generated by ALDSP from an ADO.NET-enabled web services map 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.
Enabling ALDSP Support for ADO.NET Clients

The process of providing ADO.NET clients with access to data services is a server-side operation that is initially enabled in Eclipse and takes place in the context of an application and the ALDSP Data Services Studio.

The instructions in this section assume that you have created a dataspace project and that you want to provide an ADO.NET client application with access to data services. (For information about designing and developing data services, see the *Data Services Developer’s Guide*.)

Enabling an ALDSP application to support ADO.NET clients involves doing the following:

- Generating an ALDSP Web Services Mapper
- Creating a Web Reference in ADO.NET Client by Providing the ALDSP WSDL URL

In some cases, of course, there will already be existing operations that you want to make available to an ADO.NET client.

Generating an ALDSP Web Services Mapper

You need to generate an ALDSP Web Services Mapper that maps data service functions to Web Service operations.

To generate a Web Services Mapper, do the following:

1. Right-click the project folder and select New > Web Service Mapper. Enter a filename for your .WS map file.

   You can then drag and drop data service files and functions from the Project Explorer into your mapper.
2. Select Windows > Show View > Properties to display the properties for the Web Service Mapper, and the operations that you put into your Web Service mapper.
Supporting ADO.NET Clients

3. Click on the Mapper Bar and on the value across from ADO-NET-enabled. Then, select true for the ADO.NET-enabled option.

![Configuration Property Sheet]

4. Redeploy your project by right-clicking on the project name and checking Deploy Project.

**Viewing an ADO.NET-Enabled WSDL**

The system automatically generates a Web Services Description Language (WSDL) file that can be used by Web service clients to invoke operations on the ADO.NET-enabled Web service:

1. Right-click on the Web Service Mapper file created in “Generating an ALDSP Web Services Mapper.”
2. Select View WSDL.
Figure 8-6 Generated WSDL in AquaLogic Data Services Studio

- See “Web Services Description Language (WSDL) File for Microsoft ADO.NET Clients” on page 8-19 for information about the format of the WSDL.

**Note:** The building of RPC-style Web services on top of ALDSP is not supported. For this reason, RPC-style Web services built on cannot be created from ADO.NET clients utilizing ALDSP.
Creating a Web Reference in ADO.NET Client by Providing the ALDSP WSDL URL

From your ADO.NET client application, specify the path to locate the WSDL. The example uses VisualStudio as a client application.

1. Right-click on Web Reference to import the WSDL.
2. Add “Web Ref.”
3. In the window, enter the URL for the WSDL.
   
   http://host:port/dataspace project name/ folder/.../mapperfile.ws?WSDL

   For example, the web services mapper file created in Data Services Studio:
   
   http://localhost:7001/NewProject/TestMapper.ws?WSDL

4. Click Go.

Once you have imported the WSDL you will be able to execute its data service operations assuming that the ALDSP-enabled server is running and your application has sufficient access privileges.

Adapting ALDSP 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 ALDSP server, specifically explaining how to:

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

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

### Approaches to Adapting XML Types for ADO.NET

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

- Develop ADO.NET-compatible data services above the physical data service layer. You can develop data services on top of physical data sources that are specifically intended to be consumed by ADO.NET clients. (Details are described in “XML Type Requirements for Working With ADO.NET DataSets” on page 8-14.)
  
  **Note:** Any ADO.NET-compatible data service XML types also can be used by non-ADO.NET clients.

- Develop ADO.NET-compatible data services above a logical data service layer. If existing logical data services that are not ADO.NET-compatible must be reused, you can build an additional layer of ADO.NET-compatible data services on top of the logical data services.
  
  **Note:** This approach may increase the likelihood of having to work with inverse functions and custom updates.
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:

- Define 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 the following:

- 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 referred to 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 example:

```xml
<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 the following:

- 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 ALDSP update override function to handle the update.)
- 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 the following site:

http://www.w3.org/TR/xmlschema-0
Creating a Data Service Based on an RPC-Style Web Service

For RPC-style web services, results are return as qualified or unqualified based on the setting of the schema attribute:

```xml
<elementFormDefault
```

In general, for web services, you can override the `elementFormDefault` by setting the form attribute for any child element. However, these individual settings are ignored for RPC-style web services since only the global setting (qualified or unqualified) is taken into account.

For example:

```xml
<s:schema elementFormDefault="qualified"
    namespace="http://temp.openuri.org/SampleApp/CustomerOrder.xsd"
    xmlns:s0="http://temp.openuri.org/SampleApp/CustomerOrder.xsd"
    xmlns:s="http://www.w3.org/2001/XMLSchema">
    <s:complexType name="ORDER">
        <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" form="unqualified"
                name="ORDER_ID" type="s:string"/>
            <s:element minOccurs="0" maxOccurs="1" form="unqualified"
                name="CUSTOMER_ID" type="s:string"/>
        </s:sequence>
    </s:complexType>
</s:schema>
```

In this code sample, the global element is qualified but a child element (`ORDER_ID`) is unqualified.

In the standard case, the special setting of unqualified for `ORDER_ID` is honored. In the case of RPC-style web services, however, the runtime generates qualified attributes for all of the elements, including `ORDER_ID`.

**Note:** RPC-style web services such as those generated by ADO.NET may contain child elements with `form` attributes which do not match the schema's `elementFormDefault` declaration. To turn these web services into executable data service operations, make sure that all form element attributes and the `elementFormDefault` attribute are in agreement (either qualified or unqualified).

Multi-dimensional arrays in RPC mode are not supported.
Generated Artifacts Reference

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

- Typed DataSet Schema file - This file is not located in the dataspace project physically like the web services mapper file. It is dynamically generated on the server and sent to the .NET client when WSDL and its imported xsd are retrieved on the client side

- ADO.NET Enabled Web Services Map File

Technical specifications for these artifacts are included in this section.

XML Schema Definition for ADO.NET Types DataSet

The Typed DataSet schema file is referred to in the dynamically-generated WSDL. The schema file is retrieved by the .NET client dynamically during web reference creation.

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 upon 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 ALDSP data service)

Listing 8-1 shows an excerpt of a schema—CustomerDS.xsd—for a typed DataSet generated from an ALDSP Customer schema.
Listing 8-1  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="CustomerDataSet">
        <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 WSDL generated from the mapper 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 the following 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 follows:

```
<s:complexType name="CustomerDataSetWrapper">
    <s:sequence>
        <s:any namespace="http://temp.openuri.org/schemas/Customer.xsd"/>
    </s:sequence>
</s:complexType>
```
Below is a sample CUSTOMER_VIEW DataSet.xsd file:

```xml
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
    xmlns:tns="ld:logicalDS/CUSTOMER_VIEW"
    targetNamespace="ld:logicalDS/CUSTOMER_VIEW"
    id="CUSTOMER_VIEWDataSet" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element msdata:IsDataSet="true"
    name="CUSTOMER_VIEWDataSet">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element ref="tns:CUSTOMER_VIEW" />
        <xs:choice>
          <xs:complexType>
            <xs:element name="CUSTOMER_VIEW">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="CUSTOMER_ID" type="xs:string" />
                  <xs:element name="FIRST_NAME" type="xs:string" />
                  <xs:element name="LAST_NAME" type="xs:string" />
                  <xs:element name="CUSTOMER_SINCE" type="xs:dateTime" />
                  <xs:element name="EMAIL_ADDRESS" type="xs:string" />
                  <xs:element minOccurs="0" name="TELEPHONE_NUMBER" type="xs:string" />
                  <xs:element minOccurs="0" name="SSN" type="xs:string" />
                  <xs:element minOccurs="0" name="BIRTH_DAY" type="xs:dateTime" />
                  <xs:element minOccurs="0" name="DEFAULT_SHIP_METHOD" type="xs:string" />
                  <xs:element minOccurs="0" name="EMAIL_NOTIFICATION" type="xs:integer" />
                  <xs:element minOccurs="0" name="NEWS_LETTER" type="xs:integer" />
                  <xs:element minOccurs="0" name="ONLINE_STATEMENT" type="xs:integer" />
                  <xs:element minOccurs="0" name="CREDIT_LIMIT" type="xs:decimal" />
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:complexType>
        </xs:choice>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
<xs:element name="ORDERS">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="ORDER">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="CUSTOMER_ID" type="xs:string" />
            <xs:element name="ORDER_ID" type="xs:string" />
            <xs:element name="ORDER_DATE" type="xs:dateTime" />
            <xs:element name="SHIP_METHOD" type="xs:string" />
            <xs:element name="HANDLING_CHARGE" type="xs:decimal" />
            <xs:element name="SUBTOTAL" type="xs:decimal" />
            <xs:element name="TOTAL_ORDER_AMOUNT" type="xs:decimal" />
            <xs:element name="SHIP_TO" type="xs:string" />
            <xs:element name="SHIP_TO_NAME" type="xs:string" />
            <xs:element name="BILL_TO" type="xs:string" />
            <xs:element name="ESTIMATED_SHIP_DATE" type="xs:dateTime" />
            <xs:element name="STATUS" type="xs:string" />
            <xs:element minOccurs="0" name="TRACKING_NUMBER" type="xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
Below is a sample CUSTOMER_VIEW_Net WSDL file:

```xml
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:tns="ld:LogicalDSs/Customer_view_net.ws"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="ld:LogicalDSs/Customer_view_net.ws"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
    <documentation>AquaLogic Data Services Web Service</documentation>
    <import namespace="ld:logicalDS/CUSTOMER_VIEW"
        location="DSP_DOT_NET_SCHEMA/LogicalDSs/schemas/CUSTOMER_VIEW"
        /CUSTOMER_VIEW DataSet.xsd/>
    <types>
        <xs:schema xmlns:stns="ld:logicalDS/customer_view_net.ws"
            xmlns:dsns0="ld:logicalDS/CUSTOMER_VIEW"
            elementFormDefault="qualified"
            targetNamespace="ld:logicalDSs/customer_view_net.ws">
            <xs:element name="getFirst">
                <xs:complexType>
                    <xs:sequence/>
                </xs:complexType>
            </xs:element>
            <xs:element name="getFirstResponse">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element minOccurs="0" name="getFirstResult"
                            type="stns:CUSTOMER_VIEWDataSetWrapper"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="createCUSTOMER_VIEW">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element minOccurs="0" maxOccurs="unbounded"
                            ref="dsns0:CUSTOMER_VIEW"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:schema>
    </types>
</definitions>
```
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="createCUSTOMER_VIEWResponse">
    <xs:complexType>
        <xs:sequence />
    </xs:complexType>
</xs:element>
<xs:element name="updateCUSTOMER_VIEW">
    <xs:complexType>
        <xs:sequence>
            <xs:element minOccurs="0" name="p" type="stns:CUSTOMER_VIEWDataSetWrapper" />
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="updateCUSTOMER_VIEWResponse">
    <xs:complexType>
        <xs:sequence />
    </xs:complexType>
</xs:element>
<xs:element name="deleteCUSTOMER_VIEW">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="p">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element minOccurs="0" maxOccurs="unbounded" ref="dsns0:CUSTOMER_VIEW" />
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
Supporting ADO.NET Clients

<xs:element name="deleteCUSTOMER_VIEWResponse">
<xs:complexType>
<xs:sequence />
</xs:complexType>
</xs:element>

<xs:element name="getAll">
<xs:complexType>
<xs:sequence />
</xs:complexType>
</xs:element>

<xs:element name="getAllResponse">
<xs:complexType>
<xs:sequence>
<xs:element minOccurs="0" name="getAllResult" type="stns:CUSTOMER_VIEWDataSetWrapper">
<xs:sequence>
</xs:complexType>
</xs:element>
</xs:complexType>
</xs:element>

<xs:complexType name="CUSTOMER_VIEWDataSetWrapper">
<xs:sequence>
<xs:any namespace="ld:logicalDS/CUSTOMER_VIEW" /> 
</xs:sequence>
</xs:complexType>
</xs:schema>

<message name="getAllIn">
<part name="parameters" element="tns:getAll" />
</message>

<message name="getAllOut">
<part name="parameters" element="tns:getAllResponse" />
</message>

<message name="getFirstIn">
<part name="parameters" element="tns:getFirst" />
</message>

<message name="getFirstOut">
<part name="parameters" element="tns:getFirstResponse" />
</message>

<message name="createCUSTOMER_VIEWIn"
<portType name="Customer_view_netPT">
  <operation name="getAll">
    <input message="tns:getAllIn" />
    <output message="tns:getAllOut" />
  </operation>
  <operation name="getFirst">
    <input message="tns:getFirstIn" />
    <output message="tns:getFirstOut" />
  </operation>
  <operation name="createCUSTOMER_VIEW">
    <input message="tns:createCUSTOMER_VIEWIn" />
    <output message="tns:createCUSTOMER_VIEWOut" />
  </operation>
  <operation name="updateCUSTOMER_VIEW">
    <input message="tns:updateCUSTOMER_VIEWIn" />
    <output message="tns:updateCUSTOMER_VIEWOut" />
  </operation>
  <operation name="deleteCUSTOMER_VIEW">
    <input message="tns:deleteCUSTOMER_VIEWIn" />
    <output message="tns:deleteCUSTOMER_VIEWOut" />
  </operation>
</portType>
Supporting ADO.NET Clients

</operation>
</portType>

<binding name="Customer_view_netSoapBinding"
type="tns:Customer_view_netPT">
  <soap:binding
    transport="http://schemas.xmlsoap.org/soap/http" />
  <operation name="getAll">
    <soap:operation
      soapAction="ld:LogicalDSs/Customer_view_net.ws/getAll"
      style="document" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
  <operation name="getFirst">
    <soap:operation
      soapAction="ld:LogicalDSs/Customer_view_net.ws/getFirst"
      style="document" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
  <operation name="createCUSTOMER_VIEW">
    <soap:operation
      soapAction="ld:LogicalDSs/Customer_view_net.ws/createCUSTOMER_VIEW"
      style="document" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
</binding>
<operation>
  <operation name="updateCUSTOMER_VIEW">
    <soap:operation
      soapAction="ld:LogicalDSs/Customer_view_net.ws/updateCUSTOMER_VIEW"
      style="document" />
  </operation>
</operation>

<operation>
  <operation name="deleteCUSTOMER_VIEW">
    <soap:operation
      soapAction="ld:LogicalDSs/Customer_view_net.ws/deleteCUSTOMER_VIEW"
      style="document" />
  </operation>
</operation>

</binding>

</service>
</definitions>
Supporting ADO.NET Clients
Advanced Topics

This chapter describes miscellaneous features that are related to client programming with BEA AquaLogic Data Services Platform. It includes the following topics:

- ALDSP Control Source Upgrade
- Accessing Metadata Using Catalog Services
- Filtering, Sorting, and Fine-tuning Query Results

ALDSP Control Source Upgrade

You need to upgrade ALDSP 8.x controls to 9.x using source upgrade. After the upgrade, a version of ALDSP Control.jar is placed in the WEB-INF/lib folder. The Control.jar file contains control runtime classes.

While performing the upgrade, ensure that the domain to which the Web-App is to be deployed includes the following files:

- wldso.jar file under <ALDSP_HOME>/lib/
- BEA V2 XmlBeans, <ALDSP>/lib/sdo.jar in its server classpath or Web-App library path

Source upgrade also upgrades the schemas used by ALDSP Control.

For more information on upgrading ALDSP Control, refer to Upgrading an ALDSP Control from 2.5 to 3.0 section in the Installation Guide.
Accessing Metadata Using Catalog Services

BEA AquaLogic Data Services Platform maintains metadata about data services, application, functions, and schemas through Catalog Services, which is a system catalog-type data service. Catalog services provide a convenient way for client-application developers to programmatically obtain information about Data Services Platform applications, data services, schemas, functions, and relationships.

Catalog Services are also data services; you can view them using the DSP Console, the ALDSP Palette, and ALDSP Controls.

Some advantages of using Catalog Services are as follows:

- Client application developers can use the Catalog Services in the same way as they use any other data service in ALDSP.
- Application developers can create dynamic applications based on the metadata underlying the data service applications that have been deployed.
- For enterprise, third-party, and other developers, Catalog Services leverage the development of dynamic, metadata driven, query-by-form (QBF) applications.
- Catalog Services enable interoperability with other metadata repositories.

This section provides details about installing and using Catalog Services to access metadata for any Data Services Platform application. It includes the following topics:

- Installing Catalog Services
- Using Catalog Services
Installing Catalog Services

You can install Catalog Services as a project for an ALDSP application or as a JAR file that is added to the Library folder in Data Services Studio. The Catalog Services project (_catalogservices) contains data services that provide information about the application, folders, data services, functions, schemas, and relationships available with the application.

DataServiceRef and SchemaRef are additional data services that consist of functions that retrieve the paths to the data services and schemas available with the Data Services Platform application. For more information about the data services and functions available with Catalog Services, refer to “Using Catalog Services” on page 9-4.

To install Catalog Services as a project:

1. Right-click the Data Services Platform application in Data Services Studio.

2. Select the Install Catalog Services (Expanded) option if you want to use the catalog services for development, as shown in Figure 9-1. If you need catalog services only during runtime then select Install Catalog Services (Jar) option.

Figure 9-1 Installing Catalog Services
Advanced Topics

Using Catalog Services

After installing Catalog Services, the catalog services project, _catalogservices, is created for the Data Services Platform application. All the data services associated with catalog services are available under this project. You can invoke the data service functions to access metadata. The client Mediator API is used to invoke the Catalog Service methods.

The data services available under _catalogservices include:

- Application (application.ds)
- DataServiceRef (DataServiceRef.ds)
- Folder (folder.ds)
- Function (Function.ds)
- Relationship (Relationship.ds)
- Schema (Schema.ds)

The following table provides the declaration and description for the functions available in Schema.ds

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getApplication() as schema-element(t1:Application) external;</td>
<td>This function returns the name of the AquaLogic Data Services Platform application. It does not take any parameters.</td>
</tr>
</tbody>
</table>

Note: To use data service functions available with Catalog services, refer to the code samples available at:

http://dev2dev.bea.com/wiki/pub/CodeShare/Sample1/catalogsrv_output.zip

Application (application.ds)

The following table provides the declaration and description for the getApplication() function in Application.ds.

Table 9-2 Functions in Application.ds
**DataService (DataService.ds)**

Table 9-3 provides declaration and description information for the functions available in DataService.ds.

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Description</th>
<th>Sample Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDataServiceRef($arg as element(md:DataService)) as element(md:DataServiceRef) {$arg/md:DataServiceRef}</td>
<td>This function returns the path of the data service associated with the function. For this function, you need to specify the following: • Path of the data service • Path of the schema for the data service • Function ID of the function for which you need the data service reference</td>
<td><code>&lt;urn:DataService kind=&quot;javaFunction&quot; xmlns:acc=&quot;ld:RTLAppDataServices/Custome rDB/Customer&quot; xmlns:urn=&quot;urn:metadata.ld.bea.com&quot;&gt; &lt;urn:DataServiceRef&gt; &lt;id&gt;ld:RTLAppDataServices/CustomerDB/Cus tomer.ds&lt;/id&gt; &lt;/urn:DataServiceRef&gt; &lt;returnType name=&quot;CUSTOMER&quot; kind=&quot;read&quot; quantifier=&quot;*&quot; schemaId=&quot;ld:RTLAppDataServices/Customer DB/schemas/CUSTOMER.xsd&quot;/&gt; &lt;!--Zero or more repetitions:--&gt; &lt;key&gt; &lt;!--1 or more repetitions:--&gt; &lt;path&gt;ld:RTLAppDataServices/CustomerDB/Cus tomer.ds&lt;/path&gt; &lt;/key&gt; &lt;!--Zero or more repetitions:--&gt; &lt;urn:FunctionId name=&quot;CUSTOMER&quot; arity=&quot;0&quot;/&gt; &lt;/urn:DataService&gt;</code></td>
</tr>
<tr>
<td>getDataService($x1 as element(t1:DataServiceRef)) as schema-element(t1:DataServiceRef)? external</td>
<td>This function returns the attributes of the specified data service such as the schema path, functions, and relational data source. Specify the path of the data service to retrieve the required result.</td>
<td><code>&lt;DataServiceRef xmlns=&quot;urn:metadata.ld.bea.com&quot;&gt; &lt;id xmlns=&quot;&quot;&gt;ld:DataServices/CustomerDB/CUST OMER.ds&lt;/id&gt; &lt;/DataServiceRef&gt;</code></td>
</tr>
</tbody>
</table>
**DataServiceRef (DataServiceRef.ds)**

The following table provides the declaration and description for the functions available in DataServiceRef.ds.

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Description</th>
<th>Sample Input</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getDataServiceRefsByFolder($x1 as xsd:string, $x2 as xsd:boolean) as schema-element(t1:DataServiceRef)* external</code></td>
<td>This function returns the data services that exist within a folder in the project. You need to specify the path of the project folder and set the boolean value to true for this function.</td>
<td>String parameter = ld:RTLAppDataServices/CustomerDB/ Boolean = true</td>
</tr>
<tr>
<td><code>getDataServiceRefs() as schema-element(t1:DataServiceRef)* external</code></td>
<td>This function returns the path to all the data services in the project. It does not require any parameters.</td>
<td>No input required.</td>
</tr>
<tr>
<td><code>getDependents($x1 as element(t1:DataServiceRef), $x2 as xsd:boolean) as schema-element(t1:DataServiceRef)* external</code></td>
<td>This function returns the path of the data services on which the specified data service depends. For this function, you need to specify the path of the data service whose dependents you need to determine. For example, if you need to find out the dependents for CUSTOMER.ds then specify the path of the data service as: ld:DataServices/CustomerDB/CUSTOMER.ds</td>
<td><code>&lt;urn:DataServiceRef xmlns:urn=&quot;urn:metadata.ld.bea.com&quot;&gt; &lt;id&gt;ld:DataServices/CustomerDB/CUSTOMER.ds&lt;/id&gt; &lt;/urn:DataServiceRef&gt;</code></td>
</tr>
<tr>
<td><code>getDependencies($x1 as element(t1:DataServiceRef), $x2 as xsd:boolean) as schema-element(t1:DataServiceRef)* external</code></td>
<td>This function returns the dependencies for the specified data service. For this function, you need to specify the path of the data service whose dependencies you need to determine.</td>
<td><code>&lt;urn:DataServiceRef xmlns:urn=&quot;urn:metadata.ld.bea.com&quot;&gt; &lt;id&gt;ld:DataServices/Demo/CustomerProfile.ds&lt;/id&gt; &lt;/urn:DataServiceRef&gt;</code></td>
</tr>
</tbody>
</table>
### Table 9-4 Functions in DataService

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Description</th>
<th>Sample Input</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>getFunctions</strong>($x1 as element(t1:DataServiceRef)) as schema-element(t1:Function)* external</td>
<td>This function returns the list of data service functions and their attributes such as function kind, arity, and schema path. For this function, specify the path of the data service as input.</td>
<td>&lt;DataServiceRef xmlns=&quot;urn:metadata.ld.bea.com&quot;&gt; &lt;id xmlns=&quot;&quot;&gt;ld:RTLAppDataServices/CustomerDB/CUSTOMER.ds&lt;/id&gt; &lt;/DataServiceRef&gt;</td>
</tr>
<tr>
<td><strong>getRelationships</strong>($x1 as element(t1:DataServiceRef)) as schema-element(t1:Relationship)* external</td>
<td>This function retrieves the path of data services which have any relationship with the specified data service. You need to specify the path of the data service, such as ld:RTLAppDataServices/CustomerDB/CUSTOMER.ds</td>
<td>DataServiceRef xmlns=&quot;urn:metadata.ld.bea.com&quot;&gt; &lt;id xmlns=&quot;&quot;&gt;ld:RTLAppDataServices/CustomerDB/CUSTOMER.ds&lt;/id&gt; &lt;/DataServiceRef&gt;</td>
</tr>
<tr>
<td><strong>getSchemaRefs</strong>($arg as element(t1:DataServiceRef), $transitive as xs:boolean) as element(t1:SchemaRef)* external</td>
<td>For this function, enter the path of the data service and set the boolean value to true for retrieving the list of associated schemas. This function also lists the paths of schemas for data services, which have a relationship with the specified data service.</td>
<td>&lt;urn:DataServiceRef xmlns:urn=&quot;urn:metadata.ld.bea.com&quot;&gt; &lt;id&gt;ld:RTLAppDataServices/CustomerDB/CUSTOMER.ds&lt;/id&gt; &lt;/urn:DataServiceRef&gt; Enter true as the boolean parameter.</td>
</tr>
<tr>
<td><strong>getDataService</strong>($x1 as element(t1:DataServiceRef)) as schema-element(t1:DataService)? external</td>
<td>This function returns the attributes of the specified data service such as the schema path, functions, and relational data source. Specify the path of the data service to retrieve the required result.</td>
<td>&lt;urn:DataServiceRef xmlns:urn=&quot;urn:metadata.ld.bea.com&quot;&gt; &lt;id xmlns=&quot;&quot;&gt;ld:RTLAppDataServices/CustomerDB/CUSTOMER.ds&lt;/id&gt; &lt;/urn:DataServiceRef&gt;</td>
</tr>
</tbody>
</table>
### Folder (folder.ds)

The following table provides the declaration and description for the functions available in Folder.ds.

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Description</th>
<th>Sample Input</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>getFolder()</strong> as schema-element(t1:Folder)* external</td>
<td>This function provides a list of paths of folders and data services that exist within the Data Services Platform project. It does not require any parameters.</td>
<td>No input required.</td>
</tr>
</tbody>
</table>
| **getFolder($x1 as xsd:string, $x2 as xsd:boolean) as schema-element(t1:Folder)* external** | This folder returns the paths of all the data services that exists within a specified folder. You need to specify two parameters for this function, which include: | • Parameter 1 (string) = ld:RTLAppDataServices/CustomerDB  
• Parameter 2 (boolean) = true |
| **getDataServiceRefs($x1 as element(t1:Folder)) as schema-element(t1:DataServiceRef)* external** | This function also provides the paths of the data services that exist within a folder. To retrieve this information, specify the path of the folder as input. | <Folder xmlns="urn:metadata.ld.bea.com">
  <id xmlns="" xmlns="ld:RTLAppDataServices/CustomerDB"></id>
</Folder> |
**Function (Function.ds)**

The following table provides the declaration and description for the functions in Function.ds.

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Description</th>
<th>Sample Input</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getFunctionById($x1 as element(t1:Function Id)) as schema-element(t1:Function) external4</code></td>
<td>This function returns the path of the data service and schema along with function arity, function kind and return type information about the specified function. For this function, specify the function ID and arity as input.</td>
<td><code>&lt;FunctionId name=&quot;cus:CUSTOMER &quot; arity=&quot;0&quot; xmlns:cus=&quot;ld:RTLAppDataServices/CustomerDB/CUSTOMER&quot; xmlns=&quot;urn:metadata.ld.bea.com&quot;/&gt;</code></td>
</tr>
<tr>
<td><code>getDataService($arg as element(md:Function Id)) as element(md:DataService)</code></td>
<td>This function returns the function arity and physical data source information for the specified function. For this function, you need to specify the function ID, path of the data service and schema.</td>
<td><code>&lt;xml version=&quot;1.0&quot; encoding=&quot;UTF-8&quot; ?&gt; &lt;urn:Function kind=&quot;read&quot; xmlns:acc=&quot;ld:RTLAppDataServices/CustomerDB/CUSTOMER&quot; xmlns:urn=&quot;urn:metadata.ld.bea.com&quot; &gt; &lt;urn:FunctionId arity=&quot;0&quot; name=&quot;acc:getAll&quot;&gt;&lt;/urn:FunctionId&gt; &lt;returnType kind=&quot;element&quot; name=&quot;urn:Account&quot; quantifier=&quot;1&quot; schemaId=&quot;ld:RTLAppDataServices/CustomerDB/schemas/CUSTOMER.xsd&quot;&gt; &lt;/returnType&gt; &lt;urn:DataServiceRef&gt; &lt;id&gt;ld:RTLAppDataServices/CustomerDB/CUSTOMER.ds&lt;/id&gt; &lt;/urn:DataServiceRef&gt; &lt;/urn:Function&gt;</code></td>
</tr>
</tbody>
</table>
**Advanced Topics**

### Table 9-6 Functions in Function.ds

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Description</th>
<th>Sample Input</th>
</tr>
</thead>
</table>
| `getRelationship($arg as element(md:Function)) as element(md:Relationship)` external `getFunction()relationship function` | This function returns the relationship target and path of the data services with which the navigation function has a relationship. Specify the function ID, path of the data service and schema as input. | `<xml version="1.0" encoding="UTF-8"?>
<urn:Function kind="navigate"
xmlns:acc="ld:RTLAppDataServices/CustomerDB/Customer"
xmlns:urn="urn:metadata.ld.bea.com"
>
    <urn:FunctionId arity="1"
name="acc:getDISCOUNT">
    </urn:FunctionId>
    <returnType kind="element"
name="urn:getDISCOUNT" quantifier="1"
schemaId="ld:RTLAppDataServices/CustomerDB/schemas/CUSTOMER.xsd">
        <parameter name="arg">
            <type kind="navigate"
name="urn:DISCOUNT" quantifier="*"
schemaId="ld:RTLAppDataServices/CustomerDB/CUSTOMER.xsd">
            </type>
        </parameter>
    </returnType>
    <urn:DataServiceRef>
        <id ld:RTLAppDataServices/CustomerDB/CUSTOMER.ds"></id>
    </urn:DataServiceRef>
    <urn:DataServiceRef>
        <roleId>DISCOUNT</roleId>
    </urn:DataServiceRef>
</urn:Function>` |

**Note:** This function is applicable to navigation functions only.
For this function, specify the function ID and path of the data service to retrieve the path of the schemas associated with the data service.

Table 9-6  Functions in Function.ds

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Description</th>
<th>Sample Input</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getSchemaRefs($x1 as element(t1:Function), $x2 as xsd:boolean) as schema-element(t1:SchemaRef)* external DataServicesRef_</code></td>
<td>For this function, specify the function ID and path of the data service to retrieve the path of the schemas associated with the data service.</td>
<td><code>&lt;urn:Function kind=&quot;navigate&quot; xmlns:acc=&quot;ld:RTLAppDataServices/CustomerDB/CUSTOMER&quot; xmlns:urn=&quot;urn:metadata.ld.bea.com&quot;  &gt;  &lt;urn:FunctionId name=&quot;acc:getDISCOUNT&quot; arity=&quot;1&quot;/&gt;  &lt;returnType name=&quot;DISCOUNT&quot; kind=&quot;element&quot; quantifier=&quot;*&quot; schemaId=&quot;ld:RTLAppDataServices/CustomerDB/schemas/CUSTOMER.xsd&quot;/&gt;  &lt;!--Zero or more repetitions:--&gt;  &lt;parameter name=&quot;arg&quot;&gt;  &lt;type name=&quot;DISCOUNT&quot; kind=&quot;element&quot; quantifier=&quot;*&quot; schemaId=&quot;ld:RTLAppDataServices/CustomerDB/schemas/CUSTOMER.xsd&quot;/&gt;  &lt;/parameter&gt;  &lt;urn:DataServiceRef&gt;&lt;id&gt;ld:RTLAppDataServices/CustomerDB/Customer.ds&lt;/id&gt;  &lt;/urn:DataServiceRef&gt;  &lt;id&gt;ld:RTLAppDataServices/CustomerDB/Customer.ds&lt;/id&gt;  &lt;/urn:Function&gt;</code></td>
</tr>
</tbody>
</table>
Relationship (Relationship.ds)

The following table provides the declaration and description for the functions available in Relationship.ds.

**Note:** The functions in Relationship.ds can be used to access metadata only for navigation functions.

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Description</th>
<th>Sample Input</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>getFunctions</strong> ($arg as element(md:Relationship)) as element(md:Function)</td>
<td>This function returns the attributes of the function that you specify as input. You need to specify the following parameters for this function:</td>
<td>&lt;urn:Relationship xmlns:acc=&quot;ld:RTLAppDataServices/CustomerDB/CUSTOMER&quot; xmlns:urn=&quot;urn:metadata.ld.bea.com&quot;&gt; &lt;!--1 to 2 repetitions:--&gt; &lt;relationshipTarget roleName=&quot;DISCOUNT&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; description=&quot;&quot;&gt; &lt;!--Zero or more repetitions:--&gt; <a href="">urn:DataServiceRef</a> &lt;id&gt;ld:RTLAppDataServices/CustomerDB/CUSTOMER.ds&lt;/id&gt; &lt;/urn:DataServiceRef&gt; &lt;!---Zero or more repetitions:--&gt; &lt;urn:FunctionId name=&quot;acc:getDISCOUNT&quot; arity=&quot;1&quot;/&gt; &lt;/relationshipTarget&gt; &lt;/urn:Relationship&gt;</td>
</tr>
</tbody>
</table>
You need to specify the following parameters for this function:

- String parameter = Path of the data service
- Function ID
- Values for minOccurs and maxOccurs

Example:

```xml
<urn:Relationship
  xmlns:acc="ld:RTLAppDataServices/CustomerDB/CUSTOMER" xmlns:urn="urn:metadata.ld.bea.com">
  <!--1 to 2 repetitions:-->  
  <relationshipTarget roleName="DISCOUNT"
    minOccurs="1" maxOccurs="1" description="">
    <urn:DataServiceRef>
      <id>ld:RTLAppDataServices/CustomerDB/CUSTOMER.ds</id>
      </urn:DataServiceRef>
    </urn:FunctionId name="acc:getDISCOUNT" arity="1"/>
  </relationshipTarget>
</urn:Relationship>
```
### Functions in Relationship.ds

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Description</th>
<th>Sample Input</th>
</tr>
</thead>
</table>
| **getDataServices** ($arg as element(md:Relationship) as element(md:DataService)) | This function returns the attributes, such as relational datasource and function arity, of the navigation function of the data service. For this function, you need to specify the following parameters:  
  - String parameter = Path of the data service  
  - String parameter = Path of the schema  
  - Values for maxOccurs and minOccurs  
  - FunctionID |  
| | | `<xml version="1.0" encoding="UTF-8" ?>  
| | | <urn:Relationship xmlns:acc="ld:RTLAppDataServices/CustomerDB/CUSTOMER" xmlns:urn="urn:metadata.ld.bea.com">  
| | | <relationshipTarget description="" maxOccurs="1" minOccurs="1" roleName="DISCOUNT">  
| | | <urn:DataServiceRef>  
| | | <id>ld:RTLAppDataServices/CustomerDB/CUSTOMER.ds</id>  
| | | </urn:DataServiceRef>  
| | | <urn:FunctionId arity="1" name="acc:getDISCOUNT">  
| | | </urn:FunctionId>  
| | | </relationshipTarget>  
| | | <relationshipTarget description="" maxOccurs="1" minOccurs="1" roleName="DISCOUNT" xmlns:acc="ld:RTLAppDataServices/CustomerDB/CUSTOMER">  
| | | <urn:DataServiceRef>  
| | | <id>ld:RTLAppDataServices/CustomerDB/CUSTOMER.ds</id>  
| | | </urn:DataServiceRef>  
| | | <urn:FunctionId arity="1" name="acc:getDISCOUNT">  
| | | </urn:FunctionId>  
| | | </relationshipTarget>  
| | | </urn:Relationship>` |
**Schema (Schema.ds)**

The following table provides the declaration and description for the functions available in Schema.ds

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Description</th>
<th>Sample Input</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getSchema($x1 as element(t1:SchemaRef)) as schema-element(t1:SchemaRef)* external</code></td>
<td>This function returns the schema attributes of the schema associated with the data service. You need to specify the path of the schema which you need to access as string parameter. For example: <code>ld:RTLAppDataServices/CustomerDB/schemas/CUSTOMER.xsd</code></td>
<td><code>&lt;urn:SchemaRef xmlns:urn=&quot;urn:metadata.ld.bea.com&quot;&gt; &lt;id&gt;ld:RTLAppDataServices/CustomerDB/schemas/CUSTOMER.xsd&lt;/id&gt; &lt;/urn:SchemaRef&gt;</code></td>
</tr>
<tr>
<td><code>getSchemaRef($x1 as element(t1:Schema)) as schema-element(t1:SchemaRef)</code></td>
<td>This function returns the path of the schema of the data service. Specify the schema path to get the reference to the schema. For example: <code>ld:RTLAppDataServices/CustomerDB/schemas/CUSTOMER_TABLE.xsd</code></td>
<td><code>&lt;urn:SchemaRef xmlns:urn=&quot;urn:metadata.ld.bea.com&quot;&gt; &lt;id&gt;ld:RTLAppDataServices/CustomerDB/schemas/CUSTOMER_TABLE.xsd&lt;/id&gt; &lt;/urn:SchemaRef&gt;</code></td>
</tr>
</tbody>
</table>
The following table provides the declaration and description for the functions available in SchemaRef.ds.

### Table 9-9 Functions in SchemaRef.ds

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Description</th>
<th>Sample Input</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getDependencies($x1 as element(t1:SchemaRef), $x2 as xs:boolean) as schema-element(t1:SchemaRef)* external</code></td>
<td>This function returns the dependencies of the specified data service. You need to specify the path of the schema for the data service as a string parameter. For example: <code>&lt;urn:SchemaRef xmlns:urn=&quot;urn:metadata.ld.bea.com&quot;&gt; &lt;id xmlns=&quot;&quot;&gt;ld:DataServices/Demo/schemas/CustomerProfile.xsd&lt;/id&gt; &lt;/urn:SchemaRef&gt;</code></td>
<td><code>&lt;urn:SchemaRef xmlns:urn=&quot;urn:metadata.ld.bea.com&quot;&gt; &lt;id xmlns=&quot;&quot;&gt;ld:DataServices/Demo/schemas/CustomerProfile.xsd&lt;/id&gt; &lt;/urn:SchemaRef&gt;</code></td>
</tr>
<tr>
<td><code>getSchema($x1 as element(md:SchemaRef)) as schema-element(md:Schema)* external</code></td>
<td>This function returns the schemas associated with the data service. You need to specify the path of the schema for the data service as a string parameter. For example: <code>&lt;urn:SchemaRef xmlns:urn=&quot;urn:metadata.ld.bea.com&quot;&gt; &lt;id xmlns=&quot;&quot;&gt;ld:RTLAppDataServices/CustomerDB/schemas/CUSTOMER.xsd&lt;/id&gt; &lt;/urn:SchemaRef&gt;</code></td>
<td><code>&lt;urn:SchemaRef xmlns:urn=&quot;urn:metadata.ld.bea.com&quot;&gt; &lt;id xmlns=&quot;&quot;&gt;ld:RTLAppDataServices/CustomerDB/schemas/CUSTOMER.xsd&lt;/id&gt; &lt;/urn:SchemaRef&gt;</code></td>
</tr>
</tbody>
</table>
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 the application 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 clients may require and to implement a data service function for each view. Instead, designers can choose to specify a broader, more generic interface for accessing a business entity and allow client applications to control views as desired through filters.

Using the API, you can specify that only objects that meet a particular condition in the function return set be returned to the client. A filter is therefore 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.

The effects of a filter can vary, depending on the desired results. Consider, for example, the CUSTOMERS data object shown in Figure 9-10. The data object contains multiple complex elements (CUSTOMER and ORDERS) and several simple elements, including ORDER_AMOUNT.

You can apply a filter to any element in this hierarchy. For example, you could apply a filter to return all CUSTOMER objects but filter ORDERS than have an ORDER_AMOUNT greater than 1000. Similarly, you could apply a filter to return only the CUSTOMER objects that have at least one large order.

You can also use a filter to specify the order criteria (ascending or descending) in which results should be returned from the data service. Finally, you can use a filter to set the maximum number of results to be returned.
Advanced Topics

**Note:** Filter 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 data sources.
Introducing the Filter API

You specify filters using the FilterXQuery object, which includes the following methods enabling you to add a filter, create a filter to apply later, specify the sort order, and set a limit on the number of results returned:

- `addFilter()`
- `createFilter()`
- `addOrderBy()`
- `setLimit()`

**addFilter()**

The `addFilter()` method enables you to create a filter and add it to the list of filters. The `addFilter()` method has several signatures with different parameters, including the following:

```java
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` specifies the node (the XPath relative to the document element) that the 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 comprise the condition statement. The `operator` parameter specifies the type of comparison to be made against the specified `value`. See Table 9-11, “Filter Operators,” on page 9-21 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.
**createFilter()**

The `createFilter()` method enables you to create a filter that you can later apply to any of the objects in the hierarchy. The `createFilter()` method has several signatures with different parameters, including the following:

```java
public void createFilter(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:

- **field** is the node against which the filtering condition is tested (specified as the XPath relative to the document element).

- **operator** and **value** together comprise the condition statement. The **operator** parameter specifies the type of comparison to be made against the specified **value**. See Table 9-11, “Filter Operators,” on page 9-21 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.

**addOrderBy()**

The `addOrderBy()` method enables you to add a sort criteria (either ascending or descending) to the specified object. The `addOrderBy()` method has the following signature:

```java
public void addOrderBy(java.lang.String appliesTo,
                      java.lang.String field,
                      java.lang.String sort)
```

The method takes the following arguments:

- **appliesTo** specifies the node returned by the filter (specified as the XPath relative to the document element).

- **field** specifies the node to which the ordering is applied, relative (not the full path) to the **appliesTo** node.

- **sort** is the sort criteria (either ascending or descending)
setLimit()

The setLimit() method enables you to specify the maximum number of entries to return of the specified object. The setLimit() method has the following signature:

```java
public void setLimit(java.lang.String appliesTo,
                     java.lang.String max)
```

The method takes the following arguments:

- **appliesTo** specifies the node (the XPath relative to the document element) to which the filter is applied.
- **max** is the maximum number of entries to return (an int value specified as a string, for example, “10”).

Exploring the Filter Operators

Table 9-11 describes the operators that you can apply to filter conditions.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Usage Note or Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>LESS_THAN</td>
<td>Can also use &quot;&lt;&quot;. For example:</td>
</tr>
</tbody>
</table>
|                           | myFilter.addFilter("CUST/CUST_ORDER/ORDER",
|                           |   "CUST/CUST_ORDER/ORDER/ORDER_AMOUNT","<",
|                           |   "1000");                                              |
|                           | is identical to                                        |
|                           | myFilter.addFilter("CUST/CUST_ORDER/ORDER",
|                           |   "CUST/CUST_ORDER/ORDER/ORDER_AMOUNT",
|                           |   FilterXQuery.LESS_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.                             |
Using Filters

Filtering capabilities are available to Mediator and ALDSP Control client applications. 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:

```java
com.bea.ld.filter.FilterXQuery;
```

To use a filter, perform the following steps:

1. Create an `FilterXQuery` object, such as:
   ```java
   FilterXQuery myFilter = new FilterXQuery();
   ```

2. Add a condition to the filter object using the `addFilter()` method.

   The following example shows how to add a filter to have orders with an order amount greater than 1000 returned (note that the optional `everyChild` parameter is not specified, so order amounts below 1000 will also be returned):
   ```java
   myFilter.addFilter("CUSTOMERS/CUSTOMER/ORDER",
                    "CUSTOMERS/CUSTOMER/ORDER/ORDER_AMOUNT",
                    ">",
                    "1000");
   ```

Note: Filter API Javadoc, and other Data Services Platform APIs are available on e-docs.
3. Use the Mediator API call `setFilter()` to add the filter to a data service, passing the `FilterXQuery` instance as an argument. For example,

```java
RequestConfig config = new RequestConfig();
config.setFilter(myFilter);
CUSTOMERDAS custDAS = CUSTOMER.getInstance(ctx, "RTLApp");
custDAS.myOperation(config);
```

4. Invoke the data service function.

For more information on invoking data service functions, see Chapter 3, “Invoking Data Services from Java Clients.”

## Filtering Examples

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, but containing all ORDERS (small and large) for each such CUSTOMER.

The following examples show how filters can be applied in several different ways:

- **Returns all CUSTOMER objects but only their large ORDER objects:**
  ```java
  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 objects:**
  ```java
  FilterXQuery myFilter = new FilterXQuery();
  myFilter.addFilter("CUSTOMERS/CUSTOMER",
    "CUSTOMERS/CUSTOMER/ORDER/ORDER_AMOUNT",
    FilterXQuery.GREATER_THAN,"1000");
  ```
Advanced Topics

- Returns only CUSTOMER objects that have at least one large order and return only large ORDER objects:

  ```java
  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");
  ```

- Returns only CUSTOMER objects for which every ORDER_AMOUNT is greater than 1000:

  ```java
  FilterXQuery myFilter = new FilterXQuery();
  myFilter.addFilter("CUSTOMERS/CUSTOMER",
                      "CUSTOMERS/CUSTOMER/ORDER/ORDER_AMOUNT",
                      FilterXQuery.GREATER_THAN,"1000",true);
  ```

  Note that the `everyChild` flag is set to true; by default this parameter is false.

**Specifying a Compound Filter**

You can create a filter with two conditions using logical AND and OR operators. **Listing 9-1** uses the AND operator to apply a combination of filters to a result set, given a data service instance `customerDS`.

**Listing 9-1   Example of Combining Filters by Using Logical Operators**

```java
FilterXQuery myFilter = new FilterXQuery();
Filter f1 = myFilter.createFilter("CUSTOMERPROFILE/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

You can specify the order criteria (ascending or descending) in which results should be returned from the data service. The `addOrderBy()` method accepts a property name as the criterion upon which the ascending or descending decision is based.

Listing 9-2 provides an example of creating a filter to return customer profiles in ascending order, based on the date each person became a customer.

Listing 9-2  Example of Applying an Ordering Filter

```java
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 to be returned using the `setLimit()` method. Listing 9-3 shows how to use the `setLimit()` method to limit the number of active addresses in the result set to 10, given a data service instance `ds`.

Listing 9-3  Example of Applying a Filter that Truncates (Limits) Results

```java
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 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 9-4 shows a complete example; the code returns the results of a data service function named getCustomers(), which is in the namespace:

\[
\text{ld:DataServices/RTLServices/Customer}
\]

Listing 9-4   Invoking Data Service Functions using an Ad Hoc Query

```java
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();
```
ALDSP 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 9-4 specifies a <Results> container object to hold the array of CUSTOMER_PROFILE elements that will be returned by the getCustomer() data service function.

Security policies defined for a data service apply to the data service calls in an ad hoc query as well. If an ad hoc query uses secured resources, the appropriate credentials must be passed when creating the JNDI initial context. (For more information, see “Obtaining the WebLogic JNDI Context for ALDSP” on page 3-44.)

As with the PreparedStatement interface of JDBC, the PreparedExpression interface supports dynamically binding variables in ad hoc query expressions. PreparedExpression provides several methods (bindType( ) methods; see Table 9-12), for binding values of various data types.

Table 9-12 PreparedExpression Methods for Bind Variables

<table>
<thead>
<tr>
<th>To bind data type of...</th>
<th>Use bind method...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>bindBinary(javax.xml.namespace.QName qname, byte[] abyte0)</td>
</tr>
<tr>
<td>BinaryXML</td>
<td>bindBinaryXML(javax.xml.namespace.QName qname, byte[] abyte0)</td>
</tr>
<tr>
<td>Boolean</td>
<td>bindBoolean(javax.xml.namespace.QName qname, boolean flag)</td>
</tr>
<tr>
<td>Byte</td>
<td>bindByte(javax.xml.namespace.QName qname, byte byte0)</td>
</tr>
<tr>
<td>Date</td>
<td>bindDate(javax.xml.namespace.QName qname, java.sql.Date date)</td>
</tr>
<tr>
<td>Calendar</td>
<td>bindDateTime(javax.xml.namespace.QName qname, java.util.Calendar calendar)</td>
</tr>
<tr>
<td>DateTime</td>
<td>bindDateTime(javax.xml.namespace.QName qname, java.util.Date date)</td>
</tr>
<tr>
<td>DateTime</td>
<td>bindDateTime(javax.xml.namespace.QName qname, java.sql.Timestamp timestamp)</td>
</tr>
</tbody>
</table>
To bind data type of... | Use bind method...
---|---
BigDecimal | `bindDecimal(javax.xml.namespace.QName qname, java.math.BigDecimal bigdecimal)`
double | `bindDouble(javax.xml.namespace.QName qname, double d)`
Element | `bindElement(javax.xml.namespace.QName qname, org.w3c.dom.Element element)`
Object | `bindElement(javax.xml.namespace.QName qname, java.lang.String s)`
float | `bindFloat(javax.xml.namespace.QName qname, float f)`
int | `bindInt(javax.xml.namespace.QName qname, int i)`
long | `bindLong(javax.xml.namespace.QName qname, long l)`
Object | `bindObject(javax.xml.namespace.QName qname, java.lang.Object obj)`
short | `bindShort(javax.xml.namespace.QName qname, short word0)`
String | `bindString(javax.xml.namespace.QName qname, java.lang.String s)`
Time | `bindTime(javax.xml.namespace.QName qname, java.sql.Time time)`
URI | `bindURI(javax.xml.namespace.QName qname, java.net.URI uri)`

To use the `bindType` methods, pass the variable name as an XML qualified name (`QName`) along with its value; for example:

```java
adHocQuery.bindInt(new QName("i"), 94133);
```

Listing 9-5 shows an example of using a `bindInt()` method in the context of an ad hoc query.
Listing 9-5  Binding a Variable to a QName (Qualified Name) for use in an Ad Hoc Query

PreparedExpression adHocQuery = DataServiceFactory.preparedExpression(
    context, "RTLApp",
    "declare variable $i as xs:int external;
    <result><zip>{fn:data($i)}</zip></result>";
adHocQuery.bindInt(new QName("i"), 94133);
XmlObject adHocResult = adHocQuery.executeQuery();

Note:  For more information on QNames, see:

http://www.w3.org/TR/xmlschema-2/#QName

Listing 9-6 shows a complete ad hoc query example, using the PreparedExpression interface and QNames to pass values in bind methods.

Listing 9-6  Sample Ad Hoc Query

import com.bea.ld.dsmediator.client.DataServiceFactory;
import com.bea.ld.dsmediator.client.PreparedExpression;
import com.bea.xml.XmlObject;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.xml.namespace.QName;
import weblogic.jndi.Environment;
public class AdHocQuery{
    public static InitialContext getInitialContext() throws NamingException {
        Environment env = new Environment();
        env.setProviderUrl("t3://localhost:7001");
        env.setInitialContextFactory("weblogic.jndi.WLInitialContextFactory");
        env.setSecurityPrincipal("weblogic");
        env.setSecurityCredentials("weblogic");
        return new InitialContext(env.getInitialContext().getEnvironment());
    }
    public static void main(String args[]){
        System.out.println("-------- Ad Hoc Client --------");
        try{
            StringBuffer xquery = new StringBuffer();
            xquery.append("declare variable $p_firstname as xs:string external; \n");

```
xquery.append("declare variable $p_lastname as xs:string external; \n");

xquery.append("declare namespace ns1="ld:DataServices/MyQueries/XQueries"; \n");
xquery.append("declare namespace ns0="ld:DataServices/CustomerDB/CUSTOMER"; \n\n");

xquery.append("<ns1:RESULTS> \n");
xquery.append("\n");
xquery.append(" for $customer in ns0:CUSTOMER() \n");
xquery.append(" where ($customer/FIRST_NAME eq $p_firstname \n");
xquery.append(" and $customer/LAST_NAME eq $p_lastname) \n");
xquery.append(" return \n");
xquery.append(" $customer \n");
xquery.append(" } \n");
xquery.append("</ns1:RESULTS>\n");

PreparedExpression pe = DataServiceFactory.prepareExpression(
    getInitialContext(), "RTLApp", xquery.toString());
pe.bindString(new QName("p_firstname"), "Jack");
pe.bindString(new QName("p_lastname"), "Black");
XmlObject results = pe.executeQuery();
System.out.println(results);

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