Oracle® Fusion Middleware
Developing Data Service Integrator Applications

12*c* (12.1.3)

E47944-01

May 2014

Oracle Fusion Middleware Developing Data Service Integrator Applications, 12*c* (12.1.3)

E47944-01

Contents

7 Advanced Topics

Preface

This document describes how to develop applications for the Oracle Data Service Integrator.

Audience

This document is intended for application developers.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Related Documents

For more information, see the following documents in the Oracle Data Service Integrator documentation set:

- *Oracle Fusion Middleware Using Data Service Integrator XQuery Engine*
- *Oracle Fusion Middleware Administering Data Service Integrator*
- *Oracle Fusion Middleware Installing Data Service Integrator*
- *Oracle Fusion Middleware Developing Applications with Data Service Integrator*
- *Oracle Fusion Middleware Data Services Java API for Oracle Data Integrator*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1 Introducing Data Services for Client Applications

This chapter provides an overview of Oracle Data Service Integrator for client application developers. It includes the following topics:

1.1 Introduction

Oracle Data Service Integrator brings data access into the world of service-oriented architecture (SOA). Oracle Data Service Integrator 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 Oracle Data Service Integrator. 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 Oracle Data Service Integrator, 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, Oracle Data Service Integrator 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 Oracle Data Service Integrator-aware client applications. It explains the various client access mechanisms that Oracle Data Service Integrator 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 Oracle Data Service Integrator update framework.

- For information about server-side aspects of creating and managing data services, see the *Oracle Fusion Middleware Data Service Integrator Developer's Guide*.
- For information on administering data services, including metadata, cache, and security management, see the *Oracle Fusion Middleware Administering Data Service Integrator* guide.

1.2 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 *Oracle Fusion Middleware Data Service Integrator Developer's Guide*.

1.3 What is an Oracle Data Service Integrator Client Application?

An Oracle Data Service Integrator 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, Oracle 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.

Regardless of the client type, Oracle Data Service Integrator 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, Oracle Data Service Integrator:

- 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 Oracle Data Service Integrator 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 Oracle, IBM, SAP, and others.

1.4 Choosing a Client Programming Model

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

Table 1–1 Summary of Techniques for Exposing Data Services to Clients

Data Access Technique	Description	Advantages	Other Details
Java Data Service Mediator	Instantiate a remote data service interface and invoke public methods on the interface. See Chapter 3, "Invoking Data Services from Java Clients."	Full read/write access to data.	Requires adequate Java programming skills.
Web services	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."	Read/write access to data. Industry standard.	N/A
SQL	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."	Accepted by commonly used reporting tools.	Read-only, and for use SQL-based clients only.
ADO.NET	Allows interoperability between Oracle Data Service Integrator data services and ADO.NET. See Chapter 6, "Supporting ADO.NET Clients."	Enables Oracle Data Service Integrator data services to be used in Microsoft ADO.NET client applications.	Specific to ADO.NET applications.

1.5 Introducing Service Data Objects (SDO)

Service Data Objects (SDO), a specification proposed jointly by Oracle, IBM, 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. Oracle Data Service Integrator implements such an SDO mediator.

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

1.6 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. Oracle Data Service Integrator provides a Data Service Mediator API, a server-side component of the Oracle Data Service Integrator 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"

Note: Oracle Data Service Integrator 10gR3 does not support backward compatibility with ALDSP 2.x (all ALDSP 2.x deprecated APIs are no longer supported in Oracle Data Service Integrator Release 12*c* (12.1.3). This includes all classes in the `com.bea.dsp.dsmediator.client` and `com.bea.ld.dsmediator.update` packages.

1.7 Typical Client Application Development Process

Developing an Oracle Data Service Integrator-enabled client applications encompasses these steps:

1. Identify the data services you want to use in your application. The Oracle Data Service Integrator Administration Console can be used to find all services available on your Oracle WebLogic Server. The Oracle Data Service Integrator Administration Console serves as a data service registry within the Oracle Data Service Integrator 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 1–1 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.)

1.8 Security Considerations in Client Applications

Oracle Data Service administrators can control access to deployed Oracle Data Service Integrator resources through role-based security policies. Oracle Data Service Integrator leverages and extends the security features of the underlying WebLogic platform. Roles can be set up in the WebLogic Administration Console. (See the *Oracle Fusion Middleware Administering Data Service Integrator* guide for detailed information about the Oracle Data Service Integrator Administration 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 Oracle Data Service Integrator security, see "Securing Oracle Data Service Integrator Resources," in the *Oracle Fusion Middleware Administering Data Service Integrator* guide.

For complete information on WebLogic security, see *Programming WebLogic Security.*

1.9 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 the Oracle Data Service Integrator auditing capabilities to obtain performance profile information that you can use to identify and resolve performance problems if they occur. For detailed information on Oracle Data Service Integrator audit capabilities see the *Oracle Fusion Middleware Administering Data Service Integrator* guide.

1.10 Client Classpath Settings

The following tables provide classpath requirements for:

- Java Mediator API clients (dynamic and static)
- Web Service clients (dynamic and static)
- JMX Mbean Management API clients
- JDBC API clients

1.10.1 Java Mediator API Clients

Client applications using the Oracle Data Service Integrator Mediator API need one of the following classpath settings:

Note: For information on generating `wlfullclient.jar`, see *Oracle Fusion Middleware Programming Stand-alone clients for Oracle Weblogic Server.*

Example 1–1 Static Java Mediator API Client Classpath (with weblogic.jar)

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

Example 1–2 Static Java Mediator API Client Classpath (with generated wlfullclient.jar)

```
CLASSPATH=
   <ODSI_HOME>/lib/ld-client.jar
   <ODSI_HOME>/lib/sdo.jar
   <WL_HOME>/modules/com.bea.core.xml.xmlbeans_2.4.0.0_2-5-1.jar
   <WL_HOME>/server/lib/wlfullclient.jar
```

Example 1–3 Dynamic Mediator API Classpath (with weblogic.jar)

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

Example 1–4 Dynamic Mediator API Classpath (with generated wlfullclient.jar)

```
CLASSPATH=
   <ODSI_HOME>/lib/ld-client.jar
   <ODSI_HOME>/lib/sdo.jar
   <WL_HOME>/modules/ com.bea.core.xml.xmlbeans_1.0.0.0_2-6-0.jar
   <WL_HOME>/server/lib/wlfullclient.jar
```

1.10.2 Web Services Clients

Client applications using the Oracle Data Service Integrator Native Web Services feature need one of the following classpath settings:

Example 1–5 Static Web Service Client Classpath (with weblogic.jar)

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

Example 1–6 Static Web Service Client Classpath (with generated wlfullclient.jar)

```
CLASSPATH=
   <ODSI_HOME>/lib/ld-client.jar
   <ODSI_HOME>/lib/sdo.jar
   <WL_HOME>/modules/ com.bea.core.xml.xmlbeans_1.0.0.0_2-6-0.jar
   <WL_HOME>/modules/ com.bea.core.xml.beaxmlbeans_1.0.0.0_2-6-0.jar
   <WL_HOME>/server/lib/wlfullclient.jar

   <COMMON_COMPONENTS_HOME>/modules/clients/com.oracle.webservices.fmw.client_
12.1.3.jar
   <WL_HOME>/modules/clients/ com.oracle.webservices.wls.jaxws-wlswss-client.jar
```

Example 1–7 Dynamic Web Service Clients (with weblogic.jar)

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

Example 1–8 Dynamic Web Service Clients (with generated wlfullclient.jar)

```
CLASSPATH=
   <ODSI_HOME>/lib/ld-client.jar
   <ODSI_HOME>/lib/sdo.jar
   <WL_HOME>/modules/ com.bea.core.xml.xmlbeans_1.0.0.0_2-6-0.jar
   <WL_HOME>/modules/ com.bea.core.xml.xmlbeans_1.0.0.0_2-6-0.jar <WL_
HOME>/server/lib/wlfullclient.jar

   <COMMON_COMPONENTS_HOME>/modules/clients/com.oracle.webservices.fmw.client_
12.1.3.jar
   <WL_HOME>/modules/clients/ com.oracle.webservices.wls.jaxws-wlswss-client.jar
```

1.10.3 JMX Mbean Management API Client Classpath

The JMX Mbean Management API needs the following classpath settings:

Example 1–9 JMX Mbean Management API Client Classpath (with weblogic.jar)

```
CLASSPATH=
   <ODSI_HOME>/lib/ld-client.jar
   <WL_HOME>/server/lib/weblogic.jar
```

Example 1–10 JMX Mbean Management API Client Classpath (with generated wlfullclient.jar)

```
CLASSPATH=
   <ODSI_HOME>/lib/ld-client.jar

   Thin-client -> [<WL_HOME>/server/lib/wljmxclient.jar and <WL_
HOME/server/lib/wlclient.jar]
   (OR)
   Thick-client ->[<WL_HOME>/server/lib/wlfullclient.jar]
```

1.10.4 Oracle Data Service Integrator JDBC API Client Classpath

The Oracle Data Service Integrator JDBC API client needs the following classpath settings:

Example 1–11 Oracle Data Service Integrator JDBC API Client Classpath (with weblogic.jar)

```
CLASSPATH=
   <ODSI_HOME>/lib/ld-client.jar
   <ODSI_HOME>/lib/ldjdbc.jar
   <WL_HOME>/server/lib/weblogic.jar
```

Example 1–12 Oracle Data Service Integrator JDBC API Client Classpath (with wlfullclient.jar)

```
CLASSPATH=
   <ODSI_HOME>/lib/ld-client.jar
   <ODSI_HOME>/lib/ldjdbc.jar
   <WL_HOME>/server/lib/wlfullclient.jar
```

2 Data Programming Model and Update Framework

Oracle Data Service Integrator implements Service Data Objects (SDO) as its data client-application programming model. This chapter discusses SDO concepts and APIs that are of interest to Oracle Data Service Integrator client application developers.

- Section 2.1, "Introduction"
- Section 2.2, "Oracle Data Service Integrator and SDO"
- Section 2.3, "Role of the Mediator API and SDO"

Note: Oracle Data Service Integrator 10gR3 does not support backward compatibility with ALDSP 2.x (all ALDSP 2.x deprecated APIs are no longer supported in Oracle Data Service Integrator 10gR3). This includes all classes in the `com.bea.dsp.dsmediator.client` and `com.bea.ld.dsmediator.update` packages.

2.1 Introduction

SDO is an architecture and set of APIs for working with data objects while disconnected from their source. In Oracle Data Service Integrator, 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 Section 1.5, "Introducing Service Data Objects (SDO).")

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 Section 1.6, "Introducing the Data Service Mediator API," the Oracle Data Service Integrator 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 the Oracle Data Service Integrator implementation of the SDO data programming model, as well as its update framework.

2.2 Oracle Data Service Integrator 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.

Note: For information on the Mediator API, see Chapter 3, "Invoking Data Services from Java Clients."

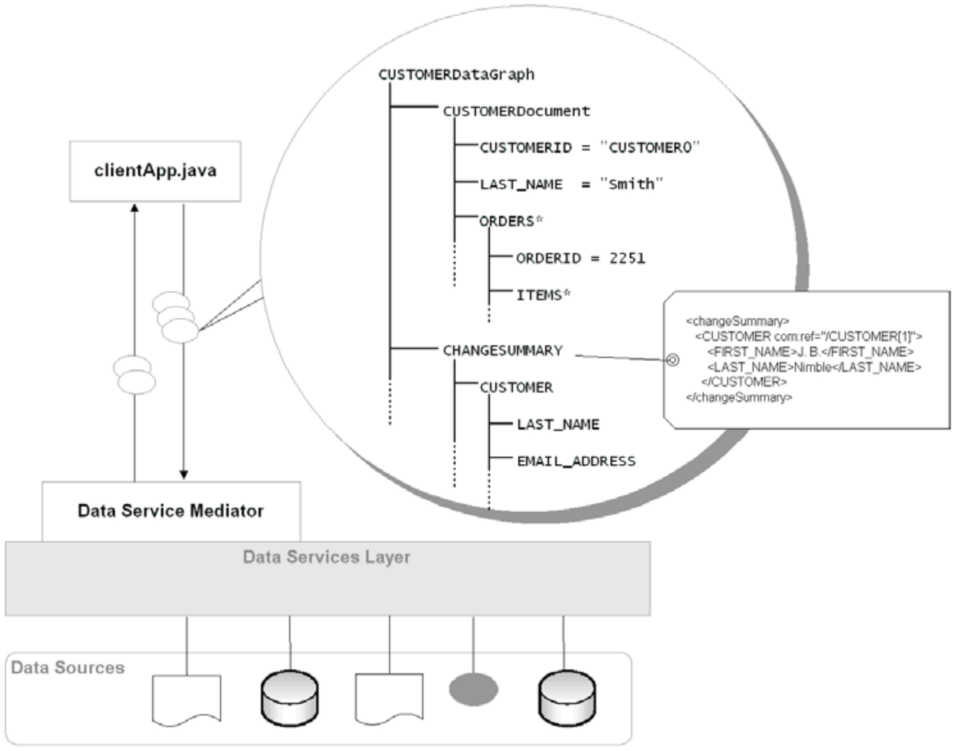
Data objects represent the contents of a complex type. A data object contains properties, which represent elements and attributes. The properties can be of simple or complex types. In SDO, a simple type property is called a datatype property, while a complex type property contains a data object (which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

Figure 2–1 Structure of a DataGraph



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

2.2.1 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 2–2.
- **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 2–1 summarizes the advantages of each approach.

Table 2–1 Static and Dynamic Data Object APIs

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

2.2.1.1 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 WebLogic Workshop, or by using one of the provided tools.

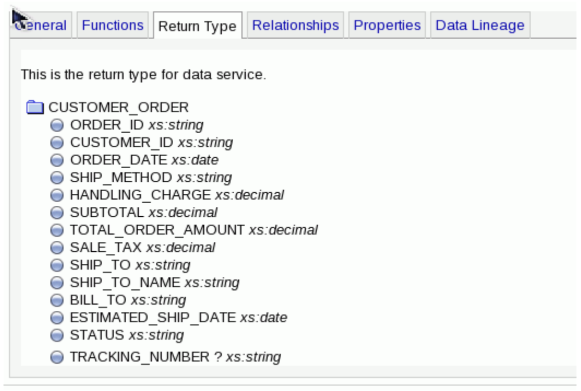
The generated interfaces extend the commonj.sdo.DataObject interface and provide typed getters and setters for all properties of the XML datatype.

An interface is also generated for each complex property (such as CREDIT and ORDER shown in Figure 2–2), 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 maxOccurs 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 2–2, 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).

Figure 2–2 CUSTOMER Return Type Displayed in Oracle Data Service Integrator Administration Console's Metadata Browser



This is the return type for data service.

```
CUSTOMER_ORDER
    ORDER_ID xs:string
    CUSTOMER_ID xs:string
    ORDER_DATE xs:date
    SHIP_METHOD xs:string
    HANDLING_CHARGE xs:decimal
    SUBTOTAL xs:decimal
    TOTAL_ORDER_AMOUNT xs:decimal
    SALE_TAX xs:decimal
    SHIP_TO xs:string
    SHIP_TO_NAME xs:string
    BILL_TO xs:string
    ESTIMATED_SHIP_DATE xs:date
    STATUS xs:string
    TRACKING_NUMBER ? xs:string
```

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:

```
import appDataServices.AddressDocument;
```

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

Static Data Object API (Generated)	Description	Examples
`Type get`*PropertyName*`()`	Returns the value of the property. Generated for boolean-valued properties.	`String name = getLAST_NAME()`
`List<Type> get`*PropertyName*`()`	For multiple occurrence elements, returns all *PropertyName* elements.	`List<ORDER> orders = getORDER()`
`void set`*PropertyName*`(Type newValue)`	Sets the value of the property to the newValue.	`setLAST_NAME("Smith")`
`boolean is`*PropertyName*`()`	Determines whether the *PropertyName* element or attribute exists in the data object.	`isSPECIAL_DELIVERY()`
`void create`*PropertyName*`()`	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.	`createORDER()`
`boolean isSet`*PropertyName*`()`	Determines whether the property is set to some value.	`isSetLAST_NAME()`
`void unset`*PropertyName*`()`	Unsets the property. The property is then considered not to be set.	`unsetLAST_NAME()`

2.2.1.2 XML Schema-to-Java Type Mapping Reference

Oracle Data Service Integrator client application developers can use the Oracle Data Service Integrator Administration Console to view the XML schema types associated with data services (see Figure 2–2). 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 2–3.

Note: The following XQuery types are discussed by the SDO specification and are listed in Table 2–3 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 2–3 XML Schema to Java Data Type Mapping

XML Schema Type	SDO Java Type
xs:anyType	commonj.sdo.DataObject
xs:anySimpleType	java.lang.Object
xs:anyURI	String
xs:base64Binary	byte[]
xs:boolean	boolean or java.lang.Boolean
xs:byte	byte or java.lang.Byte
xs:date	String
xs:dateTime	String
xs:decimal	java.math.BigDecimal
xs:double	double or java.lang.Double
xs:duration	String
xs:ENTITIES	List<String>
xs:ENTITY	String
xs:float	float or java.lang.Float
xs:gDay	String
xs:gMonth	String
xs:gMonthDay	String
xs:gYear	String
xs:gYearMonth	String
xs:hexBinary	byte[]
xs:ID	String
xs:IDREF	String
xs:IDREFS	List<String>
xs:int	int or java.lang.Integer
xs:integer	java.math.BigInteger

Table 2–3 (Cont.) XML Schema to Java Data Type Mapping

XML Schema Type	SDO Java Type
xs:language	String
xs:long	long or java.lang.Long
xs:Name	String
xs:NCName	String
xs:negativeInteger	java.math.BigInteger
xs:NMTOKEN	String
xs:NMTOKENS	List<String>
xs:nonNegativeInteger	java.math.BigInteger
xs:nonPositiveInteger	java.math.BigInteger
xs:normalizedString	String
xs:NOTATION	String
xs:positiveInteger	java.math.BigInteger
xs:QName	String
xs:short	short or java.lang.Short
xs:string	String
xs:time	String
xs:token	String
xs:unsignedByte	short or java.lang.Short
xs:unsignedInt	long or java.lang.Long
xs:unsignedLong	java.math.BigInteger
xs:unsignedShort	int or java.lang.Integer
xs:keyref	String

2.2.1.3 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 2–4 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" at http://www.osoa.org/display/Main/Service+Data+Objects+Specifications.

See also Service Data Objects at http://www.oracle.com/technology/pub/articles/dev2arch/2005/11/sdo.html.

Table 2–4 lists dynamic Data Object API getters and setters.

Table 2–4 Dynamic (Untyped) Data Object API Getters and Setters

Dynamic Data Object API	Description	Example
`get(int PropertyIndex)`	Returns the *PropertyName* child element at the specified index.	`get(5)`
`set(int PropertyIndex, Object newValue)`	Sets the value of the property to the newValue.	`set(5, CUSTOMER3)`
`set(String PropertyName, Object newValue)`	Sets the value of the PropertyName to the newValue.	`set("LAST_NAME", "Nimble")`
`set(commonj.sdo.Property property, Object newValue)`	Sets the value of Property object to the newValue.	`set(LASTNAME, "Nimble")`
`getType(String PropertyName)`	Returns the value of the *PropertyName*. *Type* indicates the specific data type to obtain.	`getBigDecimal("CreditScore")`
`unset(int PropertyIndex)`	Unsets the property. The property is then considered not to be set.	`unset(5)`
`unset(commonj.sdo.Property property)`	Unsets the property. The property is then considered not to be set.	`unset(LASTNAME)`
`unset(String PropertyName)`	Unsets the property. The property is then considered not to be set.	`unset("LAST_NAME")`
`createDataObject(commonj.sdo.Property property)`	Returns a new DataObject for the specified containment Property.	`createDataObject(LASTNAME)`
`createDataObject(String PropertyName)`	Returns a new DataObject for the specified containment property.	`createDataObject("LAST_NAME")`
`createDataObject(int PropertyIndex)`	Returns a new DataObject for the specified containment property.	`createDataObject(5)`
`createDataObject(String PropertyName, String namespaceURI, String typeName)`	Returns a new DataObject for the specified containment property.	`createDataObject("LAST_NAME","http://namespaceURI_here", "String")`
`delete()`	Removes the object from its container and unsets all writeable properties.	`delete(CUSTOMER)`

XPath Expressions in the Dynamic Data Object API

Oracle Data Service Integrator 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 2–5 compares the XPath standard and SDO augmented notations to refer to the same element, the first ORDER child node under CUSTOMER (Table 2–5).

Table 2–5 XPath Standard and SDO Augmented Notation

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

Zero-based indexing is convenient for Java programmers who are accustomed to zero-based counters, and may want to use counter values as index values without adding 1. Oracle Data Service Integrator fully supports both the traditional index notation and the augmented notation.

Keep in mind these other points regarding Oracle Data Service Integrator 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 Oracle Data Service Integrator 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 Section 3.9.4, "Specifying XPath Expressions as Arguments."

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 2–6) and the Property interface (see Table 2–7) comprise a minimal metadata API that can be used for introspecting the model of data objects. For example, the following obtains a data object's type and prints a property's value:

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

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

Example 2–1 Using SDO's Type Interface to Obtain Data Object Properties

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

Table 2–6 lists other useful methods in the Type interface.

Table 2–6 Type Interface Methods

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

Table 2–7 lists the methods of the Property interface.

Table 2–7 Property Interface Methods

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

2.3 Role of the Mediator API and SDO

In Oracle Data Service Integrator, 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 the Oracle Data Service Integrator update framework, including the Mediator, and how the default update process works, are described in more detail in "Managing Update Maps" in the *Oracle Fusion Middleware Data Service Integrator Developer's Guide*.

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

- Chapter 3, "Invoking Data Services from Java Clients"
- Chapter 4, "Invoking Data Services Through Web Services"

3 Invoking Data Services from Java Clients

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:

- Section 3.1, "Introducing the Mediator API"
- Section 3.2, "Getting Started"
- Section 3.3, "Sample Static Mediator Application"
- Section 3.4, "Sample Dynamic Mediator Application"
- Section 3.5, "Creating New DataObjects"
- Section 3.6, "Mediator API Basics"
- Section 3.7, "Mapping Data Service Types to Java Types"
- Section 3.8, "Web Services Support"
- Section 3.9, "Advanced Topics"

3.1 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:

- Section 3.1.1, "What is SDO?"
- Section 3.1.2, "What is the Mediator API?"
- Section 3.1.3, "Dynamic and Static Mediator APIs"

- Section 3.1.4, "API Overview"
- Section 3.1.5, "Summary"

3.1.1 What is SDO?

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

Note: See Section 1.5, "Introducing Service Data Objects (SDO)" for a general overview of SDO. For a more in-depth discussion of SDO, see Chapter 2, "Data Programming Model and Update Framework." Finally, see the dev2dev article Service Data Objects at `http://www.oracle.com/technology/pub/articles/dev2arch/2005/11/sdo.html`, which provides links to the SDO specifications and Javadoc.

3.1.2 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 Oracle Data Service Integrator 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 Section 2.2, "Oracle Data Service Integrator and SDO" for a general overview of SDO data objects and other artifacts.

3.1.3 Dynamic and Static Mediator APIs

The Oracle Data Service Integrator 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 Section 3.4, "Sample Dynamic Mediator Application."
- 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 Section 3.3, "Sample Static Mediator Application."

Note: 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.

3.1.4 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 3–1 Oracle Data Service Integrator Mediator API

Interface or Class Name	Description
DataAccessService	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`)
DASResult	The Mediator APIs that return data sets return an object called DASResult (Data Access Service Result). DASResult is similar to a Java Iterator. See Section 3.6.5, "Understanding DASResult." (Package: `com.bea.dsp.das`)
PreparedExpression	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 Section 3.9.5, "Making Ad Hoc Queries." (Package: `com.bea.dsp.das`)
DataAccessServiceFactory	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`)
HelperContextCache	Oracle Data Service Integrator 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 Section 3.5, "Creating New DataObjects." (Package: `com.bea.dsp.das`)
RequestConfig	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`)
SDOUtil	This utility class contains methods for manipulating SDO data objects in the context of Oracle Data Service Integrator. 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`)

Sample of both Static and Dynamic Mediator clients applications are provided. See Section 3.3, "Sample Static Mediator Application" and Section 3.4, "Sample Dynamic Mediator Application."

3.1.5 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 Oracle Data Service Integrator 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" at `http://www.osoa.org/display/Main/Service+Data+Objects+Specifications`.

The mediator APIs, on the other hand, are Oracle Data Service Integrator-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 Oracle Data Service Integrator uses SDO, see Chapter 2, "Data Programming Model and Update Framework."

3.2 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:

- Section 3.2.1, "Basic Steps"
- Section 3.2.2, "Setting the CLASSPATH"
- Section 3.2.3, "Running the Sample Applications"

3.2.1 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.

   Note: You can discover data services that are available to you by using the Oracle Data Service Integrator Console. See Viewing Metadata Using the Service Explorer in the *Oracle Fusion Middleware Administering Data Service Integrator* guide.

2. Decide whether to use the Static or Dynamic Mediator API to interact with the data service from your Java client. See Section 3.1.3, "Dynamic and Static Mediator APIs" 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 *Oracle Fusion Middleware Data Service Integrator Developer's Guide*.

   Note: 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 Section 3.2.2, "Setting the CLASSPATH" 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 Section 3.2.3, "Running the Sample Applications."

3.2.2 Setting the CLASSPATH

You can set the CLASSPATH by either adding the Oracle Data Service Integrator client library to the project or by manually setting the CLASSPATH.

3.2.2.1 Adding the Oracle Data Service Integrator Client Library

You can add the Oracle Data Service Integrator client library to your project by doing either of the following:

- Adding the library to an existing project
- Adding the library when creating a new project

Adding the Library to an Existing Project

You can add the Oracle Data Service Integrator client library to an existing project.

Complete the following steps:

1. Right-click the project and choose **Properties**. A dialog showing the properties for the project appears.
2. Select **Java Build Path**.
3. Click the **Libraries** tab, and click **Add Library**.
4. Select **Oracle Data Service Integrator** client library, click **Next**, and click **Finish**.

Alternatively, you can do the following:

1. Right-click the project and choose **Build Path > Configure Build Path**. A dialog showing the properties for the project appears.
2. Select **Java Build Path**.
3. Click the **Libraries** tab, and click **Add Library**.
4. Select **Oracle Data Service Integrator** client library, click **Next**, and click **Finish**.

Adding the Library When Creating a New Project

You can add the Oracle Data Service Integrator client library when creating a new Java project.

Complete the following steps:

1. Right-click in the Project Explorer, and choose **New > Project**. The New Project wizard appears.
2. Select **Java Project** and click **Next**.
3. Type a name for the project and click **Next**.
4. Click the **Libraries** tab, and click **Add Library**.
5. Select **Oracle Data Service Integrator** client library, click **Next**, and click **Finish**.
6. Click **Finish** to create the new project.

3.2.2.2 Manually Setting the CLASSPATH

You can optionally set the CLASSPATH manually, if required. The CLASSPATH settings depend on whether you are using the Static or Dynamic Mediator API.

Note: You can use the Java Mediator API with either the weblogic.jar or the wlfullclient.jar file. For more information about choosing between `weblogic.jar` or `wlfullclient.jar`, see "Overview of Stand-alone Clients" in the Oracle WebLogic Server documentation. For more information about creating the wlfullclient.jar file, see "Using the WebLogic JarBuilder Tool" in the Oracle WebLogic Server documentation.

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.

Example 3–1 Static Java Mediator API Client Classpath (with weblogic.jar)

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

Example 3–2 Static Java Mediator API Client Classpath (with wlfullclient.jar)

```
CLASSPATH=
   <ODSI_HOME>/lib/ld-client.jar
   <ODSI_HOME>/lib/sdo.jar
   <WL_HOME>/modules/com.bea.core.xml.xmlbeans_2.4.0.0_2-5-1.jar
   <WL_HOME>/server/lib/wlfullclient.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.

Example 3–3 Dynamic Mediator API Classpath (with weblogic.jar)

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

Example 3–4 Dynamic Mediator API Classpath (with wlfullclient.jar)

```
CLASSPATH=
   <ODSI_HOME>/lib/ld-client.jar
   <ODSI_HOME>/lib/sdo.jar
   <WL_HOME>/modules/com.bea.core.xml.xmlbeans_2.4.0.0_2-5-1.jar
   <WL_HOME>/server/lib/wlfullclient.jar
```

3.2.2.3 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 Example 3–5 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.

Example 3–5 Example Code: Getting and Setting the Class Loader

```
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 3–5, you could use the following code to obtain a PreparedExpression object: `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).
- *`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.

3.2.3 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 Section 3.3, "Sample Static Mediator Application" and Section 3.4, "Sample Dynamic Mediator Application."

3.3 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 Section 3.1.3, "Dynamic and Static Mediator APIs." See also Section 3.6, "Mediator API Basics" and Section 3.9, "Advanced Topics."

Topics include:

- Section 3.3.1, "Setting Up the Sample Data Service"

- Section 3.3.2, "Generating the Mediator Client JAR File"
- Section 3.3.3, "Setting Up the Java Project"
- Section 3.3.4, "Running and Testing the Code"
- Section 3.3.5, "Examining the Sample Code"

3.3.1 Setting Up the Sample Data Service

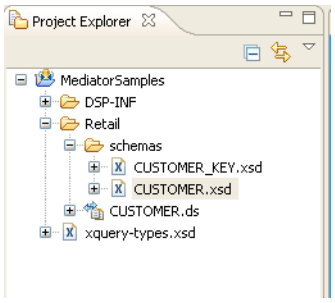
Before you can build and test the sample Java application, you need to set up an Oracle Data Service Integrator data service. The instructions assume that you are familiar with the Oracle Data Service Integrator perspective in the Eclipse IDE, as described in the *Oracle Fusion Middleware Data Service Integrator 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 Oracle Data Service Integrator.
2. In Workshop for WebLogic, create a server that uses the Oracle Data Service Integrator samples domain.
3. Start the server.
4. Create an Oracle Data Service Integrator dataspace called **MediatorSamples**.
5. Copy the sample data service (Example 3–6) into a file called `MediatorSamples/Retail/CUSTOMER.ds.`
6. Copy the schema file (Example 3–7) into a file called `MediatorSamples/Retail/schemas/CUSTOMER_KEY.xsd`.
7. Copy the schema file (Example 3–8) into a file called `MediatorSamples/Retail/schemas/CUSTOMER.xsd`.

Figure 3–1 shows the resulting Dataspace configuration:

Figure 3–1 Sample Dataspace Configuration



Note: Example 3–6 is a simple data service file, containing the XQuery code that defines the service and its operations. Example 3–7 and Example 3–8 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 *Oracle Fusion Middleware Data Service Integrator Developer's Guide*.

Example 3–6 CUSTOMER.ds

```
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"/>
    </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_LETTTER" type="xs:short">
            <extension nativeXpath="NEWS_LETTTER" 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>
        <key name="CUSTOMER_0_SYSTEMNAMEDCONSTRAINT__PRIMARYKEY"
        type="cus:CUSTOMER_KEY" inferredSchema="true"
        xmlns:cus="ld:Retail/CUSTOMER"/>
    </x:xds>::)

    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:deleteCUSTOMER($p as element(t1:CUSTOMER)*) as empty()
       external;
```

Example 3–7 CUSTOMER_KEY.xsd

```
<?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>
```

Example 3–8 CUSTOMER.xsd

```
<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="NEWS_LETTTER" type="xs:short" minOccurs="0"/>
            <xs:element name="ONLINE_STATEMENT" type="xs:short" minOccurs="0"/>
            <xs:element name="LOGIN_ID" type="xs:string" minOccurs="0"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

3.3.2 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.

Note: You can generate a Mediator Client JAR file using the IDE, the Oracle Data Service Integrator Console, or an Ant script. These methods are described in detail in the *Oracle Fusion Middleware Data Service Integrator 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 **Oracle Data Service Integrator > 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**.

Note: For detailed information on how generated class names in the JAR file are derived, see Section 3.6.4, "Naming Conventions for Generated Classes."

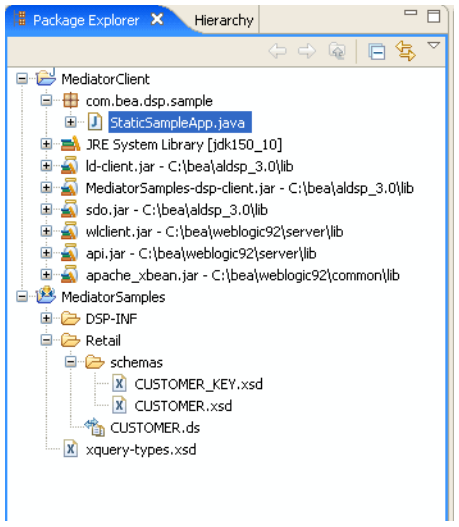
3.3.3 Setting Up the Java Project

Example 3–10 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 Section 3.2.2, "Setting the CLASSPATH." To do this, select **Project > Properties > Java Build Path**. Be sure to include the Mediator Client JAR file, as discussed in Section 3.3.2, "Generating the Mediator Client JAR File."
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 Example 3–9 into the source file.
6. Save the file. Figure 3–2 shows the completed project configuration in the Eclipse IDE.

Figure 3–2 Completed Project Configuration



```
MediatorClient
  com.bea.dsp.sample
    StaticSampleApp.java
  JRE System Library [jdk150_10]
  ld-client.jar - C:\bea\aldsp_3.0\lib
  MediatorSamples-dsp-client.jar - C:\bea\aldsp_3.0\lib
  sdo.jar - C:\bea\aldsp_3.0\lib
  wlclient.jar - C:\bea\weblogic92\server\lib
  api.jar - C:\bea\weblogic92\server\lib
  apache_xbean.jar - C:\bea\weblogic92\common\lib
MediatorSamples
  DSP-INF
  Retail
    schemas
      CUSTOMER_KEY.xsd
      CUSTOMER.xsd
    CUSTOMER.ds
  xquery-types.xsd
```

Note: The imported classes CUSTOMERDAS and CUSTOMER (see Example 3–9) are located in the Static Mediator Client JAR file, which must be in the CLASSPATH.

Example 3–9 StaticSampleApp.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();
        }
    }
}
```

3.3.4 Running and Testing the Code

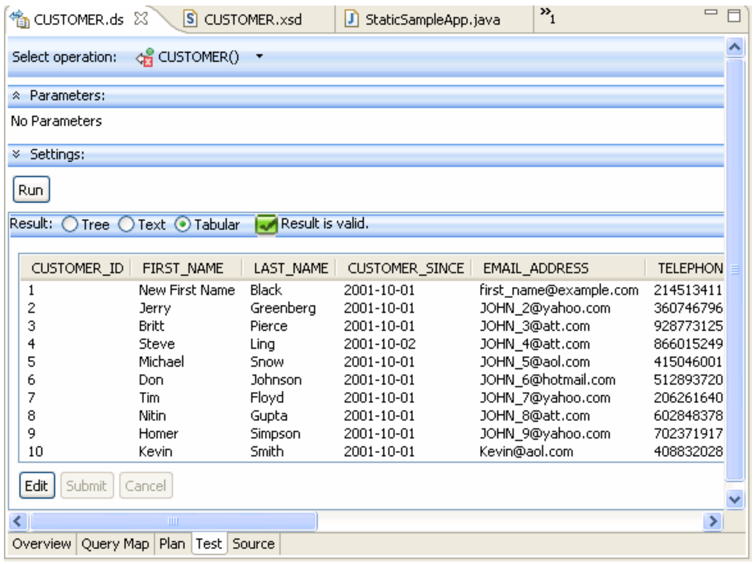
To test the application:

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 3–3).
3. Select an operation from the drop down menu. For this example, select the **CUSTOMER()** operation.
4. Click Run (see Figure 3–3).
5. Inspect the first row of the data table. The client application changes the first customer's name and email address to "N**ew First Name**" and "**first_name@example.com**" as shown in Figure 3–3.

Figure 3–3 Testing the Client



Select operation: CUSTOMER()

Parameters:

No Parameters

Settings:

Result: ◯ Tree ◯ Text [✓ Tabular] Result is valid.

CUSTOMER_ID	FIRST_NAME	LAST_NAME	CUSTOMER_SINCE	EMAIL_ADDRESS	TELEPHON
1	New First Name	Black	2001-10-01	first_name@example.com	214513411
2	Jerry	Greenberg	2001-10-01	JOHN_2@yahoo.com	360746796
3	Britt	Pierce	2001-10-01	JOHN_3@att.com	928773125
4	Steve	Ling	2001-10-02	JOHN_4@att.com	866015249
5	Michael	Snow	2001-10-01	JOHN_5@aol.com	415046001
6	Don	Johnson	2001-10-01	JOHN_6@hotmail.com	512893720
7	Tim	Floyd	2001-10-01	JOHN_7@yahoo.com	206261640
8	Nitin	Gupta	2001-10-01	JOHN_8@att.com	602848378
9	Homer	Simpson	2001-10-01	JOHN_9@yahoo.com	702371917
10	Kevin	Smith	2001-10-01	Kevin@aol.com	408832028

3.3.5 Examining the Sample Code

This section examines the parts of the Java sample in Example 3–9. This section discusses:

- Section 3.3.5.1, "Importing Packages"
- Section 3.3.5.2, "Obtaining a Data Access Service Handle"
- Section 3.3.5.3, "Retrieving Data from the Service"
- Section 3.3.5.4, "Obtaining a DataObject from the Result"
- Section 3.3.5.5, "Disposing the Result Object"
- Section 3.3.5.6, "Modifying the DataObject"
- Section 3.3.5.7, "Returning Changes to the Server"

3.3.5.1 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.

```
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;
```

3.3.5.2 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 Section 3.6.6, "Obtaining the WebLogic JNDI Context for Oracle Data Service Integrator."
- The name of the Dataspace project in which the data service is deployed. In this sample, the project is called MediatorSamples.

```
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");
```

3.3.5.3 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 Section 3.6.5, "Understanding DASResult."

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

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:

```
(::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;
```

The entire data service file is shown in Example 3–6.

3.3.5.4 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 Oracle Fusion Middleware Using Data Service Integrator XQuery Engine *Oracle Data Service Integrator Concepts Guide*.

```
CUSTOMER customer = result.next();
```

3.3.5.5 Disposing the Result Object

You must call DASResult.dispose() whenever you are finished iterating through a result object. For more information on dispose(), see Section 3.6.5.2, "Disposing of DASResult Objects."

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

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

3.3.5.6 Modifying the DataObject

After you obtain a DataObject, you can modify it; however, if you intend to submit these changes back to the Oracle Data Service Integrator 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 Section 3.6.7, "Working with Data Objects." After the customer object has change-tracking enabled, the generated setters are called to modify certain values in the customer object.

Note: 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");
```

3.3.5.7 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 });
```

Note: In this example, the update method generated by Oracle Data Service Integrator 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.

3.4 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:

- Section 3.4.1, "Setting Up and Running the Sample Code"
- Section 3.4.2, "Sample Java Client Code (Dynamic Mediator API)"
- Section 3.4.3, "Examining the Sample Code"

3.4.1 Setting Up and Running the Sample Code

To set up and run this sample code, follow the basic instructions in Section 3.3, "Sample Static Mediator Application." 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 Example 3–10 in your project.

3.4.2 Sample Java Client Code (Dynamic Mediator API)

This section shows sample Java client code for the Dynamic Mediator API.

Example 3–10 DynamicSampleApp.java

```
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,"welcome1");
        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
            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();
            }
        }
    }
```

3.4.3 Examining the Sample Code

This section examines the parts of the Java sample in Example 3–10. This section discusses:

- Section 3.4.3.1, "Importing Classes"
- Section 3.4.3.2, "Obtaining a DataAccessService Handle"
- Section 3.4.3.3, "Retrieving Data from the Service"
- Section 3.4.3.4, "Obtaining a DataObject from the Result"
- Section 3.4.3.5, "Disposing the Result Object"
- Section 3.4.3.6, "Modifying the DataObject"
- Section 3.4.3.7, "Returning Changes to the Server"

3.4.3.1 Importing Classes

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

```
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;
```

3.4.3.2 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 Section 3.6.6, "Obtaining the WebLogic JNDI Context for Oracle Data Service Integrator." For more information on WebLogic JNDI context objects, see Oracle Fusion Middleware Programming JNDI for Oracle WebLogic Server.
- 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:

```
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");
```

3.4.3.3 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 Section 3.6.5, "Understanding DASResult." Note that the CUSTOMER operation takes no arguments.

```
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.

3.4.3.4 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. See also Section 3.1.1, "What is SDO?."

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

3.4.3.5 Disposing the Result Object

You must call DASResult.dispose() whenever you are finished iterating through a result object. For more information on dispose(), see Section 3.6.5.2, "Disposing of DASResult Objects."

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

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

3.4.3.6 Modifying the DataObject

After you obtain a DataObject, you can modify it; however, if you intend to submit these changes back to the Oracle Data Service Integrator 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 Section 3.6.7, "Working with Data Objects." 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."

```
SDOUtil.enableChanges(customer);

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

3.4.3.7 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.

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

Note: 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.

3.5 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 Oracle Data Service Integrator server. As with previous examples, both the static and dynamic APIs are illustrated.

3.5.1 Creating a New DataObject with the Static API

The Java program in Example 3–11 creates a new DataObject, modifies it, and updates it on the Oracle Data Service Integrator server.

3.5.1.1 Setting Up and Running the Sample

The sample code in Example 3–11 is designed to work in the same Java project and with the same data service project that are described in Section 3.3, "Sample Static Mediator Application."

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.

Example 3–11 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();
        }

        // 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.
    }
}
```

3.5.1.2 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.

3.5.1.3 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 Section 3.6.6, "Obtaining the WebLogic JNDI Context for Oracle Data Service Integrator." For more information on WebLogic JNDI context objects, see Oracle Fusion Middleware Programming JNDI for Oracle WebLogic Server.

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

  ```
  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");
  ```

3.5.1.4 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");
```

3.5.1.5 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.

Note that this name is the XML schema name, not the type. However, as noted, for global elements that do not specify a type, the type that is provided has the same name as the element.

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

3.5.1.6 Modifying the DataObject

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

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 Oracle Data Service Integrator server.

```
customer.setFIRST_NAME("New First Name");
customer.setLAST_NAME("New Last Name");
customer.setCUSTOMER_SINCE("2007-10-18T12:27:41Z");
customer.setEMAIL_ADDRESS("first_name@example.com");
customer.setTELEPHONE_NUMBER("867-5309");
```

Note: You can omit optional fields or fields with default values.

3.5.1.7 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.

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

3.5.1.8 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.

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

finally {
   result.dispose();
}
```

Note: 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.

3.5.2 Creating a New DataObject with the Dynamic API

The Java program in Example 3–11 creates a new DataObject, modifies it, and updates it on the Oracle Data Service Integrator server.

3.5.2.1 Running the Sample

The sample code in Example 3–12 is designed to work in the same Java project and with the same data service project that are described in Section 3.4, "Sample Dynamic Mediator Application." You can run the sample code presented here by following the setup instructions in that section.

Example 3–12 DynamicCreateSample.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", "Dynammic");
        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.
    }
}
```

3.5.2.2 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 Oracle Data Service Integrator. The use of HelperContextCache is described in the next section Section 3.5.2.3, "Creating a DataFactory."

3.5.2.3 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 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();
```

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

3.5.2.4 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.

Note that this name is the XML schema name, not the type. However, as noted, for global elements that do not specify a type, the type that is provided has the same name as the element.

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

3.5.2.5 Modifying the DataObject

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

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 Oracle Data Service Integrator server.

```
customer.set("FIRST_NAME", "New First Name");
customer.set("LAST_NAME", "New Last Name");
customer.set("CUSTOMER_SINCE", "2007-10-18T12:27:41Z");
customer.set("EMAIL_ADDRESS", "first_name@example.com");
customer.set("TELEPHONE_NUMBER", "867-5309");
```

Note: You can omit optional fields or fields with default values.

3.5.2.6 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 Section 3.6.6, "Obtaining the WebLogic JNDI Context for Oracle Data Service Integrator." For more information on WebLogic JNDI context objects, see Oracle Fusion Middleware Programming JNDI for Oracle WebLogic Server.
- 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.

The DataAccessServiceFactory returns the handle.

```
// 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 CUTOMER_KEYS objects in the DASResult object.

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

3.5.2.7 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.

```
try {
   DataObject key = (DataObject) result.next();
      System.out.println("New customer key: " + key.get("CUSTOMER_ID"));
}
finally {
   result.dispose();
}
```

Note: 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.

3.6 Mediator API Basics

This section discusses various Mediator API topics.

- Section 3.6.1, "Beyond the Sample Applications"
- Section 3.6.2, "More on the Static Mediator API"
- Section 3.6.3, "More on the Dynamic Mediator API"
- Section 3.6.4, "Naming Conventions for Generated Classes"

- Section 3.6.5, "Understanding DASResult"
- Section 3.6.6, "Obtaining the WebLogic JNDI Context for Oracle Data Service Integrator"
- Section 3.6.7, "Working with Data Objects"

3.6.1 Beyond the Sample Applications

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

- Section 3.3, "Sample Static Mediator Application"
- Section 3.4, "Sample Dynamic Mediator Application"

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.

3.6.2 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 Section 3.6.5, "Understanding DASResult."

3.6.3 More on the Dynamic Mediator API

This section provides additional information on the Dynamic Mediator API.

- Section 3.6.3.1, "Invoking Data Service Operations"
- Section 3.6.3.2, "Getters and Setters"

3.6.3.1 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 Section 3.6.5, "Understanding DASResult."

You can see the invoke() method in use in the Dynamic Mediator API sample in Example 3–10:

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

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

3.6.3.2 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:

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

or

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

See also Section 3.9.4, "Specifying XPath Expressions as Arguments" and Chapter 2, "Data Programming Model and Update Framework."

3.6.4 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:

3.6.4.1 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.

3.6.4.2 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.

3.6.5 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:

- Section 3.6.5.1, "Overview of DASResult"
- Section 3.6.5.2, "Disposing of DASResult Objects"
- Section 3.6.5.3, "Dynamic Mediator APIs and DASResult"
- Section 3.6.5.4, "Static Mediator APIs and DASResult"
- Section 3.6.5.5, "Retrieving an Array of Objects"

3.6.5.1 Overview of DASResult

By default, data is returned to the Mediator from the Oracle Data Service Integrator server in small blocks. This "streaming" behavior means that large result sets are never held in memory all at once on either the Oracle Data Service Integrator 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 Section 3.9.2, "Support for Stateless Operations."

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

```
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 Section 3.7, "Mapping Data Service Types to Java Types." See also Section 3.9.5, "Making Ad Hoc Queries."

Note: 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.

3.6.5.2 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 Oracle Data Service Integrator server.
- boolean isDisposed() – Returns whether the connection to the Oracle Data Service Integrator 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 Section 3.9.2, "Support for Stateless Operations."
- You use the DASResult.getItems() method. For more information, see Section 3.6.5.5, "Retrieving an Array of Objects."

3.6.5.3 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 Section 3.1.4, "API Overview" for information on RequestConfig.
- All forms of PreparedExpression.executeQuery() for ad-hoc queries. For details on forming ad-hoc queries, see Section 7.2.4, "Using Ad Hoc Queries to Fine-tune Results from the Client."

3.6.5.4 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 Section 3.7, "Mapping Data Service Types to Java Types" for more information.

3.6.5.5 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 Section 3.6.5.2, "Disposing of DASResult Objects."

- 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.

3.6.6 Obtaining the WebLogic JNDI Context for Oracle Data Service Integrator

Java client applications use JNDI to access named objects, such as data services, on a Oracle WebLogic Server. To use any of the Mediator APIs, you need to obtain the WebLogic Server JNDI context for Oracle Data Service Integrator.

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 Oracle Fusion Middleware Programming JNDI for Oracle WebLogic Server.

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

InitialContext jndiCtxt = new InitialContext(hashtable);

Table 3–2 lists the keys and values that you can insert into the hashtable parameter.

Table 3–2 JNDI Context Keys and Values

Key	Value
Context.INITIAL_CONTEXT_FACTORY	weblogic.jndi.WLInitialContextFactory
Context.PROVIDER_URL	URL of the WebLogic Server hosting Oracle Data Service Integrator. For example: `t3://localhost:7001`.
Context.SECURITY_PRINCIPAL	(optional) A username
Context.SECURITY_CREDENTIALS	(optional) A password

Example 3–13 shows example code for obtaining the JNDI context.

Example 3–13 Obtaining the JNDI Context

```
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);
```

3.6.7 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 Section 3.6.5, "Understanding DASResult.")

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."

3.6.7.1 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:

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

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

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

Note: 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.

3.6.7.2 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 Section 2.2.1.1, "Static Data Object API" and Section 2.2.1.3, "Dynamic Data Object API."

Example static (typed) method call:

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

Example dynamic (untyped) method call:

```
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. Oracle Data Service Integrator then handles the details of performing the update. For example, from Example 3–9:

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

3.6.7.3 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 Section 3.5, "Creating New DataObjects."

3.7 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:

- Section 3.7.1, "Conversion of Simple Types"
- Section 3.7.2, "Conversion of Date/Time Types"
- Section 3.7.3, "Passing Empty Sequence Arguments"
- Section 3.7.4, "Quantified Return Types"
- Section 3.7.5, "What is Autoboxing?"

3.7.1 Conversion of Simple Types

Table 3–3 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 Section 3.6.5.4, "Static Mediator APIs and DASResult.")

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

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 3–3 Simple XQuery to Java Type Conversion

XQuery Type	Mediator Accepts	Mediator Returns
xs:boolean	Boolean	Boolean
xs:byte	Byte	Byte

Table 3–3 (Cont.) Simple XQuery to Java Type Conversion

xs:short	Short	Short
xs:int	Integer	Integer
xs:long	Long	Long
xs:integer	BigInteger	BigInteger
xs:negativeInteger xs:positiveInteger xs:nonNegativeInteger xs:nonPositiveInteger	BigInteger	BigInteger
xs:unsignedByte	Short	Short
xs:unsignedShort	Integer	Integer
xs:unsignedInt	Long	Long
xs:unsignedLong	BigInteger	BigInteger
xs:float	Float	Float
xs:double	Double	Double
xs:decimal	BigDecimal	BigDecimal
xs:string	String	String
xs:anyURI	String, java.net.URI	String
xs:base64Binary	byte[]	byte[]
xs:hexBinary	byte[]	byte[]
xs:QName	javax.xml.namespace.QName	String 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.

3.7.2 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 `http://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 3–4 XQuery Date/Time Types to Java Conversions

XQuery Type	Mediator Accepts	Mediator Returns
xs:date	String, java.sql.Date	String
xs:time	String, java.sql.Time	String

Table 3–4 (Cont.) XQuery Date/Time Types to Java Conversions

XQuery Type	Mediator Accepts	Mediator Returns
xs:dateTime	String, java.sql.Timestamp, java.util.Date, java.util.Calendar	String
xs:duration, xs:gDay, xs:gMonth, xs:gMonthDay, xs:gYear, xs:gYearMonth	String	String

3.7.3 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).

3.7.4 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, `xx:int*`) have static mediator methods that are declared to return DASResult<type>.

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

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

is converted to a method like this:

```
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.

3.7.5 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 Example 3–14, 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.)

Example 3–14 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();
```

3.7.6 Support for Derived Simple Types

Oracle Data Service Integrator enables you to employ user-derived simple types as parameters or return types in data service operations accessed through the Oracle Data Service Integrator Mediator API.

The Mediator API is the Java API for retrieving artifacts from a data service and returning them to their source. In your Java client, you can call Mediator API methods to connect to a data service and invoke data service operations.

This enables you to create a simple type in a schema which restricts a built-in schema type, and use that type as either a parameter type or a return type for a data service operation, successfully invoking the operation through the Oracle Data Service Integrator Mediator API.

For example, you might declare a simple type called ZipCode that derives from xsd:string, as shown in the following:

```
<xs:simpleType name="ZipCode">
   <xs:restriction base="xs:string">
      <xs:pattern value="[0-9]{5}(-[0-9]{4})?"/>
   </xs:restriction>
</xs:simpleType>
```

Oracle Data Service Integrator enables you to use this type as a parameter type or return type for a data service operation.

3.7.6.1 Mapping Derived Schema Types to Java Types

The mapping of derived schema types to Java types is defined by the SDO specification and, with one exception, is identical to the mapping of the corresponding built-in schema types. For example, if a user type is derived from xs:string, the Java class is java.lang.String. Similarly, if a user type is derived from xs:byte, the Java class is java.lang.Byte.

The single exception involves using XSD types that are derived from xs:integer and include facets (such as minInclusive, maxInclusive, minExclusive, maxExclusive, or enumeration) constraining the range to be within the range of the Java data type int. Oracle Data Service Integrator maps these XSD types to the Java type java.lang.Integer instead of the default type java.math.BigInteger.

Note that this is also true for user XSD types that derive from the following schema built-in types (which are themselves derived from xs:integer):

- xs:positiveInteger
- xs:negativeInteger
- xs:nonPositiveInteger
- xs:nonNegativeInteger
- xs:long
- xs:unsignedLong

Note: You cannot use types derived from xs:QName. Attempting to use these types in data service operations may cause an exception.

3.8 Web Services Support

The Oracle Data Service Integrator 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.

3.9 Advanced Topics

This section includes these topics:

- Section 3.9.1, "Schema Management"
- Section 3.9.2, "Support for Stateless Operations"
- Section 3.9.3, "Cache Management"
- Section 3.9.4, "Specifying XPath Expressions as Arguments"
- Section 3.9.5, "Making Ad Hoc Queries"

3.9.1 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.

3.9.1.1 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 Section 3.9.1.3, "Schema Cache Management" for more information on the HelperContext API.

3.9.1.2 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.

Note: 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.

3.9.1.3 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 Oracle Data Service Integrator 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.

- 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 DataAccesssService instance just to obtain the schemas to create DataObjects. (See also Section 3.5, "Creating New DataObjects".) 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.

3.9.2 Support for Stateless Operations

By default, the Mediator API holds resources on the Oracle Data Service Integrator server open while data is being returned. As discussed in Section 3.6.5, "Understanding DASResult," data objects are returned through DASResult one object at a time. Oracle Data Service Integrator 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 Oracle Data Service Integrator 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 Oracle Data Service Integrator 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 Oracle Data Service Integrator 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.

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.MaxMessageSizeExceededException. 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 Section 3.3.5.5, "Disposing the Result Object."

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.

3.9.3 Cache Management

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

3.9.3.1 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 *Oracle Fusion Middleware Administering Data Service Integrator* guide.) 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.setIntegerAttribute() method. The setting of REFRESH_CACHE_

EARLY to a particular value requires that a cached record must have at least *n* seconds of remaining TTL before it can be used.

If the record is set to expire in less than *n* seconds, it will not be retrieved. Instead its value is recalculated based on the underlying data and the data cache associated with that function is refreshed. The same REFRESH_CACHE_EARLY value applies to all cache operations during a query evaluation.

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

3.9.4 Specifying XPath Expressions as Arguments

Oracle Data Service Integrator 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:

```
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:

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

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

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

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

Oracle Data Service Integrator fully supports both the traditional index notation and the augmented notation. See the content on XPath expressions in Section 2.2.1.3, "Dynamic Data Object API" for details.

3.9.5 Making Ad Hoc Queries

The DataAccessServiceFactory.prepareExpression() method lets you create ad hoc queries against the data service. Example 3–15 shows an example of the preparedExpression() method. For more information on ad hoc queries, see Section 7.2.4, "Using Ad Hoc Queries to Fine-tune Results from the Client."

Example 3–15 Example of preparedExpression Method

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

```
int answer = (int) result.next();
result.dispose();
```

3.10 Understanding Transaction Behavior

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

3.10.1 Transaction Behavior for Read/Write Operations

The Oracle Data Service Integrator server always creates a new transaction if necessary when executing read/write operations, such as create, update, or delete.

3.10.2 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**.

```
RequestConfig config = new RequestConfig();
config.setEnumAttribute(RequestConfig.ReadTransactionMode.SUPPORTS);
```

4

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:

4.1 Overview

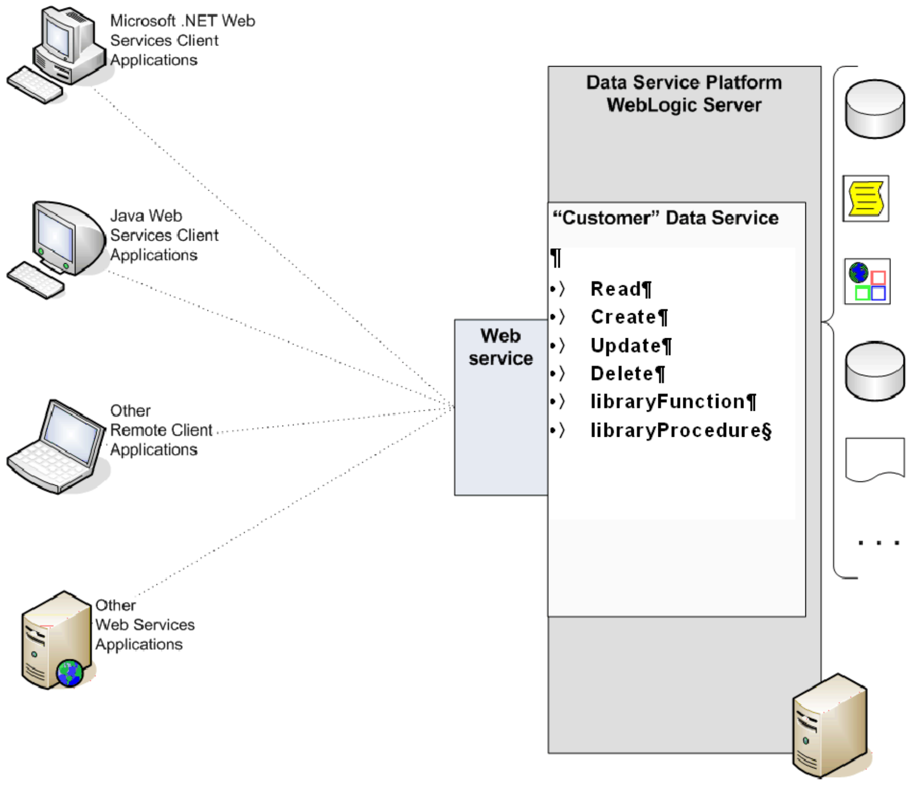
The Oracle Data Service Integrator Native Web Services feature lets you map data services to web services directly. Client applications access data through web services with the Data Services Mediator API.

Note: 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.

Note: For detailed information on creating data service operations (read, create, update, delete, libraryFunction, and libraryProcedure) see the *Oracle Fusion Middleware Data Service Integrator Developer's Guide*.

Figure 4–1 Web Services Enable Client Access to Oracle Data Service Integrator-Enabled Applications


Microsoft .NET Web Services Client Applications
Java Web Services Client Applications
Other Remote Client Applications
Other Web Services Applications
Web service
Data Service Platform WebLogic Server
"Customer" Data Service
¶
Read¶
Create¶
Update¶
Delete¶
libraryFunction¶
libraryProcedure§
. . .

4.2 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).

Note: If you are unfamiliar with web services, you can refer to the WebLogic Server document "WebLogic Web Services: Getting Started" at `http://download.oracle.com/docs/cd/E12840_01/wls/docs103/webserv/index.html`. 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 Oracle Data Service Integrator. See Chapter 2, "Data Programming

Model and Update Framework" for information on SDO.

- You are familiar with XML.

4.3 Getting Started

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

- Section 4.3.1, "Basic Steps"
- Section 4.3.2, "Setting the CLASSPATH"
- Section 4.3.3, "Running the Sample Applications"

4.3.1 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 *Oracle Fusion Middleware Data Service Integrator 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 Section 3.1.3, "Dynamic and Static Mediator APIs" 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 *Oracle Fusion Middleware Data Service Integrator Developer's Guide*.

   Note: 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 Section 4.3.2, "Setting the CLASSPATH" 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: Section 4.4, "Sample Static Mediator Application" and Section 4.5, "Sample Dynamic Mediator Application."

4.3.2 Setting the CLASSPATH

You can set the CLASSPATH by either adding the Oracle Data Service Integrator client library to the project or by manually setting the CLASSPATH.

4.3.2.1 Adding the Oracle Data Service Integrator Client Library

You can add the Oracle Data Service Integrator client library to your project by doing either of the following:

- Adding the library to an existing project
- Adding the library when creating a new project

Adding the Library to an Existing Project

You can add the Oracle Data Service Integrator client library to an existing project.

Complete the following steps:

1. Right-click the project and choose Properties. A dialog showing the properties for the project appears.
2. Select Java Build Path.
3. Click the Libraries tab, and click Add Library.
4. Select Oracle Data Service Integrator client library, click Next, and click Finish.

Alternatively, you can do the following:

1. Right-click the project and choose Build Path > Configure Build Path. A dialog showing the properties for the project appears.
2. Select Java Build Path.
3. Click the Libraries tab, and click Add Library.
4. Select Oracle Data Service Integrator client library, click Next, and click Finish.

Adding the Library When Creating a New Project

You can add the Oracle Data Service Integrator client library when creating a new Java project.

Complete the following steps:

1. Right-click in the Project Explorer, and choose New > Project. The New Project wizard appears.
2. Select Java Project and click Next.
3. Type a name for the project and click Next.
4. Click the Libraries tab, and click Add Library.
5. Select Oracle Data Service Integrator client library, click Next, and click Finish.
6. Click Finish to create the new project.

4.3.2.2 Manually Setting the CLASSPATH

You can optionally set the CLASSPATH manually, if required. The CLASSPATH settings depend on whether you are using the Static or Dynamic Mediator API.

Note: You can use the Java Mediator API with either the weblogic.jar or the wlfullclient.jar file. For more information about choosing between weblogic.jar or wlfullclient.jar, see Overview of Stand-alone Clients in the Oracle WebLogic Server documentation at `http://download.oracle.com/docs/cd/E12840_01/wls/docs103/client/basics.html`. For more information about creating the wlfullclient.jar file, see Using the WebLogic JarBuilder Tool at `http://download.oracle.com/docs/cd/E12840_01/wls/docs103/client/jarbuilder.html`.

Static Web Service Client CLASSPATH

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

Example 4–1 Static Web Service Client Classpath (with weblogic.jar)

```
CLASSPATH=
   <dataspace-ws-client>.jar <= this is the generated static client jar
                               for the webservices transport
   <ALDSP_HOME>/lib/ld-client.jar
   <BEA_HOME>/modules/com.bea.core.sdo_1.1.0.0.jar
   <WL_HOME>/server/lib/weblogic.jar
```

Example 4–2 Static Web Service Client Classpath (with wlfullclient.jar)

```
CLASSPATH=
   <dataspace-ws-client>.jar <= this is the generated static client jar
                               for the webservices transport
   <ALDSP_HOME>/lib/ld-client.jar
   <BEA_HOME>/modules/com.bea.core.sdo_1.1.0.0.jar
   <BEA_HOME>/modules/com.bea.core.xml.xmlbeans_1.0.0.0_2-4-0.jar
   <BEA_HOME>/modules/com.bea.core.xml.beaxmlbeans_1.0.0.0_2-4-0.jar
   <BEA_HOME>/modules/glassfish.jaxws.rt_2.1.3.jar
   <WL_HOME>/server/lib/webserviceclient.jar
   <WL_HOME>/server/lib/wseeclient.jar
   <WL_HOME>/server/lib/wlfullclient.jar
```

Dynamic Web Service Client CLASSPATH

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

Example 4–3 Dynamic Web Service Client Classpath (with weblogic.jar)

```
CLASSPATH=
   <ALDSP_HOME>/lib/ld-client.jar
   <BEA_HOME>/modules/com.bea.core.sdo_1.1.0.0.jar
   <WL_HOME>/server/lib/weblogic.jar
```

Example 4–4 Dynamic Web Service Client Classpath (with wlfullclient.jar)

```
CLASSPATH=
   <ALDSP_HOME>/lib/ld-client.jar
   <BEA_HOME>/modules/com.bea.core.sdo_1.1.0.0.jar
```

```
<BEA_HOME>/modules/com.bea.core.xml.xmlbeans_1.0.0.0_2-4-0.jar
<BEA_HOME>/modules/com.bea.core.xml.beaxmlbeans_1.0.0.0_2-4-0.jar
<BEA_HOME>/modules/glassfish.jaxws.rt_2.1.3.jar
<WL_HOME>/server/lib/webserviceclient.jar
<WL_HOME>/server/lib/wseeclient.jar
<WL_HOME>/server/lib/wlfullclient.jar
```

4.3.3 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 Section 4.4, "Sample Static Mediator Application" and Section 4.5, "Sample Dynamic Mediator Application."

4.4 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 Section 3.1.3, "Dynamic and Static Mediator APIs."

Topics include:

- Section 4.4.1, "Setting Up the Sample Data Service"
- Section 4.4.2, "Creating a Web Service Map File"
- Section 4.4.3, "Generating the Web Services Mediator Client JAR File"
- Section 4.4.4, "Setting Up the Java Project"
- Section 4.4.5, "Running and Testing the Code"
- Section 4.4.6, "Examining the Sample Code"

4.4.1 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 Section 3.3.1, "Setting Up the Sample Data Service."

4.4.2 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 Workshop for WebLogic. See the *Oracle Fusion Middleware Data Service Integrator Developer's Guide* for detailed instructions.

To create the file using Workshop for WebLogic:

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.

Note: You can generate a Web Services Map file using Workshop for WebLogic, the Oracle Data Service Integrator Console, or an Ant script. These methods are described in detail in the *Oracle Fusion Middleware Data Service Integrator Developer's Guide*.

4.4.3 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.

Note: You can generate a Web Services Mediator Client JAR file using Workshop for WebLogic, the Oracle Data Service Integrator Console, or an Ant script. These methods are described in detail in the *Oracle Fusion Middleware Data Service Integrator Developer's Guide*.

To generate a Web Services Mediator Client JAR file using Workshop for WebLogic:

1. Generate the Web Service Map file, as discussed previously in Section 4.4.2, "Creating a Web Service Map File."
2. Select File > Export.
3. In the Select dialog, select Oracle Data Service Integrator > 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 Section 4.3.2, "Setting the CLASSPATH" for more information.

Note: Generated classes in the JAR file are named according to the conventions described in Section 3.6.4, "Naming Conventions for Generated Classes."

4.4.4 Setting Up the Java Project

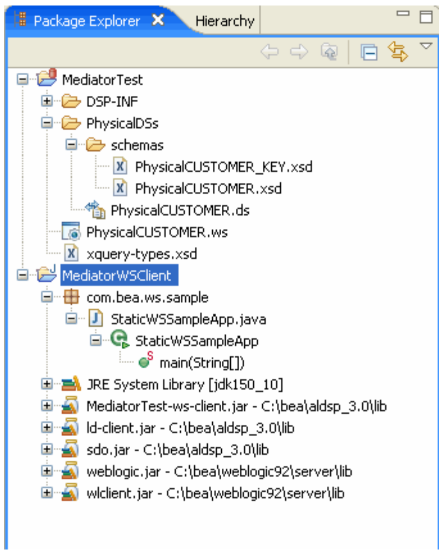
Example 4–5 lists the sample Java client that uses the Static Mediator API to call WSDL operations. The application simply retrieves a DataObject from a data store, modifies the object, and returns it to the data store.

This example assumes you are using Workshop for WebLogic, 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 Section 4.3.2, "Setting the CLASSPATH." 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 Example 4–5 into the source file.
6. Save the file. Figure 4–2 shows the completed project configuration.

Figure 4–2 Completed Project Configuration



```
MediatorTest
  DSP-INF
  PhysicalDSs
    schemas
      PhysicalCUSTOMER_KEY.xsd
      PhysicalCUSTOMER.xsd
    PhysicalCUSTOMER.ds
  PhysicalCUSTOMER.ws
  xquery-types.xsd
MediatorWSClient
  com.bea.ws.sample
    StaticWSSampleApp.java
      StaticWSSampleApp
        main(String[])
  JRE System Library [jdk150_10]
  MediatorTest-ws-client.jar - C:\bea\aldsp_3.0\lib
  ld-client.jar - C:\bea\aldsp_3.0\lib
  sdo.jar - C:\bea\aldsp_3.0\lib
  weblogic.jar - C:\bea\weblogic92\server\lib
  wlclient.jar - C:\bea\weblogic92\server\lib
```

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

Example 4–5 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 {

        // 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,"welcome1");
        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 interating 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 });
    }
}
```

4.4.5 Running and Testing the Code

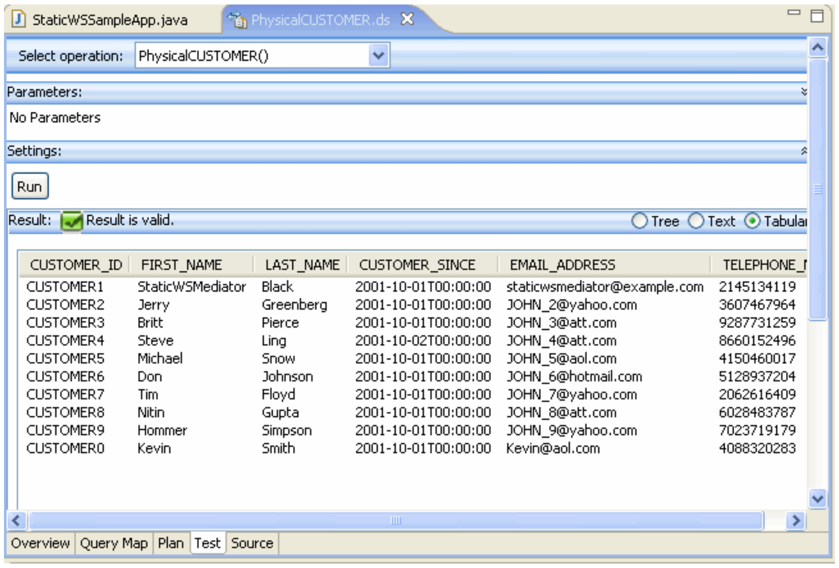
To test the application, simply start the server and run the Java client as a Java application. In Workshop for WebLogic, this is commonly done by right-clicking the Java file and selecting Run As > Java Application.

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



Select operation: PhysicalCUSTOMER()

No Parameters

Result: Result is valid. ( ) Tree ( ) Text [✓ Tabular]

CUSTOMER_ID	FIRST_NAME	LAST_NAME	CUSTOMER_SINCE	EMAIL_ADDRESS	TELEPHONE_I
CUSTOMER1	StaticWSMediator	Black	2001-10-01T00:00:00	staticwsmediator@example.com	2145134119
CUSTOMER2	Jerry	Greenberg	2001-10-01T00:00:00	JOHN_2@yahoo.com	3607467964
CUSTOMER3	Britt	Pierce	2001-10-01T00:00:00	JOHN_3@att.com	9287731259
CUSTOMER4	Steve	Ling	2001-10-02T00:00:00	JOHN_4@att.com	8660152496
CUSTOMER5	Michael	Snow	2001-10-01T00:00:00	JOHN_5@aol.com	4150460017
CUSTOMER6	Don	Johnson	2001-10-01T00:00:00	JOHN_6@hotmail.com	5128937204
CUSTOMER7	Tim	Floyd	2001-10-01T00:00:00	JOHN_7@yahoo.com	2062616409
CUSTOMER8	Nitin	Gupta	2001-10-01T00:00:00	JOHN_8@att.com	6028483787
CUSTOMER9	Hommer	Simpson	2001-10-01T00:00:00	JOHN_9@yahoo.com	7023719179
CUSTOMER0	Kevin	Smith	2001-10-01T00:00:00	Kevin@aol.com	4088320283

4.4.6 Examining the Sample Code

This section discusses the parts of the Java sample in Example 4–5. This section examines the following components of the sample code:

- Section 4.4.6.1, "Importing Packages"
- Section 4.4.6.2, "Obtaining a Data Access Service Handle"
- Section 4.4.6.3, "Retrieving Data from the Service"
- Section 4.4.6.4, "Obtaining a DataObject from the Result"
- Section 4.4.6.6, "Modifying the DataObject"
- Section 4.4.6.7, "Returning Changes to the Server"
- Section 4.4.6.5, "Disposing the Result Object"

4.4.6.1 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.

```
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;
```

4.4.6.2 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 Section 3.6.6, "Obtaining the WebLogic JNDI Context for Oracle Data Service Integrator." For more information on WebLogic JNDI context objects, see Programming WebLogic JNDI at `http://download.oracle.com/docs/cd/E12840_01/wls/docs103/jndi/jndi.html`. For information on security settings, see Section 4.7, "Securing Your Web Services Application."
- A WSDL URL the specifies the address of the web service to access.

```
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,"welcome1");
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);
```

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.

4.4.6.3 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 Section 3.6.5, "Understanding DASResult."

DASResult<PhysicalCUSTOMER> result = das.PhysicalCUSTOMER();

4.4.6.4 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* at
`http://edocs.bea.com/aldsp/docs25/concepts/index.html`.

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

4.4.6.5 Disposing the Result Object

You must call DASResult.dispose() whenever you are finished iterating through a result object. For more information on dispose(), see Section 3.6.5.2, "Disposing of DASResult Objects."

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

4.4.6.6 Modifying the DataObject

After you obtain a DataObject, you can modify it; however, if you intend to submit these changes back to the Oracle Data Service Integrator 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 Section 3.6.7, "Working with Data Objects." After the customer object has change-tracking enabled, the generated setters are called to modify certain values in the customer object.

> **Note:** The getters and setters 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);

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

4.4.6.7 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.

```
das.updatePhysicalCUSTOMER(new PhysicalCUSTOMER[] { customer });
```

4.5 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.

- Section 4.5.1, "Setting Up and Running the Sample Code"
- Section 4.5.2, "Sample Java Client Code (Dynamic Mediator API)"
- Section 4.5.3, "Examining the Sample Code"

4.5.1 Setting Up and Running the Sample Code

To run this sample code, follow the basic setup instructions in Section 4.4, "Sample Static Mediator Application." 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.

4.5.2 Sample Java Client Code (Dynamic Mediator API)

This section shows sample Java client code for using the dynamic mediator API.

Example 4–6 DynamicWSSampleApp.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 interating 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@oracle.com");

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

4.5.3 Examining the Sample Code

This section discusses the parts of the Java sample in Example 4–6. This section examines the following components of the sample code:

- Section 4.5.3.1, "Importing Classes"
- Section 4.5.3.2, "Obtaining a DataAccessService Handle"
- Section 4.5.3.3, "Retrieving Data from the Service"
- Section 4.5.3.4, "Obtaining a DataObject from the Result"
- Section 4.5.3.5, "Disposing the Result Object"
- Section 4.5.3.6, "Modifying the DataObject"
- Section 4.5.3.7, "Returning Changes to the Server"

4.5.3.1 Importing Classes

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

```
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;
```

4.5.3.2 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 Section 3.6.6, "Obtaining the WebLogic JNDI Context for Oracle Data Service Integrator." For more information on WebLogic JNDI context objects, see Programming WebLogic JNDI at `http://download.oracle.com/docs/cd/E12840_01/wls/docs103/jndi/jndi.html`. For more information on security, see Section 4.7, "Securing Your Web Services Application."
- A WSDL URL that specifies the address of the web service to access.

```
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);
```

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.

4.5.3.3 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 Section 3.6.5, "Understanding DASResult." 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])
```

4.5.3.4 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 *Oracle Data Service Integrator Concepts Guide* at `http://edocs.bea.com/aldsp/docs25/concepts/index.html`. See also Section 3.1.1, "What is SDO?."

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

4.5.3.5 Disposing the Result Object

You must call DASResult.dispose() whenever you are finished iterating through a result object. For more information on dispose(), see Section 3.6.5.2, "Disposing of DASResult Objects."

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

4.5.3.6 Modifying the DataObject

After you obtain a DataObject, you can modify it; however, if you intend to submit these changes back to the Oracle Data Service Integrator 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 Section 3.6.7, "Working with Data Objects." After the customer object has change-tracking enabled, the set method is called to modify certain values in the customer object.

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

4.5.3.7 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.

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

4.6 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 Section 3.10, "Understanding Transaction Behavior.".

Example 4–7 shows how to set the com.bea.dsp.RequestConfig.ReadTransactionMode attribute.

Example 4–7 Setting the ReadTransactionMode Attribute

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

if(readTransactionRequired) {
   config.setEnumAttribute(RequestConfig.ReadTransactionMode.REQUIRED);
}
```

4.7 Securing Your Web Services Application

Oracle Data Service Integrator Native Web Services supports the following security features:

- Basic authentication (Web Application Security)

- Transport level security (HTTPS)
- Message level security (Web Services Security)

Note: For detailed information on configuring these security options, see "Configure Security for Web Services Applications" in the *Oracle Fusion Middleware Data Service Integrator 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 Example 4–8 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 the Oracle Data Service Integrator Oracle Service Bus transport. This transport allows Oracle Data Service Integrator 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 at `http://download.oracle.com/docs/cd/E12840_01/wls/docs103/jndi/jndi.html`.

- DSPWebServicesProperties.USER_NAME
- DSPWebServicesProperties.PASSWORD

Example 4–8 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 *Oracle Fusion Middleware Data Service Integrator Developer's Guide*.

Example 4–8 Example X.509 Certificate Token Profile Setup

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

// Create emtpy 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 = "welcome1";
// 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);
```

5 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 Oracle Data Service Integrator JDBC driver. The chapter covers the following topics:

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

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

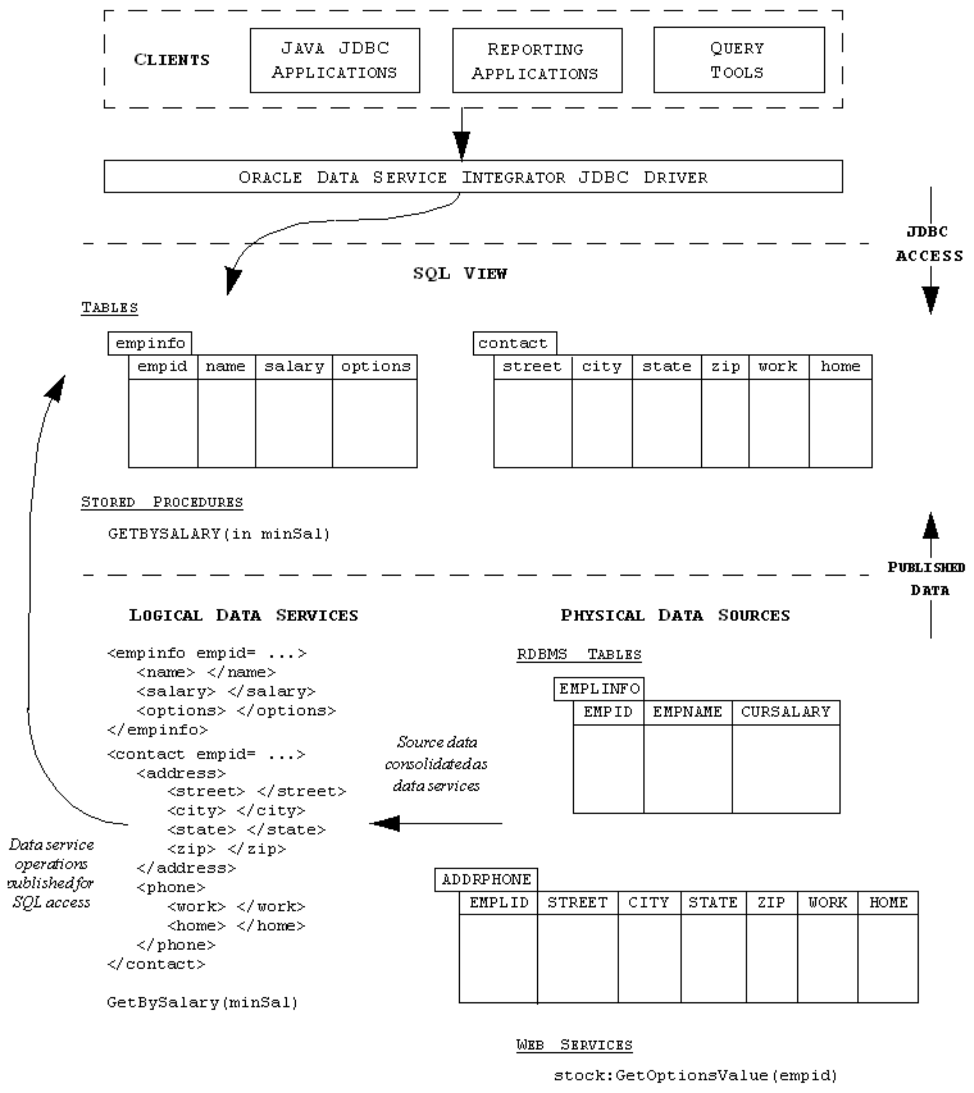
As Figure 5–1 shows, source data can be consolidated, integrated, and transformed using Oracle Data Service Integrator data services. The source data itself can come from disparate sources throughout the enterprise, including relational databases and Web services, among others.

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 Oracle Data Service Integrator.

Note that the Oracle Data Service Integrator JDBC driver does impose the following constraints on data services:

- You can use the Oracle Data Service Integrator 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 Oracle Data Service Integrator 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



5.1.1 Features of the Oracle Data Service Integrator JDBC Driver

The Oracle Data Service Integrator JDBC driver implements the java.sql.* interface in JDK 1.7x to provide access to an Oracle Data Service Integrator server through the JDBC interface. The driver has the following features:

- Supports SQL-92 SELECT statements
- Implements the JDBC 4.0 Application Programming Interface (API)

- Supports Oracle Data Service Integrator with JDK 1.7
- 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 Oracle Data Service Integrator 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 Oracle Data Service Integrator console configuration should be set to check access control. For more information, refer to the "Securing Data Services Platform Resources" section in the *Oracle Fusion Middleware Administering Data Service Integrator* guide.

5.1.2 Exploring Oracle Data Service Integrator and JDBC Artifacts

The Oracle Data Service Integrator views data retrieved from a database in the form of data sources and operations. Table 5–1 shows the equivalent terminology.

Table 5–1 Oracle Data Service Integrator and JDBC Driver Artifacts

Oracle Data Service Integrator	JDBC
Dataspace Project	JDBC connection parameter (Driver URL)
Operation with parameters	Stored procedure
Operation without parameters	Table or stored procedure

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`.

5.2 JDBC and SQL Support in Oracle Data Service Integrator

This section describes the JDBC and SQL support in the Oracle Data Service Integrator JDBC driver

5.2.1 JDBC Support

The Oracle Data Service Integrator JDBC driver implements the following interfaces from the `java.sql` package as specified in JDK 1.7x:

- 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

The Oracle Data Service Integrator JDBC driver supports the following methods:

Table 5–2 Oracle Data Service Integrator JDBC Driver Methods

Interface	Supported Methods	Supported Methods
java.sql.Blob	getBinaryStream	position
	getBytes	truncate
	length	
java.sql.CallableStatement	clearParameters	setFloat
	executeQuery	setInt
	setAsciiStream	setLong
	setBigDecimal	setNull
	setBoolean	setObject
	setByte	setShort
	setBytes	setString
	setCharacterStream	setTime
	setDate	setTimestamp
	setDouble	
java.sql.Connection	clearWarnings	getWarnings
	close	isClosed
	createStatement	isReadOnly
	getAutoCommit	nativeSQL
	getCatalog	prepareCall
	getHoldability	prepareStatement
	getLogPrintWriter	setAutoCommit
	getMetaData	setCatalog
	getSchema	setHoldability
	getTransactionIsolation	setReadOnly
	getTypeMap	

Table 5–2 (Cont.) Oracle Data Service Integrator JDBC Driver Methods

Interface	Supported Methods	Supported Methods
java.sql.DatabaseMetaData	allProceduresAreCallable	getDatabaseMinorVersion
	allTablesAreSelectable	getDatabaseProductName
	dataDefinitionCausesTransactionCommit	getDatabaseProductVersion
	dataDefinitionIgnoredInTransactions	getDefaultTransactionIsolation
	deletesAreDetected	getDriverMajorVersion
	doesMaxRowSizeIncludeBlobs	getDriverMinorVersion
	getAttributes	getDriverName
	getBestRowIdentifier	getDriverVersion
	getCatalogs	getExportedKeys
	getCatalogSeparator	getExtraNameCharacters
	getCatalogTerm	getIdentifierQuoteString
	getColumnPrivileges	getImportedKeys
	getColumns	getIndexInfo
	getConnection	getJDBCMajorVersion
	getCrossReference	getJDBCMinorVersion
	getDatabaseMajorVersion	

Table 5–2 (Cont.) Oracle Data Service Integrator JDBC Driver Methods

Interface	Supported Methods	Supported Methods
java.sql.DatabaseMetaData	getMaxBinaryLiteralLength	getSearchStringEscape
	getMaxCatalogNameLength	getSQLKeywords
	getMaxCharLiteralLength	getSQLStateType
	getMaxColumnNameLength	getStringFunctions
	getMaxColumnsInGroupBy	getSuperTables
	getMaxColumnsInIndex	getSuperTypes
	getMaxColumnsInOrderBy	getSystemFunctions
	getMaxColumnsInSelect	getTablePrivileges
	getMaxColumnsInTable	getTables
	getMaxConnections	getTableTypes
	getMaxCursorNameLength	getTimeDateFunctions
	getMaxIndexLength	getTypeInfo
	getMaxProcedureNameLength	getUDTs
	getMaxRowSize	getURL
	getMaxSchemaNameLength	getUserName
	getMaxStatementLength	getVersionColumns
	getMaxStatements	insertsAreDetected
	getMaxTableNameLength	isCatalogAtStart
	getMaxTablesInSelect	isReadOnly
	getMaxUserNameLength	locatorsUpdateCopy
	getNumericFunctions	nullPlusNonNullIsNull
	getPrimaryKeys	nullsAreSortedAtEnd
	getProcedureColumns	nullsAreSortedAtStart
	getProcedures	nullsAreSortedHigh
	getProcedureTerm	nullsAreSortedLow
	getResultSetHoldability	othersDeletesAreVisible
	getSchemas	othersInsertsAreVisible
	getSchemaTerm	othersUpdatesAreVisible

Table 5–2 (Cont.) Oracle Data Service Integrator JDBC Driver Methods

Interface	Supported Methods	Supported Methods
java.sql.DatabaseMetaData	ownDeletesAreVisible ownInsertsAreVisible ownUpdatesAreVisible storesLowerCaseIdentifiers storesLowerCaseQuotedIdentifiers storesMixedCaseIdentifiers storesMixedCaseQuotedIdentifiers storesUpperCaseIdentifiers storesUpperCaseQuotedIdentifiers supportsAlterTableWithAddColumn supportsAlterTableWithDropColumn supportsANSI92EntryLevelSQL supportsANSI92FullSQL supportsANSI92IntermediateSQL supportsBatchUpdates supportsCatalogsInDataManipulation supportsCatalogsInIndexDefinitions supportsCatalogsInPrivilegeDefinitions supportsCatalogsInProcedureCalls supportsCatalogsInTableDefinitions	supportsColumnAliasing supportsConvert supportsCoreSQLGrammar supportsCorrelatedSubqueries supportsDataDefinitionAndDataManipu lationTransactions supportsDataManipulationTransactions Only supportsDifferentTableCorrelationName s supportsExpressionsInOrderBy supportsExtendedSQLGrammar supportsFullOuterJoins supportsGetGeneratedKeys supportsGroupBy supportsGroupByBeyondSelect supportsGroupByUnrelated supportsIntegrityEnhancementFacility supportsLikeEscapeClause supportsLimitedOuterJoins
java.sql.DatabaseMetaData	supportsMinimumSQLGrammar supportsMixedCaseIdentifiers supportsMixedCaseQuotedIdentifiers supportsMultipleOpenResults supportsMultipleResultSets supportsMultipleTransactions supportsNamedParameters supportsNonNullableColumns supportsOpenCursorsAcrossCommit supportsOpenCursorsAcrossRollback supportsOpenStatementsAcrossCommit supportsOpenStatementsAcrossRollback supportsOrderByUnrelated supportsOuterJoins supportsPositionedDelete supportsPositionedUpdate supportsResultSetConcurrency supportsResultSetHoldability supportsResultSetType	supportsSavepoints supportsSchemasInDataManipulation supportsSchemasInIndexDefinitions supportsSchemasInPrivilegeDefinitions supportsSchemasInProcedureCalls supportsSchemasInTableDefinitions supportsSelectForUpdate supportsStatementPooling supportsStoredProcedures supportsSubqueriesInComparisons supportsSubqueriesInExists supportsSubqueriesInIns supportsSubqueriesInQuantifieds supportsTableCorrelationNames supportsTransactionIsolationLevel supportsTransactions supportsUnion supportsUnionAll
java.sql.DatabaseMetaData	updatesAreDetected usesLocalFilePerTable	usesLocalFiles

Table 5–2 (Cont.) Oracle Data Service Integrator JDBC Driver Methods

Interface	Supported Methods	Supported Methods
java.sql.ParameterMetaData	close	getParameterTypeName
	getParameterClassName	getPrecision
	getParameterCount	getScale
	getParameterMode	isNullable
	getParameterType	isSigned
java.sql.PreparedStatement	addBatch	setCharacterStream
	clearParameters	setDate
	close	setDouble
	execute	setFloat
	executeQuery	setInt
	getMetaData	setLong
	getParameterMetaData	setNull
	setAsciiStream	setObject
	setBigDecimal	setShort
	setBlob	setString
	setBoolean	setTime
	setByte	setTimestamp
	setBytes	
java.sql.ResultSet	clearWarnings	getDouble
	close	getFetchDirection
	findColumn	getFetchSize
	getAsciiStream	getFloat
	getBigDecimal	getInt
	getBlob	getLong
	getBoolean	getMetaData
	getByte	getObject
	getBytes	getRow
	getCharacterStream	getShort
	getConcurrency	getStatement
	getDate	getString

Table 5–2 (Cont.) Oracle Data Service Integrator JDBC Driver Methods

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

5.2.2 SQL Support

This section outlines SQL-92 support in the Oracle Data Service Integrator JDBC driver, and contains the following sections:

- Section 5.2.2.1, "Supported SQL Statements"
- Section 5.2.2.2, "Supported SQL Functions"
- Section 5.2.3.1, "Table Parameter Support"

5.2.2.1 Supported SQL Statements

The Oracle Data Service Integrator 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.

5.2.2.2 Supported SQL Functions

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

- Section 5.2.2.3, "Numeric Functions"
- Section 5.2.2.4, "String Functions"
- Section 5.2.2.5, "Datetime Functions"
- Section 5.2.2.6, "Aggregate Functions"
- Section 5.2.3, "JDBC Metadata Search Patterns"

5.2.2.3 Numeric Functions

The Oracle Data Service Integrator JDBC driver supports the numeric functions described in Table 5–3.

Table 5–3 Numeric Functions

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

5.2.2.4 String Functions

The Oracle Data Service Integrator JDBC driver supports the string functions described in Table 5–4.

Table 5–4 String Functions

Function	Signature	Comment
CONCAT	varchar CONCAT(varchar s1, varchar s2)	CONCAT returns *s1* concatenated with *s2*. If any argument is NULL, it is considered to be equivalent to the empty string.
Left	varchar left (varchar s, numeric n)	Left returns the left n characters of s.
LENGTH	numeric LENGTH(varchar s)	LENGTH returns the length of *s*. The function returns 0 if *s* is NULL.

Table 5–4 (Cont.) String Functions

Function	Signature	Comment
LOWER	varchar LOWER(varchar s)	LOWER returns *s*, with all letters lowercase. If *s* is NULL, the function returns an empty string.
LPAD	varchar lpad(varchar v, numeric n, varchar p)	LPAD returns v, with n characters of an infinitely repeating p appended to the left.
LTRIM	varchar LTRIM(varchar s)	LTRIM trims leading blanks from *s*. If *s* is NULL, the function returns NULL.
Right	varchar right (varchar s, numeric n)	Right returns the right n characters of s.
RPAD	varchar rpad(varchar v, numeric n, varchar p)	RPAD returns v, with n characters of an infinitely repeating p appended to the right.
RTRIM	varchar RTRIM(varchar s)	RTRIM trims trailing blanks from *s*. If *s* is NULL, the function returns NULL.
SUBSTR	varchar SUBSTR(varchar s, numeric start)	SUBSTR with two arguments returns substring of *s* starting at start, inclusive. The first character in *s* is located at index 1. If *s* is NULL, the function returns an empty string.
TRIM	varchar TRIM(varchar s)	TRIM trims leading and trailing blanks from *s*. If *s* is NULL, TRIM returns NULL.
UPPER	varchar UPPER(varchar s)	UPPER returns *s*, with all letters uppercase. If *s* is NULL, UPPER returns the empty string.

5.2.2.5 Datetime Functions

The Oracle Data Service Integrator JDBC driver supports the datetime functions described in Table 5–5.

Table 5–5 Datetime Functions

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

Table 5–5 (Cont.) Datetime Functions

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

5.2.2.6 Aggregate Functions

The Oracle Data Service Integrator JDBC driver supports the aggregation functions described in Table 5–6.

Table 5–6 Aggregate Functions

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

5.2.3 JDBC Metadata Search Patterns

The Oracle Data Service Integrator JDBC driver supports standard JDBC API search patterns, as shown in Table 5–7.

Table 5–7 JDBC Driver Metadata Search Patterns

Pattern	Purpose
"string"	Matches the identified string.
""	Uses the default catalog/schema.
"%"	Wildcard; equivalent to * in regular expressions.
"_"	Matches a single character; equivalent to . (period) in regular expressions.
null	Wildcard; same as "%"

For more information about using the JDBC metadata API, refer to the Java documentation.

Assuming that the default_catalog is catalog1 and default_schema is schema1, Table 5–8 shows some common matching patterns.

Table 5–8 JDBC Driver Metadata Search Patterns

Pattern	Matching Example
"Oracle"	Matches the identified string, Oracle.
"abc%d"	Matches: ■ abc10d ■ abcd ■ abc_practically anything_d But not: ■ abc10e ■ abc10def
abc%d_	Matches: ■ abc10d ■ abcd ■ abc_practically anything_d ■ abc10dg But not: ■ abc10dgh ■ abc10dgPattern
""	A call to:
and	`DBDatabaseMetadata.getTables("",null,"abc%")`
null	would return all tables starting with abc under catalog 1.

5.2.3.1 Table Parameter Support

The Oracle Data Service Integrator 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 Section 5.5.1, "Using Table Parameters."

5.2.4 Additional Details and Limitations

When using the ALSDSP JDBC driver, each connection points to one Oracle Data Service Integrator dataspace. Table 5–9 notes the Oracle Data Service Integrator JDBC driver limitations that apply to SQL language features.

Table 5–9 Oracle Data Service Integrator JDBC Driver Limitations Applying to SQL Language Features

Feature	Comments	Example

Assignment in select	Not supported.	`SELECT MYCOL = 2`
		`FROM VTABLE`
		`WHERE COL4 IS NULL`

The CORRESPONDING BY construct with the set-Operations
(UNION, INTERSECT and EXCEPT)
	The SQL-92 specified default column ordering in the set operations *is* supported.
	Both the table-expressions (the operands of the set-operator) must conform to the same relational schema.
		`(SELECT NAME, CITY FROM CUSTOMER1) UNION CORRESPONDING BY (CITY, NAME) (SELECT CITY, NAME FROM CUSTOMER2)`
		The supported query is:
		`(SELECT NAME, CITY FROM CUSTOMER1) UNION (SELECT NAME, CITY FROM CUSTOMER2)`

5.3 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:

- Section 5.3.1, "Publishing Data Service Operations"
- Section 5.3.2, "Configuring the Oracle Data Service Integrator JDBC Driver"

5.3.1 Publishing Data Service Operations

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

> **Note:** SQL objects published through Oracle Data Service Integrator 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 Oracle Data Service Integrator dataspace.

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

5.3.2 Configuring the Oracle Data Service Integrator JDBC Driver

The Oracle Data Service Integrator JDBC driver is located in the `ldjdbc.jar` file, which is available in the `<ALDSP_HOME>/lib` directory after you install Oracle Data Service Integrator. To use the Oracle Data Service Integrator 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 Section 5.6.1, "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>
      ```

      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
```

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 Oracle Data Service Integrator 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" on page 5-17 or "Configuring the Connection in the JDBC URL" on page 5-17 respectively.

5.4 Accessing Data Services Using SQL From a Java Application

You can have a Java application access information from data services using SQL through the Oracle Data Service Integrator JDBC driver.

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

1. Obtain a connection to the Oracle Data Service Integrator dataspace.

   For more information, see Section 5.4.1, "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 Section 5.4.1.1, "Using the PreparedStatement Interface" and Section 5.4.1.2, "Using the CallableStatement Interface" respectively.

5.4.1 Obtaining a Connection

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

```
jdbc:dsp@<WLServerAddress>:<WLServerPort>/<DataspaceName>
(/default catalog/default schema;
param(=value1; param2=value2;)?
```

For example:

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

The name of the Oracle Data Service Integrator JDBC driver class is:

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

Configuring the Connection Using the Properties Object

You can establish a connection to an Oracle Data Service Integrator dataspace using the Properties object as follows:

```
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 Oracle Data Service Integrator dataspace name, `TestProjectDataSpace`, in the connection object itself, as shown in the following segment:

```
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:

```
// 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>);
```

5.4.1.1 Using the PreparedStatement Interface

You can use the `preparedQueryWithParameters` 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 Oracle WebLogic Server, which hosts Oracle Data Service Integrator.

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`.

```
public ResultSet preparedQueryWithParameters(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 Section 5.3.1, "Publishing Data Service Operations."

5.4.1.2 Using the CallableStatement Interface

After you establish a connection to a server where Oracle Data Service Integrator 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 Oracle Data Service Integrator 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`.

```
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;
}
```

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

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

5.5 Advanced Features

This section describes advances features and uses of the Oracle Data Service Integrator JDBC driver and contains the following sections

- Section 5.5.1, "Using Table Parameters"
- Section 5.5.2, "Accessing Custom Database Functions Using JDBC"

5.5.1 Using Table Parameters

This section describes how to use the Oracle Data Service Integrator JDBC driver to pass table parameters to data services.

5.5.1.1 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

```
? 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.

5.5.1.2 Setting Table Parameters Using JDBC

The Oracle Data Service Integrator JDBC driver passes table parameters to data services through its TableParameter class. The class (shown in its entirety in Example 5–1) represents an entire table parameter and the rows it represents.

Example 5–1 Table Parameter Interface

```
public class TableParameter implements Serializable
    /**
     * Constructor
     *
     * @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 {
        /**
         * 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:

```
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:

```
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:

```
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

   ```
   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 Example 5–2:

Example 5–2 JDBC Code Supporting Table Parameter Example

```
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 Oracle Data Service Integrator
    * 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();
```

```
      }
      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 = 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);
      // Run the query and print the results
      ResultSet rs = ps.executeQuery();
      printResultSet(rs);
   }
}
```

Table Parameter ValueTypes

Table 5–10 lists the table parameter ValueTypes supported by the Oracle Data Service Integrator JDBC driver.

Table 5–10 TableParameter ValueTypes

Type Name	Type Value	Java Type
ValueType.REPEATING_SMALLINT	16 bit signed integer	Short
ValueType.REPEATING_INTEGER	32 bit signed integer	Integer
ValueType.REPEATING_BIGINT	64 bit signed integer	Long
ValueType.REPEATING_REAL	32 bit floating point	Float
ValueType.REPEATING_DOUBLE	64 bit floating point	Double
ValueType.REPEATING_DECIMAL	decimal	BigDecimal
ValueType.REPEATING_VARCHAR	string	String
ValueType.REPEATING_DATE	date	java.sql.Date
ValueType.REPEATING_TIME	time	java.sql.Time
ValueType.REPEATING_TIMESTAMP	datetime	java.sql.Timestamp
ValueType.REPEATING_BLOB	byte array	char[]
ValueType.REPEATING_BOOLEAN	Boolean	Boolean

Table 5–10 (Cont.) TableParameter ValueTypes

Type Name	Type Value	Java Type
ValueType.REPEATING_YMINTERVAL	year month interval	weblogic.xml.query.datetime.YearMonthDuration
ValueType.REPEATING_DTINTERVAL	day time interval	weblogic.xml.query.datetime.DayTimeDuration
ValueType.REPEATING_INTERVAL	both year month & day time interval	weblogic.xml.query.datetime.Duration

5.5.2 Accessing Custom Database Functions Using JDBC

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 Oracle Data Service Integrator through a library. After registering the functions, you can use them in SQL statements submitted to the Oracle Data Service Integrator 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 Oracle Data Service Integrator 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.
- Oracle Data Service Integrator does not support functions of the form `TRIM( TRAILING ' ' FROM $column)` as custom database functions.
- Oracle Data Service Integrator does not support special columns such as `SYSDATE`, `ROWNUM`, or similar columns as parameters to custom database functions.
- You must explicitly expose Oracle Data Service Integrator artifacts in the SQL Map for the dataspace.

5.6 Accessing Data Services Using SQL-Based Applications

You can access data services using both SQL-based applications and applications that connect to the Oracle Data Service Integrator 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:

- Section 5.6.1, "Accessing Data Services Using SQL Explorer"
- Section 5.6.2, "Connecting to the Oracle Data Service Integrator Client Using OpenLink ODBC-JDBC Bridge"
- Section 5.6.3, "Using OpenLink with Reporting Tools"

You can also use the Oracle Data Service Integrator 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`.

5.6.1 Accessing Data Services Using SQL Explorer

You can use the Oracle Data Service Integrator JDBC driver with Eclipse SQL Explorer to access data services. This section describes how to configure SQL Explorer to use the Oracle Data Service Integrator 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 Oracle Data Service Integrator 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:

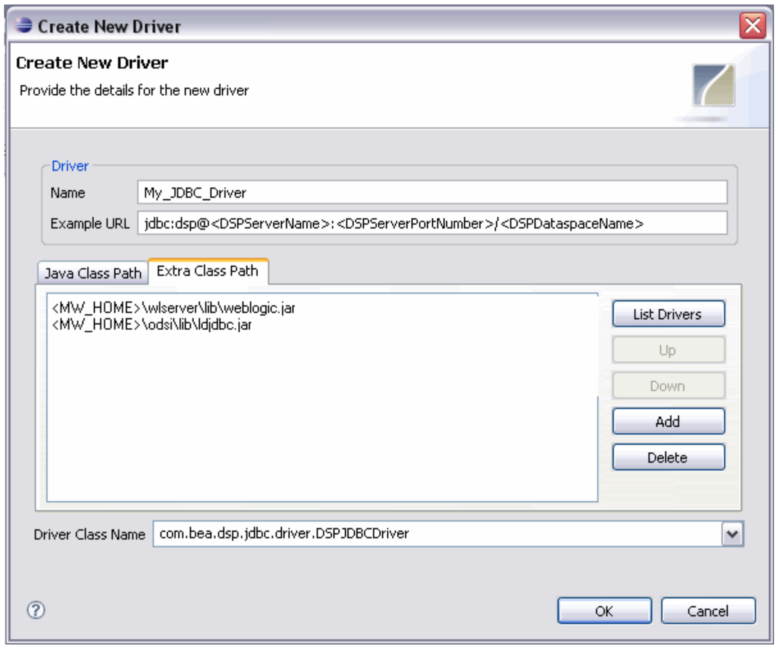
1. Download the SQL Explorer software from the following link:

   ```
   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 Oracle Data Service Integrator Perspective. Start the web server within Eclipse and open the dataspace (project).
6. Choose Window Æ 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 `odsi_jdbc_driver`.
   - Set the example URL to:

     ```
     jdbc:dsp@<DSPServerName>:<DSPServerPortNumber>/<DSPDataspaceName>
     ```

Figure 5–2 Create New Driver Dialog



Name: My_JDBC_Driver

Example URL: jdbc:dsp@<DSPServerName>:<DSPServerPortNumber>/<DSPDataspaceName>

<MW_HOME>\wlserver\lib\weblogic.jar
<MW_HOME>\odsi\lib\ldjdbc.jar

Driver Class Name: com.bea.dsp.jdbc.driver.DSPJDBCDriver

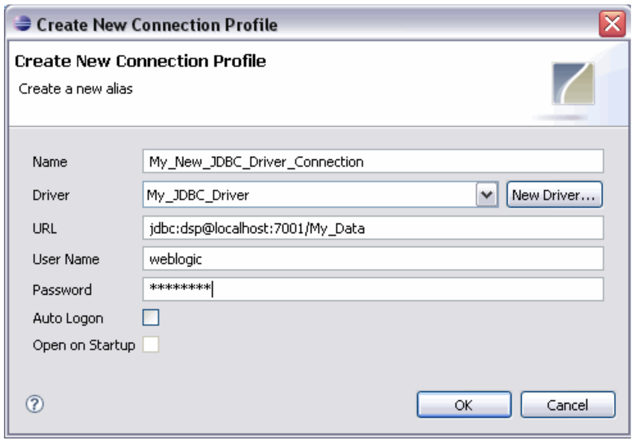
7. Click the Extra Class Path tab and then click Add. Enter the paths for two JAR files, as follows:
   - <ALDSP_HOME>/lib/ldjdbc.jar
   - <BEA_HOME>/wlserver_10.3/server/lib/weblogic.jar

   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 Oracle Data Service Integrator JDBC Driver from the drop-down list. Enter the URL for the JDBC Driver, then enter the user name and password, and click OK.

Figure 5–3 Create New Connection Profile Dialog


Create New Connection Profile

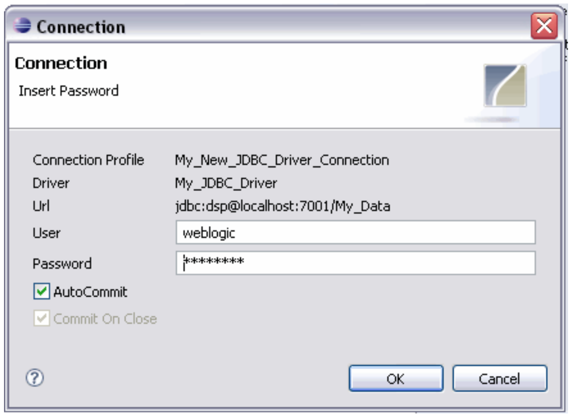
Create New Connection Profile

Create a new alias

Name	My_New_JDBC_Driver_Connection
Driver	My_JDBC_Driver
URL	jdbc:dsp@localhost:7001/My_Data
User Name	weblogic
Password	********
Auto Logon	☐
Open on Startup	☐

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.

Figure 5–4 Connection Dialog


Connection

Connection

Insert Password

Connection Profile	My_New_JDBC_Driver_Connection
Driver	My_JDBC_Driver
Url	jdbc:dsp@localhost:7001/My_Data
User	weblogic
Password	********

✓ AutoCommit

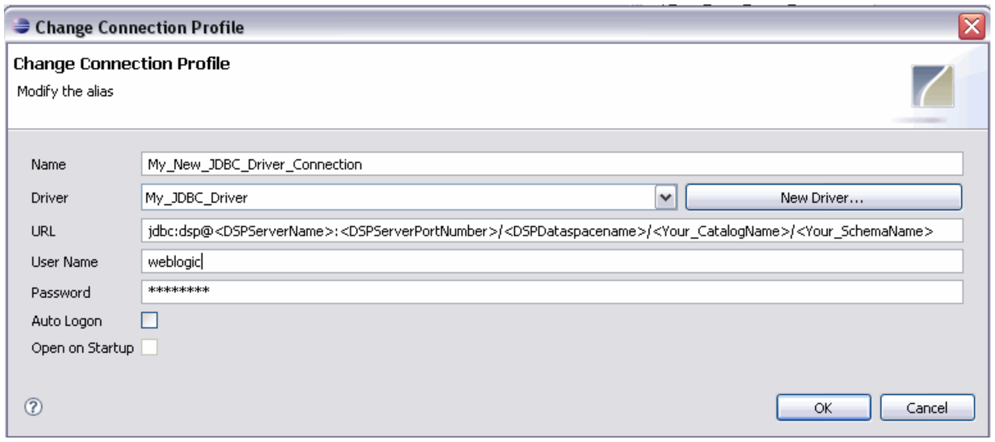
✓ Commit On Close

The data displays in the Database Structure and Database Detail window.

1. If the JDBC Driver window is not open, choose Window > Show View > Other > SQL Explorer Database Structure > OK to display the client data.
2. 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>
```

Figure 5–5 Change Connection Profile Dialog



Name: My_New_JDBC_Driver_Connection
Driver: My_JDBC_Driver
URL: jdbc:dsp@<DSPServerName>:<DSPServerPortNumber>/<DSPDataspacename>/<Your_CatalogName>/<Your_SchemaName>
User Name: weblogic
Password: ********
Auto Logon
Open on Startup

5.6.2 Connecting to the Oracle Data Service Integrator Client Using OpenLink ODBC-JDBC Bridge

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

You can use the Openlink ODBC-JDBC driver to interface with the Oracle Data Service Integrator JDBC driver to query Oracle Data Service Integrator 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 `http://download.openlinksw.com/download/login.vsp?pform=2&pfam=1&pcat=1&prod=odbc-jdbc-bridge-st&os=i686-generic-win-32&os2=i686-generic-win-32&release-dbms=6.0-jdbc`.

   Note: For Windows platforms, be sure to save the value of your CLASSPATH before installation.

2. Create a system DSN and configure it for your Oracle Data Service Integrator 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:

```
<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 1.7 (6.0) and click Finish.
7. Specify the DSN name, for example, openlink-odsi.
8. Click Next. Then on the next screen, enter the following next to the JDBC driver:

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

9. Type the following in the URL string field:

   ```
   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 Oracle Data Service Integrator Oracle 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.

5.6.3 Using OpenLink with Reporting Tools

This section describes how to configure and use reporting tools with the Oracle Data Service Integrator ODBC-JDBC driver.

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.

Note: Support for third party reporting tools is deprecated in Oracle Data Service Integrator 10gR3.

5.6.3.1 Microsoft Access 2003-ODBC

This section describes the procedure to connect Microsoft Access 2003 to Oracle Data Service Integrator through an ODBC-JDBC bridge.

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-odsi to connect to the Oracle Data Service Integrator JDBC driver and generate the report.

Limitations and Usage Notes

Note the following:

- 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 Oracle Data Service Integrator using MS Access.
- In MS Access, to sort data retrieved from Oracle Data Service Integrator, select a Unique Record Identifier when you link tables imported from Oracle Data Service Integrator. If you do not select the Unique Record Identifier, then an exception occurs when you try to sort data.

5.6.3.2 Microsoft Excel 2003-ODBC

This section describes the procedure for connecting Microsoft Excel 2003 to Oracle Data Service Integrator through an ODJB-JDBC bridge using OpenLink.

To connect Microsoft Excel to Oracle Data Service Integrator, perform the following steps:

1. Launch Workshop for WebLogic and then start the WebLogic Server.
2. Build and deploy the Oracle Data Service Integrator 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-odsi 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 Section 5.6.2, "Connecting to the Oracle Data Service Integrator Client Using OpenLink ODBC-JDBC Bridge."
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 Oracle Data Service Integrator 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 here:
`http://support.microsoft.com/kb/298955/en-us`.

6 Supporting ADO.NET Clients

This chapter describes how to enable interoperability between Oracle Data Service Integrator 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 Oracle Data Service Integrator 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 Oracle Data Service Integrator operations (functions and procedures)—all the work is done on the server-side.

ADO.NET-client-application developers need only incorporate the Oracle Data Service Integrator-generated web service into their programming environments, as you would when creating any Web service client application.

Information about how Oracle Data Service Integrator 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:

- Section 6.1, "Overview of ADO.NET Integration in Oracle Data Service Integrator"
- Section 6.1.4, "Enabling Oracle Data Service Integrator Support for ADO.NET Clients"
- Section 6.2, "Adapting Oracle Data Service Integrator XML Types (Schemas) for ADO.NET Clients"
- Section 6.3, "Creating a Data Service Based on an RPC-Style Web Service"
- Section 6.4, "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.

6.1 Overview of ADO.NET Integration in Oracle Data Service Integrator

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 Oracle Data Service Integrator client applications.

See Section 2.3, "Role of the Mediator API and SDO" 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, Oracle Data Service Integrator provides ADO.NET-compliant DataSets so that ADO.NET client developers can leverage data services provided by the Oracle Data Service Integrator, just as they would any ADO.NET-specific data sources.

Enabling a Oracle Data Service Integrator data service to support ADO.NET involves the following steps:

- Section 6.1.5, "Generating an Oracle Data Service Integrator Web Services Mapper"
- Section 6.1.7, "Creating a Web Reference in ADO.NET Client by Providing the Oracle Data Service Integrator WSDL URL"

6.1.1 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.

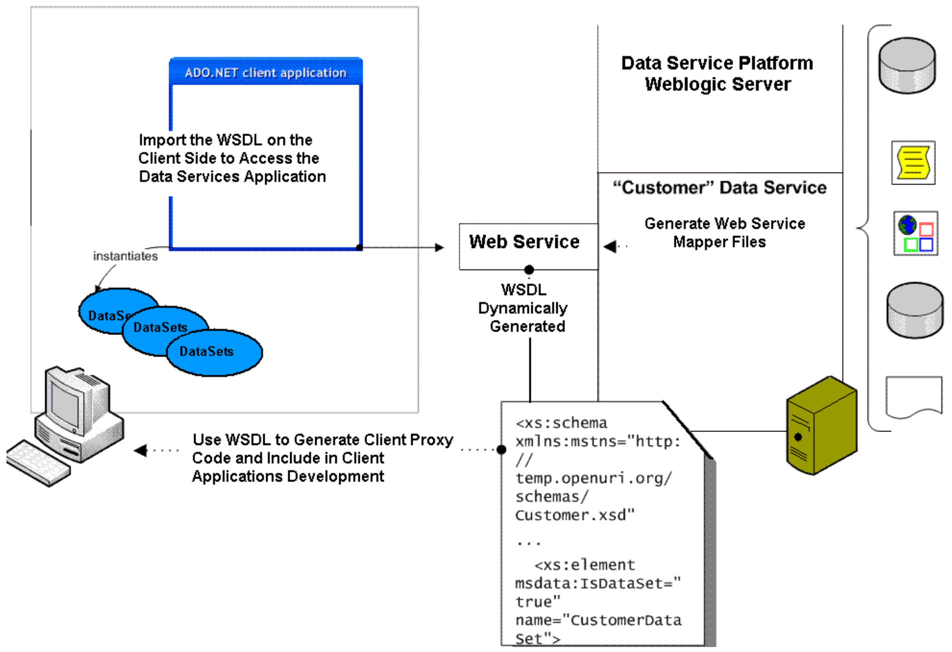
Note: Although ADO.NET supports both connected (direct) and disconnected models, only the disconnected model is supported in the Oracle Data Service Integrator.

6.1.1.1 ADO.NET Client Application Development Tools

ADO.NET client applications are typically created using Microsoft Windows Forms, Web Forms, C#, or Visual Basic. Microsoft Windows Forms is a collection of classes used by client application developers to create graphical user interfaces for the Windows .NET managed environment.

Web Forms provides similar client application infrastructure for creating web-based client applications. Any of these client tools can be used by developers to create applications that leverage ADO.NET for data sources.

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


ADO.NET client application
Import the WSDL on the Client Side to Access the Data Services Application
instantiates
DataSets
DataSets
DataSets
Use WSDL to Generate Client Proxy Code and Include in Client Applications Development
Web Service
WSDL Dynamically Generated
Data Service Platform Weblogic Server
"Customer" Data Service
Generate Web Service Mapper Files
<xs:schema xmlns:mstns="http://temp.openuri.org/schemas/Customer.xsd"
...
<xs:element msdata:IsDataSet="true" name="CustomerDataSet">

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 Section 6.1.5, "Generating an Oracle Data Service Integrator Web Services Mapper."

6.1.2 Understanding How Oracle Data Service Integrator Supports ADO.NET Clients

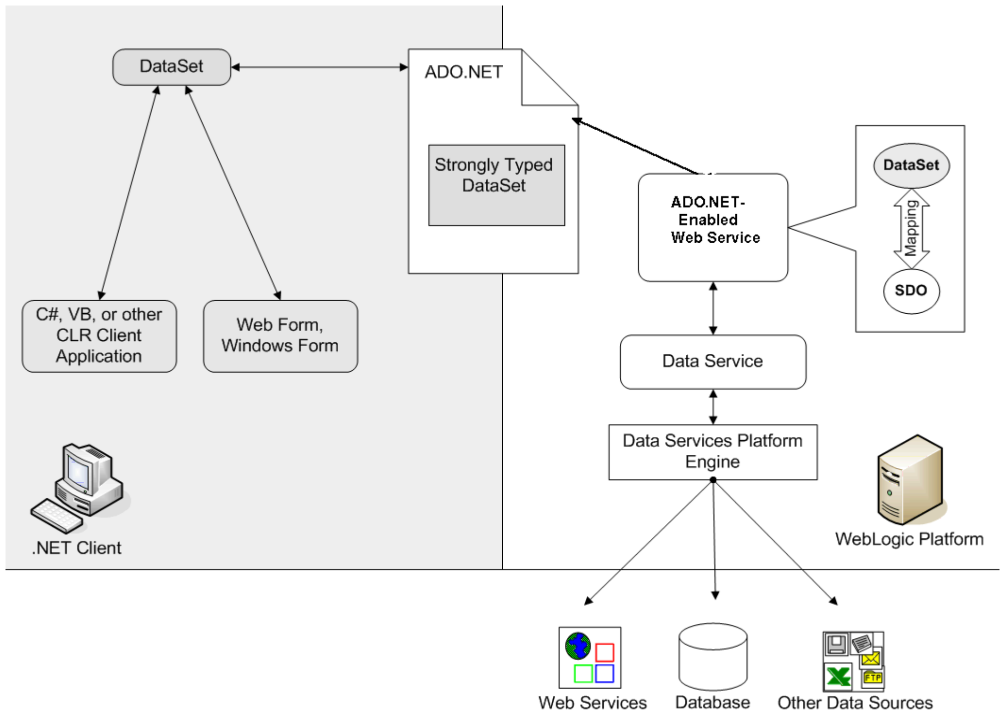
Oracle Data Service Integrator supports ADO.NET at the data object level. That is, the Oracle Data Service Integrator maps inbound ADO.NET DataSet objects to SDO DataObjects, and maps outbound SDOs to DataSets. The mapping is performed transparently on the server, and is bidirectional.

Table 6–1 ADO.NET and SDO Data Objects Compared

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

As shown in Figure 6–2, the ADO.NET typed DataSet is submitted to and returned by Oracle Data Service Integrator. 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.

Figure 6–2 Oracle Data Service Integrator and .NET Integration



Mapping, transformation, and packaging processes are transparent to client application developers and data services developers. Only the items listed in Table 6–2 are exposed to data service developers.

Table 6–2 Oracle Data Service Integrator—Java and ADO.NET-Enabled Artifacts

Name	Example	Description
Data Service	Customer.ds	An XQuery file that instantiates operations such as read functions, navigation functions, procedures, and update functionality at runtime.
Data Service Schema	Customer.xsd	The schema associated with the XML type of the original data service.
DataSet Schema	CustomerDataSet.xsd	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.
Web Service Map	CustomerNET.ws	Web Services mapper file that maps data service operations to web service operations.

6.1.3 Supporting Java Clients

The WSDL generated by Oracle Data Service Integrator from an ADO.NET-enabled web services maps 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.

6.1.4 Enabling Oracle Data Service Integrator 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 Workshop for WebLogic.

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 *Oracle Fusion Middleware Data Service Integrator Developer's Guide*.

Enabling an Oracle Data Service Integrator application to support ADO.NET clients involves doing the following:

- Section 6.1.5, "Generating an Oracle Data Service Integrator Web Services Mapper"
- Section 6.1.7, "Creating a Web Reference in ADO.NET Client by Providing the Oracle Data Service Integrator WSDL URL"

In some cases, of course, there will already be existing operations that you want to make available to an ADO.NET client.

6.1.5 Generating an Oracle Data Service Integrator Web Services Mapper

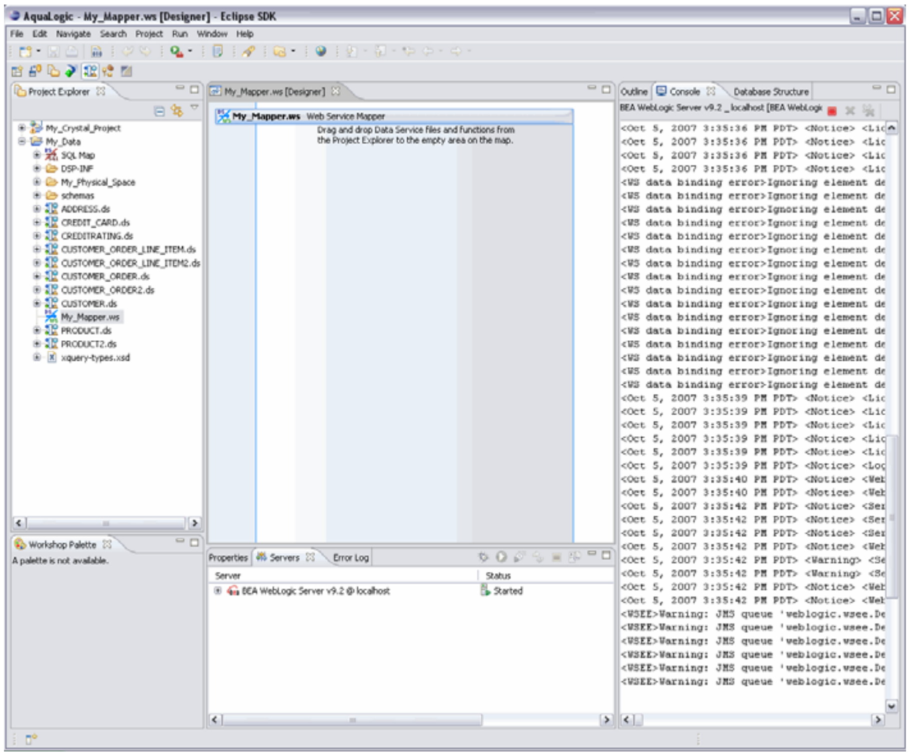
You need to generate an Oracle Data Service Integrator Web Services Mapper that maps data service functions to Web Service operations.

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.

Figure 6–3 Creating a Web Service Mapper



A palette is not available.

Drag and drop Data Service files and functions from the Project Explorer to the empty area on the map.

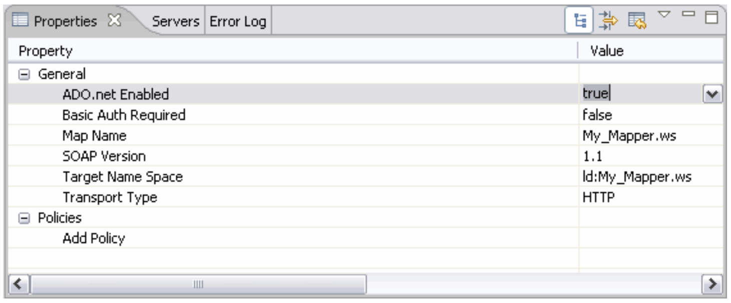
Server	Status
BEA WebLogic Server v9.2 @ localhost	Started

BEA WebLogic Server v9.2 _localhost [BEA WebLogic

```
<Oct 5, 2007 3:35:36 PM PDT> <Notice> <Lic
<Oct 5, 2007 3:35:36 PM PDT> <Notice> <Lic
<Oct 5, 2007 3:35:36 PM PDT> <Notice> <Lic
<Oct 5, 2007 3:35:36 PM PDT> <Notice> <Lic
<WS data binding error>Ignoring element de
<WS data binding error>Ignoring element de
<WS data binding error>Ignoring element de
<WS data binding error>Ignoring element de
<WS data binding error>Ignoring element de
<WS data binding error>Ignoring element de
<WS data binding error>Ignoring element de
<WS data binding error>Ignoring element de
<WS data binding error>Ignoring element de
<WS data binding error>Ignoring element de
<WS data binding error>Ignoring element de
<WS data binding error>Ignoring element de
<WS data binding error>Ignoring element de
<WS data binding error>Ignoring element de
<WS data binding error>Ignoring element de
<WS data binding error>Ignoring element de
<Oct 5, 2007 3:35:39 PM PDT> <Notice> <Lic
<Oct 5, 2007 3:35:39 PM PDT> <Notice> <Lic
<Oct 5, 2007 3:35:39 PM PDT> <Notice> <Lic
<Oct 5, 2007 3:35:39 PM PDT> <Notice> <Lic
<Oct 5, 2007 3:35:39 PM PDT> <Notice> <Lic
<Oct 5, 2007 3:35:39 PM PDT> <Notice> <Log
<Oct 5, 2007 3:35:40 PM PDT> <Notice> <Web
<Oct 5, 2007 3:35:40 PM PDT> <Notice> <Web
<Oct 5, 2007 3:35:42 PM PDT> <Notice> <Ser
<Oct 5, 2007 3:35:42 PM PDT> <Notice> <Ser
<Oct 5, 2007 3:35:42 PM PDT> <Notice> <Ser
<Oct 5, 2007 3:35:42 PM PDT> <Notice> <Web
<Oct 5, 2007 3:35:42 PM PDT> <Warning> <Se
<Oct 5, 2007 3:35:42 PM PDT> <Warning> <Se
<Oct 5, 2007 3:35:42 PM PDT> <Notice> <Web
<Oct 5, 2007 3:35:42 PM PDT> <Notice> <Web
<WSEE>Warning: JMS queue 'weblogic.wsee.De
<WSEE>Warning: JMS queue 'weblogic.wsee.De
<WSEE>Warning: JMS queue 'weblogic.wsee.De
<WSEE>Warning: JMS queue 'weblogic.wsee.De
<WSEE>Warning: JMS queue 'weblogic.wsee.De
<WSEE>Warning: JMS queue 'weblogic.wsee.De
```

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.

3. Click on the Mapper Bar and on the value across from ADO-NET-enabled. Then, select true for the ADO.NET-enabled option.



Property	Value
General	
ADO.net Enabled	true
Basic Auth Required	false
Map Name	My_Mapper.ws
SOAP Version	1.1
Target Name Space	ld:My_Mapper.ws
Transport Type	HTTP
Policies	
Add Policy	

4. Redeploy your project by right-clicking on the project name and checking Deploy Project.

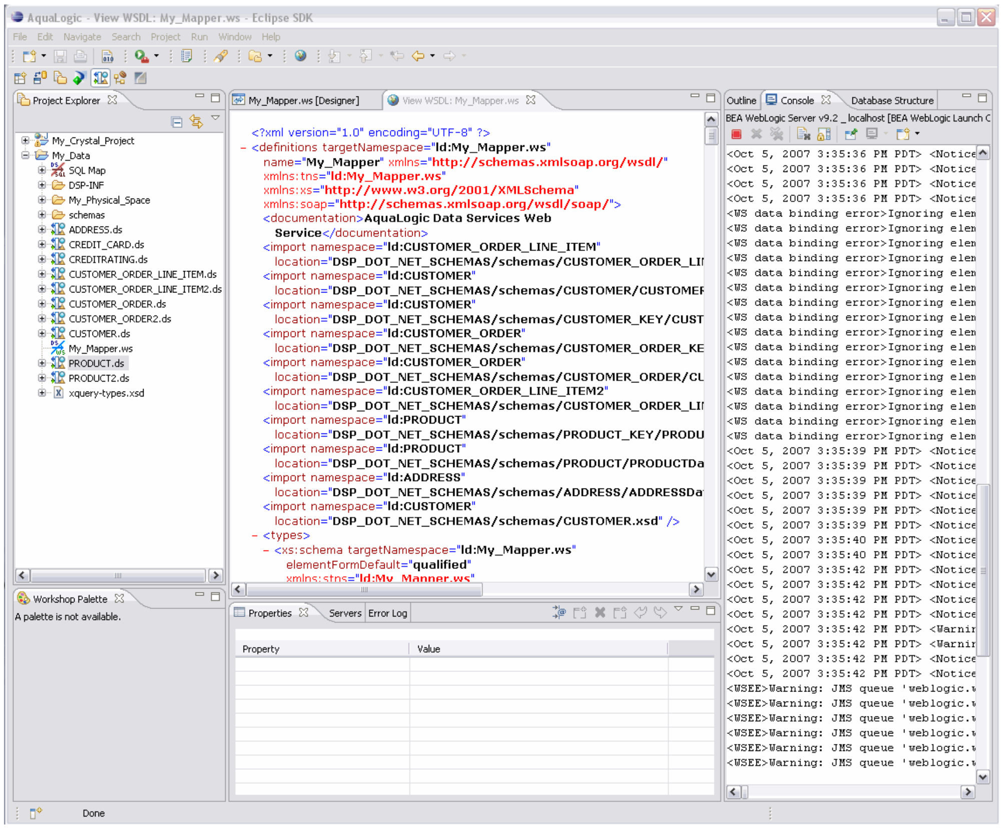
6.1.6 Viewing an ADO.NET-Enabled WSDL

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 Section 6.1.5, "Generating an Oracle Data Service Integrator Web Services Mapper."

2. Select View WSDL.

Figure 6–4 Generated WSDL in Workshop for WebLogic



```
<?xml version="1.0" encoding="UTF-8" ?>
- <definitions targetNamespace="ld:My_Mapper.ws"
    name="My_Mapper" xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="ld:My_Mapper.ws"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
    <documentation>AquaLogic Data Services Web
      Service</documentation>
    <import namespace="ld:CUSTOMER_ORDER_LINE_ITEM"
      location="DSP_DOT_NET_SCHEMAS/schemas/CUSTOMER_ORDER_LI
    <import namespace="ld:CUSTOMER"
      location="DSP_DOT_NET_SCHEMAS/schemas/CUSTOMER/CUSTOMER
    <import namespace="ld:CUSTOMER"
      location="DSP_DOT_NET_SCHEMAS/schemas/CUSTOMER_KEY/CUST
    <import namespace="ld:CUSTOMER_ORDER"
      location="DSP_DOT_NET_SCHEMAS/schemas/CUSTOMER_ORDER_KE
    <import namespace="ld:CUSTOMER_ORDER"
      location="DSP_DOT_NET_SCHEMAS/schemas/CUSTOMER_ORDER/CL
    <import namespace="ld:CUSTOMER_ORDER_LINE_ITEM2"
      location="DSP_DOT_NET_SCHEMAS/schemas/CUSTOMER_ORDER_LI
    <import namespace="ld:PRODUCT"
      location="DSP_DOT_NET_SCHEMAS/schemas/PRODUCT_KEY/PRODU
    <import namespace="ld:PRODUCT"
      location="DSP_DOT_NET_SCHEMAS/schemas/PRODUCT/PRODUCTDa
    <import namespace="ld:ADDRESS"
      location="DSP_DOT_NET_SCHEMAS/schemas/ADDRESS/ADDRESSDa
    <import namespace="ld:CUSTOMER"
      location="DSP_DOT_NET_SCHEMAS/schemas/CUSTOMER.xsd" />
  - <types>
    - <xs:schema targetNamespace="ld:My_Mapper.ws"
        elementFormDefault="qualified"
        xmlns:stns="ld:My_Mapper.ws"
```

```
BEA WebLogic Server v9.2 _ localhost [BEA WebLogic Launch C
<Oct 5, 2007 3:35:36 PM PDT> <Notice
<Oct 5, 2007 3:35:36 PM PDT> <Notice
<Oct 5, 2007 3:35:36 PM PDT> <Notice
<Oct 5, 2007 3:35:36 PM PDT> <Notice
<WS data binding error>Ignoring elem
<WS data binding error>Ignoring elem
<WS data binding error>Ignoring elem
<WS data binding error>Ignoring elem
<WS data binding error>Ignoring elem
<WS data binding error>Ignoring elem
<WS data binding error>Ignoring elem
<WS data binding error>Ignoring elem
<WS data binding error>Ignoring elem
<WS data binding error>Ignoring elem
<WS data binding error>Ignoring elem
<WS data binding error>Ignoring elem
<WS data binding error>Ignoring elem
<WS data binding error>Ignoring elem
<WS data binding error>Ignoring elem
<WS data binding error>Ignoring elem
<Oct 5, 2007 3:35:39 PM PDT> <Notice
<Oct 5, 2007 3:35:39 PM PDT> <Notice
<Oct 5, 2007 3:35:39 PM PDT> <Notice
<Oct 5, 2007 3:35:39 PM PDT> <Notice
<Oct 5, 2007 3:35:39 PM PDT> <Notice
<Oct 5, 2007 3:35:39 PM PDT> <Notice
<Oct 5, 2007 3:35:40 PM PDT> <Notice
<Oct 5, 2007 3:35:40 PM PDT> <Notice
<Oct 5, 2007 3:35:42 PM PDT> <Notice
<Oct 5, 2007 3:35:42 PM PDT> <Notice
<Oct 5, 2007 3:35:42 PM PDT> <Notice
<Oct 5, 2007 3:35:42 PM PDT> <Notice
<Oct 5, 2007 3:35:42 PM PDT> <Warnin
<Oct 5, 2007 3:35:42 PM PDT> <Warnin
<Oct 5, 2007 3:35:42 PM PDT> <Notice
<Oct 5, 2007 3:35:42 PM PDT> <Notice
<WSEE>Warning: JMS queue 'weblogic.w
<WSEE>Warning: JMS queue 'weblogic.w
<WSEE>Warning: JMS queue 'weblogic.w
<WSEE>Warning: JMS queue 'weblogic.w
<WSEE>Warning: JMS queue 'weblogic.w
<WSEE>Warning: JMS queue 'weblogic.w
```

- See Section 6.4.2, "Web Services Description Language (WSDL) File for Microsoft ADO.NET Clients" for information about the format of the WSDL.

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

6.1.7 Creating a Web Reference in ADO.NET Client by Providing the Oracle Data Service Integrator 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 :

   ```
   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 Oracle Data Service Integrator-enabled server is running and your application has sufficient access privileges.

6.2 Adapting Oracle Data Service Integrator 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, Oracle Data Service Integrator 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 Oracle Data Service Integrator server, specifically explaining how to:

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

See the *Oracle Fusion Middleware Data Service Integrator Developer's Guide* for detailed information related to creating and working with XML types.

6.2.1 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 Section 6.2.1.1, "XML Type Requirements for Working With ADO.NET DataSets.")

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.

6.2.1.1 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 Section , "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:

```
<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 an 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.)

6.2.2 References

Further information regarding XML schemas can be found at `http://www.w3.org/TR/xmlschema-0`.

6.3 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:

```
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:

```
<s:schema elementFormDefault="qualified"
```

```
targetNamespace="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.

6.4 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.

6.4.1 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 data service)

Example 6–1 shows an excerpt of a schema—`CustomerDS.xsd`—for a typed DataSet generated from an Oracle Data Service Integrator Customer schema.

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

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

6.4.2 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 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>
   <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 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: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="SUBTOTAL" type="xs.decimal" />
   <xs:element name="SALE_TAX" 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>
   <xs.sequence>
   <xs.complexType>
</xs.schema>
```

Below is a sample CUSTOMER_VIEW_Net WSDL file:

```
<?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>Oracle Data Service Integrator Web Service</documentation>
   <import namespace="ld:logicalDS/CUSTOMER_VIEW" location=
      DSP_DOT_NET_   SCHEMAS/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 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>
         <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>
<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 name="CUSTOMER_VIEWDataSetWrapper">
<xs:sequence>
<xs:any namespace="ld:logicalDS/CUSTOMER_VIEW" />
</xs:sequence>
</xs:complexType>
</xs.schema>
</types>
<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">
   <part name="parameters" element="tns:createCUSTOMER_VIEW" />
</message>
<message name="createCUSTOMER_VIEWOut">
   <part name="parameters" element="tns:createCUSTOMER_VIEWResponse" />
</message>
<message name="updateCUSTOMER_VIEWIn">
   <part name="parameters" element="tns:updateCUSTOMER_VIEW" />
</message>
<message name="updateCUSTOMER_VIEWOut">
   <part name="parameters" element="tns:updateCUSTOMER_VIEWResponse" />
</message>
<message name="deleteCUSTOMER_VIEWIn">
   <part name="parameters" element="tns:deleteCUSTOMER_VIEW" />
</message>
<message name="deleteCUSTOMER_VIEWOut">
   <part name="parameters" element="tns:deleteCUSTOMER_VIEWResponse" />
</message>
```

```
 <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" />
 </operaton>
 <operation name="createCUSTOMER_VIEW">
    <input message="tns:createCUSTOMER_VIEWIn" />
    <output message="tns:createCUSTOMER_VIEWOut" />
 </operaton>
 <operation name="updateCUSTOMER_VIEW">
    <input message="tns:updateCUSTOMER_VIEWIn" />
    <output message="tns:updateCUSTOMER_VIEWOut" />
 </operaton>
 <operation name="deleteCUSTOMER_VIEW">
    <input message="tns:deleteCUSTOMER_VIEWIn" />
    <output message="tns:deleteCUSTOMER_VIEWOut" />
 </operaton>
 </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_viewnet.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>
 <operation name="updateCUSTOMER_VIEW">
    <soap:operation
       soapAction="ld:LogicalDSs/Customer_view_net.ws/updateCUSTOMER_VIEW"
       style="document" />
```

```
            <input>
               <soap:body use="literal" />
            </input>
            <output>
               <soap:body use="literal" />
            </output>
         </operation>
         <operation name="deleteCUSTOMER_VIEW">
            <soap:operation
               soapAction="ld:LogicalDSs/Customer_view_net.ws/deleteCUSTOMER_VIEW"
               style="document" />
            <input>
               <soap:body use="literal" />
            </input>
            <output>
               <soap:body use="literal" />
            </output>
         </operation>
         </binding>
         <service name="Customer_view_netSoapService">
         <port name="Customer_view_netSoapPort"
            binding="tns:Customer_view_netSoapBinding"
         <soap:address
            location="http://172.16.38.38:7001/RTLApp/ODSIWebService/LogicalDSds/
            Customer_view_net.ws" />
      </port>
   </service>
</definitions>
```

7 Advanced Topics

This chapter describes miscellaneous features that are related to client programming with Oracle Data Service Integrator. It includes the following topics:

- Section 7.1, "Accessing Metadata Using Catalog Services"
- Section 7.2, "Filtering, Sorting, and Fine-tuning Query Results"
- Section 7.3, "Using Inverse Functions to Improve Query Performance"

7.1 Accessing Metadata Using Catalog Services

Oracle Data Service Integrator 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 Oracle Data Service Integrator applications, data services, schemas, functions, and relationships.

Catalog Services are also data services; you can view them using the Oracle Data Service Integrator Administration Console, the Oracle Data Service Integrator Palette, and Data Service 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 Oracle Data Service Integrator.
- 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 Oracle Data Service Integrator application. It includes the following topics:

- Section 7.1.1, "Installing Catalog Services"
- Section 7.1.2, "Using Catalog Services"

7.1.1 Installing Catalog Services

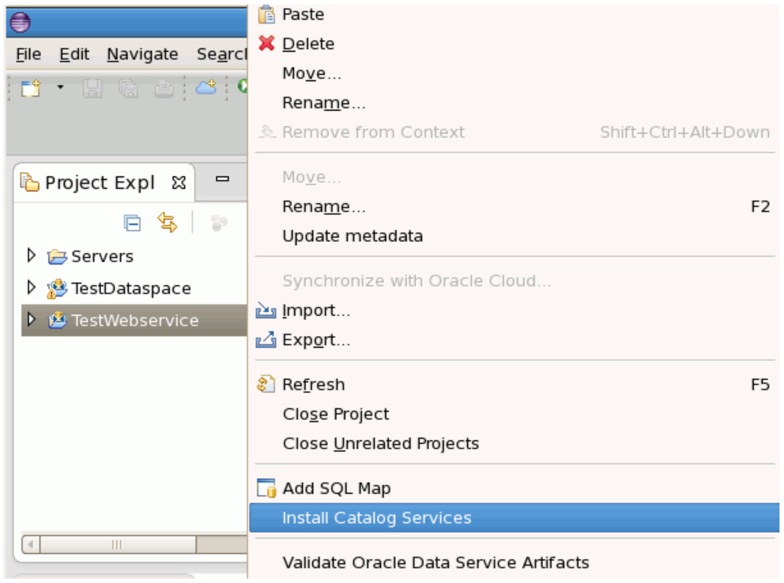
You can install Catalog Services as a project for an Oracle Data Service Integrator application or as a JAR file that is added to the Library folder in Workshop for WebLogic. The Catalog Services project (_catalogservices) contains data services that provide information about the application, folders, 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 Oracle Data Service Integrator application. For more information about the data services and functions available with Catalog Services, refer to Section 7.1.2, "Using Catalog Services."

To install Catalog Services as a project:

1. Right-click the Oracle Data Service Integrator application in .
2. Select the Install Catalog Services (Expanded) option if you want to use the catalog services for development, as shown in Figure 7–1. If you need catalog services only during runtime then select Install Catalog Services (Jar) option.

Figure 7–1 Installing Catalog Services



7.1.2 Using Catalog Services

After installing Catalog Services, the catalog services project, _catalogservices, is created for the Oracle Data Service Integrator 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:

- Section 7.1.2.1, "Application (application.ds)"
- Section 7.1.2.3, "DataServiceRef (DataServiceRef.ds)"
- Section 7.1.2.4, "Folder (folder.ds)"
- Section 7.1.2.5, "Function (Function.ds)"
- Section 7.1.2.6, "Relationship (Relationship.ds)"
- Section 7.1.2.7, "Schema (Schema.ds)"

7.1.2.1 Application (application.ds)

The following table provides the declaration and description for the `getApplication()` function in Application.ds.

Table 7–1 Functions in Application.ds

Function Declaration	Description
getApplication() as schema-element(t1:Application) external;	This function returns the name of the Oracle Data Service Integrator application. It does not take any parameters.

7.1.2.2 DataService (DataService.ds)

Table 7–2 provides declaration and description information for the functions available in DataService.ds.

Table 7–2 Functions in DataService.ds

Function Declaration	Description	Sample Input

getDataServiceRef($arg as element(md:DataService)) as element(md:DataServiceRef) {$arg/md:DataServiceRef}	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
		`<urn:DataService kind="javaFunction" xmlns:acc="ld:RTLAppDataServices/CustomerDB/Customer" xmlns:urn="urn:metadata.ld.bea.com">`
		`<urn:DataServiceRef>`
		`<id>ld:RTLAppDataServices/CustomerDB/Customer.ds</id>`
		`</urn:DataServiceRef>`
		`<returnType name="CUSTOMER" kind="read" quantifier="*" schemaId="ld:RTLAppDataServices/CustomerDB/schemas/CUSTOMER.xsd"/>`
		`<!--Zero or more repetitions:-->`
		`<key>`
		`<!--1 or more repetitions:--> <path>ld:RTLAppDataServices/CustomerDB/Customer.ds</path>`
		`</key>`
		`<!--Zero or more repetitions:-->`
		`<urn:FunctionId name="CUSTOMER" arity="0"/>`
		`</urn:DataService>`

getDataService($x1 as element(t1:DataServiceRef)) as schema-element(t1:DataService)? external	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.
		`<DataServiceRef xmlns="urn:metadata.ld.bea.com">`
		`<id xmlns="">ld:DataServices/CustomerDB/CUSTOMER.ds</id>`
		`</DataServiceRef>`

7.1.2.3 DataServiceRef (DataServiceRef.ds)

The following table provides the declaration and description for the functions available in DataServiceRef.ds.

Table 7–3 Functions in DataService

Function Declaration	Description	Sample Input
getDataServiceRefsByFolder($x1 as xsd:string, $x2 as xsd:boolean) as schema-element(t1:DataServiceRef)* external	This function returns the data services that exist within a folder in the project. You need to specify the path of the project folder and set the boolean value to true for this function.	String parameter = ld:RTLAppDataServices/CustomerDB/ Boolean = true
getDataServiceRefs() as schema-element(t1:DataServiceRef)* external	This function returns the path to all the data services in the project. It does not require any parameters.	No input required.
getDependents($x1 as element(t1:DataServiceRef), $x2 as xsd:boolean) as schema-element(t1:DataServicRef)* external	This function returns the path of the data services on which the specified data service depends. For this function, you need to specify the path of the data service whose dependents you need to determine. For example, if you need to find out the dependents for CUSTOMER.ds then specify the path of the data service as: ld:DataServices/CustomerDB/CUSTOMER.ds	<urn:DataServiceRefdat xmlns:urn="urn:metadata.ld.bea.com"><id>ld:DataServices/CustomerDB/CUSTOMER.ds</id> </urn:DataServiceRef>
getDependencies($x1 as element(t1:DataServiceRef), $x2 as xsd:boolean) as schema-element(t1:DataServiceRef)* external	This function returns the dependencies for the specified data service. For this function, you need to specify the path of the data service whose dependencies you need to determine.	<urn:DataServiceRef xmlns:urn="urn:metadata.ld.bea.com"><id>**ld:DataServices/Demo/CustomerProfile.ds**</id> </urn:DataServiceRef>
getFunctions($x1 as element(t1:DataServiceRef)) as schema-element(t1:Function)* external	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.	<DataServiceRef xmlns="urn:metadata.ld.bea.com"> <id xmlns="">**ld:RTLAppDataServices/CustomerDB/CUSTOMER.ds**</id> </DataServiceRef>
getRelationships($x1 as element(t1:DataServiceRef)) as schema-elemen43t(t1:Relationship)* external	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	DataServiceRef xmlns="urn:metadata.ld.bea.com"> <id xmlns="">**ld:RTLAppDataServices/CustomerDB/CUSTOMER.ds</id>** </DataServiceRef>

Table 7–3 (Cont.) Functions in DataService

Function Declaration	Description	Sample Input
getSchemaRefs($arg as element(t1:DataServiceRef), $transitive as xs:boolean) as element(t1:SchemaRef)* external	For this function, enter the path of the data service and set the boolean value to true for retrieving the list of associated schemas. This function also lists the paths of schemas for data services, which have a relationship with the specified data service.	<urn:DataServiceRef xmlns:urn="urn:metadata.ld.bea.com"> <id>**ld:RTLAppDataServices/CustomerDB/CUSTOMER.ds**</id> </urn:DataServiceRef> .
getDataService($x1 as element(t1:DataServiceRef)) as schema-element(t1:DataService)? external	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.	<urn:DataServiceRef xmlns:urn="urn:metadata.ld.bea.com"> <id xmlns="">**ld:RTLAppDataServices/CustomerDB/CUSTOMER.ds**</id> </urn:DataServiceRef>

7.1.2.4 Folder (folder.ds)

The following table provides the declaration and description for the functions available in Folder.ds.

Table 7–4 Functions in Folder.ds

Function Declaration	Description	Sample Input
getFolder() as schema-element(t1:Folder)* external	This function provides a list of paths of folders and data services that exist within the Oracle Data Service Integrator project. It does not require any parameters.	No input required.
getFolder($x1 as xsd:string, $x2 as xsd:boolean) as schema-element(t1:Folder)* external	This folder returns the paths of all the data services that exists within a specified folder. You need to specify two parameters for this function, which include: ■ Path of the folder such as `ld:RTLAppDataServices/CustomerDB` ■ Boolean value (usually set to true)	■ Parameter 1 (string) = `ld:RTLAppDataServices/CustomerDB` ■ Parameter 2 (boolean) = `true`
getDataServiceRefs($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="">**ld:RTLAppDataServices/CustomerDB**</id> </Folder>

7.1.2.5 Function (Function.ds)

The following table provides the declaration and description for the functions in Function.ds.

Table 7–5 Functions in Function.ds

Function Declaration	Description	Sample Input
getFunctionById($x1 as element(t1:FunctionId)) as schema-element(t1:Function) external4	This function returns the path of the data service and schema along with function arity, function kind and return type information about the specified function. For this function, specify the function ID and arity as input.	<FunctionId name=" **cus:CUSTOMER** " **arity="0"** xmlns:cus="**ld:RTLAppDataServices/CustomerDB/CUSTOMER**" xmlns="urn:metadata.ld.bea.com"/>
getDataService($arg as element(md:Function)) as element(md:DataService	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.	<?xml version="1.0" encoding="UTF-8" ?> <urn:Function kind="**read**" xmlns:acc="**ld:RTLAppDataServices/CustomerDB/CUSTOMER**" xmlns:urn="urn:metadata.ld.bea.com"> <urn:FunctionId arity="**0**" name="**acc:getAll**"></urn:FunctionId> <returnType kind="**element**" name="**urn:Account**" quantifier="**1**" schemaId="**ld:RTLAppDataServices/CustomerDB/schemas/CUSTOMER.xsd**"> </returnType> <urn:DataServiceRef> <id>**ld:RTLAppDataServices/CustomerDB/CUSTOMER.ds**</id> </urn:DataServiceRef> </urn:Function>

Table 7–5 (Cont.) Functions in Function.ds

Function Declaration	Description	Sample Input
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. This function is applicable to navigation functions only.	<?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**"> </returnType> <parameter name="**arg**"> <type kind="**navigate**" name="urn:**DISCOUNT**" quantifier="*" schemaId="**ld:RTLAppDataServices/CustomerDB/CUSTOMER.xsd**"> </type> </parameter> <urn:DataServiceRef> <id>**ld:RTLAppDataServices/CustomerDB/CUSTOMER.ds**</id> </urn:DataServiceRef> <roleName>**DISCOUNT**</roleName> </urn:Function>

Table 7–5 (Cont.) Functions in Function.ds

Function Declaration	Description	Sample Input
getSchemaRefs($x1 as element(t1:Function), $x2 as xsd:boolean) as schema-element(t1:SchemaRef)* external DataServicesRef_	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.	<urn:Function kind="**navigate**" xmlns:acc="**ld:RTLAppDataServices/CustomerDB/CUSTOMER**" xmlns:urn="urn:metadata.ld.bea.com"> <urn:FunctionId name="**acc:getDISCOUNT**" arity="**1**"/> <returnType name="**DISCOUNT**" kind="**element**" quantifier="*****" schemaId="**ld:RTLAppDataServices/CustomerDB/schemas/CUSTOMER.xsd**"/> <!--Zero or more repetitions:--> <parameter name="**arg**"> <type name="**DISCOUNT**" kind="**element**" quantifier="*****" schemaId="**ld:RTLAppDataServices/CustomerDB/schemas/CUSTOMER.xsd**"/> </parameter> <urn:DataServiceRef> <id>**ld:RTLAppDataServices/CustomerDB/Customer.ds**</id> </urn:DataServiceRef> <!--Optional:--> <roleName>**DISCOUNT**</roleName> </urn:Function>

7.1.2.6 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 7–6 Functions in Relationship.ds

Function Declaration	Description	Sample Input
getFunctions($arg as element(md:Relationship)) as element(md:Function)	This function returns the attributes of the function that you specify as input. You need to specify the following parameters for this function: ▪ String parameter = Path of the data service ▪ Function ID ▪ Values for minOccurs and maxOccurs	<urn:Relationship xmlns:acc="**ld:RTLAppDataServices/CustomerDB/CUSTOMER**" xmlns:urn="urn:metadata.ld.bea.com"> <!--1 to 2 repetitions:--> <relationshipTarget roleName="**DISCOUNT**" minOccurs="**1**" maxOccurs="**1**" description=""> <urn:DataServiceRef> <id>**ld:RTLAppDataServices/CustomerDB/CUSTOMER.ds**</id> </urn:DataServiceRef> <!--Zero or more repetitions:--> <urn:FunctionId name="**acc:getDISCOUNT**" arity="**1**"/> </relationshipTarget> </urn:Relationship>
getDataServiceRefs($x1 as element(t1:Relationship)) as schema-element(t1:DataServiceRef)	You need to specify the following parameters for this function: ▪ String parameter = Path of the data service ▪ Function ID ▪ Values for minOccurs and maxOccurs	<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> <!--Zero or more repetitions:--> <urn:FunctionId name="**acc:getDISCOUNT**" arity="**1**"/> </relationshipTarget> </urn:Relationship>

Table 7–6 (Cont.) Functions in Relationship.ds

Function Declaration	Description	Sample Input
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>

7.1.2.7 Schema (Schema.ds)

The following table provides the declaration and description for the functions available in Schema.dsSchemaRef (SchemaRef.ds)

Table 7–7 Functions in Schema.ds

Function Declaration	Description	Sample Input
getSchema($x1 as element(t1:SchemaRef)) as schema-element(t1:Schema)* external	This functions returns the schema attributes of the schema associated with the data service. You need to specify the path of the schema which you need to access as string parameter. For example: ld:RTLAppDataServices/CustomerDB/schemas/CUSTOMER.xsd	<urn:SchemaRef xmlns:urn="urn:metadata.ld.bea.com"> <id>**ld:RTLAppDataServices/CustomerDB/schemas/CUSTOMER.xsd**</id> </urn:SchemaRef>
getSchemaRef($x1 as element(t1:Schema)) as schema-element(t1:SchemaRef)	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: ld:RTLAppDataServices/CustomerDB/schemas/CUSTOMER_TABLE.xsd	<urn:Schema xmlns:urn="urn:metadata.ld.bea.com"> <urn:SchemaRef> <id>**ld:RTLAppDataServices/CustomerDB/schemas/CUSTOMER_TABLE.xsd**</id> </urn:SchemaRef> </urn:Schema>

The following table provides the declaration and description for the functions available in SchemaRef.ds.

Table 7–8 Functions in SchemaRef.ds

Function Declaration	Description	Sample Input
getDependencies($x1 as element(t1:SchemaRef), $x2 as xs:boolean) as schema-element(t1:SchemaRef)* external	This function returns the dependencies of the specified data service. You need to specify the path of the schema for the data service as a string parameter. For example: ld:DataServices/Demo/schemas/CustomerProfile.xsd	<urn:SchemaRef xmlns:urn="urn:metadata.ld.bea.com"> <id xmlns="">**ld:DataServices/Demo/schemas/CustomerProfile.xsd**</id> </urn:SchemaRef> The second parameter is boolean and the value can be either or .
getSchema($x1 as element(md:SchemaRef)) as schema-element(md:Schema)* external	This functions 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: ld:RTLAppDataServices/CustomerDB/schemas/CUSTOMER.xsd	<urn:SchemaRef xmlns:urn="urn:metadata.ld.bea.com"> <id>**ld:RTLAppDataServices/CustomerDB/schemas/CUSTOMER.xsd**</id> </urn:SchemaRef>

7.2 Filtering, Sorting, and Fine-tuning Query Results

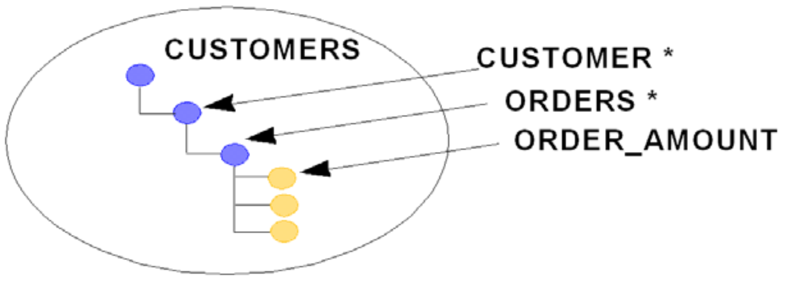
The Filter API enables client applications to apply filtering conditions to the information returned by data service functions. In a sense, filtering allows client applications to extend a data service interface by allowing the application to specify more about how data objects are to be instantiated 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 7–2. The data object contains multiple complex elements (CUSTOMER and ORDERS) and several simple elements, including ORDER_AMOUNT.

Figure 7–2 Nested Value Filtering



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.

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.

7.2.1 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:

- Section 7.2.1.1, "addFilter()"
- Section 7.2.1.2, "createFilter()"
- Section 7.2.1.3, "addOrderBy()"
- Section 7.2.1.4, "setLimit()"

7.2.1.1 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:

```
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 7–9 for information about available operators.
- `everyChild` is an optional parameter. It is set to *false* by default. Specifying true for this parameter indicates that only those child elements that meet the filter criteria will be returned. For example, by specifying an operator of GREATER_THAN (or ">") and a value of 1000, only records for customers where *all* orders are over 1000 will be returned. A customer that has an order amount less than 1000 will not be returned, although other order amounts might be greater than 1000.

7.2.1.2 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:

```
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 7–9 for information about available operators.
- `everyChild` is an optional parameter. It is set to *false* by default. Specifying true for this parameter indicates that only those child elements that meet the filter criteria will be returned.

7.2.1.3 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:

```
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)

7.2.1.4 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:

```
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 maximim number of entries to return (an int value specified as a string, for example, "10").

7.2.1.5 Exploring the Filter Operators

Table 7–9 describes the operators that you can apply to filter conditions.

Table 7–9 Filter Operators

Operator	Usage Note or Example

LESS_THAN	Can also use "<". For example:
	myFilter.addFilter("CUST/CUST_ORDER/ORDER", "CUST/CUST_ORDER/ORDER/ORDER_AMOUNT", "<", "1000");
	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.
BEA_SQL_LIKE	Tests whether a string contains a specified pattern in a manner similar to the SQL LIKE clause.
AND	Compound operator that can apply to more than one filter.
OR	Compound operator that can apply to more than one filter.
NOT	Compound operator that can apply to more than one filter.

Note: Filter API Javadoc, and other Oracle Data Service Integrator APIs are available on the Oracle Technology Network.

7.2.2 Using Filters

Filtering capabilities are available to Mediator and Oracle Data Service Integrator 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:

```
com.bea.ld.filter.FilterXQuery;
```

To use a filter, perform the following steps:

1. Create an `FilterXQuery` object, such as:

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

2. Add a condition to the filter object using the `addFilter()` method.

   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):

   ```
   myFilter.addFilter("CUSTOMERS/CUSTOMER/ORDER",
                      "CUSTOMERS/CUSTOMER/ORDER/ORDER_AMOUNT",
                      ">",
                      "1000");
   ```

3. Use the Mediator API call `setFilter()` to add the filter to a RequestConfig object, and pass the RequestConfig as an argument to the data service operation invocation. For example,

   ```
   RequestConfig config = new RequestConfig();
   config.setFilter(myFilter);
   CUSTOMERDAS custDAS = CUSTOMER.getInstance(ctx, "RTLApp");
   custDS.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."

7.2.3 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:

  ```
  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:

  ```
  FilterXQuery myFilter = new FilterXQuery();
  myFilter.addFilter("CUSTOMERS/CUSTOMER",
                       "CUSTOMERS/CUSTOMER/ORDER/ORDER_AMOUNT",
                        FilterXQuery.GREATER_THAN,"1000");
  ```

- Returns only CUSTOMER objects that have at least one large order and return *only large* ORDER objects:

  ```
  FilterXQuery myFilter = new FilterXQuery();
  myFilter.addFilter("CUSTOMERS/CUSTOMER",
                       "CUSTOMERS/CUSTOMER/ORDER/ORDER_AMOUNT",
                        FilterXQuery.GREATER_THAN,"1000");
  myFilter.addFilter("CUSTOMERS/CUSTOMER/ORDER",
                       "CUSTOMERS/CUSTOMER/ORDER/ORDER_AMOUNT",
                        FilterXQuery.GREATER_THAN,"1000");
  ```

- Returns only CUSTOMER objects for which *every* ORDER_AMOUNT is greater than 1000:

  ```
  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.

7.2.3.1 Specifying a Compound Filter

You can create a filter with two conditions using logical AND and OR operators. Example 7–1 uses the AND operator to apply a combination of filters to a result set, given a data service instance `customerDS`.

Example 7–1 Example of Combining Filters by Using Logical Operators

```
FilterXQuery myFilter = new FilterXQuery();
Filter f1 = myFilter.createFilter("CUSTOMER_PROFILE/ADDRESS/ISDEFAULT",
                                   FilterXQuery.NOT_EQUAL,"0");
Filter f2 = myFilter.createFilter("CUSTOMER/ADDRESS/STATUS",
                                   FilterXQuery.EQUAL, "\"ACTIVE\"");
Filter f3 = myFilter.createFilter(f1,f2, FilterXQuery.AND);
```

7.2.3.2 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.

Example 7–2 provides an example of creating a filter to return customer profiles in ascending order, based on the date each person became a customer.

Example 7–2 Example of Applying an Ordering Filter

```
FilterXQuery myFilter = new FilterXQuery();
myFilter.addOrderBy("CUSTOMER_PROFILE", "CustomerSince" ,FilterXQuery.ASCENDING);
```

Similarly, you can set the maximum number of results to be returned using the `setLimit()` method. Example 7–3 shows how to use the `setLimit()` method to limit the number of active addresses in the result set to 10.

Example 7–3 Example of Applying a Filter that Truncates (Limits) Results

```
FilterXQuery myFilter = new FilterXQuery();
Filter f2 = myFilter.createFilter("CUSTOMER_PROFILE/ADDRESS",
                                   FilterXQuery.EQUAL,"\"INACTIVE\"");
myFilter.addFilter("CUSTOMER_PROFILE", f2);
myFilter.setLimit("CUSTOMER_PROFILE", "10");
```

7.2.4 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 declared by the query string before you can reference the methods in your ad hoc query.

Example 7–4 shows a complete example; the code returns the results of a data service function named getCustomers( ), which is in the namespace:

```
ld:DataServices/RTLServices/Customer
```

Example 7–4 Invoking Data Service Functions using an Ad Hoc Query

```
import com.bea.dsp.das.DataAccessServiceFactory;
import com.bea.dsp.das.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 );
DASResult<Object> result = adHocQuery.executeQuery();
```

Note that the return type of the `executeQuery( )` method is `DASResult<Object>`. The kinds of Objects that can be returned from this `DASResult` are the same as for data service operations invoked using the dynamic mediator API. Simple schema types, such as `xs:int` and `xs:decimal` are returned as Java Objects (`java.lang.Integer`, `java.math.BigDecimal`) according to the mapping described in Table 2–5. Complex types are returned as SDO `DataObjects`.

A single ad-hoc query may return multiple Objects, corresponding to the sequence of items in the result of the XQuery expression. Each of these items are returned as a single Object from calls to `result.next()`.

Because ad-hoc queries are defined inside the client code itself, the Mediator API cannot know the return type of the query. That is why the return value of `executeQuery( )` is `DASResult<Object>` rather than a more specific type such as `DASResult<Customer>`, even if the query only returns Customer DataObjects.

However, if the query does return DataObjects whose schema is defined in a static mediator client JAR, as described in Chapter 3, "Invoking Data Services from Java Clients" and that static mediator client JAR is on the client's CLASSPATH, it is possible to cast the Objects from the `DASResult` to the corresponding typed `DataObject`, just as it is with the dynamic mediator API. For instance,

```
DASResult<Object> result = adHocQuery.executeQuery();
Customer cust = (Customer) cust.next();
```

Note that if the results of the ad-hoc query are not actually Customer DataObjects, the above code throws a ClassCastException when attempting to cast the result of `cust.next()`.

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 Section 3.6.6, "Obtaining the WebLogic JNDI Context for Oracle Data Service Integrator.")

As with the PreparedStatement interface of JDBC, the PreparedExpression interface supports dynamically binding variables in ad hoc query expressions. PreparedExpression provides several methods (bind*Type*( ) methods; see Table 7–10), for binding values of various data types.

Table 7–10 PreparedExpression Methods for Bind Variables

To bind data type of...	Use bind method...
Binary	bindBinary(javax.xml.namespace.QName qname, byte[] abyte0)
Boolean	bindBoolean(javax.xml.namespace.QName qname, boolean flag)
Byte	bindByte(javax.xml.namespace.QName qname, byte byte0)
Date	bindDate(javax.xml.namespace.QName qname, java.sql.Date date)
Calendar	bindDateTime(javax.xml.namespace.QName qname, java.util.Calendar calendar)
DateTime	bindDateTime(javax.xml.namespace.QName qname, java.util.Date date)
DateTime	bindDateTime(javax.xml.namespace.QName qname, java.sql.Timestamp timestamp)
BigDecimal	bindDecimal(javax.xml.namespace.QName qname, java.math.BigDecimal bigdecimal)
double	bindDouble(javax.xml.namespace.QName qname, double d)
Element	bindElement(javax.xml.namespace.QName qname, org.w3c.dom.Element element)
Object	bindElement(javax.xml.namespace.QName qname, java.lang.String s)
float	bindFloat(javax.xml.namespace.QName qname, float f)
int	bindInt(javax.xml.namespace.QName qname, int i)
long	bindLong(javax.xml.namespace.QName qname, long l)

Table 7–10 (Cont.) PreparedExpression Methods for Bind Variables

To bind data type of...	Use bind method...
Object	bindObject(javax.xml.namespace.QName qname, java.lang.Object obj)
short	bindShort(javax.xml.namespace.QName qname, short word0)
String	bindString(javax.xml.namespace.QName qname, java.lang.String s)
Time	bindTime(javax.xml.namespace.QName qname, java.sql.Time time)
URI	bindURI(javax.xml.namespace.QName qname, java.net.URI uri)

To use the bind*Type* methods, pass the variable name as an XML qualified name (*QName*) along with its value; for example:

```
adHocQuery.bindInt(new QName("i"),94133);
```

Example 7–5 shows an example of using a `bindInt()` method in the context of an ad hoc query. Note that all variables to be bound must be explicitly declared as `external` variables in the ad-hoc query, as shown in the example.

Example 7–5 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);
DASResult<Object> result = adHocQuery.executeQuery();
```

Note: For more information on QNames, see `http://www.w3.org/TR/xmlschema-2/#QName`.

Example 7–6 shows a complete ad hoc query example, using the PreparedExpression interface and QNames to pass values in bind methods.

Example 7–6 Sample Ad Hoc Query

```
import com.bea.dsp.das.PreparedExpression;
import com.bea.dsp.das.DataAccessServiceFactory;
import commonj.sdo.DataObject;
import javax.naming.InitialContext;
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 = DataAccessServiceFactory.prepareExpression(
                          getInitialContext(), "RTLApp",  xquery.toString());
           pe.bindString(new QName("p_firstname"), "Jack");
           pe.bindString(new QName("p_lastname"),  "Black");
           DASResult<Object> results = pe.executeQuery();

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

7.3 Using Inverse Functions to Improve Query Performance

When designing and implementing data services, one of the principal goals is to provide a set of abstractions that enable client applications to see and manipulate integrated enterprise data in a clean, unified, meaningful, canonical form. Doing so invariably requires transforming data, which can include restructuring and unifying the schemas and the instance-level data formats of the disparate data sources.

In such cases, names may be reformatted, addresses normalized, and differences in units reconciled, among other operations, to provide application developers (the consumers of data services) with a natural and easily manipulable view of the underlying data. Such transformations, while highly useful to the end consumers of the data, can lead to performance challenges when retrieving underlying data.

When the resulting data is queried, it is crucial for performance that much of the query processing (especially for selections and joins) be pushable to the underlying sources, particularly to relational data sources. This requires updates to the transformed view of the data to be translatable back into appropriate source updates.

Unfortunately, if data transformations are written in a general-purpose programming language, such as Java, both of these requirements can be difficult. This is because, unlike user-written XQuery functions, such general-purpose functions are opaque to the Oracle Data Service Integrator query and update processors.

7.3.1 The Inverse Function Solution

To solve this issue, Oracle Data Service Integrator enables data service developers to register inverse functions with the system, enabling you to define and use general user-defined data transformations without sacrificing query pushdown and updatability. Using this information, Oracle Data Service Integrator is able to perform a reverse transformation of the data when analyzing query predicates or attempting to decompose updates into underlying data source updates.

This means that you can use inverse functions to retain the benefits of high-performance data processing for your logical data without giving up application-oriented convenience data functions. In addition, inverse functions enable automated updates without the need to create Java update overrides.

Note: Using inverse functions effectively and correctly requires careful design. In particular, you must ensure that the functions are true inverses of one another, otherwise Oracle Data Service Integrator may perform undesired operations on your data. While inverse functions are an intuitive and useful idea, be aware that the details require careful attention.

7.3.1.1 Understanding Invertible Functions

The thing to keep in mind when creating inverse functions is that the functions you create need to be truly invertible.

For example, in the following case date is converted to a string value:

```
public static String dateToString(Calendar cal) {
   SimpleDateFormat formatter;
   formatter = new SimpleDateFormat("MM/dd/yyyy hh:mm:ss a");
   return formatter.format(cal.getTime()) ;
}
```

However, notice that the millisecond value is not in the return string value. You get data back but you have lost an element of precision. By default, all values projected are used for optimistic lock checking, so a loss of precision can lead to a mismatch with the database's original value and thus an update failure.

Instead the above code should have retained millisecond values in its return string value, thus ensuring that the data you return is exactly the same as the original value.

7.3.2 How Inverse Functions Can Improve Performance

Here are some additional scenarios where inverse functions can improve performance, especially when large amounts of data are involved:

- Type mismatches. A UK employees database stores date of hire as an integer number; the U.S. employees database stores hire dates in a datetime format. You can convert the integer values to datetime, but then searching on hire date would require fetching every record in the database and sorting at the middleware layer. In this situation, you could use inverse functions to take advantage of each database's hire date index.
- Data Normalization. In order to avoid confusion of UK and U.S. employees, a data service function prepends a country code to the employee IDs of both groups. Again, sorting based on these values will be time consuming since the processing cannot be achieved on the backend without modifying the underlying data.

Inverse functions can be used to remove the prepended country code when pushing the operation down to the underlying database.

- Data Transformation. A logical data service has a fullname operation that concatenates firstname and lastname elements. Any attempt to sort by fullname would be penalized by the required retrieval of information on all customers, followed by local processing of the returned results.
- Data Conversion. There are many cases where values need to be converted to their inverse based on established formulas. For example there could be a requirement that the application retrieve customers by date as an xs:dataTime rather than as a numeric. In this way users could supply date information in a variety of formats. Consider an example where the data architect creates the following XQuery function:

```
declare function tns:getEmpWithFixedHireDate() as element(ns0:usemp)*{
   for $e in ns1:USEMPLOYEES()
      return
         <emp>
            <eid>{fn:data($e1/ID)}</eid>
            <name>{mkName($e1/LNAME, $e1/FNAME)}</name>
            <hiredate>{int2date($e1/HIRED)}</hiredate>
            <salary>)fn:data($e1/SAL)}</salary>
         </emp>
}
```

  Given such a function, issuing a filter query on hiredate, on top of this function, results in inefficient execution since every record from the back-end must be retrieved and then processed in the middle tier.

You can use inverse functions in these and other situations to improve performance, especially when processing sizable amounts of data.

7.3.2.1 A Closer Look

Consider the case of a logical data service that has a fullname operation that concatenates firstname and lastname elements. It is clear that performance would be adversely affected when running the fullname operation against large data sets.

The ideal would be to have a function or functions which decomposed fullname into its indexed components, passes the components to the underlying database, gets the results and reconstitutes the returned results to match the requirements of the fullname operation. In fact, that is the basis of inverse functions.

Of course there are no XQuery functions to magically deconstruct a concatenated string. Instead you need to define, as part of your data service development process, custom functions that inverse engineer fullname.

In many cases complimentary inverse functions are needed. For example, `fahrenheitToCentigrade()` and `centigradeToFahenheit()` would be inverses of each other. Complimentary inverse functions are also needed to support fullname.

In addition to creating inverse functions, you also need to identify inverse functions when defining the data service.

7.3.3 Examining the Inverse Functions Sample

You need to complete the following actions to use inverse functions:

- Create the underlying Java functions

- Create physical data services based on the functions
- Add comparison logic to the data service
- Configure the inverse functions
- Create the data service

7.3.3.1 Creating the Underlying Java Functions

The inverse functions sample includes logic to perform transformations between:

- first/last names and full names
- department names and numbers
- employee IDs and names

The string manipulation logic to manipulate first and last names needed by the inverse function is in the following Java file in the JavaFunctionLib project:

```
JavaFunctionLib/JavaFuncs/NameLib.java
```

This file defines three string manipulation functions.

```
package JavaFuncs;
public class NameLib {
   public static String fullname(String fn, String ln) {
      return (fn == null || ln == null) ? null : (fn + " " + ln);
   }

   public static String firstname(String name) {
      try {
         int sepidx = name.indexOf(' ');
         if (sepidx < 0) return null;
            return name.substring(0, sepidx);
      }
      catch (Exception e) { return null; }
   }

   public static String lastname(String name) {
      try {
         int sepidx = name.indexOf(' ');
         if (sepidx < 0) return null;
            return name.substring(sepidx+1, name.length());
         }
      catch (Exception e) { return null; }
   }

   public static void main(String[] args) {
      String first = "John";
      String last  = "Doe";
      String full  = "John Doe";
      System.out.println(fullname(first, last));
      System.out.println(firstname(full));
      System.out.println(lastname(full));

      System.out.println(firstname(first));
      System.out.println(lastname(first));
   }
}
```

Notice that the function fullname() simply concatenates the first and last names. In contrast, the `firstnname()` and `lastname()` functions deconstruct the resulting full name using the required space in the full name string as the marker identifying the separation between first and last names. Or, put another way, the `fullname()` function is the invertible of `firstname()` and `lastname()`.

Similar functions are available in the `DeptLib` and `EmpIdLib` packages supporting transformations between department names and numbers, and employee IDs and names respectively.

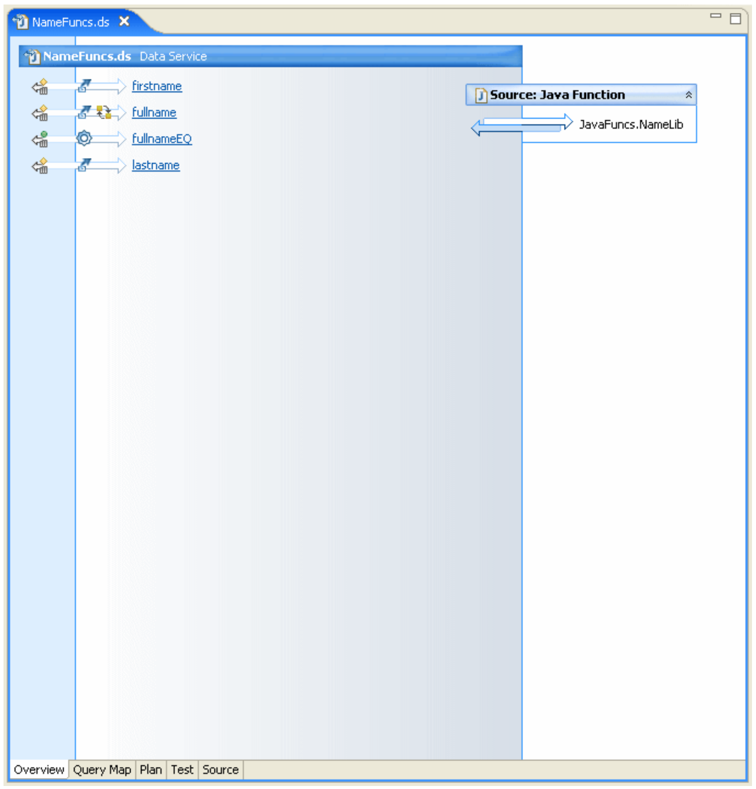
7.3.3.2 Creating the Physical Data Services Based on the Functions

After you have compiled the Java functions, you can create a physical data service from the resulting class file. In the sample, physical data services were created using the `NameLib.class`, `DeptLib.class`, and `EmpIdLib`.class files.

Note: See Create a Physical Data Service from a Java Function at `http://download.oracle.com/docs/cd/E13162_01/odsi/docs10gr3/datasrvc/Create%20a%20Physical%20Data%20Service%20from%20a%20Java%20Function.html` for step-by-step instructions for creating a physical data service from a class file.

In the sample, the resulting operations corresponding to the string manipulation logic reside in the `NameFunc.ds` data service, as illustrated by the following:

Figure 7–3 NameFunc.ds Data Service



Adding Comparison Logic to the Data Service

As is often the case, some additional programming logic is needed. In the case of the sample, a function, `fullnameEQ()`, compares names and returns a Boolean value indicating whether the names are identical.

```
declare function f1:fullnameEQ($full1 as xs:string?, $full2 as xs:string?)
      as xs:boolean? {
   (f1:firstname($full1) eq f1:firstname($full2)) and
   (f1:lastname($full1) eq f1:lastname($full2))
};
```

Note: You can define additional functions for specific conditions, such as "is greater-than" or "is-less-than." Later, when configuring the inverse functions, you can create associations for these conditionals enabling the the XQuery engine to substitute the custom logic for a simple conditional.

7.3.3.3 Configuring the Inverse Functions

You need to configure the inverse functions to perform the reverse transformation of the data.

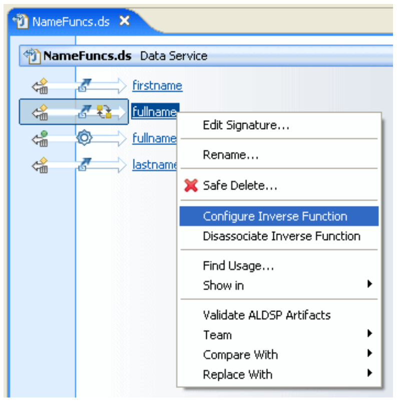
In this particular case, this means that you need to identify an inverse function for each parameter in the `fullname()` function.

Note: Inverse functions can only be defined when the input and output function parameters are atomic types.

To association the parameters of a function with inverse functions:

1. Double-click on the data service in Project Explorer. For example, double-click NameFuncs.ds in the sample.
2. Right-click on the function with which you want to associate inverse functions, and choose Configure Inverse Function. In the sample, right-click the fullname operation and choose Configure Inverse Function.

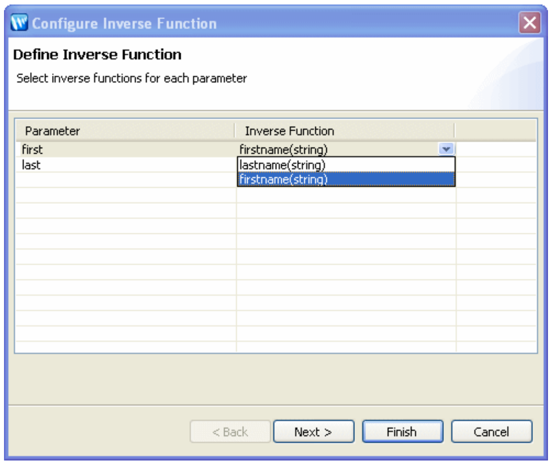
Figure 7–4 Selecting Configure Inverse Function



A dialog appears enabling you to select the inverse functions for each parameter.

3. Choose the corresponding inverse functions for each parameter using the drop-down lists, and click Next.

Figure 7–5 Selecting Configure Inverse Function


Configure Inverse Function
Define Inverse Function
Select inverse functions for each parameter
< Back
Next >
Finish
Cancel

Parameter	Inverse Function
first	firstname(string)
last	

lastname(string)
firstname(string)

4. Specify the equivalent transforms.

7.3.3.4 Associating Custom Conditional Logic with Functions

After you have associated inverse functions with the correct parameters, you may want to associate custom conditional logic with the functions. You do this by substituting a custom function for such generic conditions as eq (is equal to) and gt (is greater than). The following table lists conditional operations available for such transformations.

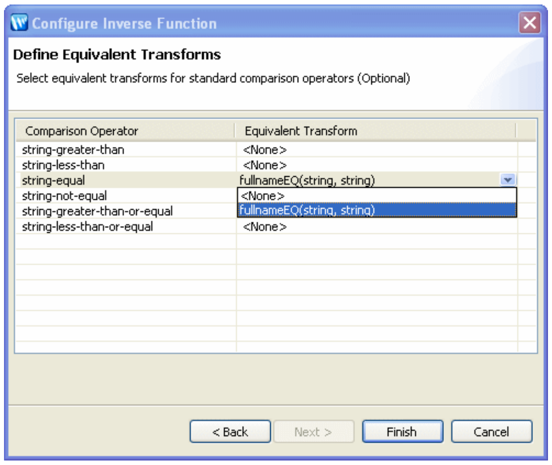
Table 7–11 Conditional Operations

Conditional Operation	Definition
gt	string-greater-than
ne	string-not-equal
lt	string-less-than
ge	string-greater-than-or-equal-to
eq	string-equal
le	string-less-than-or-equal-to

Associating a particular conditional (such as "is greater-than") with a transformational function allows the XQuery engine to substitute the custom logic for a simple conditional. As is always the case with Oracle Data Service Integrator, the original basis of the function does not matter.

It could be created in your data service, or externally in a Java or other routine. In this example the transformational function, fullnameEQ, is in the Java-based physical data service.

Figure 7–6 Defining the Equivalent Transforms


Configure Inverse Function

Define Equivalent Transforms

Select equivalent transforms for standard comparison operators (Optional)

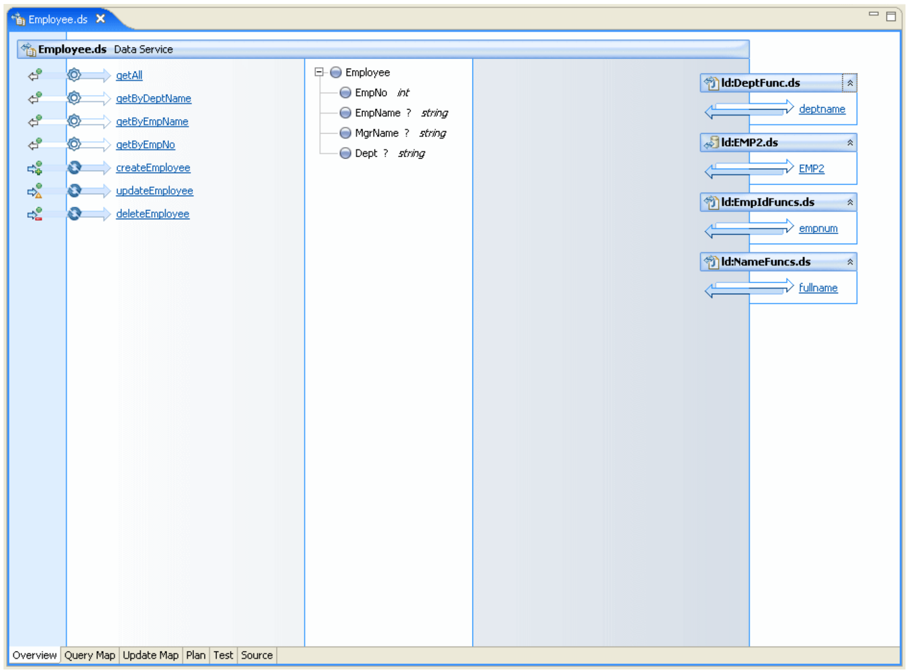
Comparison Operator	Equivalent Transform
string-greater-than	<None>
string-less-than	<None>
string-equal	fullnameEQ(string, string)
string-not-equal	<None>
string-greater-than-or-equal	fullnameEQ(string, string)
string-less-than-or-equal	<None>

7.3.3.5 Creating the Data Service

The final step is to build the data service that contains the operations to create, read, update, and delete the data. In the sample dataspace, this data service, `Employee.ds`, includes operations such as `createEmployee`, `getAll`, `updateEmployee`, and `deleteEmployee`. The data service also includes operations such as `getByDeptName`, `getByEmpName`, and `getByEmpNo`.

The following shows the overview of the `Employee.ds` data service.

Figure 7–7 Employee.ds Data Service



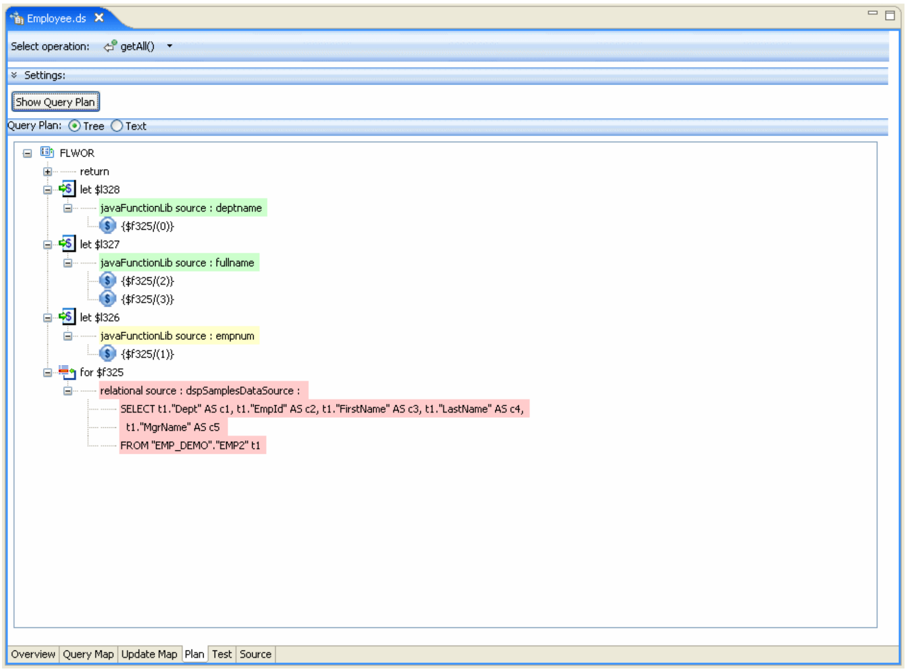
The data service uses XML types associated with the `Employee.xsd` schema. This schema could have been created through the XQuery Editor, through the Oracle Data Service Integrator schema editor, or through a third-party editing tool.

The `getAll()` operation returns all employee records, as shown in the following listing:

```
declare function ns1:getAll() as element(ns1:Employee)* {
   for $EMP2 in emp2:EMP2()
   return
      <tns:Employee>
         <EmpNo>{emp1:empnum($EMP2/EmpId)}</EmpNo>
         <MgrName?>{fn:data($EMP2/MgrName)}</MgrName>
         <Dept?>{dep:deptname($EMP2/Dept)}</Dept>
      </tns:Employee>
};
```

Examining the query plan for the `getAll()` operation, as shown in the following, you can see that predicates are being pushed despite data transformations, because of the use of inverse functions.

Figure 7–8   Query Plan for getAll Operation



```
FLWOR
   return
   let $l328
      javaFunctionLib source : deptname
         {$f325/(0)}
   let $l327
      javaFunctionLib source : fullname
         {$f325/(2)}
         {$f325/(3)}
   let $l326
      javaFunctionLib source : empnum
         {$f325/(1)}
   for $f325
      relational source : dspSamplesDataSource :
         SELECT t1."Dept" AS c1, t1."EmpId" AS c2, t1."FirstName" AS c3, t1."LastName" AS c4,
          t1."MgrName" AS c5
         FROM "EMP_DEMO"."EMP2" t1
```

The case is the same for the other read methods `getByDeptName()`, `getByEmpName()`, and `getByEmpNo()`. Examining the corresponding query plans, you can see that predicates are being pushed regardless of the specific transformations because of the corresponding inverse functions.

Note: The getByEmpName() operation illustrates a typical case where the transformation involves performing a concatenation and the inverse function reverses the operation. In this case, N values are merged into 1 or vice versa. The getByDeptName() and getByEmpNo() operations are both 1:1 examples, transforming between numeric and string values.

7.3.4 How To Set Up the Inverse Functions Sample

This section describes how to import and configure the Oracle Data Service Integrator inverse functions sample dataspace project.

7.3.4.1 Requirements

You can install and work with the inverse function sample on any system with Oracle Data Service Integrator 10gR3 (Oracle WebLogic Server 10gR3) installed.

The inverse function sample is available as a ZIP file from `http://edocs.bea.com/aldsp/docs30/code/InverseFunctions.zip`.

It is recommended that the ZIP file be extracted into an Oracle Data Service Integrator directory such as:

```
<ALDSP_HOME>/user_projects/workspaces/default/InverseFunctionSample
```

7.3.4.2 Importing the Dataspace Project

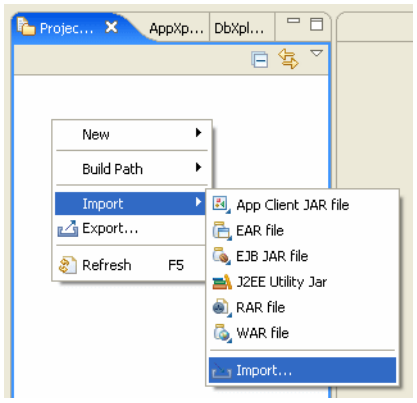
The inverse functions sample consists of two projects:

- The inverse functions sample dataspace
- A Java project that defines the functions used for transforming the data

To import the dataspace project:

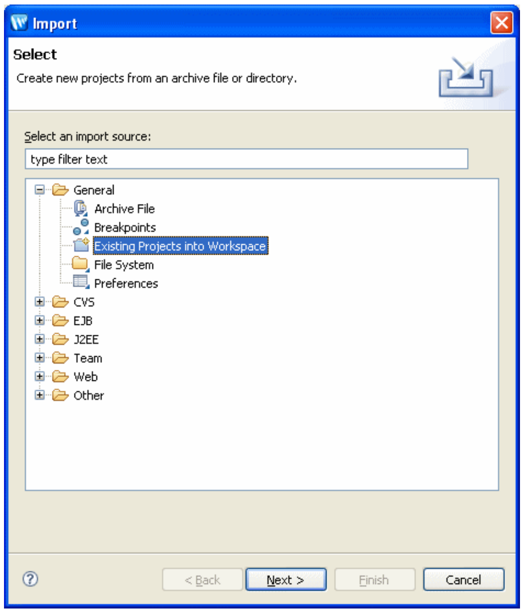
1. Launch Workshop for WebLogic.
2. Right-click in the Project Explorer and choose Import > Import.

Figure 7–9 Import Dialog



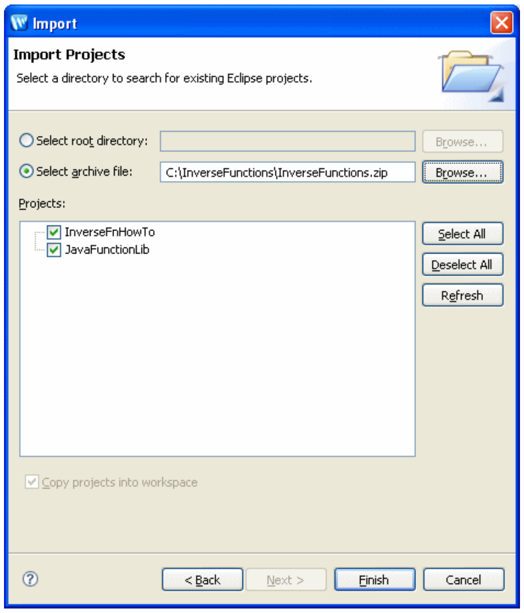
3. Select General > Existing Projects into Workspace, and click Next.

Figure 7–10 Import Menu



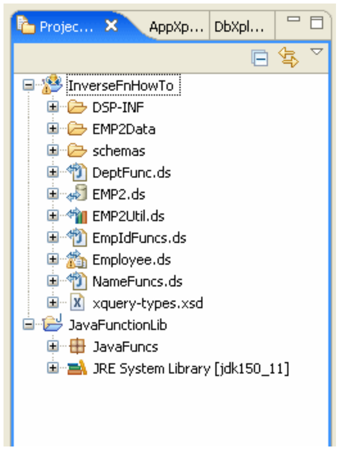
4. Click the Select archive file button, and click Browse.
5. Navigate to the InverseFunctions.zip file, select the file, and click Open.

Figure 7–11 Importing the Projects



6. Click Finish. Workshop for WebLogic imports two projects: `InverseFnHowTo` and `JavaFunctionLib`.

Figure 7–12 Inverse Function Projects



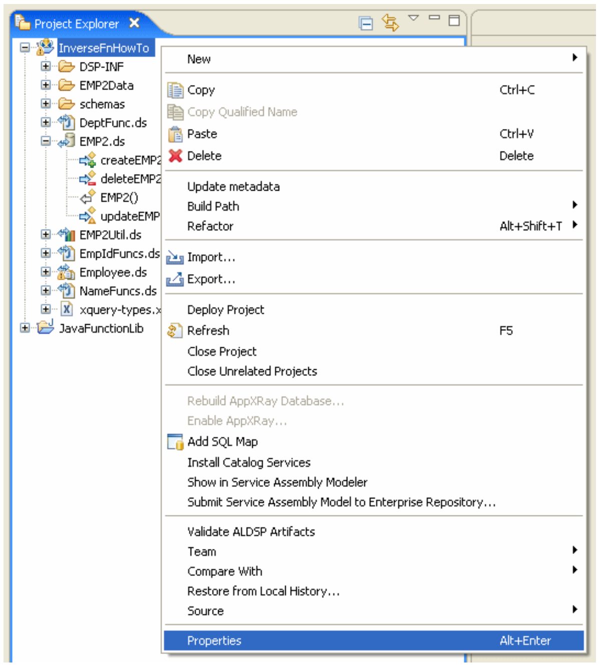
7.3.4.3 Assigning a Targeted Runtime

Before examining the inverse functions sample, you need to start an Oracle Data Service Integrator-enabled server and assigned a targeted runtime server to the project.

To assign a targeted runtime:

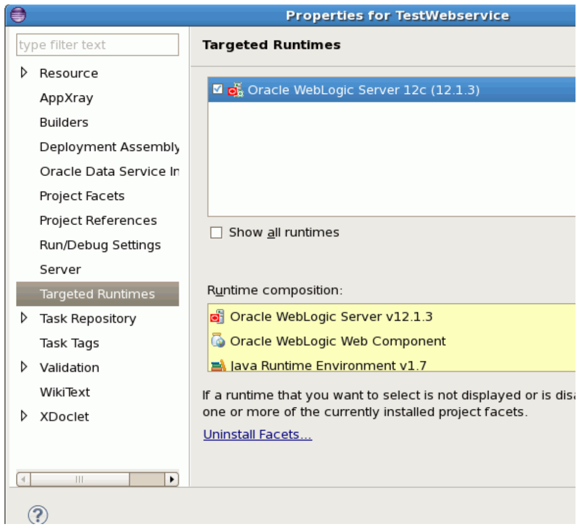
1. Start an Oracle Data Service Integrator-enabled server.
2. In the Project Explorer, right-click the `InverseFnHowTo` project and choose Properties.

Figure 7–13 Choosing Properties



3. Click Targeted Properties. The list of available servers appears.

Figure 7–14 Selecting the Targeted Runtime



☑ Oracle WebLogic Server 12c (12.1.3)

Runtime composition:
- Oracle WebLogic Server v12.1.3
- Oracle WebLogic Web Component
- Java Runtime Environment v1.7

If a runtime that you want to select is not displayed or is dis
one or more of the currently installed project facets.

4. Select a server and click OK. Workshop for WebLogic assigns a runtime server to the project.

7.3.5 Exploring the Inverse Functions Sample

The inverse functions sample consists of two projects.

Table 7–12 Inverse Function Sample Projects

Project	Description
InverseFnHowTo	The inverse functions sample dataspace, including the top-level data service, a relational physical data service, physical data services corresponding to Java transformation functions, and utility data services.
JavaFunctionLib	A Java project that defines the functions used for transforming the da

7.3.5.1 Exploring the Projects

This section describes the principal entities within the two projects that comprise the inverse functions sample.

Exploring the InverseFnHowTo Dataspace Project

The following table describes the data services defined in the `InverseFnHowTo` dataspace project:

Table 7–13 Data Services in the InverseFnHowTo Dataspace

Data Service	Description
Employee	The top-level data service for the project. Examining the query plans for the read methods in the data service shows that predicates are pushed despite data transformations because of inverse functions.
EMP2	The physical data service that accesses the data in the underlying relational database.

Table 7–13 (Cont.) Data Services in the InverseFnHowTo Dataspace

Data Service	Description
DeptFunc EmpIdFuncs NameFuncs	The physical data services corresponding to the Java transformation functions in the JavaFuncLib project.
EMP2Util	Contains functions useful for manipulating and fixing the sample data through the test view.

Exploring the JavaFunctionLib Project

The following table describes the data services defined in the `JavaFunctionLib` project:

Table 7–14 Data Services in the JavaFunctionLib Project

Class	Method	Description
DeptLib	deptname()	Converts a department number to a department name.
	deptno()	Converts a department name to a department number.
EmpIdLib	empid()	Converts an employee number to a string of the following format: EMPid_number.
	empnum()	Converts a string of format EMPid_number to an employee id.
NameLib	firstname()	Extracts the first name from a string (containing a person's full name).
	lastname()	Extracts the last name from a string (containing a person's full name).
	fullname()	Concatenates the first and last name to form a full name.