Introduction to Data Services

This page last changed on Mar 12, 2008 .

Introduction to Data Services

Concepts

- Data in the 21st Century
- Typical Data Service Development Process
- ALDSP - Roles and Responsibilities

How-to...

- Configure the Retail Dataspace Sample Application
- Configure the Retail Dataspace Sample Application for ALDSP 3.2 NEW

Example

- Creating Your First Data Services

Reference

- ALDSP Start Menu
- ALDSP Start Menu for Version 3.2 NEW
- Data Service Types and Functions

Related Topics

- Getting the Most from the ALDSP Eclipse Framework
- Create a Data Service with a Flat Return Type

Data in the 21st Century

This page last changed on Mar 07, 2008 .

Data in the 21st Century

In modern enterprises data is generally readily available. While this has reduced the need to move physical data into data warehouses, data marts, data mines, or other costly replications of existing data structures, the problems of dynamic data integration, immediate secured access and update, data transformation, and data synchronization remain some of the most vexing challenges facing the IT world.

ALDSP provides a comprehensive approach to this challenge by:

- Providing a unified means of importing metadata representing the structure of any data source using its Metadata Import wizard.
- Allowing for the creation of hierarchical data structures from tradition column-row data.
- Providing a query-driven interface to extend the physical model so data specialists can create powerful transformations of existing data and queries.
- Automatically creating data models that introspect physical data structures (and their contents) *in situ*, normalizes representation of diverse data, and allow the representation of the relationship of physical and logical data.
- Maintaining the accuracy of metadata through automated updates from the data source.

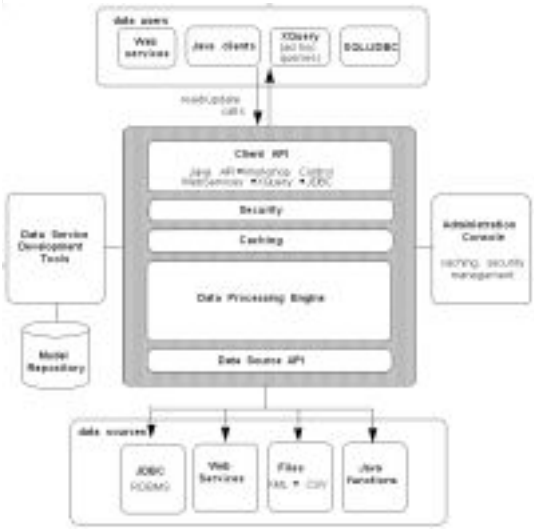
ALDSP can be used to create, refine, and validate logical data structures through a process of importing data sources, creating physical and logical models, and designing queries for use by applications in an infrastructure that provides for easy maintenance, while enhancing security and performance.

Through standardized Service Data Objects (SDO) technology, web-based applications can automatically read and update relational data. Through simple Java programs ALDSP update capabilities can be extended to support any logical data source.

Data Access Integration Architecture

In contemporary enterprise computing, data typically passes through multiple processing and storage layers. While enterprise data can easily be accessed, turning that data into useful information economically and efficiently, particularly updateable information, remains a difficult and high-

maintenance task.



ALDSP approaches the problem of creating integration architectures by building logical data services around physical data sources and then allowing business logic to be added as part of easily maintained, graphically designed XML query functions (also called XQueries).

Using standard protocols such as JDBC, ALDSP automatically introspects data sources, creating physical data services and corresponding schemas that model a physical data source. Optional model diagrams capture relationships between relational data sources, such as primary and foreign keys.

Any WebLogic Workshop application can include ALDSP-based projects. And any application can access ALDSP queries — including update functions — through a mediator API or a ALDSP Control. In the case of relational data, updates can be performed automatically through Service Data Objects (SDO) (For details see Programming with Service Data Objects in the AquaLogic Data Services Platform Client Application Developer's Guide.)

ALDSP provides for the development of integrated queries within any WebLogic Workshop application. Each application can contain multiple ALDSP-based projects, as well as any other types of projects offered by WebLogic Workshop.

Typical Data Service Development Process

This page last changed on Feb 26, 2008 .

Typical Data Service Development Process

The following steps summarize a typical ALDSP-based project development cycle.

1. **Create your project.** Create a ALDSP-based project in a new or existing WebLogic Workshop application.
2. **Create physical data services.** Metadata representing physical data sources can be obtained for any data source that is available through your local application or BEA WebLogic Server. This may include relational data, Web service data, delimited files (spreadsheet data), custom Java functions, and XML files.
3. **Create a data model.** You can optionally build a data model that shows the relationships and cardinality between the data services you have selected (see Modeling Data Services for details). Through the data model, you can also modify and extend relationships between various data services as well as their return type.
4. **Develop data services.** You can elaborate on existing physical data through queries that span multiple physical and/or logical data services. The built-in Query editor ncludes standard XQuery functions and language construct prototypes. Using the editor you can map source elements or transformations to a return type. Queries and data service logic are maintained in a single, editable source file that is fully integrated with your data service (Working with XQuery Source).
5. **Test your function.** You can select any query in the current data service, add a simple or complex parameter (if required), run the query, and see the results. You can also update source data through Test View and create ad hoc queries and procedures.
6. **Review your query plan.** You can view the query plan prior to or after running your query. The query plan describes the generated statements used to retrieve and update data. Execution time statistics are also available.

ALDSP - Roles and Responsibilities

This page last changed on Feb 26, 2008 .

ALDSP: Roles and Responsibilities

The following summarizes typical roles and responsibilities related to creating and maintaining data services.

Physical Data Service Development. Any team member can quickly create a set of *physical data services* from enterprise data sources.

Entity Data Service Development. A data architect with knowledge of the relationships between enterprise data sources can then create data services based on physical and previously developed *logical data services*.

Query Development. Once data services are created, an IT team member can create reusable query functions using the graphical XQuery Editor. The editor is directly tied to a Source View that facilitates code-based modifications to automatically-generated designs.

Deployment. Once data services are developed, they can be deployed from the IDE or by an administrator through the ALDSP Administration Console.

Application Development. Application designers can use data service query functions in their BEA WebLogic applications. Through Service Data Objects (SDO) and the Mediator API or an ALDSP Workshop Control, applications can retrieve and update data, yet remaining insulated from the complexities of managing the underlying data interaction.

Metadata Management. Administrators, architects, and designers can use the Service Explorer for real-time introspection of disparate data source metadata that has been developed through AquaLogic Data Services Platform.

Configure the Retail Dataspace Sample Application

This page last changed on Mar 10, 2008 .

How To Configure the Retail Dataspace Sample Application

This topic describes how to set up the Retail Dataspace Sample Application after completing the installation of ALDSP.

Prerequisites

A prerequisite to configuring the retail dataspace sample application is to have the ALDSP Data Services Studio installed on a supported platform.

Choose the guide for the version you are running:

- Installation Guide for ALDSP 3.0
- Installation Guide for ALDSP 3.2

About Data Services Studio and Eclipse

This tutorial uses the version of Eclipse that is installed with ALDSP.

The Eclipse framework often provides multiple ways of achieving same result. In many cases there is no "correct" or "better" way. In other words, there are often many paths to the same results.

Start Data Services Studio

In version 3.2 Data Services studio is part of Workspace Studio 1.1. If you are using ALDSP 3.2 configuration instructions are available here.

Open the Data Services Studio using the following Windows Start menu command:

```
Start > All Programs > BEA Products > AquaLogic Data Services Platform 3.0 > Data Services Studio
```

In some cases the AquaLogic Data Services Platform 3.0 option may not be nested in the BEA Products section. If this is the case just proceed; it is not a problem.

Select a Workspace

ALDSP projects are called *dataspace* projects. These projects in turn are located in a workspace folder.

The first step in creating a dataspace is to select a workspace.

1. Use the default location:

```
C:\bea92mp2\aldsp_3.0\samples\workspaces\aldsp
```

2. Click OK.

Selecting a Workspace

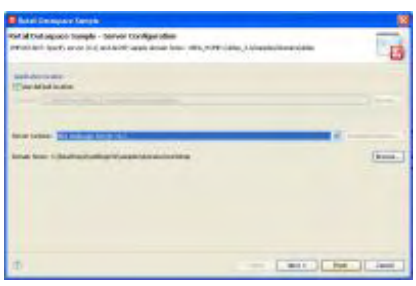
If this is the first time you have opened Studio, the ALDSP Welcome screen appears.

Data Services Studio Welcome Page

In the Install Sample Application section click on:

```
Retail Dataspace Sample
```

Retail Dataspace Sample Dialog



Important

Make sure your domain changed from the Workshop sample domain to the ALDSP samples domain, as shown in the steps in the table below.

Configure the ALDSP-enabled Server Environment

Some simple domain server configuration is required. These steps are described in the following table. (The path information in the table below may be truncated in your browser. Click and select to view the entire path.

Configuring Actions

Step	Dialog	Field	Action	Comment
1.	**Retail Database Sample - Server Configuration**	Server runtime:	click **Installed Runtimes...**	
2.	**Installed Server Runtime Environments**		click **Add...**	
3.	**New Server Runtime**	Type of runtime:	select **BEA WebLogic Server v9.2**	
4.			Next >	
5.	**Define a WebLogic Runtime**	WebLogic Home:	Browse...	
6.	**Browse For Folder**	Folder	locate and select the WebLogic home directory: `<BEA_HOME>/weblogic92`	Example: `c:\bea92mp2\weblogic92`
7.			OK	
8.	**Define a WebLogic Runtime**		Finish	
9.	**Installed Server Runtime Environments**		OK	
10.	**Retail Dataspace Sample - Server Configuration**	Domain home:	Browse...	The Workshop domain is to be replaced by the ALDSP sample domain.

11.	**Browse For Folder**	Select domain home	Locate and click on the ALDSP Sample domain: `<ALDSP_HOME>/samples/domains/aldsp`	Example: `c:\bea92mp2\aldsp30\samples\domains\aldsp`
12.			OK	
13.	**Retail Dataspace Sample - Server Configuration**		Finish	Workspace is built (this may take a few minutes).

Your server should now be properly configured for ALDSP and ready to be started.

Start the Server

An ALDSP-enabled server is a version of WebLogic Server with additional functionality to support ALDSP deployment and runtime. The ALDSP server must be running in order to access sample data and to deploy your project.

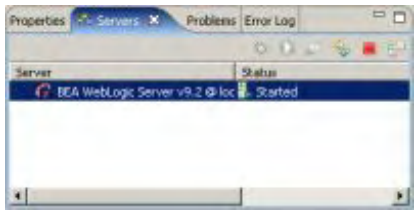
To start your server from Studio:

1. Locate the Servers window. If it isn't visible, use the following option command:

   ```
   Window > Show View > Servers
   ```

2. In the Server window locate BEA WebLogic Server v 9.2@localhost (this may be the only server listed). Its status is: stopped.
3. Right-click on the server name and select Start. (The start-up operation can take several minutes.) Notice the running log of server startup actions in the Console window.

   Server Window

4. After your project has automatically deployed (assuming default settings of Data Services Studio), the dataspace deployment status dialog is displayed. Click:

   OK

See Also

Create Your First Data Services

Configure the Retail Dataspace Sample Application for ALDSP 3.2

This page last changed on Apr 01, 2008 .

How To Configure the Retail Dataspace Sample Application for ALDSP 3.2

This topic describes how to set up the Retail Dataspace Sample Application after completing the installation of ALDSP.

Prerequisites

A prerequisite to configuring the retail dataspace sample application is to have the ALDSP Data Services Studio installed on a supported platform.

Choose the guide for the version you are running:

- Installation Guide for ALDSP 3.0
- Installation Guide for ALDSP 3.2

About WorkSpace Studio, Data Services Studio, and Eclipse

WorkSpace studio was formerly named Data Services Studio.

This tutorial uses the version of Eclipse that is installed with ALDSP.

The Eclipse framework often provides multiple ways of achieving same result. In many cases there is no "correct" or "better" way. In other words, there are often many paths to the same results.

Start WorkSpace Studio

Open WorkSpace Studio using the following Windows Start menu command:

Start > All Programs > *BEA Products* > WorkSpace Studio 1.1

Select a Workspace

ALDSP projects are called *dataspace* projects. These projects in turn are located in a workspace folder.

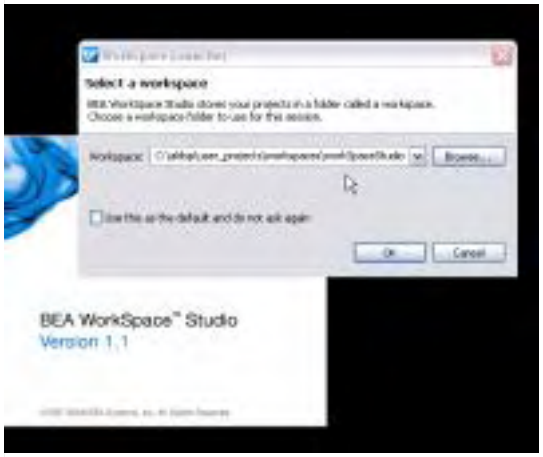
The first step in creating a dataspace is to select a workspace.

1. Use the default location:

   aldsp_home\user_projects\workspaces\workSpaceStudio

2. Click OK.

Selecting a Workspace



If this is the first time you have opened Studio, the WorkSpace Studio screen appears.

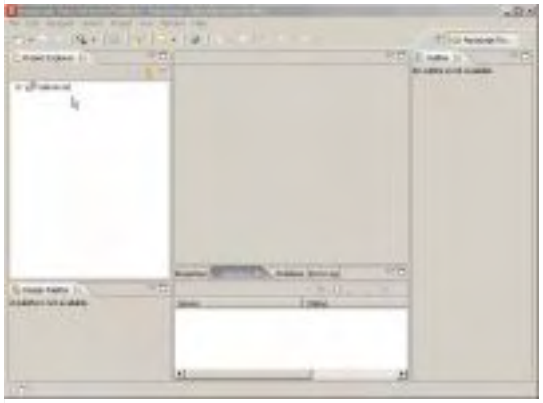
In the Samples section click on:

```
Install Retail Dataspace Sample
```

Retail Dataspace Server Configuration Dialog

1. Click Finish. This will import the RetailDataspace project and find or create a server for the ALDSP sample domain and associate it with the project.
2. Answer Yes to the question about associating your project with the ALDSP perspective.

Initial ALDSP Perspective



Start the Server

An ALDSP-enabled server is a version of WebLogic Server with additional functionality to support ALDSP

deployment and runtime. The ALDSP server must be running in order to access sample data and to deploy your project.

To start your server from Studio:

1. Locate the Servers window. If it isn't visible, use the following option command:

```
Window > Show View > Servers
```

2. In the Servers window locate the ALDSP Samples Server (this may be the only server listed). Notice that its status is Stopped.
3. Right-click on the server name and select Start. (The start-up operation can take several minutes.) Notice the running log of server startup actions in the Console window.

Server Window

Deploy Your Projects

Each project should be deployed to validate the installation.

1. Right-click on ElectronicsWS
2. Choose Deploy Project from the menu. A message should appear indicating successful deployment.
3. Click OK.

Also deploy the RetailDataspace project.

Create the Retail Dataspace Sample Web Application

If Workshop for WebLogic Platform is available, you can also create the Retail Sample Application.

1. From the WorkSpace 1.1 menu select:

File > New > Example...

2. Locate the option:

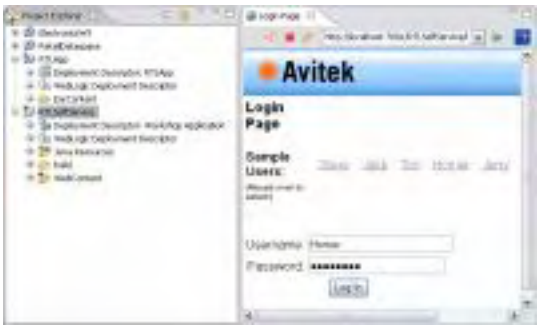
   Retail Dataspace Sample Web Application (WebLogic Workshop only)

3. Click Next, then Finish.
4. In the Project Explorer view, right-click on the RTLSelfService project, and choose:

   Run As > Run on Server

   This will initially deploy you projects and then open the sample Avitek login page.

Avitek Login Page



5. Mouse over one of the names and log in. After a few moments information about the fictitious customer will appear.

See Also

- Create Your First Data Services
- Retail Dataspace Sample Application Guide

Create Your First Data Services

This page last changed on Apr 01, 2008 .

How To Create Your First Data Services

Creating a data service from scratch — as you will if you follow this tutorial — is a good way to get the feel of working with Data Services Studio, as well as other aspects of data services. In the process a logical data service you will also automatically create several physical data services. Physical data services represent physical data sources.

Topics

Goal of the Tutorial

The goal of this tutorial is to illustrate an approach to creating a logical data service, including creating an XML Type (schema), using Data Services Studio. Along the way you will use many of Studio's facilities:

- Drag-and-drop Query Map
- Source Editor
- Test Editor
- Query Plan
- Update Map

This example uses data provided with the Retail Dataspace Sample Application (RTLApp).

Requirements

The requirement for the demonstration project are to develop a logical data service from several physical data services. When run by a client, the data service will return a consolidated view of a particular customer's orders, as well as all the items in each order.

Before You Begin

Before you can begin the tutorial make sure you:

- Properly install ALDSP.

  Reference:
  ALDSP Installation Guide

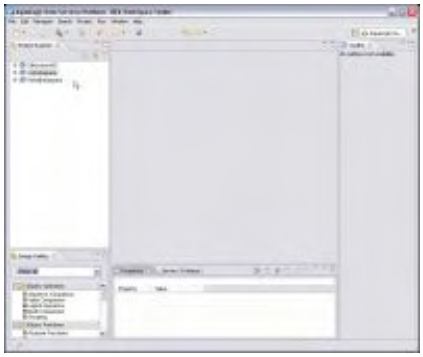
- Configure the Retail Dataspace Sample Application.

  - Configure the Retail Dataspace Sample Application
  - Configure the Retail Dataspace Sample Application for ALDSP 3.2

- Have the application open in Studio and the ALDSP-enable WebLogic 9.2MP2 server running.

  Also describe in Configure the Retail Dataspace Sample Application.

ALDSP Default Perspective After Adding myDataspace



Tip:
Click on image to view it enlarged in a separate window.

Data Services Studio is now part of WorkSpace Studio. Wherever possible the name Studio is used to avoid confusion. However, images will likely reflect the original name.

Creating a Dataspace Project

Data services are created within Studio as Eclipse projects, called dataspace projects. With the ALDSP-enabled server running, the first step is to create a new dataspace project.

1. From the ALDSP menu select:

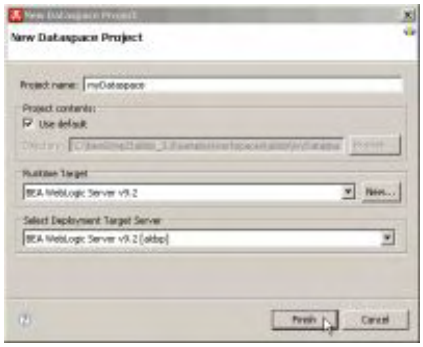
   ```
   File > New > Dataspace Project
   ```

2. Give your project a name such as:

   ```
   myDataspace
   ```

Finish

Creating a New Dataspace Project



Set Up a Folder for Physical Data Services

Data services are typically created inside project folders. The recommended first step in creating one or several data services is to create containers (folders).
In this tutorial two folders will be created:

- One for physical data services.
- One for logical data services.

1. In the Project Explorer window right-click on myDataspace, choose:

   New > Folder

2. Name your folder:

   logical

   Finish

3. Create another folder under myDataspace named:

   physical

   Finish

   Physical data services represent physical data such as tables in relational databases or web services. Logical data services are build upon existing physical or logical data services.

   - Creating and Updating Physical Data Services
   - Designing Logical Data Services

Creating a New Folder

4. Right-click on your new physical folder and choose:

   New > Physical Data Service

Creating Physical Data Services

Physical data services are based on existing data sources.

Whenever you create physical data services, you must first identify the data source. Available options include:

- Relational
- Web Service
- Java Function
- Delimited Data
- XML Data

To take advantage of data provided with the sample application, a relational data source is used.

The sample databases RTLAPPLOMS and RTLCUSTOMER provided with the Retail Sample Application contain five tables. In this section you will create physical data services corresponding to those tables.

Data Sources and Data Services

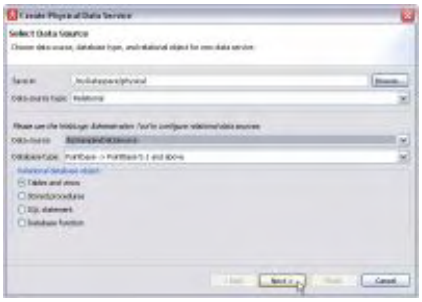
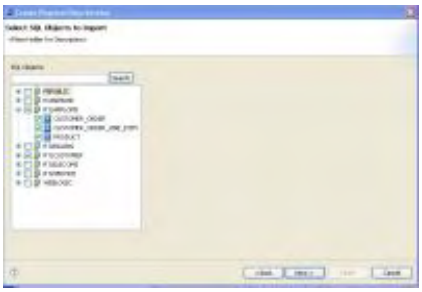
Data Source	Name	Table	Data Service
RTL Appliance Order Management System	**RTLAPPOMS**	• CUSTOMER_ORDER • CUSTOMER_ORDER_LINE_ITEM • PRODUCT	• CUSTOMER_ORDER.DS • CUSTOMER_ORDER_LINE_ITEM.DS • PRODUCT.DS
RTL Customer Data	**RTLCUSTOMER**	• ADDRESS • CUSTOMER	• ADDRESS.DS • CUSTOMER.DS

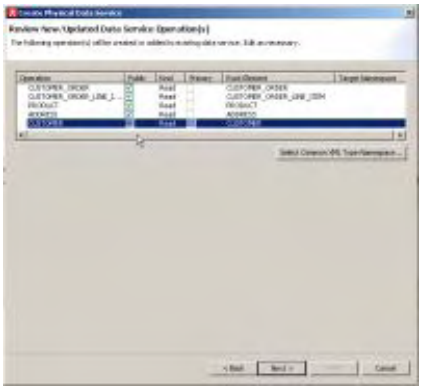
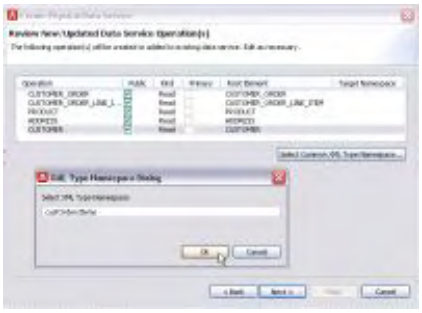
Select a Data Source

The select a data source dialog initially allows you to select a data source type (such as relational or web service). Once that selection is made, additional options appears. The following table lists the actions required to select the relational data sources that will be used throughout this tutorial.

Format similar to that shown in the table below is used to describes the steps needed to work through multi-page wizards.

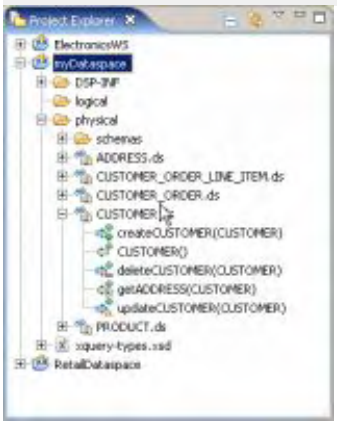
Setting Up Sources for Data Services

Step	Dialog	Field / Column	Action	Comment
	Select Data Source	Save in:		Use default (/myDataspace/physical).
1.		Data source type:	select **Relational**	From dropdown list.
2.		Data source:	select **dspSamplesDataSource**	
3.			Next >	
4.	**Select SQL Sources**	Select SQL objects:	• checkbox next to RTLAPPLOMS • checkbox next to RTLCUSTOMER	Expand (+ symbol to left of data source name) to see tables in the data sources. The information retrieved through introspection of relational data sources is represented as the potential creation of the five primary Read operations, as well as their containing data services. Public operations are available to any authorized calling application.
5.			Next >	
6.	**Review New / Updated Data Service Operation(s)**	Public	mark all five operations **Public** by clicking the checkbox in the Public column	**Note:** The Primary option only applies to create, update, and delete functions.

7.		Select Common XML Type Namespace... button	click the button	Because you are building up an XML Type for your logical data service from several physical data services that each have an underlying XML type, it is necessary for each type to share a namespace.
8.	**XML Type Namspace**	Select XML Type Namespace:	enter **custOrdersItems**	
9.		OK		Notice that the target namespace column now shows the new namespace for your operations.
10.	**Review New/Updated Data Service Operation(s)**		Next >	
11.	**Review New Data Service(s)**		Finish	It is necessary to modify names when: • A data service of the same name already exists in the specified folder. • You are attempt to import two data sources with the same name.In this example, however, there are no name conflicts and no changes are needed.
12.	**Open Data Service Files**	Option to open each new physical data service in Studio	select **No**	

Your new data services appear in your physical folder in the Project Explorer.

Newly Created Data Services



If you expand your new data services you will see that each physical data service has been created with functions corresponding to standard relational operations. For example the CUSTOMER.ds data service contains the following operations:

- createCUSTOMER(CUSTOMER)
- CUSTOMER()
- deleteCUSTOMER(CUSTOMER)
- getADDRESS(CUSTOMER)
- updateCUSTOMER(CUSTOMER)

Some relationship operations (such as getADDRESS(CUSTOMER)) have been created automatically. This operation returns an ADDRESS type when it is passed a CUSTOMER type as a parameter. The operation can be inferred during the data service creation process because ADDRESS contains a foreign key that that is a unique custID in the CUSTOMER data service (and underlying source). Relationship functions are described in detail in the Modeling Data Services Relationships section.

Schemas Directory

You should find a schemas folder adjacent to the newly created data services. This folders contains schema files created during the metadata import process. For relational sources, schemas are created for both the data source (table or view) and the primary keys found during the introspection of the relational source. For example:

- CUSTOMER.xsd
- CUSTOMER_KEY.xsd

If you look in the schemas directory you will see that for each physical data service created, two schemas were created. One representing the physical data service and the other to describe the primary keys in the data source.

Expanded View of Project Explorer

When a logical entity data service is created, it is either:

- Associated with an existing schema *or*
- A return type associated with a function becomes the basis of a generated XML type that is then associated with the data service.

Deploy Your Dataspace Project

You deploy your dataspace project to a server when it is ready to be accessed through a web browser. Deployment is also useful during the project development phase because in its default configuration, when you deploy an ALDSP application in Studio, your project is automatically built. The build process identifies error conditions, if any.

To deploy your application:

1. Right-click on the myDataspace project name in Project Explorer.
2. Choose Deploy Project.

A message indicating successful deployment should appear.

Creating a Logical Data Service

A logical data service can be thought of as a "virtual" data source. Logical data services are built upon existing physical or logical data services.

> ⚠ The ALDSP Retail Sample Application is a good dsource for best practices associated with creating layered data services.

To create a logical data service:

1. Right-click on the folder named logical that you previously created.
2. Select:

```
New > Logical Data Service
```

3. Set the data service name to:

```
CUST_ORDERS_ITEMS
```

Finish

After making these selections, your new entity data service appears in Overview mode.

Since no functions have yet been added to your data service, the work area of the data service is empty.

Options available for creating and testing your new data service appear at the bottom of the workspace. In addition to Overview, you will see the following tabs:

- Query Map
- Update Map
- Plan
- Test
- Source

Attempt To Deploy Your Dataspace Project

There are times when attempts to deploy your data service under development will not be successful. This is expected since as you create your query in the Query Map, source is created simultaneously. (When a data service is in such a state, you will notice a red **x** on its associated icon in Project Explorer.)

Project After Unsuccessful Deployment Effort

You can get the feel of this system if you try to deploy your project now.

1. Right-click on the myDataspace project name in Project Explorer.
2. Choose Deploy Project.

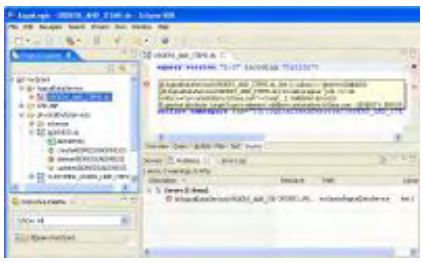
Unlike the previously successful deployment, you will now get a message indicating that your project contains build errors and cannot be deployed.

In this case your newly created ORDERS_AND_ITEMS data service is invalid. You can verify this several ways after clicking OK.

- Inspect your code by clicking on the Source tab.
- Double-click on the error reported in the Problems window.

- Inspect the contents of the Error log window.

Incomplete Logical Data Service Validation Error



Although an error condition exists, you can continue creating on your data service.

Bottom Up or Top Down

Data services can be designed from the top-down or bottom-up. The following table compares these two approaches.

Data Services Design Models

Data Service Design Model	Description
Top-down	The new data service is based on an existing XML Type (schema) that is either drawn from an existing data service or developed externally.
Bottom-up	The new data service is created by: • Identifying one or more data sources. • Building up a Return type in the Query Map. • Saving your data service and associating it with the schema created from the newly designed Return type.

This tutorial uses a bottom-up design.

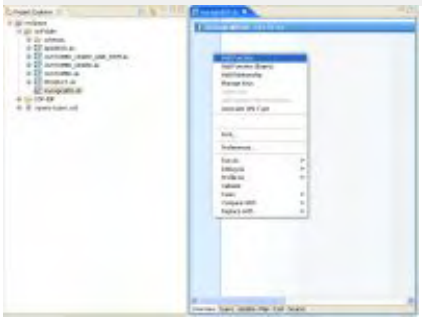
Add an Operation to CUST_ORDERS_ITEMS

The next step is to add a read function to your new data service that will return a document containing all the orders placed by a particular customer, and all the items in each order.

To add your new function:

1. Select the Overview tab.
2. Right-click in the CUST_ORDERS_ITEMS data service's work area.
3. Choose Add Operation... from available options.

Creating a New Operation

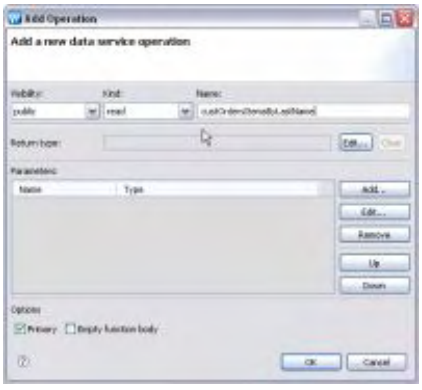


The next steps will create a publicly available Read function for your new data service.

Add Operation Dialog Options

Step	Option	Action	Comment / Reference
	Visibility		Options are private (internal to data service), protected (from public), and public. Default setting is public.
	Kind		All operations are functions other than library procedures. The Read function simply retrieves information from your data source. Default operation is read.
1.	Name	**custOrdersItemsByLastName**	Any valid XML name can be entered; spaces are not allowed.
	Return Type:		Bottom-up designs of a data service create the Return type in the Query Map.
	Parameters:		Can be added here or in the Query Map. Leave unselected.
	Options: Primary		Defines function as the Primary Read function in the entity data service. Default is selected.
	Options: Empty Function Body		Default is not selected.
2.		OK	

Add Operation Dialog



Every artifact and artifact element in Overview has properties. In some cases these properties — such as name and type — are either directly editable or adjustable through dropdown list boxes. The Properties window is, by default, visible in the Studio perspective. If the Properties window is not visible you can retrieve it using the command:

```
Window > Reset Perspective
```

New Data Service Operation and Properties

Building Your Query

Click on the custOrdersItemsByLastName function name in the work area to enter Query Map mode.

Initial Query View

Changes made in the Query Map editor are immediately reflected in source and vice-versa. When there is an error is source, the Query map may not be available. You can typically correct such a condition using the Undo menu option or Ctrl-Z. Alternatively, click the Source tab and edit as needed.

Building Your FLWR Statement Graphically

XQueries are often described as being build upon "FLWR" statements:

- For/Let
- Where
- Return

Changes made in source are immediately rendered graphically in the query map.

Adding Data Sources to Query View - the For/Let Statements

It is through the Query Map that you can bring together representations of existing data sources and associate their elements with the Return type of a new data service.

In the current example your new data service is to provide a consolidated view drawn from the CUSTOMER, CUSTOMER_ORDER, and CUSTOMER_ORDER_LINE_ITEM data services. The Read functions from these physical data services therefore need to be represented in the work area of the new data service.

Follow these steps to add these representations to your Query map:

1. In the **physical** folder expand the following data services:
    - CUSTOMER.ds
    - CUSTOMER_ORDER.ds
    - CUSTOMER_ORDER_LINE_ITEM.ds
2. Drag and drop the Read operations of the following data services CUSTOMER, CUSTOMER_ORDER, and CUSTOMER_ORDER_LINE_ITEM into the query work area. Read operations are identified by the a white-arrow-with-green-ball icon as shown below.

Each of these operational building blocks will become **for** statements in the XQuery description of your new data service.

Data Source Representations in Work Area

The Data Source Representations in Work Area graphic shows the artifacts useful in tailoring your query:

- Data sources are represented in three XQuery For: statements.
- The 'empty empty' element in the Return type is a placeholder for the elements and their type that will eventually be projected.
- The lines from the three statements to the empty global element in the Return type represents current scopings. By adjusting these lines when a Return type is populated you can alter the arrangement of information returned by your query. (Described below.)

Add a Parameter

Parameters can be added when your operation is created or in the Query Map. Parameters can be of simple (primitive) type or complex, such as the XMLtype from another data service.

In this case you create a single xs:string parameter that will allow retrieval of one or more records by a customer's last name.

To add a parameter:

1. In the Query Map work area right-click in a blank area and select:

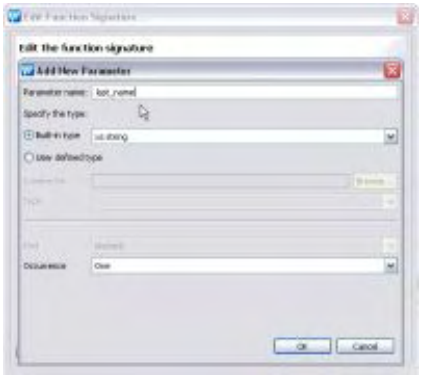
Edit Signature...

2. If asked to save modified resources click OK.
3. In the Parameters section click Add.
4. Complete the Edit Function Signature... dialog.

Edit Function Signature Dialog Options

Step	Field	Action	Comment/Reference
1.	Parameter name	last_name	
	Parameter type		xs:string is the default primitive type.
	Occurance		Default is One.
2.		OK	
3.		OK	In the Edit Function Signatures dialog.

Add New Parameter Dialog



The last_name parameter appears in the work area.

Partial Source of CUST_ORDERS_ITEMS After Addition of Read Functions and last_name Parameter

```
xquery version "1.0" encoding "UTF-8";

(:: pragma ... ::)

declare namespace cus2= "ld:physical/CUSTOMER";
declare namespace cus1= "ld:physical/CUSTOMER_ORDER";
declare namespace ust= "custOrdersItems";
declare namespace cus= "ld:physical/CUSTOMER_ORDER_LINE_ITEM";
declare namespace tns="ld:logical/CUST_ORDERS_ITEMS";

(:: pragma ... ::)

declare function tns:custOrdersItemsByLastName(){
for $CUSTOMER in cus:CUSTOMER()
for $CUSTOMER_ORDER in cus1:CUSTOMER_ORDER()
for $CUSTOMER_ORDER_LINE_ITEM in cus2:CUSTOMER_ORDER_LINE_ITEM()
return
    ()
```

```
};
```

Map Elements to the Return Type

Three icons associated with projecting elements to the Return type appear above the Query Map work area. (You may need to widen your window to see all three icons.)

Mapping Mode Icons

Icon	Mapping Mode	Keyboard equivalent	Description
	**Value**	None.	Maps simple or complex elements to identical values in the Return type. For example, a simple element can be projected to a comparable simple element in the Return type.
	**Overwrite**	Ctrl-Drag object	Overwrites simple or complex element in the Return type with the selected simple or complex element.
	**Append**	Ctrl-Shift-Drag object	Maps simple or complex object as a child to the Return type element it is associated with.

You will use these options to map representations of source data to the Return type of your new data service.

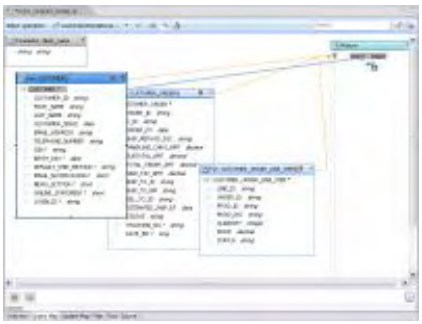
Populating the Return Clause

1. From the three mapping icons in the Select operation line at the top of the query map select the second of the three icons, Overwrite mapping ().
2. Drag the CUSTOMER complex element:

   ```
   CUSTOMER*
   ```

   over the global element placeholder labeled "empty" in the Return type.

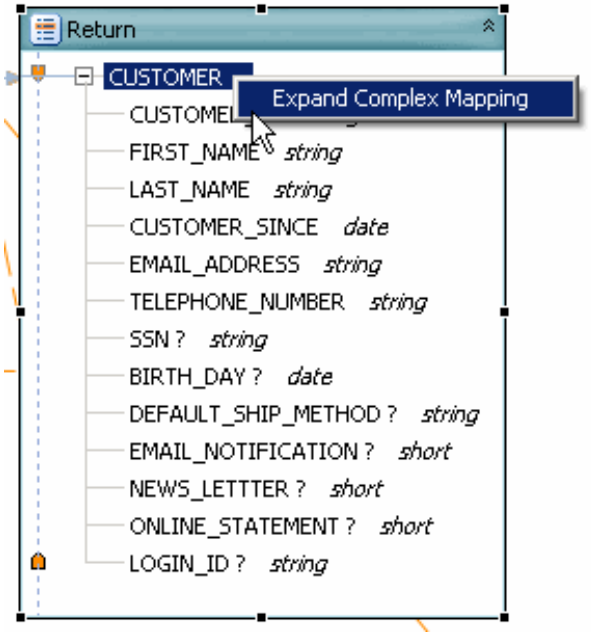
Mapping Complex Element to Return Type



1. Right-click on the new CUSTOMER element in the Return type and select:

   ```
   Expand Complex Mapping
   ```

Expanding Complex Mapping



This gesture is a shortcut for drawing lines from each element in the for statement to the Return type. This gesture is also necessary if you want to add a complex child element to the type. Notice that individual mapping lines now connect each element in the For: node with an element in the Return type. Individual mappings can be added or deleted using drag-and-drop or the Delete key, respectively. The next steps will add elements from the CUSTOMER_ORDER data service to your Return type.

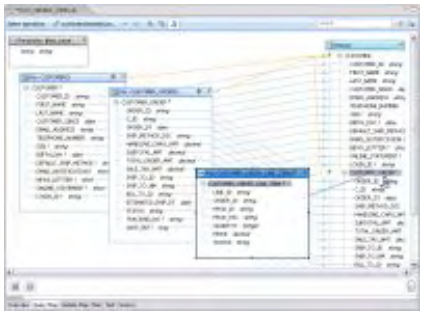
1. Select Append Mapping mode ( ).
2. Drag the CUSTOMER_ORDER complex element:

   CUSTOMER_ORDER*

   over the CUSTOMER element in the Return type. Notice that the CUSTOMER_ORDER global element and the names of its children now appear after the CUSTOMER elements.

3. Expand complex mapping for the CUSTOMER_ORDER global element.
4. From the work area drag the CUSTOMER_ORDER_LINE_ITEM complex element over the CUSTOMER_ORDER element in the Return type.
5. Expand complex mapping for these elements.

Adding Child Elements to Return Type



Set Statement Scoping

Click the Source tab to inspect your generated code. Notice that the Return type contains all three For: statements.

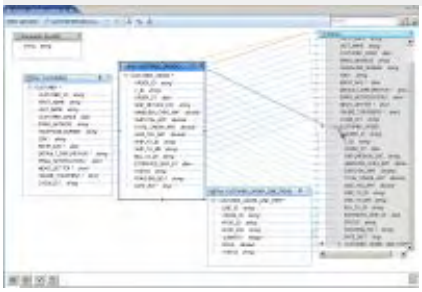
Function cust_orders_items_byLastName(string) in Source View

```
declare function tns:custOrdersItemsByLastName($last_name as xs:string) {
for $CUSTOMER in cus:CUSTOMER()
for $CUSTOMER_ORDER in cus1:CUSTOMER_ORDER()
for $CUSTOMER_ORDER_LINE_ITEM in cus2:CUSTOMER_ORDER_LINE_ITEM()
return
    <ust:CUSTOMER>
        <CUSTOMER_ID>{fn:data($CUSTOMER/CUSTOMER_ID)}</CUSTOMER_ID>
        <FIRST_NAME>{fn:data($CUSTOMER/FIRST_NAME)}</FIRST_NAME>
        <LAST_NAME>{fn:data($CUSTOMER/LAST_NAME)}</LAST_NAME>
        <CUSTOMER_SINCE>{fn:data($CUSTOMER/CUSTOMER_SINCE)}</CUSTOMER_SINCE>
        <EMAIL_ADDRESS>{fn:data($CUSTOMER/EMAIL_ADDRESS)}</EMAIL_ADDRESS>
        <TELEPHONE_NUMBER>{fn:data($CUSTOMER/TELEPHONE_NUMBER)}</TELEPHONE_NUMBER>
        <SSN?>{fn:data($CUSTOMER/SSN)}</SSN>
        <BIRTH_DAY?>{fn:data($CUSTOMER/BIRTH_DAY)}</BIRTH_DAY>
        <DEFAULT_SHIP_METHOD?>{fn:data($CUSTOMER/DEFAULT_SHIP_METHOD)}</DEFAULT_SHIP_METHOD>
        <EMAIL_NOTIFICATION?>{fn:data($CUSTOMER/EMAIL_NOTIFICATION)}</EMAIL_NOTIFICATION>
        <NEWS_LETTTER?>{fn:data($CUSTOMER/NEWS_LETTTER)}</NEWS_LETTTER>
        <ONLINE_STATEMENT?>{fn:data($CUSTOMER/ONLINE_STATEMENT)}</ONLINE_STATEMENT>
        <LOGIN_ID?>{fn:data($CUSTOMER/LOGIN_ID)}</LOGIN_ID>
        {
                <ust:CUSTOMER_ORDER>
                    <ORDER_ID>{fn:data($CUSTOMER_ORDER/ORDER_ID)}</ORDER_ID>
                    <C_ID>{fn:data($CUSTOMER_ORDER/C_ID)}</C_ID>
                    <ORDER_DT>{fn:data($CUSTOMER_ORDER/ORDER_DT)}</ORDER_DT>
                    <SHIP_METHOD_DSC>{fn:data($CUSTOMER_ORDER/SHIP_METHOD_DSC)}</SHIP_METHOD_DSC>
                    <HANDLING_CHRG_AMT>{fn:data($CUSTOMER_ORDER/HANDLING_CHRG_AMT)}</HANDLING_CHRG_AMT>
                    <SUBTOTAL_AMT>{fn:data($CUSTOMER_ORDER/SUBTOTAL_AMT)}</SUBTOTAL_AMT>
                    <TOTAL_ORDER_AMT>{fn:data($CUSTOMER_ORDER/TOTAL_ORDER_AMT)}</TOTAL_ORDER_AMT>
                    <SALE_TAX_AMT>{fn:data($CUSTOMER_ORDER/SALE_TAX_AMT)}</SALE_TAX_AMT>
                    <SHIP_TO_ID>{fn:data($CUSTOMER_ORDER/SHIP_TO_ID)}</SHIP_TO_ID>
                    <SHIP_TO_NM>{fn:data($CUSTOMER_ORDER/SHIP_TO_NM)}</SHIP_TO_NM>
                    <BILL_TO_ID>{fn:data($CUSTOMER_ORDER/BILL_TO_ID)}</BILL_TO_ID>
                    <ESTIMATED_SHIP_DT>{fn:data($CUSTOMER_ORDER/ESTIMATED_SHIP_DT)}</ESTIMATED_SHIP_DT>
                    <STATUS>{fn:data($CUSTOMER_ORDER/STATUS)}</STATUS>
                    <TRACKING_NO?>{fn:data($CUSTOMER_ORDER/TRACKING_NO)}</TRACKING_NO>
                    <DATE_INT?>{fn:data($CUSTOMER_ORDER/DATE_INT)}</DATE_INT>
                    {
                            <ust:CUSTOMER_ORDER_LINE_ITEM>
                                <LINE_ID>{fn:data($CUSTOMER_ORDER_LINE_ITEM/LINE_ID)}</LINE_ID>
                                <ORDER_ID>{fn:data($CUSTOMER_ORDER_LINE_ITEM/ORDER_ID)}</ORDER_ID>
                                <PROD_ID>{fn:data($CUSTOMER_ORDER_LINE_ITEM/PROD_ID)}</PROD_ID>
                                <PROD_DSC>{fn:data($CUSTOMER_ORDER_LINE_ITEM/PROD_DSC)}</PROD_DSC>
                                <QUANTITY>{fn:data($CUSTOMER_ORDER_LINE_ITEM/QUANTITY)}</QUANTITY>
                                <PRICE>{fn:data($CUSTOMER_ORDER_LINE_ITEM/PRICE)}</PRICE>
                                <STATUS>{fn:data($CUSTOMER_ORDER_LINE_ITEM/STATUS)}</STATUS>
                            </ust:CUSTOMER_ORDER_LINE_ITEM>
                    }
                </ust:CUSTOMER_ORDER>
        }
    </ust:CUSTOMER>
};
```

The current query is — in relational terminology — a cross-product or a Cartesian join. Such queries when run are very CPU intensive. In the case of this example, scoping and joining should occur before the query is run.

Using the Query Map you can adjust this quite easily by changing the scoping of the subordinate data services in the Return type, as shown in the following steps.

Adjusting Scoping Rules in the Return Type



1. Return to Query Map mode.
2. With your mouse select the zone icon ( ) in the node:

   For: $CUSTOMER_ORDER()

3. Drag the zone icon over the corresponding CUSTOMER_ORDER element in the Return type.
   Notice that the zone line from the CUSTOMER_ORDER node moves to the subordinate complex type (CUSTOMER_ORDER).
4. Drag the zone icon of CUSTOMER_ORDER_LINE_ITEM to its corresponding element in the Return type.

Nested Zoning in the Return Type

Switch to Source view to verify that the for statements are nested in the Return clause. Now, when a parameter is passed with the operation, all the customers with a particular last name will be returned which contains orders and order line items associated with that customer.

Source View of Return Type with Nested Return Types

```
declare function tns:custOrdersItemsByLastName($last_name as xs:string) {
for $CUSTOMER in cus:CUSTOMER()


return
    <ust:CUSTOMER>
        <CUSTOMER_ID>{fn:data($CUSTOMER/CUSTOMER_ID)}</CUSTOMER_ID>
        <FIRST_NAME>{fn:data($CUSTOMER/FIRST_NAME)}</FIRST_NAME>
        <LAST_NAME>{fn:data($CUSTOMER/LAST_NAME)}</LAST_NAME>
        <CUSTOMER_SINCE>{fn:data($CUSTOMER/CUSTOMER_SINCE)}</CUSTOMER_SINCE>
        <EMAIL_ADDRESS>{fn:data($CUSTOMER/EMAIL_ADDRESS)}</EMAIL_ADDRESS>
        <TELEPHONE_NUMBER>{fn:data($CUSTOMER/TELEPHONE_NUMBER)}</TELEPHONE_NUMBER>
        <SSN?>{fn:data($CUSTOMER/SSN)}</SSN>
        <BIRTH_DAY?>{fn:data($CUSTOMER/BIRTH_DAY)}</BIRTH_DAY>
        <DEFAULT_SHIP_METHOD?>{fn:data($CUSTOMER/DEFAULT_SHIP_METHOD)}</DEFAULT_SHIP_METHOD>
        <EMAIL_NOTIFICATION?>{fn:data($CUSTOMER/EMAIL_NOTIFICATION)}</EMAIL_NOTIFICATION>
        <NEWS_LETTTER?>{fn:data($CUSTOMER/NEWS_LETTTER)}</NEWS_LETTTER>
        <ONLINE_STATEMENT?>{fn:data($CUSTOMER/ONLINE_STATEMENT)}</ONLINE_STATEMENT>
        <LOGIN_ID?>{fn:data($CUSTOMER/LOGIN_ID)}</LOGIN_ID>
        {

                for $CUSTOMER_ORDER in cus1:CUSTOMER_ORDER()
                return
```

```
                <ust:CUSTOMER_ORDER>
                    <ORDER_ID>{fn:data($CUSTOMER_ORDER/ORDER_ID)}</ORDER_ID>
                    <C_ID>{fn:data($CUSTOMER_ORDER/C_ID)}</C_ID>
                    <ORDER_DT>{fn:data($CUSTOMER_ORDER/ORDER_DT)}</ORDER_DT>
                    <SHIP_METHOD_DSC>{fn:data($CUSTOMER_ORDER/SHIP_METHOD_DSC)}</SHIP_METHOD_DSC>
                    <HANDLING_CHRG_AMT>{fn:data($CUSTOMER_ORDER/HANDLING_CHRG_AMT)}</HANDLING_CHRG_AMT>
                    <SUBTOTAL_AMT>{fn:data($CUSTOMER_ORDER/SUBTOTAL_AMT)}</SUBTOTAL_AMT>
                    <TOTAL_ORDER_AMT>{fn:data($CUSTOMER_ORDER/TOTAL_ORDER_AMT)}</TOTAL_ORDER_AMT>
                    <SALE_TAX_AMT>{fn:data($CUSTOMER_ORDER/SALE_TAX_AMT)}</SALE_TAX_AMT>
                    <SHIP_TO_ID>{fn:data($CUSTOMER_ORDER/SHIP_TO_ID)}</SHIP_TO_ID>
                    <SHIP_TO_NM>{fn:data($CUSTOMER_ORDER/SHIP_TO_NM)}</SHIP_TO_NM>
                    <BILL_TO_ID>{fn:data($CUSTOMER_ORDER/BILL_TO_ID)}</BILL_TO_ID>
                    <ESTIMATED_SHIP_DT>{fn:data($CUSTOMER_ORDER/ESTIMATED_SHIP_DT)}</ESTIMATED_SHIP_DT>
                    <STATUS>{fn:data($CUSTOMER_ORDER/STATUS)}</STATUS>
                    <TRACKING_NO?>{fn:data($CUSTOMER_ORDER/TRACKING_NO)}</TRACKING_NO>
                    <DATE_INT?>{fn:data($CUSTOMER_ORDER/DATE_INT)}</DATE_INT>
                    {

                            for $CUSTOMER_ORDER_LINE_ITEM in cus2:CUSTOMER_ORDER_LINE_ITEM()
                            return
                            <ust:CUSTOMER_ORDER_LINE_ITEM>
                                <LINE_ID>{fn:data($CUSTOMER_ORDER_LINE_ITEM/LINE_ID)}</LINE_ID>
                                <ORDER_ID>{fn:data($CUSTOMER_ORDER_LINE_ITEM/ORDER_ID)}</ORDER_ID>
                                <PROD_ID>{fn:data($CUSTOMER_ORDER_LINE_ITEM/PROD_ID)}</PROD_ID>
                                <PROD_DSC>{fn:data($CUSTOMER_ORDER_LINE_ITEM/PROD_DSC)}</PROD_DSC>
                                <QUANTITY>{fn:data($CUSTOMER_ORDER_LINE_ITEM/QUANTITY)}</QUANTITY>
                                <PRICE>{fn:data($CUSTOMER_ORDER_LINE_ITEM/PRICE)}</PRICE>
                                <STATUS>{fn:data($CUSTOMER_ORDER_LINE_ITEM/STATUS)}</STATUS>
                            </ust:CUSTOMER_ORDER_LINE_ITEM>

                    }
                </ust:CUSTOMER_ORDER>

            }
        </ust:CUSTOMER>


    };
```

Creating Joins - the Where Clauses

Where clauses satisfy either specific conditions (such as where $i=5) or join conditions such as:

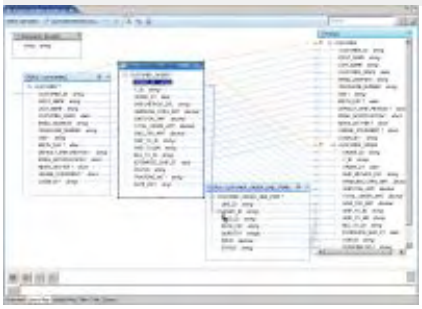
```
    where $CUSTOMER_ORDER/ORDER_ID eq $CUSTOMER_ORDER_LINE_ITEM/ORDER_ID
```

1. Return to Query Map mode.
2. To establish join conditions among your data sources, drag the specified element in one For: statement to the specified element in the target For statement:

Source and element	Target and element
$CUSTOMER/CUSTOMER_ID	$CUSTOMER_ORDER/C_ID
$CUSTOMER_ORDER/ ORDER_ID	$CUSTOMER_ORDER_LINE_ITEM/ ORDER_ID

You may need to move the For: nodes around in the work area to expose the elements.

Setting Up a Join Condition



You can verify your first join clause by clicking on target (CUSTOMER_ORDER) object. Alternatively, you can look in Source view to verify that the new where clause is modifying the CUSTOMER_ORDER_LINE_ITEM type.

```
for $CUSTOMER_ORDER in cus1:CUSTOMER_ORDER() where $CUSTOMER/CUSTOMER_ID eq $CUSTOMER_ORDER/C_ID return
```

Associate a Parameter with a For Node

An additional necessary where condition that directs the query results to a particular customer can be created by adding a parameter to an element in a node. Parameters can be simple or complex.

This project requires use of a single parameter: last_name.

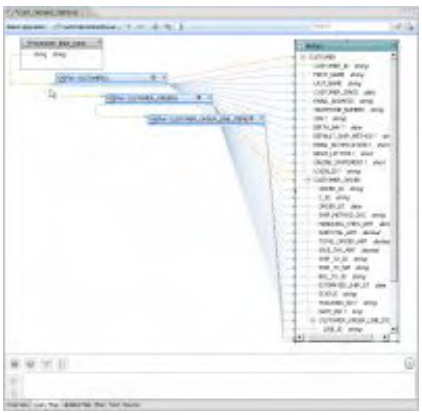
- In the Query Map drag the element:

  ```
  string string
  ```

  in the $last_name parameter over the LAST_NAME element in the CUSTOMER node.

A line connecting the parameter to the node will appear. This will also be reflected in the Query Map Expression editor when you click on the CUSTOMER For: node.

Mapped Parameter and Where Clause



The results of this operation can also be viewed in the Source tab.

```
declare function tns:custOrdersItemsByLastName($last_name as xs:string) as element(ust1:CUST_ORDERS_ITEMS)* {
for $CUSTOMER in cus:CUSTOMER() where $last_name eq $CUSTOMER/LAST_NAME
return ...
```

In Source you will also notice that the for statements now contain where clauses based on your graphical gestures.

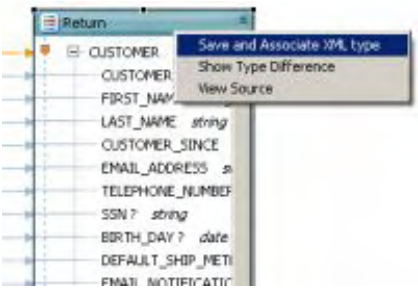
```
for $CUSTOMER in cus:CUSTOMER()
where $last_name eq $CUSTOMER/LAST_NAME
return
...
for $CUSTOMER_ORDER in cus1:CUSTOMER_ORDER()
where $CUSTOMER/CUSTOMER_ID eq $CUSTOMER_ORDER/C_ID
return
...
for $CUSTOMER_ORDER_LINE_ITEM in cus2:CUSTOMER_ORDER_LINE_ITEM()
where $CUSTOMER_ORDER/ORDER_ID eq $CUSTOMER_ORDER_LINE_ITEM/ORDER_ID
return
...
```

Creating, Saving, and Associating the XML Type

Since this entity data service is being created "bottom up", it is not yet associated with an XML Type (schema).

Now that you have a Return type, however, you create a valid XML Type by saving your Return type and associating it with a namespace that is unique to the project.

1. Go to Query Map.
2. Right-click on the Return type's title bar.
3. Select Save and Associate XML type.



4. If asked if you want to save modified resources, choose OK.
5. In the Save and Associate XML Type dialog you will notice that the current name and namespace setting of the Return type conflicts with that of an existing type in the CUSTOMER.xsd file. Change the Name of the Return type global element from:

   ```
   CUSTOMER
   ```

   to:

   ```
   CUST_ORDERS_ITEMS
   ```

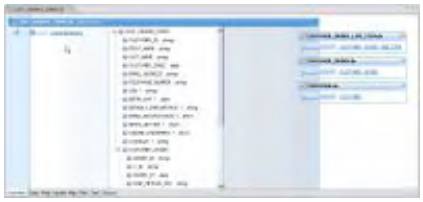
6. Leave the Update references option selected. (This option — which is by default selected — means that XML Type references in source will be updated to reflect the changes you are making.)

Save and Associate XML Type

7. Click Preview. This mode shows what changes will be performed by the name change (refactoring) operation. In this case a new schema file will be created and the target type will be renamed to CUST_ORDER_ITEMS.
8. Click:

   OK
9. Notice that the target type (root element) in your Return type has been renamed.
10. Click Overview; you will see that your entity data service is now associated with an XML type.

Newly Associated XML Type



11. Deploy your project. This deployment should be successful.

Modifying the XML Type

When an XML Type is generated, complex elements by default return a single instance of their type (for example, one CUSTOMER_ORDER will be returned even if there are many).

In order to return all customer orders and all of each orders' line items minor changes to the data service's XML type are needed. The XML markup for this is:

```
maxOccurs="unbounded"
```

In other words, the element returns "*n*", any number of document fragments that meet the criteria.

To modify your new CUST_ORDERS_ITEMS XML Type:

1. Click on the Overview tab, if it is not already selected.
2. Right-click on the topmost element in the XML type: CUST_ORDER_ITEMS.
3. Select Edit Schema. The Eclipse schema editor opens.
4. Click the schema editor's Source tab (below the editor's work area).
5. Locate the first qualified element: CUSTOMER_ORDER.
6. Place your cursor where you want to add the statement (just between the double-quote and the closing angle bracket (**>**) at the end of the line)
7. Enter a space.
8. Activate the code assistant with the combination:

   ```
   Ctrl + spacebar
   ```

   You will get a code completion dialog.

9. Perform the Ctrl+space operation twice, once for the max_occurs, and again to add the unbounded statement. The line now appears as:

```
  <xs:element form="qualified" name="CUSTOMER_ORDER" maxOccurs="unbounded">
```

10. Follow Steps 5-9 for the second qualified element, CUSTOMER_ORDER_LINE_ITEM.
11. Save the CUST_ORDERS_ITEMS.xsd file.

```
File > Save
```

(The modified schema file appears below.)

12. Close the file.

```
  File > Close
```

CUST_ORDERS_ITEMS Schema (XSD File)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="custOrdersItems" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="CUST_ORDERS_ITEMS">
    <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:date"/>
        <xs:element name="EMAIL_ADDRESS" type="xs:string"/>
        <xs:element name="TELEPHONE_NUMBER" type="xs:string"/>
        <xs:element name="SSN" maxOccurs="1" minOccurs="0" type="xs:string"/>
        <xs:element name="BIRTH_DAY" maxOccurs="1" minOccurs="0" type="xs:date"/>
        <xs:element name="DEFAULT_SHIP_METHOD" maxOccurs="1" minOccurs="0" type="xs:string"/>
        <xs:element name="EMAIL_NOTIFICATION" maxOccurs="1" minOccurs="0" type="xs:short"/>
        <xs:element name="NEWS_LETTTER" maxOccurs="1" minOccurs="0" type="xs:short"/>
        <xs:element name="ONLINE_STATEMENT" maxOccurs="1" minOccurs="0" type="xs:short"/>
        <xs:element name="LOGIN_ID" maxOccurs="1" minOccurs="0" type="xs:string"/>
        <xs:element form="qualified" name="CUSTOMER_ORDER" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ORDER_ID" type="xs:string"/>
              <xs:element name="C_ID" type="xs:string"/>
              <xs:element name="ORDER_DT" type="xs:date"/>
              <xs:element name="SHIP_METHOD_DSC" type="xs:string"/>
              <xs:element name="HANDLING_CHRG_AMT" type="xs:decimal"/>
              <xs:element name="SUBTOTAL_AMT" type="xs:decimal"/>
              <xs:element name="TOTAL_ORDER_AMT" type="xs:decimal"/>
              <xs:element name="SALE_TAX_AMT" type="xs:decimal"/>
              <xs:element name="SHIP_TO_ID" type="xs:string"/>
              <xs:element name="SHIP_TO_NM" type="xs:string"/>
              <xs:element name="BILL_TO_ID" type="xs:string"/>
              <xs:element name="ESTIMATED_SHIP_DT" type="xs:date"/>
              <xs:element name="STATUS" type="xs:string"/>
              <xs:element name="TRACKING_NO" maxOccurs="1" minOccurs="0" type="xs:string"/>
              <xs:element name="DATE_INT" maxOccurs="1" minOccurs="0" type="xs:long"/>
              <xs:element form="qualified" name="CUSTOMER_ORDER_LINE_ITEM" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="LINE_ID" type="xs:string"/>
                    <xs:element name="ORDER_ID" type="xs:string"/>
                    <xs:element name="PROD_ID" type="xs:string"/>
                    <xs:element name="PROD_DSC" type="xs:string"/>
```

```
                    <xs:element name="QUANTITY" type="xs:integer"/>
                    <xs:element name="PRICE" type="xs:decimal"/>
                    <xs:element name="STATUS" type="xs:string"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Testing Your Data Service Function

Having created a parameterized read function for your logical data service, you can now test it.

1. Click the Test tab.
2. Using the dropdown in the Select operation field, choose the function:

   ```
   custOrdersItemsByLastName(string)
   ```

3. Enter:

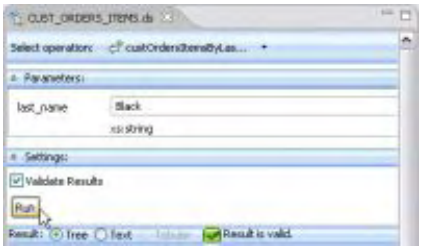
   ```
   Black
   ```

   as the last name parameter.

   Entries are case-sensitive.

4. Click Run. Your project should redeploy successfully and your data then appear.
5. Click the + to the left of CUST_ORDERS_ITEMS to view your data in Tree format. Notice that all the customer's orders are listed under customer information. If you open CUSTOMER_ORDER you will see that items for each order are also listed.

Testing a Parameterized Query



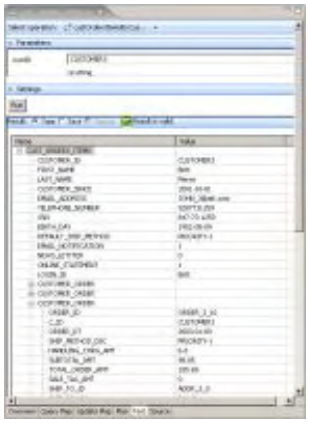
View Test Run Results

Test results from this function can be viewed in two ways:

- Tree
- Text

⚠ The Tabular option is only available for flat (non-nested) results.

Test Run Results in Tree Style Format



Review Test Run Information

When a query is run in the Test editor, you will often be able to access information on your query's performance and the generated SQL (in the case of relational data). Even if a test is unsuccessful, the attempted execution may generate useful audit event statistics.

Query Statistics in the Console Window

The Console window will always contain information on a successfully executed query. Access the Console with:

```
Window > Show View > Other... > General > Console
```

OK

Sample console output is shown below.

Query Details in the Console Window

Adding Create-Update-Delete Functions to Your Data Service

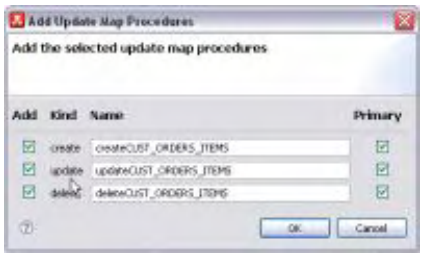
You can also edit results in the Test area. In other words, you can update your data.

To do this an update procedure based on your data service must exist. Until then, the Edit, Submit and Cancel buttons at the bottom of the Test mode work area (Edit Submit Cancel) will be grayed out.

The easiest way to create an update procedure for your logical data service is to generate a default update map procedure. When you do this you will also be given the option of creating delete and insert procedures.

To add the new procedures:

1. In the Overview tab, right-click in the work area choose Add Update Map Procedures...

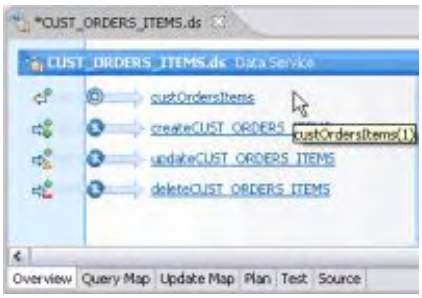


2. Leave the default Add and Primary checkbox options selected for each function.

OK

Notice that the procedures are added to your data service.

Update Map Procedures



Updating Your Results

Now that you have an updateCUST_ORDERS_ITEMS procedure, you can update data -- either through the Test tab or through authorized client applications. Here are the steps:

1. Click on the Test tab and scroll to the top of the window.
2. From the Select operation dropdown select the createCUST_ORDERS_ITEMS(CUST_ORDERS_ITEMS() operation to review the generated type.
3. From the Select operation dropdown select the read function custOrdersItemsByLastName().
4. Run the function using Black as the last_name value.
5. Your project may need to be saved.
6. Click Edit.
7. Expand the top element in the CUST_ORDERS_ITEMS tree.
8. Change the customer's first name from "Jack" to "Sachin" using the built-in line editor. Optionally change the email address as well.
9. Click the Submit button at the bottom of the work area. A message indicating that your data has been successfully submitted

appears.

Changing an Element in Test View

10. Re-run your function to see that the first name field reflects the changes you made.

Reviewing the Query Plan

Once a data service has been successfully deployed, the query plan for the service's read functions can be examined through the Plan tab. The plan can be display in tree or text mode.

1. Click the Plan tab.
2. Choose the custOrdersItemsByLastName(string) function from the Select operation dropdown.
3. Click Show Query Plan.

Tree View of Query Plan

Reviewing the Update Map

After an entity data service is successfully deployed and contains an update function, its update map can be inspected and, as necessary, edited.

- Click the Update Map tab.

CUST_ORDERS_ITEMS Update Map

For more information see:
Understanding Update Maps

Archiving Your Project

You can save your entire project to a ZIP file. Then, when you need to load it again, you can do so with a simple Import operation.

Other examples in the ALDSP documentation use this or similar examples, so having this project available will be make it easier to experiment with other ALDSP faculties.

1. In Project Explorer, right-click on the myDataspace Project.
2. Choose Export.
3. In the Export dialog choose:

   ```
   General > Archive File
   ```

   Next >

   Saving Project to a ZIP File

   !datasrvc:images-datasrvc^Saving Project to a JAR File.gif

   thumbnail!
4. In the Archive file dialog the myDataspace project is pre-selected. Browse to the location where you want to put your archive file.
5. Name your file:

   ```
   myDataspace
   ```

   Leave all other options unchanged.
6. Click Save.

Creating the Archive File

Finish

A file myDataspace.zip will be created in the directory you specified.

Summary

Congratulations! In just a few minutes you have:

- Started ALDSP.
- Created several physical data services based on existing data.
- Created a logical data service based on elements from three physical sources.
- Build a function to retrieve based on information on a particular customer, the customer's orders, and each item in each order.
- Created an XML Type based on the Return type of your function.
- Modified the XML Type to better support a master-detail arrangement of information.
- Tested your results.
- Edited your results.
- Viewed the query plan and the updated map.
- Create an archive file of your dataspace.

About 150 lines of XQuery have been generated.

ALDSP Start Menu

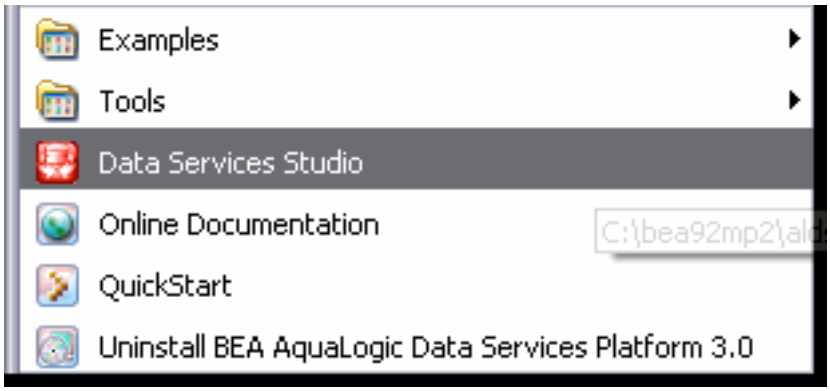
This page last changed on Feb 26, 2008 .

ALDSP Start Menu Artifacts

The ALDSP Start menu provides easy access to components used to develop ALDSP data services. Access is from the Windows Start menu:

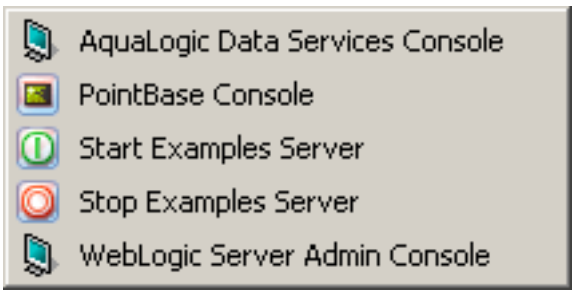
Start > ALDSP

ALSDP Start Menu



The following table describes the menu options available from the main ALDSP menu.

ALDSP Start Menu Options

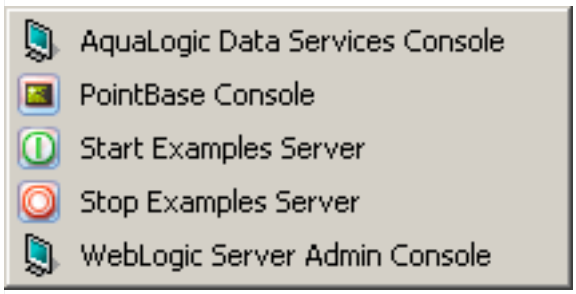
Option	Usage
Examples	
	Provides access to the Examples Menu.

Tools  	Provides access to the WebLogic Configuration Wizard where you can create a new ALDSP-based domain or extend an existing domain to support ALDSP. ALDSP Administrator's Guide
Data Services Studio	ALDSP Eclipse-based IDE
Eclipse	Eclipse used by ALDSP. May be the version of Eclipse installed with ALDSP or the default version in you did not choose to install Eclipse with ALDSP (custom install).
Online Documentation	The ALDSP e-docs home page.
QuickStart	Provides links to help get started with installed BEA products
SmartUpdate	Used in conjunction with your BEA Support ID to download any applicable patches and maintenance packs.
Uninstall BEA AquaLogic	Uninstalls ALDSP.

Examples Menu

The ALDSP Examples menu provides access to server, console, and database operations used by the RTLApp sample application.

ALDSP Examples Menu



The following table describes the ALDSP Examples menu options.

ALDSP Examples Menu Options

Option	Usage
AquaLogic Data Services Console	Provides access to the HTML ALDSP Administration Console. ALDSP Administrator's Guide

PointBase Console	Provides access to the PointBase Console. The PointBase database drives the sample data in the RTLApp demo.
Start Examples Server	Starts the examples server provided with ALDSP. The server can also be started from within Eclipse.
Stop Examples Server	Stops the examples server provided with ALDSP. The server can also be stopped from within Eclipse.
WebLogic Server Admin Console	Provides access to the WebLogic Admin Console where some ALDSP data sources are identified and some security configuration is managed.

ALDSP Start Menu for Version 3.2

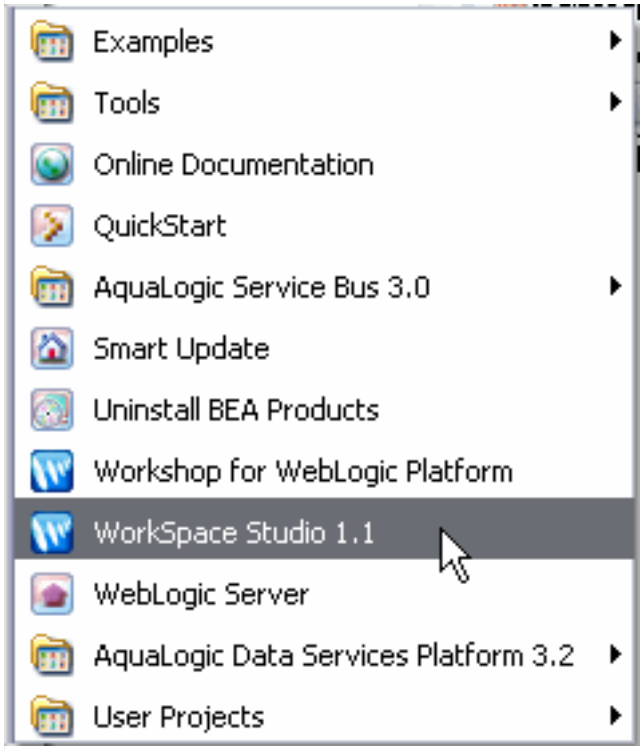
This page last changed on Mar 18, 2008 .

ALDSP Start Menu Artifacts for Version 3.2

The Start menu provides easy access to components used to develop ALDSP data services:

Start > BEA Products

Start Menu for Typical BEA Products Menu with ALDSP 3.2 Installed

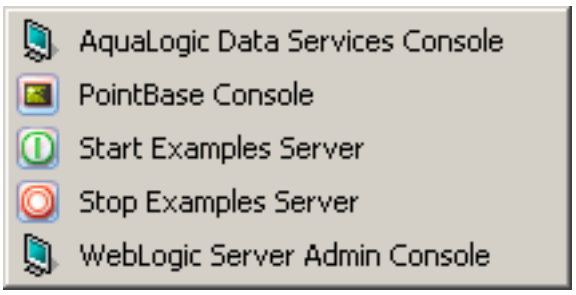


The following table describes the menu options available from the main ALDSP menu.

ALDSP Start Menu Options

Option	Usage

Examples



Provides access to the ALDSP Examples server and Examples Menu.

Tools

Provides access to the WebLogic Configuration Wizard where you can create a new ALDSP-based domain or extend an existing domain to support ALDSP.

ALDSP Administrator's Guide

Data Services Studio

Workspace Studio ("Studio") is the ALDSP Eclipse-based IDE plugin. It was formerly called Data Services Studio.

Examples

Provides links to BEA product examples, including ALDSP (see below).

Online Documentation

The ALDSP e-docs home page can be accessed from:

Start > BEA Products > AquaLogic Data Services Platform 3.2 > Online Documentation

QuickStart

Provides links to help get started with installed BEA products. For ALDSP, the example domain is started.

SmartUpdate

Used in conjunction with your BEA Support ID to download any applicable patches and maintenance packs.

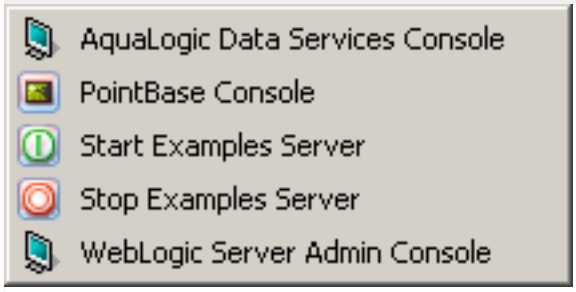
Uninstall BEA AquaLogic

Options for uninstalling various BEA products, including ALDSP.

Examples Menu

The ALDSP Examples menu provides access to server, console, and database operations used by the RTLApp sample application.

ALDSP Examples Menu



The following table describes the ALDSP Examples menu options.

ALDSP Examples Menu Options

Option	Usage
AquaLogic Data Services Console	Provides access to the HTML ALDSP Administration Console. ALDSP Administrator's Guide
PointBase Console	Provides access to the PointBase Console. The PointBase database drives the sample data in the RTLApp demo.
Start Examples Server	Starts the examples server provided with ALDSP. The server can also be started from within Eclipse.
Stop Examples Server	Stops the examples server provided with ALDSP. The server can also be stopped from within Eclipse.
WebLogic Server Admin Console	Provides access to the WebLogic Admin Console where some ALDSP data sources are identified and some security configuration is managed.

Data Service Types and Functions

This page last changed on Mar 07, 2008 .

Data Service Types and Functions

ALDSP functions can have a number of attributes. This section describes those attributes and the conditions under which they are applicable.

ALDSP Data Service Types and Attributes

Types and function attributes are mutually exclusive. For example, function access can be set to public, protected, or private. Similarly, a data service type can be logical or physical, not both. Other characteristics are simply inapplicable. For example, create-update-delete routines always operate as procedures, not functions.

Data Service Characteristics

The following table describes the characteristics of ALDSP data services. Data service characteristics are defined in the XQuery source pragma.

Data Service Characteristics

Characteristic	Description

Type	There are two types of data services: • **Physical.** The data service is directly based on metadata imported from underlying data sources. Physical data services are created during the metadata import process. • **Logical.** The data service is based wholly or partially on data derived from other data services. Logical data services are created either through the Query Map Editor or in source.
Shape	The shape of a data service is determined by its XML type, or underlying schema, if any. Shapes are: • **Entity.** An entity data service is associated with an XML type. For example, physical data services based on relational tables are entity data services. For any given entity data service, all read functions return information in the shape of the primary XML type. • **Library.** A library data service is not associated with an XML type. Library data services contain routines that can be used by other library or entity data services.

Building XQueries

Operational Characteristics

The following table describes the characteristics that can be used to describe functional routines in ALDSP. These characteristic descriptions are also part of the function's signature, visible in data service Souce editor.

ALDSP Operations Characteristics

Characteristic	Description

Access

Access or visibility to a functional routine can be set as:

- **Public.** A public operation can be called from:
  - any operation in the same data space and
  - from an ALDSP client API. Public operations are the only ones that can be called from client APIs such as Web services or the Java Mediator API.
- **Protected.** An operation with protected visibility can be called from any operation in the same data space. Protected operations cannot be accessed from ALDSP client APIs. An operation in the data space can access the function. Functions in physical data services are, by default, protected.
- **Private.** The function can only be accessed by other functions in its data service. Operations with private visibility are also off-limits to client APIs.

Primacy

Every logical entity data service identifies a single primary function for each kind of function. For example, if there are several read functions, one will be set as primary.

In the case of read functions, the data service relies on the primary read function in the data service to determine the shape of the Return type.

For create, update, and delete functions, the primacy setting is used by update templates of component data services.
In an entity data service, a function can be set as primary. Other functions of a similar type are automatically considered non-primary.

Library functions have no Return type and are not categorized as primary or non-primary.

Kind

ALDSP has several kinds of functions. For physical data services, the kind of function is inferred during the data service creation process, when metadata is imported.

Four of the functions are actually CRUD (create-read-update-delete) procedures, which operate on the underlying data.

- **Read.** Returns data from an underlying data source.
- **Create.** Creates one or several records.
- **Update.** Updates one or several records.
- **Delete.** Deletes one or several records.

Other kinds of functions include:

- **Navigate.** Navigate function have the current data service Return type as one of the input parameters; it typically returns a sequence of the return schema element from the related data Service. Example: Return type Order instead of Return type Customer.
- **Library.** Functions, which are independent of the data service XML type. Library

functions can appear in either data services and library data services

There are two types of operations:

- **Functional.** General-purpose data service functions are designed to retrieve data for clients. Functions cannot have side-effects. Functions can be defined through XQuery or XQSE. If XQSE is used, the fact that the routine is identified as a function means that it does not have side effects.
- **Procedural.** The purpose of a procedural function (also called a procedure or side-effecting procedure) is to affect external processes. A classic example of a side-effecting procedure is an RDBMS stored procedure that modifies underlying data. When a stored procedure is invoked, it operates on the data in the RDBMS without necessarily returning anything to the caller. Similarly, in ALDSP, a procedural function will primarily invoke an external process. Create-update-delete operations are, by definition, procedural.

Operation

Note:There is an important distinction between functions and procedures from the perspective of the data service optimizing engine. Procedures are always considered to have side-effects and are therefore never optimized by the XQuery engine in such a way that they do not independently execute. While a delete() function might not be executed (i.e., "optimized away"), a delete() procedure will always be called.

Functions with and Without Side Effects (click to see full size)

Green boxes represent functions without side effects; red boxes represent functions that may have side effects; boxes with both colors represent functions which can optionally contain side-effects. |

Implementation	Functions can be implemented in the following ways: • **XQuery.** The most common means of implementing an ALDSP function is through XQuery. Of course the data service itself is implemented in XQuery. • **XQSE.** The XQuery Scripting Extension provides a procedural language to extend XQuery to support certain kinds of operations. • **Template-based.** An update template defines the data flow and order for update operations for a logical data service. The update engine in the ALDSP server executes a procedure based on a template; is typically a Java routine used to manage updates of non-relational data. The same template is used by create, update, and delete routines. • **External.** External functions are based on physical sources such as Java, web services, XML, flat files, or relational sources. External functions can be created in entity or library data services.

Developing and Managing Dataspace Projects

This page last changed on Feb 26, 2008.

Developing and Managing Dataspace Projects

Concepts

Data Service File Validation During Deployment

How-to...

Create, Build, Clean, and Delete Dataspace Projects

Deploy, Publish, Configure, and Remove Dataspace Projects

Export Dataspace Projects or Project Folders

Export Dataspace Project Artifacts

Import a Dataspace Project

Handle Error Conditions in a Dataspace Project

Validate, Build, Export, and Package Dataspace Projects from the Command Line

Reference

Dataspace Projects Cheatsheet

Settings for Eclipse Initialization Parameters (.ini) NEW

Setting WorkSpace Studio Initialization Parameters (.ini) for ALDSP 3.2 NEW

Related Topics

How-to...

Create Your First Data Services

Data Service File Validation During Deployment

This page last changed on Feb 26, 2008.

In the Eclipse IDE a dataspace project's data service (.ds) files are validated automatically according to the following deployment model:

- When a user wants to deploy a dataspace project to a WebLogic server using the Deploy Project option, all the Java projects referenced by the dataspace project are built.
- The output files are packaged in JAR files (one per referenced Java project) in the dataspace project's DSP-INF/lib directory during deployment.
- The deploy action validates the dataspace project.
- All the project's artifacts are collected.
- The collected artifacts are deployed to the server.

Create, Build, Clean, and Delete Dataspace Projects

This page last changed on Mar 11, 2008.

A dataspace project is developed in Studio and deployed to a local server. While the development process typically is an iterative cycle of modification and deployment, it is important to keep in mind that the existence of a project in Studio is only loosely coupled with its deployed status. This loose coupling has implications for several types of operations:

- Development and deployment to the server
- Publishing to the server
- Configuring projects on the server
- Removal of the project
- Removal of the project from the server

Topics

Creating a Dataspace Project

You can create a new dataspace projects using the Studio's File menu.

File > New > Dataspace Project

Steps in Creating a Dataspace Project

Building a Dataspace Project

In Studio it is often a good practice to set your project to be built automatically every time you modify a

file in your project. You can establish this setting through the Studio Project menu:

Project > Build Automatically

A checkbox appears when this option is selected.

Cleaning a Dataspace Project

Applying a "clean" to a project clears out any existing build problems and build states. If your build runs into error conditions or other problems, cleaning and redeploying your project is a recommended first step.

Project > Clean...

A dataspace project can only be deployed when no other process has an editing lock on the ALDSP configuration that contains your dataspace. The ALDSP configuration can be locked through the ALDSP Administration Console (Lock and Edit), by a client process (MBean API or WLST script), or during deployment from Eclipse/Workshop.

Deleting a Dataspace Project

Dataspace projects are both created and deleted through the Project Explorer.

To delete a project:

1. Right-click on the project's name in the Project Explorer.
2. Select Delete.

You will be given two options:

- **Delete content from the file system?** If you choose:

Do not delete content

you will be able to import the project at a later time.

- **Delete the dataspace on the server?** The deployed dataspace will be removed from the server. If this option is not selected, the dataspace will remain in one of two states, depending on selected options:
    - Available to be configured on the server
    - Configured on the server

Data services can also be removed from their server through the ALDSP Administration Console.

Deploy, Publish, Configure, and Remove Dataspace Projects

This page last changed on Mar 11, 2008.

Dataspace projects are created in the Studio Eclipse plugin framework. A project that builds successfully is ready to be made available from a local supported version of WebLogic server.

Several terms can be used to describe the process of managing a server's dataspace projects.

- **Deploy.** Dataspace projects can be deployed to their server on an individual basis.
- **Publish.** All projects associated with a server can be deployed at once.
- **Available.** These are projects that have been deployed or published and are available to be configured on the server. A project must both present on the server and *configured* before it can be access by client applications. A project with an Available status can be thought of as *staged*.
- **Configured.** A configured project is available to authorized calling applications. A project with Configured status on the server can be thought of as *released*.

Use the Add and Remove Projects dialog to move projects between Available and Configured stati.

Topics

Deploying an ALDSP Project

You can deploy a dataspace project from Project Explorer by:

1. Right-clicking on the project name.
2. Selecting Deploy Project.

A confirmation message appears after your project deploys successfully. If deployment cannot be completed, you can use the Problems and Errors windows to determine the problem and corrective action.

Results of the effort are placed in a log file:

Window > Show View > Error Log

In addition, Studio will report whether the deployment was successful or not.

A successful deployment automatically configures the dataspace project on the server. You can use the Add and Remove Projects... dialog to change the deployed status from Configured to Available to be configured.

A dataspace project can only be deployed when no other process has an editing lock on the ALDSP configuration that contains your dataspace. The ALDSP configuration can be locked through the ALDSP Administration Console (Lock and Edit), by a client process (MBean API or WLST script), or during deployment from Eclipse/Workshop.

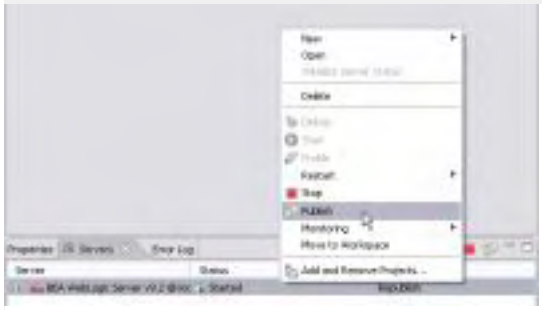
Publishing Server Projects

Sometime it is convenient to publish all the dataspace projects associated with a workspace.

The Publish option applies to all projects in the workspace.

Right-clicking on the name of your server in the Servers window and selecting Publish.

Publishing Server Projects



The state in the Servers window will be changed to Republish.

Publishing or republishing a set of projects does not affect the configuration status of each project on the server. You can modify the configuration status through the Add and Remove Projects... dialog.

Configuring Server Projects

Projects on a server are considered either *configured on the server* or *available to be configured on the server*. Configuration status is managed either through the Add and Remove Projects... dialog or directly from the Servers window.

 Only configured projects are available to client applications.

Managing Configured Projects Through Dialog

To access the dialog:

1. Right-click on the name of the server in the Servers window.
2. Select Add and Remove Projects...

Add and Remove Projects Dialog

Managing Configured Projects Through the Servers Window

You can also change a project's configuration status through the Servers window.

Server and Projects

1. Click on the + symbol next to the server name.
2. Right-click on the project you wish to unconfigure.
3. Select Remove.

   Alternatively, just select your project and click the Delete key.

   You can use the Add and Remove Projects Dialog to change the configuration status of your project..

Removing Dataspace Projects from a Server

You can permanently remove a project from the server through the right-click menu Delete option in the Project Explorer.

Deleting a Dataspace Project

Export Dataspace Projects or Project Folders

This page last changed on Feb 26, 2008.

ALDSP dataspace projects or their component folders can be exported in EAR (archive) format using standard Eclipse mechanisms. The export target is the local file system.

If an entire project is exported as an EAR, it constitutes a back-up of the project which can then be re-imported into an Eclipse-compatible IDE.

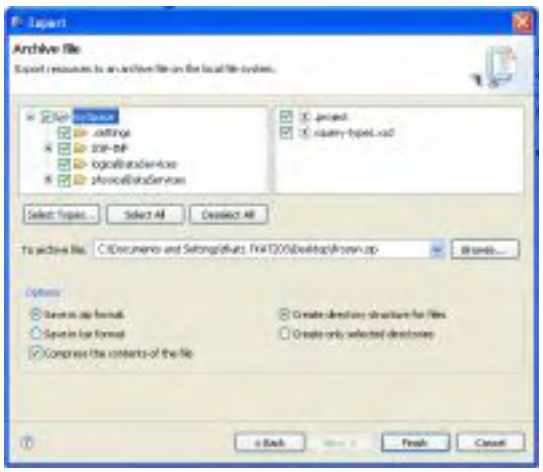
To create an archive file of a dataspace project:

1. In Project Explorer right-click on your project (or folder).
2. Navigate to:

   Export > General > Archive File

3. In the Archive File wizard select the entire project or one or several folders.
4. Select from available export options.
5. Click Finish.

Creating an EAR File for a Dataspace Project



See Eclipse Documentation for general information about the Export operation.

Export Dataspace Project Artifacts

This page last changed on Feb 26, 2008.

This section describes the various types of export operations available for ALDSP dataspace projects.

Topics

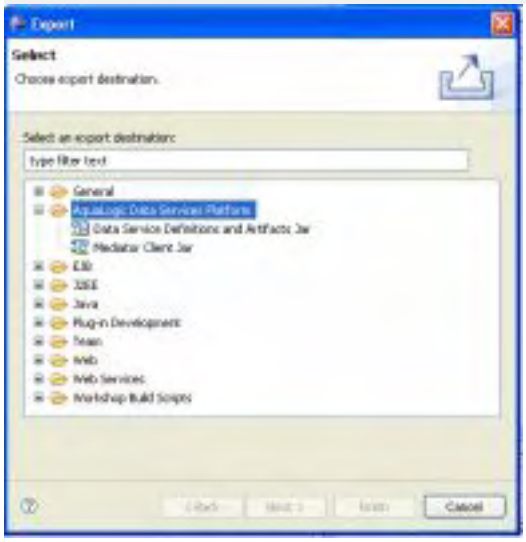
Exporting Dataspace Artifacts

The ALDSP export wizards can be accessed using the File > Export menu, then expand on the ALDSP export wizard category.

File > Export... > AquaLogic Data Services Platform

Artifacts can only be exported from deployable dataspace projects; if your project is not deployable, the export operation will not succeed.

Exporting a Dataspace Project



In the ALDSP category there are three wizards. Each generates a specific type of JAR file.

Types of ALDSP JAR File Export Operations

Export Type	Effect
Data service definitions and artifacts	All ALDSP deployable artifacts are bundled into a JAR file.
Mediator client	A Java interface for accessing data services is created.
Web services mediator client	A web services interface for accessing data services is created.

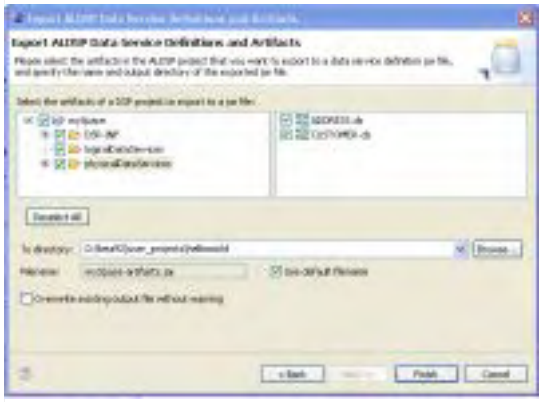
Any dataspace projects pre-selected in the Project Explorer will automatically be selected in the export wizard.

Generating a Data Service Definitions and Artifacts JAR

An exported JAR file containing a single project's server-deployable definitions and artifacts. Such a file can be imported into another ALDSP-enabled version of Eclipse. In addition, the definitions and artifacts JAR can be useful:

- As a means of transporting a dataspace from one application to another.
- In conjunction with certain refactoring operations.
- For deployment on multiple servers (clusters) at a later time.
- For debugging purposes.

Export Data Service Definitions and Artifacts



In the wizard, the contents you identify using their adjacent checkbox will be exported. For example if you check the box next to the project name, all of that projects server-deployable components will be selected.

You can fine-tune your selection by clicking on a folder. The folder's contents will appear in the right-hand column where you can use a checkbox to control which artifact will be exported.

Actions Associated with Generating a Data Service Definitions and Artifacts JAR

Item	Recommended Setting or Action	Details - Comments
Export Data Service Definitions and Artifacts Page		
Check folders or their contents that you want to export		
Deselect All		Convenience if more than one project is selected.
To Directory:	`/user_projects/helloworld`	JAR file can be exported anywhere on the system.
Filename:	`-artifacts.jar`	Example: `mySpace-dsp-client.jar`

Use default filename option	Selected	When selected editing of the generated filename is not allowed.
Overwrite existing file without warning option	Unselected	If unselected you will be asked if you want to overwrite any existing file of the same name.
Success message	**Finish**	Identifies project and target name.

Generating a Mediator Client JAR File

This wizard presents a list of open ALDSP projects to select from, and it generates an ALDSP mediator client JAR file from the currently selected project. Projects are exported one at a time.

Java programs access data services through the ALDSP Mediator API. This API is generated from the ALDSP Eclipse platform and is based on a project that can be successfully built and deployed.

Accessing Data Services from Java Clients

Export Mediator Client JAR Wizard

Steps Associated with Generating a Mediator Client API JAR File

Item	Recommended Setting or Action	Details - Comments
Select the Mediator Client JAR File export wizard		

Type:	Mediator Client JAR File	
Select a Dataspace Project Page	**Next**	
Pick a dataspace project	mySpace	Only one project at a time can be exported.
Deselect All		Convenience if more than one project is selected.
To Directory:	`/user_projects/helloworld`	JAR file can be exported anywhere on the system.
Filename:	`-dsp-client.jar`	Example: `mySpace-dsp-client.jar`
Use default filename option	Selected	When selected editing of the generated filename is not allowed.
Overwrite existing file without warning option	Unselected	If unselected you will be asked if you want to overwrite any existing file of the same name.
Success message	**Finish**	Identifies project and target name.

Generating a JAR File Containing Data Service-to-Web Service Maps

After you have created a web service map of one or more data services you can create an exported JAR file containing these maps.

Then the created JAR file can be used as your applications web service interface to available data services.

Only publicly available operations can be turned into web service operations. You can adjust access level to a data service function through the Properties window.

Exporting a Web Services Mediator Client JAR File

Actions Associated with Generating a Web Services Mediator Client JAR File

Item	Recommended Setting or Action	Details - Comments
Export Web Services Mediator Client JAR File		
Check folders or their contents that contain web service maps	Select folder	Click on the exportable dataspace folder to located selected WS file.
Deselect All		Convenience if more than one project is selected.
To Directory:	`/user_projects/helloworld`	JAR file can be exported anywhere on the system.

		Example:
Filename:	`-ws-client.jar`	`mySpace-ws-client.jar`
Use default filename option	Selected	When selected editing of the generated filename is not allowed.
Overwrite existing file without warning option	Unselected	If unselected you will be asked if you want to overwrite any existing file of the same name.
Success message	**Finish**	Identifies project and target name.

Import a Dataspace Project

This page last changed on Feb 26, 2008.

ALDSP uses standard Eclipse import mechanisms. You can import a JAR file project or an open (file-based) project.

Importing a JAR File Project

A JAR file containing an ALDSP project can be imported into your workspace.

1. Create a dataspace project, naming it appropriately. (Alternatively you may be able to use an existing project if there are no naming conflicts.)

   File > New > Dataspace Project

2. Click on your new project.
3. Choose:

   File > Import > General > Archive File

4. Next.
5. Browse to the directory location of your JAR file.
6. Open.
7. Answer Yes to All to the question regarding overwriting xquery-types.xsd.
8. Finish.
9. Deploy your project to verify a successful build and deployment.

Importing a File-based Project

A project in an accessible file system can be imported into the Eclipse workshop. You can import one or several projects at the same time.

1. Choose:

   File > Import... > General > Existing Projects into Workspace

2. Browse to your dataspace project directory.
3. Select the project or projects you wish to import.

Handle Error Conditions in a Dataspace Project

This page last changed on Feb 26, 2008.

During the course of creating your project there are times when the project will be in an error condition. There are many reasons for this. Generally speaking the way to handle such conditions is to either:

- Go forward because you understand why the condition has occurred.
- Revert using Undo.

There may be some cases, however, when the error condition comes as a surprise and/or there is no easy way to revert. Information about such conditions can be found in two places.

Problem Reporting in Dataspace Projects

Tabular window	Purpose
Problems (**Window > Show View > Problems**)	Collects and displays errors in the data service source file.
Error Log (**Window > Show View > Error Log**)	Collects and displays project-related error conditions.

Problem and Error Log Tabs in a Dataspace

The error log contains several types of messages; icons are used to differentiate their type.

Error Log Icons and Their Meaning

Icon	Meaning
	Error.
	Error with log or stack trace.
	Warning.

 Informational.

 Process icon.

Double-clicking on each line in the Error Log window will open a separate dialog that will allow you to see more information. Examples:

- Double-clicking or an error might open a dialog that will contain the related stack trace.
- Double-clicking on each line in the Problems view will, if possible, open the file having errors and highlight the error.

Validate, Build, Export, and Package Dataspace Projects from the Command Line

This page last changed on Feb 26, 2008.

This section describes how to validate, build, export and package ALDSP dataspace projects from the command line.

Topics

In the Eclipse IDE a dataspace project's data service (.ds) files are validated automatically according to the following deployment model:

- When a user wants to deploy a dataspace project to a WebLogic server using the Deploy Project option, all the Java projects referenced by the dataspace project are built.
- The output files are packaged in JAR files (one per referenced Java project) in the dataspace project's DSP-INF/lib directory during deployment.
- The deploy action validates the dataspace project.
- All the project's artifacts are collected.
- The collected artifacts are deployed to the server.

The ALDSP Export mechanism allows for a dataspace project's artifacts to be packaged in a JAR the contents of which are identical to what would be generated from the IDE for deployment to a WebLogic server.

Dataspace Packaging from the Command-line

There is also an occasional need for operations such as validate, build, export, and package to be available in a scripting environment. This section describes an Ant script file, cmdline_build.xml, provided in the "bin" directory under the ALDSP installation that can be invoked by a user to:

- Validate a dataspace project
- Generate a deployment JAR file of a dataspace project

For those not wishing or able to use Ant, Java equivalent command-line options are also described.

Syntax Summary

Command	Syntax
help	help [{*cmd*} | dsp30:all]
validate-project	validate-project {*project*}
export-mediator-client	export-mediator-client {*project*} {*jardir*} [*jarname*[dsp30:.jar]]
export-ws-client	export-ws-client {*project*} {*jardir*} [*jarname*[dsp30:.jar]] [dsp30:*ws_locator*,...]
export-artifacts	export-artifacts {*project*} {*jardir*} [*jarname*[dsp30:.jar]]

Command-Line Ant Build Targets

This section describes available ALDSP ant build targets.

Build XML File

The build XML file:

cmdline_build.xml

will be provided in the directory:

<*dsp_home*>/bin

To see a list of build targets with short descriptions in the Ant build XML file, invoke the command below at the prompt window:

```
ant -f <bea_home>/aldsp_3.0/bin/cmdline_build.xml -projecthelp
```

Notes

- It is assumed that Ant is available on your computer and is on your path. Some targets require:
  - ECLIPSE_HOME environment variable points to the Eclipse installation directory.
  - javac be available on the PATH variable.
- Commands other than "help" involving an Eclipse project requires specification of the Eclipse workspace directory that contains the project.
  - For Java commands the directory is specified via the "-data" option.
  - For Ant command, it is specified as the "-Dworkspace" property.

"help"

The "help" target is the default build target. It shows a list of available ALDSP commands and the syntax needed to invoke the command in Java.

Command	Syntax
help	help [{*cmd*} | dsp30:"all"]

Build Invocation Syntax via Java

Note that the syntax shows the portion starting with "aldsp_command" below. However, the full syntax to be entered for Java at the prompt window is:

```
java -cp <eclipse_home_dir>/startup.jar org.eclipse.core.launcher.Main -data
```

<*workspace_dir*> -application com.bea.dsp.ide.app.runCmdline <*aldsp_command*> <*cmd_param*> ...

Build Invocation Syntax via Ant

If invoked via Ant, ALDSP command parameters should be specified as Ant properties. For example, to get help about the "export-artifacts" command, enter:

```
ant -f <aldsp_install_dir>/bin/cmdline_build.xml help -Dcmd=export-artifacts
```

To get help on all ALDSP commands, specify the following property:

```
  -Dcmd=all
```

or omit the optional -Dcmd property completely:

```
ant -f <aldsp_install_dir>/bin/cmdline_build.xml help all
```

"validate-project"

The "validate-project" target validates the data service (.ds) files in the specified dataspace project. Data service error messages that would show up in the Eclipse IDE's Problems view are sent to stdout when this target is invoked. A "fail" status is returned by this target if any error exists in a .ds data service file in the project.

ant -f <*aldsp_install_dir*>/bin/cmdline_build.xml -Dworkspace=/bea/projects/*myworkspace* -Dproject=*MyFirstDspProject* validate-project

Command	Syntax
validate-project	validate-project {*project*}

"export-mediator-client"

The export-mediator-client target is for generating an ALDSP mediator client JAR file of a dataspace project.

The default value of the output JAR file name is:

<*project*>-dsp-client.jar

Command	Syntax
export-mediator-client	export-mediator-client {*project*} {*jardir*} [dsp30:*jarname*.jar]

"export-ws-client"

The export-ws-client target generates a web services mediator client JAR file from the specified comma-separated list of wsmap file locators in a dataspace project.

The default value of the output JAR filename is:

```
<project>-ws-client.jar
```

The default value of the ws_locators is all wsmap file locators in the project.

An example of a wsmap file locator is:

```
ld:logical/wsmaps/CUSTOMER.ws
```

Command	Syntax
export-ws-client	export-ws-client {*project*} {*jardir*} [*jarname*[dsp30:.jar]] [dsp30:*ws_locator*,...]

"export-artifacts"

The "export-artifacts" target creates a JAR file containing the definitions and artifacts of the dataspace project. The content would be identical to the artifact JAR file created in the IDE. By default , the name of the artifact JAR file is:

<*project*>-artifacts.jar

Command	Syntax
export-artifacts	export-artifacts {*project*} {*jardir*} [*jarname*[dsp30:.jar]]

Notes

Referenced Java Projects

Since a dataspace project may reference other Java projects in the same Eclipse workspace, you should make certain that:

- Referenced projects in your build script are also built.
- The resulting JAR files and dependent JAR files are copied to the dataspace project's DSP-INF/lib directory.

This needs to be done prior to exporting a deployable JAR file using the export-artifacts command in order for all referenced/required JAR files to be included in the artifact JAR file.

Invoking Build Commands Without Ant

The Ant targets described in the previous sections are actually implemented in Java. So the actual implementation can be invoked at the prompt window using Java directly -- or any script process -- instead of Ant.

Command-line Examples using Ant and Java

This section contains several examples of invoking the ALDSP command using Ant and Java.

Getting the help text of all the commands using Ant and Java at the prompt window

Ant:

```
ant -f <bea_home>\aldsp_3.0\bin\cmdline_build.xml help -Dcmd=all
```

Java:

java -cp <*eclipse_home*>/eclipse/startup.jar org.eclipse.core.launcher.Main -application com.bea.dsp.ide.app.runCmdline help all

Getting the help text of a specific command using Ant and Java at the prompt window

Ant:

ant -f <*bea_home*>\aldsp_3.0\bin\cmdline_build.xml help -Dcmd=export-artifacts

Java:

java -cp <*eclipse_home*>/startup.jar org.eclipse.core.launcher.Main -application com.bea.dsp.ide.app.runCmdline help export-artifacts

Exporting the artifacts of a dataspace project

This example exports the project:

```
DspProj
```

in workspace:

```
/MyWorkspace
```

to:

```
/temp
```

directory using the default JAR file name:

```
<project>-artifacts.jar
```

Ant:

ant -f <*bea_home*>\aldsp_3.0\bin\cmdline_build.xml -Dworkspace=/MyWorkspace -

Dproject=DspProj -Djardir=/temp export-artifacts

Java:

java -cp <*eclipse_home*>/startup.jar org.eclipse.core.launcher.Main -data /MyWorkspace -application com.bea.dsp.ide.app.runCmdline export-artifacts DspProj /temp

---

Dataspace Projects Cheatsheet

This page last changed on Mar 11, 2008.

Task	Location	Action	Comments
Build a project with every save	Project menu	Build Automatically	
Clean selected project	Project menu	Clean	
Close a project	Project Explorer	Right-click > Close Project	
Close a project	Project menu	Project > Close Project	
Close unrelated projects	Project Explorer	Right-click > Close Unrelated Projects	
Copy-Paste a project	Project Explorer	Right-click on project name > Copy-Paste	
Create a new project	File menu	File > New > Dataspace Project	
Delete selected project from a workspace	Project Explorer	Right-click > Delete	
Deploy a project to the server	Project Explorer	Right-click > Deploy Project	
Export selected project	File menu	Export	
Import selected project into a workspace	File menu	Import	
Open selected project	Project Explorer	Open	
Project properties	Project menu	Right-click > Properties	
Project properties	File menu	Properties	
Publish all the projects in a workspace	Server window	Right-click > Publish	
Refactor Rename-Remove	Project Explorer	Right-click > Refactor > Rename-Remove	Refactor provides for "safe" renaming or deleting of projects or project components.
Refresh Studio display	Project Explorer	Right-click > Refresh File > Refresh	

Remove a previously deployed (configured) project from the server	Server window	Right-click > Delete > Yes > Yes to 'Delete the dataspace on the server?'	
Rename a project	File menu	Rename	See Safely rename...
Rename a project safely	Project Explorer	Right-click > Refactor > Rename...	
Safely delete selected project from a workspace	Project Explorer	Right-click > Refactor > Delete...	See Safely delete...
Search for a project in a workspace	Search menu	• Search ... • Ctrl-h	
See dataspaces currently deployed on the server	Server window	Click the + next to the name of the running server.	
Show project explorer	Window menu	Show View > Project Explorer	
Show project in Service Assembly Modeler	Project Explorer	Right-click > Show in Service Assembly Modeler	
Submit a project to the AquaLogic Enterprise Repository	Project Explorer	Right-click > Submit ... to repository	
Undeploy a project	Server window	Right-click > Delete > Yes > Yes to 'Delete the dataspace on the server?'	
Update a project's metadata	Project Explorer	Right-click > Update metadata	
Validate project artifacts	Project Explorer	Right-click > Validate ...	

Settings for Eclipse Initialization Parameters (.ini)

This page last changed on Mar 19, 2008.

Data Services Studio is an Eclipse plugin. As such, it uses an initialization file that is similar to the `eclipse.ini` file available in basic installations of Eclipse. This file in turn provides directives to the Java Virtual Machine (JVM) in which the application runs.

The default parameters provides with ALDSP are located in the following file:

```
{BEA_HOME}/aldsp_3.0/bin/aldsp.ini
```

When running `aldsp.exe` the settings in this file supersede any configuration settings in the `eclipse.ini` file.

By default the `aldsp.ini` memory settings are:

```
-vmargs
-Xms384m
-Xmx768m
-XX:MaxPermSize=256m
```

If you encounter out-of-memory errors associated with running an ALDSP project, try increasing the maximum memory setting. For example:

```
-Xmx1024m
```

Setting WorkSpace Studio Initialization Parameters (.ini) for ALDSP 3.2

This page last changed on Mar 19, 2008.

Workspace Studio is an Eclipse plugin. As such, it uses an initialization file that is similar to the `eclipse.ini` file available in basic installations of Eclipse. This file in turn provides directives to the Java Virtual Machine (JVM) in which the application runs.

The default parameters provides with WorkSpace Studio are located in the following file:

```
{WORKSPACESTUDIO_HOME}/workSpaceStudio_1.1/workSpaceStudio/workSpaceStudio.ini
```

When running `aldsp.exe` the settings in this file supersede any configuration settings in the `eclipse.ini` file.

By default the `aldsp.ini` memory settings are:

```
-vmargs
-Xms384m
-Xmx768m
-XX:MaxPermSize=256m
```

If you encounter out-of-memory errors associated with running an ALDSP project, try increasing the maximum memory setting. For example:

```
-Xmx1024m
```

Creating and Updating Physical Data Services

This page last changed on Feb 26, 2008 .

Creating and Updating Physical Data Services

Concepts

- Creating Data Source Metadata
- Creating Physical Data Services from Java Functions »NEW

How To Create Physical Data Services...

- from relational tables and views
- from stored procedures
- based on SQL statements
- based on database functions
- from web services
- from Java functions »NEW
- from XML files
- from delimited files

How-to...

- Enable Optimistic Locking of Relational Objects
- Update Physical Data Service Metadata
- Create SOAP Handlers for Imported WSDLs
- Create XMLBean Support for Java Functions »NEW
- Create a Java Class to Support Metadata Import »NEW
- Create XMLBean Classes for Java Functions »NEW
- Examples: Using Imported Java Functions »NEW

Reference

- Stored Procedure Configuration
- Simple Java Types and XQuery Equivalents »NEW

Related Topics

How-to...

- Add an External Function to an Existing Physical Data Service

Creating Data Source Metadata

This page last changed on Feb 26, 2008 .

Creating Physical Data Services by Importing Source Metadata

In ALDSP metadata around a particular data source is developed during the process of creating a physical data service. For example, a list of the tables and columns in a relational database is metadata. A list of operations in a Web service is metadata.

In ALDSP, a *physical* data service is typically primarily based on metadata describing the structure of those physical data sources.

Physical data services are the building blocks for the creation of logical data services.

Data Source Support for Creating Physical Data Services

Source Type	Venue
Relational (including tables, views, stored procedures, and SQL)	JDBC
Web services (WSDL files)	URI, UDDI, WSDL
Delimited (CSV files)	File-based data, such as spreadsheets.
Java functions (.java)	Programmatic
XML (XML files)	File- or data stream-based XML

When information about physical data is developed during the creation of physical data services, two things happen:

- A physical data service (extension .ds) is created in your ALDSP-based project.
- A companion schema of the same name (extension.xsd), is created. This schema describes quite exactly the XML type of the data service. Such schemas are placed in a directory named schemas which is a sub-directory of your newly created data service.

Source View

The introspection process is done through the Physical Data Service Creation wizard. This wizard introspects available data sources and identifies data objects that can be rendered as operations for either entity or library data services. Once created, physical data services become the building-blocks for queries and logical data services through a series of pragmas created in the query source.

For example, the following source resulted from importing a Web service operation:

```
(::pragma function <f:function xmlns:f="urn:annotations.ld.bea.com" kind="read"
nativeName="getCustomerOrderByOrderID"
nativeLevel1Container="ElecDBTest" nativeLevel2Container="ElecDBTestSoap" style="document"/>:

declare function f1:getCustomerOrderByOrderID($x1 as element(t1:getCustomerOrderByOrderID))
   as schema-element(t1:getCustomerOrderByOrderIDResponse) external;
```

Notice that the imported Web service is described as a "read" function in the pragma. "External" refers to the fact that the schema is in a separate file.

For some data sources such as web services imported metadata represents functions which typically return void (in other words, these functions do something other than return data). Such routines are sometimes called *side-effecting functions* or *procedures*.

The following source resulted from importing Web service metadata that includes an operation that has been identified as a *side-effecting procedure*:

```
(::pragma function <f:function xmlns:f="urn:annotations.ld.bea.com"
kind="hasSideEffects" nativeName="setCustomerOrder" style="document"/>:

declare function f1:setCustomerOrder($x1 as element(t3:setCustomerOrder)) as schema-element(t3:
setCustomerOrderResponse) external;
```

In the above pragma the function is identified as "hasSideEffects".

Physical Data Services from Java Functions Overview

This page last changed on Mar 25, 2008 .

Physical Data Services from Java Functions Overview

In AquaLogic Data Services Platform (ALDSP), you can create physical data services based on user-defined functions implemented as Java classes. ALDSP supports Java functions returning the following types:

- Java primitive types and single-dimension arrays
- Global elements and global element arrays through XMLBean classes
- Global elements and global element arrays through SDO DataObjects

ALDSP packages operations marked as create, update, or delete functions in an Entity data service. Otherwise, the resulting data service is of type Library. Functions determined to return void are automatically marked as library procedures. When creating a new physical data service, you can change the nominated function type.

The Java method name, when used in an XQuery, becomes the XQuery function name qualified with a namespace.

The following restrictions apply to Java functions:

- Java functions intended for import into a data service must be declared as static
- Function overloading is based on the number of arguments, not the parameter types
- Array support is restricted to single-dimension arrays only
- In functions returning complex types, the return element needs to be extracted from a valid XML document

Simple Java Types and Their XQuery Counterparts

The following outlines the mapping between simple Java types and the corresponding XQuery or schema types:

Java Simple or Defined Type	XQuery/Schema Type
boolean	xs:boolean
byte	xs:byte
char	xs:char
double	xs:double
float	xs:float
int	xs:int
long	xs:long
short	xs:short
string	xd:string

java.lang.Date	xs:datetime
java.lang.Boolean	xs:boolean
java.math.BigInteger	xs:integer
java.math.BigDecimal	xs:decimal
java.lang.Byte	xs.byte
java.lang.Char	xs:char
java.lang.Double	xs:double
java.lang.Float	xs:float
java.lang.Integer	xs:integer
java.lang.Long	xs:long
java.lang.Short	xs:short
java.sql.Date	xs:date
java.sql.Time	xs:time
java.sql.Timestamp	xs:datetime
java.util.Calendar	xs:datetime

Java functions can consume parameters and return values of the following types:

- Java primitives and types listed in the previous table
- Apache XMLBeans
- BEA XMLBeans
- SDO DataObject (typed or untyped)

The elements or types referred to in the schema should be global elements.

Physical Data Service from a Java Function - Example Code

This topic provides examples showing the use of imported Java functions in an XQuery and the processing of complex types.

Using a Function Returning an Array of Java Primitives

As an example, the Java function `getRunningTotal` can be defined as follows:

```
public static float[] getRunningTotal(float[] list) {
   if (null == list || 1 >= list.length)
      return list;
   for (int i = 1; i < list.length; i++) {
      list[i] = list[i-1] + list[i];
   }
   return list;
}
```

The corresponding XQuery for executing the above function is as follows:

```
Declare namespace f1="ld:javaFunc/float"
Let $y := (2.0, 4.0, 6.0, 8.0, 10.0)
Let $x := f1:getRunningTotal($y)
Return $x
```

The results of the query is as follows:

```
2.0, 6.0, 12.0, 20.0, 30.0
```

Processing complex types represented via XMLBeans

Consider a schema called Customer (customer.xsd), as shown in the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
   <xs:schema targetNamespace="ld:xml/cust:/BEA_BB10000" xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="CUSTOMER">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="FIRST_NAME" type="xs:string" minOccurs="1"/>
            <xs:element name="LAST_NAME" type="xs:string" minOccurs="1"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

You could compile the schema using XMLBeans to generate a Java class corresponding to the types in the schema.

```
xml.cust.beaBB10000.CUSTOMERDocument.CUSTOMER
```

For more information, see http://xmlbeans.apache.org.

Following this, you can use the CUSTOMER element as shown in the following:

```
public static xml.cust.beaBB10000.CUSTOMERDocument.CUSTOMER[]
      getCustomerListGivenCustomerList(xml.cust.beaBB10000.CUSTOMERDocument.CUSTOMER[] ipListOfCust)
      throws XmlException {
   xml.cust.beaBB10000.CUSTOMERDocument.CUSTOMER [] mylocalver = pListOfCust;
   return mylocalver;
}
```

The resulting metadata information produced by the New Physical Data Service wizard will be:

```
(::pragma function <f:function xmlns:f="urn:annotations.ld.bea.com" kind="datasource" access="public">
<params>
<param nativeType="[Lxml.cust.beaBB10000.CUSTOMERDocument$CUSTOMER;"/>
</params>
</f:function>::)

declare function f1:getCustomerListGivenCustomerList($x1 as element(t1:CUSTOMER)*) as element(t1:CUSTOMER)*
external;
```

The corresponding XQuery for executing the above function is:

```
declare namespace f1 = "ld:javaFunc/CUSTOMER";
let $z := (
validate(<n:CUSTOMER xmlns:n="ld:xml/cust:/BEA_BB10000"><FIRST_NAME>John2</FIRST_NAME><LAST_NAME>Smith2</
LAST_NAME>
</n:CUSTOMER>),

validate(<n:CUSTOMER xmlns:n="ld:xml/cust:/BEA_BB10000"><FIRST_NAME>John2</FIRST_NAME><LAST_NAME>Smith2</
LAST_NAME>
</n:CUSTOMER>),

validate(<n:CUSTOMER xmlns:n="ld:xml/cust:/BEA_BB10000"><FIRST_NAME>John2</FIRST_NAME><LAST_NAME>Smith2</
LAST_NAME>
</n:CUSTOMER>),

validate(<n:CUSTOMER xmlns:n="ld:xml/cust:/BEA_BB10000"><FIRST_NAME>John2</FIRST_NAME><LAST_NAME>Smith2</
LAST_NAME>
</n:CUSTOMER>))

for $zz in $z
return
```

See Also

How Tos

- Create a Physical Data Service from a Java Function
- Preparing to Create Physical Data Services From Java Functions

Create Physical Data Services from Relational Tables and Views

This page last changed on Mar 13, 2008 .

How To Create Physical Data Services from Relational Tables and Views

The following topics describe how to create physical data services from relational tables and views:

Setting Up the Physical Data Service Creation Wizard

Physical data services are created using a wizard.

Physical Data Service Creation Wizard

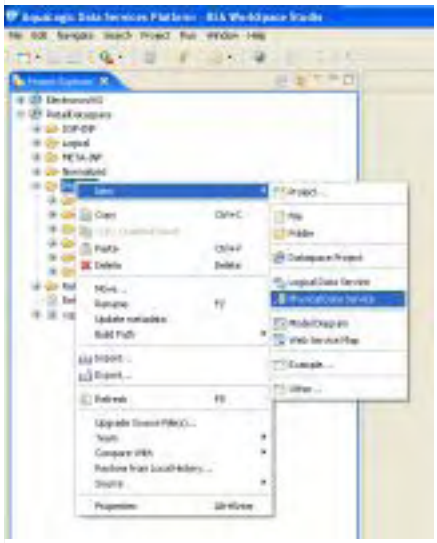
Starting the Wizard

To start the physical data service creation wizard:

1. Right-click on your dataspace project or any folder in your project.

2. Choose New > Physical Data Service

Creating a New Physical Data Service



Setting Up the Import Wizard for Relational Objects

When importing a relational object available options include the ability to:

1. Set a location for your new data service to be saved within your project.
2. Select a data source from the dropdown listbox.
3. Select the database type for the selected source (PointBase for the sample RDBMS) from the dropdown listbox.
4. Select among the relational source types listed in the following table.

Types of available relational data sources

Relational Type	Description
Tables and Views	Displays all public tables and views in the selected data source.
Stored Procedures	Displays all public stored procedures in the selected data source.
SQL Statement	Allows creation of a SQL statement for extracting relational data from the data source.
Database Function	Allows creation of an XQuery function in a library data service based on build-in or custom database functions.

Selecting SQL Table and View Objects for Import

To create a physical data service based on a relational table or view:

1. Select the Tables and Views option
2. Click Next.

A list of available database table and view SQL objects appears.

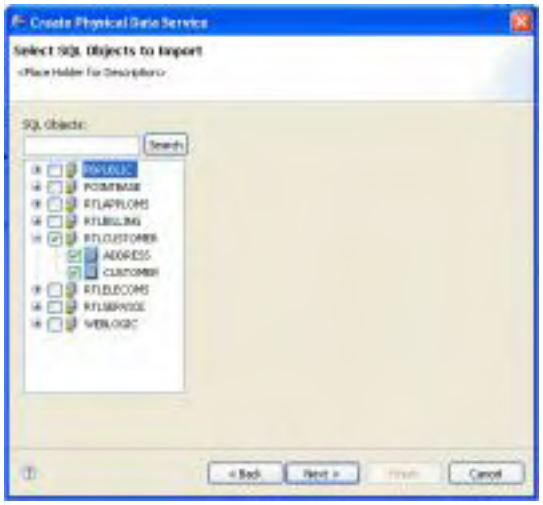
Objects are grouped based on the relational data sources catalog and/or schema.

In the example of an RTLCUSTOMER catalog, the ADDRESS and CUSTOMER tables both become physical data services.

Database-specific Catalog and Schema Considerations

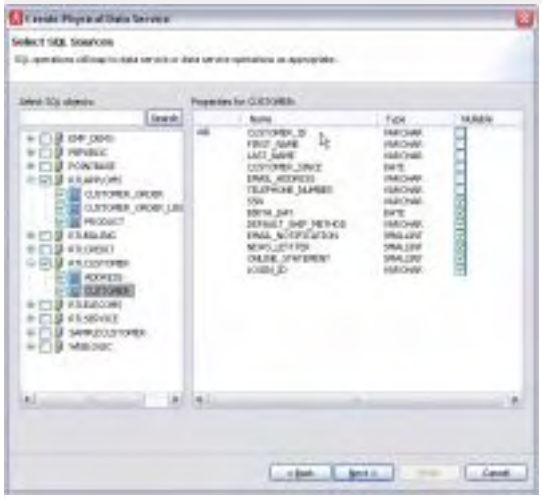
Simply check the desired objects or their container, which will select all enclosed tables or views.

Table and View Objects Selected for Import



If you click on an individual object such as ADDRESS or CUSTOMER, information describing the database's primary key(s), column name, type and nullability appears. For example the CUSTOMER table contains a CUSTOMER_ID field of type VARCHAR. That column is not nullable, meaning that it must be supplied with any updates.

Physical Data Service Properties



Filtering SQL Objects Using Search

The Search option available when creating a physical data service can be especially useful when:

- You know specific names of the data source objects you want to turn into data services.
- Your data source may be so large that a filter is needed.
- You may be looking for objects with specific naming characteristics such as:

```
%audit2003%
```

The above search command retrieves all objects that contain the enclosed string.

Using JDBC Syntax to Search SQL Objects

You can search through available SQL objects using standard JDBC wildcard syntax.

- An underscore (_) creates a wildcard for an individual character.
- A percentage sign (%) indicates a wildcard for a string. Entries are case-sensitive.

Another example:

```
CUST%, PAY%
```

entered in the Tables/Views field the above search string returns all tables and views starting with either CUST or PAY.

Special Considerations When Searching Stored Procedures

If no items are entered for a particular field, all matching items are retrieved. For example, if no filtering entry is made for the Procedure field, all stored procedures in the data object will be retrieved.

Setting Properties for New Data Service Operations

Each new entity data service is created with a Read function that contains all the metadata elements identified during data service creation. It can be thought of as comparable to the following construct in the relational world:

```
select * from <table>
```

Use the Properties dialog to:

- Optionally modify the operation name.
- Set the Public option (check if you want your function to be available to client applications).
- Set the kind of operation (in some cases only Read will be available).
- Set the Primary option (check if you want your function to be the primary of its type).

  In some cases this option may not be available.

- Select a common XML namespace for the entire data service or individual target namespaces for specific operations.
- Set the target namespace.

The root element, which is read-only, is also displayed.

Initially the root element name matches the name of the data service.

Setting Properties for New Data Service Functions

Default Naming Conventions

There are several default naming conventions associated with new data services:

- When a table, view, or other data source object is the source for a data service, the nominated name is wherever possible the same as the source object name. In some cases, however, names are adjusted to conform with XML naming conventions.
- Initially the root element name matches the name of the data service.

 XML Name Conversion Considerations

Verifying Data Service Composition

On the Review New Data Service(s) page you can set, confirm or, optionally, change suggested data service names depending on the type of physical data service you are creating.

Default Physical Data Service Names

The nominated name for a new data service is, wherever possible, the same as the source object name. In some cases, however, names are adjusted to conform with XML naming conventions.

 XML Name Conversion Considerations

About Automatic Data Service Name Changes

Name conflicts occur when there is a data service of the same name present in the target directory. Name conflicts are highlighted in red.

There are several situations where you will need to change the name of your data service:

- There already is a data service of the same name in your application.
- You are trying to create multiple data services with the same name.

Data services always have the file extension:

```
.ds
```

Database-specific Catalog and Schema Considerations

Database vendors variously support database catalogs and schemas.

Vendor Support for Catalog and Schema Objects

Vendor	Catalog	Schema
Oracle	Does not support catalogs. When specifying database objects, the catalog field should be left blank.	Typically the name of an Oracle user ID.
DB2	If specifying database objects, the catalog field should be left blank.	Schema name corresponds to the catalog owner of the database, such as db2admin.
Sybase	Catalog name is the database name.	Schema name corresponds to the database owner.
Microsoft SQL Server	Catalog name is the database name.	Schema name corresponds to the catalog owner, such as dbo. The schema name must match the catalog or database owner for the database to which you are connected.
Informix	Does not support catalogs. If specifying database objects, the catalog field should be left blank.	Not needed.
PointBase	PointBase database systems do not support catalogs. If specifying database objects, the catalog field should be left blank.	Schema name corresponds to a database name.

XML Name Conversion Considerations

When a source name is encountered that does not fit within XML naming conventions, default generated names are converted according to rules described by the SQLX standard. Generally speaking, an invalid

XML name character is replaced by its hexadecimal escape sequence (having the form *xUUUU*).

For additional details see section 9.1 of the W3C draft version of this standard:

http://www.sqlx.org/SQL-XML-documents/5WD-14-XML-2003-12.pdf

Create Physical Data Services from Stored Procedures

This page last changed on Apr 01, 2008 .

How To Create Physical Data Services from Stored Procedures

Stored procedures are database objects that group an executable set of SQL and native database programming language statements together to perform a specific task locally. Advanced DBMS systems utilize stored procedures to improve query performance, manage and schedule data operations, enhance security, and so forth.

In ALDSP you can, for specifically supported databases, create physical data services based on stored procedures.

It is often convenient to leverage independent routines as part of managing enterprise information through a data service. An obvious example would be to leverage standalone update or security functions through data services. Such functions have no XML type; in fact, they typically return nothing (or *void*).

Stored procedures are very often side-effecting from the perspective of the data service, since they perform internal operations on data. In such cases all you need to do is identify the stored procedures as a data service procedure when your physical data service is created.

After you have identified the stored procedures that you want to add to your data service, you also have an opportunity to identify which of these should be identified as data service procedures.

Each stored procedure that is imported becomes a separate data service. In other words, if you have five stored procedures, you will create five data services.

The following topics describe how to create a physicald data service from a stored procedure:

- Importing Stored Procedure Metadata Using the Physical Data Service Creation Wizard
- Setting Up the Physical Data Service Creation Wizard
- Setting Up the Import Wizard for Relational Objects
- Selecting Stored Procedure Objects for Import
- Configuring Selected Stored Procedures
- Stored Procedure Configuration Reference

- Setting Properties for New Data Service Operations
- Verifying Data Service Composition
- Adding Operations to an Existing Data Service
- Support for Stored Procedures in Popular Databases

References

- Configuration Information
- Terms Commonly Used When Discussing Stored Procedures

For details on creating and managing stored procedures in your database, see the documentation provided with the DBMS.

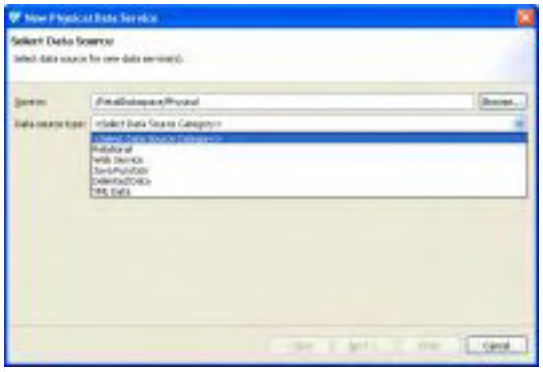
Importing Stored Procedure Metadata Using the Physical Data Service Creation Wizard

The following topics cover the actions necessary to create physical data services from relational stored procedures.

Setting Up the Physical Data Service Creation Wizard

Physical data services are created using a wizard.

Physical Data Service Creation Wizard

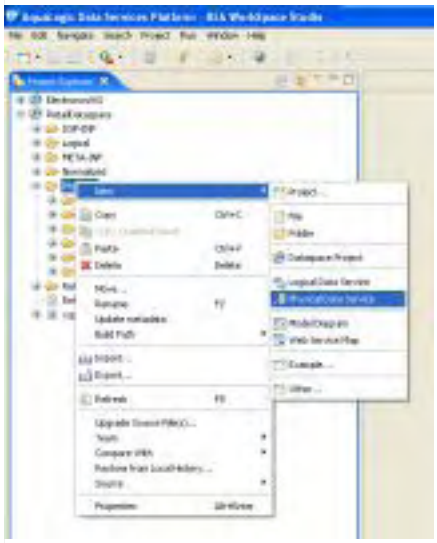


Starting the Wizard

To start the physical data service creation wizard:

1. Right-click on your dataspace project or any folder in your project.
2. Choose New > Physical Data Service

Creating a New Physical Data Service



Setting Up the Import Wizard for Relational Objects

When importing a relational object available options include the ability to:

1. Set a location for your new data service to be saved within your project.
2. Select a data source from the dropdown listbox.
3. Select the database type for the selected source (PointBase for the sample RDBMS) from the dropdown listbox.
4. Select among the relational source types listed in the following table.

Types of available relational data sources

Relational Type	Description
Tables and Views	Displays all public tables and views in the selected data source.
Stored Procedures	Displays all public stored procedures in the selected data source.
SQL Statement	Allows creation of a SQL statement for extracting relational data from the data source.
Database Function	Allows creation of an XQuery function in a library data service based on build-in or custom database functions.

Selecting Stored Procedure Objects for Import

To create physical data services based on stored procedures:

1. Select the Stored Procedures option.
2. Click Next.

A list of available stored procedures appears.

Objects are grouped based on the relational data sources catalog and/or schema.

You can use wildcards to support importing metadata on internal stored procedures. For example, entering the following string as a stored procedure filter:

```
%TRIM%
```

retrieves metadata on the system stored procedure:

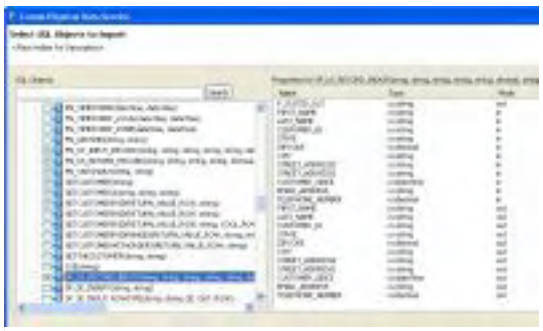
```
STANDARD.TRIM
```

In such a situation you may want to make a "nonsense" entry in the Table/View field in order to avoid retrieving all tables and views in the database.

Database-specific Catalog and Schema Considerations

Simply check the desired objects or their container, which will select all enclosed stored procedures.

Stored Procedure Objects Selected for Import



Filtering SQL Objects Using Search

The Search option available when creating a physical data service can be especially useful when:

- You know specific names of the data source objects you want to turn into data services.
- Your data source may be so large that a filter is needed.
- You may be looking for objects with specific naming characteristics such as:

```
%audit2003%
```

The above search command retrieves all objects that contain the enclosed string.

Using JDBC Syntax to Search SQL Objects

You can search through available SQL objects using standard JDBC wildcard syntax.

- An underscore (_) creates a wildcard for an individual character.
- A percentage sign (%) indicates a wildcard for a string. Entries are case-sensitive.

Another example:

```
CUST%, PAY%
```

entered in the Tables/Views field the above search string returns all tables and views starting with either CUST or PAY.

Special Considerations When Searching Stored Procedures

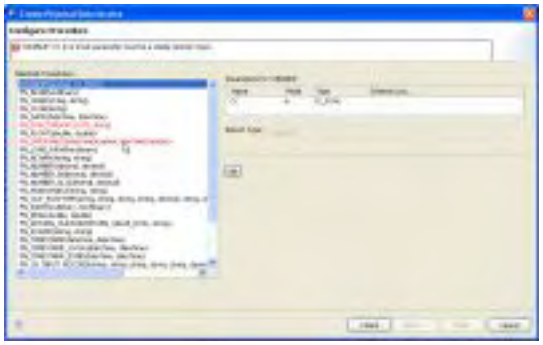
If no items are entered for a particular field, all matching items are retrieved. For example, if no filtering entry is made for the Procedure field, all stored procedures in the data object will be retrieved.

Configuring Selected Stored Procedures

When ALDSP introspects a stored procedure, the process may not be complete. For example, a required item of information such as a schema file or type cannot be determined. When such introspection problems occur, the stored procedure in question is highlighted in red. This setting means that additional information about the procedure must be provided by the user before the data service can be created.

Your goal in correcting an "<unknown>" condition associated with a stored procedure is to bring the metadata obtained by the import wizard into conformance with the actual metadata of the stored procedure. In some cases this will be by correcting the location of the return type. In others you will need to adjust the type associated with an element of the procedure or add elements that were not found during the initial introspection of the stored procedure.

Configure Stored Procedure Dialog



When several stored procedures are selected at the same time for physical data service creation, all the selected procedures must be adequately configured before any data services based on the procedures can be created.

An alternative to configuring a incomplete stored procedure before proceeding is to use the wizard Back button to de-select the procedure in question.

Here are the steps involved in editing a set of stored procedures that will be imported as data services:

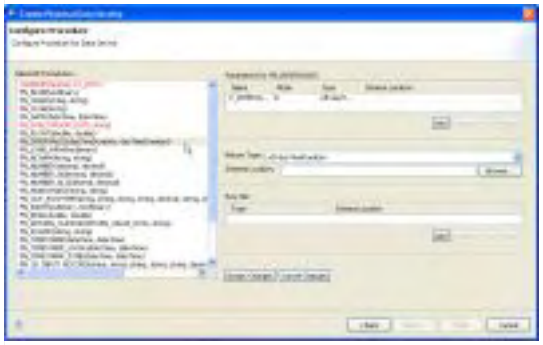
1. Scroll through the list of selected procedures.
2. For each procedure in red type, use the Edit button to correct the configuration settings.
3. Make any other changes. (In some cases the data architect may know of requirements that are not identified during the introspection process.)
4. Click Next when all the procedures in the selected set are valid.

If a stored procedure has only one return value and the value is either simple type or a RowSet which is mapping to an existing schema, no schema file is created. This stored procedure by definition become a library data service.

Editing Stored Procedure Configurations

Stored procedure configuration can be complicated. An understanding of the characteristics of the stored procedure in your database is an essential prerequisite. This section describes stored procedure options in detail.

Stored Procedure Metadata Editing Options



Once in stored procedure configuration edit mode, options are available in three general areas:

- **Parameters.** Stored procedures requiring complex parameters can only be turned into data services once a schema has been identified. In addition, retrieved information on parameters required by a stored procedure may be incorrect. For example, additional parameters may be needed.
- **Return type.** Stored procedures returning complex data require a local schema to handle data returned from the call. In addition, retrieved information on stored procedure return types may be incorrect or it may be the case that no returned data is wanted.
- **Row set.** A row set identifies a schema and its associated library data service to hold information returned by a stored procedure. In some cases multiple row sets may need to be specified.

Stored Procedure Editing Options

Category	Option	Settings	Discussion
Parameters	**Name**	Parameter name	Editable.
	Mode	on/out/inout	
	Type	XQuery type	May be derived from the stored procedure. Primative XQuery type settings are also available.
	Schema location	XSD file	Schema file must be in the project.

Return type	**Type**	XQuery type or global type from selected schema	
	Schema location	XSD file	Schema file must be in the project.
Row set	**Type**	Data service	Derived from selected schema.
	Schema location	XSD file	Schema file must be in the project.

Stored Procedure Configuration Reference

The following topics provide detailed information regarding various configuration options associated with creating data services based on stored procedures.

In Mode, Out Mode, Inout Mode

In, Out, and Inout mode settings determine how a parameter passed to a stored procedure is handled.

Parameter Mode	Effect
In	Parameter is passed by reference or value.
Inout	Parameter is passed by reference.
Out	Parameter is passed by reference. However the parameter being passed is first initialized to a default value. If your stored procedure has an OUT parameter requiring a complex element, you may need to provide a schema.

Procedure Profile

Each element in a stored procedure is associated with a type. If the item is a simple type, you can simply choose from the pop-up list of types. If the type is complex, you may need to supply an appropriate schema. Click on the schema location button and either enter a schema pathname or browse to a schema. The schema must reside in your application.

After selecting a schema, both the path to the schema file and the URI appear.

Complex Parameter Types

Complex parameter types are supported under only three conditions:

- As the output parameter
- As the Return type
- As a rowset

About Rowsets

A rowset type is a complex type.

The rowset type contains a sequence of a repeatable elements (for example called CUSTOMER) with the fields of the rowset.

In some cases the wizard can automatically detect the structure of a rowset and create an element structure. However, if the structure is unknown, you will need to provide it.

 All rowset-type definitions must conform to this structure.

The name of the rowset type can be:

- The parameter name (in case of a input/output or output only parameter).
- An assigned name.
- The referenced element name (result rowsets) in a user-specified schema.

Not all databases support rowsets. In addition, JDBC does not report information related to defined rowsets.

Using Rowset Information

In order to create data services from stored procedures that use rowset information, you need to supply the correct ordinal (matching number) and a schema. If the schema has multiple global elements, select the one you want from the Type column. Otherwise the type used match the first global element in your schema file.

The order of rowset information is significant; it must match the order in your data source. Use the Move Up / Move Down commands to adjust the ordinal number assigned to the rowset.

XML types in data services generated from stored procedures do not display native types. However, you can view the native type in the Source editor; it is located in the pragma section.

Stored Procedure Version Support

Only the most recent version of a particular stored procedure can be imported into ALDSP. For this reason you cannot identify a stored procedure version number when creating a physical data service based on a stored procedure. Similarly, adding a version number for your stored procedure in the Source editor will result in a query exception.

Supporting Stored Procedures with Nullable Input Parameter(s)

If you know that an input parameter of a stored procedure is nullable (can accept null values), you can change the signature of the function in Source View to make such parameters optional by adding a question mark at end of the parameter.

For example (question-mark (**?**) shown in bold):

```
function myProc($arg1 as xs:string) ...
```

would become:

```
function myProc($arg1 as xs:string?) ...
```

Setting Properties for New Data Service Operations

Each new entity data service is created with a Read function that contains all the metadata elements identified during data service creation. It can be thought of as comparable to the following construct in the relational world:

```
select * from <table>
```

Use the Properties dialog to:

- Optionally modify the operation name.
- Set the Public option (check if you want your function to be available to client applications).
- Set the kind of operation (in some cases only Read will be available).
- Set the Primary option (check if you want your function to be the primary of its type).

  In some cases this option may not be available.

- Select a common XML namespace for the entire data service or individual target namespaces for specific operations.
- Set the target namespace.

The root element, which is read-only, is also displayed.

Initially the root element name matches the name of the data service.

Setting Properties for New Data Service Functions

Default Naming Conventions

There are several default naming conventions associated with new data services:

- When a table, view, or other data source object is the source for a data service, the nominated name is wherever possible the same as the source object name. In some cases, however, names are adjusted to conform with XML naming conventions.
- Initially the root element name matches the name of the data service.

XML Name Conversion Considerations

Verifying Data Service Composition

On the Review New Data Service(s) page you can set, confirm or, optionally, change suggested data service names depending on the type of physical data service you are creating.

Default Physical Data Service Names

The nominated name for a new data service is, wherever possible, the same as the source object name. In some cases, however, names are adjusted to conform with XML naming conventions.

XML Name Conversion Considerations

About Automatic Data Service Name Changes

Name conflicts occur when there is a data service of the same name present in the target directory. Name conflicts are highlighted in red.

There are several situations where you will need to change the name of your data service:

- There already is a data service of the same name in your application.
- You are trying to create multiple data services with the same name.

Data services always have the file extension:

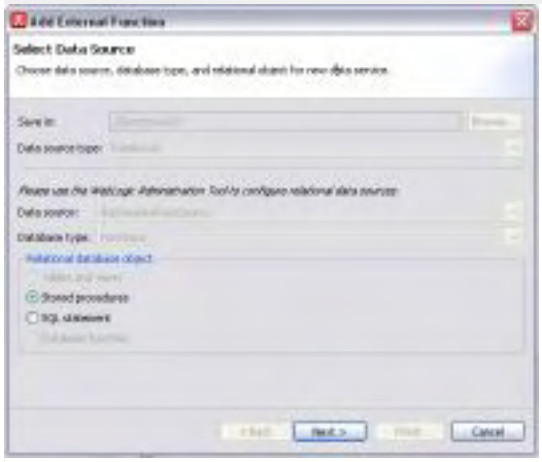
```
.ds
```

Adding Operations to an Existing Data Service

You can add SQL statement or stored procedure operations based on the same data source to an existing physical data service based a stored procedure.

Add an External Function to an Existing Physical Data Service

Adding a Stored Procedure or SQL Statement to a Data Service



Support for Stored Procedures in Popular Databases

Each database vendor approaches stored procedures differently. ALDSP support limitations generally reflect JDBC driver limitations.

General Restrictions

There are several restrictions that apply to stored procedures generally:

- ALDSP does not support rowset as an input parameter.
- Only data types supported by ALDSP can be imported as part of stored procedures.

> For a list of database types supported by ALDSP XQuery-SQL Mapping Reference

Oracle Stored Procedure Support

The following table describes data service creation support for Oracle stored procedures.

Term	Usage
Procedure types	• Procedures • Functions • Packages

Parameter modes	• Input only • Output only • Input/Output • None
Parameter data types	Any Oracle PL/SQL data type except: • ROWID • UROWID  When defining function signatures, note that the Oracle %TYPE and %ROWTYPE types must be translated to XQuery types that match the true types underlying the stored procedure's %TYPE and %ROWTYPE declarations. %TYPE declarations map to simple types; %ROWTYPE declarations map to rowset types.
Data returned from a function	Oracle supports returning PL/SQL data types such as NUMBER, VARCHAR, %TYPE, and %ROWTYPE as parameters.
Comments	The following identifies limitations associated with importing Oracle database procedure metadata. • The data service creation process can only detect the data structure for cursors that have a binding PL/SQL record. For a dynamic cursor you need to manually specify the cursor schema. • Data from a PL/SQL record structure cannot be retrieved due to an Oracle JDBC driver limitations. • The Oracle JDBC driver supports rowset output parameters only if they are defined as reference cursors in a package. • The Oracle JDBC driver does not support NATURALN and POSITIVEN as output only parameters.

Sybase Stored Procedure Support

The following table describes data service creation support for Sybase stored procedures.

Term	Usage

Procedure types	• Procedures • Grouped procedures • Functions are categorized as a scalar or inline table-valued and multi-statement table-valued function. Inline table-valued and multi-statement table-valued functions return rowsets.
Parameter modes	• Input only • Output only
Parameter data types	For a list of database types supported by ALDSP see the XQuery-SQL Mapping Reference.
Data returned from a function	Sybase functions supports returning a single value or a table. Procedures return data in the following ways: • As output parameters, which can return either data (such as an integer or character value). • As return codes, which are always an integer value. • As a rowset for each SELECT statement contained in the stored procedure or any other stored procedures called by that stored procedure. • As a global cursor that can be referenced outside the stored procedure supports, returning single value or multiple values.
Comments	The following identifies limitations associated with importing Sybase database procedure metadata: • The Sybase JDBC driver does not support input/output or output only parameters that are rowsets (including cursor variables). • The Jconnect driver and some versions of the BEA Sybase driver cannot detect the parameter mode of the procedure. In such a case, the return mode will be UNKNOWN, preventing importation of the metadata. To proceed, you need to set the correct mode.

IBM DB2 Stored Procedure Support

The following table describes data service creation support for IBM DB2 stored procedures.

Term	Usage

Procedure types	• Procedures • Functions • Packages where each function is also categorized as a scalar, column, row, or table function. Here are additional details on function categorization: • A scalar function returns a single-valued answer each time it is called. • A column function is one which conceptually is passed a set of like values (a column) and returns a single-valued answer (AVG ( )). • A row function is a function that returns one row of values. • A table function is a function that returns a table to the SQL statement that referenced it.
Parameter modes	• Input only • Output only • Input/output
Parameter data types	For a list of database types supported by ALDSP see the XQuery-SQL Mapping Reference. For a list of database types supported by ALDSP see the XQuery-SQL Mapping Reference.
Data returned from a function	DB2 supports returning a single value, a row of values, or a table.
Comments	The following identifies limitations associated with creating physical data services based on DB2 stored procedures: • Column type functions are not supported. • Rowsets as output parameters are not supported. • The DB2 JDBC driver supports float, double, and decimal input only and output only parameters. Float, double, and decimal data types are not supported as input/output parameters.

Microsoft SQL Server Stored Procedure Support

The following table describes data service creation support for Microsoft stored procedures.

Term	Usage

Procedure types	SQL Server supports procedures, grouped procedures, and functions. Each function is also categorized as a scalar or inline table-valued and multi-statement table-valued function. Inline table-valued and multi-statement table-valued functions return rowsets.
Parameter modes	SQL Server supports input only and output only parameters.
Parameter data types	SQL Server procedures/functions support any SQL Server data type as a parameter. For a list of database types supported by ALDSP see the XQuery-SQL Mapping Reference.
Data returned from a function	SQL Server functions supports returning a single value or a table. Data can be returned in the following ways: • As output parameters, which can return either data (such as an integer or character value) or a cursor variable (cursors are rowsets that can be retrieved one row at a time). • As return codes, which are always an integer value. • As a rowset for each SELECT statement contained in the stored procedure or any other stored procedures called by that stored procedure.
Comments	The following identifies limitations associated with importing SQL Server procedure metadata. • Result sets returned from SQL server (as well as those returned from Sybase) are not detected automatically. Instead you will need to manually add parameters as a result. • The Microsoft SQL Server JDBC driver does not support rowset input/output or output only parameters (including cursor variables).

Create Physical Data Services Based on SQL Statements

This page last changed on Mar 13, 2008 .

How To Create Physical Data Services Based on SQL Statements

The following topics cover the actions necessary to create physical data services from SQL statements:

Setting Up the Physical Data Service Creation Wizard

Physical data services are created using a wizard.

Physical Data Service Creation Wizard

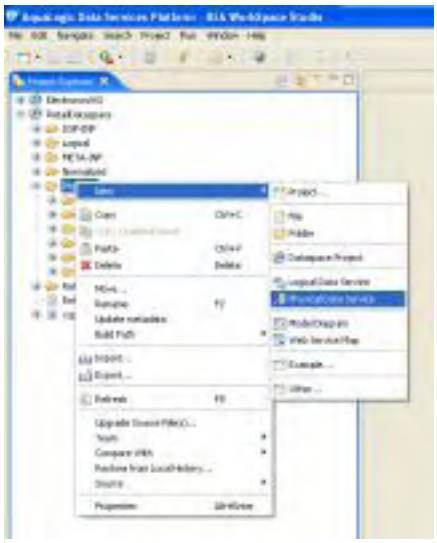
Starting the Wizard

To start the physical data service creation wizard:

1. Right-click on your dataspace project or any folder in your project.

2. Choose New > Physical Data Service

Creating a New Physical Data Service



Setting Up the Import Wizard for Relational Objects

When importing a relational object available options include the ability to:

1. Set a location for your new data service to be saved within your project.
2. Select a data source from the dropdown listbox.
3. Select the database type for the selected source (PointBase for the sample RDBMS) from the dropdown listbox.
4. Select among the relational source types listed in the following table.

Types of available relational data sources

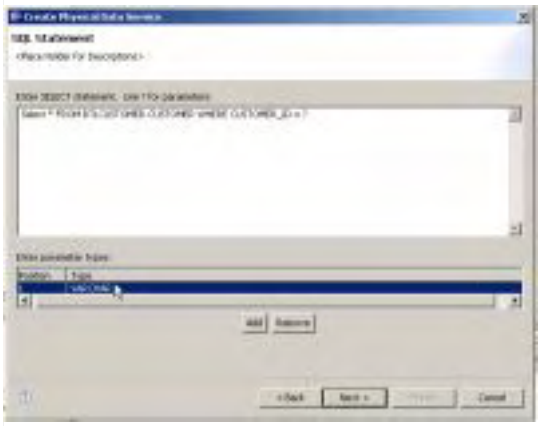
Relational Type	Description
Tables and Views	Displays all public tables and views in the selected data source.
Stored Procedures	Displays all public stored procedures in the selected data source.
SQL Statement	Allows creation of a SQL statement for extracting relational data from the data source.
Database Function	Allows creation of an XQuery function in a library data service based on build-in or custom database functions.

Entering a SQL Statement

You can build library data service functions based on SQL statements. The XQuery engine uses the statement to retrieve metadata which is, in turn, formulated into a function that can be used by other data services or made public.

After selecting the SQL Statement option the next page of the wizard allows you to enter a SELECT statement and any necessary parameters.

SQL Statement Entry Dialog



You can type or paste your SELECT statement into the SELECT statement box, indicating parameters with a question-mark symbol.

```
?
```

Using one of the ALDSP samples, the following SELECT statement can be used:

```
SELECT * FROM RTLCUSTOMER.CUSTOMER WHERE CUSTOMER_ID = ?
```

For the parameter field, you would need to select a data type. In this case, CHAR or VARCHAR.

1. Click Add to insert a new row into the parameter table, which indicates a parameter for the SQL statement.
2. Select Parameter Type from the drop-down combo box.

Notes:

- When you run your query under Test View, you will need to supply the parameter in order for the query to run successfully.
- ALDSP needs to be able to refer to the columns of the result of your SQL statement by name. To ensure that this is possible, you should use aliases as needed to ensure that computed columns indeed have usable names.
- The position of the parameter is significant.

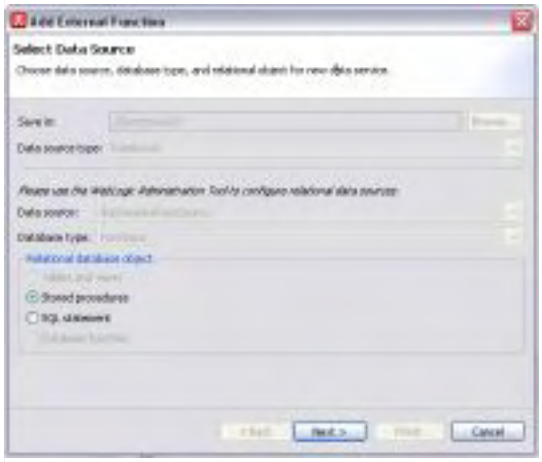
3. In Test view run your query, supplying a parameter such as CUSTOMER3.

Adding Operations to an Existing Data Service

You can add SQL statement or stored procedure operations based on the same data source to an existing physical data service based a SQL statement.

Add an External Function to an Existing Physical Data Service

Adding a Stored Procedure or SQL Statement to a Data Service



Setting Properties for New Library Functions

This general topic applies to setting properties for all types of library data service functions.

Use the Review New Data Service Operations page to:

- Change the function name.
- Set the Public option (check if you want your function to be available to client applications).
- Set the kind of function (in some cases only one option will be available).
- Set the Primary option (check if you want your function to be the primary of its type).

   In some cases this option may not be available.

- Select a common XML namespace for the entire data service.
- Set the target namespace.

The root element, which is read only, is also displayed.

Verifying Data Service Composition

On the Review New Data Service(s) page you can set, confirm or, optionally, change suggested data service names depending on the type of physical data service you are creating.

Default Physical Data Service Names

The nominated name for a new data service is, wherever possible, the same as the source object name. In some cases, however, names are adjusted to conform with XML naming conventions.

 XML Name Conversion Considerations

About Automatic Data Service Name Changes

Name conflicts occur when there is a data service of the same name present in the target directory. Name conflicts are highlighted in red.

There are several situations where you will need to change the name of your data service:

- There already is a data service of the same name in your application.
- You are trying to create multiple data services with the same name.

Data services always have the file extension:

```
.ds
```

Create Physical Data Services Based on Database Functions

This page last changed on Mar 13, 2008 .

How To Create Physical Data Services Based on Database Functions

You can create library physical data services based on two types of database functions:

- Functions that are provide with your database.
- *Custom functions* that you have created and stored in your database.

A library data service created based on database functions is restricted to that type of function. For example, a library function based on a stored procedure cannot be added to a library data service that contains database functions.

The following topics describe how you can create physical data services from database functions:

- Setting Up the Physical Data Service Creation Wizard
- Setting Up the Import Wizard for Relational Objects
- Providing Database Function Details
- Verifying Data Service Composition

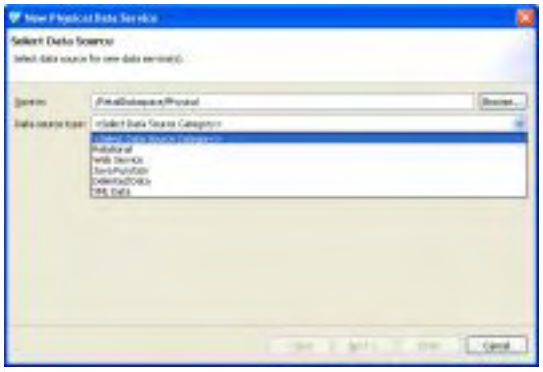
You can use the the physical data service creation wizard to:

- Select relational as the Data Source type.
- Select a data source from available relational sources.
- Choose a database type. Database types listed would be drawn from the list of available database providers for your data source. By default GenericSQL, the base platform provider, and Pointbase are provided.
- Select the Database function option.

Setting Up the Physical Data Service Creation Wizard

Physical data services are created using a wizard.

Physical Data Service Creation Wizard

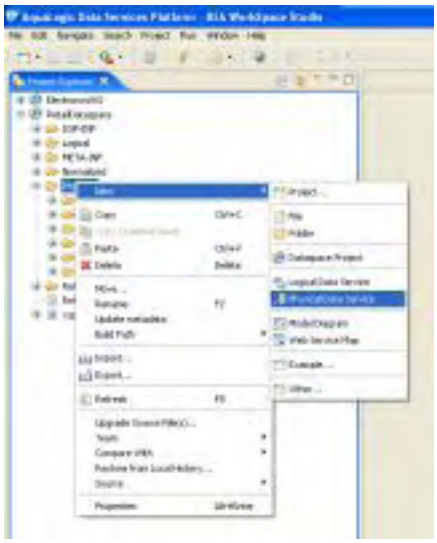


Starting the Wizard

To start the physical data service creation wizard:

1. Right-click on your dataspace project or any folder in your project.
2. Choose New > Physical Data Service

Creating a New Physical Data Service



Setting Up the Import Wizard for Relational Objects

When importing a relational object available options include the ability to:

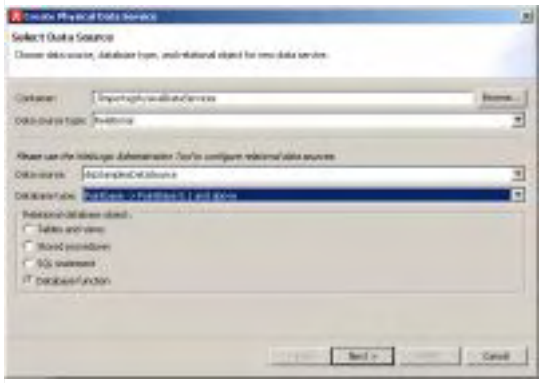
1. Set a location for your new data service to be saved within your project.
2. Select a data source from the dropdown listbox.
3. Select the database type for the selected source (PointBase for the sample RDBMS) from the dropdown listbox.
4. Select among the relational source types listed in the following table.

Types of available relational data sources

Relational Type	Description
Tables and Views	Displays all public tables and views in the selected data source.
Stored Procedures	Displays all public stored procedures in the selected data source.
SQL Statement	Allows creation of a SQL statement for extracting relational data from the data source.
Database Function	Allows creation of an XQuery function in a library data service based on build-in or custom database functions.

1. In the Select a Data Source dialog choose Database function.
2. Click Next.

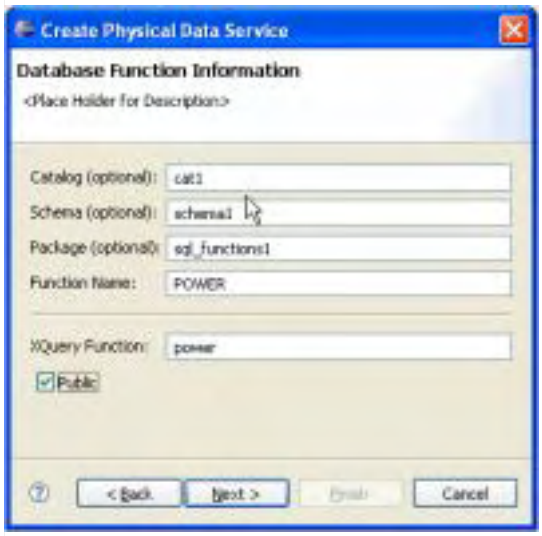
Importing Database Function Metadata



Providing Database Function Details

1. Select a data source from the dropdown list of data sources available to your server. You should identify a data source that contains the built-in or user-defined database functions you want to access through your data services.
2. Enter the information necessary to identify your database function.
3. Complete the function definition including identifying parameters in Source view.

Entering Database Function Information



Database Function Information Dialog Options

Option	Action	Comment/Reference
Catalog:	**Enter catalog name, if needed by your RDBMS**	
Schema:	**Enter schema name, if needed by your RDBMS**	
Package:	**Enter package name, if needed by your RDBMS**	
Function name:	**Database function name**	Required.
XQuery function	**XQuery function name**	Required; will invoke the database function.
Public	**Select, if you want to make your operation public**	Default for created XQuery functions is protected.
	Click Next	
Review	**Enter library data service name**	If the name of an existing library data service is provided.

There is no type checking or other type of verification regarding external function parameters.

Verifying Data Service Composition

On the Review New Data Service(s) page you can set, confirm or, optionally, change suggested data service names depending on the type of physical data service you are creating.

Default Physical Data Service Names

The nominated name for a new data service is, wherever possible, the same as the source object name. In some cases, however, names are adjusted to conform with XML naming conventions.

XML Name Conversion Considerations

About Automatic Data Service Name Changes

Name conflicts occur when there is a data service of the same name present in the target directory. Name conflicts are highlighted in red.

There are several situations where you will need to change the name of your data service:

- There already is a data service of the same name in your application.
- You are trying to create multiple data services with the same name.

Data services always have the file extension:

```
.ds
```

Create a Physical Data Service from a Web Service

This page last changed on Mar 13, 2008 .

How To Create a Physical Data Service from a Web Service

A Web service is a self-contained, platform-independent unit of business logic that is accessible through application adaptors, as well as standards-based Internet protocols such as HTTP or SOAP.

Web services greatly facilitate application-to-application communication. As such they are increasingly central to enterprise data resources. A familiar example of an externalized Web service is a frequent-update weather portlet or stock quotes portlet that can easily be integrated into a Web application.

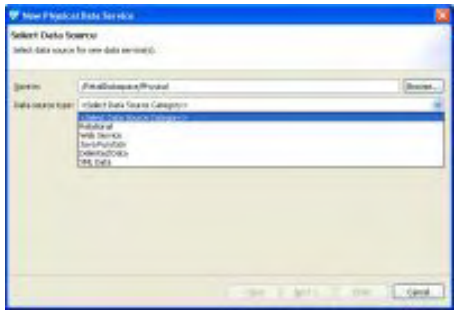
Similarly, a Web service can be effectively used to track a drop shipment order from a seller to a manufacturer.

The following topics describe how you can create physical data services from web services:

Setting Up the Physical Data Service Creation Wizard

Physical data services are created using a wizard.

Physical Data Service Creation Wizard

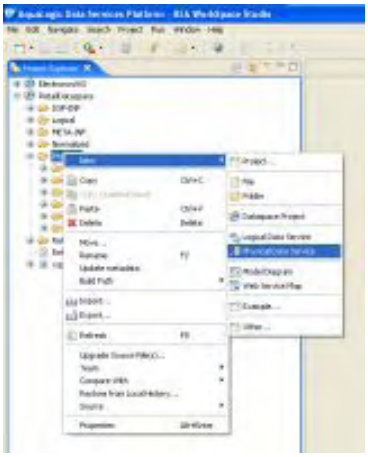


Starting the Wizard

To start the physical data service creation wizard:

1. Right-click on your dataspace project or any folder in your project.
2. Choose New > Physical Data Service

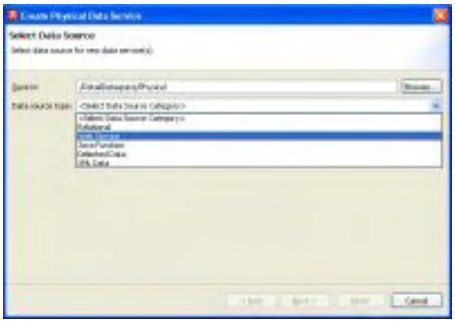
Creating a New Physical Data Service



Accessing a Web Service

Once you select web service as your data source, you will be given the option of specifying a URI or WSDL file.

Selecting Web Service as a Data Source



There are several ways to access a specific web service:

- From a Web Service Description Language (WSDL) file located in your current ALDSP project
- From a WSDL accessible via a URL
- Through an ALSB proxy service

> ⚠ You can test the ability to create a physical data service based on a web service using the following WSDL (available as of this writing):
>
> ```
> http://www.whitemesa.net/wsdl/r2/base.wsdl
> ```

Importing Metadata from a WSDL

The ALSB Proxy Service Option

To access web services through AquaLogic Service Bus (ALSB) you need to:

- Provide access and credential information to AquaLogic Service Bus.
- Select a proxy service (if there is more than one).

AquaLogic Service Bus access requires providing the following:

- Server name
- Port number
- User name
- Password

This information should be available from your AquaLogic Service Bus administrator.

> ⚠ You must configure the ALSB proxy service to use the sb transport protocol to enable access through ALDSP.

After the required information is provided, the WSDL will become available using the name of the selected proxy service.

Steps in Importing a Web Service

1. Enter a Web service URL, local WSDL, or ALSB proxy.
2. Click Next.

Selecting Web Service Operations to Import

From the list of available webservice operations grouped by serviceName and portname, choose the operation that you want to turn into data service operation.

Selecting Web Service Operations

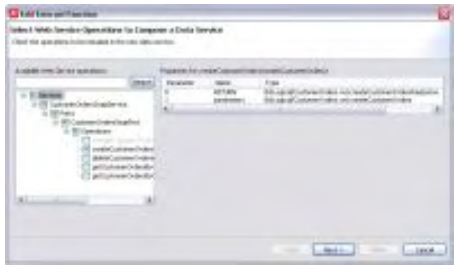
During the import process you will be choosing the operations you want to import, setting names and other characteristics. These choices will determine whether a Library or Entity data service will be created. Thus a familiarity with the operations of your Web service is needed.

Adding Operations to an Existing Data Service

You can add operations to an existing physical data service based a web service by adding an external function from the same WSDL.

Add an External Function to an Existing Physical Data Service

Adding an External Operation to a Data Service



Steps Involved in Selecting Web Service Operations

1. Select the operations you want to turn into data services or library data service functions.
2. Click Next.

Setting Characteristics of Imported Web Service Operations

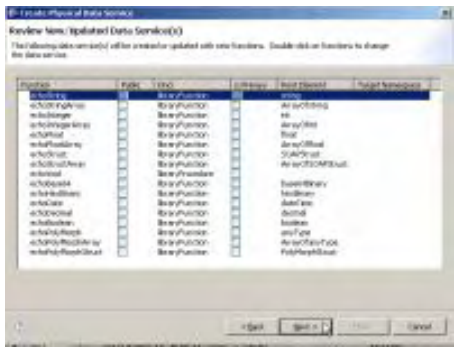
The following table describes available options for each operation you have selected to import.

Options Available for Imported Web Service Operations

Characteristic	Options	Comment

Operation name	adjust as needed	You can change the nominated name to any legal XML name using the built-in line editor.
Public	Boolean	By default Web service-derived operations are protected. A checkbox allows you to mark any function or procedure as public. (Once in a data service, operations can be marked private as needed.) Operations determined to return void are automatically marked as library procedures.
Kind	• Read • Create • Update • Delete • Library function • Library procedure	You can change the nominated function type. The wizard attempts to correctly set the function type being imported. ⚠ Operations marked as create, update, or delete functions will be packaged in an Entity data service. Otherwise, the resulting data service will be of type Library.
is Primary	Boolean	Not applicable for web service operations.
Root Element	Root element of the operation	For complex data types the topmost element is listed. In case of RPC-style web services the top-most generated element is listed.
Target Namespace	imported value	This represents the target namespace of the generated data service.

Setting Characteristics of Imported Web Service Operations



Setting the Data Service Name

You can change the name of your data service to any legal name that does not conflict with another name in the current data space.

In addition, if there already is a data service in your project based on the same WSDL an option to add the new operation to the existing data service appears.

⚠ When importing a web service operation that itself has one or more dependent (or referenced) schemas, the wizard creates second-level schemas according to internal naming conventions. If several operations reference the same secondary schemas, the generated name for the secondary schema may change if you re-import or synchronize with the Web service.

Generally Available Test WSDLs

As of this writing the following sample URIs can be used for experimentation with importing WSDLs as data services:

- http://ws.strikeiron.com/SwanandMokashi/StockQuotes?wsdl
- http://www.whitemesa.net/wsdl/std/echoheadersvc.wsdl

Implementation Notes

This section contains implementation notes.

Special Considerations when Creating a Data Service Based on a RPC-Style Web Service

In case of RPC-style web services, results are return as qualified or unqualified based on the setting of the schema attribute:

```
elementFormDefault
```

In the general case of web services, elementFormDefault can be overridden by setting the form attribute for any child element. However, such 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 the above code 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 will be honored. In the case of RPC-style web services, however, the runtime will generate "qualified" attributes for all the elements, including ORDER_ID.

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. In order for such web services to be turned 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

Multi-dimensional arrays in RPC mode are not supported.

See Also

How To Create SOAP Handlers for Imported WSDLs

Attachments:

- Intercept Handler.txt (text/plain)
- Intercept Configuration.txt (text/plain)

Create a Physical Data Service from a Java Function

This page last changed on Mar 24, 2008.

How To Create a Physical Data Service from a Java Function

You can create physical data services based on custom Java functions that return both simple and complex types.

Before you can create physical data services based on custom Java functions, you must create a Java class containing both the schema and function information. For more information, see Preparing to Create Physical Data Services From Java Functions.

For more information about supported Java types and the corresponding generated data services, see Physical Data Services from Java Functions Overview.

Topics

Setting Up the Physical Data Service Creation Wizard

Physical data services are created using a wizard.

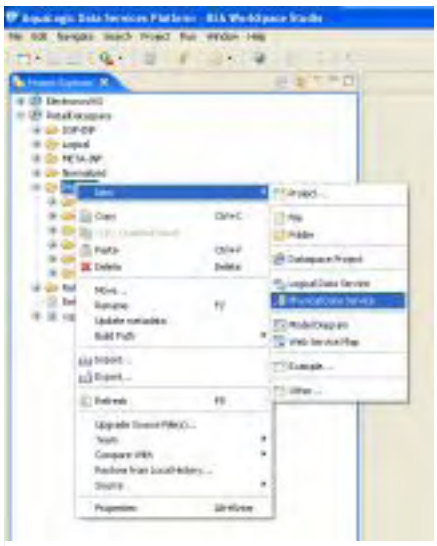
Physical Data Service Creation Wizard

Starting the Wizard

To start the physical data service creation wizard:

1. Right-click on your dataspace project or any folder in your project.
2. Choose New > Physical Data Service

Creating a New Physical Data Service



Accessing Java Functions

After you choose Java Function as your data source, you need to specify a class name containing the Java functions.

Choosing Java Function as a Data Source

Choosing the Java class:

To choose the Java class containing the Java functions:

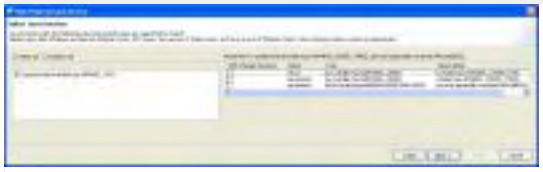
1. Choose Java Function from the Data source type drop-down list.
2. Click Browse. The Open Java Class dialog appears.
3. Select the Java `.class` file and click Open.
   The `.class` file must reside in the same dataspace into which you are importing the Java functions.
4. Click Next.

Open Java Class Dialog

Selecting Java Functions to Import

After you select Java Function as your data source, you need to select the Java functions to import.

Selecting Java Function Dialog



To select the Java functions to import:

1. Select the Java functions you want to import by checking the corresponding box.
   Select `With Change Summary` to have ALDSP declare the parameter or return value as changed-element enabling you to use it with update operations. This option is only available for SDO DataObject-generated classes.
2. Click Next.

Setting Characteristics of Imported Java Functions

After choosing the Java functions to import, you can optionally set the characteristics of the functions.

Setting Characteristics of Imported Java Functions

The following table describes the available options for each function you have selected to import.

Options Available for Imported Java Functions

Characteristic	Options	Comment
Operation name	Adjust as needed	You can change the nominated name to any legal XML name using the built-in line editor.
Public	Boolean	By default Java function-derived operations are protected. A checkbox allows you to mark any function or procedure as public. (Once in a data service, operations can be marked private as needed.)

Kind	• Read • Create • Update • Delete • Library function • Library procedure	Functions determined to return void are automatically marked as library procedures. You can change the nominated function type. The wizard attempts to correctly set the function type being imported. ⚠ Operations marked as create, update, or delete functions will be packaged in an Entity data service. Otherwise, the resulting data service will be of type Library.
is Primary	Boolean	Not applicable for Java functions.
Root Element	Root element of the operation	For complex data types the topmost element is listed.
Target Namespace	Imported value	This represents the target namespace of the generated data service.

To set the characteristics of imported Java functions:

1. Optionally edit the details of each operation:
2. Click Next.

Setting the Physical Data Service Name

You can set the name of your data service to any legal name that does not conflict with another name in the current dataspace.

To complete the wizard:

1. Type the name of the data service in the Data service name field.
2. Click Finish.

ALDSP creates a pragma (visible in Source view) that defines the function signature and relevant schema type for complex types such as schema elements or SDO types.

If there are existing data services in your project, you have the option of adding functions and procedures to that library or creating a new library for them. All the Java file functions are located in the same data service.

When importing a Java function that itself has one or more dependent (or referenced) schemas, the wizard creates second-level schemas according to internal naming conventions. If several operations reference the same secondary schemas, the generated name for the secondary schema may change if you re-import or synchronize with the Java class.

See Also

Concepts

- Physical Data Services from Java Functions Overview

How Tos

- Preparing to Create Physical Data Services From Java Functions
- Creating XMLBean Support for Java Functions

Create a Physical Data Service from an XML File

This page last changed on Mar 13, 2008 .

How To Create a Physical Data Service from XML Data

XML files are a convenient means of handling hierarchical data. XML files and associated schemas are easily turned into library data service functions.

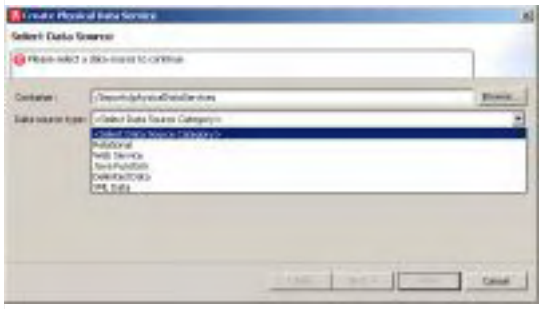
The following topics cover the actions necessary to create physical data services from XML data:

- Setting Up the Physical Data Service Creation Wizard
- Specifying XML Data Schema and File
- Setting Properties for New Library Functions
- Verifying Data Service Composition
- XML File Import Sample

You can use the the physical data service creation wizard to:

- Select XML Data as the Data Source type.
- Select a schema file and option data file.
- Create a Library data service based on the XML data.

Physical Data Service Creation Wizard



Setting Up the Physical Data Service Creation Wizard

Physical data services are created using a wizard.

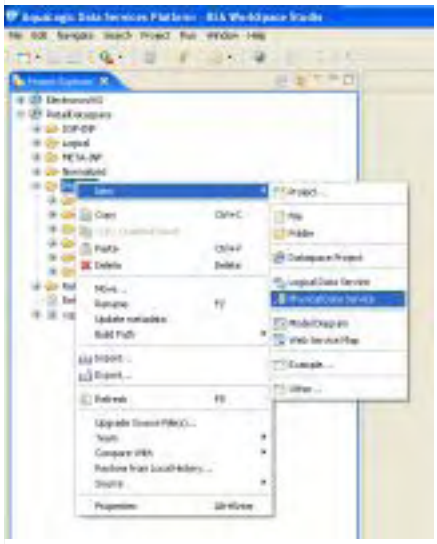
Physical Data Service Creation Wizard

Starting the Wizard

To start the physical data service creation wizard:

1. Right-click on your dataspace project or any folder in your project.
2. Choose New > Physical Data Service

Creating a New Physical Data Service



Specifying XML Data Schema and File

A physical data service based on XML data requires identification of a valid XML schema and, optionally, a data source.

Import XML Data Wizard

The scheme must be available in your dataspace.

The data source can be:

- File-based
- URI-based

In most cases the XML data will be available at runtime, through a URI.

However, in cases where the XML data is also in your project you can specify an absolute location for the file. You can also import data from any XML file on your system using an absolute path prepended with the following:

```
file:///
```

For example, on Windows systems you can access an XML file such as Orders.xml from the root C: directory using the following URI:

```
file:///c:/Orders.xml
```

On a UNIX system, you would access such a file with the following URI:

```
file:///home/Orders.xml
```

Setting Properties for New Library Functions

This general topic applies to setting properties for all types of library data service functions.

Use the Review New Data Service Operations page to:

- Change the function name.
- Set the Public option (check if you want your function to be available to client applications).
- Set the kind of function (in some cases only one option will be available).
- Set the Primary option (check if you want your function to be the primary of its type).

   In some cases this option may not be available.

- Select a common XML namespace for the entire data service.
- Set the target namespace.

The root element, which is read only, is also displayed.

Verifying Data Service Composition

On the Review New Data Service(s) page you can set, confirm or, optionally, change suggested data service names depending on the type of physical data service you are creating.

Default Physical Data Service Names

The nominated name for a new data service is, wherever possible, the same as the source object name. In some cases, however, names are adjusted to conform with XML naming conventions.

 XML Name Conversion Considerations

About Automatic Data Service Name Changes

Name conflicts occur when there is a data service of the same name present in the target directory. Name conflicts are highlighted in red.

There are several situations where you will need to change the name of your data service:

- There already is a data service of the same name in your application.
- You are trying to create multiple data services with the same name.

Data services always have the file extension:

```
.ds
```

XML File Import Sample

An XML file import sample can be found in the sample RTLApp directory:

```
DataServices/Demo
```

Testing the Import Wizard with an XML Data Source

When you create metadata for an XML data source but do not supply a data source name, you will need to identify the URI of your data source as a parameter when you execute the data service's read function.

The identification takes the form of:

```
<uri>/path/filename.xml
```

where *uri* is representative of a path or path alias, *path* represents the directory and *filename.xml* represents the filename. The `.xml` extension is required.

Create a Physical Data Service from an XML File

This page last changed on Mar 13, 2008 .

How To Create a Physical Data Service from XML Data

XML files are a convenient means of handling hierarchical data. XML files and associated schemas are easily turned into library data service functions.

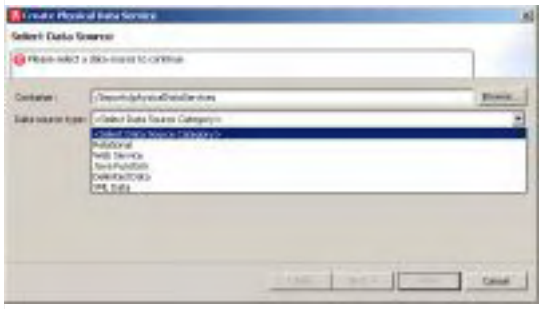
The following topics cover the actions necessary to create physical data services from XML data:

- Setting Up the Physical Data Service Creation Wizard
- Specifying XML Data Schema and File
- Setting Properties for New Library Functions
- Verifying Data Service Composition
- XML File Import Sample

You can use the the physical data service creation wizard to:

- Select XML Data as the Data Source type.
- Select a schema file and option data file.
- Create a Library data service based on the XML data.

Physical Data Service Creation Wizard



Setting Up the Physical Data Service Creation Wizard

Physical data services are created using a wizard.

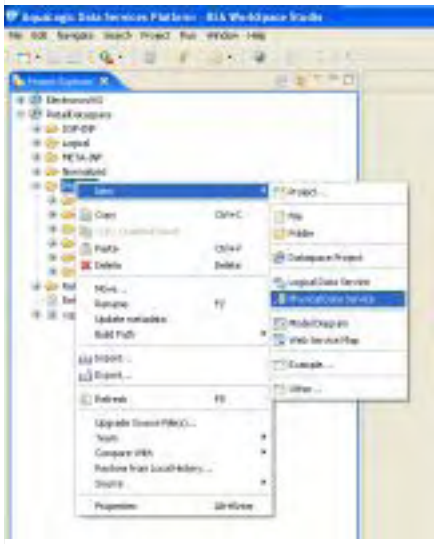
Physical Data Service Creation Wizard

Starting the Wizard

To start the physical data service creation wizard:

1. Right-click on your dataspace project or any folder in your project.
2. Choose New > Physical Data Service

Creating a New Physical Data Service



Specifying XML Data Schema and File

A physical data service based on XML data requires identification of a valid XML schema and, optionally, a data source.

Import XML Data Wizard

The scheme must be available in your dataspace.

The data source can be:

- File-based
- URI-based

In most cases the XML data will be available at runtime, through a URI.

However, in cases where the XML data is also in your project you can specify an absolute location for the file. You can also import data from any XML file on your system using an absolute path prepended with the following:

```
file:///
```

For example, on Windows systems you can access an XML file such as Orders.xml from the root C: directory using the following URI:

```
file:///c:/Orders.xml
```

On a UNIX system, you would access such a file with the following URI:

```
file:///home/Orders.xml
```

Setting Properties for New Library Functions

This general topic applies to setting properties for all types of library data service functions.

Use the Review New Data Service Operations page to:

- Change the function name.
- Set the Public option (check if you want your function to be available to client applications).
- Set the kind of function (in some cases only one option will be available).
- Set the Primary option (check if you want your function to be the primary of its type).

   In some cases this option may not be available.

- Select a common XML namespace for the entire data service.
- Set the target namespace.

The root element, which is read only, is also displayed.

Verifying Data Service Composition

On the Review New Data Service(s) page you can set, confirm or, optionally, change suggested data service names depending on the type of physical data service you are creating.

Default Physical Data Service Names

The nominated name for a new data service is, wherever possible, the same as the source object name. In some cases, however, names are adjusted to conform with XML naming conventions.

 XML Name Conversion Considerations

About Automatic Data Service Name Changes

Name conflicts occur when there is a data service of the same name present in the target directory. Name conflicts are highlighted in red.

There are several situations where you will need to change the name of your data service:

- There already is a data service of the same name in your application.
- You are trying to create multiple data services with the same name.

Data services always have the file extension:

```
.ds
```

XML File Import Sample

An XML file import sample can be found in the sample RTLApp directory:

```
DataServices/Demo
```

Testing the Import Wizard with an XML Data Source

When you create metadata for an XML data source but do not supply a data source name, you will need to identify the URI of your data source as a parameter when you execute the data service's read function.

The identification takes the form of:

```
<uri>/path/filename.xml
```

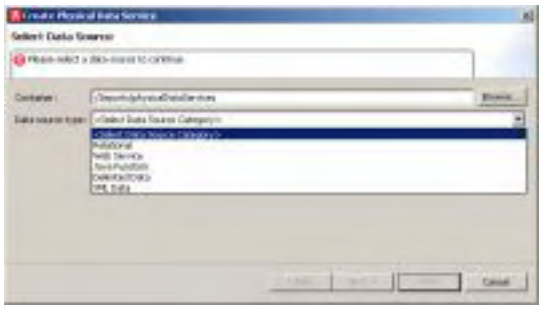
where *uri* is representative of a path or path alias, *path* represents the directory and *filename.xml* represents the filename. The `.xml` extension is required.

Create a Physical Data Service from a Delimited File

This page last changed on Mar 19, 2008 .

How To Create a Physical Data Service from a Delimited File

Spreadsheets offer a highly adaptable means of storing and manipulating information, especially information which needs to be changed quickly. You can easily turn such spreadsheet data into a data services.



Spreadsheet documents are often referred to as CSV files, standing for *comma-separated values*. Although CSV is not a typical native format for spreadsheets, the capability to save spreadsheets as CSV files is very common.

You can use the the physical data service creation wizard to:

- Select a delimited file as the Data Source type.
- Select either a schema file or a file with delimited data.
- Specify whether the information has a header or not.
- Specify delimiter.
- Specify a fixed width value for each column.

Physical Data Service Creation Wizard

The following topics cover the actions necessary to create physical data services from delimited files:

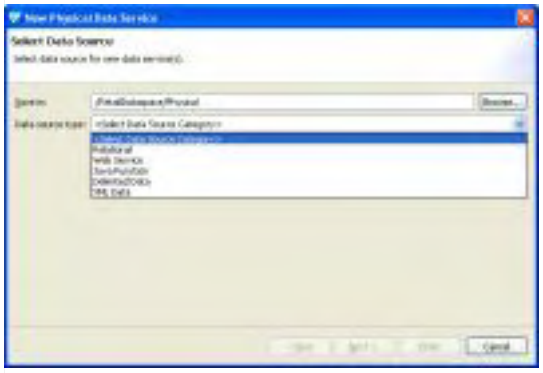
- Setting Up the Physical Data Service Creation Wizard
- Specifying Delimited File Information
- Setting Properties for New Library Functions

- Verifying Data Service Composition

Setting Up the Physical Data Service Creation Wizard

Physical data services are created using a wizard.

Physical Data Service Creation Wizard

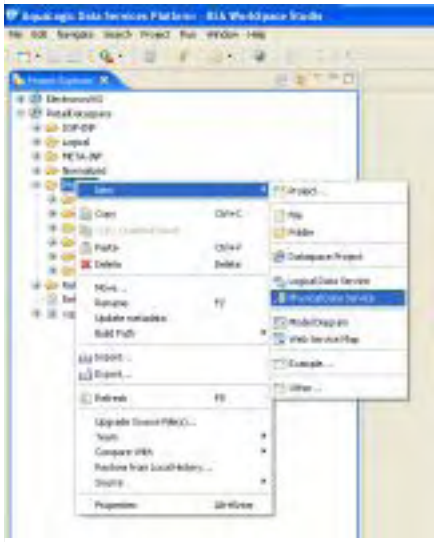


Starting the Wizard

To start the physical data service creation wizard:

1. Right-click on your dataspace project or any folder in your project.
2. Choose New > Physical Data Service

Creating a New Physical Data Service

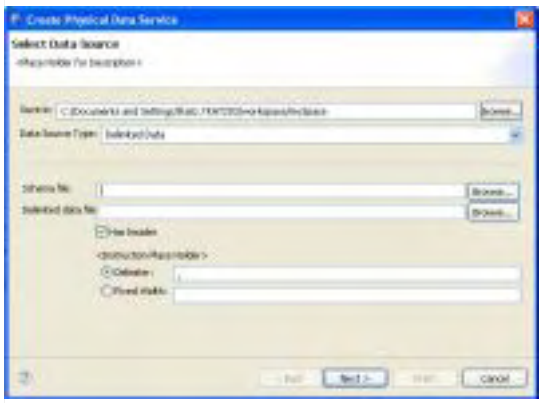


Specifying Delimited File Information

A Library data service based on delimited data requires:

1. Schema in your project and/or a
2. Location of the delimited data file

Import Delimited File Data Wizard



The schema and data file must be available in your dataspace.

Providing a Document Name, a Schema Name, or Both

There are several approaches to developing metadata around delimited information, depending on your needs and the nature of the source.

- **Provide a delimited document name only.** If you supply the import wizard with the name of a valid CSV file, the wizard will automatically create a schema based on the columns in the document. All the columns will be of type string, although you can later modify the generated schema with more accurate type information. The generated schema will have the same name as the source file.
- **Providing a schema name only.** This option is typically used when the source file is dynamic; for example, when data is streamed.
- **Providing both a schema and a document name.** Providing a schema with a CSV file gives you the ability to more accurately type information in the columns of a delimited document.

Locating the CSV File

Using the import wizard you can browse to any file in your project. You can also import data from any CSV file on your system using an absolute path prepended with:

```
file:///
```

For example, on Windows systems you can access an XML file such as Orders.xml from the root C: directory using the following URI:

```
file:///<c:/home>/Orders.csv
```

On a UNIX system, you would access such a file with the URI:

```
file:///<home>/Orders.csv
```

Import Delimited Data Options

- **Header.** Indicates whether the delimited file contains header data. Header data is located in the first row of the spreadsheet. If you check this option, the first row will not be treated as imported data.
- **Delimited or Fixed Width.** Data in your file is either separated by a specific character (such as a comma) or is of a fixed width (such as 10 spaces). If the data is delimited, you also need to provide the delimited character. By default the character is a comma.

Supported Datatypes

The following datatypes are supported for delimited file metadata import operations:

```
XMLSchemaType.BASE64BINARY
XMLSchemaType.BOOLEAN
XMLSchemaType.DATE
XMLSchemaType.DATETIME
XMLSchemaType.DECIMAL
XMLSchemaType.DOUBLE
XMLSchemaType.FLOAT
XMLSchemaType.INT
XMLSchemaType.INTEGER
XMLSchemaType.LONG
XMLSchemaType.STRING
XMLSchemaType.SHORT
```

Additional Considerations

- The number of delimiters in each row must match the number of header columns in your source minus one (# of columns-1). If subsequent rows contain more than the maximum number of delimiters (fields), subsequent use of the data service will not be successful.
- If the delimited file has rows with a variable number of delimiters (fields), you can supply a schema that contains optional elements for the trailing set of extra elements.
- Not all characters are handled the same way. Some charactters may need special escape sequences before spreadsheet data can be accessed at runtime.

Setting Properties for New Library Functions

This general topic applies to setting properties for all types of library data service functions.

Use the Review New Data Service Operations page to:

- Change the function name.
- Set the Public option (check if you want your function to be available to client applications).
- Set the kind of function (in some cases only one option will be available).
- Set the Primary option (check if you want your function to be the primary of its type).

In some cases this option may not be available.

- Select a common XML namespace for the entire data service.
- Set the target namespace.

The root element, which is read only, is also displayed.

Verifying Data Service Composition

On the Review New Data Service(s) page you can set, confirm or, optionally, change suggested data service names depending on the type of physical data service you are creating.

Default Physical Data Service Names

The nominated name for a new data service is, wherever possible, the same as the source object name. In some cases, however, names are adjusted to conform with XML naming conventions.

XML Name Conversion Considerations

About Automatic Data Service Name Changes

Name conflicts occur when there is a data service of the same name present in the target directory. Name conflicts are highlighted in red.

There are several situations where you will need to change the name of your data service:

- There already is a data service of the same name in your application.
- You are trying to create multiple data services with the same name.

Data services always have the file extension:

```
.ds
```

Enable Optimistic Locking

This page last changed on Mar 11, 2008 .

How To Enable Optimistic Locking

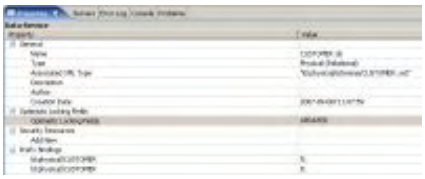
This topic describes how to enable optimistic locking in order to update a physical relational data source.

Set the Locking Policy

Define the optimistic locking policy on the physical data sources that support your logical data service before you attempt to test an update in Test view or use an update map. Optimistic locking is used with physical data sources that are relational.

The current value of optimistic locking is defined in the Optimistic Locking Fields property. You can see this property in the Properties tab in Studio Overview mode.

Checking the Optimistic Locking Policy



Updates to relational data sources use a special XML structure called a data graph. The root element of data graph is <sdo:datagraph>, and the data graph also has a <changeSummary> element.

You can use any of these values for Optimistic Locking Fields. They describe how the elements in the data graph compare to fields in the relational data source.

Value of Optimistic Locking Fields	Effect
PROJECTED	All elements in the data graph are mapped to the data source to verify whether it can be updated. Default value.
UPDATED	Only elements that have changed in your data graph are used to verify whether the data source has changed.
SELECTED FIELDS	Selected elements are used to verify whether the data source has changed. The elements must be non-key elements.

To set the locking policy:

1. Open a physical data service in Studio.
2. Click the Overview tab, then below it, the Properties tab.
3. At Optimistic Locking Fields, click in the Value column, then choose a value.

Select the Locking Fields

If you choose SELECTED FIELDS, you must also select the fields used to verify changes in the data source. You can select any number of non-key fields. The key fields are used to identify the data records to be updated. If you select a complex element, its child elements also become selected elements.

You can also disable a field once it is selected.

Choosing Fields for Optimistic Locking

To select the fields used for optimistic locking:

1. Click the Overview tab.
2. Right-click a non-key element in the return type.

   Key elements are marked with .
3. Choose Enable Optimistic Locking.

When you enable optimistic locking for a field, its icon (in the return type in the Overview tab) changes to . You can also see the optimistic locking fields in the pragma statement at the top of the service's Source tab:

```
  (::pragma  xds <x:xds targetType="t:CREDITRATING" xmlns:x="urn:annotations.ld.bea.com" xmlns:t="ld:physical/
  CREDITRATING"> ... <optimisticLockingFields>   <field name="RATING"/> </optimisticLockingFields>
```

See Also

How To

- Test an Update Procedure

Concepts

- Brief Overview of Service Data Objects (for Studio)
- Data Programming Model and Update Framework (in depth, for client applications)

Other Resources

- Introducing SDO

Update Physical Data Service Metadata

This page last changed on Mar 28, 2008 .

How To Update Physical Data Service Metadata

When you first create a physical data service its underlying metadata is, by definition, consistent with its data source. Over time, however, your metadata may become "out of sync" for several reasons:

- The structure of underlying data sources may have changed, in which case it is important to be able to identify those changes so that you can determine when and if you need to update your metadata.
- You have modified schemas or added relationships to your data service.

In some cases relationships between data services will be preserved during metadata update. See Using the Update Source Metadata Wizard, for details.

Topics

In Project Explorer you can use the right-click menu option Update metadata to see if there are any differences between your source metadata files and the underlying source.

Update Metadata Option in Project Explorer

The Update metadata option can be used with:

- Relational table and view associated with changes to the relational database including providerID, the sourceBindingProviderClassName, columns, and optimistic locking fields.
- Web services
- Java functions
- Delimited files

Metadata update cannot be applied to data services based on:

- Relational stored procedures
- XML files

Scope of Metadata Update

When you run the Metadata update option, differences between your physical data service and the underlying data source are categorized according to the following scheme:

Category	Meaning
Objects added	The data source contains one or more objects that are not currently represented in the physical data service. From the perspective of the data source, information from the existing data service is added back after the metadata update. Another way to look at this is from the perspective of the data service. In this view, certain artifacts are *retained*. A typical example is a relationship with another data service. Existing relationships are identified and retained, the metadata is updated to reflect the current data source, and the relationships are added back to the data service.
Objects deleted	One or more objects in the physical data service is not found in the underlying data source. A typical artifact that will be marked for deletion would be a schema that is referenced by an operation (such as a relationship function) in the data service. Objects marked for delete generally appear together.  You should carefully inspect the update wizard for items marked for deletion. In the case of schemas, in particular, a prudent course of action would be to retain the schema (uncheck the delete option) unless you are certain that it is not needed by an operation in your data service. Deleting a needed schema will make your data service invalid and undeployable.

Objects changed	One or more objects in the physical data service and the underlying data source do not match and an adjustment will be made. An example of an artifact that will be marked as changed would be if the relational providerID underlying the data source has changed or is unavailable.
Source unavailable	The data source underlying the physical data service could not be accessed.

If there are no differences between your metadata and the underlying source, the Update metadata wizard will report up-to-date for each data service being verified.

In the case of an unavailable data source, the issue likely relates to connectivity or permissions. In the case of the other types of discrepancies, you need to determine when and if to update data source metadata to conform with the underlying data sources.

Important Considerations When Updating Source Metadata

The update metadata operation can have both direct and indirect consequences.

Source metadata should be updated with care by someone who is quite familiar with the underlying data source. For example, if you have added a relationship between two physical data services, updating your source metadata may remove the relationship from both data services. If the relationship appears in a model diagram, the relationship line will appear in red, indicating that the relationship is no longer described by the respective data services.

Direct and Indirect Effects

Direct effects apply to physical data services. Indirect effects occur to logical data services, since such services are themselves based — at least indirectly — on physical data services.

For example, if you have created a new relationship between a physical and a logical data service (not a recommended practice), updating the physical data service can invalidate the relationship. In the case of the physical data service, there will be no relationship reference. The logical data service will retain the code describing the relationship but it will be invalid if the opposite relationship notations is no longer be present.

Using the Update Source Metadata Wizard

The Update metadata wizard allows you to update your source metadata.

Before attempting to update source metadata you should make sure that your build project has no errors.

You can perform a metadata update on your entire dataspace project, folders from the project, or any qualified data service. Generally speaking, metadata updates should be performed as specifically as possible.

Use Shift-click or Ctrl-click to select multiple data services or folders in a single dataspace.

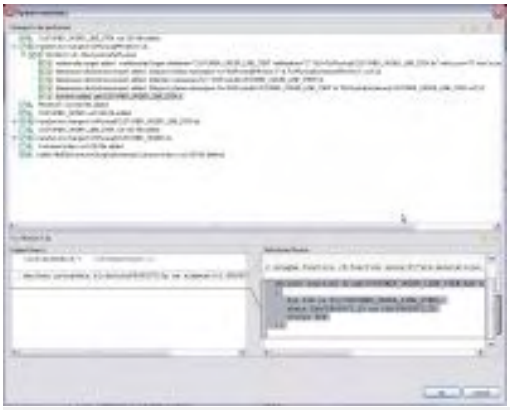
After you select your target(s), the wizard identifies the metadata that will be verified and any differences between your metadata and the underlying source.

You can select/deselect any data service listed in the dialog using the checkbox to the left of the name. You can also choose to select/deselect specific changes for the data service using the checkbox to the left of the change description.

The following screen capture is from a Update Metadata command for the ElectronicsWS project in the sample application.

Details related to the changes that will be made when the metadata is updated appear in the lower panel of the window.

Original Source and Refactored Source Details



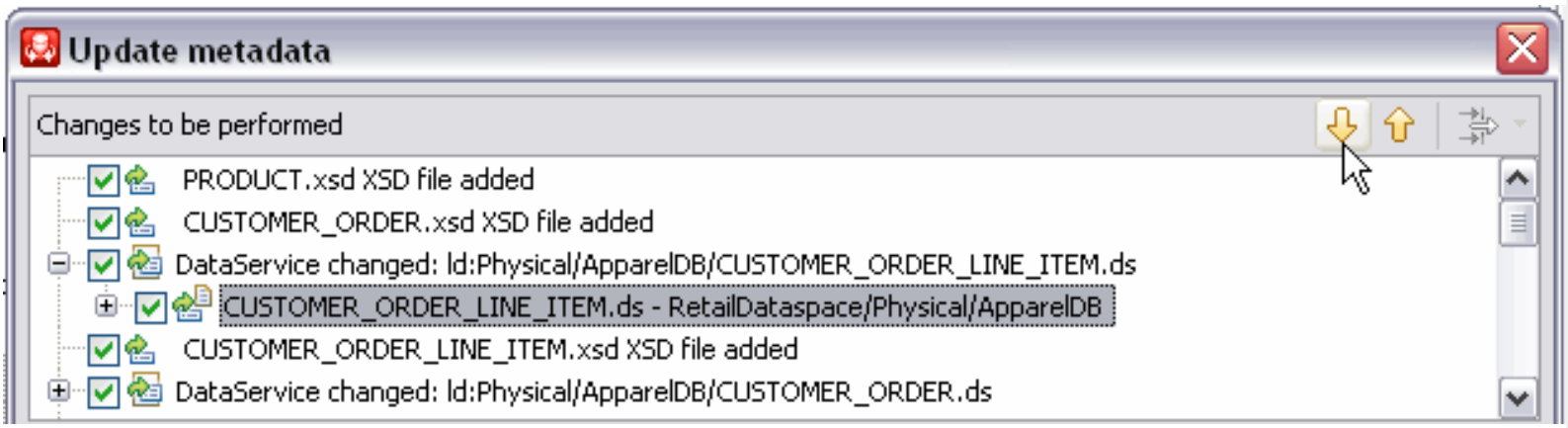
The upper portion of the Update metadata plan shows the changes to be performed. In some cases items are presented and selected (checked). In other cases items are presented but unchecked.

In the details view, the left-hand side shows the current source (called Original Source). The right-hand side shows what the result will be after metadata update (called Refactored Source).

Your only options in the dialog are to select or deselect specific changes using the adjacent checkboxes.

Up/down arrows are available on the Update Metadata titlebar to move through the possible changes. (The Filter Changes option icon next to the arrows is not applicable to metadata update and is not active.

Update Metadata Wizard Navigation Arrows



Inspecting and Reverting Changes Using Local History

You can use the Local History option provided with Eclipse to review changes that have been made through the Metadata Update Wizard.

Here are the steps involved:

1. In the Project Explorer right-click on your data service.
2. Select:

   Compare With > Local History...

   The Compare With Local History window will open. If there have been several changes made, each will be identified through a timestamp.

It is also often possible to revert a metadata update using a similar mechanism:

1. In the Project Explorer right-click on your data service.
2. Select:

   Replace With > Local History...

   The Replace With Local History window will open. If there have been several changes made, each will be identified through a timestamp.

   If you just want to revert to the immediate previous change, use the right-click option:

   Replace With > Previous from Local History...

How To Create SOAP Handlers for Imported WSDLs

This page last changed on Nov 27, 2007 .

Creating SOAP Handlers for Imported WSDLs

When you import metadata from web services for AquaLogic Data Services Platform, you can create SOAP handler for intercepting SOAP requests and responses. The handler will be invoked when a web service method is called. You can chain handlers that are invoked one after another in a specific sequence by defining the sequence in a configuration file.

To create and chain handlers, the following steps are involved:

1. Create a Java Class Implementing the Generic Handler Interface
2. Compile your intercept handler into a JAR file
3. Define a Configuration File
4. Define the Interceptor Configuration
5. Concluding Actions

Create a Java Class Implementing the Generic Handler Interface

The GenericHandler interface is:

```
javax.xml.rpc.handler.GenericHandler
```

Code Sample: Intercept Handler

The following code illustrates an example of implementing a generic handler.

Example:
Intercept Handler

For detailed information on how to write handlers, refer to Creating and Using Client-Side SOAP Message Handlers in Weblogic 9.2 documentation.

Compile your intercept handler into a JAR file.

The steps are to compile your intercept handler and JAR the class file.

Define a Configuration File

The configuration file specifies the handler chain and the order in which the handlers will be invoked.

Configuration File Schema

XML Schema for the Client-Side Handler Configuration File

The following is an example of the handler chain configuration. The handler-class attribute specifies the fully-qualified name of the
handler.

Code Sample: Handler Chain Configuration

```
<weblogic-wsee-clientHandlerChain
xmlns="http://www.bea.com/ns/weblogic/90"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:j2ee="http://java.sun.com/xml/ns/j2ee">
<handler>
        <j2ee:handler-name>sampleHandler</j2ee:handler-name>
        <j2ee:handler-class>WShandler.WShandler</j2ee:handler-class>
        <j2ee:init-param>
                <j2ee:param-name>ClientParam1</j2ee:param-name>
                <j2ee:param-value>value1</j2ee:param-value>
        </j2ee:init-param>
</handler>
</weblogic-wsee-clientHandlerChain>
```

Define the Interceptor Configuration

In your ALDSP application, define the interceptor configuration for the method in the data service to which you want to attach the handler.

Code Sample: Intercept Configuration

datasrvc:Intercept Configuration

In the attached file the aliasName attribute specifies the name of the handler chain to be invoked and the fileName attribute specifies the location of the configuration file.

Concluding Actions

- Place the JAR file that was based on the intercept handler (created above) in your project's dsp-inf/lib folder.
- Compile and run your application. Your handlers will be invoked in the order specified in the configuration file.

Attachments:

- Intercept Configuration.txt (text/plain)
- Intercept Handler.txt (text/plain)

Comments:

anil's 11/26 changes incorporated tk

Posted by tkatz at Nov 26, 2007 17:01

Creating XMLBean Support for Java Functions

This page last changed on Mar 24, 2008.

Creating XMLBean Support for Java Functions

Before you can create a Physical Data Service from Java functions, you need to create a `.class` file that contains XMLBean classes based on global elements and compiled versions of your Java functions. This topic describes how to create XMLBean classes based on a schema of your data.

Supported XMLBean Standards

Imported Java functions containing complex types must have a schema that conforms to one of the following XMLBean standards:

Version	URL
Apache	*<package name>*
BEA	apache: org.apache.xmlbeans BEA: com.bea.xml BEA: version

If your Java routines were compiled under previous versions, they will need to be recompiled before they can be imported.

The New Physical Data Service wizard requires that all the complex parameter or return types used by the functions correspond to XMLBean global element types whose content model is an anonymous type. Thus only functions referring to a top level element are imported.

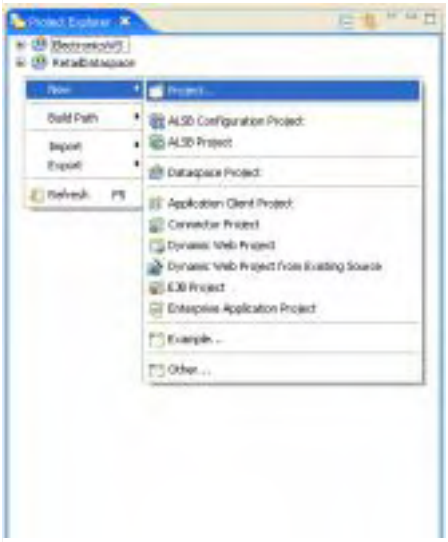
Creating XMLBean Classes for Java Functions

This topic describes how to create XMLBean classes based on a schema of your data.

Creating a New Project

You need to create a new project to build the XMLBean classes.

New Project



To create a new project:

1. Using WorkSpace Studio, create a new project by right-clicking in the Project Explorer and choosing New > Project in the menu. The New Project wizard is launched.
2. Choose Java > Java Project and click Next.

3. Type a name for the project and select Create separate source and output folders in the Project layout area.
4. Click Finish. WorkSpace Studio creates a new project in the Project Explorer.

New Java Project Wizard

Enabling XMLBeans Builder

You need to enable the XMLBeans Builder in the project to allow it to create classes based on the Java source and XML schema files.

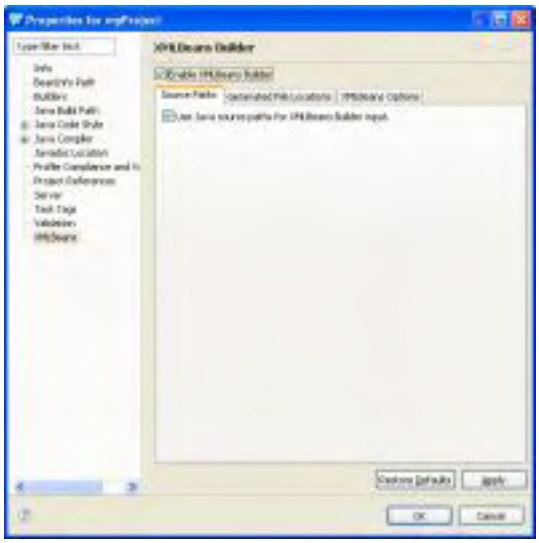
Project Properties

To enable XMLBeans Builder:

1. Right-click the new project, and choose Properties from the menu. The Properties dialog appears.

2. Click XMLBeans, select the Enable XMLBeans Builder checkbox, and click OK.

Enabling XMLBeans Builder



Importing Schema and Java Source Files

You need to import the schema files and Java source files into the project.

To import the schema and Java source files:

- Copy the schema files representing the data used by the Java functions along with the Java source files into the src folder.

Creating a Project Reference

The final step involves creating a reference to your XMLBeans-based project from the dataspace in which you want to use the Java functions.

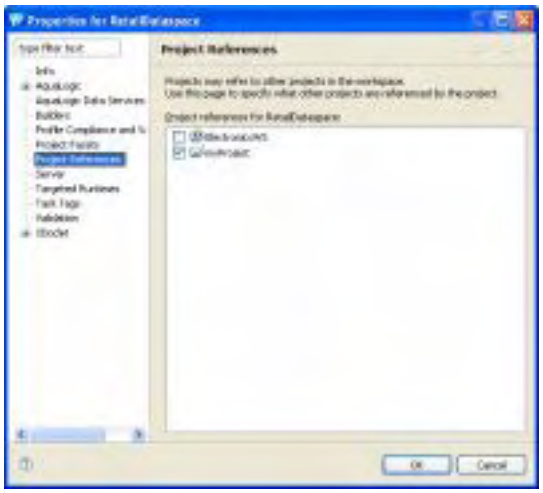
To create the project reference:

1. Right-click the dataspace project in which you want to use the Java functions and choose Properties from the menu.
2. Select Project References in the Properties dialog.
3. Select your XMLBeans-based project, and click OK.

When your project is deployed, ALDSP does the following:

- Rebuilds your XMLBeans-based project, if required, and generates a JAR file
- Copies the JAR file to the `DSP-INF/lib` folder in the dataspace project

Project References



See Also

How Tos

- Create a Physical Data Service from a Java Function

Reference

- Physical Data Services from Java Functions Overview
- XMLBeans Example Using a Metadata-rich Java Class

Preparing to Create Physical Data Services From Java Functions

This page last changed on Mar 25, 2008 .

Preparing to Create Physical Data Services From Java Functions

This topic provides an overview of how to create a new physical data service from Java functions.

Before you can create physical data services based on custom Java functions, you need to create a Java class containing both the schema and function information. The entire process involves the following:

1. Using Apache XMLBeans, BEA XMLBeans, or SDO DataObjects, create a schema of the data that is being used as parameters and return values by the Java functions.
2. Create the XMLBean classes or SDO DataObject classes and package them in a JAR file.

   For more information, see Creating XMLBean Support for Java Functions.

3. Place the JAR file in the `DSP-INF/lib` folder of the project in which you want to create the new Physical Data Service.
4. Create the new Physical Data Service based on your custom Java functions by importing the corresponding `.class` file.

   For more information, see Create a Physical Data Service from a Java Function.

See Also

Concepts

- Physical Data Services from Java Functions Overview

How Tos

- Create a Physical Data Service from a Java Function
- Creating XMLBean Support for Java Functions

Creating XMLBean Classes for Java Functions

This page last changed on Mar 27, 2008 .

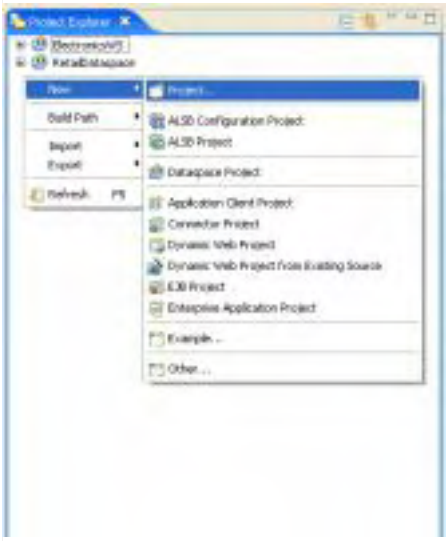
Creating XMLBean Classes for Java Functions

This topic describes how to create XMLBean classes based on a schema of your data.

Creating a New Project

You need to create a new project to build the XMLBean classes.

New Project



To create a new project:

1. Using WorkSpace Studio, create a new project by right-clicking in the Project Explorer and choosing New > Project in the menu. The New Project wizard is launched.
2. Choose Java > Java Project and click Next.
3. Type a name for the project and select Create separate source and output folders in the Project layout area.

4. Click Finish. WorkSpace Studio creates a new project in the Project Explorer.

New Java Project Wizard

Enabling XMLBeans Builder

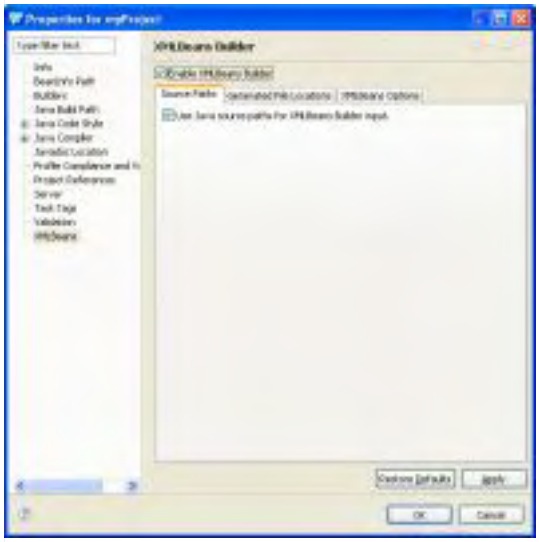
You need to enable the XMLBeans Builder in the project to allow it to create classes based on the Java source and XML schema files.

Project Properties

To enable XMLBeans Builder:

1. Right-click the new project, and choose Properties from the menu. The Properties dialog appears.
2. Click XMLBeans, select the Enable XMLBeans Builder checkbox, and click OK.

Enabling XMLBeans Builder



Importing Schema and Java Source Files

You need to import the schema files and Java source files into the project.

To import the schema and Java source files:

- Copy the schema files representing the data used by the Java functions along with the Java source files into the src folder.

Creating a Project Reference

The final step involves creating a reference to your XMLBeans-based project from the dataspace in which you want to use the Java functions.

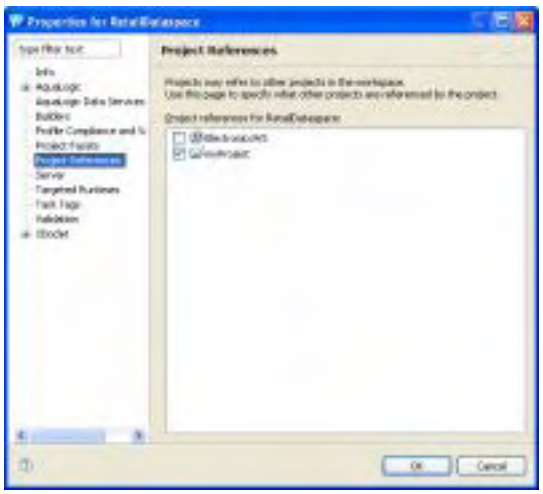
To create the project reference:

1. Right-click the dataspace project in which you want to use the Java functions and choose Properties from the menu.
2. Select Project References in the Properties dialog.
3. Select your XMLBeans-based project, and click OK.

When your project is deployed, ALDSP does the following:

- Rebuilds your XMLBeans-based project, if required, and generates a JAR file
- Copies the JAR file to the `DSP-INF/lib` folder in the dataspace project

Project References



Physical Data Service from a Java Function - Example Code

This page last changed on Mar 24, 2008.

Physical Data Service from a Java Function - Example Code

This topic provides examples showing the use of imported Java functions in an XQuery and the processing of complex types.

Using a Function Returning an Array of Java Primitives

As an example, the Java function `getRunningTotal` can be defined as follows:

```
public static float[] getRunningTotal(float[] list) {
   if (null == list || 1 >= list.length)
      return list;
   for (int i = 1; i < list.length; i++) {
      list[i] = list[i-1] + list[i];
   }
   return list;
}
```

The corresponding XQuery for executing the above function is as follows:

```
Declare namespace f1="ld:javaFunc/float"
Let $y := (2.0, 4.0, 6.0, 8.0, 10.0)
Let $x := f1:getRunningTotal($y)
Return $x
```

The results of the query is as follows:

```
2.0, 6.0, 12.0, 20.0, 30.0
```

Processing complex types represented via XMLBeans

Consider a schema called Customer (customer.xsd), as shown in the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
   <xs:schema targetNamespace="ld:xml/cust:/BEA_BB10000" xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="CUSTOMER">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="FIRST_NAME" type="xs:string" minOccurs="1"/>
            <xs:element name="LAST_NAME" type="xs:string" minOccurs="1"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

You could compile the schema using XMLBeans to generate a Java class corresponding to the types in the schema.

```
xml.cust.beaBB10000.CUSTOMERDocument.CUSTOMER
```

For more information, see http://xmlbeans.apache.org.

Following this, you can use the CUSTOMER element as shown in the following:

```
public static xml.cust.beaBB10000.CUSTOMERDocument.CUSTOMER[]
      getCustomerListGivenCustomerList(xml.cust.beaBB10000.CUSTOMERDocument.CUSTOMER[] ipListOfCust)
      throws XmlException {
   xml.cust.beaBB10000.CUSTOMERDocument.CUSTOMER [] mylocalver = pListOfCust;
   return mylocalver;
}
```

The resulting metadata information produced by the New Physical Data Service wizard will be:

```
(::pragma function <f:function xmlns:f="urn:annotations.ld.bea.com" kind="datasource" access="public">
<params>
<param nativeType="[Lxml.cust.beaBB10000.CUSTOMERDocument$CUSTOMER;"/>
</params>
</f:function>::)

declare function f1:getCustomerListGivenCustomerList($x1 as element(t1:CUSTOMER)*) as element(t1:CUSTOMER)*
external;
```

The corresponding XQuery for executing the above function is:

```
declare namespace f1 = "ld:javaFunc/CUSTOMER";
let $z := (
validate(<n:CUSTOMER xmlns:n="ld:xml/cust:/BEA_BB10000"><FIRST_NAME>John2</FIRST_NAME><LAST_NAME>Smith2</
LAST_NAME>
</n:CUSTOMER>),

validate(<n:CUSTOMER xmlns:n="ld:xml/cust:/BEA_BB10000"><FIRST_NAME>John2</FIRST_NAME><LAST_NAME>Smith2</
LAST_NAME>
</n:CUSTOMER>),

validate(<n:CUSTOMER xmlns:n="ld:xml/cust:/BEA_BB10000"><FIRST_NAME>John2</FIRST_NAME><LAST_NAME>Smith2</
LAST_NAME>
</n:CUSTOMER>),

validate(<n:CUSTOMER xmlns:n="ld:xml/cust:/BEA_BB10000"><FIRST_NAME>John2</FIRST_NAME><LAST_NAME>Smith2</
LAST_NAME>
</n:CUSTOMER>))

for $zz in $z
return
```

Stored Procedure Configuration Reference

This page last changed on Feb 26, 2008 .

Stored Procedure Configuration Reference

The following topics provide detailed information regarding various configuration options associated with creating data services based on stored procedures.

In Mode, Out Mode, Inout Mode

In, Out, and Inout mode settings determine how a parameter passed to a stored procedure is handled.

Parameter Mode	Effect
In	Parameter is passed by reference or value.
Inout	Parameter is passed by reference.
Out	Parameter is passed by reference. However the parameter being passed is first initialized to a default value. If your stored procedure has an OUT parameter requiring a complex element, you may need to provide a schema.

Procedure Profile

Each element in a stored procedure is associated with a type. If the item is a simple type, you can simply choose from the pop-up list of types. If the type is complex, you may need to supply an appropriate schema. Click on the schema location button and either enter a schema pathname or browse to a schema. The schema must reside in your application.

After selecting a schema, both the path to the schema file and the URI appear.

Complex Parameter Types

Complex parameter types are supported under only three conditions:

- As the output parameter
- As the Return type
- As a rowset

About Rowsets

A rowset type is a complex type.

The rowset type contains a sequence of a repeatable elements (for example called CUSTOMER) with the fields of the rowset.

In some cases the wizard can automatically detect the structure of a rowset and create an element structure. However, if the structure is unknown, you will need to provide it.

All rowset-type definitions must conform to this structure.

The name of the rowset type can be:

- The parameter name (in case of a input/output or output only parameter).
- An assigned name.
- The referenced element name (result rowsets) in a user-specified schema.

Not all databases support rowsets. In addition, JDBC does not report information related to defined rowsets.

Using Rowset Information

In order to create data services from stored procedures that use rowset information, you need to supply the correct ordinal (matching number) and a schema. If the schema has multiple global elements, select the one you want from the Type column. Otherwise the type used match the first global element in your schema file.

The order of rowset information is significant; it must match the order in your data source. Use the Move Up / Move Down commands to adjust the ordinal number assigned to the rowset.

XML types in data services generated from stored procedures do not display native types. However, you can view the native type in the Source editor; it is located in the pragma section.

Stored Procedure Version Support

Only the most recent version of a particular stored procedure can be imported into ALDSP. For this reason you cannot identify a stored procedure version number when creating a physical data service based on a stored procedure. Similarly, adding a version number for your stored procedure in the Source editor will result in a query exception.

Supporting Stored Procedures with Nullable Input Parameter(s)

If you know that an input parameter of a stored procedure is nullable (can accept null values), you can change the signature of the function in Source View to make such parameters optional by adding a question mark at end of the parameter.

For example (question-mark (**?**) shown in bold):

```
function myProc($arg1 as xs:string) ...
```

would become:

```
function myProc($arg1 as xs:string?) ...
```

Simple Java Types and Their XQuery Counterparts

This page last changed on Mar 24, 2008.

Simple Java Types and Their XQuery Counterparts

The following outlines the mapping between simple Java types and the corresponding XQuery or schema types:

Java Simple or Defined Type	XQuery/Schema Type
boolean	xs:boolean
byte	xs:byte
char	xs:char
double	xs:double
float	xs:float
int	xs:int
long	xs:long
short	xs:short
string	xd:string
java.lang.Date	xs:datetime
java.lang.Boolean	xs:boolean
java.math.BigInteger	xs:integer
java.math.BigDecimal	xs:decimal
java.lang.Byte	xs.byte
java.lang.Char	xs:char
java.lang.Double	xs:double
java.lang.Float	xs:float
java.lang.Integer	xs:integer
java.lang.Long	xs:long
java.lang.Short	xs:short
java.sql.Date	xs:date
java.sql.Time	xs:time

java.sql.Timestamp	xs:datetime
java.util.Calendar	xs:datetime

Java functions can consume parameters and return values of the following types:

- Java primitives and types listed in the previous table
- Apache XMLBeans
- BEA XMLBeans
- SDO DataObject (typed or untyped)

The elements or types referred to in the schema should be global elements.

Designing Logical Data Services

This page last changed on Feb 26, 2008.

Designing Logical Data Services

Concepts

- Building Logical Entity Data Services
- Data Service Keys
- XML Types and Return Types

How-to...

- Add a Read Function
- Add a Library Function or Procedure
- Create Logical Data Service Keys
- Declare a Security Resource in Studio

Examples

- Create a Logical Data Service with a Group By Clause
- Create a Data Service with a Flat Return Type

Reference

- XQuery Source of a Logical Entity Service

Related Topics

How-to...

- Create Your First Data Services
- Create a Return Type
- Add a Complex Child Element to a Return Type
- Check Namespaces in Return Types
- Create Conditional Elements in Return Types
- Test a Read Function and Simple Update
- Test a Create or Delete Procedure

Concepts

- Data Service Types and Functions

Building Logical Entity Data Services

This page last changed on Mar 11, 2008.

This topic introduces you to logical entity data services.

The Benefits of Logical Services

The benefit of data services is the ability to combine multiple data sources of different types into service-oriented architectures. Enterprise data is often stored in relational databases, non-relational databases, packaged applications (such as SAP, PeopleSoft, Siebel, and others), custom applications, or files of various types. You might also be accessing data from web services.

The goal is to create a new loosely coupled architecture by piecing together the data assets you already have. In a practical sense, this means combining data from relational data sources, web services, XML files, other files, or Java functions. Logical data services are of two types, entity and library.

Logical entity services allow you to design, model, and create a data view from many underlying data sources. Logical library services are simply a collection of related functions and procedures within a data service container. This topic introduces logical entity services.

On a tangible level, a logical entity service is an XQuery source file with functions and procedures that act on data. A logical entity service has:

- Exactly one XML schema that represents the data the service returns (its return type).
- Any number of create, update, or delete procedures, where up to one of each type is primary.
- Any number of library functions and procedures.
- Any number of relationships with other entity services.

In addition, a logical entity service must have a primary read function if you want the service to have an update map.

Design View

Logical data services have their foundation in XML web services. The backbone of a logical data service is its return type, which is a combination of data you design expressed as an XML schema.

You can see the return type in the Overview tab in Studio.

Design View of a Logical Data Service

The logical service's return type is shown in the center. You can right-click it to see the XML schema source. On the left, you see the defined for the service. On the right, you see other data services that underly the logical data service.

ALDSP Functions and Procedures

The underlying data services can be physical or logical.

The beauty of a logical data service is that a return type is a model. Logical models capture the complexity of data integration once, and allow you to write clients that remain the same even when underlying physical data sources change.

The structure of a return type does not need to match the structure of the underlying data sources. Here, the CUSTOMER element has a 1-to-many relationship with its child element ADDRESS, and a 1-to-1 relationship with its other child element, CREDITRATING. Each complex element represents a separate physical data source.

The Return Type Schema

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema targetNamespace="ld:logical/CustomerProfile" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="CUSTOMER_PROFILE">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="CUSTOMER">
          <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="EMAIL_ADDRESS" type="xs:string"/>
                <xs:element name="ADDRESS" maxOccurs="unbounded">
                  <xs:complexType>
                    <xs:sequence>
                        <xs:element name="ADDR_ID" type="xs:string"/>
                        <xs:element name="CUSTOMER_ID" type="xs:string"/>
                        <xs:element name="STREET_ADDRESS1" type="xs:string"/>
                        <xs:element name="STREET_ADDRESS2" type="xs:string" minOccurs="0"/>
                        <xs:element name="CITY" type="xs:string"/>
                        <xs:element name="STATE" type="xs:string"/>
                        <xs:element name="ZIPCODE" type="xs:string"/>
                        <xs:element name="COUNTRY" type="xs:string"/>
                    </xs:sequence>
                  </xs:complexType>
                </xs:element>
                <xs:element name="CREDITRATING" maxOccurs="1">
                  <xs:complexType>
                    <xs:sequence>
                        <xs:element name="CUSTOMER_ID" type="xs:string"/>
```

```
                        <xs:element name="RATING" type="xs:int" minOccurs="0"/>
                    </xs:sequence>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>
```

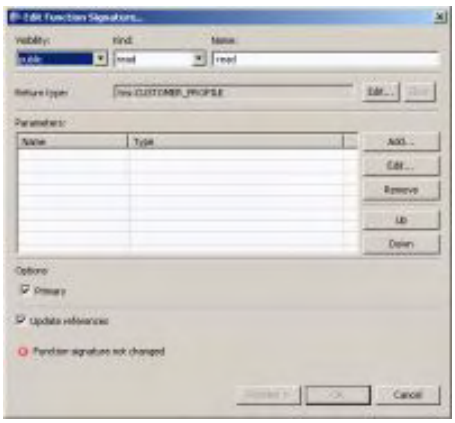
However, this structure is only by design. You could also have designed the return type with fewer elements, or in a flat structure, depending on how you want the service to return data.

The Primary Read Function

The functions and procedures in a logical entity service are implemented in XQuery, which queries XML data much as SQL queries relational data. You can get information about any function or procedure by right-clicking it in the Overview tab.

A read function, for example, often takes no parameters and returns an instance of the return type.

Viewing the Signature of a Read Function



In a logical entity service, you can designate one read function as primary. A primary read function captures the main data integration logic in the service. ALDSP generates the create, update, and delete procedures and the update map from the primary read function.

You can see the source code of the primary read function in the Source tab.

Checking the Primary Read Function Source

```
declare function tns:read() as element(tns:CUSTOMER_PROFILE)*{
for $CUSTOMER in cus1:CUSTOMER()
return
        <tns:CUSTOMER_PROFILE>
            <CUSTOMER>
                <CUSTOMER_ID>{fn:data($CUSTOMER/CUSTOMER_ID)}</CUSTOMER_ID>
                <FIRST_NAME>{fn:data($CUSTOMER/FIRST_NAME)}</FIRST_NAME>
                <LAST_NAME>{fn:data($CUSTOMER/LAST_NAME)}</LAST_NAME>
                <EMAIL_ADDRESS>{fn:data($CUSTOMER/EMAIL_ADDRESS)}</EMAIL_ADDRESS>
```

```
                {
                    for $ADDRESS in add:ADDRESS()
                    where $CUSTOMER/CUSTOMER_ID eq $ADDRESS/CUSTOMER_ID
                    return
                    <ADDRESS>
                        <ADDR_ID>{fn:data($ADDRESS/ADDR_ID)}</ADDR_ID>
                        <CUSTOMER_ID>{fn:data($ADDRESS/CUSTOMER_ID)}</CUSTOMER_ID>
                        <STREET_ADDRESS1>{fn:data($ADDRESS/STREET_ADDRESS1)}</STREET_ADDRESS1>
                        <STREET_ADDRESS2?>{fn:data($ADDRESS/STREET_ADDRESS2)}</STREET_ADDRESS2>
                        <CITY>{fn:data($ADDRESS/CITY)}</CITY>
                        <STATE>{fn:data($ADDRESS/STATE)}</STATE>
                        <ZIPCODE>{fn:data($ADDRESS/ZIPCODE)}</ZIPCODE>
                        <COUNTRY>{fn:data($ADDRESS/COUNTRY)}</COUNTRY>
                    </ADDRESS>
                }
                {
                        for $CREDITRATING in cre:CREDITRATING()
                        where $CUSTOMER/CUSTOMER_ID eq $CREDITRATING/CUSTOMER_ID
                        return
                        <CREDITRATING>
                           <CUSTOMER_ID>{fn:data($CREDITRATING/CUSTOMER_ID)}</CUSTOMER_ID>
                           <RATING?>{fn:data($CREDITRATING/RATING)}</RATING>
                        </CREDITRATING>
                }
            </CUSTOMER>
        </tns:CUSTOMER_PROFILE>
};
```

This read function returns a CUSTOMER_PROFILE element with a nested CUSTOMER element. Each CUSTOMER element has some number of ADDRESS elements and some number of CREDITRATING elements, where the CUSTOMER_ID in ADDRESS or CREDITRATING matches the CUSTOMER_ID in CUSTOMER. (The XQuery where clauses create table joins; see Add a Where Clause to a Query).

Create, Update, and Delete Procedures

A logical entity service also typically has create, update, and delete procedures that act on underlying data sources. (The difference between a function and a procedure is that a procedure can have side effects, while a function cannot; see Data Service Types and Functions).

The CustomerProfile service has one create procedure, one update procedure, and two delete procedures. It also has a library procedure named stringToShort, which casts between two data types.

Viewing Functions and Procedures

Query Map View

The Query Map view maps elements in data sources to the return type.

Mapping Data Sources to the Return Type

You can see the data sources on the left and the return type on the right. The blue lines map elements from the data sources to elements in the return type, showing how the return type receives data.

The green dashed lines between the data source blocks create joins, which become where clauses in the XQuery source, for example:

```
for $ADDRESS in add:ADDRESS()
where $CUSTOMER/CUSTOMER_ID eq $ADDRESS/CUSTOMER_ID
return
```

If you click a data element (not a container element) in the return type, you see its XQuery expression in the expression editor.

Mapping Data in an XQuery Expression

Notice that the mapping expressions use the built-in XQuery function fn:data, which extracts the data value from an XML element.

As you map elements visually in the Query Map, ALDSP creates XQuery source (for example, the read function shown above). The XQuery source is later converted to SQL queries, which you can see in Plan view.

Viewing a SQL Query in Plan View

In this query plan, you see the left outer join between the CUSTOMER and ADDRESS relational tables. This was created by the

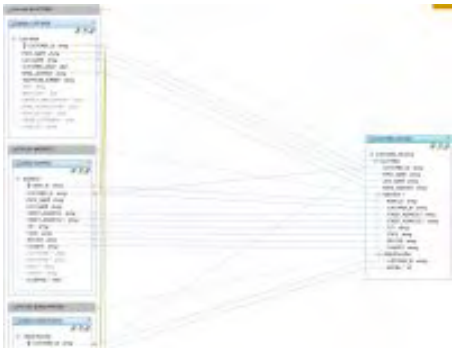
green dashed line drawn between the Customer and Address blocks in Query Map view.

When you build XQuery functions and procedures visually in Query Map view or by editing in Source view, you can test and run them on an ALDSP server. During server runtime, the functions and procedures are compiled into an executable query plan. Examine the query plan before you finalize the queries. Query Plan view gives you a peek into a query's execution logic and flags potential performance and memory problems. Building XQuery functions is an iterative process of test, view plan, and edit.

Update Map View

While Query Map view shows how a service reads from data sources, Update Map view shows how the service writes data to them.

Checking Update Map View



The data sources are on the left, with updates coming from the return type on the right. The return type is available to client applications, where users update data.

The blocks on the left are update blocks. Each mapped element in an update block has an XQuery expression that defines how the element is updated. You can see the expression in the expression editor below the mapping area.

Viewing an XQuery Update Expression

ALDSP generates the update map for you when you create a logical data service under these conditions:

- Your service has a primary read function
- You are using relational data sources

(If you are using other data source types, you must edit the update template.)

You can then customize the update map and test it in Test view, without programming.

An application client uses the Service Data Objects programming model to update data sources. SDO is an application framework that allows you to update data sources while disconnected from them, using a flexible, optimistic concurrency

model. You use only one API, the SDO API, to update multiple data sources -- relational, web service, XML files, and so on.

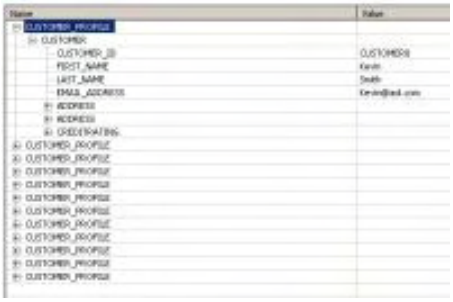
Test View

The Test view available in Studio works like a built-in client where you can easily test any function or procedure in the data service, before you build a custom client.

Choosing a Function or Procedure to Test

Testing a read function, for example, returns data as the service would to a client, in the shape of the return type.

Reading Customer Profile Data



To test a simple update, click the Edit button, edit some data in the result, then click Submit. When you test the read function again, the results show the change.

You can also test an SDO update by submitting a datagraph with a change summary (see Test an Update Procedure).

See Also

Concepts

- Understanding Update Maps

How Tos

- Create a Return Type
- Create Your First Data Services
- Add a Where Clause to a Query

Reference

- XQuery Source of a Logical Entity Service

Other Sources

- Introdution to Service Data Objects (ibm.com)

Data Service Keys

This page last changed on Mar 11, 2008.

This topic describes what data service keys are and how they are used.

Overview

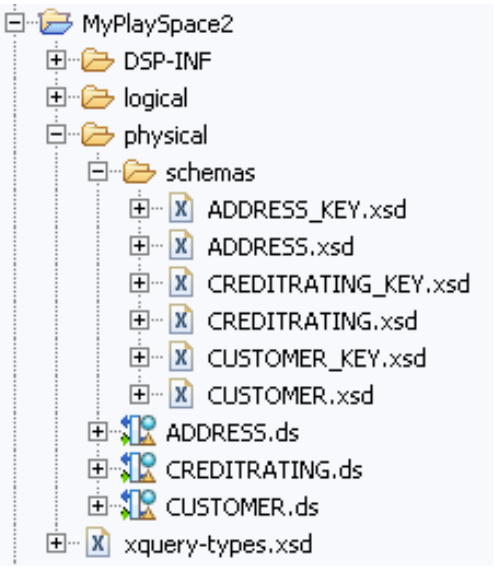
You are probably familiar with the concept of keys from relational databases, where a key is a set of one or more columns whose combined values are unique among all occurrences in a table.

When you create a physical data service, ALDSP computes keys by introspecting the physical data sources. A physical data service key can have one or more fields, which are elements taken from the service's return type. Tangibly, a key is defined as an XML schema in an XSD file.

You can see the physical data service keys in your dataspace project in Studio. They appear in schema files with names such as:

```
datasource_KEY.xsd
```

Physical Data Service Keys in Studio



In the generated XSD file, a key for a physical data service looks something like this.

Key for the CUSTOMER Table

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema targetNamespace="ld:physical/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:string"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

In this case, CUSTOMER_ID is the primary key in a relational table named CUSTOMER.

In a logical data service, a key also uniquely defines a data record. However, the data in the record can originate from multiple data sources of different types and can have a structure unlike the underlying physical data sources.

For a logical entity service, you must create the key. You can choose one of these options:

- Have ALDSP generate the key based on the service's primary read function. ALDSP generates a minimal key.
- Select the fields that make up the key. The elements that comprise the key must have a cardinality of 0 or 1 in the service's return type (with maxOccurs="1" or maxOccurs="0", but not maxOccurs="unbounded").

Parts of a Key

Suppose a logical service has a nested return type where a parent element with single cardinality can have multiple child elements, say one CUSTOMER element with many CUSTOMER_ORDER child elements.

A Nested Return Type with a One-to-Many Relationship

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema targetNamespace="ld:logical/CustomersAndOrders" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="CustomersAndOrders">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="CUSTOMER">
          <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="SSN" type="xs:string" minOccurs="0"/>
              <xs:element name="CUSTOMER_ORDER" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="ORDER_ID" type="xs:string"/>
                    <xs:element name="C_ID" type="xs:string"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
```

```
    </xs:element>
  </xs:schema>
```

This is the key that ALDSP auto-generates from this return type, from the unique CUSTOMER_ID field:

An Auto-Generated Simple Key

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

If you choose to select the key fields, you need to use a unique field or fields with single cardinality. You can choose CUSTOMER_ID or SSN, or both. You cannot define the key on ORDER_ID or C_ID, because they belong to the CUSTOMER_ORDER element, which has multiple cardinality.

If you choose SSN, the key schema file looks like this.

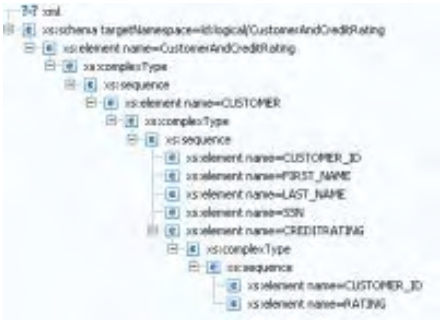
A Manually Selected Key

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="ld:logical/CustomerOrder" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="CustomersAndOrders_KEY">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="SSN" maxOccurs="1" minOccurs="0" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

A data service key has distinct parts:

- **A selector.** A key selector identifies a collection of data records. A key's selector is the element that contains the key field in the service's return type. You can see a key's selector in the Manage Key dialog when you create the key (below, it's the CUSTOMER element):

You can see that the CUSTOMER element is the root element of the return type:



- **The key fields.** The fields that make up the key uniquely identify an element in the collection. For example, one customer identified by a CUSTOMER_ID value. Within ALDSP, a key field is stored as a path which must not contain any repeating elements. Therefore, you cannot use elements with multiple cardinality in keys.

Composite Keys

With a logical service, a key can also be a composite key of multiple elements, as long as the elements have single cardinality in the return type. This is especially easy with a flat return type.

A Flat, Non-Nested Return Type

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema targetNamespace="ld:logical/MyFlatOne" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="CUSTOMERORDER">
    <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="EMAIL_ADDRESS" type="xs:string"/>
        <xs:element name="ORDER_ID" type="xs:string"/>
        <xs:element name="ORDER_DT" type="xs:date"/>
        <xs:element name="TOTAL_ORDER_AMT" type="xs:decimal"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

ALDSP auto-generates a composite key using the key fields from the underlying physical data sources (in this example, CUSTOMER_ID and ORDER_ID). The composite key generated from this return type is shown below.

An Auto-Generated Composite Key

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="ld:logical/MyFlatOne" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="MyFlatOne_KEY">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="CUSTOMER_ID" type="xs:string"/>
        <xs:element name="ORDER_ID" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

This key allows you to identify a unique combination of Customer and Order, that is, one order for one customer.

See Also

How Tos

- Create Logical Data Service Keys

XML Types and Return Types

This page last changed on Feb 26, 2008.

In entity data services there are two types of types:

- Return types
- XML types

XML types and return types are very closely related. In data service operations involving entity data services, XML types define the shape of the data service.

Physically XML Types are represented a global elements in XML schemas (XSD files.) In other words, the XML types represents in hierarchical form the shape of the data service.

A way to think of these two artifacts is to first consider the class and the instance of the class in such languages as Java.

XML types can be thought of as a class from which objects in the form of functions are created. In many cases the information needed by these functions is either:

- A subset of the overall XML types -- for example, a function that returns last name and address but not first name or social security number.
- In need of further specification -- for example, adjusting a query to list all orders inside each customer rather than to repeat customer information each time.

Return and XML types can be see in action in the following example: Creating Your First Data Services

Where XML Types are Used

ALDSP uses XML types in its model diagrams, entity data services, query editor, update mapper, and metadata browser.

Where Return Types are Used

Return types are sometimes called *target schemas*.

Return types can be thought of as the backbone of both data services and data models. Programmatically, return types are the "r" in FLWR (for-let-where-return) queries.

Return types have the following main purposes:

- Provide a template for the mapping of data from a variety of data sources and, in the case of updates, back to those data sources.
- Help determine the arrangement of the XML document generated by the XQuery.

Return types describes the structure or shape of data that a query produces when it is run.

In order to maintain the integrity of AquaLogic Data Services Platform queries used by your application, it is important that the query return type match the XML type in the containing data service. Thus if you make changes in the return type, you should use the XQuery Editor's "Save and associate schema" command to make the data service's XML type consistent with query-level changes. Alternatively, create a new data service based on your return type. For details see Creating a Simple Data Service Function.

Add a Read Function

This page last changed on Mar 11, 2008.

This topic describes how to add a read function to a logical entity service.

Overview

A read function in a logical entity service retrieves data from underlying data sources, either physical or logical, and returns XML elements in the shape of the service's return type. You can build a logical service without a read function. However, the service must have at least one read function, marked primary, to have an update map. Only one read function in a service can be primary.

A read function is associated with exactly one XML schema, which is the service's return type. The read function must return the return type, but cannot take any other actions or have any side effects.

When you create a primary read function visually in Studio, ALDSP generates a pragma annotation and XQuery source. The pragma looks something like this:

```
(::pragma  function <f:function kind="read" visibility="public" isPrimary="true" xmlns:f="urn:annotations.ld.
bea.com"/>::)
```

The initial XQuery source, before you map data types in Query Map view, shows that the read function returns an instance of the service's return type:

```
declare function tns:read() as element(tns:CustomerAndAddress)*{
    <tns:CustomerAndAddress>
        <CUSTOMER>
            <CUSTOMER_ID></CUSTOMER_ID>
            <FIRST_NAME></FIRST_NAME>
            <LAST_NAME></LAST_NAME>
            <SSN?></SSN>
            {
                <ADDRESS>
                    <ADDR_ID></ADDR_ID>
                    <CUSTOMER_ID></CUSTOMER_ID>
                    <FIRST_NAME></FIRST_NAME>
                    <ZIPCODE></ZIPCODE>
                    <COUNTRY></COUNTRY>
                </ADDRESS>
            }
        </CUSTOMER>
    </tns:CustomerAndAddress>
};
```

At this point, the return type has no values. The values are added after you map data sources to the return type in Query Map view:

```
declare function tns:read() as element(tns:CustomerAndAddress)*{
for $CUSTOMER in cus1:CUSTOMER()
return
        <tns:CustomerAndAddress>
            <CUSTOMER>
```

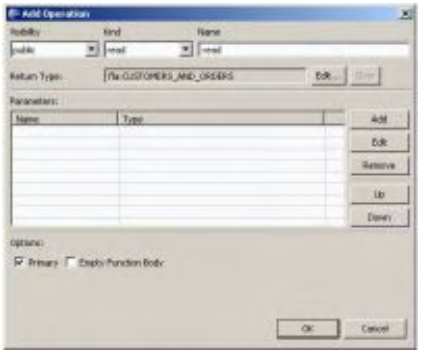
```
                <CUSTOMER_ID>{fn:data($CUSTOMER/CUSTOMER_ID)}</CUSTOMER_ID>
                <FIRST_NAME>{fn:data($CUSTOMER/FIRST_NAME)}</FIRST_NAME>
                <LAST_NAME>{fn:data($CUSTOMER/LAST_NAME)}</LAST_NAME>
                <SSN?>{fn:data($CUSTOMER/SSN)}</SSN>
                {
                    for $ADDRESS in add:ADDRESS()
                    return
                    <ADDRESS>
                        <ADDR_ID>{fn:data($ADDRESS/ADDR_ID)}</ADDR_ID>
                        <CUSTOMER_ID>{fn:data($ADDRESS/CUSTOMER_ID)}</CUSTOMER_ID>
                        <FIRST_NAME>{fn:data($ADDRESS/FIRST_NAME)}</FIRST_NAME>
                        <ZIPCODE>{fn:data($ADDRESS/ZIPCODE)}</ZIPCODE>
                        <COUNTRY>{fn:data($ADDRESS/COUNTRY)}</COUNTRY>
                    </ADDRESS>
                }
            </CUSTOMER>
        </tns:CustomerAndAddress>
};
```

Create the Function in Studio

1. Create a logical entity service.

   Create Your First Data Services

2. In the Overview tab, right-click at the left, right, or top, and choose Add Operation.



3. At Visibility, choose an access level.
   Public means the procedure can be called from the same dataspace and from client APIs; protected, only from the same dataspace; private, only from the same data service.
4. At Kind, choose read.
5. Enter a name for the function.
6. At Return Type, click Edit.
7. Click Complex Type, and choose a schema file.
8. At Kind, choose element.
9. At Occurrence, choose Zero or More.
10. Select Primary, and click OK.

See Also

How Tos

- Create a Return Type
- Test a Read Function and Simple Update

Concepts

- Data Service Types and Functions

Add a Library Function or Procedure

This page last changed on Mar 11, 2008.

This topic describes how to add a library function or procedure to a data service.

Overview

Library functions and procedures are utility operations that you can add to any service, physical, logical, or library. Library functions and procedures:

- Have a kind of library
- Are not marked as primary or non-primary
- Have a visibility of public, protected or private

A library function can return values, but has no side effects. A library procedure can return values and can have side effects.

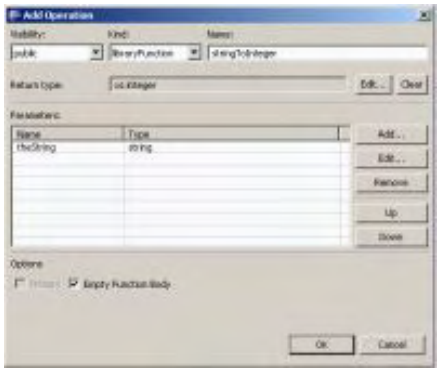
You can declare a library function or procedure visually in Studio, but you must still write the function body in the Source tab using XQuery. Alternatively, you can write the entire function or procedure and its pragma statement in XQuery.

You can call a library function or procedure from the service, the dataspace project, or from a client application, depending on the visibility level you set.

Add the Function or Procedure

The example in this section is a library function that casts a value from xs:integer to xs:string.

1. Open the service and click the Overview tab.
2. Right-click at the left, right, or top, and choose Add Operation.
3. Select a value at Visibility (public = call from anywhere; protected = from the same dataspace; private = from the same data service).
4. At Kind, choose libraryFunction or libraryProcedure.



5. Give your function or procedure a name.
6. At Return Type, click Edit and choose a simple or complex return type. Click OK.
7. At Parameters, click Edit. Enter a parameter name, and choose a simple or complex return type. Click OK.
8. Click Empty Function Body, then OK.
9. Click the Source tab.

   ALDSP has generated a pragma statement and an empty function or procedure body, like this:

```
(::pragma  function <f:function kind="library" visibility="public" isPrimary="false" xmlns:f="urn:
annotations.ld.bea.com"/>::)

declare function cus2:integerToString($theInt as xs:positiveInteger) as xs:string* {
    $var-bea:tbd
};
```

10. In the function body, delete $var-bea:tbd and add your own XQuery code, for example:

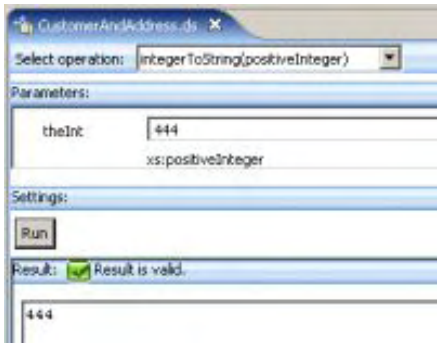
```
declare function cus2:integerToString($theInt as xs:positiveInteger) as xs:string* {
                xs:string($theInt)
};
```

Test in Studio

You can test the library function or procedure directly in Studio, before you use it from a client application.

1. Open the service, and click the Test tab.
2. At Select Operation, choose the library function or procedure you want to test.
3. Enter a value in the Parameters box.
4. (Optional) Expand Settings and enter new values for results, transactions, and authentication.
5. Click Run.

   If the function or procedure works, you see valid results.



   If not, you see an exception message that provides details, so that you can correct the error.

How Tos

- Add a Read Function
- Add Update Map Procedures

Reference

- Data Service Types and Functions

Create Logical Data Service Keys

This page last changed on Mar 11, 2008.

This topic describes how to create a key for a logical data service.

Overview

A logical data service key uniquely identifies a data record the logical service defines. Because a logical service combines data from various physical and logical services, its key can combine or be different from the keys defined on underlying data sources.

For example, you might have a logical data service with a flat return type that combines data from two relational tables, CUSTOMER and ORDER. These tables have keys CUSTOMER_ID and ORDER_ID, respectively. In your logical data service, each data record is a unique combination of Customer and Order, so you create a composite key that combines CUSTOMER_ID and ORDER_ID.

Create procedures return a key to identify the data record that was inserted. Update and Delete procedures act on the data record the key identifies. A logical data service can have one key, although you can have multiple key schema files from which you select the key. You can have ALDSP auto-generate the key, choose the elements you want in the key, or select an available schema (XSD) file to use for the key. The key definition requires specific knowledge of your data and the update map the service uses.

You can create a key for any logical data service that has a primary Read function. Once you create the key, you can view it in an update map and test it.

Generate a Key

For an auto-generated key, ALDSP uses elements from the return type of the primary Read function that are designated as keys in the underlying data sources and have single (1..1) cardinality.

To auto-generate the key:

1. Be sure the logical data service has a primary Read function.
2. Open the logical data service in Studio, and click the Overview tab.
3. Right-click in the service name bar, or at the left or right of the screen, and choose Manage Key.
4. Select Generate a New Schema.

5. Accept the default key name, or give your key a name ending in .xsd.
6. Click Next.
7. Select Auto Generate the Key, then click Finish.

Auto-Generating a Logical Data Service Key

You can now use the key as an argument or return type to an update map procedure, such as a Create, Update, or Delete procedure.

If you create a key, then delete it and create another one, you need to edit the signature of your Create procedure to return the new key:

Overview tab > right-click > Edit Signature

Select Elements for a Key

When you select elements for a key, you can add any element with single (1..1) or zero (0..1) cardinality, whether or not it is a key element in the underlying data source. An element with zero cardinality is optional and might contain null values, but you can use it as a key element. This allows you to create a wider variety of keys.

For example, you might have two data sources, one using a Social Security Number to identify records, and the other, a tax identification number. Your logical data service might have a return type that joins the two sources, so that a data record has either a social security number or a tax ID number. In the return type, both the social security number and the tax ID number are optional. The key can use either element to identify the record.

⚠ You cannot select an element that has multiple (0..m or 1..m) cardinality to be part of a key.

To create a key with elements that you select:

1. Be sure the logical data service has a primary Read function.
2. Open the logical data service in Studio.

3. Click the Overview tab.
4. Right-click in the service name bar, or the left or right of the screen, and choose Manage Keys.
5. Click Generate a New Schema.
6. Give your key schema a name ending in .xsd.
7. Click Manually select the fields that make up the key.
8. Select the key fields you want, then click Finish.

Selecting Elements for a Key

Select a Key Schema File

You can also select an existing schema (XSD) file to use as the key:

1. Be sure the logical data service has a primary Read function.
2. Open the logical data service in Studio
3. Click the Overview tab.
4. Right-click in the service name bar, or the left or right of the screen, and choose Manage Key.
5. Click Select an existing schema type, then Browse.
   The Manage Keys dialog shows you the key schema's global element and selector element.
6. Click Finish.

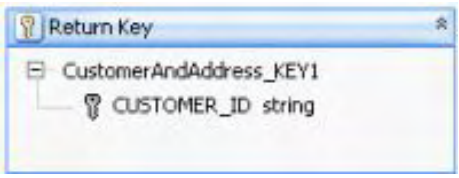
The schema in the Overview tab now displays a key icon next to the current key element or elements.

Selecting the Key Schema

View and Map a Key

Once you create the key (whether by auto-generating, identifying key fields, or selecting a key schema file), you can see the key elements in the service's update map, at the lower left.

Viewing the Key in the Update Map



The Return Key block represents the key elements a Create procedure returns when a new data record is added. In most cases, the key fields are automatically mapped to elements in the data sources on the left. If they are not mapped, you can add a mapping.

1. Locate the Update block on the left that contains the key element.
2. Drag from the key element in the Update block to the key element in the Return Key block.

Mapping a Key Element from an Update Block to the Return Key

⚠ Map the key element from an update block on the left, not from the return type on the right. If you map the key from the return type on the right, you allow the key value to be updated from data a user enters.

Once the key element is mapped, you can test it (preferably using sample data):

1. Click the Test tab.
2. At Select operation, choose one of the service's Create procedures.
3. Enter data in the XML template in the Parameters box.
4. Click Run.

The key value is returned in the Result box:

You can also view the key schema file by locating the key in the Project Explorer, right-clicking, and choosing an XML editor to open the file. A key schema looks like this:

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

In the key schema, all elements must be in the same namespace as the root element. In the previous example, the namespace of the root element is:

```
ld:logical/CustomerOrder
```

A key schema cannot contain elements in different namespaces.

⚠ If you have key schema files from a previous version of ALDSP that you want to reuse, be sure that all elements within the schema are in the same namespace.

See Also

Concepts

- Data Service Keys

Declare a Security Resource

This page last changed on Mar 11, 2008.

This topic describes how to add a security resource to a data service, so that the service returns data only if the caller has proper access.

Choose a Technique

You can add a security resource to a data service in two ways:

- The first way is to use the ALDSP Console to set elements and attributes that should be secured based on a security policy set by an administrator. This technique works in most cases for which you want to add a security policy.
- The other way, described here, is to create a custom security resource for an entity or library data service in Studio. The custom security resource is used directly in an XQuery expression to secure all or part of the service's return type. You can use the same custom security resource more than once in a single data service.

You can add a security resource to any data service, physical or logical, entity or library.

Create the Security Resource

You add a security resource to a logical entity service in Studio and then activate it using the ALDSP Console.

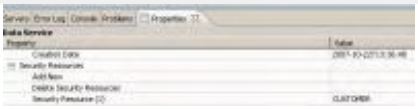
> ⚠ You can follow these steps on a physical or logical entity service. Be sure the service has a query map and a primary read function.

To create a security resource:

1. Open the service in Studio.
2. Make sure the Properties tab is displayed:

   ```
   Window > Show View > Properties
   ```

3. Click Overview, then Properties.
4. Expand the schema in the center. Locate the element you want to add the security resource to.
5. In the Properties tab, locate Security Resources.
6. Click the Add New field below it, then click  .
7. In the Value column, enter the name of the element you want to secure.



Use just an element name (CUSTOMER), not a pathname (CUSTOMER_PROFILE/CUSTOMER) or a variable ($CUSTOMER). You can use a simple element, a complex element, or the root element of the return type.

8. If needed, add more security resources and elements.
9. Click the Source tab.
   The pragma statement at the top of the XQuery source file shows the new security resource:

```
(::pragma  xds <x:xds targetType="cus:CustomerOrder" xmlns:x="urn:annotations.ld.bea.com" xmlns:cus="ld:
logical/CustomerOrder">
<creationDate>2007-10-22T13:36:48</creationDate>
<userDefinedView/>
<key name="DefaultKey" inferred="true" inferredSchema="true" type="cus:CustomersAndOrders_KEY">
  <selector xpath="CUSTOMER"/>
</key>
<secureResources>
  <secureResource>CUSTOMER</secureResource>
</secureResources>
</x:xds>::-)
```

Use the Security Resource in XQuery

The next step is to add a condition to the return type so that it is returned only if the caller has access. To do this, make changes visually in the Query Map. You want to add a conditional statement to the service's primary read function, something like this:

```
declare function tns:read() as element(cus:CustomerOrder)*{
for $CUSTOMER in cus1:CUSTOMER()
return
        <cus:CustomerOrder>
            {
            if (add-authentication-expression-here) then
                <CUSTOMER>
                        return type here ..
                </CUSTOMER>
            else
                <CUSTOMER>{return nothing here}</CUSTOMER>
            }
        </cus:CustomerOrder>
```

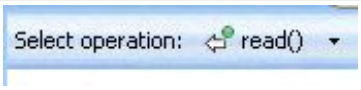
To add the conditional statement, you need to:

- Create an if .. else conditional statement.
- In the if clause, add an expression to check if the caller is authenticated and define what is returned.
- In the else clause, define which elements are returned if the caller is not authenticated.

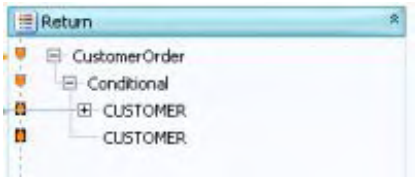
The following example shows how to create a security resource on an element in the return type, using the primary read function.

Create the If Condition

1. Click the Query Map tab.
2. At Select Operation, choose the primary read function.



3. In the return type, right-click the element for which you created a security resource in the Properties tab. Choose Make Conditional.
   A node named Conditional is added to the return type.



4. Click the Conditional node.
   You see the default conditional expression, (true), in the expression editor.

5. Make sure the Design Palette is displayed (Window > Show View > Design Palette), then click it.
6. Expand:

```
XQuery Functions > Data Services Access Control Functions
```

7. In the mapping area, click the double arrow icon  to open the expression editor.
8. Click the expression label in the editor.
9. Double-click (true), then delete it.
10. Drag the function fn-bea:is-access-allowed from the Design Palette to the editor.

```
fn-bea:is-access-allowed($label, $data_service)
```

11. For the $label argument, enter the name of your security resource as a string within quotes.
    Use the same name you used in the Properties tab.
12. For the $data_service argument, enter the namespace-qualified name of your data service as a string within quotes:

```
fn-bea:is-access-allowed("CUSTOMER", "ld:logical/CustomersAndOrders.ds")
```

13. Click the Source tab, and check the read function. Make sure it has no errors.
    Notice that the new expression is added to the if expression in the read function:

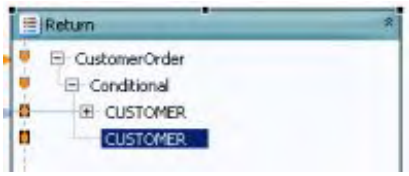
```
declare function tns:read() as element(cus:CustomerOrder)*{
for $CUSTOMER in cus1:CUSTOMER()
return
        <cus:CustomerOrder>
        {
            if (fn-bea:is-access-allowed("CUSTOMER", "ld:logical/CustomersAndOrders.ds")) then
                <CUSTOMER>
                        ...
                </CUSTOMER>
           else
                <CUSTOMER>
                        ...
                </CUSTOMER>
```

```
        }
```

14. Click Save.
    You now need to define what is returned in the else clause.

Create the Else Condition

1. Click the Query Map tab.
2. In the return type, click the second conditional element.



3. In the expression editor, enter "NA", and click Save.
4. Click the Source tab.
   The read function now shows the return value for the else clause as the string "NA".

```
declare function tns:read() as element(cus:CustomerOrder)*{
for $CUSTOMER in cus1:CUSTOMER()
return
        <cus:CustomerOrder>
            {
            if (fn-bea:is-access-allowed("CUSTOMER", "ld:logical/CustomersAndOrders.ds")) then
                <CUSTOMER>
                        <CUSTOMER_ID>{fn:data($CUSTOMER/CUSTOMER_ID)}</CUSTOMER_ID>
                        <FIRST_NAME>{fn:data($CUSTOMER/FIRST_NAME)}</FIRST_NAME>
                        <LAST_NAME>{fn:data($CUSTOMER/LAST_NAME)}</LAST_NAME>
                        <SSN?>{fn:data($CUSTOMER/SSN)}</SSN>
                        ...
                </CUSTOMER>
            else
                <CUSTOMER>{"NA"}</CUSTOMER>
            }
        </cus:CustomerOrder>
}
```

Assign Security Resources

The next step is to use the ALDSP console to create a security policy.

Securing AquaLogic Data Services Platform Resources

All you need to do in the ALDSP console is create a security policy. You have already created a custom security resource and added it to an XQuery function or procedure.

Test Security

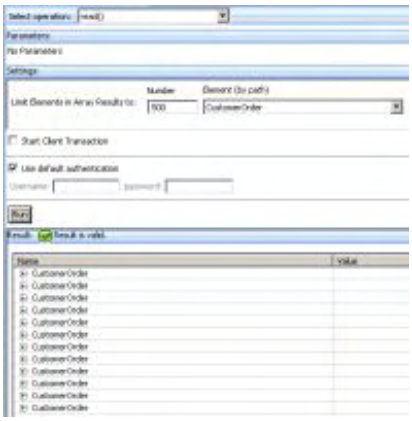
Once you establish security resources, you should test security in Test view.

To test a security resource:

1. Open the service in Studio.

2. Click the Test tab.
3. At Select Operation, choose the function you want to test.
4. Enter any parameters the function requires.
5. Expand Settings and enter the authentication credentials you want to use.
6. Click Run.

Check that the function returns either valid results if the authentication credential passes the security policy, or the string NA if it is not.



See Also

How Tos

- Add a Read Function

Other Resources

- Securing ALDSP Resources

Create a Logical Data Service with a Group By Clause

This page last changed on Mar 11, 2008.

This topic shows how to add a group by clause to a logical data service, using the BEA extensions to XQuery.

Overview

In relational data sources, a SQL GROUP BY statement is used with aggregate functions to group retrieved data by one or more columns. If you want to retrieve a list of distinct customers and the total amount of all orders each customer has placed from a relational data source, you might use a SQL statement like this:

```
SELECT CUSTOMER_ID, SUM(TOTAL_ORDER_AMOUNT) FROM ORDERS
    GROUP BY CUSTOMER_ID
```

The output produced groups all orders by customer and then totals the order amounts for each:

CUSTOMER_ID	TOTAL_OF_ALL_ORDERS
Customer0	9155.10
Customer1	5336.5
Customer2	11245.05
Customer3	1419.95

ALDSP logical data services use XQuery 1.0 to query data. XQuery, as defined by the W3C standard, does not support group by clauses. However, ALDSP has extended XQuery to allow a group by clause in an XQuery FLWOR statement:

```
declare function tns:read() as element(ord1:ORDER_GROUP_BY)*{
for $CUSTOMER_ORDER in cus:CUSTOMER_ORDER()
group $CUSTOMER_ORDER as $CUSTOMER_ORDER_group by $CUSTOMER_ORDER/CUSTOMER_ID as $CUSTOMER_ID_group
return ...
```

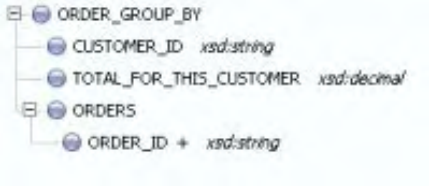
You can add the XQuery group by statement to a logical data service visually in Studio. You should first make sure the service has a return type that supports the group by.

Suppose that after you retrieve all customer orders, group them by customer, and find the total amount of all orders each customer has placed, you also want a list of order IDs for each customer. You can design a logical data service to do this, doing part of the work in the mapping editor (in Studio) and part in the XQuery source.

Design the Return Type Schema

The return type schema needs an element to group by, such as a customer ID, and an element to hold an aggregate value, such as a sum or an average. The return type can also have a complex element that contains additional elements that provide information. This example provides the list of order IDs that are totalled for each customer, as one element with multiple cardinality within a complex element.

Return Type Schema for a Group By



If you want to design the schema top down using an XML editor, you can start with code like this and refactor it for your use case:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="ld:logical/OrderGroupBy">
   <xs:element name="ORDER_GROUP_BY">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="CUSTOMER_ID" type="xs:string"/>
            <xs:element name="TOTAL_FOR_THIS_CUSTOMER" type="xs:decimal"/>
            <xs:element name="ORDERS">
                        <xs:complexType>
                                <xs:sequence>
                                    <xs:element name="ORDER_ID" type="xs:string" maxOccurs="unbounded"
form="unqualified" />
                                </xs:sequence>
                        </xs:complexType>
           </xs:element>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

You can also create the return type bottom up, as you design the query map (see Create Your First Data Services).

Create the Logical Data Service

Once you have defined the return type, create the logical data service and add the group by statement visually, using the mapping editor.

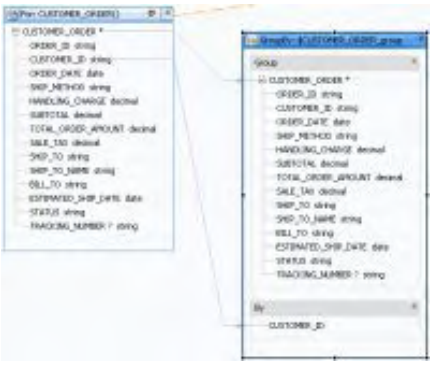
1. Create a new data space and import physical data sources (see Create Your First Data Services).
2. Create a new logical data service.
3. Click Overview, right-click the name bar, choose Associate XML Type, and select the schema file for the return type.
4. Create a primary Read function.
5. Click Query Map. Drag the primary Read function from the relevant physical data source.

At this point, do not draw additional mapping lines from the For block to the return type.

Create the Group By Node

Now create the group by node visually:

1. Right-click the element in the For block that you want to use as a grouping element, and select Create Group By.
   A Group By node is created, and mappings are automatically drawn to it. The lower section of the Group By block shows the grouping element.



2. Drag a mapping from the grouping element in the By section of the Group By node to the grouping element in the return type (here, from GroupBy CUSTOMER_ID to Return CUSTOMER_ID).
3. Drag a mapping from the appropriate element in the top section of the Group By node to the aggregate element in the return type (here, from Group By TOTAL_ORDER_AMOUNT to Return TOTAL_FOR_THIS_CUSTOMER).

Create the For Node

To map the information element, edit the XQuery code in the Source tab.

1. In the Source tab, add an XQuery for clause to the correct node in the primary Read function (here, the ORDERS node):

```
declare function tns:read() as element(ord1:ORDER_GROUP_BY)*{
for $CUSTOMER_ORDER in cus:CUSTOMER_ORDER()
group $CUSTOMER_ORDER as $CUSTOMER_ORDER_group by $CUSTOMER_ORDER/CUSTOMER_ID as $CUSTOMER_ID_group
return
        <ord1:ORDER_GROUP_BY>
            <CUSTOMER_ID>{fn:data($CUSTOMER_ID_group)}</CUSTOMER_ID>
            <TOTAL_FOR_THIS_CUSTOMER>{fn:data($CUSTOMER_ORDER_group/TOTAL_ORDER_AMOUNT)}</
TOTAL_FOR_THIS_CUSTOMER>
            <ORDERS>  {
                        for $order in $CUSTOMER_ORDER_group/ORDER_ID
                        return
                        <ORDER_ID>{fn:data($order)}</ORDER_ID>
                }
            </ORDERS>
        </ord1:ORDER_GROUP_BY>
```

```
};
```

The for statement declares a variable (here $order) and then looks for an element ($CUSTOMER_ORDER_group/ORDER_ID) in the first group the group by statement declares (CUSTOMER_ORDER_group). The for clause then returns the value of the element using the fn:data function.

2. Click Query Map. Notice that a For node has been added.

Add an Aggregate Function

Last, add an aggregate function to the aggregate element in the return type (here, TOTAL_FOR_THIS_CUSTOMER).

1. In Query Map, click the aggregate element in the return type.
   Notice that it uses the fn:data function, for example:

```
{fn:data($CUSTOMER_ORDER_group/TOTAL_ORDER_AMOUNT)}
```

2. Click in the expression. Make sure the Save and Cancel icons  are enabled.
3. Click the Design Palette (Window > Show View > Design Palette).
4. Expand XQuery Functions, then Aggregate Functions.
5. Choose a function (here, the fn:sum function with one argument) and drag it to the expression editor. Leave the existing expression there.
6. Edit the expression to use the existing expression as an argument to the aggregate function, for example:

```
{fn:sum( fn:data($CUSTOMER_ORDER_group/TOTAL_ORDER_AMOUNT) ) }
```

7. Click Save  .

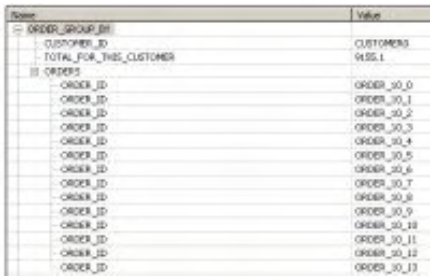
Test the Service

The only way to test a logical data service with a group clause is to run the primary Read function in the Test tab. This type of data service does not have an update map, so you cannot edit data and submit it or test an Update procedure. Likewise, you cannot test a Create procedure.

1. Click Test.
2. At Select Operation, choose the primary Read function.
3. Click Run.

You should see data grouped by the grouping element, with a result for the aggregate element, and containing a number of information elements.

Results of a Group By Statement



See Also

Examples

- Create Your First Data Services

Other Resources

- W3C XQuery Language Specification
- Extending XQuery for Grouping, Duplicate Elimination, and Outer Joins

Create a Data Service with a Flat Return Type

This page last changed on Mar 11, 2008.

This topic shows you how to create an update map from a logical data service with a flat, non-nested return type, using the sample database that ships with AquaLogic Data Services Platform.

Overview

A return type can be non-nested, or flat, even if it joins two relational tables, where one table has a one-to-many relationship with the other table . An example is one customer in a CUSTOMER table with many Orders in an ORDERS table. One approach to the return type is to nest an Orders element of multiple cardinality beneath the Customer element.

A Nested Customer-and-Orders Schema

Because you can design a logical data service with any structure, regardless of the underlying data sources, it is just as valid to define a flat return type to model the relationship between Customers and Orders.

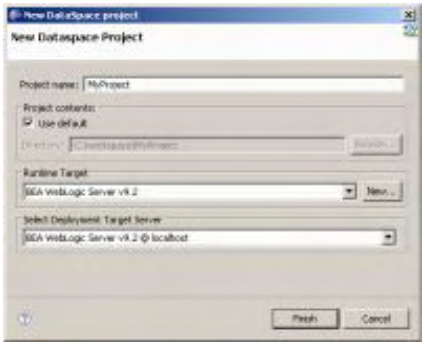
A Flat Customer-and-Orders Schema

Create a Dataspace Project

First, create a new dataspace project to contain your physical and logical data services:

1. In Studio, choose File > New > Dataspace Project.
2. Enter a project name such as FlatReturnType, then click Finish.
3. Right-click the new dataspace project name, and choose New > Folder.
4. Create folders named physical and logical. Within logical, create a folder named schemas.
   Using separate folders for physical and logical services helps separate the physical and logical integration layers.

Adding a New Dataspace Project



Create the Return Type

The return type the logical data service uses combines data from the CUSTOMER table and the ORDERS table. It has a non-nested XML structure, even though the data shows that customers and orders have a one-to-many relationship.

You can define the return type by creating an XML schema (XSD) file. In an XML editor, create a schema file like this one:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema targetNamespace="ld:logical/FlatReturnType" xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="CUSTOMERS_AND_ORDERS">
      <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="EMAIL_ADDRESS" type="xs:string"/>
            <xs:element name="ORDER_ID" type="xs:string"/>
            <xs:element name="ORDER_DT" type="xs:date"/>
            <xs:element name="TOTAL_ORDER_AMT" type="xs:decimal"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

Be sure to:

- Define targetNamespace to make sense for your dataspace project.
  Make sure you have only one top-level element of the name you choose (here, CUSTOMERORDER) in your target namespace.
  You can give the targetNamespace the same name as the dataspace project, but you are not required to.
- Save the schema file in the logical/schemas folder within your dataspace project.

Note that the cardinality of all elements uses the default values, minOccurs="1" and maxOccurs="1". Each customer has many

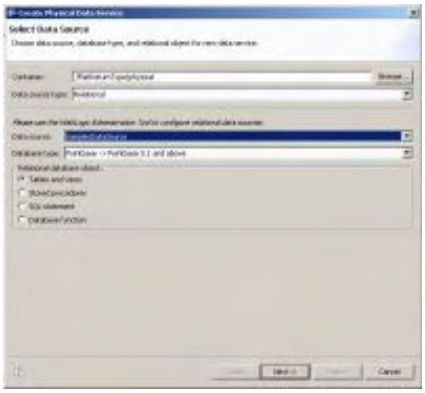
orders, but there is only one combination of customer and order, so the cardinality of the order elements (ORDER_ID, ORDER_DT, and TOTAL_ORDER_AMT) is still 1.

Create Physical Data Services

Now, create physical data services based on the sample database or your own physical data sources.

1. In Project Explorer, right-click the physical folder in your dataspace project.
2. Choose New > Physical Data Service.
3. Choose Relational for Data source type and dspSamplesDataSource for Data source, then click Next.
4. Expand RTLCUSTOMER and select CUSTOMER.
5. Expand RTLAPPLOMS and select CUSTOMER_ORDER, then click Next.
6. Select Public for both CUSTOMER and CUSTOMER_ORDER, then click Next.
7. Click Finish.
8. When asked if you want to open the new data services, click No.

Adding Physical Data Services



Create a Logical Data Service

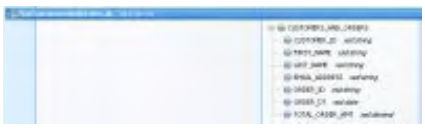
Now that you have physical data services and a schema for the return type, you can create the logical data service.

1. Right-click the logical folder, then choose New > Logical Data Service.
2. Enter a name for the service, such as FlatCustomersAndOrders.
3. Make sure Entity Data Service is selected, then click Finish.

Now associate a return type with the service:

1. Right-click in the Overview tab and choose Associate XML Type.
2. Select the schema and click OK.

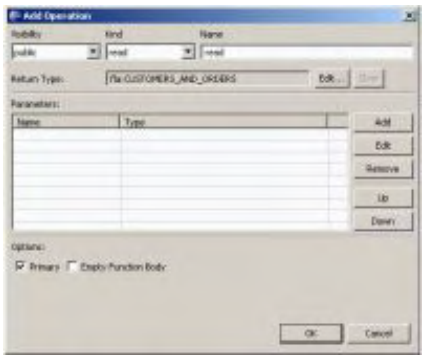
A New Logical Data Service with a Return Type



You also need to define a primary Read function, in order to create both the query map and update map.

1. Right-click in the service name bar at the top, and choose Add Operation.
2. Make sure Kind is set to read, then enter a function name, such as read.
3. Make sure Primary is selected, then click OK.

Creating a Primary Read Function



Create the Query Map

Now you need to create the query map visually in Studio, which in turn generates an update map.

1. Click the Query Map tab.
2. In Project Explorer, expand the physical data services CUSTOMER.ds and CUSTOMER_ORDER.ds.
3. Drag the Read function -- like this: CUSTOMER_ORDER() -- from each physical service to the mapping area.
   Notice that you cannot scope the CUSTOMER_ORDER block to a subtype in the return type, because the return type has no subtypes.
4. Drag mappings from the CUSTOMER block on the left to the return type for CUSTOMER_ID, FIRST_NAME, LAST_NAME, and EMAIL_ADDRESS.
5. Drag mappings from the CUSTOMER_ORDER block on the left to the return type for ORDER_ID, ORDER_DT, and TOTAL_ORDER_AMT.
6. In the For blocks, drag from CUSTOMER/CUSTOMER_ID to CUSTOMER_ORDER/CUSTOMER_ID.
   This creates a join between the two data sources.

At this point, the query map looks like this. You can see the mappings to the return type, as well as the join (the dotted line) between CUSTOMER and CUSTOMER_ORDER.

A Query Map with Mappings and a Join

If you click the Source tab and expand the Read function, you see XQuery code like this:

```
declare function tns:read() as element(fla:CUSTOMERS_AND_ORDERS)*{
for $CUSTOMER_ORDER in cus1:CUSTOMER_ORDER()
for $CUSTOMER in cus:CUSTOMER()
where $CUSTOMER/CUSTOMER_ID eq $CUSTOMER_ORDER/C_ID
return
```

```
        <fla:CUSTOMERS_AND_ORDERS>
            <CUSTOMER_ID>{fn:data($CUSTOMER/CUSTOMER_ID)}</CUSTOMER_ID>
            <FIRST_NAME>{fn:data($CUSTOMER/FIRST_NAME)}</FIRST_NAME>
            <LAST_NAME>{fn:data($CUSTOMER/LAST_NAME)}</LAST_NAME>
            <EMAIL_ADDRESS>{fn:data($CUSTOMER/EMAIL_ADDRESS)}</EMAIL_ADDRESS>
            <ORDER_ID>{fn:data($CUSTOMER_ORDER/ORDER_ID)}</ORDER_ID>
            <ORDER_DT>{fn:data($CUSTOMER_ORDER/ORDER_DT)}</ORDER_DT>
            <TOTAL_ORDER_AMT>{fn:data($CUSTOMER_ORDER/TOTAL_ORDER_AMT)}</TOTAL_ORDER_AMT>
        </fla:CUSTOMERS_AND_ORDERS>

};
```

Notice that the XQuery code has a for statement nested directly within another for statement. This creates an inner join between the two tables in SQL. To confirm the SQL that is created:

1. Click the Test tab.
2. At Select operation, make sure the primary Read function is selected.
3. Click Run (saving your data service as necessary).

You should see an XQuery FLWOR statement node. If you expand it, you should see a SQL query like this, showing an inner join:

```
SELECT t1."ORDER_DT" AS c1, t1."ORDER_ID" AS c2, t1."TOTAL_ORDER_AMT" AS c3,
  t2."CUSTOMER_ID" AS c4, t2."EMAIL_ADDRESS" AS c5, t2."FIRST_NAME" AS c6, t2."LAST_NAME" AS c7
FROM "RTLAPPLOMS"."CUSTOMER_ORDER" t1
JOIN "RTLCUSTOMER"."CUSTOMER" t2
ON (t2."CUSTOMER_ID" = t1."C_ID"
```

The inner join is created because the logical data service has a flat return type. When you mouse over the SQL query, you see this message:

```
Generated SQL query does not have a WHERE clause. This may cause the query to take longer to finish and use
excessive memory resources.
```

See Also

- Create Your First Data Services

XQuery Source of a Logical Entity Service

This page last changed on Feb 26, 2008.

This topic shows sample XQuery source code for a logical entity data service.

Topics

Source Code

```
xquery version "1.0" encoding "UTF-8";

(::pragma  xds <x:xds targetType="cus:CUSTOMER_PROFILE" xmlns:x="urn:annotations.ld.bea.com" xmlns:cus="ld:
logical/CustomerProfile">
    <creationDate>2007-10-05T10:29:01</creationDate>
    <userDefinedView/>
    <key name="DefaultKey" inferred="true" inferredSchema="true" type="cus:CustomerProfile_KEY">
        <selector xpath="CUSTOMER"/>
    </key>
</x:xds>::)


import schema namespace cus="ld:logical/CustomerProfile" at "ld:logical/schemas/CustomerProfile.xsd";

declare namespace cus1= "ld:physical/CUSTOMER";

declare namespace add= "ld:physical/ADDRESS";

declare namespace cre= "ld:physical/CREDITRATING";

import schema namespace cus2="ld:logical/CustomerProfile" at "ld:logical/schemas/CustomerProfile_KEY.xsd";

declare namespace tns="ld:logical/CustomerProfile";

declare function tns:stringToShort($theString) as xs:short {
        xs:short($theString)
};

(::pragma  function <f:function kind="read" visibility="public" isPrimary="true" xmlns:f="urn:annotations.ld.
bea.com">
   <uiProperties>
        <component identifier="returnNode" minimized="false" x="842" y="11" w="244" h="601">
            <treeInfo id="0">
                <collapsedNodes>
                    <collapsedNode>CUSTOMER_PROFILE\CUSTOMER</collapsedNode>
                    <collapsedNode>CUSTOMER_PROFILE\CUSTOMER\ADDRESS</collapsedNode>
                    <collapsedNode>CUSTOMER_PROFILE\CUSTOMER\CREDITRATING</collapsedNode>
                </collapsedNodes>
            </treeInfo>
        </component>
        <component identifier="CUSTOMER" x="44" y="56" h="300" w="219" minimized="false"/>
        <component identifier="ADDRESS" x="303" y="216" h="336" w="193" minimized="false"/>
        <component identifier="CREDITRATING" x="547" y="485" h="102" w="170" minimized="false"/>
    </uiProperties>
</f:function>::)


declare function tns:read() as element(tns:CUSTOMER_PROFILE)*{
```

```
for $CUSTOMER in cus1:CUSTOMER()
return
        <tns:CUSTOMER_PROFILE>
            <CUSTOMER>
                <CUSTOMER_ID>{fn:data($CUSTOMER/CUSTOMER_ID)}</CUSTOMER_ID>
                <FIRST_NAME>{fn:data($CUSTOMER/FIRST_NAME)}</FIRST_NAME>
                <LAST_NAME>{fn:data($CUSTOMER/LAST_NAME)}</LAST_NAME>
                <EMAIL_ADDRESS>{fn:data($CUSTOMER/EMAIL_ADDRESS)}</EMAIL_ADDRESS>
                {
                    for $ADDRESS in add:ADDRESS()
                    where $CUSTOMER/CUSTOMER_ID eq $ADDRESS/CUSTOMER_ID
                    return
                    <ADDRESS>
                        <ADDR_ID>{fn:data($ADDRESS/ADDR_ID)}</ADDR_ID>
                        <CUSTOMER_ID>{fn:data($ADDRESS/CUSTOMER_ID)}</CUSTOMER_ID>
                        <STREET_ADDRESS1>{fn:data($ADDRESS/STREET_ADDRESS1)}</STREET_ADDRESS1>
                        <STREET_ADDRESS2?>{fn:data($ADDRESS/STREET_ADDRESS2)}</STREET_ADDRESS2>
                        <CITY>{fn:data($ADDRESS/CITY)}</CITY>
                        <STATE>{fn:data($ADDRESS/STATE)}</STATE>
                        <ZIPCODE>{fn:data($ADDRESS/ZIPCODE)}</ZIPCODE>
                        <COUNTRY>{fn:data($ADDRESS/COUNTRY)}</COUNTRY>
                    </ADDRESS>
                }
                {
                        for $CREDITRATING in cre:CREDITRATING()
                        where $CUSTOMER/CUSTOMER_ID eq $CREDITRATING/CUSTOMER_ID
                        return
                        <CREDITRATING>
                        <CUSTOMER_ID>{fn:data($CREDITRATING/CUSTOMER_ID)}</CUSTOMER_ID>
                        <RATING?>{fn:data($CREDITRATING/RATING)}</RATING>
                        </CREDITRATING>
                }
            </CUSTOMER>
        </tns:CUSTOMER_PROFILE>


};

(::pragma  function <f:function kind="delete" visibility="public" isPrimary="true" xmlns:f="urn:annotations.
ld.bea.com">
    <nonCacheable/>
    <implementation>
        <updateTemplate/>
    </implementation>
</f:function>::)

declare procedure tns:deleteCUSTOMER_PROFILE($arg as element(tns:CUSTOMER_PROFILE)*) as empty() external;


(::pragma  function <f:function kind="create" visibility="public" isPrimary="true" xmlns:f="urn:annotations.
ld.bea.com">
    <nonCacheable/>
    <implementation>
        <updateTemplate/>
    </implementation>
</f:function>::)

declare procedure tns:createCUSTOMER_PROFILE($arg as element(tns:CUSTOMER_PROFILE)*) as element(tns:
CustomerProfile_KEY)* external;


(::pragma  function <f:function kind="update" visibility="public" isPrimary="true" xmlns:f="urn:annotations.
ld.bea.com">
    <nonCacheable/>
    <implementation>
        <updateTemplate/>
    </implementation>
</f:function>::)
```

```
declare procedure tns:updateCUSTOMER_PROFILE($arg as changed-element(tns:CUSTOMER_PROFILE)*) as empty()
external;

(::pragma  function <f:function kind="delete" visibility="public" isPrimary="false" xmlns:f="urn:annotations.
ld.bea.com"/>::)

declare procedure tns:deleteByKey($arg0 as element(tns:CustomerProfile_KEY)){
 do return ();
};
```

See Also

Concepts

- Building Logical Entity Data Services

How Tos

- Create Your First Data Services

Modeling Data Services Relationships

This page last changed on Feb 26, 2008.

Modeling Data Services Relationships

Concepts

- Relationship Between Data Services and Models

How-to...

- Create Your First Data Services Model
- Work with Large Models
- Generate a Relationship Modeler Report

Reference

- Relationship Modeler Options
- Model Diagram Rules
- Notable Relationship Modeler Properties
- Relationship Models in Source View

Relationship Between Data Services and Models

This page last changed on Feb 26, 2008.

In large enterprises modeling is — or at least should be — an early task in developing a data services layer. By starting with a graphical representation of data resources it is easier to view data resources globally, leveraging existing information in interesting and useful ways. It is also easy to see opportunities for creating additional business logic in the form of logical services.

Model diagrams are quite flexible; they can be based on existing data services (and corresponding underlying data sources), planned data services, or a combination. You can also create and modify data services and data service XML types directly from the model.

Relationships can be surfaced through the Relationship Modeler in several ways:

- **Automatically.** By dragging two or more relational-based data services into a model diagram simultaneously. In such cases primary/foreign key relationships -- already available in the respective data service -- appear.
- **Graphically.** Through gestures you make in your model diagram or through the right-click menu*. *
- **Programmatically.** Through a data service Source editor.

Relationship functions allows data associated with one data service (such as Customer) to serve as a complex parameter for a related data service (such as Orders). Models can represent any combination of logical and physical data services.
A visual representation of a relationship between two data services can convey a considerable amount of information:

- **Cardinality.** Is the relationship one-to-zero (customers and promotional offers), one-to-one (customer and primary email), one-to-many (customers and orders), or many-to-many (customer orders and ordered items)?
- **Direction.** Arrows indicate possible navigation paths. Is there an originating entity associated with a subordinate entity (such as orders and order items) or is the relationship bidirectional (such as customers and orders)?
- **Roles.** A name matching the name of the adjacent data services navigation function (see below). Does the assigned relationship name capture the purpose of the navigation function it represents?

Many data service-related operations can be performed from the relationship modeler including:

- Modeling a high-level, visual view of data resources
- Viewing and adding to the relationships between data services
- Accessing or creating a data service
- Add operations to a data service
- Change the XML type (schema) associated with a data service

Navigation functions are visible as properties of each data service in the binary relationship. They can be fully inspected in the Source editor for each data service. Navigation functions also appear as mouse-over text over each endpoint of the relationship line.

By default, types shown in model diagrams are XML schema types, but you can change this to display native data source types in the case of physical data services.

For more information on data service modeling concepts see Modeling and a Service-Oriented Architecture in the ALDSP 2.5 Concepts Guide.

Create Your First Data Services Model

This page last changed on Feb 26, 2008.

This section provides a basic overview of modeling in ALDSP and a tutorial.

Topics

Introduction

Using ALDSP, you can create and maintain models of your enterprise data services. A model diagram is a graphical representation of a data model supported by ALDSP.

Through data models you can:

- Model a high-level, visual view of data resources.
- View and extend relationships between data services.
- Access and create a data service.
- Add operations to a data service.
- Change the XML type (schema) associated with a data service.

In model diagrams, a relationship can be created by the gesture of drawing a line from one data service to another. In some cases (such as relational data services) relationships and the lines representing the relationship can be automatically inferred. In other cases, you need to create the relationship.

A visual representation of a relationship between two data services conveys a considerable amount of information:

- **Cardinality.** Is the relationship one-to-zero (customers and promotional offers), one-to-one (customer and primary email), one-to-many (customers and orders), or many-to-many (customer orders and ordered items)?
- **Direction.** Arrows indicate possible navigation paths. Is there an originating entity associated

with a subordinate entity (such as orders and order items) or is the relationship bidirectional (such as customers and orders)?

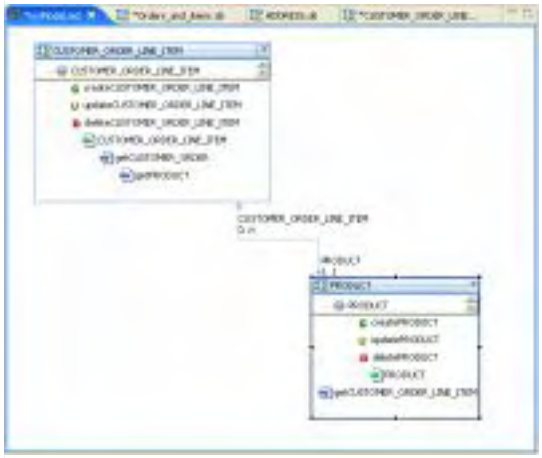
- **Roles.** A name matching the name of the adjacent data services navigation function (see below). Does the assigned relationship name capture the purpose of the navigation function it represents?

Navigation functions are visible as properties of each data service in a binary relationship. Navigation functions also appear as mouse-over text over each endpoint of the relationship line.

Types shown in model diagrams are XML schema types.

> ⚠ For more information on data service modeling concepts see Modeling and a Service-Oriented Architecture in the ALDSP 2.5 Concepts Guide.

Model Diagram of Physical Data Services



Building a Simple Data Service Relationship Model

You can create a sample data service relationship model by selecting a dataspace project and choosing:

File > New > Relationship Modeler

You can locate your model diagram anywhere in your project. Any legal filename can be used.

About the Data Services

This example assumes that you are using the ALDSP RTLApp as a data source.

The physical data services used in this sample are:

- CUSTOMER
- CUSTOMER_ORDER
- CUSTOMER_ORDER_LINE_ITEM

Creating and Updating Physical Data Services

Adding Data Services to the Modeler

You can add data services to your model using simple drag-and-drop from the Project Explorer. In the Project Explorer you can multi-select data services using either:

- Shift-click (contiguous services) or
- Control-click (individual services)

If you drag a set of data services into a model diagram, existing relationships to other data services in the model will be shown.

Populating the Relationship Modeler

Since the data services in this example are representations of relational sources, a several bidirectional relationships between CUSTOMER_ORDER and CUSTOMER_ORDER_LINE_ITEMS were inferred:

- The role named CUSTOMER_ORDER has a 1-to-1 relationship with CUSTOMER_ORDER_LINE_ITEM, meaning that a line item can only belong to one order.
- The role named CUSTOMER_ORDER_LINE_ITEM has a 0-to-n relationship with CUSTOMER_ORDER, meaning meaning that there can be many line items associated with an order.

Creating an Additional Relationship

A next step could be to create a relationship between CUSTOMER and CUSTOMER_ORDER.

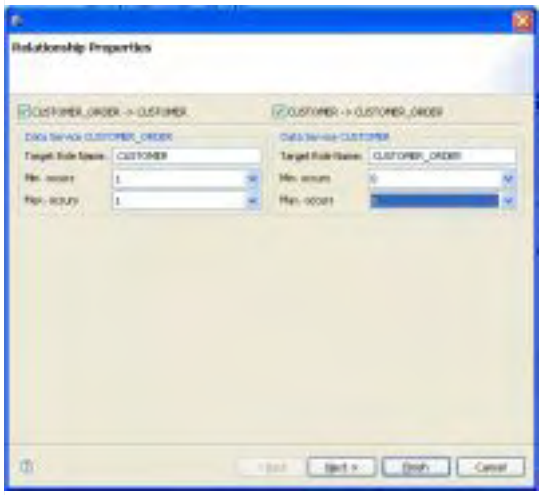
1. Right-click on CUSTOMER_ORDER node.
2. Select Create Relationship to another data service.
3. Select CUSTOMER as the target data service.
4. Click OK.

The Relationship Properties wizard appears.

Setting Relationship Properties

Relationship properties can be uni- or bi-directional.

Setting Relationship Options



Relationship Properties Dialog Options

Option	Action	Comment / Reference

Set directionality	**Select the directions to be supported in the relationship. The example is bidirectional so the default checked condition for the following relationships need not be changed:** • CUSTOMER_ORDER -> CUSTOMER • CUSTOMER -> CUSTOMER_ORDER	Creating relationships in a model automatically creates relationship functions between data services. Bi-directional settings mean that "get" functions for the related data service will be created on both sides of the relationship. By default, relationships are bidirectional.
Target Role Name	**Enter the name of the role function. In the example, default names can be used:** • CUSTOMER • CUSTOMER_ORDER	By default the name will be based on the name of the related data service. It can be changed to any unique and legal name in your dataspace project.
Set maximum and minimum occurances	**Enter cardinality settings for the respective function. For the example the following settings are used:** • CUSTOMER_ORDER -> CUSTOMER: 1-to-1 • CUSTOMER -> CUSTOMER_ORDER 0-to-n	The minimum and maximum occurrence settings definite the nature of the relationship between the two services.

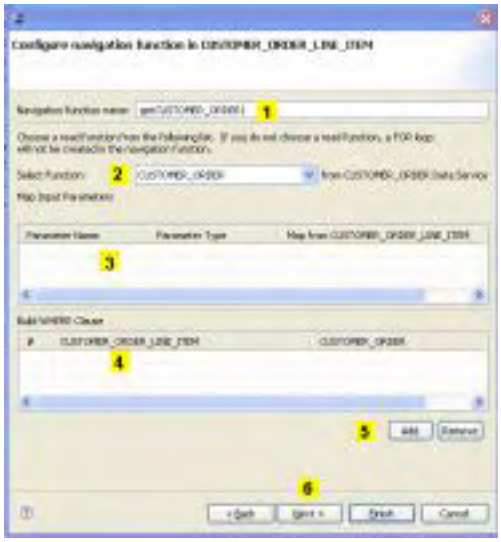
Click Next.

Configuring Navigation Functions

Each navigation function (one or two) being created also needs to be configured. Configuration includes:

- Setting a name for the navigation function
- Selecting a function from the newly related data service.
- Mapping input parameters
- Building a WHERE clause

Configuring a Navigation Function



Specifying Relationship Wizard Function Name, Parameters, and Where Clauses

Element	Purpose
Navigation function name	By default, the navigation function name is the name of the target data service with "get" prepended, as in "getCustomer". If a function of that name exists, numbers will be appended to the function name as in getCustomer1. When you invoke the Relationship wizard through a model diagram the opposite data service is determined by the gesture of drawing a line from one data service to another. In such cases the option of selecting a navigation function name is not present.
Related data service function	By default, the root function in the target data service is selected. However, you can select any available read function in the target data service.
Map input parameters	If the related function has input parameters, the name and type of the available parameters appear. You can then use a pulldown menu to select an element from the target data service to map as the input parameter.
Build WHERE clause	Where clauses can be added to the function using pulldown menus that allow you to select join elements from each side of the relationship.

Add or Remove	Allows you to add additional where clauses or delete an identified where clause.
Next	When the relationship between data services is bidirectional clicking Next changes the focus to the second data service, where you can identify a navigation function name, parameters, and add where clauses for the second side of the relationship.

Customer-Order-Item Model

Work with Large Models

This page last changed on Feb 26, 2008.

Model diagrams can hold any number of data services. The only limitation is that each data service must reside in the same dataspace.

Some tools are available in cases where very large models have been created.

Search

You can locate any data services in your model diagram.

1. Right-click in the white space (not on a data service representation) and select Find Data Service.
2. Type in the name of your data service using standard search options available in the dialog.

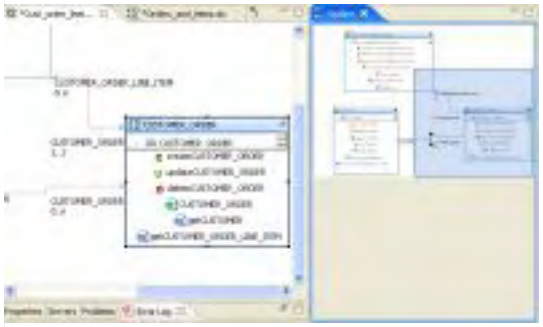
A dialog will appear containing a dropdown list of matching data services. The data service you select will be appear.

Outline Mode

For larger models you can use Outline view which will allow you to scroll through your model.

Window > Show View > Outline

Model Diagram Outline View



Generate a Relationship Modeler Report

This page last changed on Feb 26, 2008.

Both summary and detailed reports are available from the currently selected relationship model.

To create a report:

1. Right-click in any blank space in your model diagram.
2. Select:

   ```
   Generated Report > Detailed or Generate Report > Summary
   ```

3. Select a filename and location.
4. Click Finish.

Summary and Detailed Report Categories Compared

Type	Description
Summary Report	Provides general information related the model including: • Location of each data service in the model • Type: logical or physical • Allows updates: true/false • Data source type • Data source name • Owner (if any) • Comment (if any) • Date created • Date last modified
Detailed Report	A detailed model report contains all summary information listed above and, for each relationship between data services, the following additional information: • Return type fully qualified name (the *qname*) • Details on each Read function including Return type, description, and comments • Details on the data service relationships including role name, target data service, minimum and maximum occurrences, opposite role name, navigation functions including Return type, description, comment and user-defined properties • Dependencies — a list of all dependent data services

When you choose the Create a Model Report right-click option you are asked to select a name for the HTML document that is generated. By default, the name of the summary report is:

```
<model_name>_md_summary.html
```

and the name of the detail report is:

```
<model_name>_md_detail.html
```

You can save the report to any location in your application, including to a new folder.

Model Report Format

The model report is in HTML format.

Print your report from any browser or application that supports HTML printing.

Relationship Modeler Options

This page last changed on Feb 26, 2008.

This section describes some of the common operations you will use when working with the relationship modeler.

Model Right-click Menu Options

You can edit your model using a combination of right-click menu options and the model Property Editor. Table 5-14 describes right-click options based on the functional area of the model diagram that is in scope.

Notable Data Model Options

Scope	Command	Meaning
Data Model	**New Data Service**	Allows you to create a new data service. After selecting a name and physical location for the data service, the service is created and placed on the diagram.
	Find Data Service	Locates a data service within your model.
	Select Router Type	Adjusts visual presentation of relationship lines based on the Manhattan model or shortest-path model.
	Generate Report	Creates either a Summary or Detail report in an Eclipse HTML-based page. The report describes data services in your model, their bilateral relationships, and a description of each data service.
Data Service	**Open**	Opens the currently selected data service in Design View (see Creating a Data Service).
	Create Relationship to Another Data Service	Dialog allows you to select from a list of data services in the model diagram. As with drawing a line between two data services, this option brings up the Relationship wizard. (See Using the Relationship Wizard to Create Navigation Functions.

Add Related Data Service	The Add Related command is available when one or several data services are selected in the model. Add Related lists data services that contain navigation functions referencing your currently selected data source. Click on the service you want to add and then repeat the process to add other available related services, if any.
Remove from Diagram	Removes the selected data service from the model diagram. Alternatively, use the Delete key.  This operation does not affect the underlying data service.
Refactor	Provides for either safe delete or renaming of the currently selected data service. This is comparable to operations available for a data service from the Overview tab.
Associate XML Type	Provides a dialog where a different schema (XSD) file can be selected from the current project.  Changing a schema type for a data service can affect its functions as well as its relationships to other data services.
Manage Key...	Opens the Manage Key dialog box, allowing for modification of the key associated with the current data service.
Delete Key...	Deletes any key asociated with the current data service.
Add Operation	Adds an operation (function or procedure) to the currently selected data service.
Show / Hide Native XML Types	Optionally displays/hides native types for elements representing physical objects associated with simple data types. Example: VARCHAR(25).
Show Function Signatures	Displays/hides full read function signatures such as: getAddress() as element(Address)

Relationship lines connecting data services can be deleted by first selecting the line, then pressing Delete.

Model Diagram Rules

This page last changed on Feb 26, 2008.

Model diagrams follow a set of rules:

- Each entity in the model has a title which is the local name of the data service (the fully-qualified name is visible as a mouse-over).
- Associated Read functions can be displayed, with or without signatures.
- Model diagrams do not "own" data services, but simply reference them. Multiple models can, without limit, contain representations of the same data service or relationships between data services.
- Models are not nested. That is, a model diagram cannot reference another.
- Multiple models can be defined and located anywhere in your project.
- Changes made to a model diagram can be reversed using the Edit Undo command. However it is important to keep in mind that changes to any underlying files such as schemas (XML types) or data services made through the model will not be undone. Instead, edit the data service directly or close and reopen your application before saving your changes.

Changes to a model diagram that affect data services such as when a new relationship is created are only made permanent in WebLogic Workshop after you do a File > Save All

Notable Relationship Modeler Properties

This page last changed on Feb 26, 2008.

Properties both reflect and define relationships created in the model diagram.

Notable Data Modeling Properties

Scope	Property	Settings	Comments
Data Service			Properties described in Managing Your Data Service.
Relationships Nodes		Read only	Shows names of the related data services and their respective roles. Roles are assigned as source data service and target data service, but these assignments are arbitrary in the case of bidirectional relationships.
	Source and Target Cardinality	Drop down	Value can be 0-to-1, 0-to-many, 1-to-1, 1-to-many, and many-to-many.
Operations			Properties described in Managing Your Data Service.

Relationship Models in Source View

This page last changed on Feb 26, 2008.

An example of a navigation function in the underlying source is:

```
(::pragma function <f:function xmlns:f="urn:annotations.ld.bea.com" kind="navigate" roleName="ADDRESS"/>::-)
```

This specifies a relationship to the Address data service from the Customer data service.

Data services also contain declarations describing the nature of the relationship; this information is the source for the role names and cardinality values that appear in your model diagram.

For example, the data service Address contains the following relationship declarations:

```
<relationshipTarget roleName="CUSTOMER" roleNumber="1" XDS="ld:DataServices/CustomerDB/CUSTOMER.ds"
opposite="ADDRESS"/>
```

For each data service, a relationship is created which identifies its role name, cardinality, opposite data service, and a unique (to the data service) role number.

In the above example, a navigation function is automatically created that retrieves customer information based on the customerID.

In the case of the relationship between Customer and Address, the relationship is 0-to-n for the Address role (it can make and appearance any number of times or not at all) based on CustomerID being a foreign key in Address and a primary key in the Customer data service (and the underlying relational data sources respectively).
Since the relationships are bilateral, Customer's opposite is Address while Address's opposite is Customer.

If your data model is composed of both physical and logical data services, you should keep in mind that a metadata update on any underlying physical data services will remove any relationships you have created involving those data services.

Building XQueries

This page last changed on Feb 26, 2008.

Building XQueries

How-to...

- Create a Return Type
- Add a Complex Child Element to a Return Type
- Check Namespaces in Return Types
- Create Conditional Elements in Return Types
- Add a Where Clause to a Query
- Use the XQuery Expression Editor
- Use the Source Editor

Reference

- XQuery Language Version Support
- Built-in XQuery Functions

Related Topics

How-to...

- Create Your First Data Services
- Test a Read Function and Simple Update
- Test a Create or Delete Procedure
- How To Develop Good XQSEs

Concepts...

- Understanding Data Service Annotations

Reference...

- XQuery Scripting Extensions

Create a Return Type

This page last changed on Mar 11, 2008.

This topic describes the basics of creating return types for logical entity data services in the Query Mapper and directly in XML.

Choose a Technique

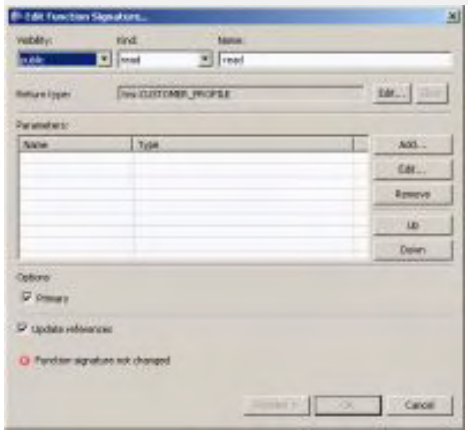
Data services use both XML types and return types.

XML types represent the shape of a logical data service, in the form of an XML schema. They are templates from which return types are created, comparable to a Java class. You use an XML type when you first create a logical entity service and add an XML schema to define its shape.

Adding an XML Type to a Service

Return types represent the shape of data that a query produces when it is run. They are specific instances of an XML type, comparable to a Java object. Return types are the R in an XQuery FLWOR clause. For example, a service's primary read function returns a return type.

Checking the Return Type of a Read Function



An XML type is the backbone of a logical data service, because it defines the data the service returns. The XML schema that represents the XML type can combine any elements from any data sources the logical data service uses, including relational sources, web services, XML files, text files, and Java methods.

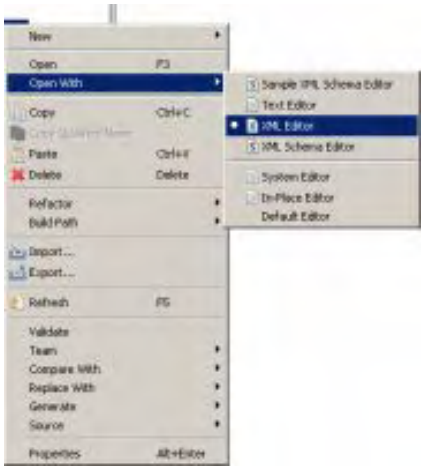
The schema for the logical data service is designed as a separate layer of the dataspace project, regardless of the actual structure of the underlying physical data sources. The schema is not required to use all elements in, or the same structure as, the physical data sources.

You can create a return type schema, an XSD file, in two ways:

- Top down, in an XML editor, either the one built into Studio or a standalone editor.
- Bottom up, by building the service visually in Query Map view and then using the Save and Associate XML Type command.

You should create the XSD file in the logical layer of your dataspace project, as it belongs to the logical data service. Studio provides several XML editors, which you can see if you right-click an XSD file in the Project Explorer and choose Open With.

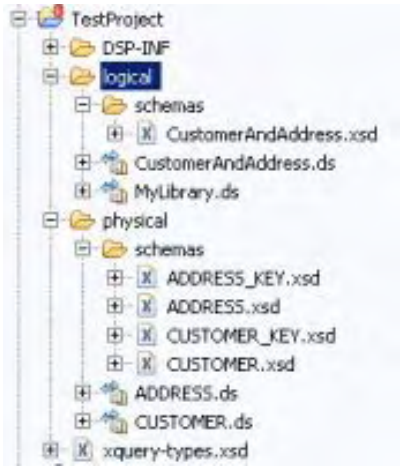
Choosing an XML Editor in Studio



Write a Return Type Schema

To create the schema in an XML editor in Studio:

1. Choose a location for logical data service schemas in your dataspace project.
   You may want to create a folder for schemas in the logical layer of your project (for example, MyDataSpace/logical/schemas) separate from the schemas folder that ALDSP auto-generates for physical data services.



2. Choose File > New > Other.
3. Choose XML > XML Schema, and click Next.
4. Choose a folder, enter a file name that ends in .xsd, and click Finish.
   The generated schema looks something like this:

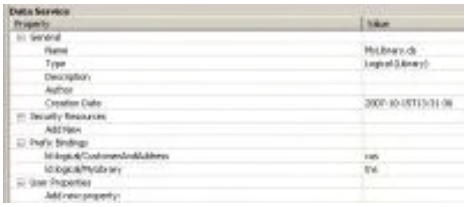
```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
                targetNamespace="http://www.example.org/MySchema"
                xmlns:tns="http://www.example.org/MySchema" elementFormDefault="qualified">

</schema>
```

5. In the XML editor, change the URL of targetNamespace to one within your dataspace project:

```
targetNamespace="ld:logical/CustomerAndAddress"
```

   The targetNamespace URL should start with the prefix ld:, and logical indicates that the schema resides in the folder named logical in your dataspace project. The identifier that follows (here, CustomerAndAddress) defines the namespace.

6. Delete the namespace definition for xmlns:tns, if your service binds tns to a different namespace. You can check this by clicking the Overview tab, then the Properties tab.



At this point, your schema file should like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
              targetNamespace="ld:logical/CustomersAndOrders"
              elementFormDefault="qualified">

</schema>
```

7. Continue adding complex types, elements, and attributes using the XML editor.
8. Save the file, then right-click anywhere in it and choose Validate.

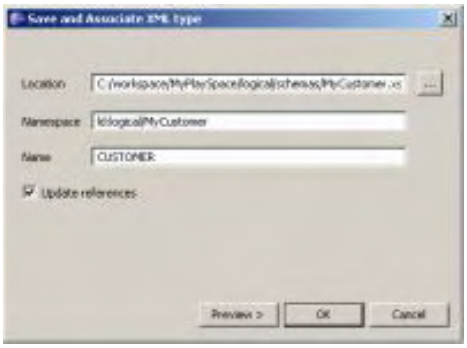
You can also create the return type schema using an XML editor outside Studio and then move the XSD file to your Studio dataspace project.

Generate a Schema File

You can also have ALDSP generate the return type schema after you build the query map visually.

To generate the schema in Studio, follow these instructions (or see Create Your First Data Services for detailed instructions):

1. Create a dataspace.
2. Create physical data services in the dataspace.
3. Also in the dataspace, create a logical data service (File > New > Logical Data Service).
4. Create a Read function in the logical data service (Overview tab, right-click, Add Operation).
5. Drag the Read functions of the physical services you want to use to the Query Map tab.
6. Click Overwrite  , and drag the root element in the For box to the root element in the Return type.
7. Right-click on the complex element in the Return type, and choose Expand Complex Mapping.
8. Right-click the return type box, and choose Save and Associate XML Type.



For Location, select the correct folder for logical schemas. In Namespace, enter a namespace that starts with ld:logical, such as ld:logical/MyCustomer. Be sure that the name of the root element (here, CUSTOMER) is unique within the namespace. (The ld namespace refers to the original name of ALDSP, Liquid Data).

9. Click OK.
10. Save the file, then right-click anywhere in it and choose Validate.

See Also

How Tos

- Add a Complex Child Element to a Return Type
- Create Your First Data Services

Other Resources

- XML Schema Tutorial at W3Schools
- XML Schema Part 1: Structures
- XML Schema Part 2: Datatypes

Add a Complex Child Element to a Return Type

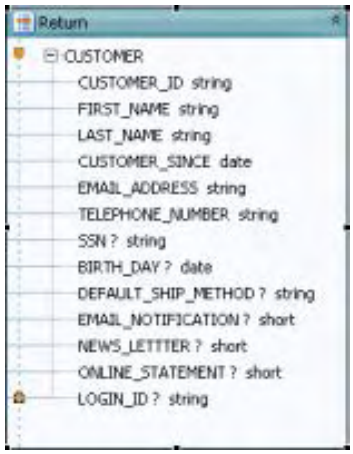
This page last changed on Mar 11, 2008.

This topic describes how to add a complex child element to a return type, in Studio or in the XML source of the return type.

Add the Child Element Visually

Once you create a return type, you can add a complex type as a child of any element, in Query Map view. The complex child element must represent a physical data service. The parent element can have a one-to-many or one-to-one relationship with the child, depending on how you want the result data returned.

A Simple Return Type Before Adding a Child Element



To add a complex child element to a return type visually:

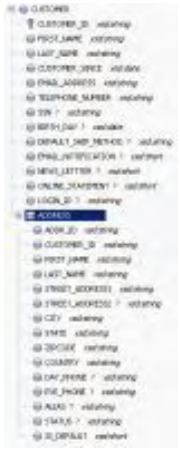
1. Open the logical data service in Studio.
2. Check Project Explorer. Be sure that your dataspace project has a physical data service for the complex child element you want to add. If it does not, add one

   ```
   File > New > Physical Data Service
   ```

3. Click the Query Map tab.
4. In the return type, right-click the new parent element, and choose Add Complex Child Element.

5. For the Schema File field, browse (...) to the schema of the physical data service that represents the complex child element.
6. For Type, choose a complex type from the schema, then click OK.
7. From Project Explorer, drag the primary read function of the physical data service to the Query Map.
8. Starting from the child element's For block, drag the zone icon to the child element in the return type.
9. Starting from the child element's For block, drag the parent type of the complex element to the return type.
   This step maps all of the elements in the complex child to the return type.

10. Right-click the title bar of the return type, and choose Save and Associate XML Type.
11. Click the Overview tab, and expand the schema to view the complex child in the return type.
    You can also right-click the schema and choose Edit Schema to view the XML source.



Edit the XML Source

Adding the complex child element to the return type in the XML source accomplishes the same thing as adding it visually.

To add a complex child element to a return type in XML source:

1. Open the logical data service in Studio.

2. Check Project Explorer. Be sure that your dataspace project has a physical data service for the complex child element you want to add. If it does not, add it:

```
File > New > Physical Data Service
```

3. Click the Overview tab.
4. Right-click the return type schema in the center, and choose Edit Schema.
   You see the schema for the logical data service, without the child element:

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

5. In Project Explorer, right-click the schema file of the physical data service that represents the child element, and choose Open With.
   You see the schema of the child element.
6. Copy the complex type from the physical data service schema to the logical data service schema. Take only the complex type:

```
<xs:complexType>
         <xs:sequence>
            <xs:element name="ADDR_ID" type="xs:string"/>
            <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="STREET_ADDRESS1" type="xs:string"/>
            <xs:element name="STREET_ADDRESS2" type="xs:string" minOccurs="0"/>
            <xs:element name="CITY" type="xs:string"/>
            <xs:element name="STATE" type="xs:string"/>
            <xs:element name="ZIPCODE" type="xs:string"/>
            <xs:element name="COUNTRY" type="xs:string"/>
            <xs:element name="DAY_PHONE" type="xs:string" minOccurs="0"/>
            <xs:element name="EVE_PHONE" type="xs:string" minOccurs="0"/>
            <xs:element name="ALIAS" type="xs:string" minOccurs="0"/>
            <xs:element name="STATUS" type="xs:string" minOccurs="0"/>
            <xs:element name="IS_DEFAULT" type="xs:short"/>
         </xs:sequence>
 </xs:complexType>
```

7. Right-click in the schema, and choose Validate.

See Also

How Tos

- Create a Return Type

- Check Namespaces in Return Types

Other Resources

- XML Schema Tutorial (W3Schools)
- XML Schema Part 1: Structures (W3C)
- XML Schema Part 2: Datatypes (W3C)

Check Namespaces in Return Types

This page last changed on Mar 11, 2008.

This topic shows you how to make sure the namespaces used in your return type are correct.

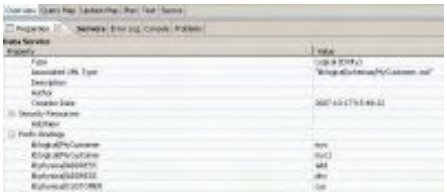
Check Prefix Bindings

In the return type, a child element must be in the same namespace as its parent. If a return type uses elements in different namespaces, you cannot deploy the logical data service to the server or test it from Studio.

The exception to this rule is when the parent and child are in different namespaces, but both namespaces have the same prefix binding. Check prefix bindings first, and then edit the namespace, if needed.

To check prefix bindings in the Overview tab:

1. Click the Overview tab.
2. Click the Properties tab (if it's not visible, choose Window > Show View > Properties).



To check prefix bindings in Source:

1. Click the Source tab.
2. Look for the XQuery namespace statements:

```
import schema namespace myc="ld:logical/MyCustomer" at "ld:logical/schemas/MyCustomer.xsd";
declare namespace cus= "ld:physical/CUSTOMER";
import schema namespace myc1="ld:logical/MyCustomer" at "ld:logical/schemas/MyCustomer_KEY.xsd";
```

In both these examples, the myc and myc1 namespaces have the same prefix binding. You can have a parent element in one and a child element in another. But if you have a parent element in myc and a child in cus, you need to change one namespace in the return type.

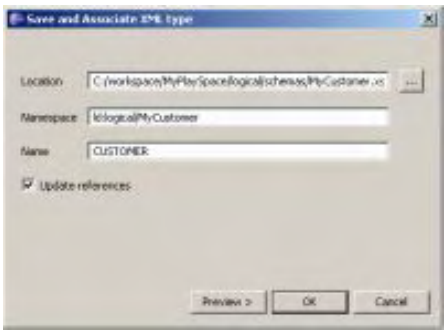
Edit the Namespace

Once you check the prefix binding, you can check a namespace used in a return type and change it in the Query Map or Source view.

To edit a namespace in Query Map view:

1. Click the Query Map tab.
2. Select the parent element in the return type, then click it.
   Be sure to select and then click; do not double-click.

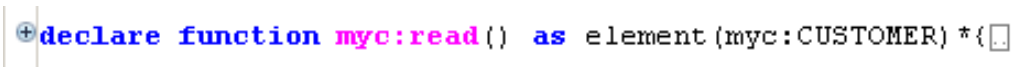
3. Select the child element in the return type, then click it.
4. If the child element is in a different namespace, change it to the namespace of the parent.
5. Right-click the title bar of the return type, and choose Save and Associate XML Type.



6. Enter the correct location, namespace, and root element name for the return type. Click OK.

To edit a namespace in Source view:

1. Click the Source tab.
2. Expand the primary Read function:



3. Locate the namespace of the child element and change it to the namespace of the parent, both in the start and end elements:

```
declare function myc:read() as element(myc:CUSTOMER)*{
for $CUSTOMER in cus:CUSTOMER()
return
    <myc:CUSTOMER>
    ...
   {
        for $ADDRESS in add:ADDRESS()
        where $CUSTOMER/CUSTOMER_ID eq $ADDRESS/CUSTOMER_ID
        return
        <myc:ADDRESS >
        ...
        </myc:ADDRESS>
   }
```

4. Save the changes.

See Also

How To

- Create a Return Type
- Add a Complex Child Element to a Return Type

Other Resources

- XML Schema Tutorial at W3Schools
- XML Schema Part 1: Structures
- XML Schema Part 2: Datatypes

Create Conditional Elements in Return Types

This page last changed on Feb 26, 2008.

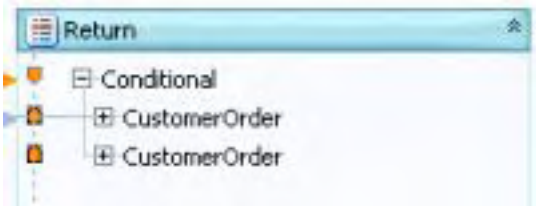
This topic describes how to add a condition to a return type and determine the elements that are returned when the condition is true or false.

Add the Condition

A condition in a return type defines two groups of elements: those returned when an expression is true, and those returned when an expression is false. When you add a condition to a return type, you see two groups of return type elements.

To add a condition to the return type:

1. Click the Query Map tab.
2. Right-click an element in the return type, and choose Make Conditional.
   The conditional element is now duplicated.



Create the Expression

You must add the conditional expression, that determines which element is returned, in the XQuery source. You cannot add a conditional expression in the expression editor.

1. Click the Source tab.
   The primary Read function now has an if..else clause:

```
declare function tns:read() as element(cus:CustomerOrder)*{
for $CUSTOMER in cus1:CUSTOMER()
return
    if (true()) then
       <cus:CustomerOrder>
         ...
       </cus:CustomerOrder>
    else
       <cus:CustomerOrder>
         ...
       </cus:CustomerOrder>
};
```

The expression after the if statement is evaluated, and the service returns either the first or second set of elements. The XQuery true() function simply returns the Boolean value true.

2. In the XQuery source, replace true() with another XQuery expression, for example:

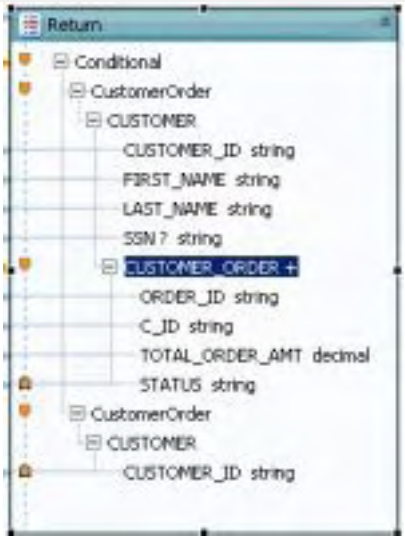
```
if ( fn:data( $CUSTOMER/LAST_NAME ) = "Black" ) then
```

You can use any XQuery expression that returns a value of true or false. In this example, if a customer has the last name Black, the first element group is returned. If not, the second element group is returned.

To add the value of an element in a For block, use the XQuery fn:data function, which takes the value of an element:

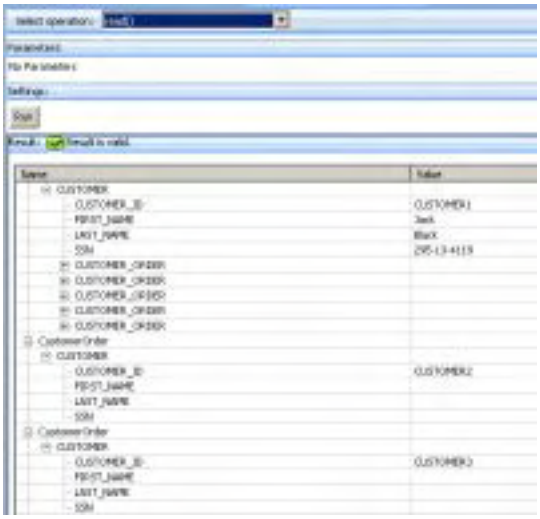
```
<LAST_NAME>Black</LAST_NAME>
```

3. Click the Query Map tab.
4. In the return type, add or delete elements in either group to create the return groups you want. Remember that the first group is returned if the expression is true, and the second group if the expression is false.



5. Click the Test tab. Choose the Read function, and click Run. Check that the results are what you intend.

   In this example, the full group of elements is only returned for customers with the last name Black. For other customers, only the CUSTOMER_ID is returned.



See Also

How Tos

- Create a Return Type
- Add a Complex Child Element to a Return Type
- Test a Read Function and Simple Update

Other Resources

- Introduction to XQuery
- XQuery Tutorial
- XQuery 1.0 Specification

Add a Where Clause to a Query

This page last changed on Feb 26, 2008.

This topic describes several ways of adding XQuery where clauses to queries to join relational data sources.

Define the Condition

A where clause in XQuery specifies criteria defining some return data. This is a simple XQuery where clause:

```
where $CUSTOMER/CUSTOMER_ID = "1111"
```

A where clause is usually part of an XQuery FLWOR (for-let-where-order by-return) expression. The where clause can be any XQuery expression, including another FLWOR expression. A common use of a where clause is to join two relational data sources, for example:

```
for $CUSTOMER_ORDER in cus1:CUSTOMER_ORDER()
where $CUSTOMER/CUSTOMER_ID eq $CUSTOMER_ORDER/C_ID
return
... xml elements here ...
```

The where clause here specifies a condition that defines a subset of results to return. The SQL statement ALDSP generates from this XQuery expression creates a left outer join between two tables:

```
SELECT t1."CUSTOMER_ID" AS c1, t1."FIRST_NAME" AS c2, t1."LAST_NAME" AS c3, t1."SSN" AS c4,
  t2."C_ID" AS c5, t2."ORDER_ID" AS c6, t2."STATUS" AS c7, t2."TOTAL_ORDER_AMT" AS c8
FROM "RTLCUSTOMER"."CUSTOMER" t1
LEFT OUTER JOIN "RTLAPPLOMS"."CUSTOMER_ORDER" t2
ON (t1."CUSTOMER_ID" = t2."C_ID")
ORDER BY t1."CUSTOMER_ID" ASC
```

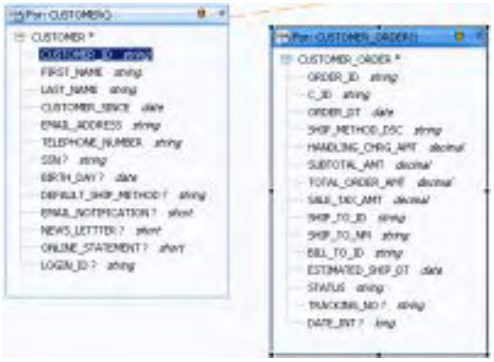
Before you add a where clause to a logical data service, think about how to structure it. If you want to join two data sources, you can only do so on a key field that appears in both. In this example, the CUSTOMER table has a primary key named CUSTOMER_ID joined to a CUSTOMER_ORDER table with a foreign key named C_ID.

Join Tables with a Where Clause

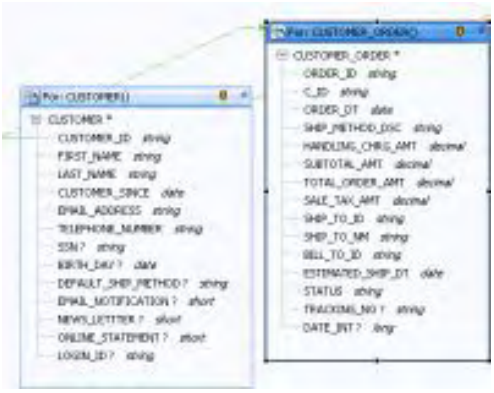
The simplest way to create a where clause between two relational data sources is to map it in Query Map view.

To map the where clause:

1. Open a logical data service in Studio.
2. Click Query Map.
3. Drag the read functions of at least two physical data sources from Project Explorer to the Query Map view.



4. In Query Map view, drag from a key element in the first data source to the corresponding key element in the second.



If you click the second data source, you see the XQuery where clause in the expression editor:

```
Where $CUSTOMER/CUSTOMER_ID eq $CUSTOMER_ORDER/C_ID
```

Use an XQuery Function in a Where Clause

A where clause can also contain an XQuery function, including any built-in or BEA-defined functions available from the Design Palette. The where clause is defined on an element within a For node.

To create a where clause with an XQuery function:

1. Click Query Map.
2. Click the For title bar of the node that contains the element.
3. Click Add Where Clause  to insert the where clause.
4. Open the Design Palette (Window > Show View > Design Palette).
5. Expand XQuery Functions, then choose a function (for example: Duration, Date, and Time Functions > fn: year-from-date).
6. Drag the function to the expression editor.
7. Delete $arg in the function, then click the element in the For node that you want to add.
8. Add an operator and a value to complete the expression.

   ```
   fn:year-from-date($CUSTOMER/CUSTOMER_SINCE) < 2000
   ```

   You can use any of the XQuery operators available in Design Palette > XQuery Operators.

9. Click Save  .
   In Source view, the where clause in the read function looks like this:

   ```
   declare function tns:read() as element(tns:CUSTOMER_PROFILE)*{
   for $CUSTOMER in cus1:CUSTOMER()
   where fn:year-from-date($CUSTOMER/CUSTOMER_SINCE) < 2000
   return
   ... xml elements here ...
   ```

10. Test the query in Test view, preferably on sample data, to make sure the results are what you expect.

See Also

How Tos

- Test a Read Function and Simple Update
- Test a Create or Delete Procedure

The XQuery Expression Editor

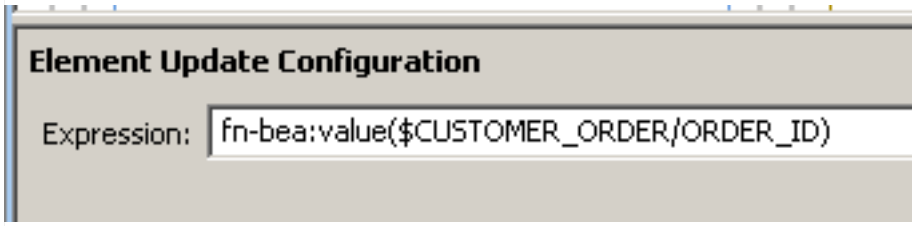
This page last changed on Mar 11, 2008.

This topic describes how to edit XQuery expressions in the expression editor in Studio.

Overview

You can edit the generated XQuery expressions in an update map using the expression editor.

The Expression Editor in an Update Map



The update map expression language is a subset of XQuery syntax. In an update map, you can use any of the following XQuery constructs.

Type	Description	Example
Variable	A variable already defined in a For Each or Update block in the update map. $$root is a special predefined variable that refers to the root of the service's XML type.	$ORDER_WITH_LINE_ITEM $CUSTOMER
Constant	A numeric, string, or other constant.	"a" "12345"
Constant Cast	A constant cast to another XSD data type using the parentheses operator.	xsd:date("2007-01-01")

Function	A call to any XQuery function. You can see the built-in and BEA-provided functions in the Design Palette. You can use a variable, path, or constant as an argument to a function.	fn-bea:value($CUSTOMER/FIRST_NAME)
Path	An expression that locates an XML element in a tree using variables, elements, and attributes. The syntax is: $VARIABLE_NAME /elementName @attributeName	$ORDER_WITH_LINE_ITEM/CUSTOMER_ORDER/ ORDER_ID

Namespace prefixes are declared in the data service's XQuery source, which you can see in the Source tab. If a namespace is only used in the update map, and not in the logical data service, you must declare it. If a namespace cannot be resolved, it is shown with the prefix ns?.

The most common ways you use the expression editor are to:

- Add a constant to an unmapped element
- Cast a constant to an XSD data type, especially to resolve update block elements with no mappings
- Use an XQuery function available in the Design Palette to cast a value
- Use a custom XQuery cast function you have written

The fn-bea:value Function

A mapping between an element in a return type and an element in an update block uses the fn-bea:value function with a path name, for example:

```
fn-bea:value($CUSTOMER/CUSTOMER_ID)
```

An update mapping should always use fn-bea:value, whether ALDSP auto-generates the mapping or you draw it. If you remove the fn:bea:value function from the expression and simply use an XQuery path expression ($CUSTOMER/CUSTOMER_ID), the element becomes disabled in the update map and you see this error message:

The expression does not match the expected type for this element
The expression assigned to this element is not valid
Hint: did you forget to use the value function?

The fn-bea:value function is required, because an update map updates a Service Data Object (SDO) and requires a special XML structure called a datagraph that includes a change summary showing both the old and new values. The fn-bea:value function handles the update to the SDO correctly.

If you do not use fn-bea:value, ALDSP throws an exception when you attempt to update the value.

See Also

Concepts

- Understanding Update Maps

How Tos

- Handle Unmapped Required Values (includes Cast a Constant)
- Cast Using a Built-In XQuery Function
- Cast Using a Custom XQuery Function

Other Sources

- W3Schools XQuery Tutorial

Use the Source Editor

This page last changed on Feb 26, 2008.

This section describes the ALDSP Source editor and highlights its editing features. The following topics are included:

What is the Source Editor?

The Source editor is available from a tab in the ALDSP Eclipse perspective. As you build up your data service, the underlying source is always available from this editor.

Data service source typically:

- References a schema as the data service's XML type (for Entity data services).
- Defines functions in the data service.
- Declares namespaces for referenced data services.
- Contains pragma directives to the query engine.

In addition, data services created from physical data sources contain physical source metadata. For example, data services based on relational data describe the XML type (such as xs:string), the XPath, native size, native type, null-ability setting and so forth.

In developing data services there are many occasions when it is necessary or convenient to view and/or modify source.

The Source editor allows you to directly edit data service source code, as well as schemas. Changes to source are immediately reflected in other data service modes such as the Query view editor; similarly, source is immediately updated when changes are made through the Query editor or in Overview mode.

When a data service is created the root level of your dataspace has "ld:" as its namespace. ld referred to the original name of ALDSP, Liquid Data.

```
declare namespace ns4= "ld:Update/PhysicalDSs/SDO_WLCO_SET";
```

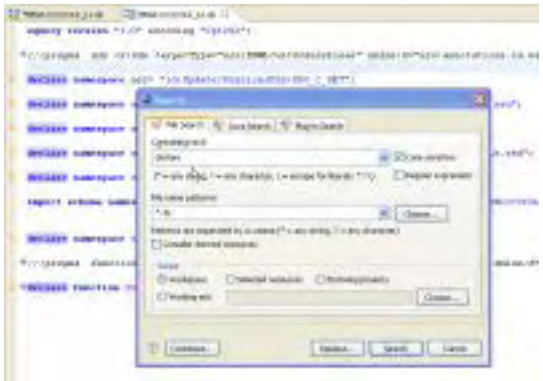
Data Service Annotations

Searching Source

Eclipse offers several types of search.

- You can find all occurrences of a string in Source view using Eclipse Search menu. Each instance of the term in your project will be highlighted.
- You can use page search (Ctrl-F). search to find the next occurrence of a term. Standard search/ replace functionality is available.

File Search in Eclipse



Navigating to Specific Functions

To open Source View to a particular query function in Overview mode, first select the function, then click the Source tab.

Color Coding

XQuery documents in Source View are color-coded according to the following scheme:

Color	Meaning
Blue	Keywords
Dark gray	Comments
Magenta	Variable
Dark green	XML markup
Red	Error conditions

Code Completion

Code completion is available for XPath built-in and user-defined functions. Similarly, function completion is invoked when you type a namespace prefix followed by a colon.

Function Completion from Namespace

XPath Completion in Source View

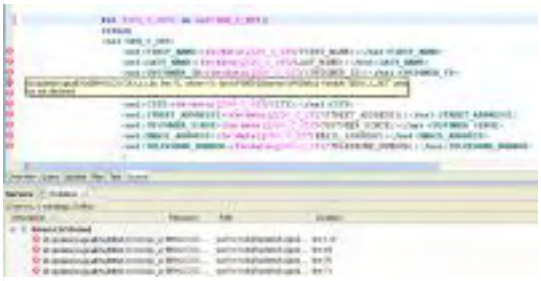
Selecting from Available XQuery Functions

Error Identification

Syntax errors that occur in source either as a result of editing or as a result of changes made in the XQuery Editor appear in the Problems tab.

Windows > Show View > Problems

Induced Error Condition in Source View



Tips

- Click on the error condition in the Problems tab, your cursor will be placed on the relevant line of code.
- Mouse over the error indicator in the Source editor, the complete error condition will appear.
- Right-click on the left margin of Source view several options appear including the option to make line numbering active.
- Right-click anywhere in Source view to access Source editor Preferences including permanently displaying line numbering.

XQuery Language Version Support

This page last changed on Feb 26, 2008.

ALDSP supports the XQuery language as specified in *XQuery 1.0: An XML Query Language, W3C Working Draft of July, 23, 2004*. You can use any feature of the language described by the specification.

ALDSP supplements the base XQuery syntax with a set of elements and directives that appear in Source View as pragmas. Pragmas are a standard XQuery feature that give implementors and vendors a way to include custom elements and directives within XQuery code.

The BEA implementation of XQuery also contains some extensions to the language and additional functions. BEA extensions to XQuery and links to W3C documentation are described in the XQuery and XQSE Developer's Guide.

XQuery Functions

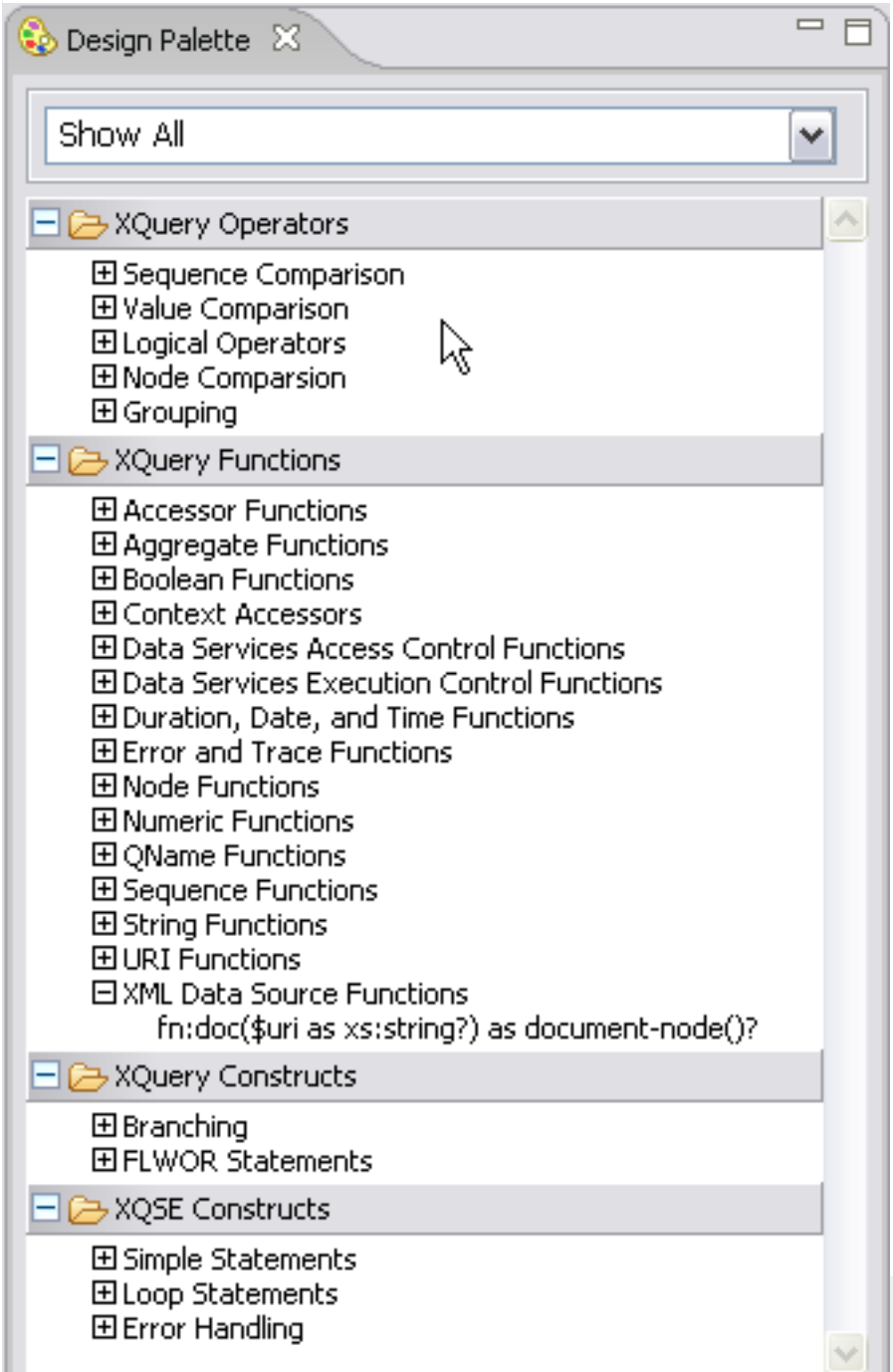
This page last changed on Mar 11, 2008.

Studio provides numerous XQuery functions in the Design Palette. If it is not visible you can access it with:

```
Window > Show View > Design Palette
```

XQuery functions can be utilized in both Query and Update Map views.

XQuery Functions in the Design Palette



- For information on fn-bea XQuery functions, see the XQuery and XQSE Developer's Guide.
- For information on standard XQuery functions, see the W3C XQuery 1.0 and XPath 2.0 Functions and Operators specification.

Testing Data Services

This page last changed on Feb 26, 2008.

Testing Data Services

Concepts

Test Update Procedures Using SDO Data Graphs

How-to...

Test a Read Function and Simple Update

Related Topics

How-to...

Test a Create or Delete Procedure

Test an Update Map Cast

Test an Update Procedure

Enable Optimistic Locking of Relational Objects

Brief Overview of Service Data Objects

This page last changed on Mar 11, 2008.

Test Update Procedures Using SDO Data Graphs

This topic is a brief overview of Service Data Objects and data graphs, which you use to test update procedures in Studio.

Key Points

- To test an Update procedure in Test view, you must submit a data graph as an argument.
- A data graph is an XML structure that contains the data you are changing, as well as the original data.
- When you update a relational data source, ALDSP uses optimistic locking. The data source is locked at update, not when the data is initially retrieved.

Updates in Test View

When you test an Update procedure in Test view, you are actually updating a Service Data Object (SDO) from within Studio.

Selecting an Update Procedure in Test View

SDO is a programming model for Java platforms that unifies data programming across many types of data sources. SDO is based on data objects, which are simply object instances that contain data. You can update the data objects using either static or dynamic data APIs. With a static API, the shape of the data is defined in advance. However, with a dynamic data API, you can update properties at run time that are not known at development time.

The SDO model is based on data graphs, which are collections of tree-structured data, usually XML. A client retrieves a data graph from a data source, modifies it, and applies the data graph back to the data source.

A data graph contains a <changeSummary> element with the original data you are updating. It also contains an XML element with the new data. When both the old and new data are passed back to the data object, the object can be updated.

A Data Graph with Old and New Data

```
<sdo:datagraph xmlns:sdo="commonj.sdo">
```

```
    <changeSummary>
      <sim:SIMPLE_CUSTOMER sdo:ref="#/sdo:datagraph/sim:SIMPLE_CUSTOMER" xmlns:sim="ld:logical/SimpleCustomer">
        <CUSTOMER_SINCE>1999-01-01T00:00:00</CUSTOMER_SINCE>
      </sim:SIMPLE_CUSTOMER>
    </changeSummary>
    <sim:SIMPLE_CUSTOMER xmlns:sim="ld:logical/SimpleCustomer">
      <CUSTOMER_ID>CUSTOMER7</CUSTOMER_ID>
      <CUSTOMER_SINCE>2007-11-11T00:00:00</CUSTOMER_SINCE>
    </sim:SIMPLE_CUSTOMER>
  </sdo:datagraph>
```

Optimistic Locking

When an SDO updates a relational source, it uses optimistic locking to avoid change conflicts. With optimistic locking, the data source is not locked after the client acquires the data. Later, when an update is needed, the data in the source is compared to a copy of the data taken when it was acquired. If any of the underlying data was changed before the client applies the changes, the update is rejected, and the client must recover.

The optimistic locking policy is set for each relational data source.

See Also

How To

- Test an Update Procedure
- Enable Optimistic Locking in Relational Objects

Concepts

- Data Programming Model and Update Framework (in depth, for client applications)

Other Resources

- Introducing SDO

Test a Read Function and Simple Update

This page last changed on Mar 11, 2008.

This topic describes how to test any Read function in an entity data service (either logical or physical) in Test view.

Adjust Settings

A *read function* fetches data.

A physical or logical data service usually has at least one read function. You can test a read function with either sample data or real-time data, from the Test tab in Studio. Testing a read function ensures that:

- You can read data from the physical data sources.
- The returned data has the structure you want, especially for a logical data service.
- The query performs well.

The returned data is displayed in the Test tab.

Checking the Data Returned by a Read Function

When you test a read function, Studio deploys the service to the server if it is not yet deployed, or if it has changed since it was last deployed.

A read function allows you to specify various settings.

Settings You Can Change for the Read Function

You might want to test with a smaller result set. Select

> Limit Elements in Array Results To

to limit the result to a specific number of elements of the return type, to specific child elements in the return type, or both.

Use Start Client Transaction to query multiple relational sources using XA transaction drivers (if selected, ALDSP uses the Required transaction mode to query data sources; if not, ALDSP uses the NonSupported EJB transaction method).

Run the Read Function

A read function optionally can have parameters. It is quite common for a read function to have no parameters, with a name something like read() or ADDRESS().

If you are working with a logical service, the update map must be completely enabled before you can run a read function in Test view. If the update map has yellow update blocks or disabled procedures  , you must resolve them before you can test a Read function.

To test a Read function:

1. Open an entity data service in Studio.
2. Click the Test tab.
3. At Select Operation, choose the name of the read function you want to test.
4. Expand the Settings tab.
5. Choose values for Limit Elements in Array Results To, Start Client Transaction, and Use Default Authentication.
6. Click Run.

   If the results are correct, you see this: 

. You can now click Tree, Text, or Tabular to inspect the returned data.

Perform a Simple Update

The easiest way to test that you can update a data source is to use the Edit and Submit buttons in Test view.

Before you test an update, be sure that:

- The service has a primary update procedure. You can check this by right-clicking an update procedure in the Overview tab and making sure Primary is selected. You can also check for

  ```
  isPrimary="true"
  ```

  in the procedure's pragma statement in the Source tab, for example:

  ```
  (::pragma  function <f:function kind="update" visibility="public" isPrimary="true" xmlns:f="urn:annotations.
  ld.bea.com"> ... ::)
  ```
- The update map is fully enabled for a logical entity service that has an update map. If the update map is not, see Recognize When Something is Wrong. If you are updating a logical data service, you are actually testing an update map. The update will work on any underlying data sources that you have permission to update.

To update data after running a Read function:

1. Click Edit, and edit the field you choose.
2. Click Submit.

   If the data is submitted correctly, you see this message:

   ```
   The data has been submitted.
   ```

See Also

Concepts

- Recognize When Something is Wrong

How Tos

- Test an Update Procedure
- Enable Optimistic Locking

Understanding Query Plans

This page last changed on Feb 26, 2008.

Understanding Query Plans
Concepts Understanding Query Plans

Query Plan Overview

This page last changed on Feb 26, 2008.

You can obtain a query plan for any function in your data service. Simply select the Query Plan tab and select a function, just as you would in Test View. In addition, as a convenience, you can obtain an ad hoc query plan for XQuery or SQL.

Using Query Plan View

The interface for Query Plan View is quite similar to that used for testing your query functions. You select a function or procedure from a drop down list and then click the Show Query Plan button.

A query plan identifies the following query components:

- Joins
- Outer join
- Select statements
- Data sources
- Custom function calls
- Order-bys
- Remove duplicates

There are several ways that a query plan can be viewed:

- **Tree view.** A collapsible graphical presentation of the query plan.
- **Text view.** Presents the information as text.

custOrdersItems Query Plan

Query Plan Information and Warnings

The query plan shows both informational and warning messages. When a section of the plan is flagged with a warning, the plan segment is highlighted in red. If you mouse over the segment, the warning message appears.

Informational messages also can appear with plan segments. Such segments are highlighted in yellow.

Informational and Warning Messages Associated With Query Plans

Warning Message Type

- **XQuery compiler: Typematch.** Typematch issues will be resolved by the compiler (may affect performance)
- **XQuery compiler: No where clause.** There is no predicate associated with the query function (will affect performance).
- **XQuery compiler: Untyped data.** Possible untyped atomic data found in the node constructor.
- **XQuery compiler: No such element.** The element (name provided) is not found in in-scope schemas.
- **SQL generation: missing key.** Underlying table/view does not have a key.

Informational Message Type

- **Audit.** Auditing has been set for this particular function (will affect performance).
- **Cache.** Function is cached (may enhance performance).
- SQL pushdown generation details.

- **SQL generation: cannot generate subquery.** isSubquery property is set to false on the data service. (See the "Function Annotations" section of the Understanding Data Services Platform Annotations section of the XQuery Developer's Guide.
- **SQL generation: cannot generate SQL for join expression.** Unable to translate join condition.
- **SQL generation: cannot generate SQL for aggregate expression (named).** Function does not operate on a sequence.
- **SQL generation: fn:string( ) function encountered.** Use xs:string( ) instead since xs:string( ) can be pushed down to the database for processing.

Printing or Saving Your Query Plan

There are two right-click options associated with query plans:

- Prints the plan
- Saves the plan

The default file name for the saved file will appear in the form:

```
<dataServiceName_qp>
```

If you right-click on the root element of the plan, Plan A right-mouse option on the root element in the plan allows you to print a query plan to a printer or a file. Right-click on any node in the plan and select either the print or print to a file option.

If you print to a file the filename will be of type XML. The name of the file will be the function name followed by the letters _qp, as in: getCustomerView_qp.xml
The file can be saved anywhere in your application.

Loading a Previously Saved Query Plan

You can load a previously saved query plan using the following steps:

1. Select

   ```
   Load from file...
   ```

   from the plan drop down box.

2. In the Browse File dialog locate an existing query plan in the current project.
3. Click Open.

The selected query plan will be appear.

Analyzing a Sample Query

Assume a query returns data related to order details after it is passed an order ID and a customer ID.

The following is a "pseudocode" description of the query:

for electronic orders matching CustomerID and OrderID
return order information and ship-to information
for credit card information matching an AddressID
return credit information and bill-to address information
for electronic line item information matching the line item in the order
return line item information

The statements represent mappings or *projections* in the data service. This can be useful when trying to trace performance issues.

The join conditions are identified in the plan as a left-outer join driven by a complex parameter. By definition, joins have left and right sides, each of which can contain additional joins. One of the best uses of the query plan is to see how the query logic works up the various data threads to return results.

Working With a Query Plan

Two options are available in Query Plan:

- **Expand All.** This right-click menu option expands the currently selected element and any children. If applied to the top-most element in the plan, all elements are expanded.
- **Match highlighting.** When you click on a variable name any elements (open or closed) containing a match for that variable are highlighted. This feature helps you trace variables in the query plan.

Identifying Problematic Conditions Through the Query Plan

When you show a query plan for a particular function, you may notice red or yellow highlighting of particular routines. These correspond to warnings or informational messages from the plan interpreter. For example, if a for statement is missing a where clause (potentially leading to slow performance or retrieval of a massive amount of data) a red warning will appear adjacent to the statement. Simply mouse-over the highlighted section of the plan to view the information or warning.

Managing Update Maps

This page last changed on Feb 26, 2008.

Handle Non-Unique Joins

Handle Non-Unique Values

Handle Unmapped Required Values

Testing Update Maps

How-to...

Test an Update Procedure

Related Topics

How-to...

Enable Optimistic Locking

Test a Read Function and Simple Update

Understanding Update Maps

This page last changed on Mar 17, 2008.

An update map allows you to easily update your logical entity data service without having to write Java or XQSE code. This overview provides a foundation for understanding what an update map is and how you can use one.

BEA XQuery Scripting Extension (XQSE)

ALDSP generates a default update map automatically when you create a *logical entity data service* with a primary read function. You can see the update map associated with a data service by clicking the "Update Map" tab at the bottom of the screen (see the example that follows; click to enlarge image).

Example: a simple update map

In this overview, as a running example we use an update map for a data service that joins together customers and orders. This example may be downloaded if you wish to know the details of the data service that are not covered here.

CustomerOrders.zip

The image to the left shows the update map for the data service (CustomerOrders.ds). The orange arrow identifies the location of the "Update Map" tab.

An *update map procedure* is a create, update, or delete procedure that is *implemented* by an update map. The update map *maps* values from the input to the update map procedure to the inputs of the procedures in the underlying data services. These underlying data services that the logical entity data service is composed of are referred to as the *source data services*. In the previous example, the input is mapped to the two source data services `CUSTOMER` and `ORDERS`. The blue arrows in the update map show how the values are mapped.

A logical entity data service has a *target type* that describes the entity that the data service is about. All read functions in the data service must return instances of the target type and all update map procedures must accept instances of the target type as input. For example, say that we have an entity data service about customers. The read functions of this data service must return customers and update map procedures must take customers as input.

The Target Box

The target box displays the data type of the input to the update map procedures and the procedure icons. There is always exactly one target block in an update map and it is displayed on the right.

Example: the target box

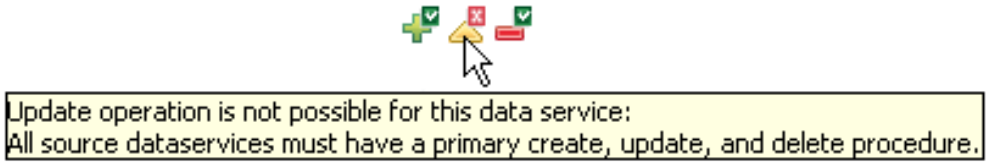
The image to the left shows the update map for the data service CustomerOrders.ds. The target box is identified by the orange rectangle on the right. The procedure icons are in the upper-right corner of the target box. In this case they all contain a green check which indicates that the create, update, and delete update map procedures will function correctly. The input type to the procedures is shown below the procedure icons.

The input type

The input type (or, target type) is the type of the data that is passed to the update map procedures. Elements and attributes from the input type are mapped to the update blocks on the left.

Procedure icons

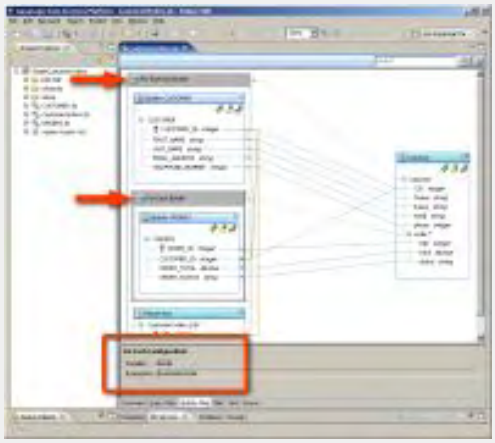
The Create, Update, and Delete procedure icons indicate the status of the corresponding update map procedures. They appear in the upper-right corner of the target box. Each icon may have a green check, a yellow exclamation or a red 'X'. A green check indicates that the update map is fully capable of implementing the procedure. A yellow exclamation indicates that you can invoke the procedure, but there may be problems at runtime. A red 'X' indicates there is a serious problem that needs to be addressed. Any time that there is a red 'X' or a yellow exclamation on the icon, you can hover the mouse pointer over the icon to get a tool tip providing more information (see the image below).



For Each Blocks

A for each block loops over elements in the input to the update map procedure. A for each block is associated with a *variable* and a *path expression*. The path expression defines the sequence to iterate over and the variable binds to elements in the sequence. The variable may be referenced by expressions inside the for each block.

Example: for each blocks



The image to the left shows the update map for the data service CustomerOrders.ds. The two orange arrows identify the two for each blocks in the map. The blocks are grey and have the title "For Each" followed by the variable name.

The second for each block titled "For Each $order" is currently selected (to select a for each block, click on it). The orange rectangle identifies the properties of the currently selected for each block. In this case we see that it defines the variable `$order` which iterates over the sequence `$customer/order`. `$customer` is a variable defined by the upper for each block.

Update Blocks

An update block invokes the *primary* create, update, or delete procedure of a source data service. It will invoke a procedure every iteration of the for each block that contains it. The contents of the update block represent the type of the input given to the procedure. Each element and attribute in the update block is assigned a mapping expression that determines what its value will be when the procedure is invoked. You can select an element or attribute to view or change the expression that determines what value it receives when the procedure is invoked (see the example below).

Procedure icons

Like the target box, an update block also has a procedure status icon. Here the icons indicate the ability of the update block to propagate creates, updates, and deletes to the underlying data service. Otherwise, the meaning of the icons is the same as it is for the target box.

Output variable

A primary create procedure may return a key. If the update block invokes a primary create procedure, it will bind the returned key to the *output variable* (also referred to as the *key variable*). The purpose of having the output variable available is for cases when the key value is generated automatically by an external source but is not part of the input. For example, your source data service is a wrapper for a customers relational database table. Say that the key of this table is an attribute `CUSTOMER_ID` which is an auto-generated number. If you are inserting a customer and some orders at the same time, you may need the auto-generated value for `CUSTOMER_ID` to pass to the input of the create procedure for `ORDERS`. When an update block results in the underlying update or delete procedure being invoked, the output variable will bind to the empty sequence.

Condition

An update block can optionally have a condition. The condition is a Boolean expression that determines if the update block should be invoked or not. If there is no condition, then the update block will always be invoked (see the example below).

Dependencies

When two or more update blocks appear as siblings within the same for-each block, it may be desirable to specify dependencies between them (e.g., due to referential constraints), so an update block can also include a list of dependencies. If update block A depends on update block B, update block B will execute before update block A in the case of a create or update operation (and in the opposite order in the case of a delete). Dependencies between update blocks that are not within the same for each block are not necessary, as the execution of an update map is implicitly outside-in.

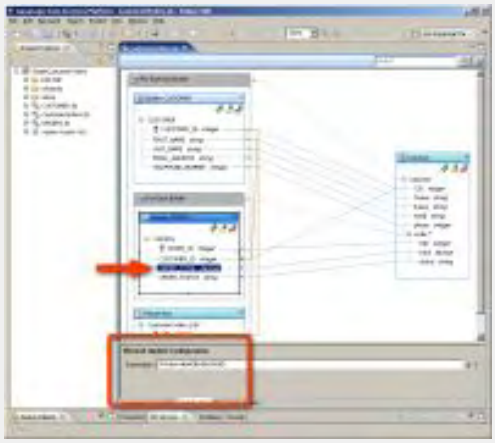
Disabling an update block

An update block can be disabled so that it will never be invoked at runtime. You can disable an update block by right clicking on it and selecting "Disable" The update block should then appear yellow instead of white to indicate that it has been disabled. Disabling an update block is effectively the same as adding a condition that is always false.

Example: update blocks

The images to the left show the update map for the data service CustomerOrders.ds. In the first image, the two orange arrows identify the two update blocks in the map. One update map is for the source data service CUSTOMER and the other is for ORDERS.

In this case, the ORDERS update block is selected and its details are identified by the orange rectangle (select an update block by clicking on it). We can see that the output variable for this update operation is $ORDERS_key. The condition is set to `fn-bea:value($order/status) eq "OPEN"` which means that this update block will only be executed when the input element status has the value "OPEN". $order is a variable that is defined by the for each block containing the update block ("For Each $order").



In the second image, the orange arrow identifies the ORDER_TOTAL element of the ORDERS update block. The orange rectangle identifies the mapping expression (`$fn-bea:value($order/total)`) for ORDER_TOTAL which is displayed because ORDER_TOTAL is currently selected. The ORDER_TOTAL element will receive the value of the total element when the source data service procedure is invoked.

The Return Key Block

The key block describes what will be returned by the update map create procedure. If the data service does not have a key specified, then there will not be a key block and there will never be more than one key block for an update map.

Example: the return key block

The image to the left shows the update map for the data service CustomerOrders.ds. The orange arrow identifies the key block in the update map. The key specified for the CustomerOrders data service is the element `CID` so the key block constructs the `CID` element to be returned and uses the output variable of the `CUSTOMER` update block to get the value.

Customization

ALDSP generates a default update map automatically when you create a *logical entity data service* with a primary read function. This default update map is generated based on the primary read function of the data service. As you change the primary read function, the update map will be regenerated automatically. However, if you customize (change) the update map then it will no longer be regenerated. In other words, a customized update map will no longer change along with the primary read function.

There are many different ways to customize an update map. See the following topics for more information:
Change a Mapping
Remove a Mapping
Revert Customizations
Edit XQuery Expressions
Add a Condition to an update block

Example

The image to the left shows the update map for the data service CustomerOrders.ds. The orange arrow identifies the "customized" symbol that appears after something in the update map has been changed. In this case, it is the mapping expression for ORDER_STATUS that has been modified.

Attachments:

- Customer Orders Update Map.png (image/png)
- The Target Box.png (image/png)
- Hover Over Procedure Icon.png (image/png)
- Red Update.png (image/png)
- Yellow Update.png (image/png)
- Green Update.png (image/png)
- Green Delete.png (image/png)
- Green Create.png (image/png)
- For Each Blocks.png (image/png)
- Update Blocks.png (image/png)
- The Return Key Block.png (image/png)
- Update Block Element.png (image/png)
- Customize.png (image/png)
- cust.png (image/png)

Change a Mapping

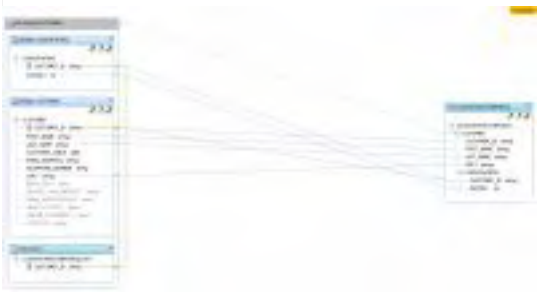
This page last changed on Mar 11, 2008.

This topic describes how to change a mapping in a default update map generated in Studio.

Overview

Once you have generated an update map, you can customize it by adding or removing mappings, changing an XQuery expression, adding dependencies, or changing the return type--all in Studio.

A Sample Update Map



Initially, an update map is generated from the primary read function of a logical data service and changes with the read function.

Once you customize an update map, it is no longer linked to the primary read function. If you change the primary read function after customizing the update map, either in a dialog box or in the Source tab, the update map does not change as a result. To re-link the update map to the primary read function, you must revert customizations.

Example

To change a mapping:

1. Click the Update Map tab.
2. Right-click an existing mapping line, and choose Delete.

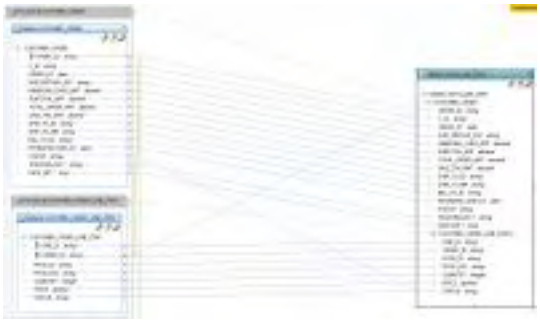
3. Drag from an element in the return type on the right to a new element in a data source on the left.
4. Make sure that the Create, Update, and Delete procedure icons (on both the right and left sides) are still enabled  and not disabled  .
5. Test the new mapping in the Test tab.

The CustomerOrderLineItem Service

In this service, you can draw a new mapping between elements of the same type.

1. Click the Update Map tab.
2. Right-click the mapping line between CUSTOMER_ORDER/STATUS in the return type and CUSTOMER_ORDER/STATUS in the update block, and choose Delete.

3. Drag a new mapping from CUSTOMER_ORDER_LINE_ITEM/STATUS, the child element in the return type, to CUSTOMER_ORDER/STATUS in the update block.
   These elements have the same data type.



4. Make sure that the procedure icons are enabled.
5. Click CUSTOMER_ORDER/STATUS on the left, and check the new mapping in the expression editor.

6. Click the Test tab.
7. Run a read function, then click Edit.
8. Choose a CUSTOMER_ORDER element, then change the value of the first

CUSTOMER_ORDER_LINE_ITEM/STATUS child element.

9. Click Submit.
10. Run the read function again, then check that the value of CUSTOMER_ORDER/STATUS has changed.

In this example, the child element (CUSTOMER_ORDER_LINE_ITEM) has a multiple cardinality, while the parent element (CUSTOMER_ORDER/STATUS) has a single cardinality. You can see this by checking the XML return type in the Overview tab. By default, the first child element value is read to update the data source. You can override this behavior by adding a dependency or writing a custom update function.

When you map one element to another, be sure that the elements have the same or compatible data types. To be compatible, data types must be in the same type hierarchy in the XML Schema DataTypes specification, such as xs:integer and xs:decimal. These types are cast automatically. If you draw a mapping between two elements of different types and hierarchies, you must cast one data type to the other, using a built-in cast function or a custom cast function.

See Also

Concepts

- Understanding Update Maps

How Tos

- Create Your First Data Services
- The XQuery Expression Editor

Remove a Mapping

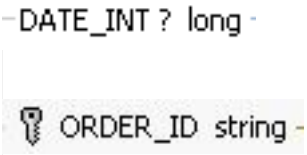
This page last changed on Feb 26, 2008.

This topic describes how to remove a mapping from an update map.

Overview

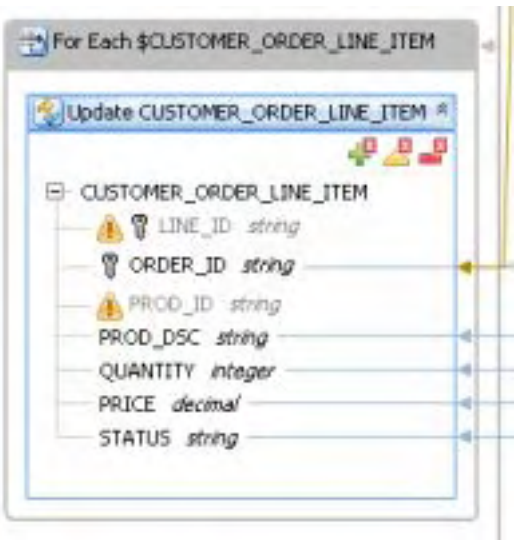
An update map shows mappings for required, optional, and key elements. In an update map, optional elements are displayed with a question mark, and key elements with a key symbol. A key element is usually required.

Optional and Key Elements in an Update Map



If you remove a mapping from a key element, it becomes disabled with a warning icon .

Mappings to LINE_ID and PROD_ID Deleted



Removing a mapping might also cause create, update, or delete procedures to become disabled

 . However, you can correct either of these conditions, by handling unmapped required values.

Example

If you need to remove a mapping, you can do so in either the update map or query map.

To remove a mapping in the update map:

1. Click the Update Map tab.
2. Right-click the mapping line, then choose Delete.
3. If the element becomes disabled in the update block on the left, resolve it.

To remove a mapping in the query map:

1. In the Query Map tab, right-click the mapping, then choose Delete.
2. Handle any required unmapped values in the update map.

See Also

Concepts

- Understanding Update Maps

How Tos

- Change a Mapping
- Revert Customizations
- Handle Unmapped Required Values

Revert Customizations

This page last changed on Feb 26, 2008.

This topic describes how remove anything you have changed in an update map, regenerating the update map from the primary read function.

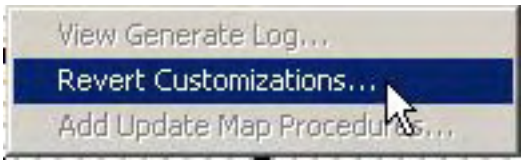
Example

You can undo all changes you have made to an update map. Undoing changes creates a new update map, generating it from the primary read function. When you choose Revert Customizations, all changes you have made to the update map are lost, even changes that you have previously saved.

If the update map had errors or warnings that your changes corrected, the errors or warnings will reappear.

To undo changes and generate a new update map:

1. Click the Update Map tab.
2. Right-click and choose Revert Customizations.



3. Correct any warnings, errors, or disabled procedure icons that appear.

See Also

Concepts

- Understanding Update Maps

How Tos

- Change a Mapping

- Remove a Mapping
- Recognize When Something is Wrong

Add a Condition

This page last changed on Feb 26, 2008.

This topic describes how to add a condition to an update block in an update map.

Overview

In the update map, you can override an Update block by defining conditions in the expression editor that determine when the block is updated.

A Condition in the Expression Editor

A condition is a Boolean expression based on XQuery functions and values defined in the update map, for example:

```
fn-bea:value($CUSTOMER/CUSTOMER_ORDER/TOTAL_ORDER_AMT) > 1000
```

For example, you might have a logical data service with a return type that combines Customer, Order, and CreditRating data. Each customer can have multiple orders and one credit rating.

Return Type with Customer, Order, and CreditRating Data

Example

In the update map, you may want to set a condition that a customer's credit rating can only be updated if the customer places an order with an amount greater than 1000.00.

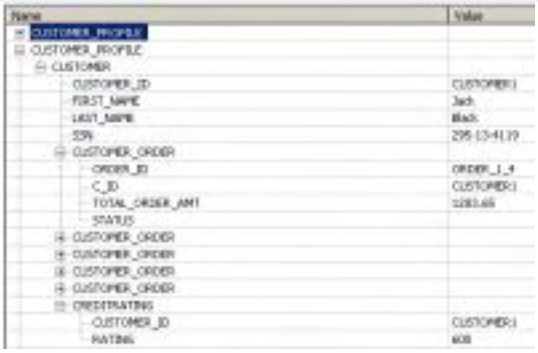
To set an update map condition:

1. Click the Update Map tab.
2. Click the update block on the left that contains the element for which you want to set the condition (for example, the CREDITRATING box for the CREDITRATING/RATING element).
   You can now enter a condition in the expression editor.

3. Enter a condition in the Condition box, for example:

```
fn-bea:value($CUSTOMER/CUSTOMER_ORDER/TOTAL_ORDER_AMT) > 1000.00
```

4. Save the data service.
5. Click the Test tab.
   The logical data service returns data that looks like this:



6. Run a read function, then click Edit and attempt to submit a value for the element that has the condition.

   When you test the update map, you can only update the CREDITRATING data source if TOTAL_ORDER_AMT for any of the customer's orders is greater than 1000.00.

See Also

Concepts

- Understanding Update Maps
- The XQuery Expression Editor

How Tos

- Test a Read Function and Simple Update

Other Resources

- W3Schools XQuery Tutorial

Add Update Map Procedures

This page last changed on Mar 11, 2008.

This topic describes how to add a create, update, or delete procedure to a logical entity service.

Overview

In a logical entity service, you can add create, update, and delete procedures (called update map procedures) that act on underlying data sources. A procedure is an operation that can have side effects, for example, a create procedure that adds a new record to a database table and returns a key value.

You can create update map procedures visually in Studio and have the framework generate XQuery pragma statements and source code, or you can write the source code directly in XQuery or XQSE.

The XQuery pragma statement looks something like this:

```
(::pragma  function <f:function kind="create" visibility="public" isPrimary="true" xmlns:f="urn:annotations.
ld.bea.com">
```

This statement defines a create procedure, with public visibility, that is primary. Even though the pragma statement uses the keyword function, the operation you define is a procedure, as you can see from the declaration:

```
declare procedure cus:createCustomerAndAddress($arg as element(cus:CustomerAndAddress)*) as element(cus:
CustomerAndAddress_KEY)\* external;
```

This line declares the procedure with the name createCustomerAndAddress, defines one argument with the service's return type, and specifies a key as a return value.

Generate Default Procedures

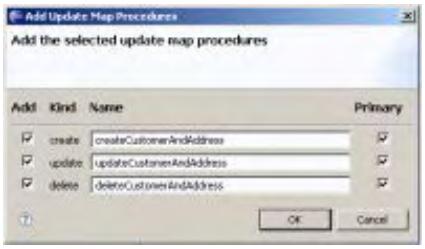
When you generate default update map procedures, they have these parameters and return values:

Type	Parameters	Return Value
Create	The service's Return type	The current key, empty if no key is defined
Update	The service's Return type using a changed-element kind	Empty
Delete	The service's return type	Empty

Before you create update map procedures, especially create procedures, add a key to your service. A primary create procedure must return a key. Primary update and delete procedures require the Return type as an argument; their non-primary equivalents can be written to accept a key instead.

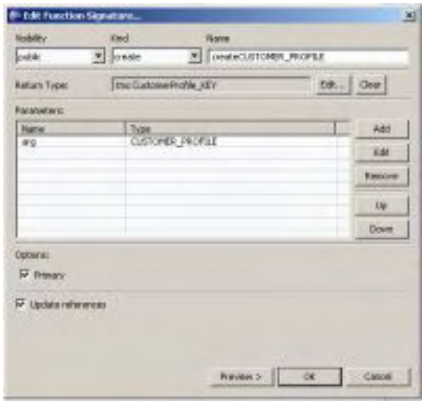
To generate a default update map procedure:

1. Create a key for your service.
2. In the Overview tab, right-click at the left, right, or top, and choose Add Update Map Procedures.



3. Select Add to indicate which procedures to add.
4. Add names in the Name fields.
5. Mark Primary to indicate if each procedure should be primary.
6. Click OK.

7. In the Overview tab, right-click a procedure name and choose Edit Signature.



8. Make any necessary changes to the procedure signature in the dialog box.

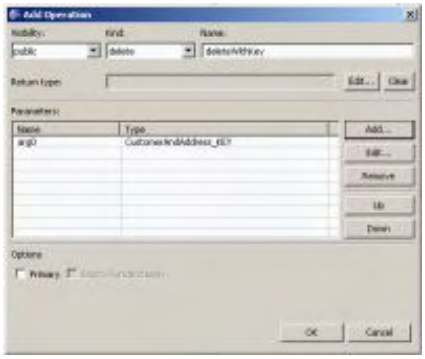
Design Custom Procedures

You can also create procedures with the arguments and return types you choose. This is useful for procedures in addition to the

primary create, update, and delete procedures.

To design custom procedures:

1. Click Overview.
2. Right-click at the top, left, or right, and choose Add Operation.
3. Choose a value for Visibility.
4. At Kind, choose create, update, or delete.
5. At Name, enter a procedure name.



6. (Optional) At Return Type, click Edit. Choose a primitive or complex type, then click OK.
7. At Parameters, click Add.
8. Choose a primitive or complex type from an XML or XSD file, then click OK.
9. At Kind, choose a value.
   Choose element to use the exact XML element you selected as a parameter; changed-element, if values in the element must be updated; schema-element, if the element must be validated according to an XML schema.
10. At Occurrence, choose a value.
11. Click OK in both dialog boxes.

See Also

Concepts

- Data Service Keys

How Tos

- Add a Read Function
- Create Logical Data Service Keys
- Test a Create or Delete Procedure
- Test an Update Procedure

Update Map Functions

This page last changed on Mar 17, 2008.

Update Map Functions

The following functions are useful in update map expressions (e.g. update block conditions and mapping expressions).

fn-bea:coalesce($arg ... as xdt:anyAtomicType) as xdt:anyAtomicType?
The function fn-bea:coalesce takes 1 or more arguments and returns the first that is not empty.

fn-bea:coalesce-equal($arg ... as xdt:anyAtomicType) as xdt:anyAtomicType?
The function fn-bea:coalesce-equal takes 1 or more arguments and returns the first that is not empty. If any of its non empty arguments are not equal then it will throw an exception at runtime. (see also How to update a foreign key values mapped using fn-bea:coalesce-equal)

fn-bea:value($arg as item()?) as xdt:anyAtomicType?
The function fn-bea:value is essentially the same as fn:data except that it additionally indicates to the ALDSP runtime that the variable $arg may bind to a changed-element() depending on the context (like in the case of an update procedure driven by the update map). As a general rule, fn-bea:value should always be used in place of fn:data in update map expressions since they should be written to support all three flavors of update procedures (create, update, and delete).

fn-bea:ambiguous($arg ... as xdt:anyAtomicType) as xdt:anyAtomicType?
The function fn-bea:ambiguous may appear in the default update map when there are multiple target values (on the right) that map to the same source value (on the left). For example, if the same value is projected (i.e. returned) more than once in the primary read function, this may result in fn-bea:ambiguous being used in the corresponding update map by default. It is expected that the user will manually remove the call to fn-bea:ambiguous when resolving the ambiguous mapping. For example, the user may choose to pick one of the arguments in the fn-bea:ambiguous call to be the new mapping expression and disregard the others.

Variable Usage Rules

This page last changed on Mar 17, 2008.

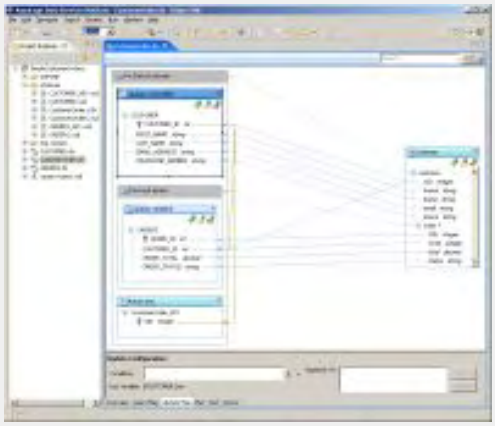
Variable Usage Rules

This section describes how variables may be used when customizing an update map expression. If you are new to update maps, it is recommended that you first read Understanding Update Maps.

Variable Types and Scoping Rules

Variables may be defined by a for-each block or by an update block (as an output variable). An output variable may be used in an expression if the expression is contained within an update block that depends on the update block that defines the variable. A for each block variable may be used by an expression if the expression is immediately inside the for-each block that defines the variable. However, if the defining for-each block contains any other for each blocks which also contain the expression, the variable may not be used in the expression.

Example: an update map for a customer-orders data service



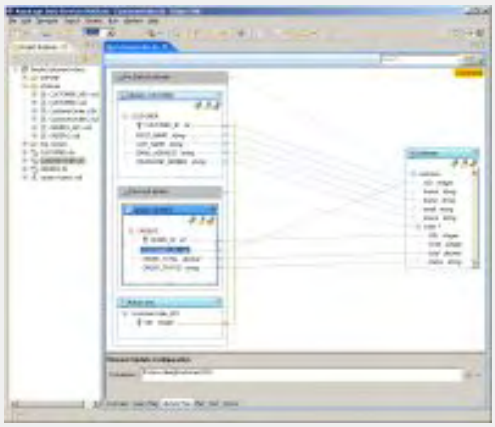
The image to the left shows an update map for a logical data service about customers and orders (see Understanding Update Maps for more info). Notice that the update block for customers is selected and the name of its output variable `$CUSTOMER_key` is shown at the bottom of the screen (the output variable is also referred to as the key variable). In this case, `$CUSTOMER_key` can be referenced anywhere from within the update block for orders (e.g. in a mapping expression or in the condition for the update block).

Also notice that we have two for-each blocks that define the variables `$customer` and `$order`. Within the update block for customers, only the `$customer` variable is visible and within the update block for orders only the `$order` variable is visible (with one exception which will be discussed next).

These restrictions are in place to prevent unintuitive or complicated behavior by the update map at runtime. If these rules are too restrictive for your application, you may want to consider using XQSE. However, there is one exception to the for each block variable usage rule. If the variable is used as part of a path expression that references a key value, then usage is valid as long as the expression is within the defining block. It is **important to**

note that updates will be effectively disabled for such "outside" mappings (creates and deletes will still work).

Example: an outside mapping to a key value



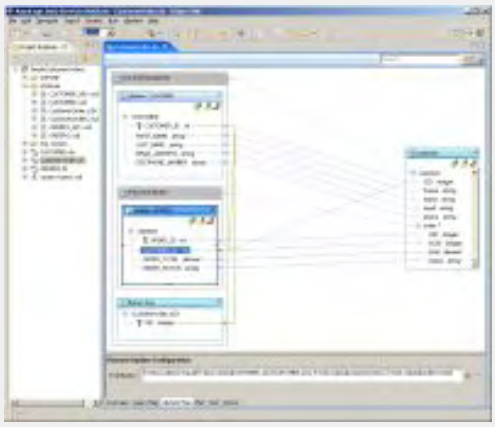
In the image on the left, the CUSTOMER_ID element in the update block for orders is selected and its mapping expression (fn-bea:value ($customer/CID)) is shown at the bottom of the screen. Normally, the variable $customer could not be used within the orders update block since it is not directly contained within the corresponding for-each block. However, since it is being used to reference CID it is allowed because CID is equivalent to the key value of the customer data service. Updates for this outside mapping will be disabled. That is, if customer/CID is modified in the input to the update map procedure, the CUSTOMER_ID element in the orders update block will not have the modified value.

Updating Foreign Key Values

The function fn-bea:coalesce takes 1 or more arguments and simply returns the first argument that is not empty. The function fn-bea:coalesce-equal works the same way except that it additionally checks that all non empty arguments are equal. If it finds that any two non empty arguments are not equal, it will throw an exception at runtime. The automatically generated update map may use fn-bea:coalesce-equal for the mapping expression for foreign key values if it can be inferred from the target data service that the values should always be equal.

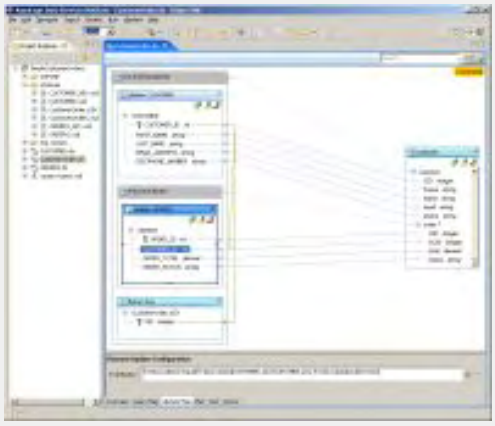
If an argument to fn-bea:coalsece-equal contains an a path expression that falls under the exception to the for each block variable rule mentioned above, then updates will be disabled for the entire expression containing fn-bea:coalsece-equal. If your automatically generated update map is in this situation and you wish to be able to update the foreign key value, you can simply remove the argument that contains the offending mapping. (See the example that follows)

Example: coalesce-equal



In the first image on the left, the CUSTOMER_ID element in the update block for orders is selected and its mapping expression (fn-bea:coalesce-equal(...)) is shown at the bottom of the screen. When the update map is used to create a customer with orders, the value for CID may not be known as it may be auto generated by an underlying relational database. This means that $customer/CID and $order/OCID may be empty. In this case the generated key value will be returned and the orders will get the value for CUSTOMER_ID via fn-bea:value ($CUSTOMER_key/CUSTOMER_ID). If the update map is used to update or delete customers and orders, $CUSTOMER_key/CUSTOMER_ID

will be empty but `$customer/CID` and `$order/OCID` should not be.



This mapping contains an outside reference to the variable `$customer` so updates will be disabled for `customer/order/OCID` in the input to the update map procedure. To enable updates to `OCID`, we can remove the outside mapping. The second image on the left shows what the update map looks like after this modification.

Recognize When Something is Wrong

This page last changed on Feb 26, 2008.

This topic describes why an update map might appear disabled and points you to solutions.

Understand the Symptoms

The signs of a disabled update map appear on the update map itself, in the Generate Log, and in the Problems tab.

In the update map, you may see disabled (or yellow) update blocks. When an update block is completely or partially disabled, updates do not occur in the data source the block maps to.

A Disabled Update Block

An update map procedure that is disabled has a yellow or red status indicator at the upper right.

Disabled Procedure Icons

Create  **Update**  **Delete** 

You might also see a message with a link to view the Generate Log.

A View Generate Log Message

The update map may not provide a complete and unambiguous mapping back to the data sources. View the generate log for details.

Clicking the link displays the Update Map Generate Log window.

Update Map Generate Log

Check the Problems Tab

If you see disabled procedure icons or other symptoms, you should also check the Problems tab for detailed Error and Warning messages. The Problems tab shows errors and warnings that the View Generate Log message does not. For example, this update map shows two errors and three warnings.

The Problems Tab for a Disabled Update Map

Errors prevent you from deploying the update map to the ALDSP server and testing it. Warnings tell you that something is not supported in the update map, but the update will proceed.

To sort the Problems tab, as shown above:

1. Click the Problems tab.
2. Click the triangle icon at the upper right, and choose Sorting.
3. Sort first by Resource, then by Severity and Description.

Resolve Errors and Warnings

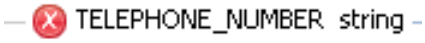
You may have a valid reason to use a certain logical data service design that initially generates an update map with constraints. This is fine. You can find workarounds and resolve most disabled update map conditions.

Disabled Update Blocks

When you encounter a disabled, or yellow, update block, you can right-click it and choose Enable. The most likely reasons an update block is disabled are shown below.

Reason	Meaning	Solution
A non-unique join	A join in the primary read function is non-unique, possibly causing duplicate values in the result.	Handle Non-Unique Joins

Once you enable the update block, you will likely see:

- Elements that have warnings  or are completely disabled 
- Disabled Create-Update-Delete procedure icons 

Disabled Procedure Icons

When you see disabled procedure icons, check the update blocks on the left. The procedure icons in the return type on the right naturally result from those on the left.

In general, the status indicators for update map procedures are:

- Green  if the update map will work at run time as you have designed it, even if parts of it are disabled
- Yellow  if some parts of the update map will work at run time, but you might see run-time errors on other parts
- Red  if the update map will not work at run time

If you want to correct an update map before run time, a red or yellow status indicator on the left can have any of the following meanings. If you mouse over the

Status	Type of Procedure	Meaning	Solution
Red	Create, Update, Delete	The data service does not have a primary procedure of that type.	Create a primary procedure (Overview tab, right-click, Add Update Map Procedures, select Primary)
Red	Update, Delete	The data service does not have a key.	Create Logical Data Service Keys
Red	Create	The update block has missing mappings or mappings of the wrong data type	Understand Mappings with Different Data Types Handle Unmapped Required Values

Red	Create	The return type contains non-element or non-attribute XML items that are not allowed.	Handle an Unsupported Node Constructor Error
Red	Create, Update, Delete	The update block references a variable from another disabled update block.	Right-click the disabled block, and choose Enable. See any topics in See Also.
Red	Update, Delete	The data service has a key, but one or more key fields have missing mappings, mappings of the wrong data type, or mappings to invalid items in the return type.	Handle Unmapped Required Values
Yellow	Update, Delete	The update block has missing mappings, mappings of the wrong data type, or mappings to invalid items in the return type.	Understand Mappings with Different Data Types Handle Unmapped Required Values

See Also

Concepts

- Understand Mappings with Different Data Types

How Tos

- Create a Return Type
- Handle Disabled Procedures in Underlying Data Sources
- Handle Non-Unique Joins
- Handle Non-Unique Values
- Handle Unmapped Required Values
- Understand Mappings with Different Data Types

Understand Mappings with Different Data Types

This page last changed on Mar 11, 2008.

This topic describes casts between elements of different data types in an update map.

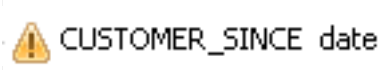
Overview

In an update map, you may need to map elements of different data types between a return type and an underlying data source.

For example, a return type might contain an xsd:dateTime element that maps to an xs:date element in the data source. When data types differ, you need to cast between them in order to enable the update map. Type differences occur because a logical data service design can differ from actual physical data sources or because data types used by an underlying data source are unknown at design time.

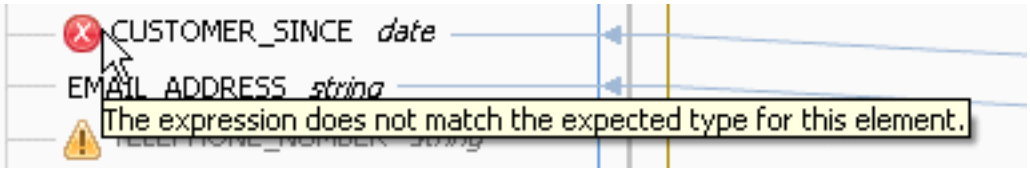
When the update map is first generated, the element in the data source has no mapping and a warning icon.

The Element Initially with No Mapping



If you draw a mapping line in Update Map view, from the xsd:dateTime value in the return type to the xsd:date value in the update block, the element becomes disabled.

An Error Due to Data Type Mismatch



You can fix this type of error by using different techniques to cast, according to the data types you are

casting.

First, review the built-in datatypes chart in the XML Schema Datatypes specification to understand the hierarchies of data types used in XML Schema. The type xs:string and its subtypes belong to one type hierarchy, and the type xs:decimal and its subtypes belong to another.

Casts between elements of different types are handled in one of three ways:

1. **Type promotion**. If both data types are in the same type hierarchy and the cast moves up the hierarchy, ALDSP casts them implicitly. This is known as type promotion. For example, xs:token is promoted to xs:string and xs:integer is promoted to xs:decimal. Implicit casts are implemented in ALDSP according to the XQuery 1.0 specification.

2. **Built-in cast function**. If the types do not use a cast up the same type hierarchy and type promotion does not occur, you can use a built-in XQuery function available from the Design Palette.

3. **Custom cast function**. If a built-in XQuery function is not available, you can write your own custom cast function in the Source tab of your primary logical data service or in a specialized library data service that performs casting.

Built-In Cast Functions

If a built-in function provides the cast you need, you can simply drag it from the Design Palette to the expression editor and enter argument values.

Built-In XQuery Casting Functions

Custom Cast Functions

Before you write a custom XQuery cast function, make sure that XQuery allows the cast you want to

perform. Check the casting section in the XQuery 1.0 specification to understand the rules for casting between types in XQuery, especially the chart that describes casting between primitive types.

Remember these general guidelines:

- The primitive type chart shows which casts can be performed between primitive types. For example, an integer (such as 44) can always be cast to a string ("44"). However, a string can only be cast to an integer in some cases. The string "55" can be cast to the integer 55, but the string "hello" cannot be cast to an integer.

- If both the source and target types are derived from the same primitive type, you can cast between them.

- If the source and target types are derived from different primitive types, you are casting across the type hierarchy. In general, you need to cast the source type up the hierarchy to its primitive type; then, cast from the primitive type of the source to the primitive type of the target; and last, cast from the primitive type of the target to the target type (see the rules in the XQuery 1.0 specification).

Once you write the cast function, you can test it in Studio, before you run it with a client application.

See Also

How To

- Cast Using a Built-In XQuery Function
- Cast Using a Custom XQuery Function
- Test an Update Map Cast

Other Resources

- Built-in datatypes chart
- Primitive types casting chart (scroll down)
- XML Schema Datatypes Specification
- XQuery 1.0 Specification

Cast Using a Built-In XQuery Function

This page last changed on Feb 26, 2008.

This topic describes how to use a built-in XQuery function to cast values of different data types in an update map.

Example

You can cast an element from one data type to another using a built-in XQuery cast function when:

- Type promotion does not occur.
- The data comes from a variable or an other source that is not a constant
- A built-in function that performs the cast you want is available in the Design Palette.

To cast using a built-in XQuery function:

1. Click the Update Map tab.
2. Click the disabled element in an update block on the left.
   In the expression editor, you see an expression that uses fn-bea:value() to map from the return type on the right, for example:

   ```
   fn-bea:value($CUSTOMER/CUSTOMER_SINCE)
   ```

   This expression represents a dateTime value coming from the return type.

3. Open the Design Palette

   ```
   Window > Show View > Design Palette
   ```

4. Expand XQuery Functions, then a category (for example, Duration, Date, and Time Functions).
5. Drag the function you want to the expression editor (for example, fn-bea:date-from-dateTime), leaving the existing expression there.
6. If feasible, use the existing expression as an argument to the function, for example:

   ```
   fn-bea:date-from-dateTime( fn-bea:value($CUSTOMER/CUSTOMER_SINCE) )
   ```

Here the original value is used as the $dateTime argument to fn-bea:date-from-dateTime().

7. Test the update map cast to make sure it works as you expect.

See Also

Concepts

- Understand Mappings with Different Data Types

How To

- Cast Using a Custom XQuery Function
- Test an Update Map Cast

Other Resources

- XQuery Tutorial at W3Schools

Cast Using a Custom XQuery Function

This page last changed on Feb 26, 2008.

This topic describes how to write a custom XQuery function to cast between elements of different data types in an update map.

Example

An example of a custom XQuery cast function is one that casts from integer to string. Suppose the logical data service's return type uses xsd:integer for the TELEPHONE_NUMBER element, while the underlying data source uses xsd:string.

Mapping from Integer to String

The mapping between the two TELEPHONE_NUMBER elements is initially disabled. The value from the return type is something like 4155551212, which can easily be converted between xsd:integer and xsd:string. Check the type casting chart in the XQuery 1.0 specification to make sure the cast you want to perform is allowed.

> When you test the cast function, you also need to perform the opposite cast (in this case, xsd:string to xsd:integer).

To write a custom XQuery cast function:

1. Click the Source tab.
2. Write an XQuery function that takes an argument of the data type you are casting from and returns a value of the data type you are casting to, for example:

```
declare function tns:intToString($theint as xs:integer) as xs:string {
              xs:string($theint)
};
```

Assign your function to an XML namespace your logical data service uses. Be sure both the parameter and return type are valid XML Schema data types. Then, write a statement that performs the cast.

3. In the Update Map tab, click the element in the data source on the left.

   At this point, the element is disabled: — TELEPHONE_NUMBER string –. Its value is taken from the return type, so its XQuery expression looks something like this:

   ```
   fn-bea:value($CUSTOMER/TELEPHONE_NUMBER)
   ```

   Remember that the value from the return type is an xs:integer.

4. Add your new cast function, using the existing expression as its argument, for example:

   ```
   tns:intToString(fn-bea:value($CUSTOMER/TELEPHONE_NUMBER))
   ```

   At this point, the update map should be completely enabled.

5. If the disabled icon on the element does not disappear immediately, click another element in the update map.
6. Test the update map cast to make sure it works as you expect.

See Also

Concepts

- Understand Mappings with Different Data Types

How To

- Cast Using a Built-In XQuery Function
- Test an Update Map Cast

Other Resources

- XQuery 1.0 Specification
- Primitive type casting chart (scroll down)

Test an Update Map Cast

This page last changed on Mar 11, 2008.

This topic describes how to test a cast between elements of different data types in an update map.

Example

The easiest way to test an update map cast function is to use Read-Edit-Submit from the Test tab in Studio.

Suppose you are casting from xs:integer to xs:string. To test the cast function, you need to retrieve data from the data source as xs:string and display it in the Test tab as xs:integer, so you also need to cast in the reverse direction. The primitive types casting chart in the XQuery 1.0 specification shows that you can always cast from xs:integer to xs:string, but you can only cast from xs:string to xs:integer in some cases.

To test an update map cast using Read-Edit-Submit, you first edit the source code of the primary Read function to do a comparable cast when the data is read from the data source. For example, suppose you want to cast from dateTime to date during an update. To test, you must first cast the date value to dateTime when you read it from the data source.

Before you use this test method, check the casting chart in the XQuery specification to make sure the XQuery cast you want to perform works in both directions. In the example given here, the cast is from xs:dateTime to xs:date in the update map and from xs:date to xs:dateTime in the primary Read function. Both casts must be valid in XQuery.

1. Click the Source tab.
2. Locate the primary Read function, which looks something like this:

```
declare function tns:read() as element(cus:CUSTOMER)*{
for $CUSTOMER in cus1:CUSTOMER()
return
        <cus:CUSTOMER>
            <CUSTOMER_ID>{fn:data($CUSTOMER/CUSTOMER_ID)}</CUSTOMER_ID>
            <FIRST_NAME>{fn:data($CUSTOMER/FIRST_NAME)}</FIRST_NAME>
            <LAST_NAME>{fn:data($CUSTOMER/LAST_NAME)}</LAST_NAME>
            <CUSTOMER_SINCE>{fn:data($CUSTOMER/CUSTOMER_SINCE}</CUSTOMER_SINCE>
            <EMAIL_ADDRESS>{fn:data($CUSTOMER/EMAIL_ADDRESS)}</EMAIL_ADDRESS>
            <TELEPHONE_NUMBER>{fn:data($CUSTOMER/TELEPHONE_NUMBER)}</TELEPHONE_NUMBER>
            <SSN?>{fn:data($CUSTOMER/SSN)}</SSN>
            <BIRTH_DAY?>{fn:data($CUSTOMER/BIRTH_DAY)}</BIRTH_DAY>
            <DEFAULT_SHIP_METHOD?>{fn:data($CUSTOMER/DEFAULT_SHIP_METHOD)}</DEFAULT_SHIP_METHOD>
            <EMAIL_NOTIFICATION?>{fn:data($CUSTOMER/EMAIL_NOTIFICATION)}</EMAIL_NOTIFICATION>
            <NEWS_LETTTER?>{fn:data($CUSTOMER/NEWS_LETTTER)}</NEWS_LETTTER>
            <ONLINE_STATEMENT?>{fn:data($CUSTOMER/ONLINE_STATEMENT)}</ONLINE_STATEMENT>
            <LOGIN_ID?>{fn:data($CUSTOMER/LOGIN_ID)}</LOGIN_ID>
        </cus:CUSTOMER>
};
```

3. Locate the element you want to cast and add a XQuery cast expression to it. For example, this casts an xs:date to an xs:dateTime in the CUSTOMER_SINCE element:

```
<CUSTOMER_SINCE>{ xs:dateTime(fn:data($CUSTOMER/CUSTOMER_SINCE)) }</CUSTOMER_SINCE>
```

To cast an xs:string to an xs:integer in TELEPHONE_NUMBER, enter this:

```
<TELEPHONE_NUMBER>{xs:integer(  fn:data($CUSTOMER/TELEPHONE_NUMBER)) }</TELEPHONE_NUMBER>
```

4. Click the Test tab.
5. At Select Operation, choose the service's primary Read function and click Run.
   In the Result pane, you might see that the values have been cast, if the new type looks different.
6. Click a customer record, then Edit.
7. Change one of the values you have just cast.
   If you are working with xs:date and xs:dateTime, change the date portion of the value, rather than the time. The time is truncated when you store the value in the data source as an xs:date. When you read it back as an xs:dateTime, it looks like 00:00:00.
8. Click Submit.
   You should see this message:

```
Data has been submitted
```

9. Click Run again to verify the change.

See Also

How To

- Cast Using a Built-In XQuery Function
- Cast Using a Custom XQuery Function

Other Resources

- XQuery 1.0 Specification

Handle Disabled Procedures in Underlying Data Sources

This page last changed on Mar 11, 2008.

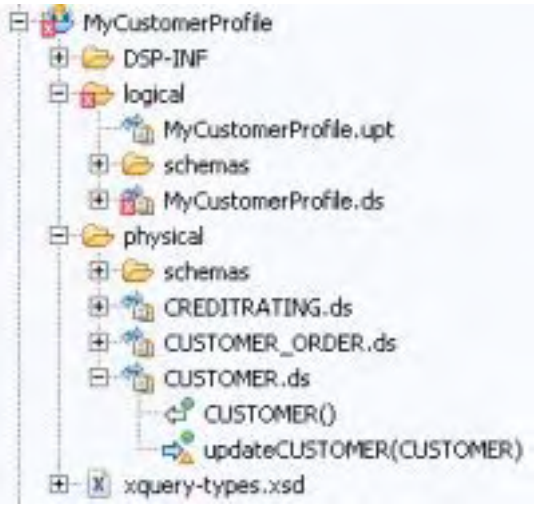
This topic explains how to enable an update map for a logical data service when an underlying data source has disabled procedures.

Check the Data Sources

If a Create, Update, or Delete procedure is disabled in a data source that your logical data service uses, part of the update map is disabled as well. Specifically, the update block that maps to the data source is disabled.

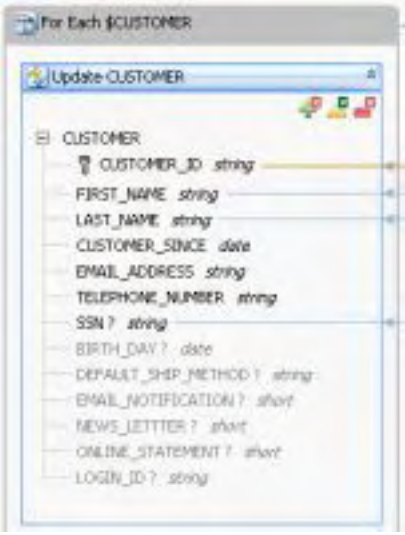
For example, you might have a physical data service that is missing a Create, Update, or Delete procedure.

Physical Data Service with No Create or Delete Procedure



As a result, the update block that maps to this data source has its Create and Delete procedures disabled.

Update Block with Disabled Create and Delete Procedures



When you mouse over the disabled procedure icons, you see tooltips telling you that create and delete operations are not possible for the service.

Resolve the Disabled Procedures

You need to resolve the disabled procedures so that you can deploy the service to the server. To resolve them, you can:

1. Disable the update block that contains the disabled procedures.
2. Enable the procedures in the underlying data source.
3. Change the XML schema of the return type. For example, you can remove the XML element that maps to the disabled data source.

The solution you choose depends on your needs.

Disable the Update Block

If you do not need to use the procedures that are disabled in the underlying data source, you can disable the entire update block:

1. Click Update Map.
2. Right-click the update block, and choose Disable.

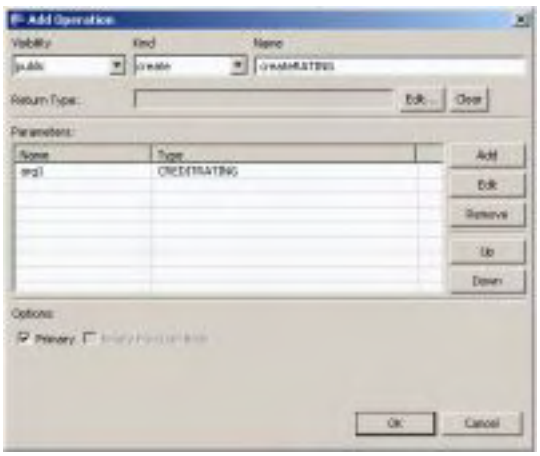
Disabling the block might also disable procedures or key elements in other blocks.

3. Resolve any mappings that become disabled.

Add or Enable Procedures in the Underlying Data Source

You can also enable procedures in or add them to the underlying data source. For example, to add a procedure to a physical data service:

1. Open the physical data service, and click the Overview tab.
2. Right-click near the top, and choose Add Operation.



3. Choose the Visibility and Kind of the procedure, then enter a name.
4. Click Add to add a parameter. Enter a Parameter Name, then choose a Type, Kind, and Occurrence. Click OK.
5. Select Primary if you want the procedure to be primary for its type.
6. Click the Update Map tab.

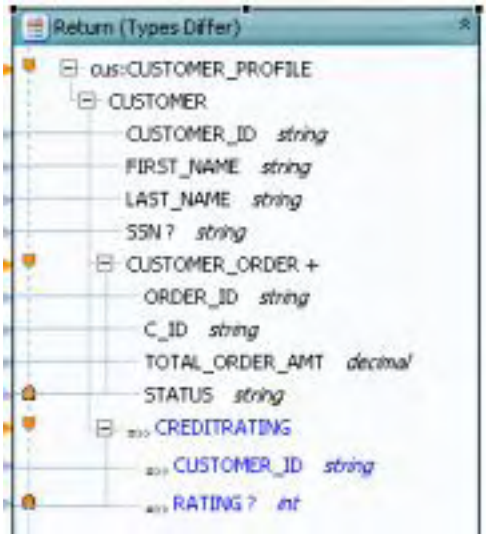
7. Right-click in the update map, then choose Revert Customizations.
   Be sure that the procedures in the update block that maps to the underlying data source is enabled.

Change the XML Return Type

You can also change the XML schema the logical data service uses for its return type. For example, you might remove the element that attempts to update the disabled data source. You can even do this dynamically from Studio.

To change the return type from Studio:

1. Open the logical data service, and click the Overview tab.
2. Right-click the schema, then choose Edit Schema.
3. Remove the entire element, between the <xs:element> and </xs:element> tags.
4. Click the Query Map tab.
5. Right-click the return type, then choose Show Type Difference.
   You should see the removed elements in blue.



6. Right-click the removed element, and choose Remove Element.
7. Click the Update Map tab.
8. Resolve any disabled elements or procedures.

See Also

Concepts

- Understanding Update Maps
- Recognize When Something is Wrong

How Tos

- Test a Read Function and Simple Update

Handle Non-Unique Joins

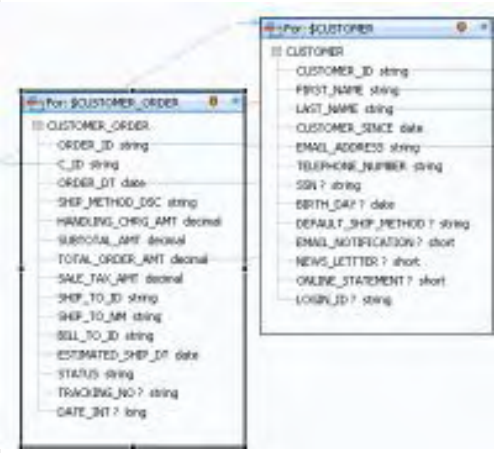
This page last changed on Feb 26, 2008.

This topic shows how to enable an update map when a logical data service uses a non-unique join between relational data sources.

Understand the Join

In a logical data service, you can join tables visually in the Query Map by dragging from a key element in one data source to a corresponding key element in another data source.

Joining Tables in the Query Map



You can also create a join by adding an XQuery WHERE statement in the expression editor or the Source tab:

```
where $CUSTOMER/CUSTOMER_ID eq $CREDIT_CARD/CUSTOMER_ID
```

If both tables are in the same database, the XML return type is nested, and you are joining on a unique key, ALDSP creates a left outer join. You can see the SQL in the query plan for the service (click the Plan tab, then Show Query Plan):

```
SELECT ...
FROM "RTLCUSTOMER"."CUSTOMER" t1
LEFT OUTER JOIN "RTLCUSTOMER"."ADDRESS" t2
ON (t1."CUSTOMER_ID" = t2."CUSTOMER_ID")
```

If the XML return type is flat, ALDSP creates an inner join, and the SQL looks like this:

```
SELECT ...
FROM "RTLAPPLOMS"."CUSTOMER_ORDER" t1
JOIN "RTLCUSTOMER"."CUSTOMER" t2
ON (t2."CUSTOMER_ID" = t1."C_ID")
```

A left outer join returns rows from the left (meaning, the first) table, even if they do not match any rows in the right (second) table.

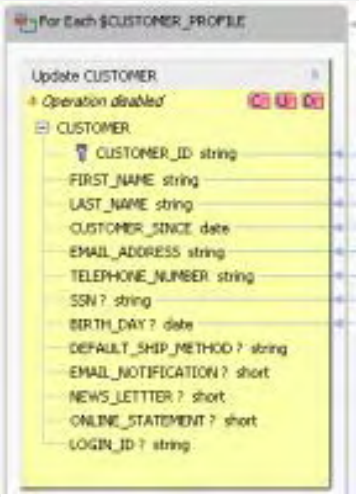
An inner join requires that a value in the left table match a value in the right table in order for the left values to be included in the result. For example, you might match one customer to many orders, creating a joined table like this:

CUSTOMER_ID	FIRST_NAME	LAST_NAME	EMAIL_ADDRESS	ORDER_ID	ORDER_DT	TOTAL_ORDER_AMT
CUSTOMER1	Jack	Black	jack@yahoo.com	ORDER_1_0	2001-10-01	156.39
CUSTOMER1	Jack	Black	jack@yahoo.com	ORDER_1_1	2002-02-17	596.65
CUSTOMER1	Jack	Black	jack@yahoo.com	ORDER_1_2	2002-07-07	656.65

Here, CUSTOMER_ID is a unique key and has one row in the relational source. However, in the joined table, CUSTOMER1 has three orders and three rows. If you update information for CUSTOMER1 such as FIRST_NAME in the joined table, where each customer has multiple rows, the value to use to update the underlying data source is ambiguous.

With a non-unique join, all or part of the update map is temporarily disabled and looks like this:

A Disabled Update Block



When you click View Generate Log in the update map, you see a message like this one:

```
The primary read function has a non-unique join involving this data source.
```

In your function or procedure code, in the Source tab, you might see `for` statements directly nested within each other, without an intervening WHERE clause:

```
for $CUSTOMER in ns1:CUSTOMER()
for $CREDIT_CARD in ns2:CREDIT_CARD()
return
```

Or, you might see XML elements directly nested within each other without intervening SQL statements:

```
<ns7:CUSTOMER_PROFILE>
      <CUSTOMER>
        ...
      {
           <CREDIT_CARD>
              ...
           </CREDIT_CARD>
      }
      </CUSTOMER>
</ns7:CUSTOMER_PROFILE>
```

These are all symptoms of a non-unique join. You need to enable the update map so that you can deploy the service, test it, and make it available to client applications.

In an update map, the most common causes of a non-unique join are:

- A logical data service with a flat (non-nested) return type.
- An incorrect block scope in the query map.
- An incorrect table join, or no table join, in the query map.

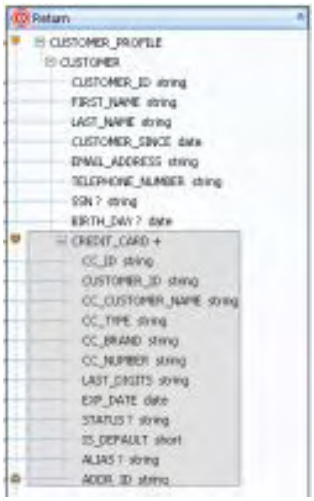
- An attempt to join on a field other than a key field.

Correct the Block Scope

If your logical data service has a nested XML return type, scope the data sources to XML blocks within the return type.

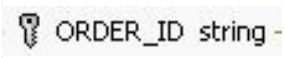
1. In Query Map, click the zone icon of a data source.
2. Drag the zone icon from the data source to the nested element in the return type.
3. Mouse over the zone icon in the data source. Verify that only the nested element is highlighted in the return type.

Checking the Scope in the Return Type



Correct the Table Join

You might also get a non-unique join if the data sources are not joined correctly. You can join the tables either visually in the Query Map or by entering a WHERE clause in the expression editor or the Source tab. Be sure to join tables on a key element, marked like this:



To join tables visually:

1. Click the Query Map tab.
2. Drag from a key element in one data source to the same key element in another data source (for example, $CUSTOMER/CUSTOMER_ID to $ADDRESS/CUSTOMER_ID).
3. Click the Source tab and expand the read function to check the location of the WHERE clause. For example, if your XML return type is nested, the XQuery code should also be nested:

```
for $CUSTOMER in ns1:CUSTOMER()
return
      ...
      for $CREDIT_CARD in ns2:CREDIT_CARD()
      where $CUSTOMER/CUSTOMER_ID eq $CREDIT_CARD/CUSTOMER_ID
      return
      ...
```

To use the expression editor:

1. Click the Query Map tab.
2. Click the For block of the data source you are joining to.
3. In the expression editor, click Add Where Clause  .
4. After the Where keyword, add the elements to be joined (for example, $CUSTOMER/CUSTOMER_ID eq $CREDIT_CARD/CUSTOMER_ID).
5. Click Save  .
6. Check the WHERE clause in the Source tab, as described above.

Remember that ALDSP creates a left outer join if both tables are in the same database and the XML return type is nested. If the XML return type is flat, ALDSP creates an inner join.

Enable Update Blocks and Procedures

If your service has a return type with a flat structure, you may get a non-unique join, even if the join is correct in the Query Map and the Source tab.

If this happens, or if all or part of the update map is disabled for any reason, you can enable an update block or the Create-Update-Delete procedures within the block.

To enable a disabled (yellow) update block:

1. Right-click in the block, and choose Enable.
   The update block should now have a white (enabled) background. The Create, Update, or Delete procedure icons might still appear red or yellow, if they are disabled. However, you should be able to test the primary read function.
2. Click the Test tab.
3. At Select Operation, choose the primary read function, and click Run.

To enable an update map procedure:

1. If an element is marked with a Warning icon  indicating that a mapping is required, select it.
2. In the expression editor, give the element a value with the correct data type.
3. Continue for all disabled elements.

4. In the Test tab, test an update procedure to ensure that the value overrides you have entered do what you want.

Test a Non-Unique Join

Let's go back to the sample joined table data (which we can see in the Test tab, by choosing the primary read function and clicking Run):

CUSTOMER_ID	FIRST_NAME	LAST_NAME	EMAIL_ADDRESS	ORDER_ID	ORDER_DT	TOTAL_ORDER_AMT
CUSTOMER1	Jack	Black	jack@yahoo.com	ORDER_1_0	2001-10-01	156.39
CUSTOMER1	Jack	Black	jack@yahoo.com	ORDER_1_1	2002-02-17	596.65
CUSTOMER1	Jack	Black	jack@yahoo.com	ORDER_1_2	2002-07-07	656.65

In this case, the XML return type is flat, and ALDSP has created an inner join between the CUSTOMER and CUSTOMER_ORDER tables in underlying relational data sources. In the joined table view, one customer has many orders. The CUSTOMER_ID can appear multiple times, but the ORDER_ID is unique.

Once the update map is enabled, you can update data in either the CUSTOMER or CUSTOMER_ORDER table in the data sources:

1. Click a row in the joined table data, then click Edit.
2. Locate the correct node in the XML tree data, and expand it.
3. Click the value you want to change, then edit it.
4. Click Submit.

If you update TOTAL_ORDER_AMT, from the CUSTOMER_ORDER table, the amount changes in one row of the joined table view.

However, if you update EMAIL_ADDRESS, the email address changes in one row of the data source table and in all rows for that customer in the joined table view.

See Also

Concepts

- Recognize When Something is Wrong

How Tos

Document generated by Confluence on Mar 26, 2008 14:37

Handle Non-Unique Values

This page last changed on Feb 26, 2008.

This topic describes how to handle an update map that is disabled because two values in a return type map to one value in a data source.

Example

In a query map, you might attempt to map one value in a data source to two values in an XML return type. When the update map is generated and the flow is reversed, two values map from the return type to one in the data source, which creates an update error.

An Error from a Non-Unique Value

The cause of the error is that two values are attempting to update one in the data source. This creates a build error in the logical data service, and you cannot deploy or test it. You cannot right-click and enable the update block either. The update doesn't work unless you write a custom update function in XQSE.

The best solution is to disable the multiple mapping in the Query Map tab:

1. Click Query Map.
2. Delete the mapping line from the data source to the second, duplicate element in the return type. This should reverse the error.
3. Save the data service and click Update Map to check the change.
4. If the error still exists, right-click and choose Revert Customizations.

See Also

Concepts

- Understanding Update Maps
- Recognize When Something is Wrong

How Tos

- Change a Mapping
- Remove a Mapping
- Revert Customizations

Handle Unmapped Required Values

This page last changed on Mar 11, 2008.

This topic describes how to enable an update map when the data sources on the left have required elements that are not mapped from the return type on the right.

Overview

When required mappings are missing, the Create-Update-Delete procedures for the update block are disabled. That means you cannot create, update, or delete the underlying data sources. In Studio, the update map looks like this.

Required Mappings Are Missing

- A mapping that was deleted from or did not exist in the query map.
- An XML return type that does not contain all required elements. This can be valid, especially if you do not want to expose all elements in your data sources to a client application.

If an element is required but does not have a value, it is marked with a Warning icon .

In either case, the Create, Update, or Delete procedures do not work, so you need to resolve the error. You can do either of these:

- Draw the mapping in Query Map view.
- Enter an override value (either an expression or a constant) in the expression editor.

Draw the Mapping

To draw the mapping in the Query Map tab:

1. Click Query Map.
2. Drag from an element in a data source on the left to the matching element in a return type on the right.

Make sure the elements have the same data types or similar data types that are cast implicitly.

Cast a Constant

If you enter a constant to override the missing mapping, it is only used with Create procedures, to insert data into the data source. Update procedures ignore the override values you enter and leave the data source unchanged. (Of course, Delete procedures delete a record from the data source, so override values are not relevant to them.)

When you enter an override value, make sure the value you enter has the data type the element in the physical data source requires. You can enter a constant like "44" or "2007-01-01" and cast it to an XML Schema data type such as xs:integer or xs:date, using either of these:

- A built-in XQuery cast function
- The parentheses cast operator, as in xs:date("string"), to invoke an XML Schema type constructor function

The parentheses cast operator uses any XML Schema data type outside the parentheses and a string that is appropriate for the data type you are casting to within the parentheses. For example, you can perform these casts:

```
xs:date("2007-01-01")
xs:dateTime("2007-01-01T16:44:44")
xs:integer("44")
```

But you cannot perform these:

```
xs:date("2007-01-01T16:44:44")
xs:dateTime("date")
xs:integer("text")
```

To cast a constant in the expression editor:

1. Click the Update Map tab.
2. Click an unmapped element in a data source on the left.
3. In the expression editor, enter a constant that has the data type the element requires. For example, for an element of type xs:string, you might enter:

   ```
   "Bob"
   ```

   If the element has another data type, enter a string within a cast expression, for example:

   ```
   xs:integer("44")
   xs:dateTime("2007-07-17T09:00:00")
   ```

4. Continue for all disabled elements.
5. In the Test tab, test an update using Run - Edit - Submit to make sure the value overrides work as you expect.

See Also

Concepts

- Recognize When Something is Wrong

How Tos

- Cast Using a Built-In XQuery Function
- Cast Using a Custom XQuery Function
- Test an Update Map Cast

Other Resources

- XML Schema Datatypes Specification (W3C)

Document generated by Confluence on Mar 26, 2008 14:37

Test an Update Procedure

This page last changed on Mar 11, 2008.

This topic describes how to test an Update procedure in Test view in Studio.

Configure Audit Properties

To test an Update procedure in Studio, you must submit a data graph in the Parameters box in Test view. A data graph is an XML structure with a root element of <sdo:datagraph> and a <changesummary> element. The easiest way to submit a data graph is to capture one from an audit.

First, configure audit properties in the ALDSP Console.

Configuring Audit Properties in the ALDSP Console

To configure audit properties so that ALDSP generates data graphs:

1. Open the ALDSP Console and log in.
2. Click the name of a data space project.
3. Click the Audit Properties tab.
4. Click Lock & Edit in the upper left pane.
5. Navigate to the Update > Service node (be careful not to move to Update > Error > Procedure).
6. For Name, Parameters, and Result, choose Always from the Is Audited menu.
7. Click Save.
8. Click Activate Changes in the upper left pane.

Capture the Data Graph

You can then capture a data graph from the audit messages displayed in the Studio Console tab, and edit the data graph to submit to the Update procedure in Test view.

Viewing a Data Graph in the Studio Console Tab

To capture a data graph:

1. Open a logical data service in Studio.
2. Click the Test tab.
3. Choose the service's primary Read function, then click Run.
4. Click Edit, edit a value, then click Submit.
5. (Optional) Check the Studio Console tab.
   If you see the WebLogic Server console data, not the ALDSP console data, click the drop-down arrow next to the console icon  , and choose ALDSP Console.
6. Scroll up in the Studio Console tab until you locate the data graph, right-click, and copy it.

Submit the Update

When you update relational sources, the SDO update mechanism uses optimistic locking to avoid change conflicts. With optimistic locking, the data source is not locked when the SDO client acquires the data. Later, when the client wants to update, the data in the source is compared to a copy of the data at a time when it was acquired. If there are discrepancies, the update is not committed. Before you submit the data graph to the Update procedure, be sure that optimistic locking is enabled in the underlying data source you are updating.

You can then submit the data graph to the Update procedure. However, you may need to edit it, as the data graph you captured from the Studio Console tab reflected the last change you made, not the change you are presently submitting to the Update procedure.

Submitting the Data Graph to the Update Procedure

The data graph you submit to the Update procedure takes the place of the return type as an argument, even if you are updating only some of the elements in the return type.

To submit the data graph to an Update procedure:

1. Enable optimistic locking on any physical relational data sources the data graph is updating.
2. Open a data service in Studio, and click the Test tab.
3. At Select Operation, choose an Update procedure.
4. Copy a data graph you have captured from the Studio Console tab to the Parameters box.

5. Edit the data graph for the change you want to make.

   The data graph you captured applies to a change made in the visual interface. Update the change summary to the values the object presently has, and the remaining elements to the new values you want to set. For example, this is a change summary captured from the Studio Console tab:

```
<sdo:datagraph xmlns:sdo="commonj.sdo">
  <changeSummary>
      <sim:SIMPLE_CUSTOMER sdo:ref="#/sdo:datagraph/sim:SIMPLE_CUSTOMER" xmlns:sim="ld:logical/
SimpleCustomer">
      <CUSTOMER_SINCE>1999-01-01T00:00:00</CUSTOMER_SINCE>
      </sim:SIMPLE_CUSTOMER>
  </changeSummary>
  <sim:SIMPLE_CUSTOMER xmlns:sim="ld:logical/SimpleCustomer">
      <CUSTOMER_ID>CUSTOMER7</CUSTOMER_ID>
      <CUSTOMER_SINCE>2007-11-11T00:00:00</CUSTOMER_SINCE>
  </sim:SIMPLE_CUSTOMER>
</sdo:datagraph>
```

This version has been updated in the Parameters box (note the difference in the CUSTOMER_SINCE dates):

```
<sdo:datagraph xmlns:sdo="commonj.sdo">
  <changeSummary>
      <sim:SIMPLE_CUSTOMER sdo:ref="#/sdo:datagraph/sim:SIMPLE_CUSTOMER" xmlns:sim="ld:logical/
SimpleCustomer">
      <CUSTOMER_SINCE>2007-11-11T00:00:00</CUSTOMER_SINCE>
      </sim:SIMPLE_CUSTOMER>
  </changeSummary>
  <sim:SIMPLE_CUSTOMER xmlns:sim="ld:logical/SimpleCustomer">
      <CUSTOMER_ID>CUSTOMER7</CUSTOMER_ID>
      <CUSTOMER_SINCE>2008-04-04T00:00:00</CUSTOMER_SINCE>
  </sim:SIMPLE_CUSTOMER>
</sdo:datagraph>
```

1. Click Run. You should see this message in Test view:

   ```
   Operation was successful.
   ```

See Also

Concepts

- Brief Overview of Service Data Objects (for Studio)
- Data Programming Model and Update Framework (in depth, for client applications)

How Tos

- Enable Optimistic Locking for Relational Objects

Other Resources

- Introducing SDO

Preparing Services for Clients

This page last changed on Feb 26, 2008.

Preparing Services for Clients

Mediator Client

How-to...

Generate a Mediator Client JAR File

Generate a Web Services Mediator Client JAR File

Web Services

How-to...

Generate a Web Service Map and WSDL from a Data Service

Configure Security for Web Services Applications

Reference...

Web Services Map File Reference

SQL Maps

Concepts...

Understanding SQL Maps

How-to...

Map Functions and Procedures to SQL Objects

Reference...

SQL Object Mapping Rules

Constraints on Publishing Data Service Objects to SQL

Generate a Mediator Client JAR File

This page last changed on Feb 26, 2008.

To use the Static Mediator API in a client application, you must generate a Mediator Client JAR file. This JAR file contains the Static Mediator API interfaces, plus all the necessary SDO-compiled schemas for a dataspace.

One Java method is generated for each mapped data service operation. Method names match the mapped data service operation names. Client developers access data service operations by calling these methods.

This section explains how to generate a Mediator Client JAR by these two methods:

- Using the IDE
- Using the Command-Line Tool

Tip: You can also generate a Mediator Client JAR using the Administration Console. See the ALDSP Adminstration Guide for details.

Using the IDE

To generate a mediator client JAR file using the IDE:

1. Select File > Export.
2. In the Export dialog, select AquaLogic Data Services Platform > Mediator Client JAR File.
3. Click Next.
4. Complete the Mediator Client JAR File dialog as follows:

- Select a Dataspace project to export. You can only select one Dataspace project at a time.
- Specify a directory in which to place the exported JAR file. You can use the drop down list to select a recently specified directory or use the Browse button to locate one.
- Unselect the Use default name checkbox if you want to enter a name for the JAR file.
- Click Finish to create the JAR file.

The ALDSP Console view displays the export task status and any errors that may have occurred. You can click the Cancel button to cancel the export task before it has completed.

Using the Command-Line Tool

This section explains how to generate a Mediator Client JAR file using the command-line tool. Before using the command-line tool, be sure you have the following:

- WSL 9.2 MP1 or MP2 installed with ALDSP installed in the default location BEA_HOME/aldsp_3.0.
- A Dataspace project on your local filesystem that contains data service (.ds) and schema (.xsd) files. Miscellaneous IDE files within the project folder are allowed and will not affect the export.
- Ant installed and in your path.

To generate the client JAR, use this Ant command:

```
ant -Dapproot=PROJECT_HOME -f BEA_HOME/aldsp_3.0/bin/sdo_dspclientgen.xml
```

where PROJECT_HOME is the full path to the Data Space project's root folder, and BEA_HOME is the root path for your WebLogic installation.

For example (all on one line):

```
ant -Dapproot=/home/myprojects/myapp -f /home/bea/aldsp_3.0/bin/sdo_dspclientgen.xml
```

This Ant script produces a file named PROJECTNAME-dsp-client.jar in PROJECT_HOME, where PROJECTNAME is the name of the directory PROJECT_HOME (as opposed to the full path to that directory). For example, the above script produces the Mediator Client JAR file:

```
/home/myprojects/myapp/myapp-dsp-client.jar.
```

Optional command-line features include:

- Your environment must contain a WL_HOME environment variable, pointing to the WLS 9.2 installation. If it does not, you can provide an alternate by adding -Dwl.home=/path to specify the WLS root directory.
- Your ALDSP 3.0 installation must be in the default directory BEA_HOME/aldsp_3.0. If it is not, you can provide an alternate by adding -Ddsp.home=/path to specify the directory.
- To specify a full directory path for the output, add -Doutdir=/dirpath to the Ant command. You must provide an absolute path; a relative path, including ".", will not work, as it is assumed to be relative to PROJECT_HOME.
- To specify a different name for the JAR file, add -Dsdojarname=name.jar.

Generate a Web Services Mediator Client JAR File

This page last changed on Mar 11, 2008.

This section explains how to generate a Web Services Mediator Client JAR file. This JAR is required by developers writing Java clients that access data services through web services using the Static Mediator API.

This section includes these topics:

Overview

To use the Static Mediator API in a web services-enabled client application, you must generate a Web Services Mediator Client JAR file. This JAR file contains the Static Mediator API interfaces, plus all the necessary SDO-compiled schemas for a dataspace.

One Java method is generated for each data service function that is mapped to a WSDL operation. Method names match the mapped WSDL operation name. Client developers access data service functions through the web service by calling these methods. If the web service requires message-level security, you can add a credential provider and trust manager through initial context properties. For more information on security, see Configure Security for Web Services Applications.

This topic explains how to generate a Web Services Mediator Client JAR file using these methods:

Tip: You can also generate a Mediator Client JAR using the Administration Console. See the ALDSP Adminstration Guide for details.

Using Studio

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

1. Select File > Export.
2. In the Export dialog, select AquaLogic Data Services Platform > Web Services Mediator Client JAR File, and click Next.
3. Complete the Web Services Mediator Client JAR File dialog as follows, and click Finish.
4. In the left panel, select the Dataspace project that contains the .ws file(s) to export. You can only export .ws files in one Dataspace project at a time. Checking/unchecking the checkbox next to a project or a folder automatically checks/unchecks all the sub-folders and .ws files under that project/folder.
5. In the right panel, select the Web Service Map file to export. You can select one or more .ws files. To see and selectively check the .ws files in a sub-folder, you will need to expand and click on the folder on the left panel. The message under the right panel shows the total number of .ws files currently checked for export.
6. Specify a directory in which to place the exported JAR file. You can select any location on your system. You can use the dropdown list to select a recently specified directory or use the Browse button to locate one. By default, the exported JAR will be named: <data_space_name>-ws-client.jar.
7. Unselect the Use default name checkbox if you want to enter a name for the JAR file.

The ALDSP Console view displays the export task status and any errors that may have occurred. You can click the Cancel button to cancel the export task before it has completed.

Using the Command-Line Tool

This section explains now to generate the Web Services Mediator Client JAR file using Ant and presents example Ant commands. Before using the command-line tool, be sure you have the following:

- WSL 9.2 MP1 or MP2 installed with ALDSP installed in the default location BEA_HOME/aldsp_3.0.
- A Dataspace project on your local filesystem that contains data service (.ds) and schema (.xsd) files. Miscellaneous IDE files within the project folder are allowed and will not affect the export.
- Ant installed and in your path.

To generate the JAR file, run this Ant command:

```
ant -Dapproot=PROJECT_HOME -Dwslocator=locator -f BEA_HOME/aldsp_3.0/bin/sdo_dspclientgen.xml
```

Where:

- PROJECT_HOME is the path to the Dataspace project. You must specify a full path for the values of BEA_HOME and PROJECT_HOME.
- The locator option takes one of these values:
  - d:URI - Specifies a URI (or a semicolon-separated or space-separated list of URIs) to a .ws file in the Dataspace project from which to generate the JAR file. For example:

    ```
    ld:MediatorTestDataServices/CustomerWeb.ws
    ```
- ALL - Generates the JAR for all .ws files in the dataspace.

  The result of executing this Ant script is a file named PROJECT-ld-client.jar in PROJECT_HOME, where PROJECT is the name of the directory PROJECT_HOME (as opposed to the full path to that directory).

Additional Ant Task Options

This section lists several optional features that you can use with the Ant tasks described in the previously:

- Your environment must contain a WL_HOME environment variable, pointing to the WLS 9.2 installation. If it does not, you can provide an alternate by adding -Dwl.home=/path to specify the WLS root directory.
- Your ALDSP 3.0 installation must be in the default directory BEA_HOME/aldsp_3.0. If it is not, you can provide an alternate by adding -Ddsp.home=/path to specify the directory.
- To specify a full directory path for the output, add -Doutdir=/dirpath to the Ant command. You must provide an absolute path; a relative path, including ".", will not work, as it is assumed to be relative to PROJECT_HOME.
- To specify a different name for the JAR file, add -Dsdojarname=name.jar.

Example 1

This example specifies multiple .ws files. The command must be entered on one line.

```
ant -Dapproot=/home/myprojects/myapp -Dwslocator='ld:MediatorTestDataServices/CustomerWeb.ws;
ld:MediatorTestDataServices/OtherCustomerWeb.ws' -f /home/bea/aldsp_3.0/aldsp_3.0/bin/sdo_dspclientgen.xml
```

Example 2

This example generates a JAR that includes all of the .ws files in the dataspace. The command must be entered on line line.

```
ant -Dapproot=/home/myprojects/myapp -Dwslocator=ALL -f /home/bea/aldsp_3.0/aldsp_3.0/bin/sdo_dspclientgen.
xml
```

Generate a Web Service Map from a Data Service

This page last changed on Feb 26, 2008.

If you intend to access a data service through web services using the Data Services Mediator API, you must generate a web service map file first. A web service map file maps data service functions to web service operations. The map file is also used for setting and configuring security policies for web services applications.

Topics

Creating a Map File

This section describes the basic steps that are required to create a map file. You can accomplish all of these tasks using the ALDSP Eclipse IDE. The procedure assumes that you have created or have access to the data service (.ds) file from which you want to create a data service.

Web Services Map File Reference

There are two ways to create a web service map file (.ws file):

Method 1:

1. Obtain access to the data space project containing the data service you wish to make accessible from a web service.
2. Right-click on the data service name in Project Explorer and select Create Web Service Map. The map file (.ws file) is created with the same name as the data service file, and the map file is opened in the editor.

Method 2:

1. Obtain access to the data space project containing the data service you wish to make accessible from a web service.
2. Right-click on the data service name in Project Explorer and select:

   New > Web Service Map

3. Use the dialog to create an empty web service map file with a name of your choosing (example: OrderService).
4. Click Finish. The empty map file opens in the editor.
5. Drag either an entire data service file onto the map file or drag individual data service operations

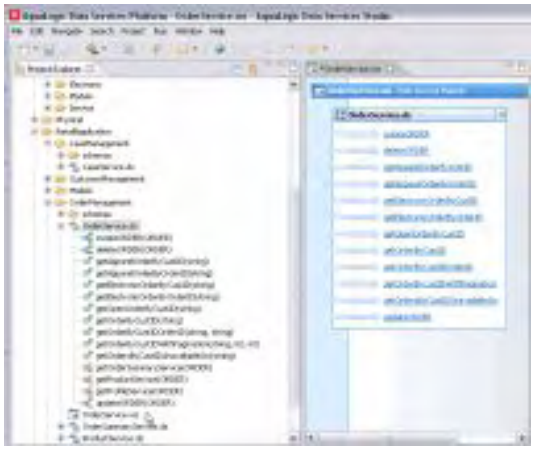
   Example: RetailApplication > OrderManagement > OrderService.ds

6. Click OK. A file named OrderService.ws will be created. It will be located in the same folder as the source data service.

   Only the data service functions that are mapped in the map file are available to clients. Only public data service operations can be mapped.

   The following figure shows a data service file called OrderService.ds as the source for a map file called OrderService.ws (created using Method 2).

Adding a Data Service to a Web Services Map File



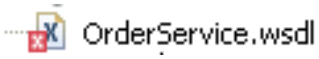
You can add additional public operations from other data services to the same web service map file.

Generating a WSDL File from Map File

To generate a WSDL file from a .WS file:

1. Right-click on the .ws file.
2. Choose Save WSDL As...
3. Specify a name for the WSDL file.
4. Click Save. Your new WSDL file should appear in the directory identified in the Save as dialog.
5. Double-click on the WSDL file to verify the soap service for the WSDL.

A WSDL that has more than one schema section pointing to the same target namespace will result in validation errors with Eclipse WTP default WSDL validator. The WSDL generated in the above example is valid; however the project will indicate a validation error condition.



The error condition will not interfere with your ability to build and deploy the project. Also, you can use the following Eclipse option settings to prevent the validation error report from displaying:

1. Select the following option:

```
Project > Properties > Validation
```

2. Select Override validation preferences.
3. Uncheck the Build option associated with the WSDL Validator.
4. Click Apply, then OK.
5. Select the following option:

```
Project > Clean
```

2. Click OK.

The validation error warnings should disappear.

Examining the Generated WSDL

You can examine the generated WSDL file. See Web Services Map File Reference for details.

Testing the Generated WSDL

You can test the generated WSDL file. See Web Services Map File Reference for details.

Modifying the Map File

This section describes additional ways to add data services and operation to a map, and how to delete operations from an existing map.

Adding Data Services and Operations

You can drag and drop either an entire data service or individual data service operations from the Project Explorer onto an existing map file in the map file editor.

You can right-click in the map editor and select Add Data Services/Operations to Map. Use the Select Resources to Add to Map dialog to add data service resources to the map.

Deleting Data Services and Operations from a Map File

To delete one or more operations, select the operations and right-click on the selected operations, and then select Delete.

To delete all operations that are related to a data service, right-click on the .ds dataservice box and select Delete.

Renaming Mapped Operations

To rename a mapped operation, select the operation, right-click and select Rename Operation. Then enter a new name for the mapped operation.

Configure Security for Web Services Applications

This page last changed on Feb 26, 2008.

ALDSP Native Web Services supports the following security features:

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

Configuring Basic Authentication

To use basic authentication, set the Basic Auth Required property of the web services map file to true. For more information, see Web Services Map File Reference.

Configuring Transport Level Security (HTTPS)

Use the web service map file property editor to change the Transport Type to HTTPS. HTTP is the default. For more information, see Web Services Map File Reference.

For HTTPS, you can configure either 1-way or 2-way SSL. For detailed information on transport level security, see the WebLogic Server document Configuring Security: Configuring Transport-Level Security on e-docs.

Configuring Web Services Security (WSS)

WSS provides message level security. For WSS, ALDSP Native Web Services supports the same standards that are supported by WebLogic Server. For detailed information on WSS, see the WebLogic Server document, Configuring Security: Updating a Client Application to Invoke a Message-Secured Web Service on e-docs.

The supported standards include:

- SOAP Message Security
- Username Token Profile
- X.509 Certificate Token Profile
- SAML Token Profile

To use Web Services Security with an ALDSP web services application:

1. Choose the type of web services security you want to use with your ALDSP application.
2. Configure security policies through the appropriate policy file(s). See the WebLogic Server document Configuring Security : Overview of Web Services Security for detailed information on configuring policy files for each type of web services security.
3. Edit the web services mapping file to include your policy file(s). You can associate policies with an entire mapping

file or for specific operations within the file. See "Specifying Policies" below for details.

Specifying Policies

You can specify policies for a map file or for individual operations in a map file.

Specifying Global Policies

To specify a policy for web services security for a map file:

1. Create the policy file. See the WebLogic Server document: "WebLogic Web Services: Security" for detailed information on configuring policy files for each type of web services security.
2. Import the policy file into your ALDSP project. The easiest way to do this is to use the IDE to import the file as a resource. The policy file must reside in the DSP-INF/policies directory.
3. Configure the web services map file to include the policy.

The following listing shows an example .ws file that includes the optional, top-level policies element. Each policy element describes one policy file. The policies element can contain one or more policy elements. The locator attribute contains either an ALDSP locator for the policy file or a fixed URI that describes the location of the standard WLS policy file.

ALDSP supports three security policy types. Their URIs are: policy: Auth, policy: Encrypt, policy: Sign. These are abstract policy files provided by WebLogic Server that describe authentication, encryption, and digital signature policies. These policy files do not have to physically reside in DSP project repository.

The policy element contains a required attribute Direction. This attribute represents at which direction the security policy will apply. The policy direction can be: REQUEST, RESPONSE, or REQUEST_RESPONSE.

- REQUEST - The policy applies only to the inbound request message.
- RESPONSE - The policy applies only to the response message.
- REQUEST_RESPONSE - The policy applies to both inbound request and the response message.

Refer to the schema definition for detailed information on the structure of the map file (see the topic Web Services Map File Reference).

Sample Map File

```
<?xml version="1.0" encoding="UTF-8"?>
<web:WebServicesMap targetNamespace="ld:myMapper.ws" soapVersion="SOAP_1.1" transportType="HTTP"
ADODotNETEnabled="false" basicAuthRequired="false" xmlns:web="http://www.bea.com/dsp/management/
configuration/webservices">
    <web:policies>
        <web:policy locator="ld:mypolicy.xml">
         <web:policy direction="REQUEST_RESPONSE">
    </web:policies>
    <web:dataServices>
        <web:dataService locator="ld:CUSTOMER.ds">
            <web:function name="deleteCUSTOMER" arity="1"            operation="deleteCUSTOMER"
```

```
returnInHeader="false">
                <web:parameterMapping>
                    <web:parameter name="p" wsdlMapping="SOAP_BODY"/>
                </web:parameterMapping>
            </web:function>
            <web:function name="updateCUSTOMER" arity="1"            operation="updateCUSTOMER"
returnInHeader="false">
                <web:parameterMapping>
                    <web:parameter name="p" wsdlMapping="SOAP_BODY"/>
                </web:parameterMapping>
            </web:function>
        </web:dataService>
    </web:dataServices>
</web:WebServicesMap>
```

Specifying Policies for a Function

To specify policies for a function in a map file:

1. Follow the same basic instructions for specifying a policy for a web service map file, described previously.
2. In the .ws file, add the policies element to the function element. The policies element contains one or more policy element. A policy element represents the security policy that applies to the WSDL operation. The optional child element ParameterMapping for the function element contains a list of parameters that are mapped to the SOAP header.

Web Services Map File Reference

This page last changed on Feb 26, 2008.

The web services map file is an XML file that provides an explicit mapping between ALDSP data service functions and web service operations. The map file is the basis for generating the WSDL that describes the web services interface for a data service. This section discusses the configurable parts of the map file in detail. For details on creating a map file, see the topic Generate a Web Service Map from a Data Service.

Topics

Map File-Level Properties

The ALDSP Eclipse IDE lets you create the web services map file (as explained in Generate a Web Service Map From a Data Service) and configure the map file. The New Web Service Map wizard creates a .ws file in a specified location within the Dataspace project. This section describes the configurable map file properties. To configure these properties, use the Properties editor in the IDE.

The following figure shows a sample map file that maps functions from a data service called Customer.ds. To view properties for a map file, select the map file in the IDE and select Window > Show View > Properties.

Map File-Level Properties

The following table describes each of the map file properties.

Map File Properties

Property	Description

ADO.net Enabled	If enabled, a .NET style WSDL is generated. This WSDL includes .NET datasets in the WSDL construct. Disabled by default. For more information on ADO.NET, see the Client Application Developer's Guide.
Basic Auth Required	If true, basic authentication is required to access the WSDL operations.
Map Name	(Read-only) The name of the map file.
SOAP Version	SOAP 1.1 and 1.2 are supported. The version is used by ALDSP to decide which kind of SOAP binding to create during WSDL generation. The default is 1.1. SOAP 1.2 encoding is not supported. Encoding is an optional feature defined by the SOAP 1.2 specification.
Target Name Space	The default value is generated from the web service based on the location of the map file and the file name.
Transport Type	HTTP and HTTPS are the only supported types. Default is HTTP.
Policies	Lets you specify security policies that apply to all the functions in the map. For information on policies, see the topic Configure Security for Web Services Applications.

Operation Level Properties

This section describes the operation-level properties that you can modify in the IDE. Operations match up with data service functions. Each data service function maps to a WSDL operation. Operation-level properties apply to the specific operation only.

The following figure shows the properties displayed for a selected data service function. To view properties for a data service operation, select the operation in the IDE and select Window > Show View > Properties.

Operation-Level Properties

The following table describes each of the operation properties.

Map File Properties

Property	Description
Data Service Name	Read-only.
Function Name	Read-only.
Operation Name	The WSDL operation name that is used to generate a WSDL. This name has to be unique within the map file.
Return Type	Maps the WSDL operation return type to either a SOAP header or body.

Parameters	Lists all parameters for the operation and lets you map each parameter to either a SOAP header or body.
Policies	Lets you specify security policies that apply to the operation. For information on policies, see the topic Configure Security for Web Services Applications.

Map File XML Schema Definition

The following listing shows the schema file for the map (.ws file) definition.

Web Services Map File Schema Definition

```
<xs:schema
targetNamespace="http://www.bea.com/dsp/management/configuration/webservices"
 xmlns:tns="http://www.bea.com/dsp/management/configuration/webservices"
 xmlns="http://www.bea.com/dsp/management/configuration/webservices"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 elementFormDefault="qualified">

  <xs:element name="WebServicesMap">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="policies" type="PoliciesType"
                    minOccurs="0"/>
        <xs:element name="dataServices"
                    type="DataServicesType"/>
      </xs:sequence>
      <xs:attribute name="targetNamespace" type="xs:anyURI"
                    use="required"/>
      <xs:attribute name="soapVersion" type="SoapVersionType"
                    default="SOAP_1.1"/>
      <xs:attribute name="transportType" type="TransportTypeType"
                    default="HTTP"/>
      <xs:attribute name="ADODotNETEnabled" type="xs:boolean"
                    default="false"/>
    </xs:complexType>
  </xs:element>

  <xs:simpleType name="SoapVersionType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="SOAP_1.1"/>
      <xs:enumeration value="SOAP_1.2"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="TransportTypeType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="HTTP"/>
      <xs:enumeration value="HTTPS"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="PoliciesType">
    <xs:sequence>
      <xs:element name="policy" type="PolicyType" minOccurs="1"
                  maxOccurs="unbounded"/>

    </xs:sequence>
  </xs:complexType>
```

```
  <xs:complexType name="PolicyType">
    <xs:attribute name="locator" type="xs:string" use="required"/>
    <xs:attribute name="direction" type="PolicyDirectionType"
              default="REQUEST_RESPONSE"/>

  </xs:complexType>

  <xs:simpleType name="PolicyDirectionType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="REQUEST"/>
      <xs:enumeration value="RESPONSE"/>
      <xs:enumeration value="REQUEST_RESPONSE"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="DataServicesType">
    <xs:sequence>
      <xs:element name="dataService" type="DataServiceType"
                  minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="DataServiceType">
    <xs:sequence>
      <xs:element name="function" type="FunctionType" minOccurs="1"
                  maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="locator" type="xs:string" use="required"/>
  </xs:complexType>

  <xs:complexType name="FunctionType">
   <xs:sequence>
    <xs:element name="policies" type="PoliciesType" minOccurs="0"/>
    <xs:element name="parameterMapping" type="ParameterMappingType"
           minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
    <xs:attribute name="arity" type="xs:integer" use="required"/>
<xs:attribute name="operation" type="xs:string" use="required"/>
<xs:attribute name="returnInHeader" type="xs:boolean"
             default="false"/>
  </xs:complexType>

  <xs:complexType name="ParameterMappingType">
    <xs:sequence>
      <xs:element name="parameter" type="ParameterType" minOccurs="1"
                  maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="ParameterType">
    <xs:attribute name="name" type="xs:string" use="required"/>
<xs:attribute name="wsdlMapping" type="WSDLMappingType"
              use="required"/>
  </xs:complexType>

  <xs:simpleType name="WSDLMappingType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="SOAP_HEADER"/>
     <xs:enumeration value="SOAP_BODY"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

How Data Service Types Map to WSDL Message Types

This section explains how data service types are mapped to WSDL message types when you map a data service function to a WSDL operation.

Two Schema Elements Per Function

For each data service function, two WSDL schema elements are generated. The first element is the name of the request message, and it is the same as the data service function name that is mapped to the WSDL message. The second represents the response message. The response message name is the same as the function name with "Response" appended to it. The following listing shows an example schema where getCustomer is the request name and getCustomerResponse is the response name. The response element contains the return type of the data service function, which can be complex or simple.

Operation Element and Return Element

```
  <types>
    <xsd:schema targetNamespace="ld:DataServices/RTLServices/Customer.ws"
                xmlns:dsns0="urn:retailerType">
    <xsd:import namespace="urn:retailerType"/>

    <xsd:element name="getCustomer">
      <xsd:complexType>
        <xsd:sequence/>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="getCustomerResponse">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element ref="dsns0:CUSTOMER_PROFILE" minOccurs="0"
                       maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </types>
```

Mapping of Update Functions with DataGraphs

This section explains how a data service update operation's parameters and return type are mapped to a WSDL schema definition.

Consider the following data service definition for an operation called updateADDRESS:

```
  (::pragma function <f:function xmlns:f="urn:annotations.ld.bea.com"
  visibility="public" kind="update" isPrimary="true" nativeName="ADDRESS"
  nativeLevel2Container="RTLCUSTOMER" style="table">

  <nonCacheable/>   </f:function>::)
```

```
    declare procedure f1:updateADDRESS($p as changed-element(t1:ADDRESS)*) as empty()  external;
```

Note that the operation's parameter type is changed-element(UserType). In this case the element is ADDRESS. The changed-element type is translated to a DataGraph in the WSDL schema. The WSDL schema must also include a schema definition for the DataGraph. The following listing shows the translated updateADDRESS operation and the schema definition for the DataGraph.

WSDL Schema for an Update Operation

```
  <xs:element name="updateADDRESS">
  <xs:complexType>
  <xs:sequence>
  <xs:element name="p">
  <xs:complexType>
  <xs:sequence>
    <xs:element ref="dsns0:ADDRESSDataGraph" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
      </xs:complexType>
      </xs:element>
      </xs:sequence>
      </xs:complexType>
      </xs:element>
  <xs:element name="updateADDRESSResponse">
  <xs:complexType>
      <xs:sequence />
      </xs:complexType>
      </xs:element>
  ...
  <xs:schema targetNamespace="ld:ADDRESS" xmlns:dsns0="ld:ADDRESS"    xmlns:sdo="commonj.sdo">
    <xs:import namespace="commonj.sdo"      schemaLocation="http://www.osoa.org/sdo/2.1/schemas/datagraph.
  xsd" />
    <xs:element name="ADDRESSDataGraph" type="dsns0:ADDRESSDataGraphType" />
  <xs:complexType name="ADDRESSDataGraphType">
  <xs:complexContent>
  <xs:extension base="sdo:BaseDataGraphType">
  <xs:sequence>
    <xs:element ref="dsns0:ADDRESS" />
      </xs:sequence>
      </xs:extension>
      </xs:complexContent>
      </xs:complexType>
      </xs:schema>
  ...
```

Overloading Data Service Functions

Data service functions can be overloaded, meaning that two functions in the same data service have the same name but a different number of parameters. For example, in the following listing two getCustomer() functions are declared, each with a different parameter set. To support WSDL generation for overloaded data service functions, the web services map requires the overloaded function to be mapped to a different WSDL operation. In other words, if you drag two functions with the same name from a data service onto a web service map file, ALDSP generates different WSDL operation names for the two functions. You can accept the default names or change them.

Overloaded Functions

```
  declare function ns9:getCustomer() as element(ns2:CUSTOMER_PROFILE)*
```

```
declare function ns9:getCustomer($customerID as xs:string) as element(ns2:CUSTOMER_PROFILE)*
```

Examining the Generated WSDL

You can examine the generated WSDL file. The dataspace project's associated WebLogic server must be started and the dataspace project be deployed to the server to view the WSDL or test the Web Service.

1. Right-click on the web service file name (example: CUSTOMER.ws)
2. Choose:

```
View WSDL
```

The WSDL will appear in its own window in the work area.

View of Generated WSDL

You can also request the WSDL for a deployed project by entering the following URL:

```
http://host:port/dataSpaceProjectName/folderName/.../mapFileName.ws?WSDL
```

For example:

```
http://localhost:7001/myDataSpace/myWSMapper.ws?WSDL
```

Testing the Generated WSDL

You can test the generated WSDL file using these steps. The dataspace project's associated WebLogic server must be started and the dataspace project be deployed to the server to view the WSDL or test the Web Service.

1. Right-click on the web service file name (example: CUSTOMER.ws)
2. Choose:

```
Test Web Service
```

The WSDL will appear in its own window.

View of Tested Web Service

Copying and Saving a WSDL Generated from a Map

You can copy or save a WSDL by right-clicking the map file and selecting Copy WSDL URL or Save WSDL As.

Understanding SQL Maps

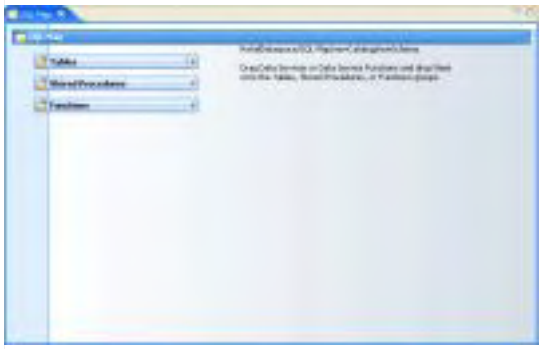
This page last changed on Feb 26, 2008.

This topic describes mapping functions in data services to SQL objects.

Overview

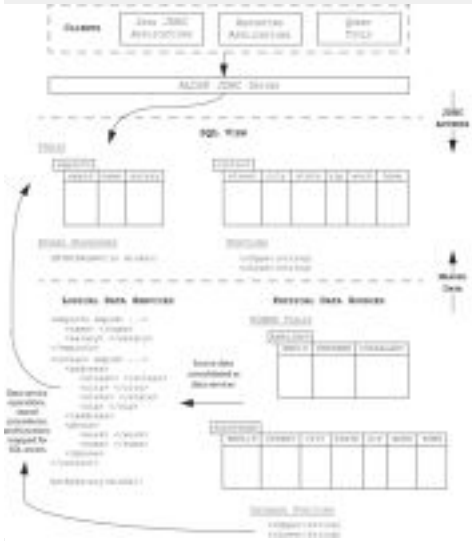
A SQL Map enables you to publish data service functions as SQL objects (which are created when you specify the mapping). Using SQL Maps, you can expose data services modeled in ALDSP as relational data sources. This enables you to use reporting tools (such as Crystal Reports and Microsoft Access, among others), Java applications, and development tools (such as Data Tools Platform or SQL Explorer) to access information from data services using SQL queries (through a JDBC client).

Sample SQL Map



As the following figure shows, source data can be consolidated, integrated, and transformed using ALDSP data services. The source data itself can come from disparate sources throughout the enterprise, including relational databases and web services, among others. Using SQL Maps you can, 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 ALDSP.

SQL Mapping Overview



You can publish the following types of data service artifacts for SQL access:

- Data service functions, either with or without parameters.
- External database functions. These are database-specific functions which are either built into a particular commercial database or which were custom-designed on the database side and then declared to ALDSP as external database functions.

Publishable Operations

SQL mappable data service functions can be thought of as relationally-compatible XQuery functions. Depending on their signature, you can publish such functions for use as SQL tables or stored procedures. The association between the function and the SQL object is defined at design time when creating a SQL map.

The following summarizes the types of data service functions you can publish as SQL tables or stored procedures:

- You can map non-parameterized data service functions as SQL tables, and parameterized data service functions as stored procedures.
- You cannot map private or protected functions as part of a SQL Map.
- You cannot map procedures as part of a SQL Map.

You can map library data service database functions to functions, but not to SQL tables and stored procedures.

See Mapping Rules for details about permitted mappings.

General Conditions

The following general conditions apply when exposing data service operations as relational data sources:

- The exposed data service XQuery function signatures must only involve types that are supported by the relational (JDBC) type system.
- The structure of the underlying schema (the Return type) must have a relationally-compatible data shape, which means that the data service type cannot include repeating data (data elements with cardinality greater than 1). This is because SQL provides a traditional, two-dimensional approach to data access, as opposed to the multi-level, non-normalized approach defined by XML.

  For more information about creating data services with flat schemas, see Create a Data Service with a Flat Return Type.

- You cannot map data service operations that return scalar (primitive) values as SQL tables or stored procedures.

See Also

How Tos

- Map Functions and Procedures to SQL Objects

Reference

- Mapping Rules
- Constraints on Publishing Data Service Objects to SQL

Map Functions and Procedures to SQL Objects

This page last changed on Mar 11, 2008.

This topic describes how to create SQL objects (tables, stored procedures, and database functions) from dataspace project operations including conforming physical and logical functions and procedures. Once created, the objects will be available to client applications through JDBC.

Creating an SQL Map

To create an SQL map:

1. In Studio, right-click a dataspace project folder in the Project Explorer and choose Add SQL Map. Data Service Studio creates an SQL Map with a default catalog and schema.

   You can define only a single SQL Map for a dataspace. You can, however, add multiple catalogs and schemas to an SQL Map.

   You can rename the default catalog and schema by expanding the SQL Map in the Project Explorer, right-clicking the catalog or schema, choosing Rename. and entering the new name.
2. Select the schema to which you want to map the data service functions and procedures. Studio displays folder tabs for Tables, Stored Procedures, and Functions.
3. Drag-and-drop the data service functions and procedures from the Project Explorer to the corresponding folder tab in the SQL Map.

   If you drag-and-drop an entire data service to a folder tab in the SQL Map, Studio attempts to map all functions contained in the data service to the corresponding SQL object type (Table, Stored Procedure, or Function).

   If you attempt to map a dataspace object which does not meet SQL map criteria, a dialog will appear, explaining the problem.

   You can map the same data service function or procedure to multiple schemas. You can also map a function or procedure to multiple SQL object types, however, Studio displays an alert dialog

(see Map Data Service Functions for SQL Use Alert Dialog) in case of a naming conflict and suggests a new SQL name. You can edit this new name, as required.

Populated SQL Map

Removing an SQL Map

To remove an SQL map:

- Right-click the dataspace project folder and choose Remove SQL Map.

See Also

Concepts

- Understanding SQL Maps

Reference

- Map Data Service Functions for SQL Use Alert Dialog
- Mapping Rules
- Constraints on Publishing Data Service Objects to SQL

Mapping Rules

This page last changed on Feb 26, 2008.

The function and procedure types used in data services map to various types of SQL objects. The general mapping rules are as follows.

Type	Tables	Stored Procedures	Functions
Read functions	Yes	Yes	No
Navigation functions	Yes	Yes	No
Private functions	No	No	No
Protected functions	No	No	No
Procedures	No	No	No
Library data service functions (non-database)	Yes	Yes	No
Library data service database functions	No	No	Yes

See Also

Concepts

- Understanding SQL Maps

How Tos

- Map Functions and Procedures to SQL Objects

Reference

- Constraints on Publishing Data Service Objects to SQL

Constraints on Publishing Data Service Objects to SQL

This page last changed on Feb 26, 2008.

There are some semantic and structural constraints to publishing data service objects to SQL.

Semantic constraints include some general types of objects as private functions.

See Mapping Rules for a matrix showing publishable ALDSP object types and their corresponding SQL object types.

The following table outlines the structural constraints on publishing data service artifacts to SQL.

Limitation	Discussion
Limitation affecting all SQL objects	**Limitations in this section affect publication to any type of SQL object.**
Functions referring to types that are neither simple nor elements	Examples of such types include item, node, and attribute.
Functions with simple types that have no corresponding SQL type	The simple type on the XQuery side must correspond with a JDBC-supported SQL type, such as QName for example.
Functions with anonymous element types	Functions containing elements where the name is not defined are not mapable. For example: `declare function f() as element()`

Functions declarations using recursive XML types	For example, a function declaration with a complex type (PersonType) containing an element that is also of type PersonType is not mapable, as shown by the following:
	
	`<element name="PERSON" type="tns:PersonType"/>`
	
	`<complexType name="PersonType">`
	`  <sequence>`
	`  <element name="first_name" type="string"/>`
	`  <element name="last_name" type="string"/>`
	`  <element name="contact" type="tns:PersonType"/>`
	`  </sequence>`
	`</complexType>`

XML types with content models containing wildcards	XML wildcards include: • xs:any • xs:anyAttribute

XML types with mixed content	An example of a document containing mixed content is:
	
	`<a>`
	`  <child/>`
	`  this is simply text`
	`  <child/>`
	`</a>`

Limitations affecting publishing as a SQL Table	**Limitations in this category affect publishing as SQL tables**
Functions with parameters	Functions with parameters can be mapped as stored procedures.
Functions containing simple return types	Functions containing simple return types can be mapped as SQL functions.
Functions containing any non-tabular element type	See How Non-Tabular Element Types Affect the Ability to Publish Functions as SQL Objects. Also applies to stored procedures.

Functions with any AtomicType types	Also applies to stored procedures.
Limitations affecting publishing as a stored procedure	**Limitations in this category affect publishing as a stored procedure**
Functions accepting element parameter types	These functions cannot be published as stored procedures.
Functions containing a sequence of simple return types, such as xs:string*	The function declaration is not eligible. For example: `declare function f($p as xs:string*) as xs:int`
Functions with anyAtomicType types	Also applies to tables.
Functions with any non-tabular element types	See How Non-Tabular Element Types Affect the Ability to Publish Functions as SQL Objects. Also applies to tables.
Limitations affecting publishing as a SQL Function	**Limitations in this category affect publishing as a SQL functions**
Function with a sequence parameter type and an arity greater than 1.	An example shows xs:int* as the sequence parameter type: `declare function f($p as xs:int*, $q as xs:string) as xs:int`

Functions with element types

```
declare function f ($p as element(e)) as xs:int
```

How Non-Tabular Element Types Affect the Ability to Publish Functions as SQL Objects

The structure of a data service function determines whether it can be mapped to an SQL object or not. For example, a parameterized function cannot be published as an SQL table since by definition SQL tables do not take parameters. Some structural constraints are practically self-evident; others are less obvious.

A quick way to determine if a particular function can be published to a particular type of SQL object is to drag the function to a SQL object table, stored procedure, or functions folder. Even if the function is grayed out — meaning that it cannot be published to any type of SQL object — an alert dialog will appear explaining why the selected object cannot be published.

For example, functions with non-tabular element types cannot be published as tables or stored procedures because XML output structure cannot be mapped to a normalized SQL table.

Underlying each data service is an XML type, or schema. Some XML types are readily mapped for JDBC use because they are — like SQL tables — two dimensional.

```
<CUSTOMER>
   <FIRST_NAME>
   <LAST_NAME>
   <CUSTOMER_ID>
</CUSTOMER>
```

When published as SQL, the table structure corresponds to the following:

FIRST_NAME	LAST_NAME	CUSTOMER_ID
Jack	Black	CUSTOMER1

As long as the object mapper can reduce the structure of the XML document to rank-one, the mapping

can occur. For example:

```
<CUSTOMER>
   <FIRST_NAME>
   <LAST_NAME>
   <CUSTOMER_ID>
   <CUSTOMER_ORDER>
      <ORDER_ID>
      <C_ID>
      <ORDER_DT>
   </CUSTOMER_ORDER>
</CUSTOMER>
```

is publishable as a table in the following form as long as there is *one or fewer* customer orders associated with the customer:

FIRST_NAME	LAST_NAME	CUSTOMER_ID	ORDER_ID	C_ID	ORDER_DT
Jack	Black	CUSTOMER1	ORDER_1_0	CUSTOMER1	2001-10-01

If, however, the CUSTOMER_ORDER type is unbounded, meaning that it can represent more than one order associated with a single customer, the structure no longer corresponds to a well-formed relational table and the mapping is not allowed.

See Also

Concepts

- Understanding SQL Maps

How Tos

- Map Functions and Procedures to SQL Objects

Reference

- Mapping Rules

Data Service Annotations

This page last changed on Feb 26, 2008.

Data Service Annotations

Concepts

Understanding Data Service Annotations

Reference

Data Service Annotations Schema

Understanding Data Service Annotations

This page last changed on Mar 11, 2008.

This section describes the syntax and semantics of annotations in data service documents developed within Studio. Data service documents define collections of XQuery functions and/or XQSE functions or procedures. Annotations are XML fragments comprising the character content of XQuery pragmas.

There are two types of annotations:

- **Global annotations.** These pertain to the entire entity or library data service document. Global annotations are also referred to as XDS or XFL annotations respectively.
- **Local annotations.** These pertain to a particular function. Local annotations are also referred to as function annotations.

Topics

See also:
Data Service Annotations Schema

XDS Annotations

There is a single XDS ("XQuery Data Service") annotation per entity data service document, which appears before all function annotations. The identifier for the pragma carrying the XDS annotation is xds. The qualified name of the top level element of the XML fragment corresponding to an XDS annotation has the local name xds and the namespace URI:

```
urn:annotations.ld.bea.com
```

Each entity data service is associated with a unique target type. The prime type of the return type of every read function must match its target type. The target type of an entity data service is an element type whose qualified name is specified by the targetType attribute of the xds element. It is defined in a schema file associated with the entity data service.

The contents of the top-level xds element is a sequence of the following blocks of properties:

- General Properties
- Data Access Properties
- Target Type Properties
- Key Properties
- Relationship Properties
- Update Properties
- Security Properties

The following excerpt provides an example of an XDS annotation. In this case, the target type t:CUSTOMER associates the entity

data service with a t:CUSTOMER type in a schema file.

```
(::pragma xds <x:xds xmlns:x="urn:annotations.ld.bea.com" targetType="t:CUSTOMER" xmlns:t="ld:oracleDS/
CUSTOMER">

<author>Joe Public</author>
<relationalDB name="OracleDS"/>

<field type="xs:string" xpath="FIRST_NAME">
        <extension nativeFractionalDigits="0" nativeSize="64"
                nativeTypeCode="12" nativeType="VARCHAR2"
                nativeXpath="FIRST_NAME"/>
                <properties nullable="false"/>
</field>

<field type="xs:string" xpath="LAST_NAME">
        <extension nativeFractionalDigits="0" nativeSize="64"
                nativeTypeCode="12" nativeType="VARCHAR2"
                nativeXpath="LAST_NAME"/>
        <properties nullable="false"/>
</field>

<field type="xs:string" xpath="CUSTOMER_ID">
        <extension nativeFractionalDigits="0" nativeSize="64"
                nativeTypeCode="12" nativeType="VARCHAR2"
                nativeXpath="CUSTOMER_ID"/>
        <properties nullable="false" nativeKey="true"/>
</field>

<field type="xs:dateTime" xpath="CUSTOMER_SINCE">
        <extension nativeFractionalDigits="0" nativeSize="7"
                nativeTypeCode="93" nativeType="DATE"
                nativeXpath="CUSTOMER_SINCE"/>
        <properties nullable="false"/>
</field>

<field type="xs:string" xpath="EMAIL_ADDRESS">
        <extension nativeFractionalDigits="0" nativeSize="32"
                nativeTypeCode="12" nativeType="VARCHAR2"
                nativeXpath="EMAIL_ADDRESS"/>
        <properties nullable="false"/>
</field>

<key name="CUSTOMER_ID"/>

<relationshipTarget roleName="CUSTOMER_ORDER" roleNumber="2"
        XDS="ld:oracleDS/CUSTOMER_ORDER.xds" minOccurs="0"
        maxOccurs="unbounded" opposite="CUSTOMER"/>
</x:xds>::-)
```

General Properties

There are two types of general XDS properties:

- Standard Document Properties
- User-Defined Properties

Standard Document Properties

You can specify a set of standard document properties consisting of optional XML elements containing information pertaining to the

author, creation date, or version of the document. You can also use the optional element named "documentation" to specify related documentation. The names and types of the elements in the standard document properties block, as well as examples of their use, are shown in the table below.

Standard Document Properties

Element Name	Element Type	Optional	Example Instance
author	xs:string	Yes	`<author>J. Public</author>`
creationDate	xs:date	Yes	`<creationDate>2004-05-31</creationDate>`
version	xs:decimal	Yes	`<version>2.2<version>`
documentation	xs:string	Yes	`<documentation> Models an online Customer </documentation>`

User-Defined Properties

In addition to the standard properties, you can specify custom properties pertaining to the entire data service document using a sequence of zero (0) or more "property" elements. Each property element must be named using its "name" attribute and may contain any string content. For example:

```
<property name="data-refresh-rate">week</property>
```

Data Access Properties

A data service may be used to model access to an external data source or to model a transformation on top of one or more data sources or other transformations. Data services modeling external data sources are referred to as *physical*. Transformation data services not representing a particular data source are referred to as *logical*.

The block of data access properties allows each data service to define whether it is physical or not. When a data service is physical, the data access annotation describes the type of the external source being accessed by its external functions (there may be a single external source per data service) and its connection properties. When a data service is logical, the data service is designated as a user-defined view, and no connection information is required.

The following types of physical data services are supported:

- Relational
- Web service
- Java function
- Delimited content
- XML content

The following sections describe the data access annotation for the physical data service types, as well as for data services that are designated as user-defined views. You can specify only one of these annotations in each data service. If no annotation is provided, the data service is considered a user-defined view.

Relational Data Service Annotations

The data access annotation for a relational data service consists of the element relationalDB with two required attributes, described in the following table:

Required Attributes for the relationalDB Element

Attribute	Description
name	The JNDI name by which the external relational data source has been registered with the application server.
providerId	The identifier of the ALDSP relational provider in use for the specified relational data source.

```
<relationalDB name="OracleDS" providerId="Oracle-9"/>
```

In addition, the relationalDB element can contain the following optional parts:

- An optional element, named "properties", that exposes relational provider-specific attributes, such as the values of specific settings of the Relational Database Management System (RDBMS) represented by the relational source.
- An optional attribute, named sourceBindingProviderClassName, that specifies the transformation used to determine the relational source that should be used at system runtime in the place of the statically defined source.

Source Binding Provider

The value of the optional sourceBindingProviderClassName attribute should be bound to the fully-qualified name of a user-defined Java class implementing the interface:

```
com.bea.ld.bindings.SourceBindingProvider
```

defined by the following:

```
package com.bea.ld.bindings;
public interface SourceBindingProvider
{
        public String getBinding(String genericLocator, boolean isUpdate);
}
```

The user-defined implementation should provide the transformation that, given the statically configured relational source name (parameter genericLocator) and a Boolean flag indicating whether the relational source is accessed in query or update mode (parameter isUpdate), determines the name of the relational source name used by the system at runtime.

You can use this transformation mechanism to perform credential mapping. In this case, a single set of query or update operations to be performed in the name of two distinct users *U1* and *U2* against the same statically-configured relational source *R0*, is executed against two distinct relational sources *R1* and *R2* respectively (where all sources *R0*, *R1*, *R2* represent the same RDBMS and the security policies applied to the connection credentials used for *R1* and *R2* correspond to the security policies applied to the

application credentials of user *U1* and *U2*, respectively).

> ⚠ You should set the source binding provider name uniformly across all relational data services sharing the same relational source JNDI name. Although this restriction is not enforced, its violation could result in unpredictable behavior at runtime.

Web Service Data Service Annotations

The data access annotation for a data service based on a Web service consists of the empty element webService with two required attributes, described in the following table:

Required Attributes for the webService Element

Attribute	Description
wsdl	A valid http: or ld: URI pointing to the location of the WSDL file containing the definition of the external Web service source.
targetNamespace	A valid URI that is identical to the targetNamespace URI of the WSDL.

Example:

```
<webService targetNamespace="urn:GoogleSearch"
        wsdl="ld:google/GoogleSearch.wsdl"/>
```

In addition, if the physical data service models an ALSB proxy service, the webService element can carry the following optional attributes:

Optional Attributes for the webService Element

Attribute	Description
sbProxyServiceName	The name of the ALSB proxy service.
sbTransportProtocol	The name of the protocol used by the proxy service. Valid values are: t3, iiop, http, t3s, iiops or https.

Java Function Data Service Annotations

The data access annotation for a Java function data service consists of the empty element javaFunction with a single required attribute named class, whose value should be set to the fully qualified name of the Java class serving as the external source.

Example:

```
<javaFunction class="com.example.Test"/>
```

Delimited Content Data Service Annotations

The data access annotation for a delimited content data service is the empty element delimitedFile, accepting the optional attributes described in the following table:

Optional Attributes for the delimitedFile Element

Attribute	Description
file	A valid URI pointing to the location of the delimited file.
schema	A valid URI pointing to the location of the XML schema file defining the type (structure) of the delimited contents. If absent, the schema is derived based on the contents.
inferredSchema	Specifies whether the schema was inferred or provided by the user. The default value is false.
delimiter	The string used as the delimiter. If absent, the fixedLength attribute should be present.
fixedLength	The fixed length of the tokens contained in fixed length content. If absent, the delimiter attribute should be present.
hasHeader	A Boolean flag indicating whether the first line of the content should be interpreted as a header. The default value is false.

Example:

```
<delimitedFile schema="ld:df/schemas/ALL_TYPES.xsd" hasHeader="true"
        delimiter="," file="ld:df/ALL_TYPES.csv"/>
```

XML Content Data Service Annotations

The data access annotation for an XML content data service is the empty element xmlFileaccepting the attributes described in the following table:

Attributes for the xmlFile Element

Attribute	Description
file	(Optional) A valid URI pointing to the location of the XML file.
schema	A valid URI pointing to the location of the XML schema file defining the type (structure) of the XML contents.

Example:

```
<xmlFile schema="ld:xml/somewhere/CUSTOMER.xsd"
        file="ld:xml/CUSTOMER_NESTED.xml"/>
```

User Defined View XDS Annotations

The data access annotation for a user-defined view data service is also known as a *logical* data service. It consists of the single empty element:

```
userDefinedView
```

Example:

```
<userDefinedView/>
```

Target Type Properties

The optional block of target type properties enables you to annotate simple valued fields in the target type of the entity data service with native type information pertaining to the following:

- The type of the corresponding field in the underlying external source (applicable only to data source data services)
- Information about the field's properties with respect to its update behavior. Each annotated field is represented by the element named "field"with two required attributes, described in the following table:

Required Attributes for the field Element

Attribute	Description
xpath	An XPath value pointing to the field
type	The qualified name of the field's simple XML schema or XQuery type.

The following excerpt provides an example of a field element definition:

```
<field type="xs:string" xpath="FIRST_NAME">
        <extension nativeSize="64" nativeTypeCode="12" nativeType="VARCHAR2"
                nativeXpath="FIRST_NAME"/>
        <properties nullable="false"/>
</field>
```

Native Type Properties

Each "field" element can contain an optional "extension" element that accepts the optional attributes described in the following table:

Optional Attributes for the extension Element

Attribute	Description
nativeXpath	A native XPath value pointing to the corresponding native field in the external source.
nativeType	The native name of the native type of the corresponding native field, as it is known to the external source.
nativeTypeCode	The native type code of the native type of the corresponding native field, as it is known to the external source. In the case of relational sources, this is the type code as reported by JDBC.
nativeSize	The native size of the native type of the corresponding native field, as it is known to the external source. In the case of relational sources, this is the size as reported by JDBC.
nativeFractionalDigits	The native scale of the native type of the corresponding native field, as it is known to the external source. In the case of relational sources, this is the scale as reported by JDBC.
nativeKey	A Boolean value indicating whether the field participates in the native record's key. The default is false.

Update-related Type Properties

Each "field" element can also contain an optional "properties" element that accepts the optional attributes described in the following table:

properties Element Optional Attributes

Attribute	Description
immutable	A Boolean value specifying whether the field is immutable (read-only) or not. The default value is false.
nullable	A Boolean value specifying whether the field accepts null values or not. The default value is false.

Key Properties

The optional block of key properties enables you to specify an identity constraint (key) on the entity data service target type. An identity constraint for an entity data service is represented by the element "key" along with an XML schema specifying the key type.

The "key" element accepts a required attribute "type", whose value should be bound to the qualified name of the element type defining the locations of the data fields comprising the key. The key type should in turn be specified by an XML schema imported by the data service.

The "key" element may also carry the following optional attributes:

key Element Optional Attributes

Attribute	Description
name	Serves as the key alias. Might be used as a user-friendly description of the semantic constraints expressed by the key.
inferred	A Boolean value specifying whether the key was auto-derived or user-defined. The default is true.
inferredSchema	A Boolean value specifying whether the key schema was auto-derived or user-defined. The default is true.

In most cases, the identity constraint refers to the collection of data bindings returned by the entity data service's read functions, with each binding's type being the data service target type. In the case that a data service returns an XML document, the collection on which the identity constraint may be specified is normally defined by some element nested within the document element. In such a case, the "key" element contains an optional "selector" element that is used to specify the collection. The "selector" element carries a required "xpath" attribute, whose value is an XPath value pointing to the nested element defining the collection root. The XPath forms accepted by this attribute are simplified XPaths, using only the element or attribute axes and no predicates.

The following excerpt provides an example of a "key" element definition:

```
<key name="CUSTOMER_ID"/>
        <selector xpath="CUSTOMER"/>
</key>
```

Relationship Properties

The optional block of relationship properties enables you to specify a set of relationship targets. A relationship target of an entity data service is an entity data service with which first service maintains a unidirectional or bidirectional relationship. Unidirectional relationships are realized through one or more *navigate*functions in the first data service that returns one or more instances of objects of the second service target type. Bidirectional relationships require that reciprocal functions are present in the second data service as well.

A relationship target is represented by the element relationshipTarget that accepts the attributes described in the following table:

Attributes for the relationshipTarget Element

Attribute	Description
roleName	A string that uniquely identifies the relationship target inside the data service.
roleNumber	(Optional) Either 1 or 2 (default is 1). The roleNumber specifies the index of the relationship target within the relationship.
XDS	The AquaLogic Data Services Platform URI of the data service serving as the relationship target.
minOccurs	(Optional) The minimum cardinality of relationship target instances participating in this relationship. Possible values are all non-negative integers and the empty string. The default value is the empty string.
maxOccurs	(Optional) The maximum cardinality of relationship target instances participating in this relationship. Possible values are all positive integers, the string unbounded, and the empty string. The default is the empty string.
opposite	(Optional) String attribute that indicates the reciprocal relationship target in the case of bidirectional relationships. The value of this attribute is the identifier used to identify this data service as a relationship target in the data service identified by the value of the XDS attribute.

Additionally, the relationshipTarget element can itself contain the element "relationship" which in turn contains the nested element "description" that contains a human readable description about the relationship.

The following excerpt provides an example of a relationshipTarget element definition:

```
<relationshipTarget roleName="CUSTOMER_ORDER" roleNumber="2"
        XDS="ld:oracleDS/CUSTOMER_ORDER.xds" minOccurs="0"
        maxOccurs="unbounded" opposite="CUSTOMER"/>
```

Update Properties

The optional block of update properties enables you to specify a set of properties that establish certain policies about updating an entity data service's underlying sources. In particular, you can specify the following policies:

- The fields to use for optimistic locking purposes.

Optimistic Locking Fields

SDO update assumes optimistic locking transactional semantics. The data service being updated can specify the fields that should be checked for updates during the interim using the empty element optimisticLockingFields that accepts one of the following as its content:

- An empty element, named updated, to specify only updated fields.
- An empty element, named projected, to specify all projected fields.
- One or more elements, named "field", that accept a required string-valued attribute named name to specify user-specified fields.

The following excerpt provides an example of a functionForDecomposition element definition:

```
<optimisticLockingFields>
        <updated/>
</optimisticLockingFields>
```

Security Properties

You can use a data service to define one or more user-defined, logical protected resources.
The element secureResources, containing one or more string-valued elements named secureResource, can be used for this purpose.
For example:

```
<secureResources>
        <secureResource>MyResource</secureResource/>
        <secureResource>MyOtherResource</secureResource/>
</secureResources>
```

You can link a logical resource defined using this syntax to a user-provided security policy using the AquaLogic Data Services Console. Query content can inquire about a user's ability to access a logical resource using the built-in function isAccessAllowed().

Function Annotations

There is a single function annotation per data service function or procedure, which appears before the function or procedure declaration in the document. The identifier for the pragma carrying the function annotation is "function". The qualified name of the top level element of the XML fragment corresponding to a function annotation has the local name "function" and the namespace URI urn:annotations.ld.bea.com.

Modeling Kind

Each entity data service function or procedure is classified using one of the following categories:

- Create procedure
- Read function
- Update procedure
- Delete procedure
- Navigate function
- Library function or procedure

The classification of a data service method is determined by the value of the optional attribute "kind" in the function element, which accepts the values create, read, update, delete, navigate, or library to denote the corresponding categories. The default value is library.

Each library data service function or procedure is always of kind library.

The prime type of the return type of a read function must match the target type of the entity data service. In addition, the function element for a navigate function must carry a string-valued attribute returns whose value must match the role name of a relationship target defined in the data service. Moreover, the prime type of the return type of a navigate function must match the target type of the data service serving as the relationship target.

An operation designated as a procedure has in the general case side-effects. In other words, its invocation entails modifications of the state of the affected data sources. Therefore, a procedure may not be referenced by AquaLogic Data Services Platform functions.

A library function residing in a relational database function library data service file is always external. It may not be invoked directly by clients. Instead, it should be referenced by other data service functions or ad-hoc queries.

Visibility

Functions or procedures may also be classified based on their visibility using one of the following categories:

- Public
- Protected
- Private

The classification of a data service method is determined by the value of the optional attribute "visibility" in the function element, which accepts the values public, protected, or private to denote the corresponding categories. The default value is protected.

Public methods are accessible by ALDSP dataspace clients as well as other data services within the dataspace.

Protected methods are not accessible by ALDSP dataspace clients but can be accessed by other data services within the dataspace.

Private methods may be accessed only by other methods within the data service in which they are defined.

Primary

The optional boolean attribute "isPrimary" may also be used to classify entity data service methods as primary or non-primary. The default value is false.

This property is applicable only to create, update and delete procedures or read functions.

In the case of a procedure, when this property is set to true, it denotes that the procedure should be the one to be automatically used by the update maps of logical data services directly depending on the data service defining the procedure, in order to perform the corresponding update operation (i.e. create, update or delete).

In the case of a read function, when this property is set to true, it denotes that the read function should be the one to be used to infer the data service update map.

There may exist at most one primary method of each kind specified within an entity data service.

URI

Finally, the namespace URIs of the qualified names of all the functions and/or procedures in a data service must specify the location of the data service document in the ALDSP repository. For example:

```
ld:{directory path to data service folder}/{data service file name without extension}
```

The function element accepts the additional optional attributes described in the table below:

Optional Attributes for the function Element

Attribute	Description

nativeName	Applicable to data source functions or procedures, nativeName is the name of the function or procedure as it is known to the external source. In the case of relational sources, for example, it corresponds to the table name.
nativeLevel1Container	Applicable to data source functions or procedures that represent external sources employing hierarchical containment schemes; nativeLevel1Container is the name of the top-level native container, as it is known to the external source.In the case of relational sources, for example, it corresponds to the catalog name, whereas, in the case of Web service sources, it corresponds to the service name.
nativeLevel2Container	Applicable to data source functions or procedures that represent external sources employing hierarchical containment schemes; nativeLevel2Containeris the name of the second-level native container, as it is known to the external source. In the case of relational sources, for example, it corresponds to the schema name. In the case of Web service sources, it corresponds to the port name.
nativeLevel3Container	Applicable to data source functions or procedures that represent external sources employing hierarchical containment schemes; nativeLevel3Containeris the name of the top-level native container, as it is known to the external source. In the case of relational sources, for example, it corresponds to the stored procedure package name.
style	Applicable to data source functions or procedures, style is a native qualifier by which the function is known to the external source (e.g. table, view, storedProcedure, or sqlQuery for relational sources; rpc or document for Web services).
roleName	Applicable to navigate functions, roleName should match the value of the roleName attribute of the relationshipTarget implemented by the function.

The content of the top-level function element is a sequence of the following blocks of properties:

- General Properties
- UI Properties
- Cache Properties
- Transaction Properties
- Behavioral Properties
- Signature Properties
- Native Properties
- Implementation Properties

The following excerpt provides an example of a function annotation:

```
( ::pragma function
<f:function xmlns:f="urn:annotations.ld.bea.com" kind="read" nativeName="CUSTOMER"
nativeLevel2Container="RTL" style="table">
<nonCacheable/>
</f:function>::-)
```

General Properties

All standard document properties and user-defined properties defined in Standard Document Properties and User-Defined Properties are applicable to function annotations.

UI Properties

A set of user interface properties may be introduced by the XQuery Editor to persist location information about the graphical

components representing the expression in the function body. UI properties are represented by the element uiProperties which accepts a sequence of one or more elements, named component, as its content. Each "component" element accepts the attributes described in the following table.

Attributes for the component Element

Attribute	Description
identifier	An identifier for the UI component.
minimized	A Boolean flag indicating whether the UI component has been minimized or not.
x	The x-coordinate for the UI component.
y	The y-coordinate for the UI component.
w	The width of the UI component.
h	The height of the UI component.
viewPosX	The x-coordinate of the scrollbar position of the component.
viewPosY	The y-coordinate of the scrollbar position of the component.

In addition, each "component" element may optionally contain one or more treeInfo elements containing information about the tree representation of the types pertaining to the component. In the absence of the above property, the query editor uses the default layout.

Cache Properties

You can use the optional block of cache properties to specify whether a function can be cached or not. You should specify a function whose results for the same set of arguments are intrinsically highly volatile as non-cached. On the other hand, you should specify a function whose results for the same set of arguments are either fixed or remain unchanged for a period of time as cacheable.

This property of a function is represented by the empty element nonCacheable. In the absence of the nonCacheable element, a function is considered to be potentially cacheable. The following excerpt provides an example:

```
<nonCacheable/>
```

Transaction Properties

You can use the optional block of transaction properties to specify whether a procedure can participate in a transaction or not. This property is applicable only to physical procedures bound to external data sources of type Java or ALSB proxy service. A transactional procedure should rollback its effects if the overall transaction, in which it participates, fails.

This property is represented by the empty element nonTransactional. In the absence of the nonTransactional element, a procedure is considered to be transactional. The following excerpt provides an example:

```
<nonTransactional/>
```

Behavioral Properties

The optional block of behavioral properties allows you to provide information related to known associations between a function's input and its output, or across two or more functions. In particular, the user may specify the following:

- Inverse Functions
- Equivalent Transforms

Inverse Functions

Given an XQuery function *f*, the optional block of inverse functions may be used in order to denote a function *g*, defined over the range of *f*, that, when composed with *f* (i.e. *g*(*f*)), renders one of the parameters of *f*. If *f* has multiple parameters, an inverse function may be defined for each one of its parameters.

The inverse functions block is represented by an optional element, named inverseFunctions, which accepts as its content a sequence of empty elements, named inverseFunction. Each inverseFunction element accepts the following attributes: **parameterIndex.** Optional attribute denoting the index of the parameter for which the inverse function is defined. The index of the first parameter is assumed to be 1. It may be omitted if the function being annotated has a single parameter.

- **name.** Required attribute denoting the fully-qualified name of the inverse function.

  Both the annotated and the inverse function must be either built-in or external XQuery functions.

The following excerpt provides an example of an inverseFunctions element definition:

```
<inverseFunctions>
        <inverseFunction parameterIndex="2" name="p:MyInverse" xmlns:p ="urn:test"/>
</inverseFunctions>
```

Equivalent Transforms

Given an XQuery function: *f*, the optional block of equivalent transforms may be used in order to denote a pair of functions_C_ and *C'* with identical signatures and equivalent semantics, that accept *f* as one of their parameters. In simple terms, the equivalence is perceived to mean that each occurrence of *C(...,f,...)* may be safely substituted with: *C'(...,f,...)*.

The equivalent transforms block is represented by an optional element, named equivalentTransforms, which accepts as its content a sequence of empty elements, named pair. Each pair element accepts the following required attributes:

- **source.** Denotes the fully qualified name of the source transform (i.e.: *C*).
- **target.** Denotes the fully qualified name of the target transform (i.e.: *C'*).
- **arity.** Denotes the (common) arity of the source and target transforms.

  The source transform may be either a built-in or external function. Both source and target transforms must not be defined as invertible functions.

The following excerpt provides an example of an equivalentTransforms element definition:

```
<equivalentTransforms>
        <pair source="p:sourceFunction_1" target="p:targetFunction_1"   arity="1" xmlns:p ="urn:test1"/>
        <pair source="q:sourceFunction_2" target="q:targetFunction_2"   arity="3" xmlns:q="urn:test2"/>
</equivalentTransforms>
```

Polymorphic Functions

A library function residing in a relational database function library data service may be designated as polymorphic if its actual return type can be determined from the actual type of one of its parameters. A polymorphic function is annotated by an optional element, named isPolymorphic, which accepts as its content an empty element, named parameter. The parameter element accepts the following optional attribute:

index. Denotes the index of the parameter whose actual type determines the function's actual return type. The index of the first parameter is assumed to be 1. It may be omitted if the function being annotated has a single parameter.

The following excerpt provides an example of an equivalentTransforms element definition:

```
<sPolymorphic>
        <parameter index = "2"/>
</sPolymorphic>
```

Signature Properties

You can use the optional block of signature properties to annotate the parameters of a data service function or procedure with additional information to that provided by the function signature. These properties are applicable to physical data service functions or procedures.

The signature properties block is represented by the element params which accepts a sequence of one or more elements, named param, as its content. Each param element is an empty element that accepts the optional attributes described in the following table:

param Element Optional Attributes

Attribute	Description
name	The name of the parameter, as it is known to the external source.
nativeType	The native type of the parameter, as it is known to the external source.
nativeTypeCode	The native type code of the parameter, as it is known to the external source.
xqueryType	The qualified name of the XML Schema or XQuery type used for the parameter.
kind	One of the following values: unknown, in, inout, out, return or result (applicable to stored procedures).

The following excerpt provides an example of a params element definition:

```
<params>
        <param nativeType="java.lang.String"/>
        <param nativeType="java.lang.int"/>
</params>
```

Native Properties

You can use native properties to further annotate a data source function or procedure based on the type of the external source that it represents. There are two types of native properties pertaining to relational and Web service sources respectively:

- SQL query properties
- SOAP handler properties

SQL Query Properties

The function annotation element of a function that represents a user-defined SQL query has its style attribute set to sqlQuery and accepts a nested element, named "sql". The sql element accepts string content that corresponds to the statement of the (possibly parameterized) SQL query that the function represents.

If required, the statement can be escaped inside a CDATA section to account for reserved XML characters (e.g. <, >, &). The sql element also accepts the optional attribute isSubquery whose boolean value indicates whether the SQL statement may be used as a nested SQL sub-query. If the attribute is absent, its value defaults to true.

The following excerpt provides an example of a sqlQuery element definition:

```
<sql isSubquery="true">
        SELECT t.FIRST_NAME FROM RTLALL.dbo.CUSTOMER t</sql>
```

SOAP Handler Properties

The "function" annotation element of a function or procedure that represents a Web service call accepts a nested element, named interceptorConfiguration. The interceptorConfiguration element accepts two required attributes, as described below:

Required Attributes for the interceptorConfiguration Element

Attribute	Description
fileName	The location of the file containing the configuration of the SOAP handler chains that are applicable to the Web service.
aliasName	The alias name by which the SOAP handler chain has been configured.

Implementation Properties

You can use implementation properties to specify that an external create, update or delete procedure is implemented by the update map of the data service in which it is defined.

The optional element "implementation" accepts the required empty element "updateTemplate" as its content.

The following excerpt provides an example:

```
<implementation>
        <updateTemplate/>
</implementation>
```

XFL Annotations

There is a single XFL ("XQuery Function Library") annotation per library data service document, which appears before any function annotation in the document. The identifier for the pragma carrying the XFL annotation is "xfl". The qualified name of the top level

element of the XML fragment corresponding to an XFL annotation has the local name:

```
xfl
```

and the namespace URI:

```
urn:annotations.ld.bea.com
```

The contents of the top-level xfl element is a sequence of the following blocks of properties.

- General Properties
- Data Access Properties

The following sections provide detailed descriptions of each block of properties, while the following excerpt provides an example of a XFL annotation, which may serve as a reference.

```
(::pragma xfl <x:xfl xmlns:x="urn:annotations.ld.bea.com">
<creationDate>2005-03-09T17:48:58</creationDate>
<webService targetNamespace="urn:GoogleSearch"
        wsdl="ld:google/GoogleSearch.wsdl"/>
</x:xfl>::-)
```

General Properties

The general properties applicable to an library data service document are identical to the general properties for an entity data service document, as described in General Properties.

Data Access Properties

Each library data service document defines one or more XQuery functions and/or XQSE functions or procedures that serve as library operations that can be used either inside other entiry or library data service documents.

Since library data service documents do not have a target type, the return types of the library functions found inside these document may differ from each other. In particular, a function inside a library data service document may return a value having a simple type (or any other type). Library data service functions can be external data source functions or user-defined.

The following types of library data service documents are supported:

- Relational (physical)
- Web service (physical)
- Java function (physical)
- Relational database function (physical)
- User-defined view (logical)

You can specify only one of the annotations in each library data service. If no annotation is provided, the library data service is considered a user-defined view.

The data access properties for Relational, Web service, Java function, and user-defined view library data service documents are the

same as the corresponding properties for entity data service documents, as described above.

A relational database function library data service contains native functions, either database vendor-provided or user-defined in the database, from one or more relational data sources, modeled as external XQuery functions.

The data access annotation for a relational database function library data service comprises an element named customNativeFunctions with a single child element, named relational, whose content is a sequence of one or more elements named dataSource. Each dataSource element contains a single text value, which should be set to the JNDI name by which the external relational source has been registered with the application server.

Here is an example:

```
<customNativeFunctions>
<relational>
        <dataSource>oracleDS1</dataSource>
        <dataSource>oracleDS2</dataSource>
        </relational>
</customNativeFunctions>
```

Security Properties

The same as in entity data services.

Data Service Annotations Schema

This page last changed on Feb 26, 2008.

```
<?xml version="1.0"?>
<xs:schema targetNamespace="urn:annotations.ld.bea.com" xmlns:tns="urn:annotations.ld.bea.com" xmlns:
xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="unqualified" attributeFormDefault="unqualified">
  <!--==================-->
  <!--  XDS annotation  -->
  <!--==================-->
  <xs:element name="xds">
    <xs:complexType>
      <xs:sequence>
        <!-- document properties -->
        <xs:element name="author" type="xs:string" minOccurs="0"/>
        <xs:element name="comment" type="xs:string" minOccurs="0"/>
        <xs:element name="creationDate" type="xs:dateTime" minOccurs="0"/>
        <xs:element name="documentation" type="xs:string" minOccurs="0"/>
        <xs:element name="version" type="xs:decimal" minOccurs="0"/>
        <!-- user defined properties -->
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
          <xs:element name="property">
            <xs:complexType>
              <xs:simpleContent>
                <xs:extension base="xs:string">
                  <xs:attribute name="name" type="xs:string"/>
                </xs:extension>
              </xs:simpleContent>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <!-- data access properties -->
        <xs:choice>
          <!-- choice 1: java functions -->
          <xs:element name="javaFunction">
            <xs:complexType>
              <xs:attribute name="class" type="xs:string" use="required"/>
            </xs:complexType>
          </xs:element>
          <!-- choice 2: web services -->
          <xs:element name="webService">
            <xs:complexType>
              <xs:attribute name="wsdl" type="xs:anyURI" use="required"/>
              <xs:attribute name="targetNamespace" type="xs:anyURI" use="required"/>
              <xs:attribute name="sbProxyServiceName" type="xs:string"/>
              <xs:attribute name="sbTransportProtocol" type="tns:SBTransportProtocolType"/>
            </xs:complexType>
          </xs:element>
          <!-- choice 3: relational sources -->
          <xs:element name="relationalDB">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="properties" minOccurs="0">
                  <xs:complexType>
                    <xs:anyAttribute processContents="lax" />
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
              <xs:attribute name="name" type="xs:string" use="required"/>
              <xs:attribute name="providerId" type="xs:string" />
              <xs:attribute name="dbType" type="xs:string"/>
              <xs:attribute name="dbVersion" type="xs:string"/>
              <xs:attribute name="driver" type="xs:string"/>
              <xs:attribute name="uri" type="xs:string"/>
```

```
            <xs:attribute name="username" type="xs:string"/>
            <xs:attribute name="password" type="xs:string"/>
            <xs:attribute name="SID" type="xs:string"/>
            <xs:attribute name="sourceBindingProviderClassName" type="xs:string"/>
          </xs:complexType>
        </xs:element>
        <!-- choice 4: delimited files -->
        <xs:element name="delimitedFile">
          <xs:complexType>
            <xs:attribute name="file" type="xs:anyURI"/>
            <xs:attribute name="schema" type="xs:anyURI"/>
            <xs:attribute name="inferredSchema" type="xs:boolean" default="false"/>
            <xs:attribute name="delimiter" type="xs:string"/>
            <xs:attribute name="fixedLength" type="xs:positiveInteger"/>
            <xs:attribute name="hasHeader" type="xs:boolean" default="false"/>
          </xs:complexType>
        </xs:element>
        <!-- choice 5: XML files -->
        <xs:element name="xmlFile">
          <xs:complexType>
            <xs:attribute name="file" type="xs:anyURI"/>
            <xs:attribute name="schema" type="xs:anyURI" use="required"/>
          </xs:complexType>
        </xs:element>
        <!-- choice 6: user defined view -->
        <xs:element name="userDefinedView" minOccurs="0"/>
        <!-- choice 7: nothing, defaults to userDefinedView -->
        <xs:sequence/>
      </xs:choice>
      <!-- field annotations -->
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element name="field">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="extension" minOccurs="0">
                <xs:complexType>
                  <xs:sequence minOccurs="0">
                    <xs:element name="autoNumber">
                      <xs:complexType>
                        <xs:attribute name="type" type="tns:autoNumberType" use="required"/>
                        <xs:attribute name="sequenceObjectName" type="xs:string"/>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                  <xs:attribute name="nativeXpath" type="xs:string"/>
                  <xs:attribute name="nativeType" type="xs:string"/>
                  <xs:attribute name="nativeTypeCode" type="xs:int"/>
                  <xs:attribute name="nativeSize" type="xs:int"/>
                  <xs:attribute name="nativeFractionalDigits" type="tns:scaleType"/>
                  <xs:attribute name="nativeKey" type="xs:boolean" default="false"/>
                  <!-- relational: autoNumber -->
                  <!-- relational: native column names and types -->
                </xs:complexType>
              </xs:element>
              <xs:element name="properties">
                <xs:complexType>
                  <xs:attribute name="immutable" type="xs:boolean" default="false"/>
                  <xs:attribute name="nullable" type="xs:boolean" default="false"/>
                  <xs:attribute name="transient" type="xs:boolean" default="false"/>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="xpath" type="xs:string" use="required"/>
            <xs:attribute name="type" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
```

```
        </xs:sequence>
        <!-- keys -->
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
          <xs:element name="key">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="selector" minOccurs="0">   <!-- defaults to . -->
                  <xs:complexType>
                    <xs:sequence>
                      <xs:element name="extension" minOccurs="0">
                        <xs:complexType>
                          <xs:attribute name="nativeXpath" type="xs:string" use="required"/>
                        </xs:complexType>
                      </xs:element>
                    </xs:sequence>
                    <xs:attribute name="xpath" type="xs:string" use="required"/>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
              <xs:attribute name="name" type="xs:string"/>
              <xs:attribute name="type" type="xs:QName"/>
              <xs:attribute name="inferred" type="xs:boolean" default="true"/>
              <xs:attribute name="inferredSchema" type="xs:boolean" default="true"/>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <!-- relationships -->
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
          <xs:element name="relationshipTarget">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="relationship" minOccurs="0">
                  <xs:complexType>
                    <xs:sequence>
                      <xs:element name="description" type="xs:string" minOccurs="0"/>
                    </xs:sequence>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
              <xs:attribute name="roleName" type="xs:string" use="required"/>
              <xs:attribute name="roleNumber" type="tns:roleType" default="1"/>
              <xs:attribute name="XDS" type="xs:string" use="required"/>
              <xs:attribute name="minOccurs" type="tns:allNNI" default="1"/>
              <xs:attribute name="maxOccurs" type="tns:allNNI" default="1"/>
              <xs:attribute name="opposite" type="xs:string"/>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <!-- SDO elements -->
        <xs:element name="functionForDecomposition" minOccurs="0">
          <xs:complexType>
            <xs:attribute name="name" type="xs:QName" use="required"/>
            <xs:attribute name="arity" type="xs:int" use="required"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="javaUpdateExit" minOccurs="0">
          <xs:complexType>
            <xs:attribute name="className" type="xs:string" use="required"/>
            <xs:attribute name="classFile" type="xs:string"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="optimisticLockingFields" minOccurs="0">
          <xs:complexType>
            <xs:choice>
              <xs:element name="updated">
                <xs:complexType/>
```

```
            </xs:element>
            <xs:element name="projected">
              <xs:complexType/>
            </xs:element>
            <xs:element name="field" minOccurs="0" maxOccurs="unbounded">
              <xs:complexType>
                <xs:attribute name="name" type="xs:string" use="required"/>
              </xs:complexType>
            </xs:element>
          </xs:choice>
        </xs:complexType>
      </xs:element>
      <!-- security -->
      <xs:element name="secureResources" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="secureResource" type="xs:NCName" minOccurs="0" maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="readOnly" minOccurs="0">
        <xs:complexType/>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="targetType" type="xs:QName" use="required"/>
  </xs:complexType>
</xs:element>
<!--==================-->
<!--  XFL annotation  -->
<!--==================-->
<xs:element name="xfl">
  <xs:complexType>
    <xs:sequence>
      <!-- document properties -->
      <xs:element name="author" type="xs:string" minOccurs="0"/>
      <xs:element name="comment" type="xs:string" minOccurs="0"/>
      <xs:element name="creationDate" type="xs:dateTime" minOccurs="0"/>
      <xs:element name="documentation" type="xs:string" minOccurs="0"/>
      <xs:element name="version" type="xs:decimal" minOccurs="0"/>
      <!-- user defined properties -->
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element name="property">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
                <xs:attribute name="name" type="xs:string"/>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <!-- data access properties -->
      <xs:choice>
        <!-- choice 1: java functions -->
        <xs:element name="javaFunction">
          <xs:complexType>
            <xs:attribute name="class" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
        <!-- choice 2: web services -->
        <xs:element name="webService">
          <xs:complexType>
            <xs:attribute name="wsdl" type="xs:anyURI" use="required"/>
            <xs:attribute name="targetNamespace" type="xs:anyURI" use="required"/>
            <xs:attribute name="sbProxyServiceName" type="xs:string"/>
            <xs:attribute name="sbTransportProtocol" type="tns:SBTransportProtocolType"/>
```

```
      </xs:complexType>
    </xs:element>
    <!-- choice 3: relational sources -->
    <xs:element name="relationalDB">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="properties" minOccurs="0">
            <xs:complexType>
              <xs:anyAttribute processContents="lax" />
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="providerId" type="xs:string" />
        <xs:attribute name="dbType" type="xs:string"/>
        <xs:attribute name="dbVersion" type="xs:string"/>
        <xs:attribute name="driver" type="xs:string"/>
        <xs:attribute name="uri" type="xs:string"/>
        <xs:attribute name="username" type="xs:string"/>
        <xs:attribute name="password" type="xs:string"/>
        <xs:attribute name="SID" type="xs:string"/>
        <xs:attribute name="sourceBindingProviderClassName" type="xs:string"/>
      </xs:complexType>
    </xs:element>
    <!-- choice 6: user defined view -->
    <xs:element name="userDefinedView" minOccurs="0"/>
    <!-- choice 7: nothing, defaults to userDefinedView -->
    <xs:sequence/>
    <!-- choice 8: custom native functions -->
    <xs:element name="customNativeFunctions">
      <xs:complexType>
        <xs:choice>
          <xs:element name="relational">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="dataSource" type="xs:string" maxOccurs="unbounded"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:choice>
      </xs:complexType>
    </xs:element>
  </xs:choice>
   <!-- field annotations -->
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element name="field">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="extension" minOccurs="0">
              <xs:complexType>
                <xs:sequence minOccurs="0">
                  <xs:element name="autoNumber">
                    <xs:complexType>
                      <xs:attribute name="type" type="tns:autoNumberType" use="required"/>
                      <xs:attribute name="sequenceObjectName" type="xs:string"/>
                    </xs:complexType>
                  </xs:element>
                </xs:sequence>
                <xs:attribute name="nativeXpath" type="xs:string"/>
                <xs:attribute name="nativeType" type="xs:string"/>
                <xs:attribute name="nativeTypeCode" type="xs:int"/>
                <xs:attribute name="nativeSize" type="xs:int"/>
                <xs:attribute name="nativeFractionalDigits" type="tns:scaleType"/>
                <!-- relational: autoNumber -->
                <!-- relational: native column names and types -->
              </xs:complexType>
```

```
                </xs:element>
                <xs:element name="properties">
                  <xs:complexType>
                    <xs:attribute name="immutable" type="xs:boolean" default="false"/>
                    <xs:attribute name="nullable" type="xs:boolean" default="false"/>
                    <xs:attribute name="transient" type="xs:boolean" default="false"/>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
              <xs:attribute name="xpath" type="xs:string" use="required"/>
              <xs:attribute name="type" type="xs:string" use="required"/>
            </xs:complexType>
          </xs:element>
        </xs:sequence>

    <xs:element name="secureResources" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="secureResource" type="xs:NCName" minOccurs="0" maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!--=======================-->
<!--  function annotation  -->
<!--=======================-->
<xs:element name="function">
  <xs:complexType>
    <xs:sequence>

      <!-- standard properties -->
      <xs:element name="author" type="xs:string" minOccurs="0"/>
      <xs:element name="comment" type="xs:string" minOccurs="0"/>
      <xs:element name="version" type="xs:decimal" minOccurs="0"/>
      <xs:element name="documentation" type="xs:string" minOccurs="0"/>

      <!-- user defined properties -->
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element name="property">
          <xs:complexType>
            <xs:simpleContent>
              <xs:extension base="xs:string">
                <xs:attribute name="name" type="xs:string"/>
              </xs:extension>
            </xs:simpleContent>
          </xs:complexType>
        </xs:element>
      </xs:sequence>

      <!-- UI properties -->
      <xs:element name="uiProperties" minOccurs="0">
        <xs:complexType>
          <xs:sequence maxOccurs="unbounded">
            <xs:element name="component">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="treeInfo" minOccurs="0" maxOccurs="unbounded">
                    <xs:complexType>
                      <xs:sequence>
                          <xs:element name="collapsedNodes" minOccurs="0">
                          <xs:complexType>
                            <xs:sequence>
                                  <xs:element name="collapsedNode" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
```

```
                    </xs:sequence>
                  </xs:complexType>
                </xs:element>
              </xs:sequence>
              <xs:attribute name="id" type="xs:string"/>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <xs:attribute name="identifier" type="xs:string"/>
        <xs:attribute name="minimized" type="xs:boolean" default="false"/>
        <xs:attribute name="x" type="xs:int"/>
        <xs:attribute name="y" type="xs:int"/>
        <xs:attribute name="w" type="xs:int"/>
        <xs:attribute name="h" type="xs:int"/>
        <xs:attribute name="viewPosX" type="xs:int"/>
        <xs:attribute name="viewPosY" type="xs:int"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
</xs:element>

<!-- sql statement -->
<xs:element name="sql" minOccurs="0">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="isSubquery" type="xs:boolean" default="true"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<!-- cache -->
<xs:element name="nonCacheable" minOccurs="0">
  <xs:complexType/>
</xs:element>

<!-- transactions -->
<xs:element name="nonTransactional" minOccurs="0">
  <xs:complexType/>
</xs:element>

<!-- optimization -->
<xs:element name="outputIsOrderedBy" minOccurs="0">
  <xs:complexType>
    <xs:sequence>
      <!-- absent for parameters whose order in the function signature
           coincides with their order in the order by list -->
      <xs:element name="parameter" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <!-- 1, 2, ... -->
          <xs:attribute name="index" type="xs:int" use="required"/>
          <!-- overrides default -->
          <xs:attribute name="mode" type="tns:orderingModeType"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="mode" type="tns:orderingModeType" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="inverseFunctions" minOccurs="0">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="inverseFunction" minOccurs="1" maxOccurs="unbounded">
```

```
        <xs:complexType>
          <!-- 1, 2, ... -->
          <xs:attribute name="parameterIndex" type="xs:int"/>
          <xs:attribute name="name" type="xs:QName" use="required"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="equivalentTransforms" minOccurs="0">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="pair" minOccurs="1" maxOccurs="unbounded">
        <xs:complexType>
          <xs:attribute name="source" type="xs:QName" use="required"/>
          <xs:attribute name="target" type="xs:QName" use="required"/>
          <xs:attribute name="arity"  type="xs:int"   use="required"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<!-- polymorphism -->
<xs:element name="isPolymorphic" minOccurs="0">
  <xs:complexType>
    <xs:choice>
      <xs:element name="parameter">
        <xs:complexType>
          <xs:sequence/>
          <!-- optional: defaults to 1 -->
          <xs:attribute name="index" type="xs:nonNegativeInteger"/>
        </xs:complexType>
      </xs:element>
    </xs:choice>
  </xs:complexType>
</xs:element>

<!-- signature: used by java functions and stored procedures -->
<xs:element name="params" minOccurs="0">
  <xs:complexType>
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="param">
        <xs:complexType>
          <xs:attribute name="name" type="xs:string"/>
          <xs:attribute name="nativeType" type="xs:string"/>
          <xs:attribute name="nativeTypeCode" type="xs:int"/>
          <xs:attribute name="xqueryType" type="xs:QName"/>
          <xs:attribute name="kind" type="tns:paramKindType"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!-- interceptor configuration: used by webservice SOAP interceptors -->
<xs:element name="interceptorConfiguration" minOccurs="0">
  <xs:complexType>
    <xs:attribute name="aliasName" type="xs:string" use="required"/>
    <xs:attribute name="fileName" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

<!-- implementation -->
<xs:element name="implementation" minOccurs="0">
  <xs:complexType>
    <xs:choice>
```

```
            <xs:element name="updateTemplate">
              <xs:complexType/>
            </xs:element>
          </xs:choice>
        </xs:complexType>
      </xs:element>
    </xs:sequence>

    <xs:attribute name="visibility" type="tns:functionVisibilityType" default="protected"/>
    <xs:attribute name="kind" type="tns:functionKindType" default="library"/>
    <xs:attribute name="isPrimary" type="xs:boolean" default="false"/>
    <xs:attribute name="roleName" type="xs:string"/>
    <xs:attribute name="nativeName" type="xs:string"/>
    <xs:attribute name="nativeLevel1Container" type="xs:string"/>
    <xs:attribute name="nativeLevel2Container" type="xs:string"/>
    <xs:attribute name="nativeLevel3Container" type="xs:string"/>
    <xs:attribute name="style" type="tns:functionStyleType"/>
  </xs:complexType>
</xs:element>
<!--=================-->
<!--  common types  -->
<!--=================-->
<xs:simpleType name="functionVisibilityType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="public"/>
    <xs:enumeration value="protected"/>
    <xs:enumeration value="private"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="functionKindType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="read"/>
    <xs:enumeration value="navigate"/>
    <xs:enumeration value="create"/>
    <xs:enumeration value="update"/>
    <xs:enumeration value="delete"/>
    <xs:enumeration value="library"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="functionStyleType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="table"/>
    <xs:enumeration value="view"/>
    <xs:enumeration value="storedProcedure"/>
    <xs:enumeration value="sqlQuery"/>
    <xs:enumeration value="document"/>
    <xs:enumeration value="rpc"/>
  </xs:restriction>
</xs:simpleType>
<!-- used by stored procedures -->
<xs:simpleType name="paramKindType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="unknown"/>
    <xs:enumeration value="in"/>
    <xs:enumeration value="inout"/>
    <xs:enumeration value="out"/>
    <xs:enumeration value="return"/>
    <xs:enumeration value="result"/>
  </xs:restriction>
</xs:simpleType>
<!-- used by maxOccurs in relationship -->
<xs:simpleType name="allNNI">
  <xs:union memberTypes="xs:nonNegativeInteger">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="unbounded"/>
```

```
          <xs:enumeration value=""/>
        </xs:restriction>
      </xs:simpleType>
    </xs:union>
  </xs:simpleType>
  <!-- used by relationships -->
  <xs:simpleType name="roleType">
    <xs:restriction base="xs:nonNegativeInteger">
      <xs:enumeration value="1"/>
      <xs:enumeration value="2"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="autoNumberType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="identity"/>
      <xs:enumeration value="sequence"/>
      <xs:enumeration value="userComputed"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="nullSortOrderType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="high"/>
      <xs:enumeration value="low"/>
      <xs:enumeration value="unknown"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="scaleType">
    <xs:union memberTypes="xs:int">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="null"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:union>
  </xs:simpleType>
  <xs:simpleType name="orderingModeType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="ascending"/>
      <xs:enumeration value="descending"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="stringListType">
    <xs:list itemType="xs:string"/>
  </xs:simpleType>
  <xs:simpleType name="dataSourcesType">
    <xs:restriction base="tns:stringListType">
       <xs:minLength value="1"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="SBTransportProtocolType">
      <xs:restriction base="xs:string">
          <xs:enumeration value="t3"/>
          <xs:enumeration value="iiop"/>
          <xs:enumeration value="http"/>
          <xs:enumeration value="t3s"/>
          <xs:enumeration value="iiops"/>
          <xs:enumeration value="https"/>
      </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Data Service Annotations

This page last changed on Feb 26, 2008.

Data Service Annotations

Concepts

Understanding Data Service Annotations

Reference

Data Service Annotations Schema