



BEA AquaLogic Data Services Platform™

Data Services Developer's Guide

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

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

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

Copyright

Copyright © 2005-2006 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software is protected by copyright, and may be protected by patent laws. No copying or other use of this software is permitted unless you have entered into a license agreement with BEA authorizing such use. This document is protected by copyright and may not be copied photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form, in whole or in part, without prior consent, in writing, from BEA Systems, Inc.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE DOCUMENTATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA SYSTEMS DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE DOCUMENT IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks or Service Marks

Copyright © 1995-2006 BEA Systems, Inc. All Rights Reserved. BEA, BEA JRockit, BEA WebLogic Portal, BEA WebLogic Server, BEA WebLogic Workshop, Built on BEA, Jolt, JoltBeans, SteelThread, Top End, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA AquaLogic, BEA AquaLogic Data Services Platform, BEA AquaLogic Enterprise Security, BEA AquaLogic Service Bus, BEA AquaLogic Service Registry, BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Liquid Data for WebLogic, BEA Manager, BEA MessageQ, BEA WebLogic Commerce Server, BEA WebLogic Communications Platform, BEA WebLogic Enterprise, BEA WebLogic Enterprise Platform, BEA WebLogic Enterprise Security, BEA WebLogic Express, BEA WebLogic Integration, BEA WebLogic Java Adapter for Mainframe, BEA WebLogic JDriver, BEA WebLogic Log Central, BEA WebLogic Network Gatekeeper, BEA WebLogic Personalization Server, BEA WebLogic Personal Messaging API, BEA WebLogic Platform, BEA WebLogic Portlets for Groupware Integration, BEA WebLogic Server Process Edition, BEA WebLogic SIP Server, BEA WebLogic WorkGroup Edition, Dev2Dev, Liquid Computing, and Think Liquid are trademarks of BEA Systems, Inc. BEA Mission Critical Support, BEA Mission Critical Support Continuum, and BEA SOA Self Assessment are service marks of BEA Systems, Inc.

All other names and marks are property of their respective owners.

March 16, 2006 1:49 pm

Contents

1. Introduction to Data Services

Data Services and the Enterprise	1-2
Data Access Integration Architecture	1-3
Data Services Platform Applications and Projects	1-5
DSP: Roles and Responsibilities	1-7
DSP: Typical Development Process.....	1-7
Examples, Samples, and Tutorials.....	1-8

2. Data Services Platform Projects and Components

DSP-Based BEA WebLogic Projects	2-2
Creating a DSP-based Application	2-2
Adding a DSP Project to an Existing BEA WebLogic Application	2-4
Major Components of a DSP Project.....	2-4
Using the WebLogic Workshop IDE.....	2-6
Survey of DSP Additions to WebLogic Workshop	2-9
Building and Deploying Applications, EARs, and SDO Mediator Clients	2-22
Building, Deploying, and Updating Applications.....	2-22
Creating the SDO Mediator API	2-24
Refactoring DSP Artifacts	2-25
Artifacts Supporting Refactoring	2-27
Setting Refactor Options	2-28
Impacts of Various Refactoring Operations	2-32

3. Obtaining Enterprise Metadata

Creating Data Source Metadata	3-1
Identifying DSP Procedures	3-4
Obtaining Metadata From Relational Sources	3-7
Importing Relational Table and View Metadata.	3-8
Importing Stored Procedure-Based Metadata	3-16
Using SQL to Import Metadata.	3-31
Importing Web Services Metadata.	3-37
Testing Metadata Import With an Internet Web Service URI	3-42
Importing Java Function Metadata	3-43
Supported Java Function Types	3-43
Adding Java Function Metadata Using Import Wizard	3-44
Creating XMLBean Support for Java Functions.	3-48
Inspecting the Java Source.	3-51
How Metadata for Java Functions Is Created.	3-54
Importing Delimited File Metadata	3-59
Providing a Document Name, a Schema Name, or Both	3-59
Using the Metadata Import Wizard on Delimited Files.	3-60
Importing XML File Metadata	3-63
XML File Import Sample.	3-63
Testing the Metadata Import Wizard with an XML Data Source	3-66
Updating Data Source Metadata	3-67
Considerations When Updating Source Metadata	3-68
Using the Update Source Metadata Wizard	3-68
Archival of Source Metadata	3-72

4. Designing Data Services

Data Services in the Enterprise	4-2
---	-----

Physical and Logical Data Services.	4-2
Data Service Functions	4-3
Data Service Design View Components	4-4
XML Types and Return Types.	4-7
Creating a Data Service	4-8
Adding a Function to Your Data Service.	4-10
Adding a Procedure to Your Data Service	4-11
Adding a Private Function to Your Data Service	4-11
Adding a Relationship to Your Data Service	4-11
Working with Logical Data Service XML Types	4-23
Creating an XML Type	4-25
Managing Your Data Service	4-26
Refactoring Data Service Functions.	4-27
Finding Usages of Data Services Platform artifacts	4-27
Setting Update Options	4-27
Adding Security Resources	4-31
Caching Functions	4-38
Notable Design View Properties	4-40

5. Modeling Data Services

Model-Driven Data Services.	5-3
Logical and Physical Data Models.	5-3
Rules Governing Model Diagrams	5-4
Building a Simple Model Diagram.	5-5
Displaying Relationships Automatically.	5-10
Generated Relationship Declarations in Source View	5-10
Modeling Logical Data	5-11
Building Data Service Relationships in Models.	5-12

Direction, Role, and Relationships	5-12
Working with Model Diagrams	5-16
Model Right-click Menu Options	5-17
Creating Relationships in Model Diagrams	5-19
Locating Data Services in Large Model Diagrams	5-19
Generating Reports on Your Models	5-20
Zoom Mode	5-22
Editing XML Types in Model Diagrams	5-22
How Changes to Data Services and Data Sources Can Impact Models	5-24
How Metadata Update Can Affect Models.	5-24

6. Working with the XQuery Editor

Role of the XQuery Editor	6-2
Data Source Representations.	6-4
XQuery Editor Options	6-5
Creating a New Data Service and Data Service Function	6-7
Key Concepts of Query Function Building	6-15
Data Sources	6-15
Source Schemas and Return Types	6-16
XQuery Editor Components	6-16
Setting Conditions	6-31
Using XQuery Functions	6-35
Setting Expressions	6-41
Managing Query Components.	6-41
Working With Data Representations and Return Type Elements.	6-42
Mapping to Return Types	6-43
Modifying a Return Type	6-47

7. Testing Query Functions and Viewing Query Plans

Running Queries Using Test View	7-1
Using Test View	7-3
Limiting Array Results	7-11
Starting Client Transaction Option.	7-12
Validating Results	7-13
Disregarding a Running Query	7-13
Auditing Query Performance	7-13
Analyzing Queries Using Plan View.	7-14
Using Query Plan View	7-14
Analyzing a Sample Query	7-17
Working With Your Query Plan.	7-19
Simply mouse-over the highlighted section of the plan to view the information or warning.	7-20
Creating an Ad Hoc Query	7-20
Sample Ad Hoc Queries	7-21

8. Working with XQuery Source

What is Source View?	8-1
XQuery Support.	8-2
Using Source View.	8-3
Finding Text.	8-3
Function Navigation	8-4
Code Editing Features	8-4

9. Best Practices and Advanced Topics

Using a Layered Data Integration and Transformation Approach.	9-1
Using Inverse Functions to Improve Performance During Updates	9-3

Sample Inversible Data	9-4
Considerations When Running Queries Against Logical Data	9-4
Improving Performance Using Inverse Functions: an Example	9-6
Leveraging Data Service Reusability	9-15
Modeling Relationships	9-16

Introduction to Data Services

Just as the BEA WebLogic Application Server freed application developers from the tedium associated with managing multi-user applications across the Internet, BEA Aqualogic Data Services Platform allows data application developers to concentrate on developing and extending enterprise information without a need to directly program to the underlying physical data sources.

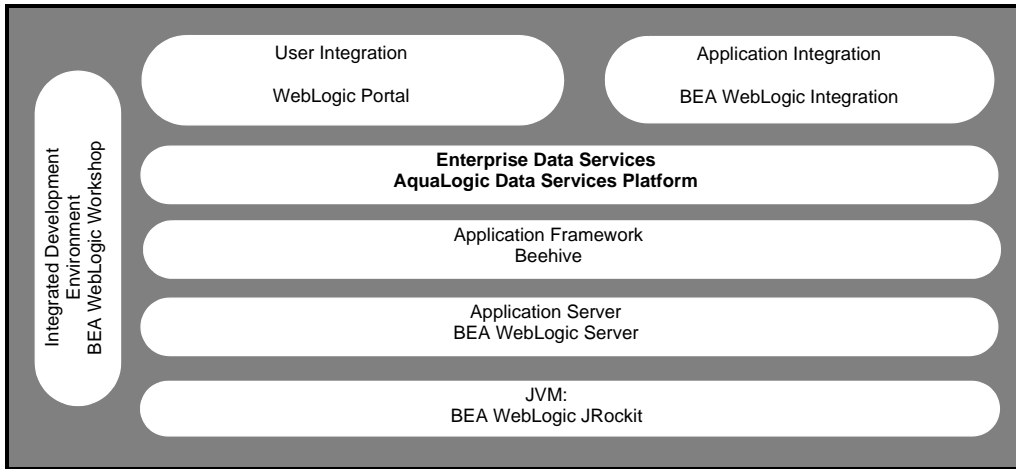
Data Services Platform (DSP) takes advantage of emerging standards to enable you to create hierarchical, enterprise-wide data services which can be accessed by any Web-based application.

Specifically, data services enable you to:

- Insulate integrated applications and processes from complexity of divergent data forms and potentially disconnected sources of enterprise data.
- Manage the metadata information imported from disparate data sources.
- Create data models showing the relationships between various data services.

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

Figure 1-1 BEA Integrated Development Environment



Data Services and the Enterprise

In modern enterprises data is generally readily available. While this has reduced that 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.

DSP 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.

DSP 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.

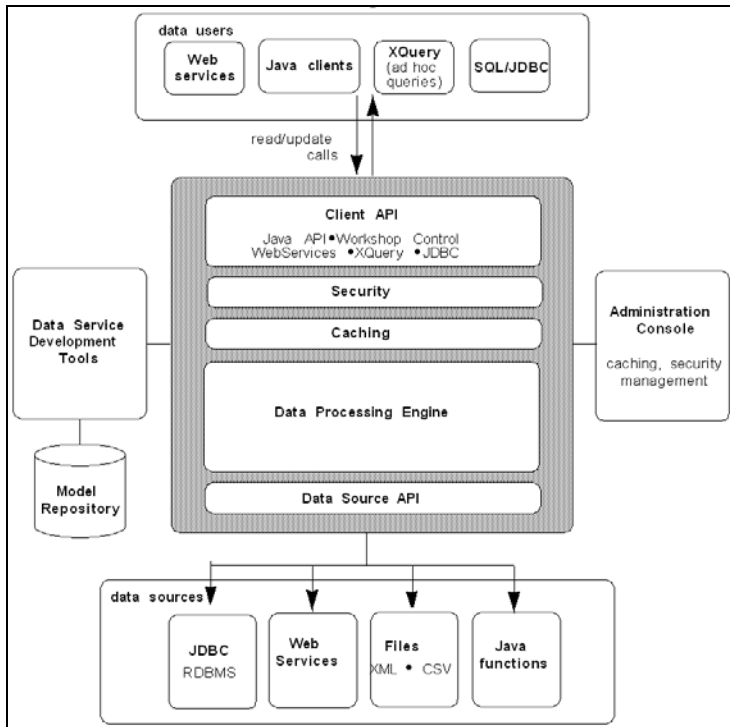
Through standardized Service Data Objects (SDO) technology, web-based applications can automatically read and update relational data. Through simple Java programs DSP update capabilities can be extended to support any logical data source.

- For an overview of the DSP system, see the Data Services Platform *Concepts Guide*.
- For detailed, hands-on tutorial illustrating many DSP features and techniques see the samples tutorial, available from the [Data Service Platform e-docs page](#).

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.

Figure 1-2 Data Services Platform Component Architecture



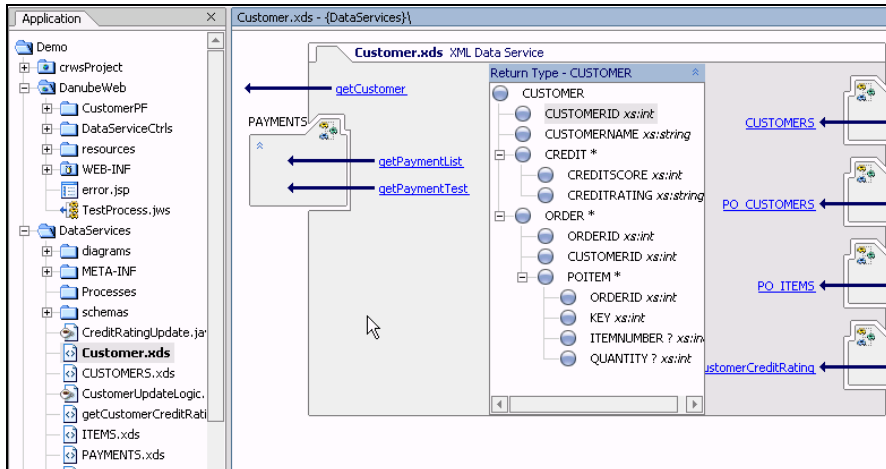
DSP approaches the problem of creating integration architectures by building logical data services around physical data sources and then allowing business logic to be added as part of easily maintained, graphically designed XML query functions (also called XQueries).

Using standard protocols such as JDBC, DSP 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 DSP-based projects. And any application can access DSP queries — including update functions — through a mediator API or a Data Services Platform 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 DSP *Client Application Developer’s Guide*.)

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

Figure 1-3 Sample Data Service



Data Services Platform Applications and Projects

DSP query and model development services are available through a DSP-based WebLogic Workshop project. After you have installed DSP (see the [Installation Guide](#)), you have two options:

- Creating a Data Services Platform-based project within any WebLogic Workshop application:
File → New → Project → DSP Project
- Creating a new Data Services Platform-based application:
File → New → Application → DSP Application

Services Available to a Data Services Platform-Based Project

A DSP-based project is comprised of a number of interrelated data services used in developing models and query functions. Service components are designed to enable rapid development, prototyping, and deployment of these services and functions in your applications.

Table 1-4 Survey of Major Services Provided by Data Services Platform

Service	Feature
Data Services and Data Modeling	<ul style="list-style-type: none"> • Physical models • Logical models • Relationships • Read functions • Procedures • Navigation functions • Roles
Metadata Management	<ul style="list-style-type: none"> • Browse metadata • Search metadata • Impact analysis • Auditing • Reports
Import Metadata	<ul style="list-style-type: none"> • Relational, Web services, XML files, delimited files, Java • Update metadata
Query Management	<ul style="list-style-type: none"> • Graphical query development • Testing • Plan analysis • Performance reporting • Auditing • Source editing • Caching • Security
Application Services	<ul style="list-style-type: none"> • Mediator API • Data Services Platform control • JDBC
Service Data Objects (SDO)	<ul style="list-style-type: none"> • Automatic read-write to relational sources • Custom update
XQuery Engine	<ul style="list-style-type: none"> • Inverse functions

For more information on WebLogic Workshop applications and projects see [“Applications and Projects”](#) in WebLogic Workshop online documentation.

DSP: Roles and Responsibilities

- **Metadata Development.** Using the DSP Metadata Import wizard, any team member can quickly create a set of physical data services from enterprise data sources.
- **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 services*.
- **Query Development.** Once data services are created, any 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.
- **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 a Data Services Platform 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 Metadata Browser for real-time introspection of disparate data source metadata that has been developed through DSP.

DSP: Typical Development Process

The following steps summarize a typical Data Services Platform-based project development cycle.

1. **Create your project.** Create a DSP-based project in a new or existing WebLogic Workshop application as described in [“Creating a DSP-based Application”](#) on page 2-2 and [“Adding a DSP Project to an Existing BEA WebLogic Application”](#) on page 2-4.
2. **Import metadata.** Metadata 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 (spreadsheets), custom Java functions, and XML files. See [Chapter 3, “Obtaining Enterprise Metadata.”](#)
3. **Create a data model.** You can graphically build a data model that shows the relationships and cardinality between the data services you have selected (see [Chapter 5, “Modeling Data Services”](#) for details). In 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 (Chapter 4, “Designing Data Services”). The built-in XQuery Editor (Chapter 6, “Working with the XQuery Editor”) includes standard XQuery functions and language construct prototypes. Using the editor you can map source elements or transformations to a return type.

The Data Service Palette provides access to all data services available to your application. Queries and data service logic are maintained in a single, editable source file that is fully integrated with your data service (Chapter 8, “Working with XQuery Source”).

5. **Test your function.** The data service functions you create can be tested at any time. 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 (Chapter 7, “Testing Query Functions and Viewing Query Plans”). If you have appropriate permissions, you can also update source data through Test View.
6. **Review the 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 (“Analyzing Queries Using Plan View” on page 7-14).

Examples, Samples, and Tutorials

Samples and examples used in this book are based on the Sample Retail Application (RTLApp) that is included with DSP. See also the “Sample Retail Application Overview” in the DSP *Installation Guide*.

A number of examples of DSP technology can be found in the DSP Samples Tutorial. This tutorial is also based on RTLApp.

To access the tutorial see the DSP e-docs page:

<http://edocs.bea.com/aldsp/docs21/index.html>

Data Services Platform Projects and Components

BEA Aqualogic Data Services Platform (DSP) can be added to WebLogic Workshop in two ways:

- As a project in a new application
- As a project in any existing BEA WebLogic application.

The basic menus, common behavior, and look-and-feel associated with WebLogic Workshop apply to DSP.

Note: WebLogic Workshop online documentation is available at:

<http://e-docs.bea.com/workshop/docs81/index.html>

This chapter discusses various WebLogic Workshop facilities that you can use in creating and managing your DSP-based projects. DSP extensions to WebLogic Workshop are also described from an interface perspective.

The following topics are covered:

- [DSP-Based BEA WebLogic Projects](#)
- [Major Components of a DSP Project](#)
- [Building and Deploying Applications, EARs, and SDO Mediator Clients](#)
- [Refactoring DSP Artifacts](#)

DSP-Based BEA WebLogic Projects

As noted above, you can create a WebLogic Workshop application that automatically includes a Data Services Platform project. Or you can add DSP projects to any BEA WebLogic application.

Note: It often makes sense to consolidate DSP queries into a WebLogic Workshop application dedicated to DSP development. Other applications can then access these queries through the DSP Mediator API or a Data Services Platform control. For complete details related to how client applications can access Data Services Platform functions and procedures see the DSP *Client Application Developer's Guide*.

Verifying Your DSP Version Number

To ascertain that DSP is available to your application or to determine the version of DSP that you are running, start your BEA WebLogic Server and access its Administration Console.

As an example, the WebLogic Server Console for the sample domain provided with BEA WebLogic can be accessed from:

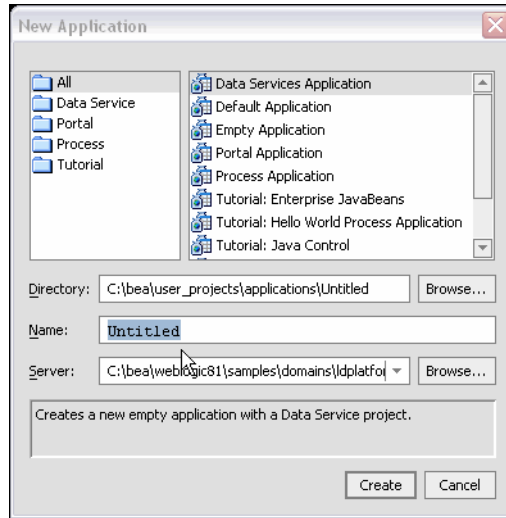
<http://localhost:7001/console>

Navigate to the Console → Versions page (Console being the top menu item) and find the version number and creation date for DSP.

Creating a DSP-based Application

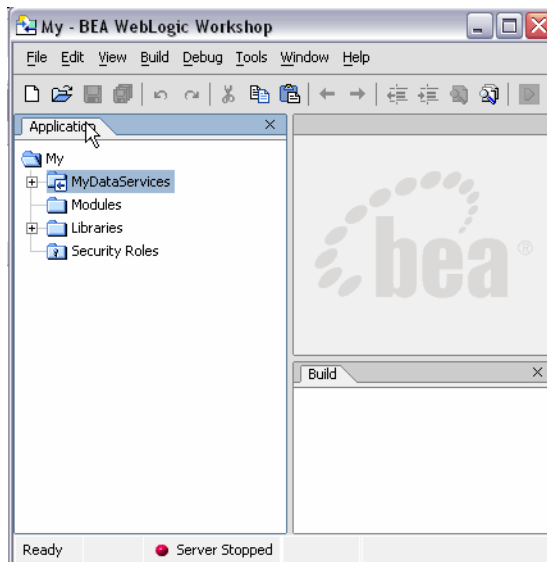
To create a DSP-based application select File → New → Application from WebLogic Workshop menu. When the dialog appears, select DSP Application ([Figure 2-1](#)).

Figure 2-1 Creating a New Data Services Platform Application



You probably will want to change the name of the application from `Untitled` to something else. Your new application automatically contains an initial DSP-based project.

Figure 2-2 Application View of a New Data Services Platform Application



You can save your application at any time using the File → Save, Save As, or Save All commands. Save All saves any modified files in your application.

When you initially create a WebLogic Workshop application such as “myLD”, a file called `myLD.work` is created in the root directory of your application. Invoking Workshop using this file also opens your application.

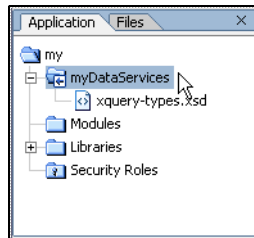
An application can contain any number of DSP or other types of WebLogic Workshop projects.

Adding a DSP Project to an Existing BEA WebLogic Application

You can also add one or several DSP projects to any WebLogic Workshop application.

To do this select File → New → Project. When the project creation dialog appears, choose DSP Project.

Figure 2-3 Application Tab of a New Data Services Platform Application



Major Components of a DSP Project

When a new Data Services Platform application or project is created, a DSP project folder is also created. This becomes the root directory of your project (see [Figure 2-3](#)). Two Java archive (.jar) files are added to the application’s Libraries folder including `ld-server-app.jar` and the `mediator.jar`. The latter file manages creation of Service Data Objects (SDOs), described in detail in the *Client Application Developer’s Guide*.

[Table 2-4](#) lists major DSP file types and their purposes.

Table 2-4 Data Services Platform Components, Including File Types

Component	Purpose
Data Services (.ds files)	<p>Data services are contained in DS files and can be located anywhere in your application. Each data service file is an XQuery document.</p> <p>Note: Since a DS file may contain numerous XQueries as well as other automatically-generated pragma directives, care should be taken when editing this file directly.</p>
Model Diagrams (.mcd files)	<p>Model diagrams provide a graphical representation of the relationships between various data services, which themselves represent the physical and logical data services available to your DSP queries.</p> <p>Model diagrams have the extension .mcd and can be located anywhere in your DSP project.</p>
Metadata information	<p>Metadata information is contained in META-INF folders associated with JAR files. The non-editable contents of this Libraries folder contains information on data sources used by data services.</p>
Schemas (.xsd files)	<p>Data services typically are associated with <i>XML types</i> whose physical representation is an XML schema file. Schema files can be located anywhere in your application. Schemas automatically created by the Metadata Import wizard are placed in a <code>schemas</code> project inside your application.</p> <p>Schema files can be manually created or modified using any text editor. There is also a built-in schema editor in DSP Design View and in model diagrams containing the data service.</p> <p>The XML type associated with a data service is also the return type of each function in your data service. The <i>return type</i> precisely describes the shape of the document to be returned by your query.</p> <p>The return type can be modified through the XQuery Editor or directly in source. However, this generally should only be done in conjunction with the Save and Associate Schema command (see “Creating a New Data Service and Data Service Function” on page 6-7 for details).</p>
XQuery function libraries (.xfl files)	<p>XQuery function libraries typically contain utility XQuery functions that can be used by application data services and in building data transformations. A typical example would be a routine for converting currencies based on daily exchange rate. Such transformational functions could be used by any data service in your application.</p>

Other files which may appear in DSP projects include Java files containing custom update logic and SDO configuration files such as `sdo.xsdconfig`, which allows XMLBean technology to create SDOs rather than XMLBeans.

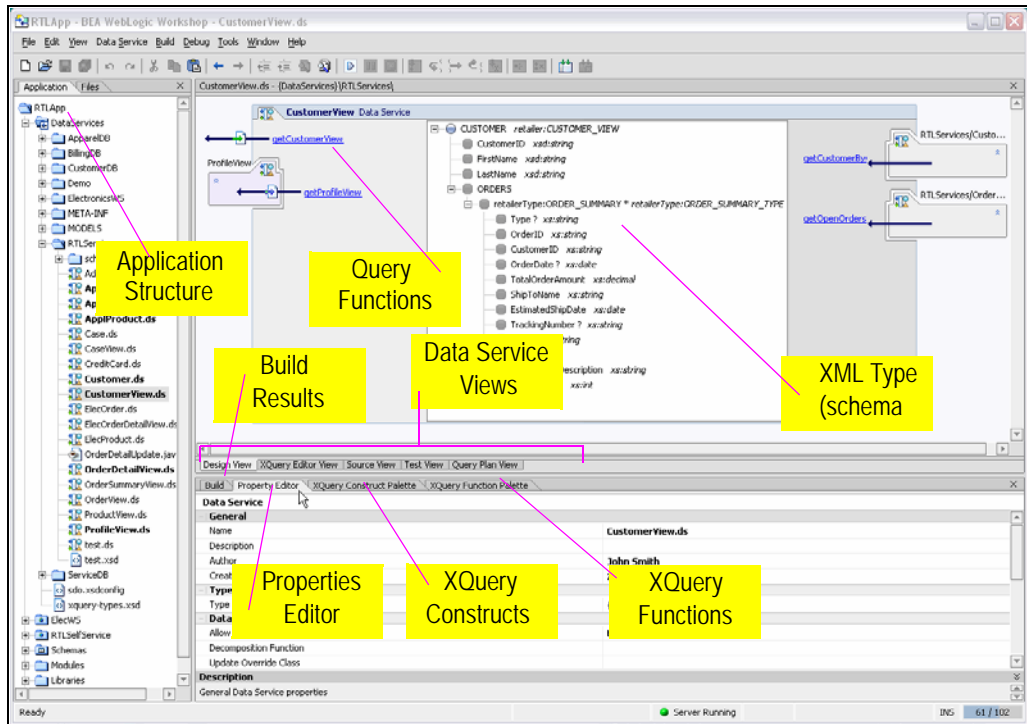
Using the WebLogic Workshop IDE

WebLogic Workshop is fully described in on-line and printed documentation. A good place to start is:

<http://e-docs.bea.com/workshop/docs81/index.html>

Alternatively, WebLogic Workshop provides complete on-line help.

Figure 2-5 Some WebLogic Workshop Components in a DSP-Based Project



The following table briefly describes:

- WebLogic Workshop functionality extensively used by DSP.
- DSP extensions to the Workshop user interface.

Table 2-6 Summary of WebLogic Workshop Pane Used by DSP

Service	Purpose
Application pane	Lists the projects and other components in your application.
Files pane	Provides an ordered listing of files used in your application.
Build pane	Provides feedback while the application is being built and reports build success or failure.
Output pane	Shows data sources accessed, execution times, and query statement.
Property Editor	Provides information on properties associated with the currently selected object. Some properties are configurable or editable.

[Table 2-7](#) describes the several WebLogic Workshop menu commands you will use with DSP projects.

Table 2-7 Summary of WebLogic Workshop Menu Services Used by DSP

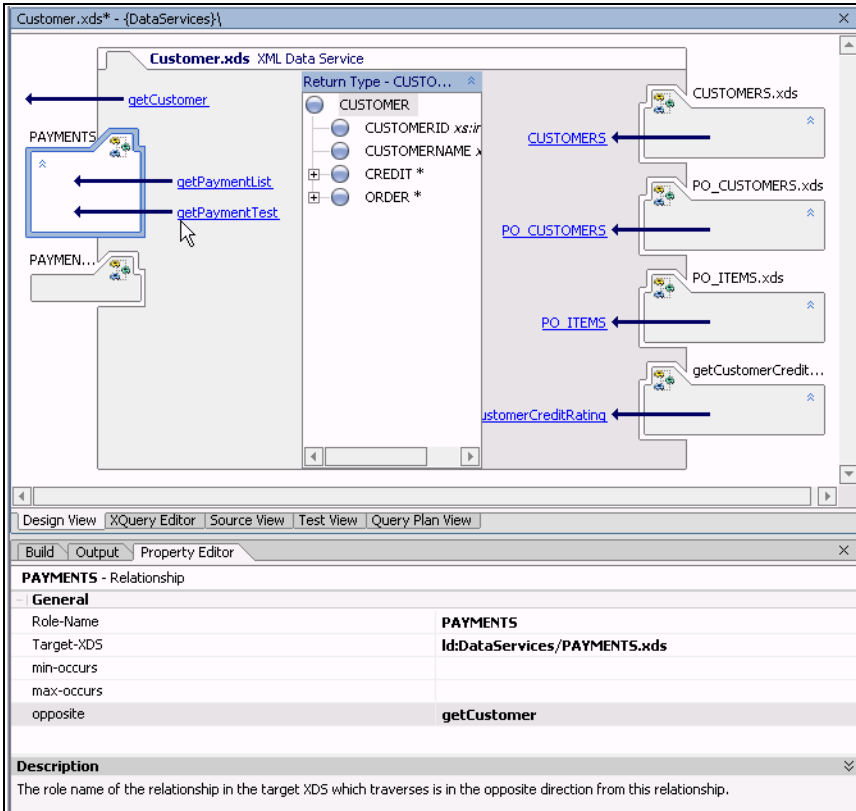
Service	Purpose
File menu	<p>When working with DSP projects you will often use the following File menu options:</p> <ul style="list-style-type: none"> • Save, Save As, Save All. The Save command saves the current file while the Save All command saves all open or modified files in your project. Use the Save All command to make sure that all changes you have made to your application will be persisted. • Import commands. Use the Import file browser to add files or libraries to your application. For example, if you have an externally developed schema you can use the Import command and associated file browser to bring a copy of it into your application.

Property Editor

You can use the Property Editor to view details related to any WebLogic Workshop artifact (see [Figure 2-8](#)). For example, in Design View (see [“Design View” on page 2-12](#)) if you click on the general data service, the Property Editor provides details on that service. If you click on a relationship

representation in your data service, property details on that relationship appear. In many cases, property settings are editable or configurable.

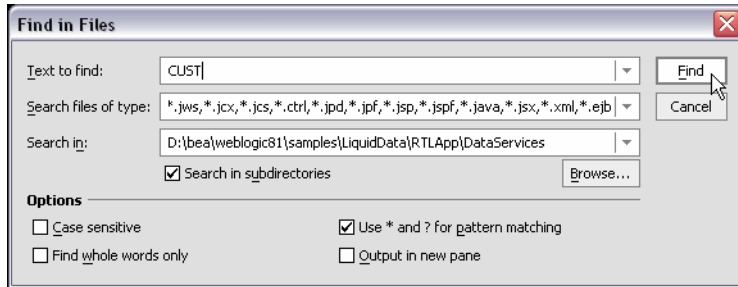
Figure 2-8 Relationship Properties in a Data Service



Finding Text in Files

WebLogic Workshop provides a comprehensive file search facility with its Find in Files option, available from the Edit menu (Edit → Find in Files).

Figure 2-9 Workshop File Search Facility



You can use Find in Files to search for references to any DSP artifacts such as particular data sources, use of functions, and so forth.

Survey of DSP Additions to WebLogic Workshop

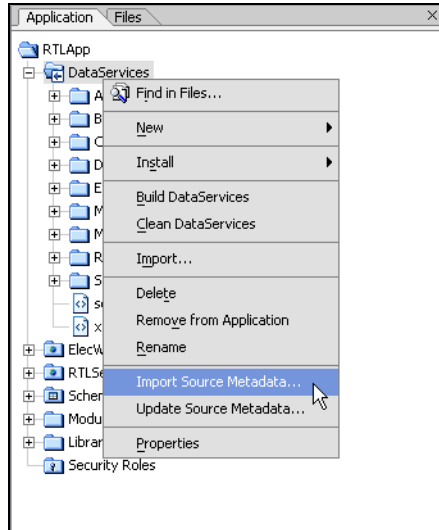
A DSP project adds menu items and views to the basic WebLogic Workshop environment to support the following functionality:

- [Metadata Import](#)
- [Data Models](#)
- [Data Services](#)
- [XQuery Function Libraries](#)
- [Usages of Data Services Artifacts](#)
- [Updating Application or Project Data Service Libraries](#)

Metadata Import

Data services are central to creating data models and physical and logical data views that can be used in DSP queries. The first step in creating a data service is to import metadata from physical data sources so that corresponding *physical data services* can be created.

Figure 2-10 Selecting Metadata Import for a DSP Project

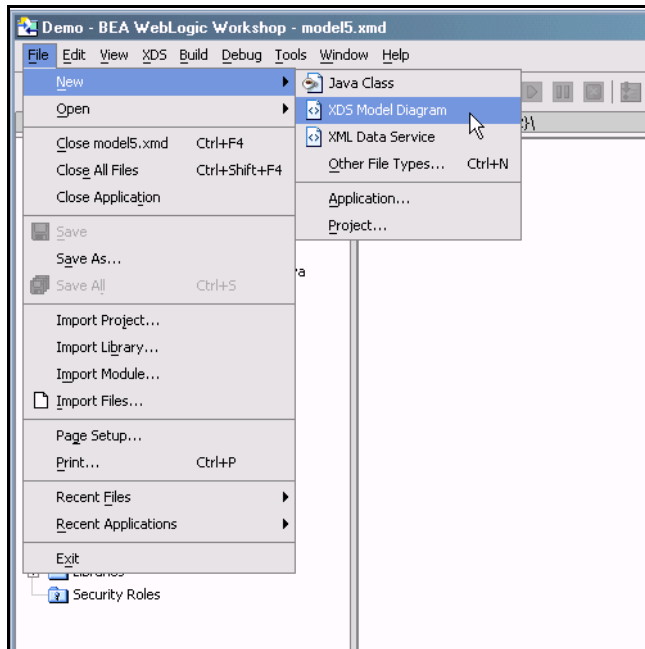


For details related to importing and updating metadata into your DSP project see [Chapter 3, “Obtaining Enterprise Metadata.”](#)

Data Models

Through the data model interface that you can:

- Establish or modify relationships between data services.
- Edit a data service’s return type.
- Create annotations to a model or a data service.

Figure 2-11 Creating a Data Model Diagram from the File Menu

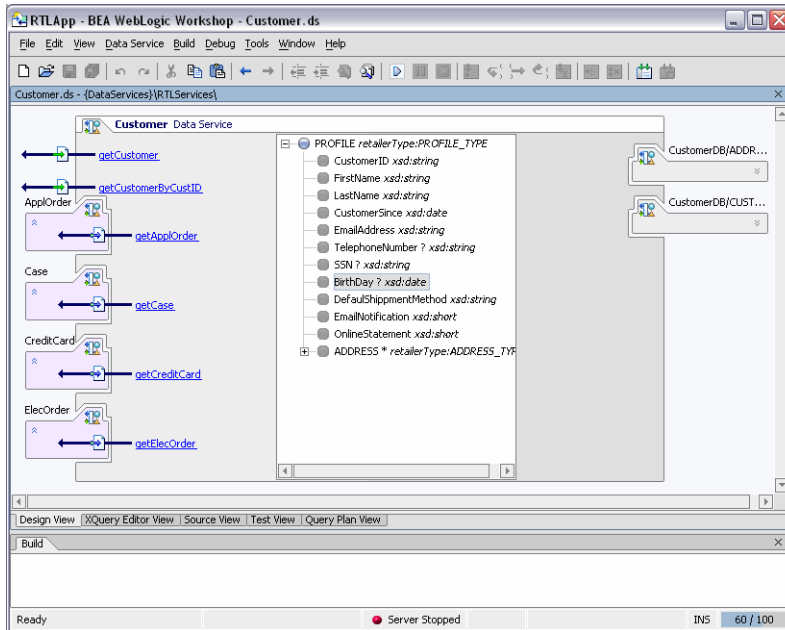
For details on developing and maintaining data models see [Chapter 5, “Modeling Data Services.”](#)

Data Services

Every data service provides a Design View, XQuery Editor View, Source View, Test View, and Query Plan View. Each data service is based around a single XQuery source file. And every data service has an associated XML type (XDS file).

Data services are composed of read and navigation functions and procedures. Read functions must return the XML type of the data service. Navigation functions, return the XML type of their native data service. Procedures, also known as side-effecting functions, need not return anything.

Figure 2-12 Sample Data Service



Design View

Design View is the central reference point of every data service. Through Design View that you can:

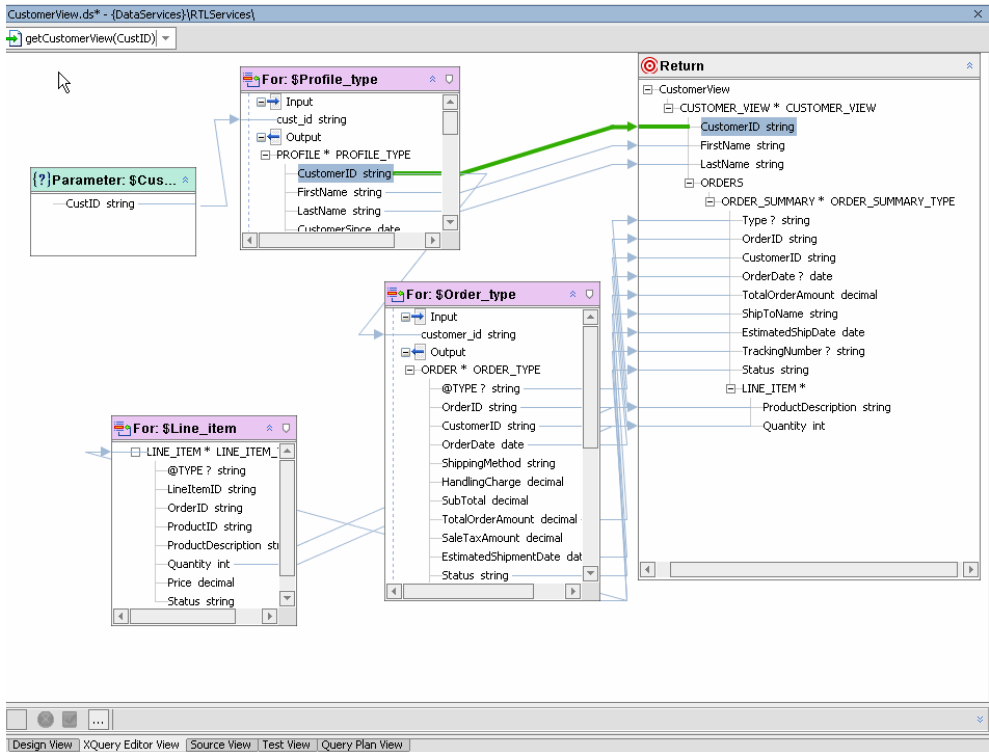
- Add or modify the XML type (associated schema).
- Add read functions using the XQuery Editor View.
- Add private functions.
- Add relationships in the form of navigation functions. These functions are typically developed using the Relationship wizard.

For details on developing and maintaining data services see [Chapter 4, “Designing Data Services.”](#)

XQuery Editor View

Through the XQuery Editor View you can develop query functions by projecting data service function elements, as well as transformations, to the function’s return type.

Figure 2-13 Sample XQuery Editor Query with Its Return Type

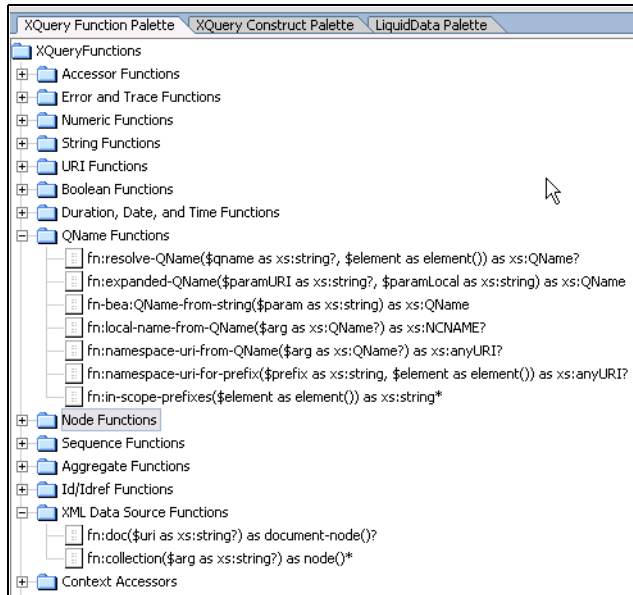


The graphical editor directly supports common constructs of the 1.0 XQuery standard. Several resources are available to help in the development and maintenance of business logic. These are all available from the WebLogic Workshop View or View → Windows menu).

For details on developing queries using XQuery Editor View see [Chapter 6, “Working with the XQuery Editor.”](#)

XQuery Function Palette

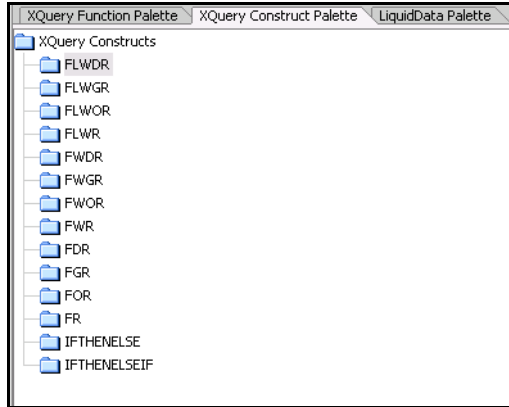
An XQuery function palette ([Figure 2-14](#)) is available that supports standard XQuery and BEA-specific functions. This function palette is also available from the Workshop View → Windows menu.

Figure 2-14 XQuery Function Palette

Like all Workshop panes, the XQuery Function Palette can be placed anywhere in the WebLogic Workshop window. Functions from this palette can be dragged into XQuery Editor View, as well as Source View.

XQuery Constructs Palette

DSP projects also have access to the XQuery Constructs palette (Figure 2-15). This palette supports creation of different types of XQuery statements in the XQuery Editor View or Source View. Many of the construct prototypes such as FLWGR, FGWOR, FWGR, and so forth are variations on the most common XQuery construct, FLWR (for-let-where-return).

Figure 2-15 XQuery Constructs Palette

For example, FLWGR adds the DSP extension Group By. The prototype is shown below in Source View.

```

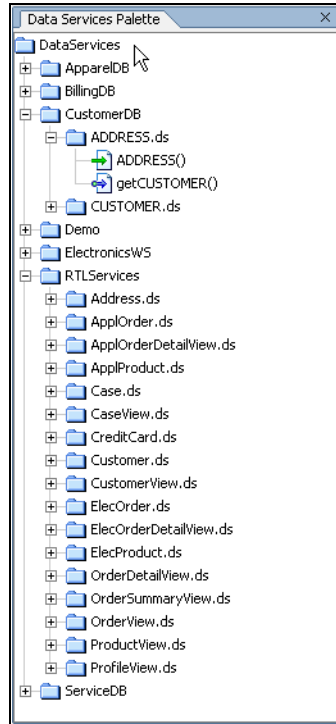
for $var in ()
let $var2:=()
where (true)
group by () as $var3 with partitions $var as $var4
return
  ()
  
```

For details on Group By and other BEA XQuery extensions see the [XQuery Developer's Guide](#).

Data Services Palette

The Data Services Palette ([Figure 2-16](#)) is only available to DSP projects. It provides the DSP XQuery Editor with access to data service and XFL (XQuery function library) routines.

Figure 2-16 Data Services Palette

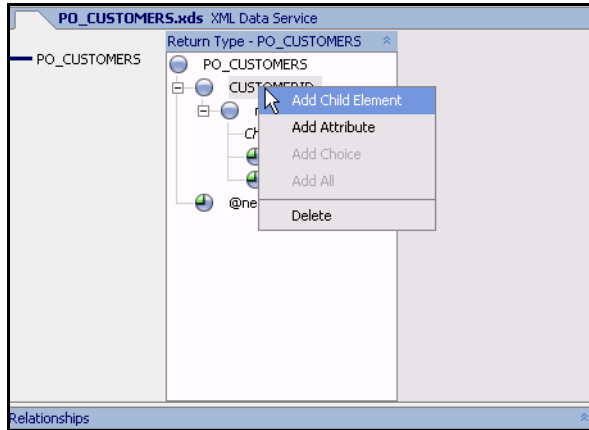


For details on using the XQuery Editor see [Chapter 6, “Working with the XQuery Editor.”](#)

Editing XML Types and Return Types

A schema editor for modifying XML types in model diagrams and data services, as well as return types in the XQuery Editor, is available. See [“Working with Logical Data Service XML Types”](#) on page 4-23. Most editor options are available from the right-click menu.

Right-click menu commands for return types differ slightly from those in the XML type editor. The reason is that you can use the XQuery Editor to create if-then-else constructs, zones, and cloned elements as a means of more exactly specifying the form your query result document should take. (See [“Modifying a Return Type”](#) on page 6-47.)

Figure 2-17 Editing an XML Type Element

Test View

After you have developed a query you can run it using Test View. For details see [Chapter 7, “Testing Query Functions and Viewing Query Plans.”](#)

Source View

If you are working in Source View you can easily add pre-built XQuery functions and constructs to your source, as well as make other editing changes to your data service. For additional details see [Chapter 8, “Working with XQuery Source.”](#)

Query Plan View

You can review the query plan developed by DSP for a particular function in order to verify the generated SQL or look for opportunities to improve performance. See [“Analyzing Queries Using Plan View” on page 7-14.](#)

XQuery Function Libraries

You can create XQuery libraries containing functions that can be used by any data service in your application. XQuery function libraries can be created in two ways:

- Using the File → New XQuery Function Library option.
- Automatically, when Java functions returning primitive types are imported as metadata (see [“Obtaining Enterprise Metadata” on page 3-1.](#))

An XQuery function library (.xfl file) is ideal for creating transformation, security, and other types of functions that are not associated with an XML type.

Also see in the Data Services Platform [Samples Tutorial Part II](#):
- Lesson 35: Creating an XQuery Function Library

XQuery Function Library Views

An XQuery function library (XFL) holds any number of functions.

XFL Design View is similar to the data service Design View (shown in “[Sample Data Service](#)” on [page 1-5](#)). The primary differences are:

- Since no schema is associated with a library, there is no XML type.
- There are no relationship functions.

The tabular modes available in data services — Source View, XQuery Editor View, Test View, and Query Plan View — are available to XQuery function libraries as well.

XFL files play an important role in creating inverse functions. See “[Using Inverse Functions to Improve Performance During Updates](#)” on [page 9-3](#) in [Chapter 9](#), “[Best Practices and Advanced Topics](#).”

Creating an XFL Function

It is not difficult to make a function in a data service available throughout your project as an XML function library.

Note: Namespace conflicts must to be resolved before you can make your function generally available.

The following function is available in the RTLApp’s DataServices/RTLServices/Credit Card data service (namespace declarations from a separate section of the source file are also included):

```
declare namespace ns1="ld:DataServices/BillingDB/CREDIT_CARD";

import schema namespace ns0="urn:retailerType" at
"ld:DataServices/RTLServices/schemas/CreditCard.xsd";

declare namespace tns="ld:DataServices/RTLServices/CreditCard";

(: ... :)
```

```

declare function tns:getCreditCard() as element(ns0:CREDIT_CARD) * {
for $CREDIT_CARD in ns1:CREDIT_CARD()
return <ns0:CREDIT_CARD>
  <CreditCardID>{fn:data($CREDIT_CARD/CC_ID)}</CreditCardID>
  <CustomerID>{fn:data($CREDIT_CARD/CUSTOMER_ID)}</CustomerID>

<CustomerName>{fn:data($CREDIT_CARD/CC_CUSTOMER_NAME)}</CustomerName>
  <CreditCardType>{fn:data($CREDIT_CARD/CC_TYPE)}</CreditCardType>
  <CreditCardBrand>{fn:data($CREDIT_CARD/CC_BRAND)}</CreditCardBrand>

<CreditCardNumber>{fn:data($CREDIT_CARD/CC_NUMBER)}</CreditCardNumber>
  <LastDigits>{fn:data($CREDIT_CARD/LAST_DIGITS)}</LastDigits>
  <ExpirationDate>{fn:data($CREDIT_CARD/EXP_DATE)}</ExpirationDate>
  {fn-bea:rename($CREDIT_CARD/STATUS,<Status/>)}
  {fn-bea:rename($CREDIT_CARD/ALIAS,<Alias/>)}
  <AddressID>{fn:data($CREDIT_CARD/ADDR_ID)}</AddressID>
</ns0:CREDIT_CARD>

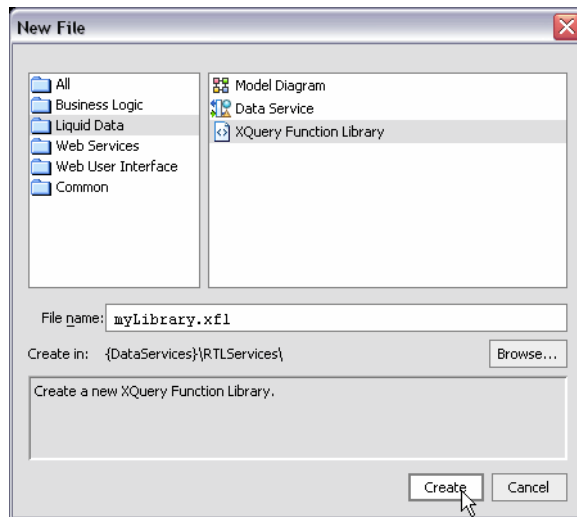
```

Here are the steps you would take to create this function in an XQuery library:

1. The first step is to create and name a library, if you do not already have one:

File → New → XQuery Function Library

Figure 2-18 Creating an XQuery Function Library



2. Name your library, such as myXQueryLibrary.
3. Copy your function into the newly created file.

4. Change the function declaration to match the namespace of your library file.

Source for the XQuery library file containing the CREDIT_CARD function appears below. To simplify, the object is returned as \$x rather than as a set of individually-mapped elements.

```
(::pragma xfl <x:xfl xmlns:x="urn:annotations.ld.bea.com"></x:xfl> ::)

xquery version "1.0" encoding "WINDOWS-1252";

declare namespace tns="lib:DataServices/MyXQueryLibrary";

declare namespace ns1="ld:DataServices/BillingDB/CREDIT_CARD";
import schema namespace ns0="urn:retailerType" at
"ld:DataServices/RTLServices/schemas/CreditCard.xsd";

(: function pragma removed for readability :)

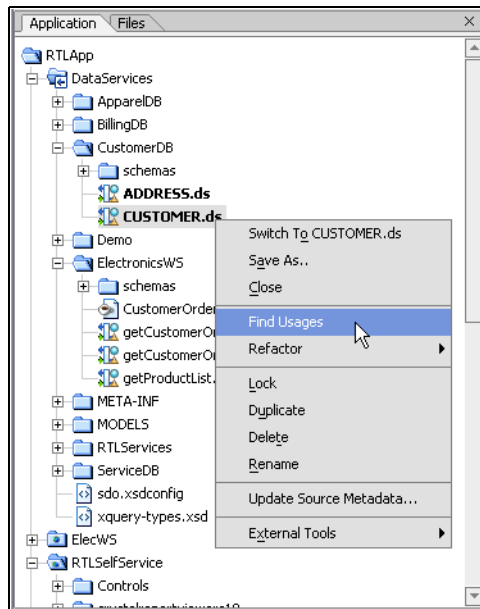
declare function tns:getCreditCard() as element(ns1:CREDIT_CARD) * {
for $x in ns1:CREDIT_CARD()
return $x
};
```

Usages of Data Services Artifacts

It is often convenient to determine which Data Services Platform artifacts are in use by which other artifacts. For example, before making changes in an XML type it is important to determine what other data services might be impacted. Of course you can do this through the Metadata Browser, described in the “Viewing Metadata” chapter of the DSP [Administration Guide](#). However, it is often more convenient to do this in the context of the WebLogic Workshop navigation pane or the DSP Design View.

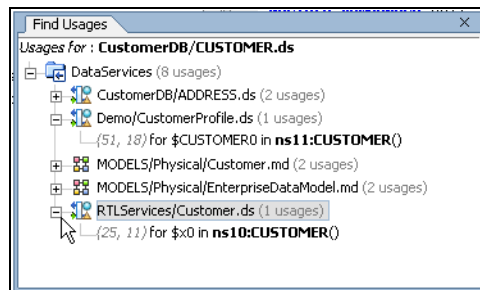
For example, in the RTLApp, right-clicking on a data service shows a number of options including Find Usages ([Figure 2-19](#)).

Figure 2-19 Finding Usages of a Data Service in RTLApp



When you pick this option, usages of the artifact are displayed, as shown in Figure 2-20.

Figure 2-20 Usages of the Customer Data Service in the RTLAPP



You can find the usages of the following types of DSP artifacts:

- Data services
- XML library function files
- Read and relationship functions
- Procedures

- Private functions
- Schemas

Updating Application or Project Data Service Libraries

When you save a DSP application its JAR libraries files are bound to that application. If you subsequently migrate to a newer version of DSP, you also need to migrate your application to the latest library files. For details see “Migrating Legacy DSP Applications” in the DSP *Installation Guide*.

Building and Deploying Applications, EARs, and SDO Mediator Clients

DSP attempts to rebuild your application as necessary. However, there are times when you will need to initiate a build directly.

Building, Deploying, and Updating Applications

The following table describes relevant Build menu options and their uses.

Build Menu Options	Usage
Build Application	<p>Builds or rebuilds your application. The result is that the contents of all the project-specific JAR files are updated according to the underlying project script. If your application has already been deployed, this option will automatically redeploy after a successful build.</p> <p>You can also build individual projects.</p>
Clean Application	<p>Attempts to undeploy EJBs and other resources that were produced by the compilation process. In some cases this is not possible because of the state of the server. If Clean Application does not solve the problem, stop and restart WebLogic Server.</p> <p>Clean Application addresses problems that occur due to cyclic compilation of Java files during iterative development, not on production servers.</p> <p>You can also clean individual projects.</p>
Build EAR	<p>Creates a Java archive (JAR) file of your application. The EAR file has the same name as your application.</p>

When to Rebuild Your DSP Project

You need to rebuild your project whenever you delete a file from a DSP-based project. Rebuilds can occur on a project or at the application level. Generally speaking, there is no need to rebuild your entire application unless you have made changes to multiple projects.

Rebuild your project (or application) in two steps:

1. Clean your project (application). You can do this by right-clicking on your project (application) in the Application pane and selecting the available Clean option. Alternatively, use the appropriate Clean option available from the WebLogic Workshop Build menu.
2. Build your project (or application) using the appropriate right-click or Build menu options.

Note: If you try to run a function in Test View and it fails unexpectedly, it is often curative to clean, then rebuild your application before attempting to run your query again.

Deploying Your Application

If your application is already deployed, it will be automatically redeployed whenever you rebuild it. Under some conditions you may want to undeploy your application first. The following table describes relevant options available when you click on your application folder in the Application pane.

Table 2-21 Usages of Various Deployment Menu Options

Application Level Right-click Menu Deployment Options	Usage
Deployment →Redeploy	Redeploys your application. Note: When you build your application it is automatically redeployed.
Deployment →Full Redeploy	First removes your application from the server, then redeploys it.
Deployment →Undeploy	Removes your application from the server.

For additional information on deploying WebLogic Workshop applications see:

- [“Building and Deploying Integrated Applications”](#)
- [“Deploying Applications to a Production Server”](#)

Creating the SDO Mediator API

After you have created and tested your application's query functions, you need to make them available to client applications. The SDO mediator API is the primary means of providing access to your updatable functions.

Note: For details on SDO programming and accessing data in Java clients through the mediator API see the Data Services Platform *Client Application Developer's Guide*.

Generating the SDO Mediator JAR in Workshop

One way to create the SDO mediator client Java archive (.jar) file is through the right-click menu option Build SDO Mediator Client. This is only available from the root folder of your application.

When successful, your SDO mediator client will be created in the root directory of your application. The file will be named as:

```
<name_of_your_application>-ld-client.jar
```

The SDO mediator JAR file will also be automatically added to your application's Libraries folder.

Note: Insure that all of your projects are up-to-date and built before creating your SDO mediator JAR file. See also [“Building, Deploying, and Updating Applications” on page 2-22](#).

Command-line Generation of the SDO Mediator API

You can also create the SDO mediator client JAR file through the command line using ant scripts.

When an EAR File Is Available

If you already have an EAR file you can use the script:

```
ant -f $WL_HOME/liquiddata/bin/ld_clientapi.xml  
-Darchive=</your_path/name_of_your_application>.ear>
```

in which case the name of your JAR file will be taken from the EAR file:

```
<name_of_your_application>-ld-client.jar
```

It will be created in the same directory as the EAR file.

Generating an SDO Mediator JAR File

You can generate an SDO mediator client JAR file (without needing an EAR file) by simply specifying an application directory:

```
ant -f $WL_HOME/liquiddata/bin/ld_clientapi.xml  
-Dapproot=</your_path/name_of_your_application>/root>
```

This approach will use the directory name of the application root to compute the JAR file name; in the above case the name would be `root-ld-client.jar`. If that's not what is wanted, you could specify:

```
-Dsdjarname=<MyApp-ld-client.jar>
```

to override this. Either way the JAR file will be generated in the application root directory.

Generating JAR Files in Non-default Directories

For either case you could specify the additional ant parameter:

```
-Doutdir=</path/to/dir>
```

to generate the JAR file to a specific directory location.

Similarly you could use:

```
-Dtmpdir=</path/to/tmp>
```

to specify an alternate directory for temporary files, including the generated `.java` code.

The default tmp file location is specified by the Java system property:

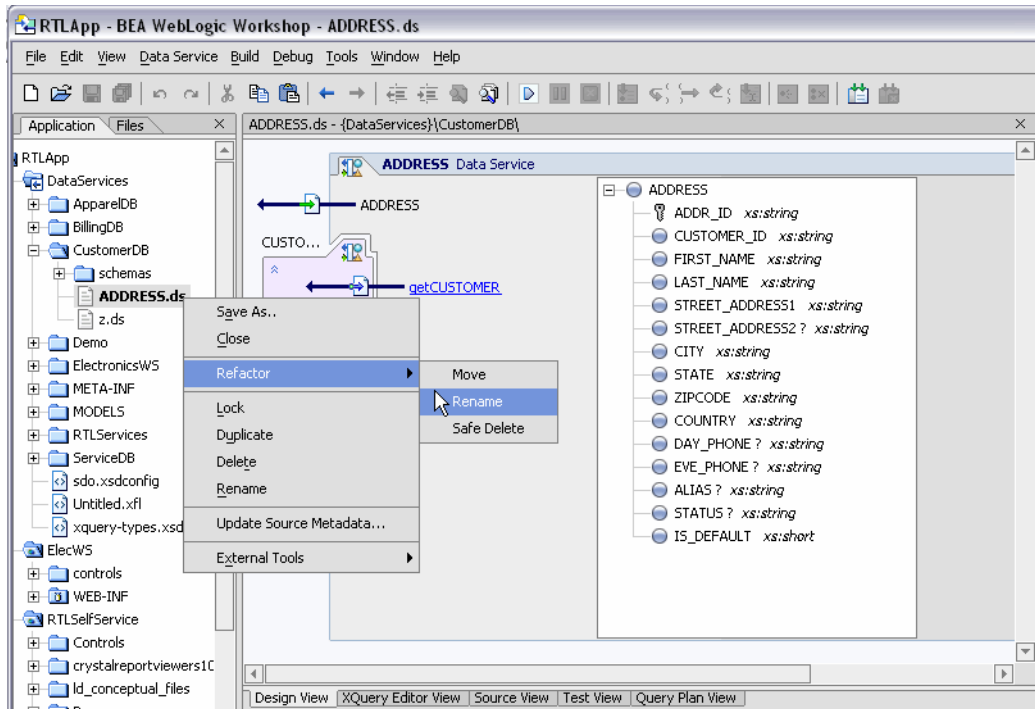
```
java.io.tmpdir
```

In any case, when building from the command line, the SDO `mediator.jar` file will not be added to your application's Libraries folder (shown in [Figure 2-2](#)).

Refactoring DSP Artifacts

There are times when you will want to move, rename, or delete artifacts in your Data Services Platform projects. A typical example: your application is first developed with test data, so as to not expose confidential information to unauthorized individuals. Then, once developed, your application is ready for deployment with the actual, secured data sources. You can use *refactoring* to greatly simplify the renaming, deleting, or relocating of DSP components.

Figure 2-22 Refactoring Options Available for the RTLApp's Address Data Service

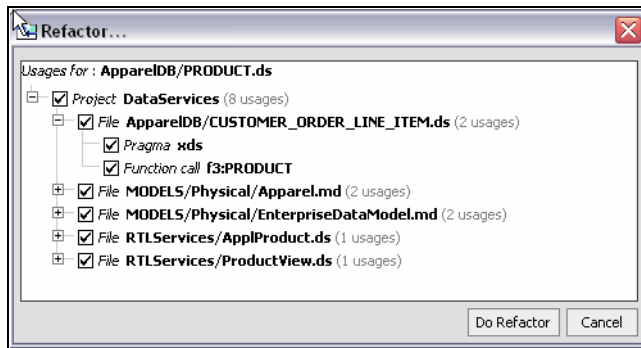


Without refactoring, changes you make to artifact names can easily result in invalid references. For example, renaming a data service file automatically invalids any relationship functions in other data services that refer to that file. The alternative to refactoring is to manually find all usages of a given artifact and make manual edits to data service source; this can be quite tedious and error-prone, particularly as projects grow.

When you use the Refactor option you initially see the effect your refactoring change will have on impacted application artifacts (Figure 2-23). A checkbox allows you to exempt any artifact from the refactoring operation.

Note: Care should be taken when deselecting elements recommended for refactoring. Without additional manual changes to the underlying source you likely will no longer be able to build or deploy your application.

Figure 2-23 Artifacts Impacted by a Refactoring



Artifacts Supporting Refactoring

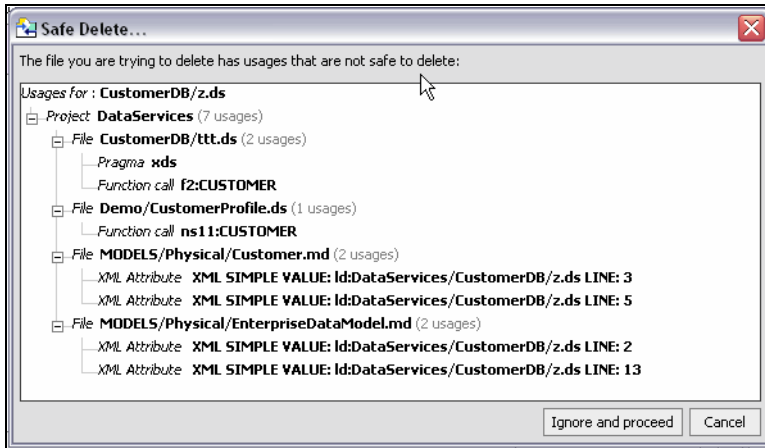
Table 2-24 describes artifacts subject to refactoring and their options.

Table 2-24 Data Service Artifacts Supporting Refactoring and Available Refactoring Options

Artifact	Refactoring Options
Data service (DS files)	Move, refactor rename, safe delete
XML File Library (XFL files)	Move, refactor rename, safe delete
Schemas (XSD files) referred to within a data service	Move, refactor rename, safe delete
Functions (data service and XFL)	Rename, safe delete, add/remove parameters
Namespace declarations	Rename selected prefix or propagate the change through the project.
Schema import (data service and XFL)	Rename selected schema import prefix or propagate the renaming through the project.

Move, rename, and add/remove parameter operations are typically accomplished without adverse consequence. Delete operations, however, can adversely affect your project. For this reason the usages of the artifact you have identified for deletion are shown (see Figure 2-25). From this information you can easily determine the trade-offs between the automation of the refactoring operation and its consequences, which may require additional manual actions on your part.

Figure 2-25 Implications of a Safe Delete Operation

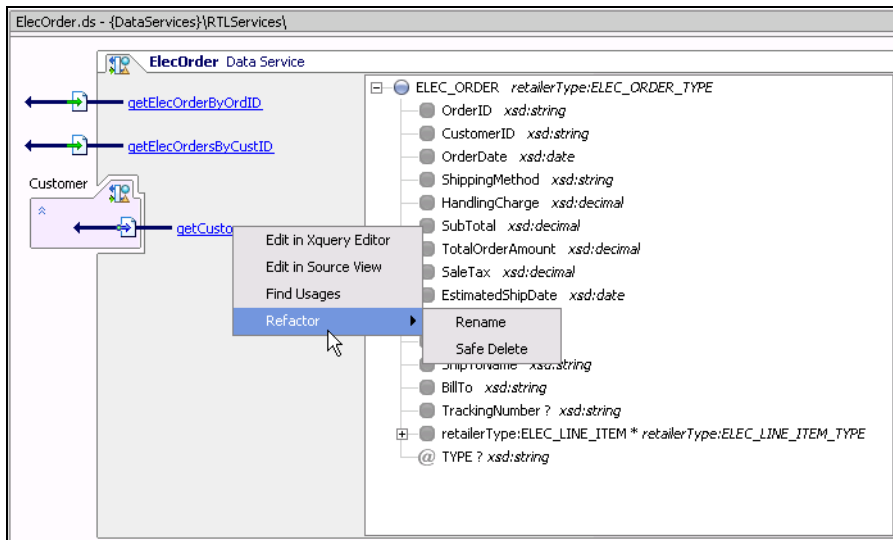


Setting Refactor Options

Access to refactor options depends on the artifact:

- **Data services, XFL files, schemas.** Refactor operations for data services, XFL files, and schemas can be accessed by right-clicking on the artifact in the Application pane.
- **Functions and procedures.** Refactor options for functions and procedures can be accessed by right-clicking on the name of the function or procedure in Design View or by right-clicking on its associated arrow.

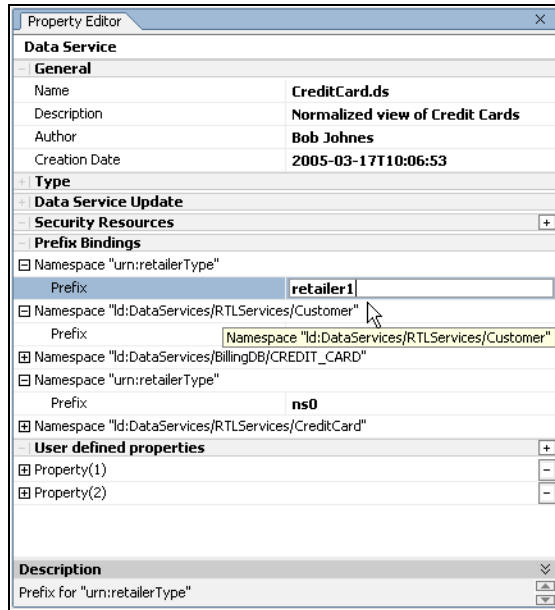
Figure 2-26 Refactoring a Data Service Function



- **Namespace declarations and schema import declarations.** Refactor operations related to namespaces and schema import declarations are accessed through the Prefix Bindings section of Property Editor.

You can refactor a namespace or external schema prefix simply by changing its prefix name (binding) in the Property Editor (Figure 2-26).

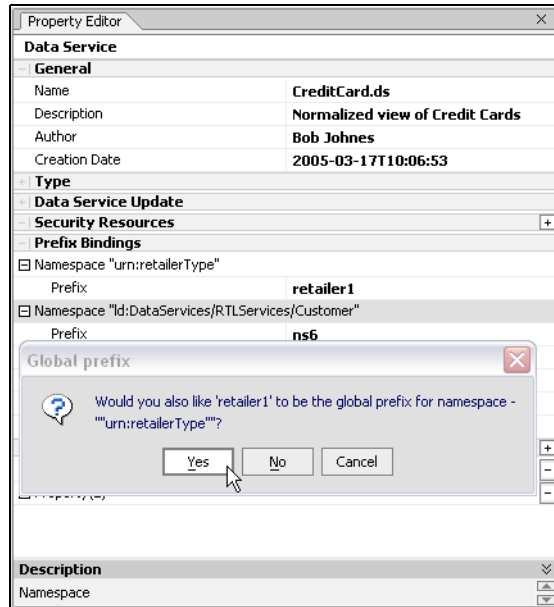
Figure 2-27 Refactoring Namespace Declarations



When you change a prefix binding you are also given the option of making the change throughout your project (globally).

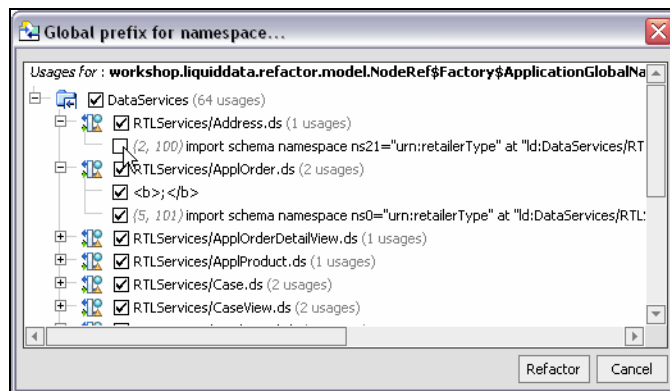
If you choose this option (see anywhere the uri (urn:retailerType) appears in your project, the prefix will become "retailer1".

Figure 2-28 Changing a Prefix Binding Throughout Your Project



If you choose Yes, a list of usages of the URI appears.

Figure 2-29 List of Prefix Bindings Potentially Affected by a Global Prefix Change



Warning: Although you can deselect any artifact that you do not want to be included in a refactor operation, doing so will invalidate that artifact and any files dependent on that artifact. For this reason selective deselection of artifacts scheduled for refactoring should generally not be employed.

Note: In the case of namespace prefixes, names should be changed (or not) based on readability or consistency issues. Neither a local or a global change will adversely affect your code.

Impacts of Various Refactoring Operations

It is useful to understand the various potential effects of a refactoring operation.

In this section each type of refactoring operation is described in terms of its potential impact on related artifacts.

Table 2-30 Refactoring Effects on Artifact Types

Artifact(s)	Renaming Operations	Move Operations	Safe Delete Operations
Data service	Renames data service Updates: <ul style="list-style-type: none"> • Name in source. • Namespace URI for data service functions • Dependent annotations • Dependent function references • Dependent model diagrams • Dependent data service controls • Function dependent on read function and relationship functions 	Moves data service to a new location in the project. Move operations update the same artifacts listed under Renaming Operations.	Deletes after warning regarding any dependencies.
<ul style="list-style-type: none"> • Data service read functions • Relationship functions • Private functions • XFL functions 	Updates: <ul style="list-style-type: none"> • Name in source. • External or internal references to this function in other function bodies, if any. • References to this function in inverse and equivalent transform annotations, if any. 	N/A	<ul style="list-style-type: none"> • Delete name in source. • Warns of any dependencies, including: <ul style="list-style-type: none"> – references to this function in other function bodies – references to this function in inverse and equivalent transform annotations.

Artifact(s)	Renaming Operations	Move Operations	Safe Delete Operations
<ul style="list-style-type: none"> • Namespace declarations • Schema import declarations 	Updates: <ul style="list-style-type: none"> • Prefix declaration and usages in source (local). • Prefix usages for the specified namespace URI for the entire project (global option). 	N/A	N/A
External schema declarations	Updates: <ul style="list-style-type: none"> • External schema URI (local) • External schema URI for the entire project (global option). 	N/A	N/A

Obtaining Enterprise Metadata

A first step in creating data services for the BEA Aqualogic Data Services Platform (DSP) is to obtain metadata from physical data needed by your application.

This chapter describes this process, including the following topics:

- [Creating Data Source Metadata](#)
- [Obtaining Metadata From Relational Sources](#)
 - [Importing Relational Table and View Metadata](#)
 - [Importing Stored Procedure-Based Metadata](#)
 - [Using SQL to Import Metadata](#)
- [Importing Web Services Metadata](#)
- [Importing Java Function Metadata](#)
- [Importing Delimited File Metadata](#)
- [Importing XML File Metadata](#)
- [Updating Data Source Metadata](#)

Creating Data Source Metadata

Metadata is simply information about the structure of a data source. 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 DSP, a *physical data service* is based almost entirely on the *introspection* of physical data sources.

Figure 3-1 Data Services Available to the RTL Sample Application

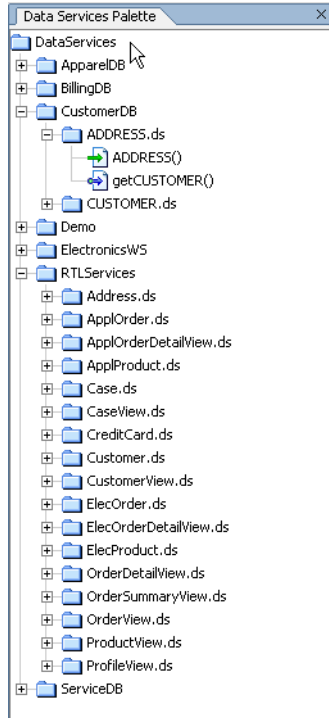


Table 3-2 list the types of sources from which DSP can create metadata.

Table 3-2 Data Sources Available for Creating Data Service Metadata

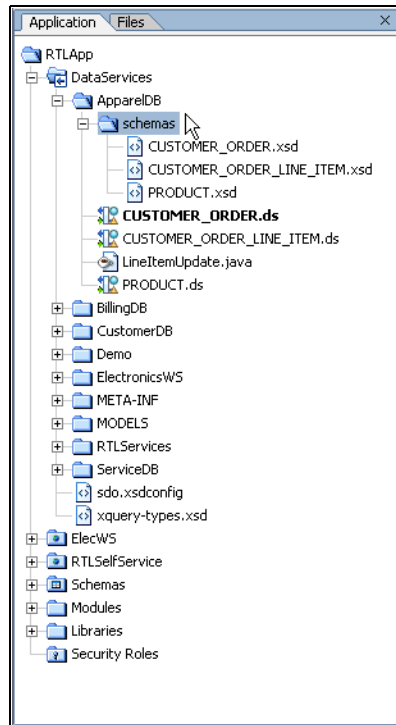
Data Source Type	Access
Relational (including tables, views, stored procedures, and SQL)	JDBC
Web services (WSDL files)	URI, UDDI, WSDL
Delimited (CSV files)	File-based data, such as spreadsheets.

Data Source Type	Access
Java functions (. java)	Programmatic
XML (XML files)	File- or data stream-based XML

When information about physical data is developed using the Metadata Import Wizard two things happen:

- A physical data service (*extension .ds*) is created in your DSP-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.

Figure 3-3 DSP Application Pane Displaying a Data Service and Its Schema Directory



You can import metadata on the data sources needed by your application using the DSP Metadata Import wizard. This wizard introspects available data sources and identifies data objects that can be rendered as data services and functions. Once created, physical data services become the building-blocks for queries and logical data services.

Data source metadata can be imported as Data Services Platform *functions* or *procedures*. 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. You can find a detailed description of source code annotations in “Understanding Data Services Platform Annotations” in the [XQuery Reference Guide](#).

For some data sources such as Web services imported metadata represents functions which typically return void (in other words, these functions perform operations rather than returning data). Such routines are classified as *side-effecting functions* or, more formally, as DSP *procedures*. You also have the option of marking routines imported from certain data sources as procedures. (See “[Identifying DSP Procedures](#)” on page 3-4.)

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”.

Note: DSP procedures are only associated with physical data services and can only be created through the metadata import process. So, for example, attempting to add procedures to a logical data service through Source View will result in an error condition.

Identifying DSP Procedures

When you import source metadata for Web services, relational stored procedures, or Java functions you have an opportunity to identify the metadata that represents side-effecting routines. A typical

example is a Web service that creates a new customer record. From the point of view of the data service such routines are procedures.

Procedures are not standalone; they always are part of a data service from the same data source.

When importing data from such sources the Metadata Import wizard automatically categorizes routines that return void as procedures. The reason for this is simply: if a routine does not return data it cannot inter-operate with other data service functions.

There are, however, routines that both return data *and* have side-effects; it is these routines which you need to identify as procedures during the metadata import process. Identification of such procedures provides the application developer with two key benefits:

- Creating a procedure during metadata input makes that the routine more easily available to the application programmer through the DSP `invokeProcedure()` API.
- If you import a routine that has side effects simply as a data service or as an executable function in an XML Function Library, invoking that routine through XQuery may have unexpected results, including the possibility that the routine will not be invoked at all. (The reason for this is that XQuery is a declarative language. You define your goals and the query engine determines how to achieve those goals. If a function is not intrinsic to achieving the overall goal of the query then its execution may be skipped even though it is specified as part of the overall query.)

[Table 3-4](#) lists common DSP operations, identifying which operations are available or unavailable for data service procedures.

Table 3-4 Data Services Platform Scope of Procedures

Artifact	Procedures Available	Procedures Unavailable
Data Services Platform IDE	<ul style="list-style-type: none"> • Metadata import operations • Function execution from Test View • DSP Control query function palette 	<ul style="list-style-type: none"> • DSP Palette • XQuery Editor function list • Query Plan Viewer function list • For use in ad hoc queries • For use in logical data services

Artifact	Procedures Available	Procedures Unavailable
Data Services Platform Console	<ul style="list-style-type: none"> • Function security settings • Left tree access 	<ul style="list-style-type: none"> • Cache operations
Data Services Platform APIs	<ul style="list-style-type: none"> • invokeProcedure() • Strongly typed API • DSP control 	<ul style="list-style-type: none"> • invoke() API (only for use with functions) • prepareExpression() for running ad hoc queries

Procedures greatly simplify the process of updating non-relational back-end data sources by providing an `invokeProcedure()` API. This API encapsulates the operational logic necessary to invoke relational stored procedures, Web services, or Java functions. In such cases update logic can be built into a back-end data source routine which, in turn, updates the data.

For information on updating non-relational sources and other special cases see “Enabling SDO Data Source Updates” in the *Client Application Developer’s Guide*.

For an example showing how you can identify side-effecting procedures during the metadata import process see “[Importing Web Services Metadata](#)” on page 3-37.

Obtaining Metadata From Relational Sources

You can obtain metadata on any relational data source available to the BEA WebLogic Platform. For details see the BEA Platform document entitled [“How Do I Connect a Database Control to a Database Such as SQL Server or Oracle.”](#)

Four types of metadata can be obtained from a relational data source:

- Table-based
- View-based
- Stored procedure-based
- SQL-based
- Functions

Note: When using an XA transaction driver you need to mark your data source’s connection pool to allow LocalTransaction in order for single database reads and updates to succeed.

For additional information in XA transaction adaptor settings see “Developing Adaptors” in BEA WebLogic Integration documentation:

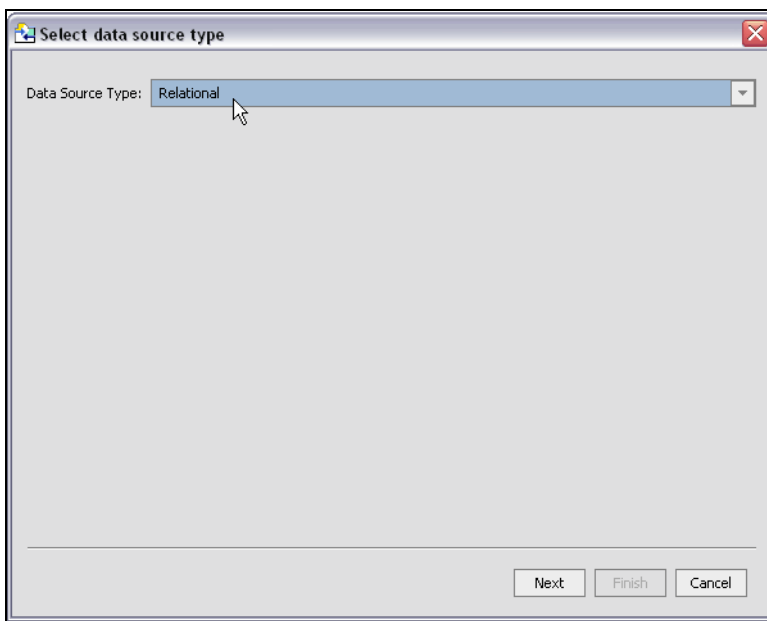
<http://e-docs.bea.com/wli/docs81/devadapt/dbmssamp.html>

Importing Relational Table and View Metadata

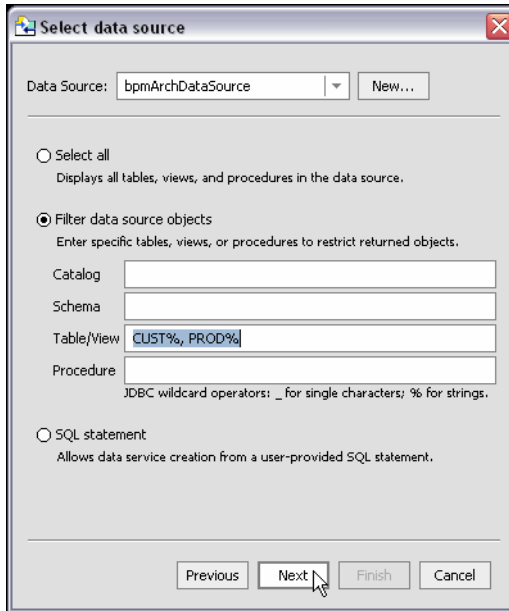
To create metadata on relational tables and views follow these steps:

1. Select the project in which you want to create your metadata. For example, if you have a project called myLDProject right-click on the project name and select Import Source Metadata... from the pop-up menu. Click Next.
2. From the available data sources in the Import Wizard select Relational (see [Figure 3-5](#)).

Figure 3-5 Selecting a Relational Source from the Import Metadata Wizard



3. Either select a data source from available sources or make a new data source available to the WLS.

Figure 3-6 Import Data Source Metadata Selection Dialog Box

Data Object Selection Options

For information on creating a new data source see [“Creating a New Data Source” on page 3-10](#).

If you choose to select from an existing data source, several options are available ([Figure 3-6](#)).

Select All Database Objects

If you choose to select all, a table will appear containing all the tables, views, and stored procedures in your data source organized by catalog and schema.

Filter Data Source Objects

Sometimes you know exactly the objects in your data source that you want to turn into data services. Or your data source may be so large that a filter is needed. Or you may be looking for objects with specific naming characteristics (such as %audit2003%, a string which would retrieve all objects containing the enclosed string).

In such cases you can identify the exact parts of your relational source that you want to become data service candidates using standard JDBC wildcards. An underscore (_) creates a wildcard for an

individual character. A percentage sign (%) indicates a wildcard for a string. Entries are case-sensitive.

For example, you could search for all tables starting with CUST with the entry: CUST%. Or, if you had a relational schema called ELECTRONICS, you could enter that term in the Schema field and retrieve all the tables, views, and stored procedure that are a part of that schema.

Another example:

```
CUST%, PAY%
```

entered in the Tables/Views field retrieves all tables and views starting with either CUST or PAY.

Note: 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.

For relational tables and views you should choose either the Select all option or Selected data source objects.

You can also 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 would also want to make a nonsense entry in the Table/View field to avoid retrieving all tables and views in the database.

For details on stored procedures see [“Importing Stored Procedure-Based Metadata” on page 3-16](#).

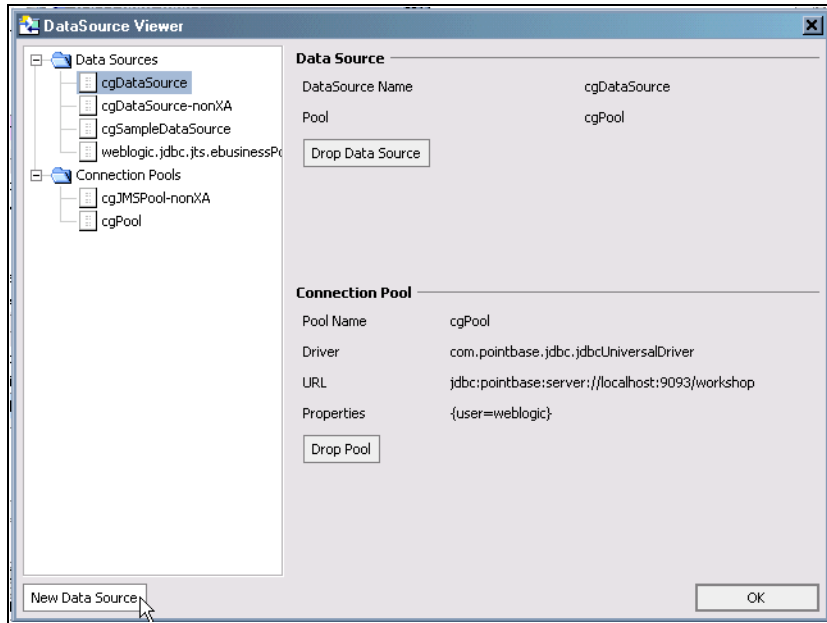
SQL statement

Allows you to enter an SQL statement that is used as the basis for creating a data service. See [“Using SQL to Import Metadata” on page 3-31](#) for details.

Creating a New Data Source

Most often you will work with existing data sources. However, if you choose New... the WLS DataSource Viewer appears ([Figure 3-7](#)). Using the DataSource Viewer you can create new data pools and sources.

Figure 3-7 BEA WebLogic Data Source Viewer



For details on using the DataSource Viewer see “[Configuring a Data Source](#)” in WebLogic Workshop documentation.

Selecting an Existing Data Source

Only data sources that have set up through the BEA WebLogic Administration Console are available to a Data Services Platform application or project. In order for the BEA WebLogic Server used by DSP to access a particular relational data source you need to set up a JDBC connection pool and a JDBC data source.

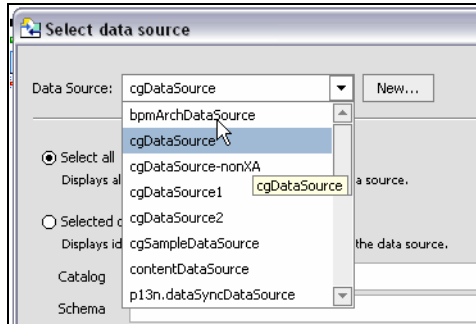
- For details on setting up a JDBC connection pool see:

http://e-docs.bea.com/wls/docs81/ConsoleHelp/domain_jdbccconnectionpool_config_general.html

- For details on setting up a JDBC data source see:

http://e-docs.bea.com/wls/docs81/ConsoleHelp/domain_jdbcdatasource_config.html

Figure 3-8 Selecting a Data Source

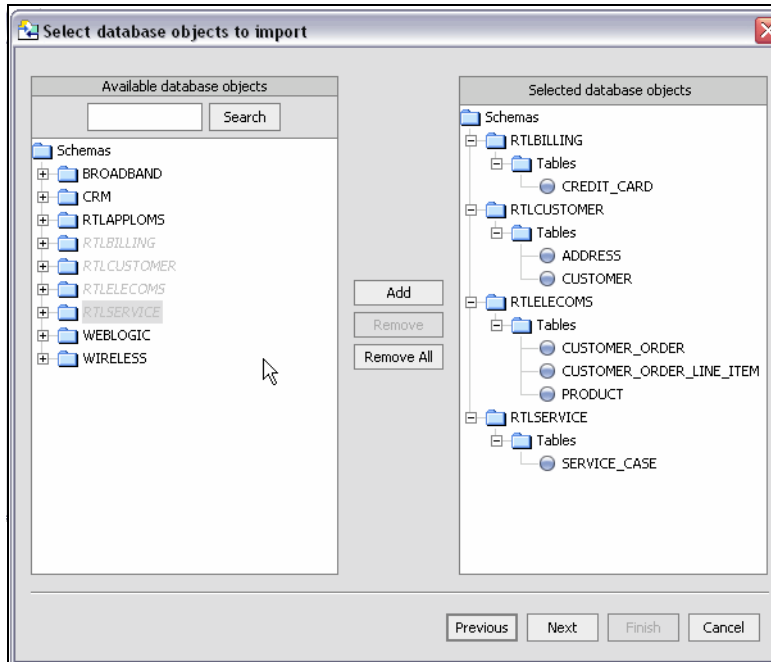


Once you have selected a data source, you need to choose how you want to develop your metadata — by selecting all objects in the database, by filtering database objects, or by entering a SQL statement. (see [Figure 3-6](#)).

Creating Table- and View-Based Metadata

Once you have selected a data source and any optional filters, a list of available database objects appears.

Figure 3-9 Identifying Database Objects to be Used as Data Services



Using standard dialog commands you can add one or several tables to the list of selected data objects. To deselect a table, select that table in the right-hand column and click Remove.

A Search field is also available. This is useful for data sources which have many objects. Enter a search string, then click Search repeatedly to move through your list.

4. Once you have selected one or several data sources, click Next to verify the location of the to-be-created data services and the names of your new data services.

The imported data summary screen:

- Lists selected objects by name. You can mouse over the XML type to see the complete path (Figure 3-10).
- Lists the location of the generated data service in the current application.
- Identifies any name conflicts. Name conflicts occur when there is an data service of the same name present in the target directory. Any name conflicts are highlighted in red.

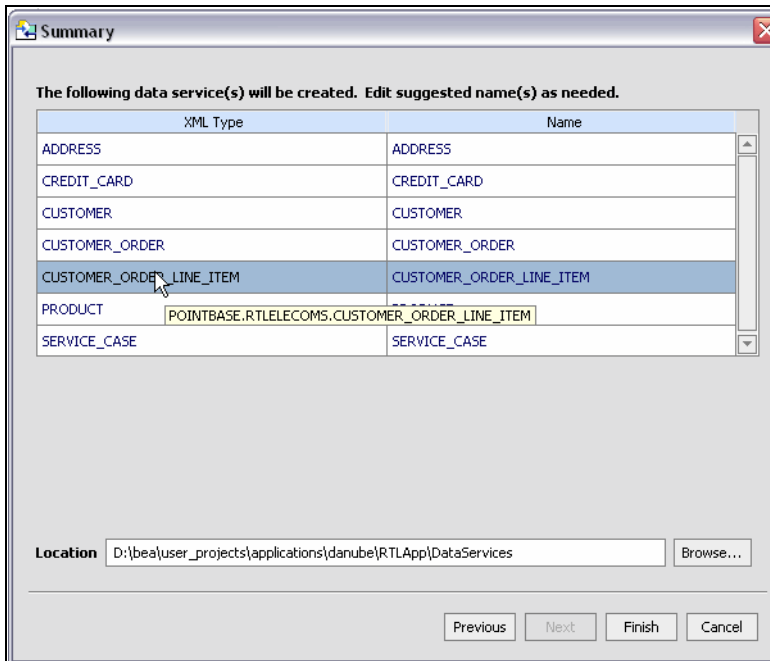
You can edit the file name to clarify the name or to avoid conflicts. Simply click on the name of the file and make any editing changes.

Alternatively, choose Remove All to return to the initial, nothing-is-selected state.

5. 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.

In such cases the name(s) of the data service(s) having name conflicts appear in red. Simply change to a unique name using the built-in line editor.

Figure 3-10 Relational Source Import Data Summary Screen



6. Click Finish. A data service will be created for each object selected. The file extension of the created data services will always be `.ds`.

Database-specific Considerations

Database vendors variously support database catalogs and schemas. [Table 3-11](#) describes this support for several major vendors.

Table 3-11 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 <code>db2admin</code> .
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 <code>dbo</code> . 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>

Once you have created your data services you are ready to start constructing logical views on your physical data. See [Chapter 4, “Designing Data Services.”](#) and [Chapter 5, “Modeling Data Services.”](#)

Importing Stored Procedure-Based Metadata

Enterprise databases utilize stored procedures to improve query performance, manage and schedule data operations, enhance security, and so forth. You can import metadata based on stored procedures. Each stored procedure becomes a data service.

Note: Refer to your database documentation for details on managing stored procedures.

Stored procedures are essentially database objects that logically group a set of SQL and native database programming language statements together to perform a specific task.

[Table 3-12](#) defines some commonly used terms as they apply to this discussion of stored procedures.

Table 3-12 Terms Commonly Used When Discussing Stored Procedures

Term	Usage
Function	A function is identical to a procedure except a function always return one or more values to the caller and a procedure never returns a value. The value can be a simple type, a row type, or a complex user defined type.
Package	A package is a group of related procedures and functions, together with the cursors and variables they use, stored together in a database for continued use as a unit. Similar to standalone procedures and functions, packaged procedures and functions can be called explicitly by applications or users.
Stored Procedure	A sequence of programming commands written in an extended SQL (such as PL/SQL or T-SQL), Java or XQuery, stored in the database where it is to be used to maximize performance and enhance security. The application can call a procedure to fetch or manipulate database records, rather than using code outside the database to get the same results. Stored procedures do not return values.
DSP Procedure	Typically a routine which performs work but does not return data. An example would be a routine callable from a data service which writes information to a log file.
Rowset	The set of rows returned by a procedure or query.
Result set	JDBC term for rowset.
Parameter mode	Procedures can have three modes: IN, OUT, and INOUT. There roughly correspond to “write”, “read”, and “read/write”.

Importing Stored Procedures Using the Metadata Import Wizard

Imported stored procedure metadata is quite similar to imported metadata for relational tables and views. The initial three steps for importing stored procedures are the same as importing any relational metadata (described under [“Importing Relational Table and View Metadata” on page 3-8](#)).

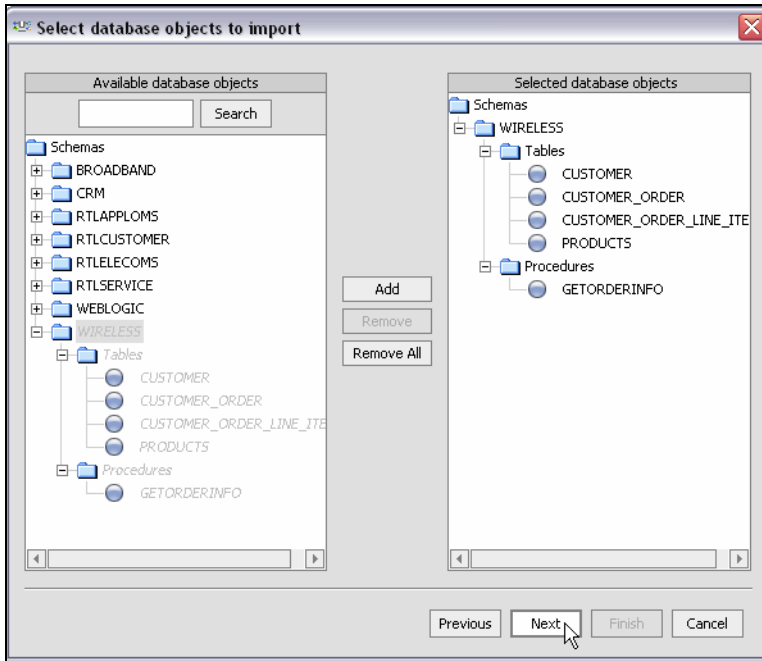
Note: If a stored procedure has only one return value and the value is either simple type or a RowSet which is mapping to an existing schema, no schema file created.

Also see in the Data Services Platform [Samples Tutorial Part II](#):
- Lesson 31: Accessing Data in Stored Procedures

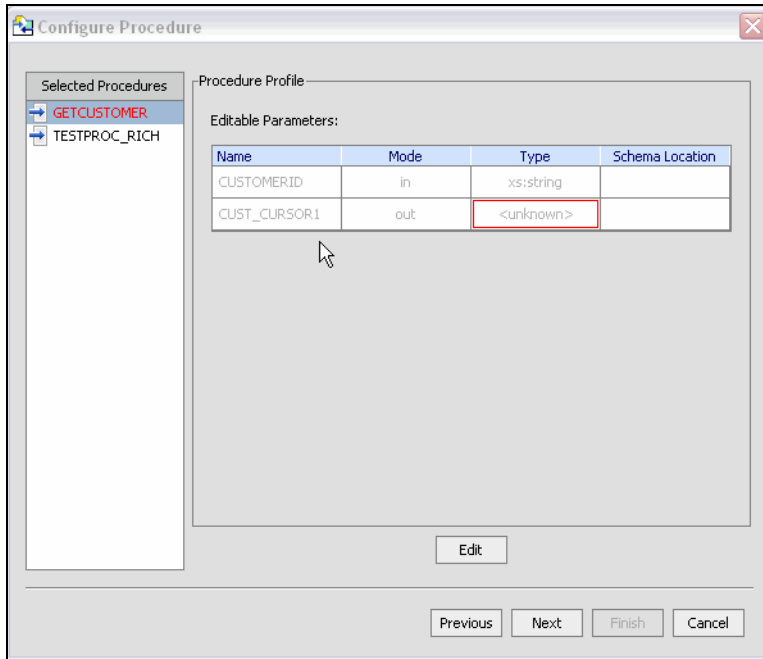
You can select any combination of database tables, views, and stored procedures. If you select one or several stored procedures, the Metadata Import wizard will guide you through the additional steps required to turn a stored procedure into a data service. These steps are:

1. Select one or several stored procedures. A data service can represent only one stored procedure. In other words, if you have five stored procedures, you will create five data services.

Figure 3-13 Selecting Stored Procedure Database Objects to Import



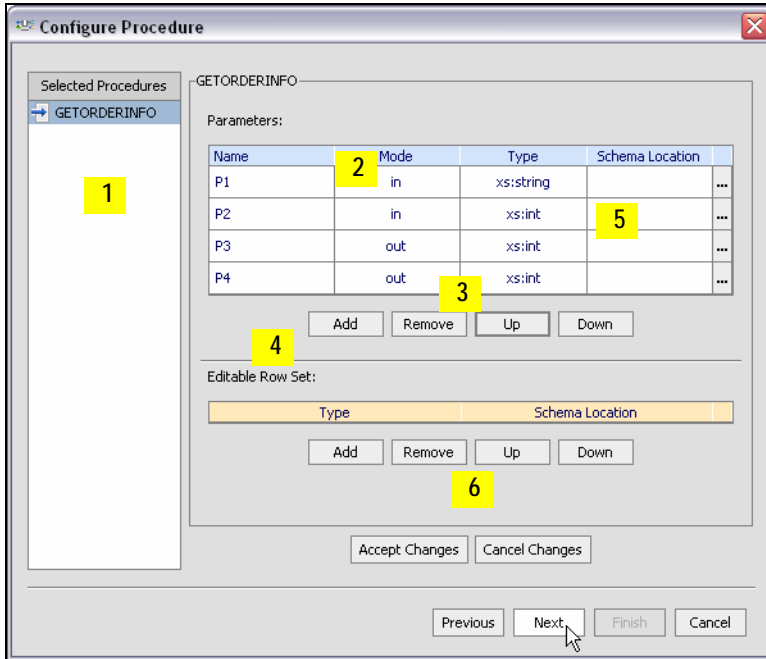
2. After you have added the database objects that you want to become data services.
3. From the selected procedures (Figure 3-14) configure each stored procedure. If your stored procedure has an OUT parameter requiring a complex element, you may need to provide a schema.

Figure 3-14 Configuring a Stored Procedure in Pre-editing Mode

Data objects in the stored procedure that cannot be identified by the Metadata Import wizard will appear in red, without a datatype. In such cases you need to enter Edit mode (click the Edit button) to identify the data type.

Your goal in correcting an “<unknown>” condition associated with a stored procedure (Figure 3-14) 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.

Figure 3-15 Stored Procedure in Editing Mode (with Callouts)



4. Edit your procedure as appropriate using the following steps:
 - a. Select a stored procedure from the complete list of stored procedures that you want to turn into data services.
 - b. Edit the stored procedure parameters including setting mode (in, out, inout), type, and for out parameters, schema location.
 - c. Verify and, if necessary, add, remove, or change the order of parameters.
 - d. Verify and, if necessary, add, remove, or change any editable rowset.
 - e. Supply a return type (either simple or complex through identifying a schema location) in cases the Metadata Import wizard was unable to determine the type.
 - f. Accept or cancel your changes.

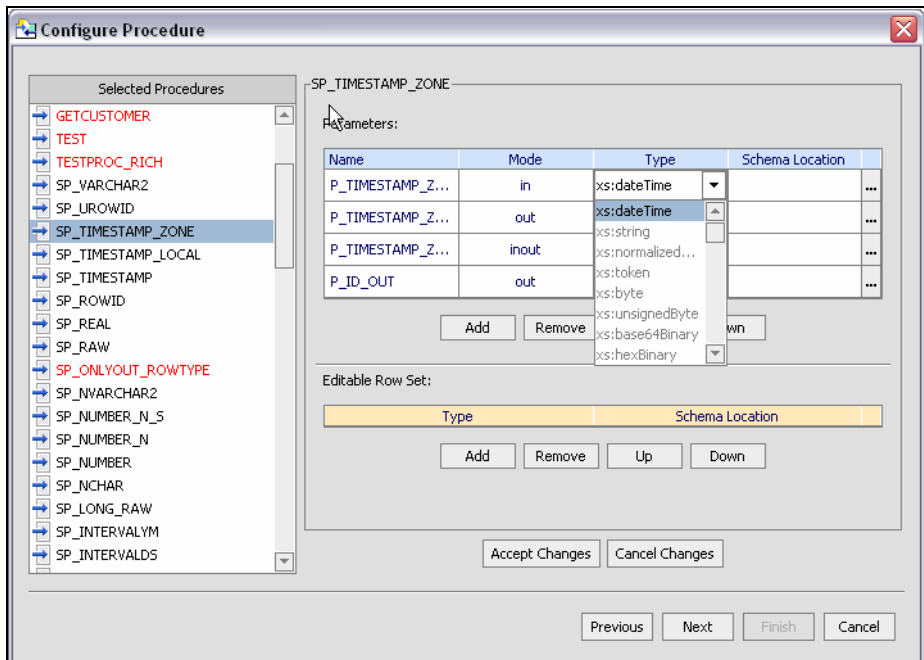
You need to complete information for each selected stored procedure before you can move to the next step. In particular, any stored procedures shown in red must be addressed.

Details for each section of the stored procedure import dialog box appear below.

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.

Figure 3-16 Changing the Type of an Element in a Stored Procedure



If the type is complex, you may need to supply an appropriate schema. Click on the schema location button and either enter a schema path name 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. For example:

```
http://temp.openuri.org/schemas/Customer.xsd}CUSTOMER
```

Procedure Parameters

The Metadata Import wizard, working through JDBC, also identifies any stored procedure parameters. This includes the name, mode (input [in], output [out], or bidirectional [inout]) and data type. The out mode supports the inclusion of a schema.

Complex type is only supported under three conditions:

- as the output parameter

- as the return type
- as a rowset

All parameters are editable, including the name.

Note: If you make an incorrect choice you can use the Previous, then Next button to return the dialog to its initial state.

Rowsets

Not all databases support rowsets. In addition, JDBC does not report information related to defined rowsets. In order to create data services from stored procedures that use rowset information, supply the correct ordinal (matching number) and a schema. If the schema has multiple global elements, you can select the one you want from the Type column. Otherwise the type will be 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.

Complete the importation of your procedures by reviewing and accepting items in the Summary screen (see [step 4. in “Importing Relational Table and View Metadata”](#) for details).

Note: XML types in data services generated from stored procedures do not display native types. However, you can view the native type in the Source View pragma (see [“Working with XQuery Source”](#)).

Handling Stored Procedure Rowsets

A rowset type is a complex type. The name of the rowset type can be:

- The parameter name (in case of a input/output or output only parameter)
- An assigned name such as RETURN_VALUE (if return value)
- The referenced element name (result rowsets) in a user-specified schema

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

Note: All rowset-type definitions must conform to this structure.

In some cases the Metadata Import 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 through the wizard.

5. Mark Appropriate Imported Stored Procedure Metadata as DSP Procedures

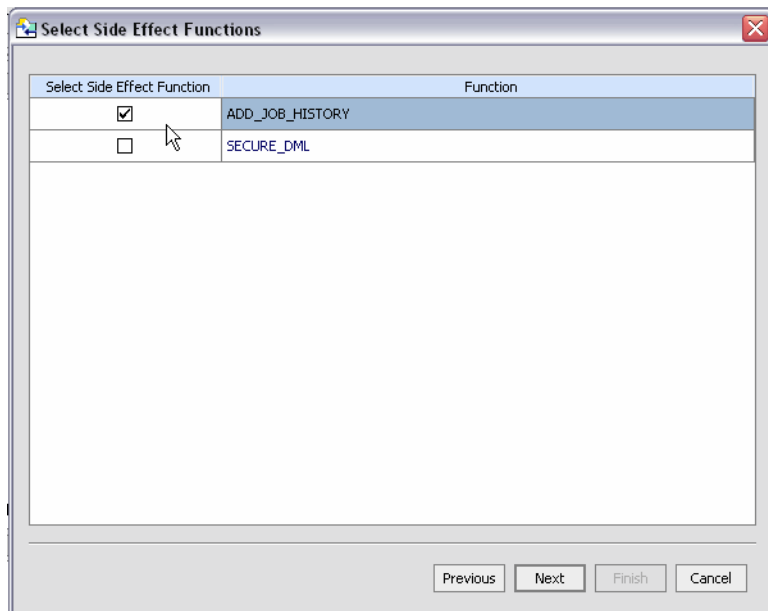
Identifying Stored Procedures as DSP Procedures

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 noXML type; in fact they typically return nothing (or *void*). Instead the data service knows that they have side-effects and are associated as procedures with a data service of the same data source.

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 DSP procedure during the metadata import process.

After you have identified the stored procedures that you want to add to your data service or XML file library (XFL), you also have an opportunity to identify which of these should be identified as DSP procedures.

Figure 3-17 Identifying Stored Procedures Having Side Effects



Note: DSP procedures based around atomic (simple) types are collected in an identified XML function library (XFL) file. Other procedures need to be associated with a data service that is local to your DSP-enabled project.

Internal Stored Procedure Support

You can import metadata for an internal stored procedures. See [“Filter Data Source Objects” on page 3-9](#) for details.

Stored Procedure Version Support

Only the most recent version of a stored procedure can be imported into DSP. For this reason you cannot identify a version number when importing a stored procedure through the Metadata Import wizard. Similarly, adding a version number to DSP source will result in a query exception.

Stored Procedure Support for Commonly Used Databases

Each database vendor approaches stored procedures differently. XQuery support limitations are, in general, due to JDBC driver limitations.

General Restriction

DSP does not support rowset as an input parameter.

Oracle Stored Procedure Support

[Table 3-18](#) summarizes DSP support for Oracle database procedures.

Table 3-18 Support for Oracle Store Procedures

Term	Usage
Procedure types	<ul style="list-style-type: none">• Procedures• Functions• Packages
Parameter modes	<ul style="list-style-type: none">• Input only• Output only• Input/Output• None

Term	Usage
Parameter data types	<p>Any Oracle PL/SQL data type except those listed below:</p> <ul style="list-style-type: none"> • ROWID • UROWID <p>Note: 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.</p> <p>For a list of database types supported by DSP see “Relational Data Types-to-Metadatas Conversion” on page 3-32.</p>
Data returned from a function	Oracle supports returning PL/SQL data types such as NUMBER, VARCHAR, %TYPE, and %ROWTYPE as parameters.
Comments	<p>The following identifies limitations associated with importing Oracle database procedure metadata.</p> <ul style="list-style-type: none"> • The Metadata Import wizard 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 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

[Table 3-19](#) summarizes DSP support for Sybase SQL Server database procedures.

Table 3-19 Support for Sybase Stored Procedures

Term	Usage
Procedure types	<ul style="list-style-type: none"> • Procedures • Grouped procedures • Functions <p>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.</p>
Parameter modes	<ul style="list-style-type: none"> • Input only • Output only
Parameter data types	For the complete list of database types supported by DSP see “Relational Data Types-to-Metadata Conversion” on page 3-32.
Data returned from a function	<p>Sybase functions supports returning a single value or a table.</p> <p>Procedures return data in the following ways:</p> <ul style="list-style-type: none"> • 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. • As a global cursor that can be referenced outside the stored procedure supports, returning single value or multiple values.
Comments	<p>The following identifies limitations associated with importing Sybase database procedure metadata:</p> <ul style="list-style-type: none"> • 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 this case, the return mode will be UNKNOWN, preventing importation of the metadata. To proceed, you need to set the correct mode in order to proceed. • Only data types generally supported by DSP metadata import can be imported as part of stored procedures.

IBM DB2 Stored Procedure Support

[Table 3-20](#) summarizes DSP support for IBM DB2 database procedures.

Table 3-20 Support for IBM Store Procedures

Term	Usage
Procedure types	<ul style="list-style-type: none"> • Procedures • Functions • Packages <p>Each function is also categorized as a scalar, column, row, or table function. Here are additional details on function categorization:</p> <ul style="list-style-type: none"> • A scalar function is one that 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 function that returns a table to the SQL statement that referenced it.
Parameter modes	<ul style="list-style-type: none"> • Input only • Output only • Input/output
Parameter data types	For the complete list of database types supported by DSP see “Relational Data Types-to-Metadate Conversion” on page 3-32.
Data returned from a function	DB2 supports returning a single value, a row of values, or a table.
Comments	<p>The following identifies limitations associated with importing DB2 database procedure metadata:</p> <ul style="list-style-type: none"> • 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. <p>Float, double, and decimal data types are not supported as input/output parameters.</p> <ul style="list-style-type: none"> • Only data types generally supported by DSP metadata import can be imported as part of stored procedures.

Informix Stored Procedure Support

[Table 3-21](#) summarizes DSP support for Informix database stored procedures.

Table 3-21 Support for Informix Stored Procedures

Term	Usage
Procedure types	<ul style="list-style-type: none"> • Procedures • Functions <p>A function may return more than one value.</p>
Parameter modes	<ul style="list-style-type: none"> • Input only • Output only • Input/output
Parameter data types	<p>For the complete list of database types supported by DSP see “Relational Data Types-to-Metadata Conversion” on page 3-32.</p>

Term	Usage
Data returned from a function	Informix supports returning single value, multiple values, and rowsets.
Comments	<p data-bbox="431 440 1170 493">Informix treats return value(s) from functions or procedures as a rowset. For this reason a rowset needs to be defined for the return value(s).</p> <p data-bbox="431 510 857 534">The following limitations have been identified:</p> <p data-bbox="431 552 767 576">Informix Native Driver Limitations</p> <ul data-bbox="431 593 1170 812" style="list-style-type: none"> <li data-bbox="431 593 1170 645">• All parameter names are missing; instead in the Metadata Import wizard parameters are assigned the same system-generated name: RETURN VALUE <li data-bbox="431 697 1170 812">• All return values are reported as parameters with mode <code>return</code> instead of mode <code>result</code>. This leads to a problem since only the first parameter should be in mode <code>return</code>. This also causes a runtime failure. The workaround is to get the value(s) using resultset. <p data-bbox="431 829 758 854">BEA WebLogic Driver Limitations</p> <ul data-bbox="431 871 1170 1072" style="list-style-type: none"> <li data-bbox="431 871 1170 1072">• Input parameter names and return values are reported as <code>result</code> mode. Since there is no name declared for those return values insider the procedure, their corresponding parameters have no name either. The problem is that this does not model “result” parameters as a group; thus result parameters are likely to repeat as multiple rows. (Unlike the Oracle cursor which has the cursor itself as an outer parameter, there is no holder for Informix result parameters.) <p data-bbox="431 1090 606 1114">Recommendations</p> <p data-bbox="431 1131 1170 1183">Due to the limitations described above, the following approach is suggested for importing Informix stored procedure metadata:</p> <ol data-bbox="431 1201 1170 1444" style="list-style-type: none"> <li data-bbox="431 1201 915 1225">1. Use the BEA WebLogic driver wherever possible. <li data-bbox="431 1242 1170 1295">2. Define a schema that matches the return value structure (using the same approach as external schemas for other databases). <li data-bbox="431 1312 1170 1416">3. In the Metadata Import wizard’s stored procedure section, remove all the parameters in the Result section using Edit mode. Add a result parameter and associate it with the schema defined in step 2. (If you are using the Informix native driver assign a proper name for the input parameters.) <li data-bbox="431 1433 1170 1451">4. Manually edit the parameter’s section of the generated data service file.

Microsoft SQL Server Stored Procedure Support

Table 3-22 summarizes DSP support for Microsoft SQL Server database procedures.

Table 3-22 DSP Support for Microsoft SQL Server Stored Procedures

Term	Usage
Procedure types	<p>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.</p> <p>Inline table-valued and multi-statement table-valued functions return rowsets.</p>
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.
Data returned from a function	<p>SQL Server functions supports returning a single value or a table.</p> <p>Data can be returned in the following ways:</p> <ul style="list-style-type: none"> • 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	<p>The following identifies limitations associated with importing SQL Server procedure metadata.</p> <ul style="list-style-type: none"> • 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). • Only data types generally supported by DSP metadata import can be imported as part of stored procedures.

Using SQL to Import Metadata

One of the relational import metadata options (see [Figure 3-6](#)) is to use an SQL statement to customize introspection of a data source. If you select this option the SQL Statement dialog appears.

Figure 3-23 SQL Statement Dialog Box

SQL Statement
Enter SELECT statement. User ? for parameters.

Parameters
Enter parameter values.

Position	Type
1	
2	
3	

Add Remove

Previous Next Finish Cancel

You can type or paste your SELECT statement into the statement box ([Figure 3-23](#)), indicating parameters with a “?” question-mark symbol. Using one of the DSP data samples, the following SELECT statement can be used:

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

RTLCUSTOMER is a schema in the data source, CUSTOMER is, in this case, a table.

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

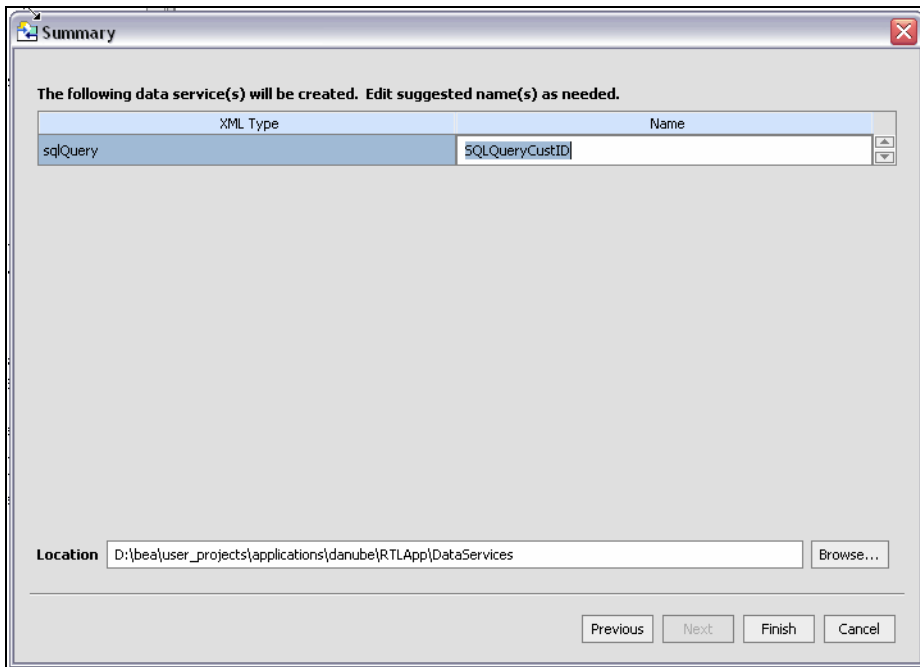
The next step is to assign a data service name.

When you run your query under Test View, you will need to supply the parameter in order for the query to run successfully.

Once you have entered your SQL statement and any required parameters click Next to change or verify the name and location of your new data service.

Also see in the Data Services Platform [Samples Tutorial Part II](#):
- Lesson 22: Creating Data Services Based on SQL Statements

Figure 3-24 Relational SQL Statement Imported Data Summary Screen



The imported data summary screen identifies a proposed name for your new data service.

The final steps are no different than you used to create a data service from a table or view.

Relational Data Types-to-Metadatas Conversion

The following table shows how data types provided by various relational databases are converted into XQuery data types. Types are listed in alphabetical order.

Table 3-25 Relational Data Types and Their XQuery Counterparts

Datatype Name	XQuery Equivalent	Oracle	IBM DB2	Sybase	Informix	Microsoft SQL Server	Pointbase
ARRAY	not supported	X					
BFILE	not supported	X					
BIGINT	xs:long		X			X	X
BINARY	xs:hexBinary			X		X	
BIT	xs:boolean			X		X	
BLOB	xs:hexBinary	X	X		X		X
BOOLEAN	xs:Boolean				X		X
BYTE	xs:hexBinary				X		
CHAR	xs:string	X	X	X	X	X	X
CHAR() FOR BIT DATA	xs:hexBinary		X				
CLOB	xs:string	X	X		X		X
DATE	xs:date		X				X
DATE	xs:datetime	X					
DATETIME	xs:datetime			X	X	X	
DECIMAL{n, s} s>0	xs:decimal		X	X	X	X	X
DECIMAL{n}	xs:integer		X	X	X	X	X
DOUBLE	xs:double		X				
DOUBLE PRECISION	xs:double			X			X
FLOAT	xs:double	X	X	X	X	X	X
IMAGE	xs:hexBinary			X		X	
INT	xs:int			X		X	
INT8	xs:long				X		
INTEGER	xs:int		X		X		X

Obtaining Enterprise Metadata

Datatype Name	XQuery Equivalent	Oracle	IBM DB2	Sybase	Informix	Microsoft SQL Server	Pointbase
INTERVAL	not supported				X		
INTERVALDS	xd:dayTimeDuration	X					
INTERVALYM	xd:yearMonthDuration	X					
LONG	xs:string	X					
LONG RAW	xs:hexBinary	X					
LONG VARCHAR	xs:string		X				
LONG VARCHAR FOR BIT DATA	xs:hexBinary		X				
LVARCHAR	xs:string				X		
MONEY	xs:decimal			X	X	X	
MSLABEL	not supported	X					
NCHAR	xs:string	X		X	X	X	
NTEXT	xs:string					X	
NUMBER	xs:double	X					
NUMBER{n, s} s<0	xs:integer	X					
NUMBER{n, s} s>0	xs:decimal	X					
NUMBER{n}	xs:integer	X					
NUMERIC{n, s} s>0	xs:decimal		X	X		X	X
NUMERIC{n}	xs:decimal		X	X		X	X
NVARCHAR	xs:string			X	X	X	
NVARCHAR2	xs:string	X					
RAW	xs:hexBinary	X					
REAL	xs:float		X	X		X	X

Datatype Name	XQuery Equivalent	Oracle	IBM DB2	Sybase	Informix	Microsoft SQL Server	Pointbase
REF	not supported	X					
ROWID	xs:string	X					
SERIAL	not supported				X		
SERIAL8	not supported				X		
SMALLDATETIME	xs:datetime			X		X	
SMALLFLOAT	xs:float				X		
SMALLINT	xs:short		X	X	X	X	X
SMALLMONEY	xs:decimal			X		X	
SQL_VARIANT	xs:string					X	
STRUCT	not supported	X					
SYSNAME	xs:string			X		X	
TEXT	xs:string			X	X	X	
TIME	xs:time		X				X
TIMESTAMP	xs:datetime	X	X				X
TIMESTAMP	xs:hexBinary					X	
TIMESTAMP WITH LOCAL TIME ZONE	xs:datetime	X					
TIMESTAMP WITH TIME ZONE	xs:datetime	X					
TINYINT	xs:short			X		X	
UNIQUEIDENTIFIER	xs:hexbinary					X	
UROWID	xs:string	X					
VARBINARY	xs:hexBinary			X		X	
VARCHAR	xs:string		X	X	X	X	X

Obtaining Enterprise Metadata

Datatype Name	XQuery Equivalent	Oracle	IBM DB2	Sybase	Informix	Microsoft SQL Server	Pointbase
VARCHAR() FOR BIT DATA	xs:hexBinary		X				
VARCHAR2	xs:string	X					

Importing Web Services Metadata

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.

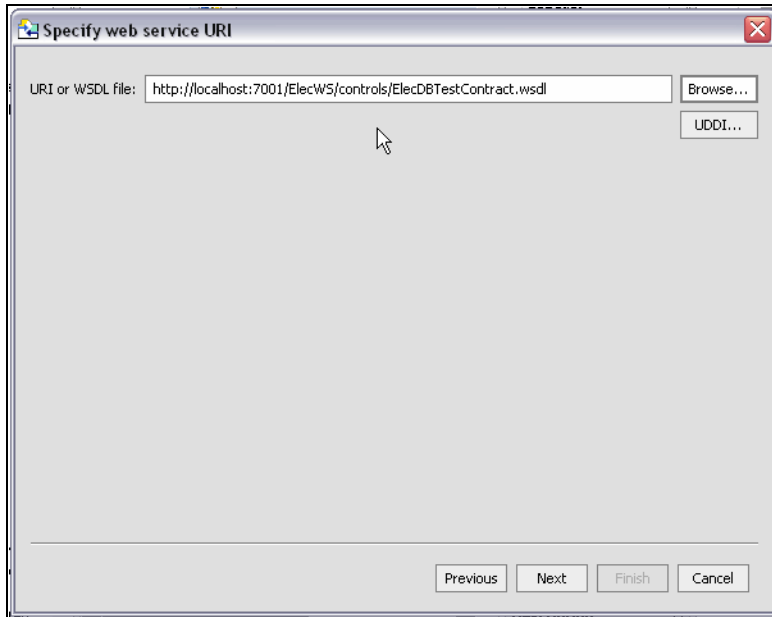
Note: Multi-dimensional arrays in RPC mode are not supported.

Creating a data service based on a Web service definition (schema) is similar to importing relational data source metadata (see [“Importing Relational Table and View Metadata” on page 3-8](#)).

Here are the Web service-specific steps involved:

1. Select the DSP-based project in which you want to create your Web service metadata. For example, if you have a project called DataServices right-click on the project name and select Import Metadata... from the pop-up menu.
2. From the available data sources in the Metadata Import wizard select Web service and click Next.
3. There are three ways to access a Web service:
 - From a Web service description language (WSDL) file that is in your current DSP project.
 - From a URI which is a WSDL accessible via a URL (HTTP).
 - From a Universal Description, Discovery, and Integration service (UDDI).

Figure 3-26 Locating a Web Service



Note: For the purpose of showing how to import Web service metadata a WSDL file from the RTLApp sample is used for the remaining steps. If you are following these instructions enter the following into the URI field to access the WSDL included with RTLApp:

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

4. From the selected Web service choose the operations that you want to turn into data services or XFL functions.
5. Identify which, if any, Web service-based data services should be marked as having side-effects.

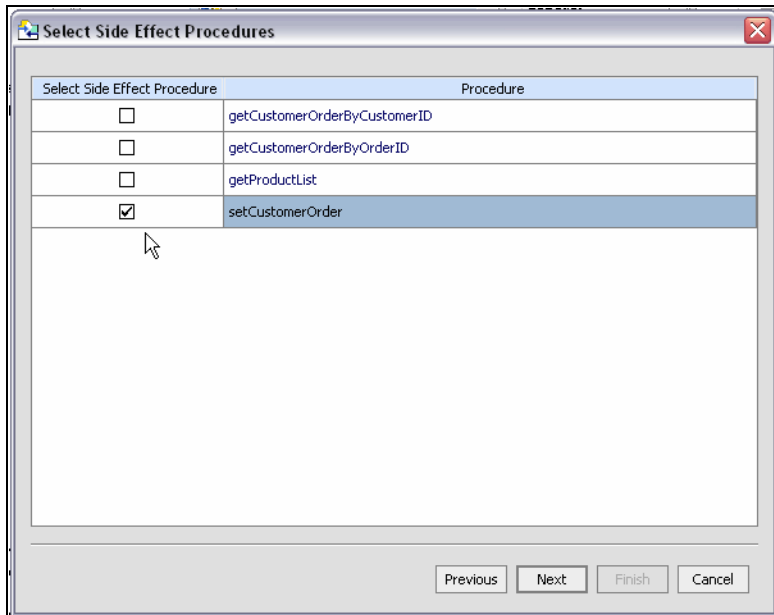
Note: Imported operations returning void are automatically imported as DSP procedures. You can identify other operations as procedures using the Select Side Effect Procedures dialog (Figure 3-27).

It is often convenient to leverage side-effecting operations as part of managing enterprise information through a data service. An obvious example would be to manage standalone update or security functions through data services. The data service registers that such operations have side-effects.

Procedures are not standalone; they always are part of a data service from the same data source.

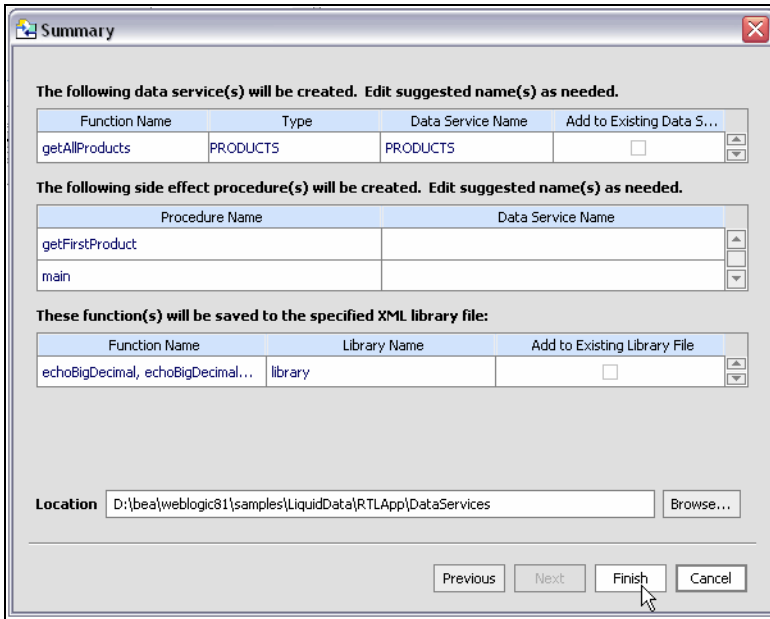
Web services are side-effecting from the perspective of the data service even when they do return data. In such cases, you need to associate the Web service operation with a data service during the metadata import process.

Figure 3-27 Marking Imported Operations DSP Procedures



Procedures must be associated with a data service that is local to a DSP-enabled project.

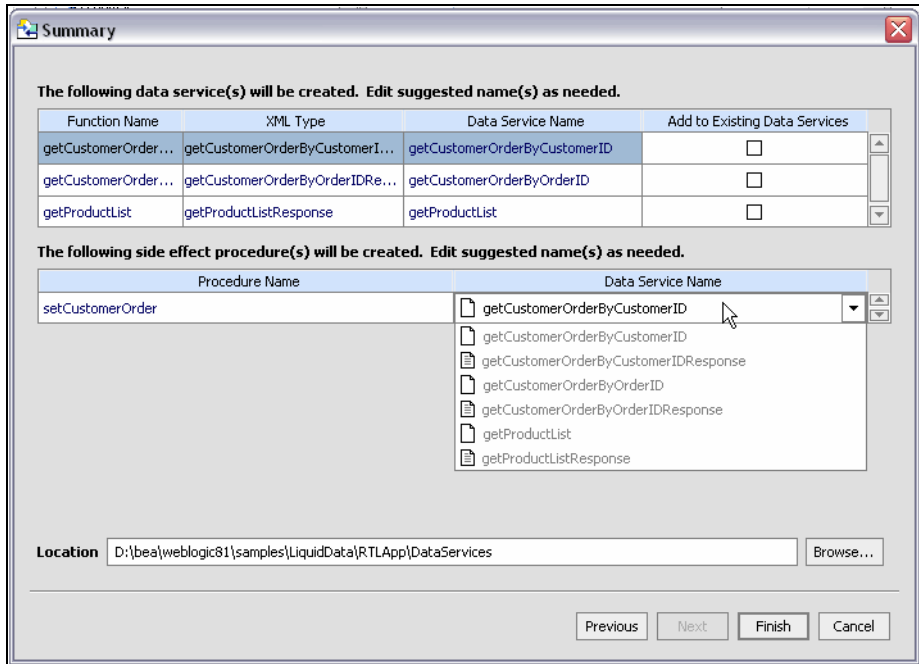
Figure 3-28 Identifying Web Service Operations to be Used as Data Services



Using standard dialog editing commands you can select one or several operations to be added to the list of selected Web service operations. To deselect an operation, click on it, then click Remove. Or choose Remove All to return to the initial state.

6. Click Next to verify the location of the to-be-created data services and their names.

Figure 3-29 Web Services Imported Data Summary Screen



The summary screen shown in [Figure 3-29](#):

- Lists the Web service operations you have selected.
- Lists the target name for the generated data services.
- Identifies in red any data service name conflicts.

Even if there are no name conflicts you may want to change a data service name for clarity. Simply click on the name of the data service and enter the new name.

- Provides an option for adding the function to an existing data service based on the same WSDL. This option is only enabled if such a data service exists in your project. If there are several data services based on the same WSDL, a dropdown menu allows you to choose the data service for your function.

Note: Web Service functions identified as side-effecting procedures must be associated with a data service based on the same WSDL.

Note: When importing a Web service operation that itself has one or more dependent (or referenced) schemas, the Metadata Import 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.

7. Click Finish. A data service will be created for each selected operation.

Also see in the Data Services Platform [Samples Tutorial Part I](#):
- Lesson 6: Accessing Data in Web Services

Testing Metadata Import With an Internet Web Service URI

If you are interested in trying the Metadata Import wizard with an internet Web service URI, the following page (available as of this writing) provides sample URIs:

```
http://www.strikeiron.com/BrowseMarketplace.aspx?c=14&m=1
```

Simply select a topic and navigate to a page showing the sample WSDL address such as:

```
http://ws.strikeiron.com/SwanandMokashi/StockQuotes?WSDL
```

Copy the string into the Web service URI field and click Next to select the operations want to turn into sample data services or procedures.

Another external Web service that can be used to test metadata import can be located at:

```
http://www.whitemesa.net/wsdl/std/echoheadersvc.wsdl
```

Importing Java Function Metadata

You can create metadata based on custom Java functions. When you use the Metadata Import wizard to introspect a `.class` file, metadata is created around both complex and simple types. Complex types become data services while simple Java routines are converted into XQueries and placed in an XQuery function library (XFL). In Source View (see [Chapter 8, “Working with XQuery Source”](#)) a pragma is created that defines the function signature and relevant schema type for complex types such as Java classes and elements.

In the `RTLApp DataServices/Demo` directory there is a sample that can be used to illustrate Java function metadata import.

Also see in the Data Services Platform [Samples Tutorial Part I](#):

- Lesson 10: Updating Data Services Using Java

Also see in Data Services Platform [Samples Tutorial Part II](#):

1- Lesson 32: Accessing Data with Java Functions

Supported Java Function Types

Your Java file can contains two types of functions:

Types of Java Functions	Use in Data Services Platform
Functions processing primitive types or arrays of primitive types	Grouped into an XQuery Function Library file, callable by any data service in the same application.
Functions processing complex types or arrays of complex types	Grouped into a data services, using XMLBean Java-to-XML technology.

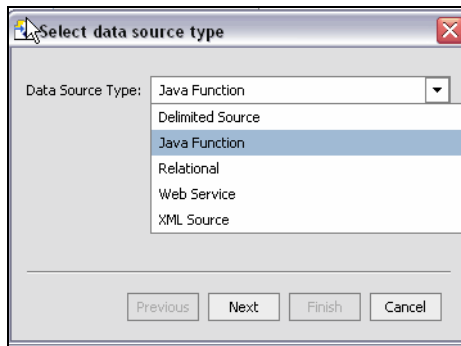
Before you can create metadata on a custom Java function you must create a Java class containing both schema and function information. A detailed example is described in [“Creating XMLBean Support for Java Functions”](#) on page 3-48.

Adding Java Function Metadata Using Import Wizard

Importing Java function metadata is similar to importing relational data source metadata (see [“Importing Relational Table and View Metadata” on page 3-8](#)). Here are the Java function-specific steps involved:

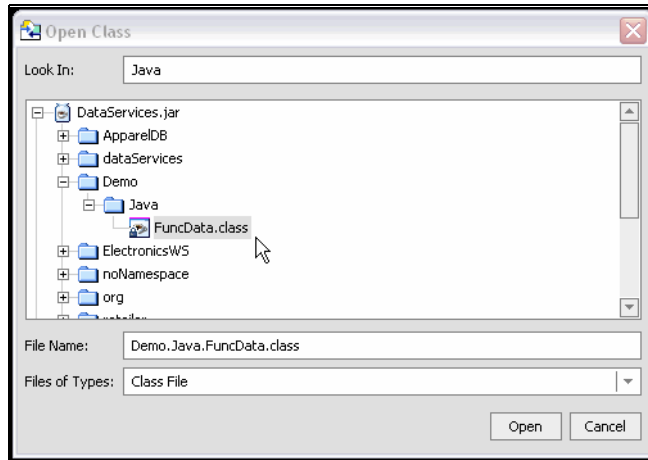
1. Select the DSP-based project in which you want to create your Java function metadata. (In the DataServices project of the RTLApp there is a special Demo folder containing XML, CSV, and Java data and schema samples.)
2. Build your project to validate its contents. A build will create a `.class` file from your `.java` function and place it in your application’s library.
3. Right-click on the Java folder and select Import Source Metadata from the pop-up menu.
4. From the available data sources in the Metadata Import wizard select Java Function (see [Figure 3-30](#)). Click Next.

Figure 3-30 Selecting a Java Function as the Data Source



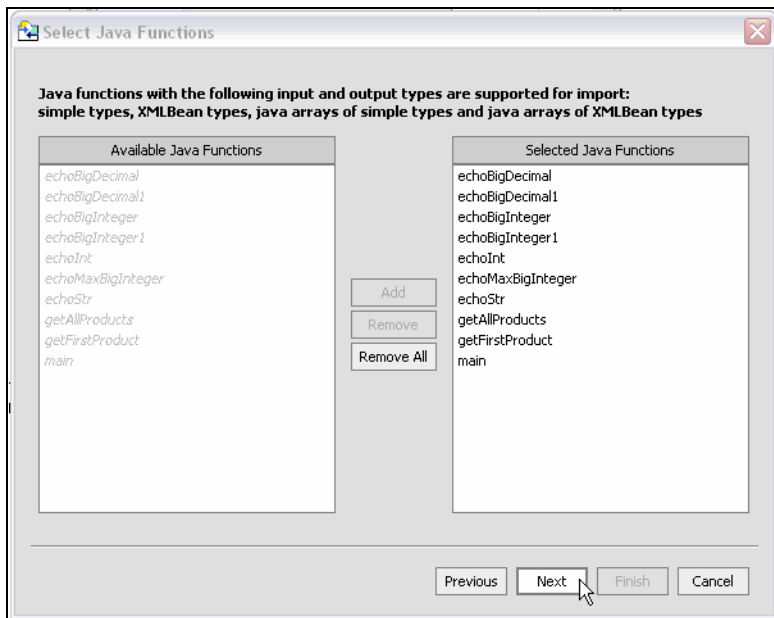
5. Your Java `.class` file must be in your BEA WebLogic application. You can browse to your file or enter a fully-qualified path name starting from the root directory of your DSP-based project.

Figure 3-31 Specifying a Java Class File for Metadata Import



6. Select Java functions for import.

Figure 3-32 Selecting Java Functions to Become Either Data Services or XFL Functions



7. Java functions with the following input and output types are supported for import:

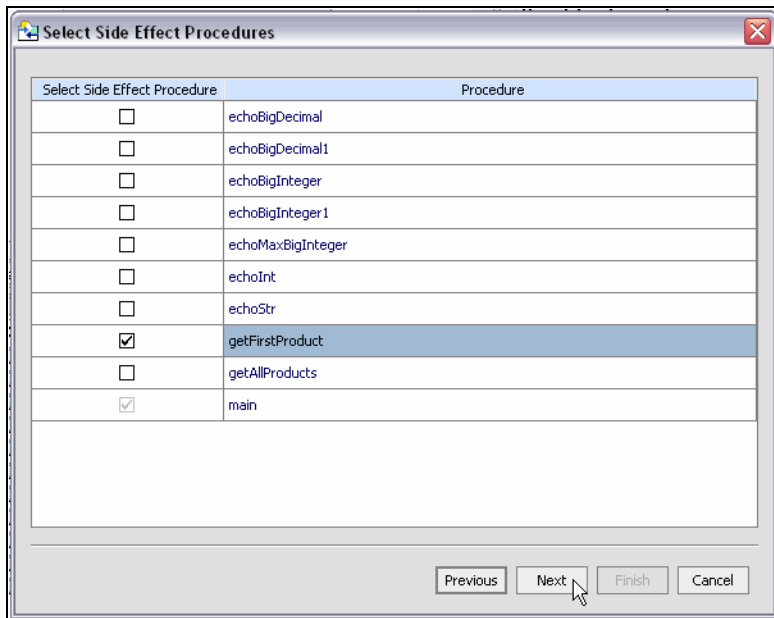
- Simply types
 - XMLBean types
 - Java arrays of simple types
 - Java arrays of XMLBean types
8. Identify which, if any, Java function-based data services should be identified as having side-effects.

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 noXML type; in fact they typically return nothing (or *void*). Instead the data service knows that the routine has side-effects, but those effects are not transparent to the service. DSP procedures can also be thought of as *side-effecting* functions.

Java functions are “side-effecting” from the perspective of the data service when they perform internal operations on data.

After you have identified the Java functions that you want to add to your project, you can also identify which, if any, of these should be treated as DSP procedures ([Figure 3-33](#)). In the case of `main()`, the Metadata Import wizard detects that it returns void so it is already marked as a procedure.

Figure 3-33 Marking Java Functions as DSP Procedures



Functions based around atomic (simple) types are collected in an identified XML function library (XFL) file.

Note: Side-effecting procedures must be associated with a data service that is from the same data source. In this case, the source is your Java file. In other words, in order to specify a Java function as a procedure, a function in the same file that returns a complex element must either be created at the same time or already exist in your project.

9. Click Next to verify the name and location of your new data service(s).

Figure 3-34 Java Function Imported Data Summary Screen

Summary

The following data service(s) will be created. Edit suggested name(s) as needed.

Function Name	Type	Data Service Name	Add to Existing Data S...
getAllProducts	PRODUCTS	PRODUCTS	<input type="checkbox"/>

The following side effect procedure(s) will be created. Edit suggested name(s) as needed.

Procedure Name	Data Service Name
getFirstProduct	
main	

These function(s) will be saved to the specified XML library file:

Function Name	Library Name	Add to Existing Library File
echoBigDecimal, echoBigDecimal...	library	<input type="checkbox"/>

Location

You can edit the proposed data service name either for clarity or to avoid conflicts with other existing or planned data services. All functions returning complex data types will be in the same data service. Click on the proposed data service name to change it.

Procedures must be associated with a data service that draws data from the same data source (Java file). In the sample shown in [Figure 3-34](#), the only available data service is PRODUCTS (or whatever name you choose).

If there are existing XFL files in your project you have the option of adding atomic functions to that library or creating a new library for them. All the Java file atomic functions are located in the same library.

10. Click Finish.

Creating XMLBean Support for Java Functions

Before you can import Java function metadata, you need to create a `.class` file that contains XMLBean classes based on global elements and compiled versions of your Java functions. To do this, you first create XMLBean classes based on a schema of your data. There are several ways to accomplish this. In the example in this section you create a WebLogic Workshop project of type Schema.

Generally speaking, the process involves:

- Creating a WebLogic Workshop project of type Schema. Schema projects (and applications) generate XMLBeans from schema files.
- Importing a schema (`.xsd` file) representing the shape of the global elements invoked by your function.
- Importing your custom Java function into your DSP-based project or Java project.
- Building your application to create a Java `.class` file, if under a DSP-based project, or you can add the JAR file from a Java project to the Library folder of your application.
- Creating metadata for your data service based on the `.class` file.
- Use the resulting data service or functions in your application.

Creating a Metadata-enriched Java Class: An Example

In the following example there are a number of custom functions in a `.java` file called `FuncData.java`. In the RTLApp this file can be found at:

```
ld:DataServices/Demo/Java/FuncData.java
```

Some functions in this file return primitive data types, while others return a complex element. The complex element representing the data to be introspected is in a schema file called `FuncData.xsd`.

File	Purpose
<code>FuncData.java</code>	Contains Java functions to be converted into data service query functions. Also contains a small data sample.
<code>FuncData.xsd</code>	Contains a schema for the complex element identified in <code>FuncData.java</code>

The schema file can be found at:

```
ld:DataServices/Demo/Java/schema/FuncData.xsd
```

To simplify the example a small data set is included in the `.java` file as a string.

The following steps will create a data service from the Java functions in `FuncData.java`:

1. Create a new DSP-based application called `CustomFunctions`.
2. Create a new project of type Schema in your application; name it `Schemas`.

3. Right-click on the newly created Schemas project and select the Import... option.
4. Browse to the RTLApp and select `FuncData.xsd` for import.

Importing a schema file into a schema project automatically starts the project build process.

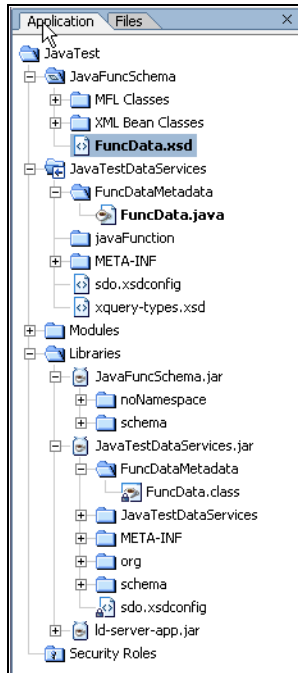
When successful, XMLBean classes are created for each function in your Java file and placed in a JAR file called `JavaFuncSchema.jar`

The JAR file is located in the Libraries section of your application.

5. Build your project.
6. In your DSP-based project (`customFunctionsDataServices`) create a folder called `JavaFuncMetadata`.
7. Right-click on the newly created `JavaFuncMetadata` folder and select the Import... option.
8. Browse to the `ld:DataServices/Demo/Java` folder in the RTLApp and select `FuncData.java` for import. Click Import.
9. Build your project.

The JAR file named for your DSP-based project is updated to include a `.class` file named `FuncData.class`; It is this file that can be introspected by the Metadata Import wizard. The file is located in a folder named `JavaFuncMetadata` in the Library section of your application.

Figure 3-35 Class File Generated Java Function XML Beans



10. Now you are ready to create metadata from your Java function. These steps are described in [“Adding Java Function Metadata Using Import Wizard” on page 3-44](#).

Inspecting the Java Source

The `.java` file used in this example contains both functions and data. More typically, your routine will access data through a data import function.

The first function in [Listing 3-1](#) simply retrieves the first element in an array of `PRODUCTS`. The second returns the entire array.

Listing 3-1 `JavaFunc.java` `getFirstPRODUCT()` and `getAllPRODUCTS()` Functions

```
public class JavaFunc {
    ...

    public static noNamespace.PRODUCTSDocument.PRODUCTS getFirstProduct() {
```

Obtaining Enterprise Metadata

```
noNamespace.PRODUCTSDocument.PRODUCTS products = null;
try{
    noNamespace.DbDocument dbDoc =
noNamespace.DbDocument.Factory.parse(testCustomer);
    products = dbDoc.getDb().getPRODUCTSArray(1);
    //return products;
}catch(Exception e){
    e.printStackTrace();
}
return products;
}

public static noNamespace.PRODUCTSDocument.PRODUCTS[] getAllProducts(){
    noNamespace.PRODUCTSDocument.PRODUCTS[] products = null;
    try{
        noNamespace.DbDocument dbDoc =
noNamespace.DbDocument.Factory.parse(testCustomer);
        products = dbDoc.getDb().getPRODUCTSArray();
        //return products;
    }catch(Exception e){
        e.printStackTrace();
    }
    return products;
}
}
```

The schema used to create XMLBeans is shown in [Listing 3-2](#). It simply models the structure of the complex element; it could have been obtained by first introspecting the data directly.

Listing 3-2 B-PTest.xsd Model Complex Element Parsed by Java Function

```
<xs:schema elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="db">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="PRODUCTS" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="AVERAGE_SERVICE_COST" type="xs:decimal"/>
</xs:schema>
```



```

<xs:element name="LIST_PRICE" type="xs:decimal"/>
<xs:element name="MANUFACTURER" type="xs:string"/>
<xs:element name="PRODUCTS">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="PRODUCT_NAME"/>
      <xs:element ref="MANUFACTURER"/>
      <xs:element ref="LIST_PRICE"/>
      <xs:element ref="PRODUCT_DESCRIPTION"/>
      <xs:element ref="AVERAGE_SERVICE_COST"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="PRODUCT_DESCRIPTION" type="xs:string"/>
<xs:element name="PRODUCT_NAME" type="xs:string"/>
</xs:schema>

```

Java functions require that an element returned (as specified in the return signature) come from a valid XML document. A valid XML document has a single root element with zero or more children, and its content matches the schema referred.

Listing 3-3 Approach When Data is Retrieved Through a Document

```

public static noNamespace.PRODUCTSDocument.PRODUCTS getNextProduct() {
    // create the dbDocument (the root)
    noNamespace.DbDocument dbDoc =
noNamespace.DbDocument.Factory.newInstance();
    // the db element from it
    noNamespace.DbDocument.Db db = dbDoc.addNewDb();
    // get the PRODUCTS element
    PRODUCTS product = db.addNewPRODUCTS();
    //.. create the children
    product.setPRODUCTNAME("productName");
    product.setMANUFACTURER("Manufacturer");
    product.setLISTPRICE(BigDecimal.valueOf((long)12.22));
    product.setPRODUCTDESCRIPTION("Product Description");
    product.setAVERAGESERVICECOST(BigDecimal.valueOf((long)122.22));
}

```

```

// .. update children of db
db.setPRODUCTSArray(0,product);

// .. update the document with db
dbDoc.setDb(db);

//.. now dbDoc is a valid document with db and is children.
// we are interested in PRODUCTS which is a child of db.
// Hence always create a valid document before processing the
    children.
// Just creating the child element and returning it, is not
// enough, since it does not mean the document is valid.
// The child needs to come from a valid document, which is created
// for the global element only.

    return  dbDoc.getDb().getPRODUCTSArray(0);

}

```

How Metadata for Java Functions Is Created

In DSP, user-defined functions are typically Java classes. The following are supported:

- Java primitive types and single-dimension arrays.
- Global elements, global complex types, and global arrays through XMLBean classes

In order to support this functionality, the Metadata Import wizard supports marshalling and unmarshalling so that token iterators in Java are converted to XML and vice-versa.

Functions you create should be defined as static Java functions. The Java method name when used in an XQuery will be the XQuery function name qualified with a namespace.

[Table 3-36](#) shows the casting algorithms for simple Java types, schema types and XQuery types.

Table 3-36 Simple Java Types and XQuery Counterparts

Java Simple or Defined Type	Schema Type
boolean	xs:boolean
byte	xs:byte
char	xs:char

Java Simple or Defined Type	Schema Type
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 also consume variables of XMLBean type that are generated by processing a schema via XMLBeans. The classes generated by XMLBeans can be referred in a Java function as parameters or return types.

The elements or types referred to in the schema should be global elements because these are the only types in XMLBeans that have static parse methods defined.

The next section provides additional code samples that illustrate how Java functions are used by the Metadata Import wizard to create data services.

Technical Details, with Additional Example Code

In order to create data services or members of an XQuery function library, you would first start with a Java function.

Processing a Function Returning an Array of Java Primitives

As an example, the Java function `getListGivenMixed()` can be defined as:

```
public static float[] getListGivenMixed(float[] fpList, int size) {
    int listLen = ((fpList.length > size) ? size : fpList.length);
    float fpListop = new float[listLen];
    for (int i =0; i < listLen; i++)
        fpListop[i]=fpList[i];
    return fpListop;
}
```

After the function is processed through the wizard the following metadata information is created:

```
xquery version "1.0" encoding "WINDOWS-1252";

(:::pragma xfl <x:xfl xmlns:x="urn:annotations.ld.bea.com">
<creationDate>2005-06-01T14:25:50</creationDate>
<javaFunction class="DocTest"/>
</x:xfl>::)

declare namespace fl = "lib:testdoc/library";

(:::pragma function <f:function xmlns:f="urn:annotations.ld.bea.com"
nativeName="getListGivenMixed">
  <params>
    <param nativeType="[F"/>
    <param nativeType="int"/>
  </params>
</f:function>::)

declare function fl:getListGivenMixed($x1 as xsd:float*, $x2 as xsd:int) as
xsd:float* external;
```

Here is the corresponding XQuery for executing the above function:

```
declare namespace f1 = "ld:javaFunc/float";
let $y := (2.0, 4.0, 6.0, 8.0, 10.0)
let $x := f1:getListGivenMixed($y, 2)
return $x
```

Processing complex types represented via XMLBeans

Consider that you have a schema called Customer (customer.xsd), as shown below:

```
<?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>
```

If you want to generate a list conforming to the CUSTOMER element you could process the schema via XMLBeans and obtain `xml.cust.beaBB10000.CUSTOMERDocument.CUSTOMER`. Now you can use the CUSTOMER element as shown:

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

Then the metadata information produced by the 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:

Obtaining Enterprise Metadata

```
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 $z
return
    f1:getCustomerListGivenCustomerList($z)
```

Restrictions on Java Functions

The following restrictions apply to Java functions:

- Function overloading is based on the number of arguments; not on the types of the parameter.
- 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.

Importing Delimited File Metadata

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 in 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 nearly universal.

Although the separator field is often a comma, the Metadata Import wizard supports any ASCII character as a separator, as well as fixed-length fields.

Note: Delimited files in a single server must share the same encoding format. This encoding can be specified through the system property `ld.csv.encoding` and set through the JVM command-line directly or via a script such as `startWebLogic.cmd` (Windows) or `startWebLogic.sh` (UNIX).

Here is the format for this command:

```
-Dld.csv.encoding=<encoding format>
```

If no format is specified through `ld.csv.encoding`, then the format specified in the `file.encoding` system property is used.

In the RTLApp DataServices/Demo directory there is a sample that can be used to illustrate delimited file metadata import.

Also see in the Data Services Platform [Samples Tutorial Part II](#):

- Lesson 34: Accessing Data in Flat Files

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 Metadata 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.

Note: The generated schema takes the name of the source file.

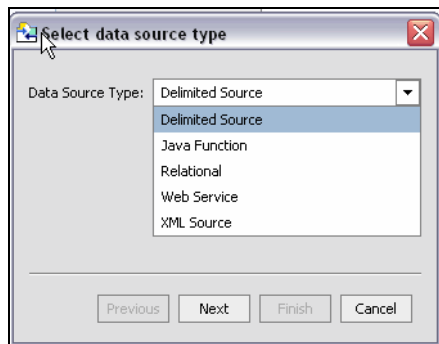
- **Providing a schema name only.** This option is typically used when the source file is dynamic; for example, when data is streamed.
- **Providing both a schema and a document name.** Providing a schema gives you the ability to more accurately type information in the columns of a delimited document.

Using the Metadata Import Wizard on Delimited Files

Importing XML file information is similar to importing a relational data source metadata (see “Importing Relational Table and View Metadata” on page 3-8). Here are the steps that are involved:

1. Select the project in which you want to create your delimited file metadata. For example, if you have a project called myProject right-click on the project name and select Import Source Metadata from the pop-up menu.
2. From the available data sources in the Metadata Import wizard select Delimited Source as the data type (see Figure 3-37).

Figure 3-37 Selecting a Delimited Source from the Import Metadata Wizard



3. You can supply either a schema name, a source file name, or both. Through the wizard you can browse to a file located in your project. You can also import data from any CSV 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:

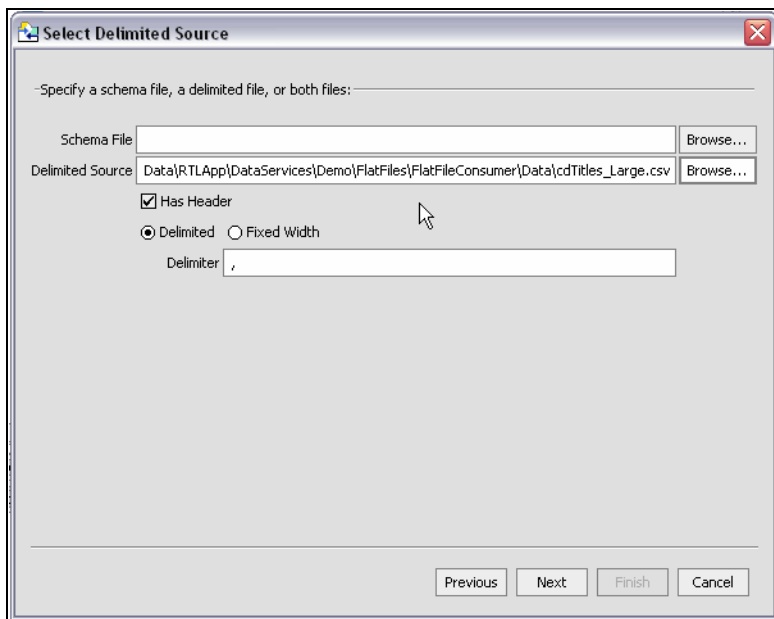
```
file:///<c:/home>/Orders.csv
```

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

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

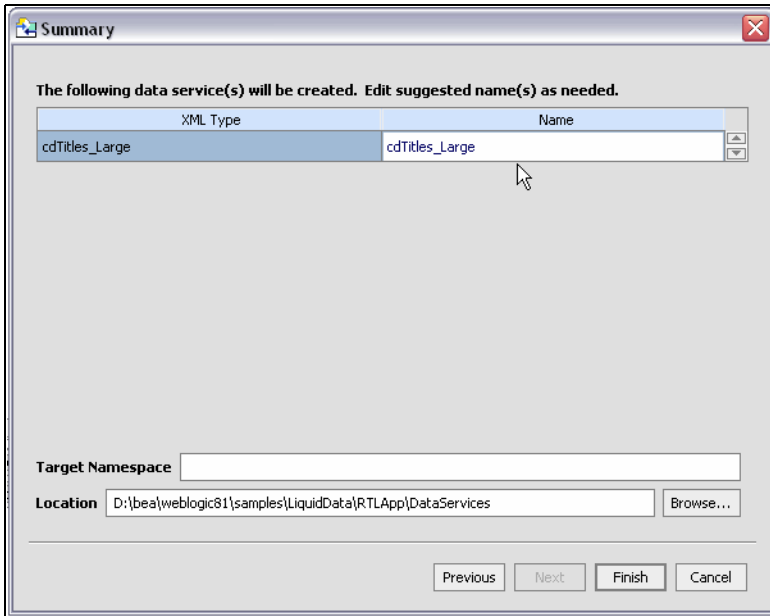
4. Select additional import options:
 - **Header.** Indicates whether the delimited file contains header data. Header data is located in the first row of the spreadsheet. If you check this option, the first row will not be treated as data.
 - **Delimited or Fixed Width.** Data in your file is either 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 (,).

Figure 3-38 Specifying Import Delimited Metadata Characteristics



5. Once you have selected a document and, optionally, a schema, click Next to verify the location and unique location/name of your new data service.

Figure 3-39 Delimited Document Imported Data Summary Screen



You can edit the data service name either to clarify the name or to avoid conflicts with other existing or planned data services. Any name conflicts are displayed in red. To change the name, double click on the name of the data service to activate the line editor.

6. Click Finish. A data service (.ds file) will be created with your schema as its XML type.

Note: When importing CSV-type data there are several things to keep in mind:

- 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 not equal. Some may need special escape sequences before spreadsheet data can be accessed at run-time.

Importing XML File Metadata

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

Importing XML file information is similar to importing a relational data source metadata (see [“Importing Relational Table and View Metadata” on page 3-8](#)).

The Metadata Import wizard allows you to browse for an XML file anywhere in your application. 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 URI:

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

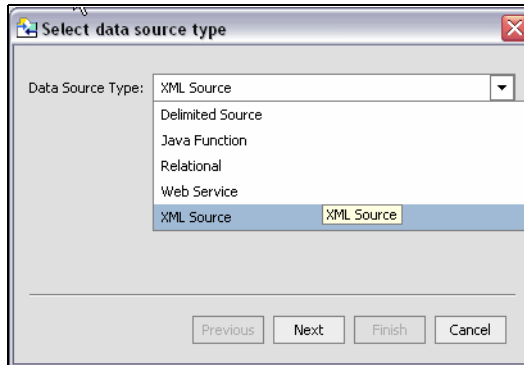
XML File Import Sample

In the RTLApp DataServices/Demo directory there is a sample that can be used to illustrate XML file metadata import.

Here are the steps involved:

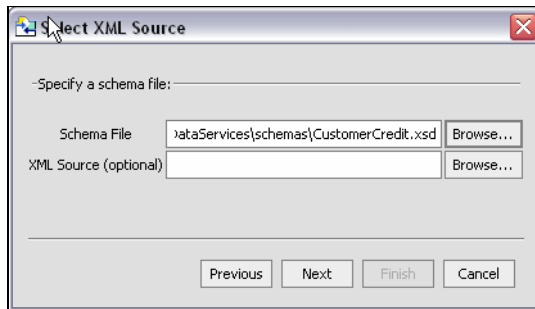
1. Select your DSP-based project in which you want to create your XML file metadata. For example, if you have a project called myProject, right-click on the project name and select Import Metadata... from the pop-up menu.
2. From the available data sources in the Metadata Import wizard select XML Source.

Figure 3-40 Selecting an XML File from the Import Metadata Wizard



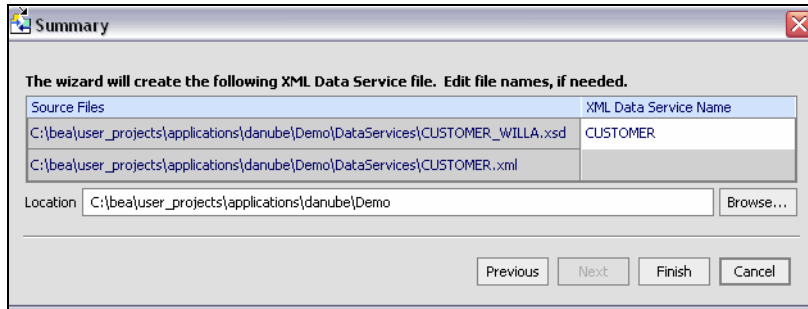
3. In order to access XML data you must first identify a schema; the schema must be located in your application.

Figure 3-41 Specify an XML File Schema for XML Metadata Import



4. Optionally specify an XML file. If the XML file exists in your DSP-based project you can simply browse to it. More likely your document is available as a URI, in which case you want to leave the XML file field empty and supply a URI at runtime.
5. Once you have selected a schema and optional document name, click Next to verify that the name of your new data service is unique to your application.

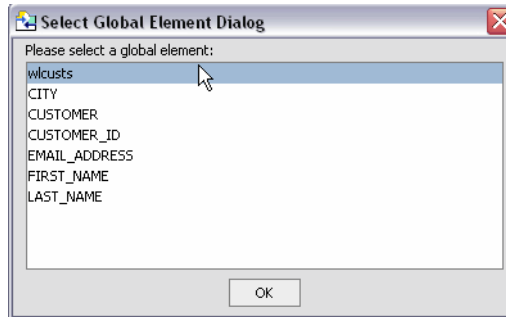
Figure 3-42 XML File Imported Data Summary Screen



You can edit the data service name either to clarify the name or to avoid conflicts with other existing or planned data services. Conflicts are shown in red. Simply click on the name of the data service to change its name. Then click Next.

- Next select a global element in your schema (Figure 3-43). Click Ok.

Figure 3-43 A Selecting a Global Element When Importing XML Metadata



- Complete the importation of your procedures by reviewing and accepting items in the Summary screen (see step 4. in "Importing Relational Table and View Metadata" for details).

Also see in the Data Services Platform [Samples Tutorial Part I:](#)

- Lesson 11: Filtering, Sorting, and Truncating XML Data

Also see in the Data Services Platform [Samples Tutorial Part II:](#)

- Lesson 33: Accessing Data in XML Files

Testing the Metadata 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 (various methods of accessing data service functions are described in detail in the [Client Application Developer's Guide](#)).

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 needed.

You can access files 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:

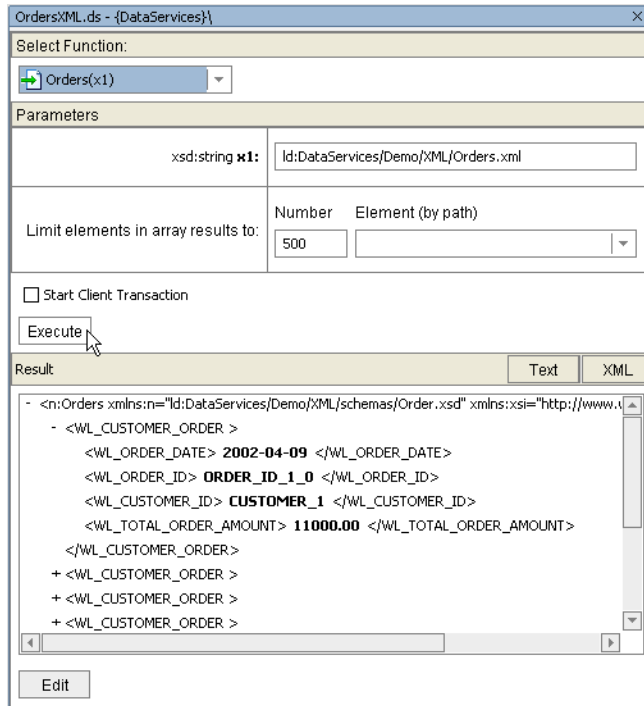
```
file:///c:/Orders.xml
```

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

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

[Figure 3-44](#) shows how the XML source file is referenced.

Figure 3-44 Specifying an XML Source URI in Test View



Updating Data Source 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.

You can use the Update Source Metadata right-click menu option to identify differences between your source metadata files and the structure of the source data including:

- Object added
- Object deleted

- Object modified
- Source Unavailable

In the case of Source Unavailable, the issue likely relates to connectivity or permissions. In the case of the other types of reports, you can determine when and if to update data source metadata to conform with the underlying data sources.

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

Considerations When Updating Source Metadata

Source metadata should be updated with care since the operation can have both direct and indirect consequences. For example, if you have added a relationship between two physical data services, updating your source metadata can potentially 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.

In many cases the Update Source Metadata Wizard can automatically merge user changes with the updated metadata. See [“Using the Update Source Metadata Wizard,”](#) for details.

Direct and Indirect Effects

Direct effects apply to physical data services. Indirect effects occur to logical data services, since such services are themselves ultimately based — at least in part — on physical data service. For example, if you have created a new relationship between a physical and a logical data service, 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.

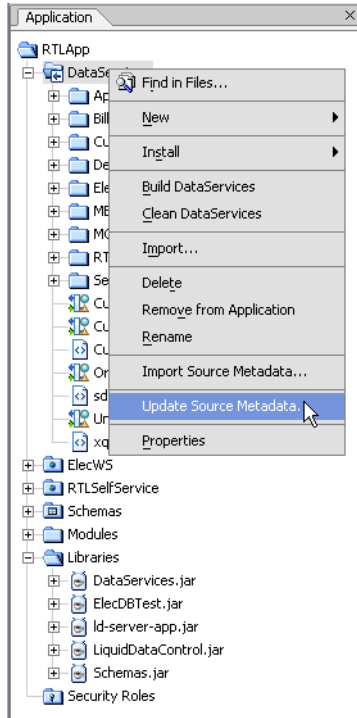
Thus updating source metadata should be done carefully. Several safeguards are in place to protect your development effort while preserving your ability to keep your metadata up-to-date. See [“Archival of Source Metadata” on page 3-72](#) for information of how your current metadata is preserved as part of the source update.

Using the Update Source Metadata Wizard

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

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

Figure 3-45 Updating Source Metadata for Several Data Services

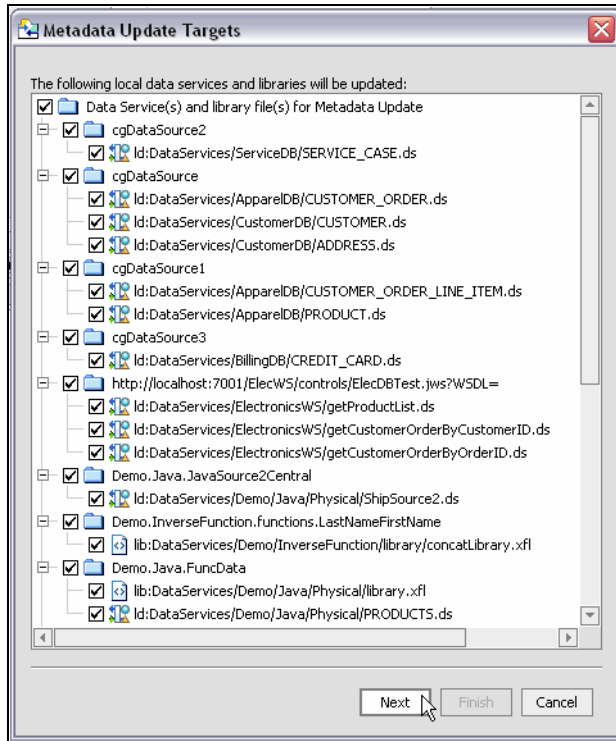


You can verify that your data structure is up-to-date by performing a metadata update on one or multiple physical data services in your DSP-based project. For example, in [Figure 3-45](#) all the physical data services in the project will be updated.

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 or XFL file listed in the dialog using the checkbox to the left of the name ([Figure 3-46](#)).

Figure 3-46 Data Services Metadata to be Updated



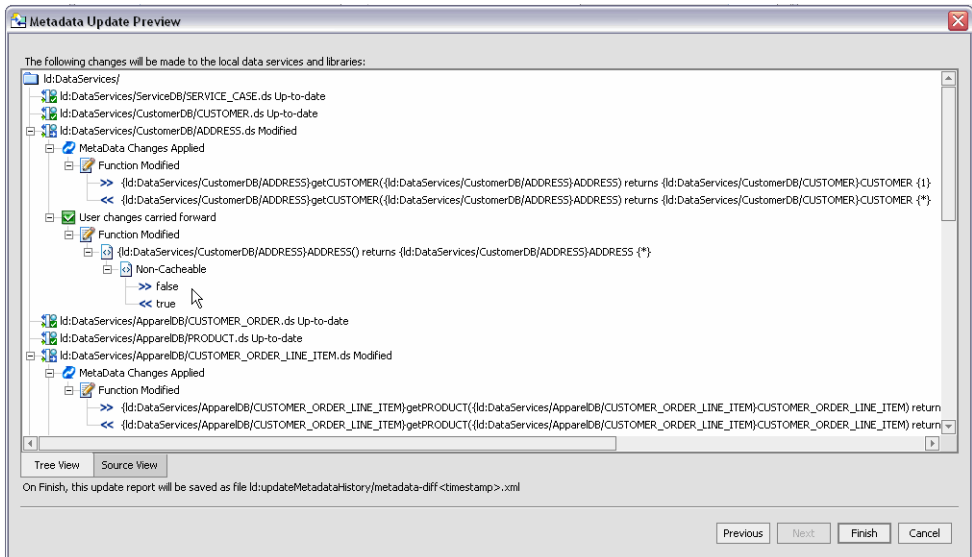
Metadata Update Analysis

Next, an analysis is performed on your metadata by the wizard. The following types of synchronization mismatches are identified:

- Structural differences between the data source and the metadata. These will be resolved in favor of the data source and includes generated schema (data services XML type) based on the physical source.
- Differences that will be overwritten in favor of the values coming from the data source.
- Additions and modifications to the physical data service that will automatically be merged back into the updated metadata. This includes data service functions.

An update preview screen report (Figure 3-47) is prepared describing these differences both generally and for field-level data.

Figure 3-47 Metadata Update Plan for RTLApp's DataServices Project



The Metadata Update Preview screen identifies:

- **Metadata changes to be applied.** These changes are necessary in order for the physical data service to remain a valid representation of a underlying physical data source.
- **User changes dropped.** These are changes which cannot be merged into the updated metadata.
- **User changes carried forward.** These are changes will can be merged into the update metadata.

Icons differentiate elements as to be added, removed, or changed. [Table 3-48](#) describes the update source metadata message types and color legends.

Table 3-48 Source Metadata Update Targets and Color Legend

Category	Color	Description
Data source field added	Green	A data source field has been added since the last metadata update.
Data service schema (XML type) modified	Black	A change has been made in a schema that was derived from a data source.

Category	Color	Description
Data source field deleted	Red	A field used by your metadata is no longer appearing in source.
Field modified	Blue	A field in your metadata does not exactly match the data source field.
Function modified	Blue	A function in your metadata does not exactly match the data source function.

Synchronization Mismatches

Under some circumstances the Update Source Metadata wizard flags data service artifacts as changed locally when, in fact, no change was made.

For example, in the case of importing a Web service operation, a schema that is dependent (or referenced) by another schema will be assigned an internally-generated filename. If a second imported Web service operation in your project references the same dependent schema, upon synchronization the wizard may note that the name of the imported secondary schema file has changed. Simply proceed with synchronization; the old second-level schema will automatically be removed.

Archival of Source Metadata

When you update source metadata two files are created and placed in a special directory in your application:

- A copy of the update report in the form:

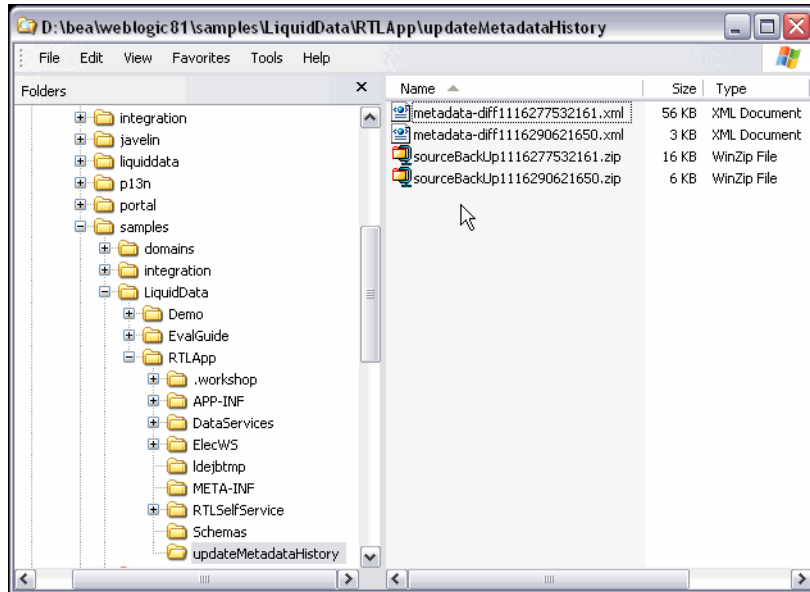
```
ld:/updateMetadataHistory/metadatadiff<timestamp>.xml
```

- The XQuery source data services and other artifacts that were overwritten by the update operation are saved in the form of:

```
ld:/updateMetadataHistory/sourceBackUp<timestamp>.zip
```

An update metadata source operations assigns the same timestamp to both generated files.

Figure 3-49 UpdateMetadataHistory Directory Sample Content



Working with a particular update operations report and source, you can often quickly restore relationships and other changes that were made to your metadata while being assured that your metadata is up-to-date.

Obtaining Enterprise Metadata

Designing Data Services

A data service gives you access to a structured view of a unit of information in the enterprise such as a customer, sales order, product, or service.

Collectively, a set of data services comprise the data integration layer in an IT environment. For additional information on data services see [“Unifying Information with Data Services”](#) in the BEA Aqualogic Data Services Platform *Concepts Guide*.

Design View presents the data service as a “integrated chip” or schematic representation ([Figure 4-1](#)) of all the query functions, underlying data sources, navigational relationships, and transformation logic needed to support returning results in a particular arrangement, the *return type*. For details see [Chapter 2, “Data Services Platform Projects and Components.”](#)

A data service exists in a DSP-based project as a single XQuery file containing query functions and metadata support. For details see [Chapter 8, “Working with XQuery Source”](#) and the Data Services Platform *XQuery Developer’s Guide*.

For information on setting security and caching policies for functions and elements see [“Securing Data Services Platform Resources”](#) in the *Administration Guide*.

The following major topics are included in this chapter:

- [Data Services in the Enterprise](#)
- [Data Service Design View Components](#)
- [Creating a Data Service](#)
- [Managing Your Data Service](#)

Also see in the Data Services Platform [Samples Tutorial Part I](#):

- Lesson 2: Creating a Physical Data Service
- Lesson 3: Creating a Logical Data Service
- Lesson 4: Integrating Data from Multiple Data Services

Data Services in the Enterprise

In modern enterprises there are increasingly two “data worlds”: the traditional relational world of tables, columns, views, and stored procedures and the world of Web services and other forms of data that is accessed through the desktop or through various Web interfaces.

Increasingly, the cost of accessing and updating data across systems with fundamentally different architectures and purposes can rival the cost of setting up the services themselves.

Comparing Data Services with Web Services

A data service is similar to a conventional Web service in the following respects:

- It consists of public functions.
- The functions that access services are modular, reusable, and extensible.
- Implementation details are hidden.

Of course a conventional Web service does not have a core XML data type that allows for easy manipulation of the shape of the return data. Another minor difference is that data services can access private functions contained in XQuery library files (`.xfl` files).

In concrete terms, a data service is a file that contains XML Query (XQuery) instructions for retrieving, aggregating, and transforming data.

Physical and Logical Data Services

There are two types of data services: *physical* and *logical*. Physical data services comprise both relational and service data. Logical data services are consumers of physical or other logical data services. The data access layer of the enterprise includes both logical and physical data services.

An important benefit of this approach is that in the case of a virtual data access layer such as the Data Services Platform provides there is no transfer or storage of data — other than for application-controlled caching. Instead data services simply expose interface calls to *read functions* that dynamically retrieve data from data sources. The retrieved data is then arranged based on the data service's XML type. Update logic is associated with each data service. In the case of relational data the update logic is automatic; otherwise custom update functions can be developed (see [Enabling SDO Data Source Updates](#) in the *Application Developer's Guide*).

Note: Logical data services are built upon physical data services which in turn represent an underlying physical data source. Physical data service are created by importing metadata on a physical source and updated through synchronization. The schema file or XML type generated by this process should never be modified either in DSP or externally. Doing so risks invalidating your data service and dependent logical data services. (If you need to modify a data service based on a single source it is a simple matter to create a logical data service based on that one source.)

Data Service Functions

Data services should be designed so as to present client applications with a sensible, uniform data access layer for obtaining and updating data.

The data service interface consists of several types of functions:

- **Read functions.** Return data in the form of the data service's XML type. Read functions can be developed either in the XQuery Editor or through Source View. Data services are ideal for encapsulating any number of specific functions with roles such as “Get all customers with pending orders”, “Find customer number ____”, and so forth.
- **Procedures.** Procedures are also known as *side-effecting functions*. They refer to external functions that typically return void. Procedures are identified during the metadata import process and are only associated with physical data services. Sources for procedures include Java routines, Web services, and relational stored procedures.
- **Navigation functions.** Return data in the form of a related data service's XML type using an instance of the current data service as a parameter. See [“Understanding Navigation Functions” on page 4-11](#).
- **Private functions.** In addition to public functions, a data service or a XQuery function library (.xfl file) can contain private functions. Private functions can be accessed only by other functions within the data service or XFL. Such functions generally contain common processing logic, that is, operations for use in more than one function in the data service. (For functions designed to be shared across data services, see [“XQuery Function Libraries” on page 2-17](#).)

In Source View you will notice that private functions are so identified in the pragma statement above the function.

- **A submit function.** Allows clients to persist changes (update) to underlying (physical) data.

Note: The single submit() function can be found in Source View. It is not represented in the Design View of the data service.

Data Service Design View Components

Design View provides a means of visualizing the entire data service (see [Figure 4-1](#)). Each data service appears in WebLogic Workshop optionally bounded by panes that describe the application components, properties of selected Design View properties, and so forth. For details on DSP-based project components see [Chapter 2, “Data Services Platform Projects and Components.”](#)

At the heart of each data service is its XML type. The XML type describes the shape of the document that will be returned when read functions are called either from this or a related navigation function. (For additional information see [“XML Types and Return Types” on page 4-7.](#))

Design View displays:

- Read functions, also called *query functions*, that return data according to the XML type associated with the service. Navigation functions to related data services; these return data in the shape of their native XML type.
- Immediate underlying physical and logical data services and their read functions.

Figure 4-1 Major Visual Components of a Data Service

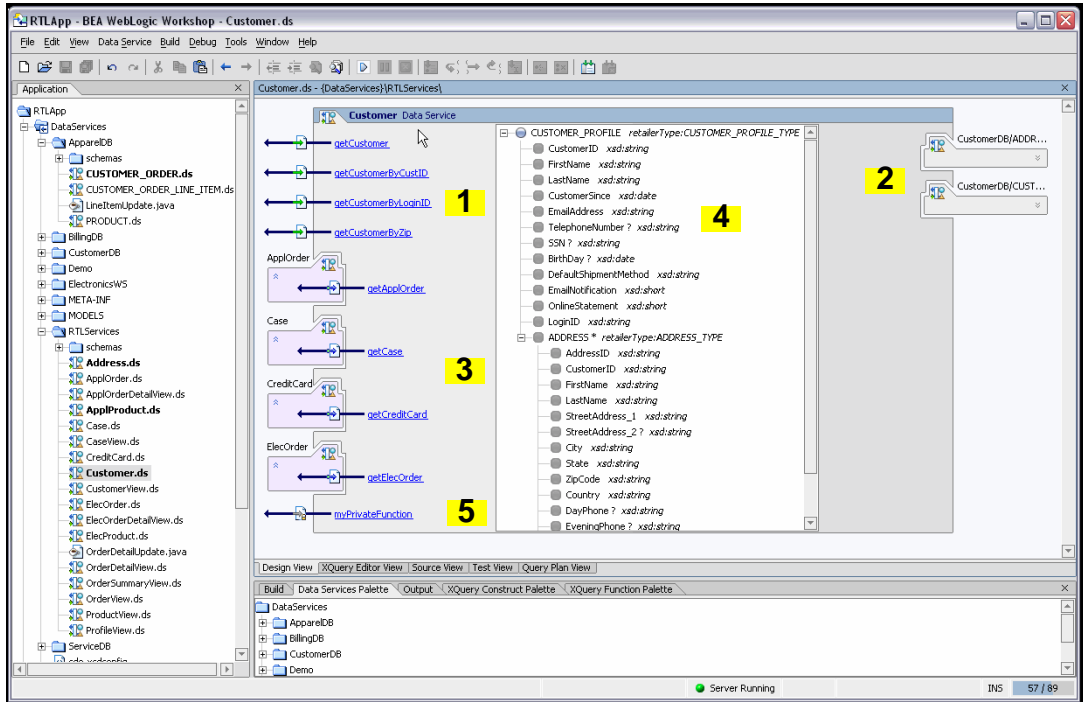




Table 4-2 details the functional components of the data service shown in Figure 4-1.

Table 4-2 Graphical Components of a Data Service

Key	Component	Purpose
1	Query functions and procedures	<p>Read function and DSP procedures are typically developed through the XQuery Editor.</p> <p>Read functions provide an API for the data service. In Figure 4-1 the <code>getCustomer()</code> function accepts a <code>custID</code> and returns data in the shape of the customer XML type. (See “Modifying a Return Type” on page 6-47).</p> <p>Procedures refer to functions which have side effects; often such functions return void.</p>
2	Base data services	<p>The data services that are used as immediate building blocks for the current data service are shown. Click on its chevron symbol  inside the underlying data service representation to view its functions that are used by the current data service. If you click on the function name itself the data service will open. (Use the Back button to return to the original data service.)</p> <p>Note: Underlying data services are only displayed to one level. Use the Metadata Browser to identify all underlying data services and dependencies (see “Viewing Metadata” in the Data Services Platform <i>Administration Guide</i>).</p>
3	Navigation functions	<p>Relationships that are both inferred (relational) or created are shown. Navigation functions return data in the shape of their native type. Clicking on the chevron symbol  inside your relationship representation, you will see the navigation functions that are defined for that relationship.</p> <p>If you click on the function name your view will switch to the XQuery Editor.</p> <p>Relationships can be created through the Data Services Platform modeler (see Chapter 5, “Modeling Data Services”) or directly in your data service using the relationship wizard (see “Using the Relationship Wizard to Create Navigation Functions” on page 4-13).</p>
4	XML type	<p>The XML type is represented by an editable XML schema. The return type of read functions shown in the XQuery Editor (see Chapter 6, “Working with the XQuery Editor”) should match the data service XML type.</p>
5	Private functions	<p>Private functions are only available to other functions in the data service. They appear in Design View between read functions and navigation functions.</p>

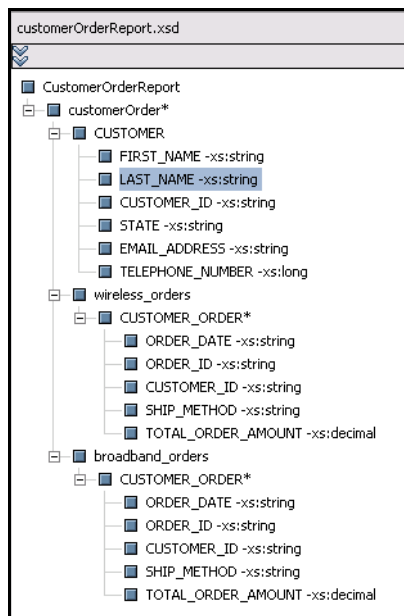
Note: Multiple data services can depend on a single XML type. In such situations it is advantageous to design such data services as a group, so that they always should return the same XML type.

XML Types and Return Types

A key product of DSP-based projects are data service query functions and return types, sometimes called *target schemas*. XML schemas are used to represent in hierarchical form physical and logical data and the shape of documents returned from DSP queries.

Return types can be thought of as the backbone of both data services and data models. Programmatically, return types are the “r” in for-let-where-return (flwr) queries.

Figure 4-3 Sample Return Type from the RTLApp



Return types have the following main purposes:

- Provide a template for the mapping of data from a variety of data sources.
- Help determine the arrangement of the XML document generated by the XQuery.

For more information on specifying the XML type in a data service see [“Associating an XML Type” on page 4-23](#).

Where XML Types are Used

The Data Services Platform modeler, data services, XQuery Editor, and Metadata Browser use XML type representations as follows:

- **Modeler.** A DSP Model shows the relationships and cardinality between data services, as well as read query functions. For details see [Chapter 5, “Modeling Data Services.”](#)
- **Data Service.** A data service generally contains an editable return type.
- **Data Sources.** Hierarchical-structured XML types represent both relational and non-relational data. For details see [Chapter 3, “Obtaining Enterprise Metadata.”](#)
- **XQuery Editor.** The XQuery Editor uses physical and logical data source representations and transformational functions to develop queries that are mapped to a return type. For details see [Chapter 6, “Working with the XQuery Editor.”](#)
- **Metadata Browser.** The Metadata Browser can display the return type associated with a data service. For details see [“Viewing Metadata”](#) in the Data Services Platform *Administration Guide*).

For the versions of the XQuery and XML specifications implemented in DSP see the DSP [XQuery Developer’s Guide](#).

Note: Data services supporting ADO.NET have additional, specific XML type requirements. For details see “Supporting ADO.NET Clients” in the [Client Application Developer’s Guide](#).

Where Return Types are Used

Return types describes the structure or shape of data that a query produces when it is run. A return type can be thought of as an object of XML type.

Note: In order to maintain the integrity of DSP 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 New Data Service and Data Service Function”](#) on page 6-7.

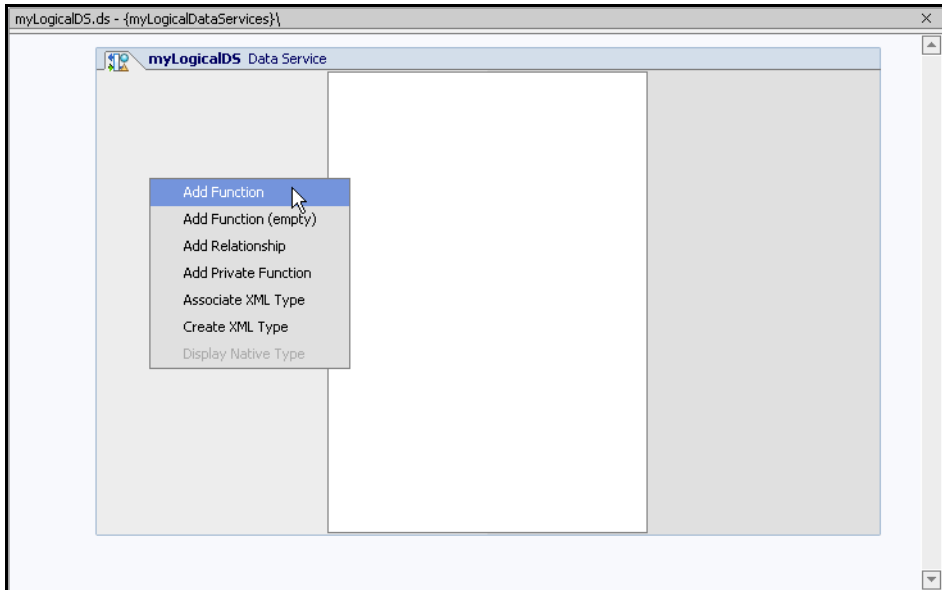
Creating a Data Service

You can create a data service in several ways:

- Through the DSP Metadata Import wizard, which automatically generates physical data services from available data sources. See [Chapter 3, “Obtaining Enterprise Metadata.”](#)

- By selecting a DSP-based project and then choosing File → New → Data Service. Alternatively, right click on the project folder and choose New → Data Service.
- By selecting Create Data Service from a data model and then opening the newly-created data service. See [Chapter 5, “Modeling Data Services.”](#)

Figure 4-4 Adding a Function to a Data Service



Data services always reside in the current DSP-based project. Once created, you can use the Data Service menu (or right-click) to develop your data service. [Table 4-5](#) lists available right-click options and their usage.

Table 4-5 Data Service Menu Options

Command	Usage
Add Function	Adds a function to your data service. After entering a name for the function, clicking on the name will open the XQuery Editor.
Add Function (empty)	Adds an empty function to your data service. An empty function will not initially contain a representation of the XML type even if a type is associated with your data service. In such cases, the schema “mark-up” can be added manually. This is particularly useful in cases where your XML type contains a large number of elements, many of which will not be used in the query functions planned for the data service.
Add Relationship	Creates a relationship to another data service. A file browser allows you to enter the name of the data service which you want to relate to your current data service. This, in turn, will bring up the Relationship wizard, where you can define the navigation functions that will relate the two services.
Add Private Function	Adds a private function to your data service. After entering a name for the function, clicking on the name will open the XQuery Editor.
Associate XML Type	Associates your data service with an XML type. You can choose the type (.xsd schema) from anywhere in your application. If your data service currently has an associated XML type, it will be replaced.
Create XML Type	Allows you to create an XML type using the built-in schema editor. Note: Once your data service is associated with a XML type, this option becomes unavailable.
Display XML Type / Display Native Type	For physical data services you can display either the element’s XML type (example: xs:int) or its native type (example: CUSTOMERID INTEGER(10)).

Subsequent sections describe each of these commands in detail.

Adding a Function to Your Data Service

Read functions can be accessed by any calling application with the appropriate security credentials. When adding a read function to your data service, you can accept the default function name or edit it

directly. Then, when you click on the name of your new function, you will be placed in the XQuery Editor. See [Chapter 6, “Working with the XQuery Editor”](#).

Note: It is important that function names in any given data service be unique even when their arity (number of parameters) does not match. This is because JDBC is not able to differentiate between functions of the same name.

Adding a Procedure to Your Data Service

Data service procedures or side-effecting functions enable you to invoke external routines that do not necessarily return data. A common scenario would be to use a procedure to invoke a Web service which in turn updates data. Another use of a procedure would be to invoke a relational stored procedure which in turn performs a database operation. The only thing returned in such a case might be a “success” message and that would only happen if the stored procedure was designed to report its status and the calling procedure was set up to handle such returned data.

Procedures are added to physical data services only, as part of the metadata import process. For details see [“Identifying DSP Procedures” on page 3-4](#).

Adding a Private Function to Your Data Service

A private function is similar to a read function, but it is only available to other functions in your data service. You can change a private function to a read function through the Property Editor or by editing the Source View pragma.

Adding a Relationship to Your Data Service

Relationships allow you to call out to another data service using an instance of your data service as a parameter. Data is returned in the shape of the related service. In this way you can populate your data services with a set of functions.

Understanding Navigation Functions

Two data services can be related by one or more relationships.

For example, CUSTOMER and ORDER might be related by a CUSTOMER-ORDER relationship that has three navigation functions in all:

```
cst:getAllOrders (CUSTOMER) →ORDER*
cst:getOpenOrders (CUSTOMER) →ORDER*
ord:getCustomer (ORDER) →CUSTOMER
```

The first two functions are different ways of navigating the CUSTOMER-ORDER relationship from a customer to all or some of their orders. The third function is a way to navigate from an ORDER to the associated CUSTOMER.

In the most common case, a relationship will result in the availability of two navigation functions, one for moving through the relationship in one direction and one for moving in the other direction.

In the less common case of a unidirectional relationship, there will be only one navigation function.

Effect of Using a Navigation Function to Return Data

In a data service the functional difference between a read function and a navigation function is the shape of the returned data. Here is a simple example:

In a read function if you have an OpenOrders data service with an XML type of:

```
<openOrders>
  <custID>
  <first_name>
  <last_name>
  <orderID>
  ..
</openOrders>
```

and pass it a customer ID such as 101 and an order ID such as LRP-111. The query result appears as:

```
<customerInfo>
  <custID>101</custID>
  <first_name>Jane</first_name>
  <last_name>Smith</last_name>
  <orderID>Smith</orderID>
  ..
</customerInfo>
```

However, if your data service has a navigation function associated with a table called TrackOrders, the query parameter can remain the same but data will be returned in the shape of the TrackOrders type, which looks like this:

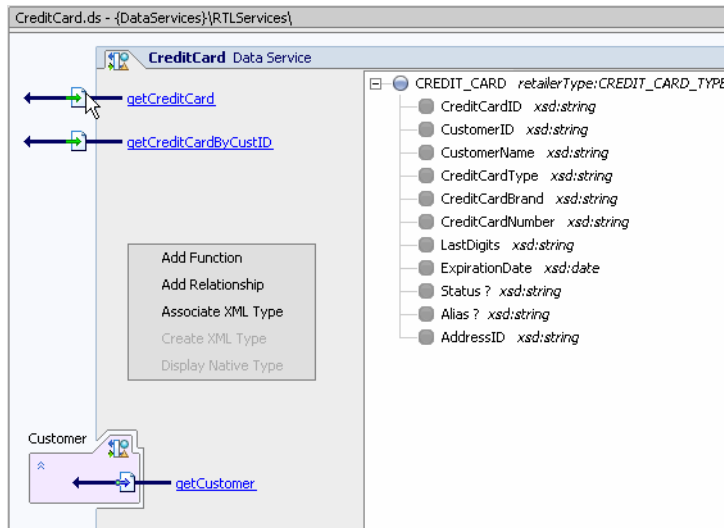
```
<TrackOrders>
  <custID>
  <first_name>
  <last_name>
  <orderID>
  <ship_date>
  <weight>
  <delivery_date>
  ...
</TrackOrders>
```

Creating a Relationship Between Data Services

In a data service adding a relationship is a three-part process:

1. Add and name the relationship.

Figure 4-6 Adding a Relationship to a Data Service Using Right-click Menu Option



2. Associate the relationship with an existing data service.
3. Use the Relationship wizard to define the relationship.

Using the Relationship Wizard to Create Navigation Functions

You can develop fully-functional binary navigation functions using the Relationship wizard.

The value of navigation functions is that client applications can call the function using complex parameters without having to know the internal structure of function, join conditions, and so forth. From the perspective of the data service creator, the internals of the function can be changed without affecting applications dependent on the ability to invoke the data service function.

When you choose to create a relationship through Design View or within a model diagram, the Relationship wizard is invoked. With the wizard you can set the following navigation function notations:

- Role names

- Direction
- Cardinality

You can also identify parameters and specify where clauses.

Setting Relationship Notations: Role Names, Direction, Cardinality

The first dialog of the Relationship wizard allows you to set role names, direction, and cardinality. [Table 4-8](#) provides details on the callouts shown in [Figure 4-7](#).

Figure 4-7 Relationship Wizard Specifying Direction, Cardinality, and Role Name

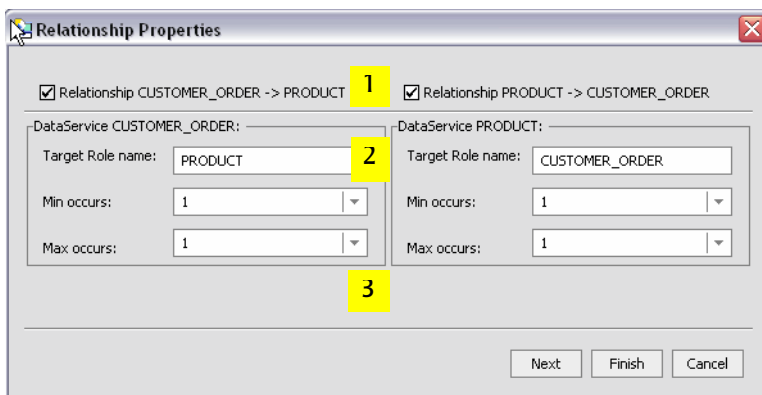


Table 4-8 Primary Relationship Settings

Key	Component	Purpose
1	Direction	<p>Query functions are typically developed using the XQuery Editor. A bidirectional relationship is the default condition. This means that each data service will have a navigation function that invokes the related data service. Direction notations have no run-time effect.</p> <p>Direction can also be specified through the Property Editor associated with each data service or through a model diagram.</p>
2	Role name	<p>Each end of a relationship can have a target role name. By default, the role name is the same as its adjacent data service. For example, the default role name for the ADDRESS data service is ADDRESS. You can change the role name in the Relationship wizard.</p> <p>Role names can also be specified through the Property Editor associated with your data service or through a model diagram showing the relationship.</p> <p>Note: Role name notations have no run-time effect.</p>
3	Cardinality	<p>Cardinality notations can be set for each side of the relationship. The default cardinality is 1-to-1 but this can be changed to any combination of <blank>, 0, 1, and <i>n</i>.</p> <p>Cardinality can also be specified through the Property Editor associated with your data service or through a model diagram showing the relationship.</p> <p>Note: Cardinality notations have no run-time effect</p>

Setting Function Name, Identifying the Opposite Data Service, Mapping Parameters, and Building Where Clauses

The second Relationship wizard dialog page allows you to set the navigation function name and other characteristics.

Figure 4-9 Relationship Wizard Dialog Specifying Function Name, Parameters, and Where Clauses

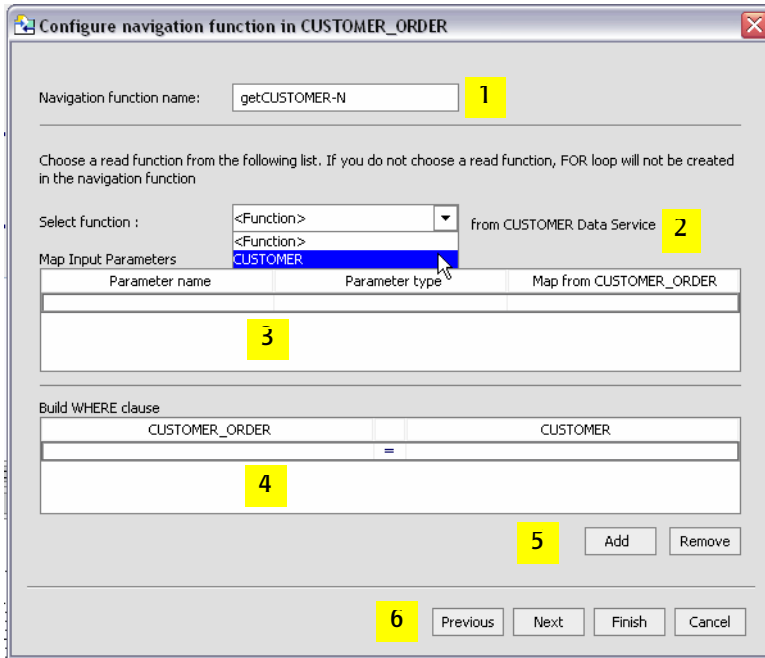


Table 4-10 provides details on callouts shown in Figure 4-9.

Table 4-10 Primary Relationship Settings

Key	Component	Purpose
1	Navigation function name	<p>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.</p> <p>However, you can change the navigation function name to any valid function name.</p> <p>Note: 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.</p>
2	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.
3	Map input parameters	If the related function has input parameters, the name and type of the available parameters are displayed. You can then use a pulldown menu to select an element from the target data service to map as the input parameter.
4	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.
5	Add or Remove	Allows you to add additional where clauses or delete a selected where clause.
6	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.

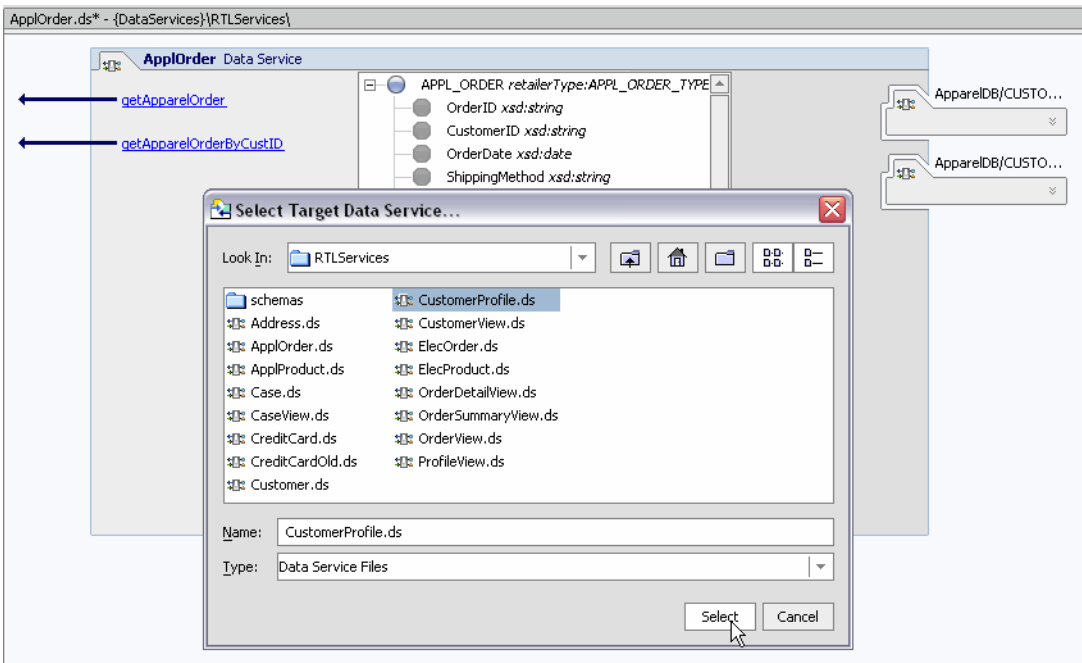
Example of Creating a Navigation Function

This section contains a small example showing how you can use the Relationship wizard to create fully-formed navigation functions. The goal is to create a navigation function that returns the first available address on file for a particular customer by supplying a customer ID.

The following steps use the RTLApp provided with DSP.

1. Starting with the RTLServices/AppOrder data service in Design View, select Add Relationship from the right-click menu.
2. Select a target data service. In this case RTLServices/CustomerProfile.

Figure 4-11 Selecting a Target Data Service for the AppOrder Navigation Function



3. Next you can set direction and cardinality.

The relationship remains bidirectional, meaning that you can get customer profile information by supplying an address object and you can get address information using a customer profile object. However, the cardinality relationship notation of Customer Profile \rightarrow Address is 1-to-*n*, since a customer can have multiple orders.

Figure 4-12 Setting Direction and Cardinality for the Relationship

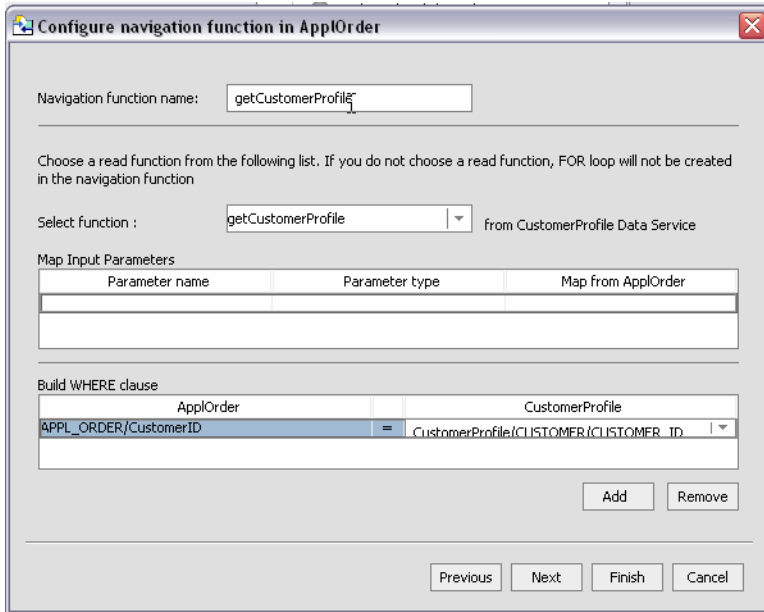
The screenshot shows a dialog box titled "Relationship Properties" with a close button in the top right corner. At the top, there are two checked checkboxes: "Relationship ApplOrder -> CustomerProfile" and "Relationship CustomerProfile -> ApplOrder". Below these are two panels. The left panel is titled "DataService ApplOrder:" and contains three fields: "Target Role name:" with a text box containing "CustomerProfile", "Min occurs:" with a dropdown menu showing "1", and "Max occurs:" with a dropdown menu showing "1". The right panel is titled "DataService CustomerProfile:" and contains three fields: "Target Role name:" with a text box containing "ApplOrder", "Min occurs:" with a dropdown menu showing "1", and "Max occurs:" with a dropdown menu showing "n". At the bottom of the dialog are three buttons: "Next", "Finish", and "Cancel".

4. Click Next. This creates the first navigation function which is given a default name of `getCustomerProfile()`.

The next stage for each navigation function is to:

- accept or change the name of the navigation function
- identify a read function contained in the navigation function (there may be more than one)
- specify parameters to invoke if parameters are supported by the underlying query function
- optionally add one or multiple where clauses

Figure 4-13 Defining the First Navigation Function



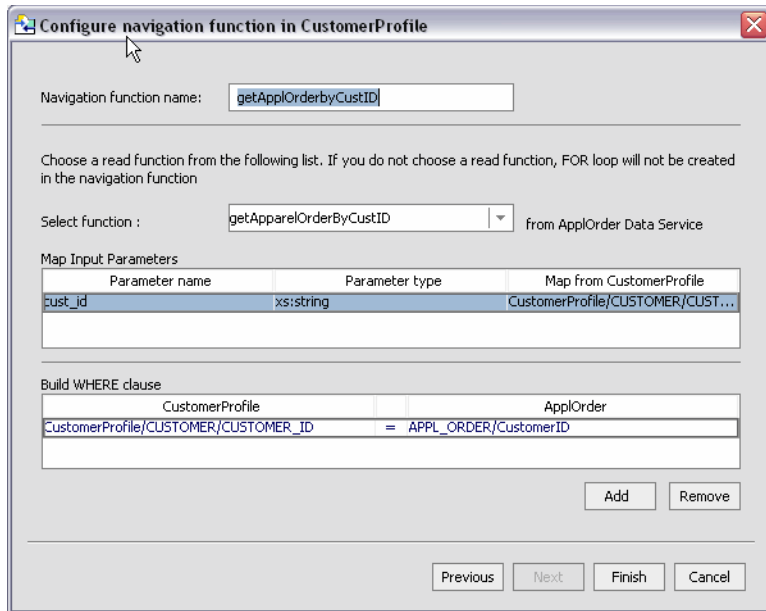
In the case of the `getCustomerProfile()` navigation function:

- there is only a single read function
 - there are no parameters
 - the where clause join elements are `APPL_ORDER/CustomerID` and `CustomerProfile/Customer/CUSTOMER_ID`
5. Click Next to define the opposite navigation function whose default name is `getAppOrder()`.

The apparel orders data service more typically contains multiple read functions. If you select `getApparelOrdersByCustID()`, then you will be able to map an element (`cust_id`) from the opposite data service.

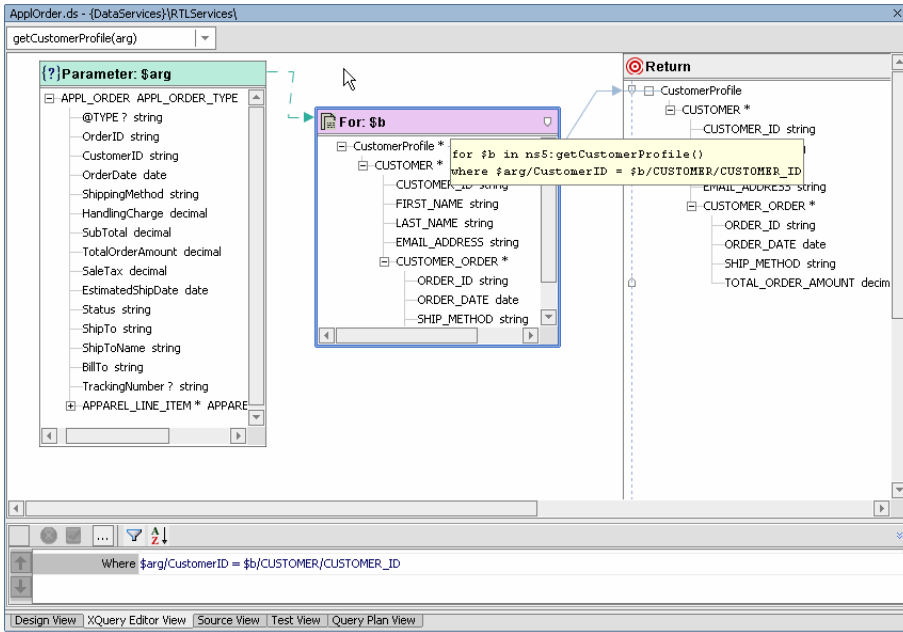
Notice in [Figure 4-14](#) that the where clause you defined for the first navigation function is pre-determined and shown in read-only format.

Figure 4-14 Selecting a Parameter



6. Click Finish.

Figure 4-15 Resulting getCustomerProfile() Navigation Function



Testing Your Navigation Function

When you execute a navigation function in Test View, you can provide input in the form of a complex parameter such as would result from, for example, getting back a customer record. Alternatively, you could use the Test View template option to supply the appropriate parameter. See [“Using the XML Type to Identify Input Parameters”](#) on page 7-10.

Navigation Functions in Source View

In data service Source View the navigation function is defined through a pragma and a function body. (For details see the Data Services Platform [XQuery Developer’s Guide](#)).

For example, a navigation function named `Payment()` has a read function `getPaymentList()`.

The navigation function appears as:

```

declare function ns1:getCustomer($arg as element(ns0:APPL_ORDER)) as
element(ns15:PROFILE) * {
    for $b in ns16:getCustomerByCustID($arg/CustomerID)
return $b
};

```

A key element in understanding this function is in the namespace ns15 which imports the schema that models the XML type, `PAYMENTList.xsd`. The namespace is defined as:

```
import schema namespace ns15="urn:retailerType" at
"ld:DataServices/RTLServices/schemas/Profile.xsd";
```

Note: If you modify a role name in the pragma of your data service, and that relationship exists in any model diagram, then you will need to similarly modify the role name in any model diagrams in which the relationship appears. Otherwise the relationship will become invalid.

Working with Logical Data Service XML Types

Read functions associated with data services return information in the shape of the data service's XML type.

Note: Logical data services are built upon physical data services which in turn represent an underlying physical data source. Physical data service are created by importing metadata on a physical source and updated through synchronization. The schema file or XML type generated by this process should never be modified either in DSP or externally. Doing so risks invalidating your data service and dependent logical data services. (If you need to modify a data service based on a single source it is a simple matter to create a logical data service based on that one source.)

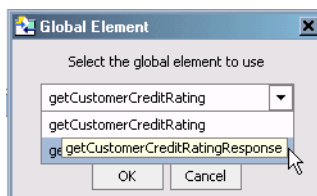
Associating an XML Type

You can add or replace an XML type that has been associated with an data service using a browser. Your type must be located in the your application file structure.

Selecting a Global Element

If the schema you select has more than one global element, a dialog allows you to choose the global element you want to use.

Figure 4-16 Select Global Element Dialog Box



Editing an XML Type

You can also edit an XML type. Several XML type right-click menu options are available ([Table 4-17](#)).

Warning: Editing changes to an XML types in Design View immediately modify the schema file upon which the XML type is based. Such changes cannot be reversed through the Undo command. For this reason, XML types should be modified carefully, with adequate backup in case you need to revert to the original version.

Table 4-17 Right-click XML Type Editing Options

Option	Purpose
Add Child	Adds a child element to the currently selected element. Available sub-menu options include special-purpose schema elements Choice and All.
Add Sibling	Adds a sibling element to the currently selected element. Available sub-menu options include special-purpose schema elements Sequence and Choice.
Add Attribute	Adds an attribute to the currently selected element.
Delete	Deletes the currently selected element or attribute. This option is not available for the root element of the schema.
Allow Global Types and Elements Editing	A toggle that applies to the entire schema. Schemas should be edited with care. To do so, this option must be selected.
Go to Source	Opens the XML type in the built-in schema editor.
Move Up	Moves the selected element towards the top of the schema.
Move Down	Moves the selected element towards the bottom of the schema.
Find	Finds text within the selected complex element (such as the root element).

Another option, Enable Optimistic Locking, becomes available for elements in relational-based XML types under some conditions. See [“Enable/Disable Optimistic Locking” on page 4-28](#).

[Table 4-18](#) identifies how various right-click options apply to different XML type elements.

Table 4-18 XML Type Editing Options / Element Matrix

Element	Add Child Element/ Choice/ All	Add Sibling Element/ Sequence/ Choice	Add Attribute	Delete	Move Up/ Move Down
Root element	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>		
Complex element	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
Leaf element	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
Conditional element	<input checked="" type="checkbox"/>			<input checked="" type="checkbox"/>	
All element	<input checked="" type="checkbox"/>			<input checked="" type="checkbox"/>	
Sequence element	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
Choice element	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		<input checked="" type="checkbox"/>	
Attribute				<input checked="" type="checkbox"/>	

In some cases complex type components that appear in schemas will not appear in your XML type.

Warning: XML types are based on schemas which may be used by other data services. For this reason, XML types should be modified carefully, with adequate backup in case you need to revert to the original version. Similarly, all the functions in your data service should be written to return the XML type of your data service.

External Editing of XML Types

In addition to the right-click menu described in [Table 4-18](#), you can use the Go to source command to edit your schema file using WebLogic Workshop's assigned text editor.

Creating an XML Type

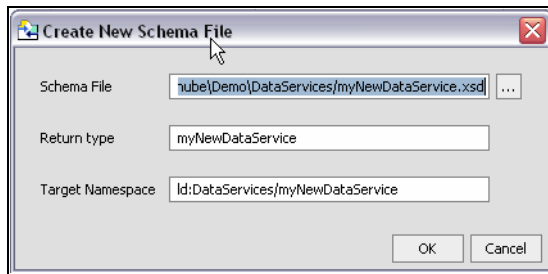
You can choose to create an XML type for a new data service. Since your data service already has a name, you need only supply:

- A schema file (XSD file) name

- An XML type root element
- A target namespace

By default, the name of your data service is the same as the schema file name, the schema, and the target namespace.

Figure 4-19 Create New Schema File Dialog

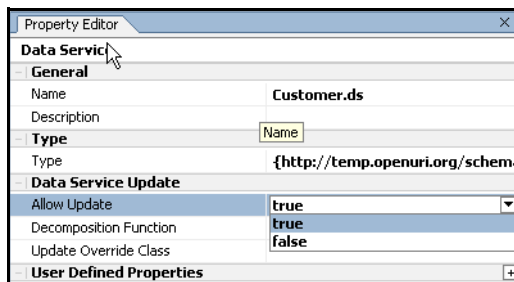


Once created, you can use the data services built-in schema editor to create your schema. Alternatively, you can create a schema in a program such as XMLSpy.

Managing Your Data Service

There are several important pre-deployment tasks you need to accomplish before you can make your data service available to client applications. This includes setting properties for your data service and its functions.

Figure 4-20 Data Service Properties



You can use the Properties Editor (View → Property Editor) to set or change key data service functionality including:

- Enabling or disabling update logic.

- Specifying the Java file to access for update logic.
- Creating user-defined properties, which then become available to the DSP Metadata Browser.
- Enabling or disabling caching for particular functions.
- Changing relationship settings include role name, target data service, and cardinality.

See [“Notable Design View Properties” on page 4-40](#).

Refactoring Data Service Functions

You can refactor data service functions insofar as they can be renamed or safely deleted. See [“Refactoring DSP Artifacts” on page 2-25](#).

Finding Usages of Data Services Platform artifacts

For most DSP artifacts you can quickly determine the artifacts usage through a right-click menu option. See [“Usages of Data Services Artifacts” on page 2-20](#).

Setting Update Options

Each data service contains a set of properties that control its update characteristics.

Note: For complete information on decomposition functions, override classes, optimistic locking settings, and other SDO-related information see [“Enabling SDO Data Source Updates”](#) in the *Application Developer’s Guide*.

Also see in the Data Services Platform [Samples Tutorial Part II](#):

- Lesson 23: Performing Custom Data Manipulation Using Update Override
- Lesson 24: Updating Web Services Using Update Override
- Lesson 25: Overriding SQL Updates Using Update Overrides

Allowing Updates

You can use the Allow Update option in the Property Editor to control whether calling applications can exercise update logic associated with your data service. This is especially important in regard to relational-based data services, since update logic is automatically available unless disabled.

Set the option to True to allow update; False to prevent updates.

Setting the Override Class

In order to update non-relational sources that are associated with your data service you need to create an update override class. In addition, you may want to overwrite built-in update logic for relational sources to apply custom logic to the update process.

Before you can set the override class, you need to develop it. The steps involved are:

- Add an appropriately named Java class to your DSP-based project.
- Within the Java file, implement the UpdateOverride interface.
- Import the required packages into your class and add a performChange() function to the class.
- Implement your processing logic.
- Associate your data service with the class.

```
<javaUpdateExit className="nameOfYourJavaClass" />
```

For information on developing an override class see *“Enabling SDO Data Source Updates”* in the *Application Developer’s Guide*.

Note: Each data service can have only one update override class. However, multiple data services can share the same update override class.

Enable/Disable Optimistic Locking

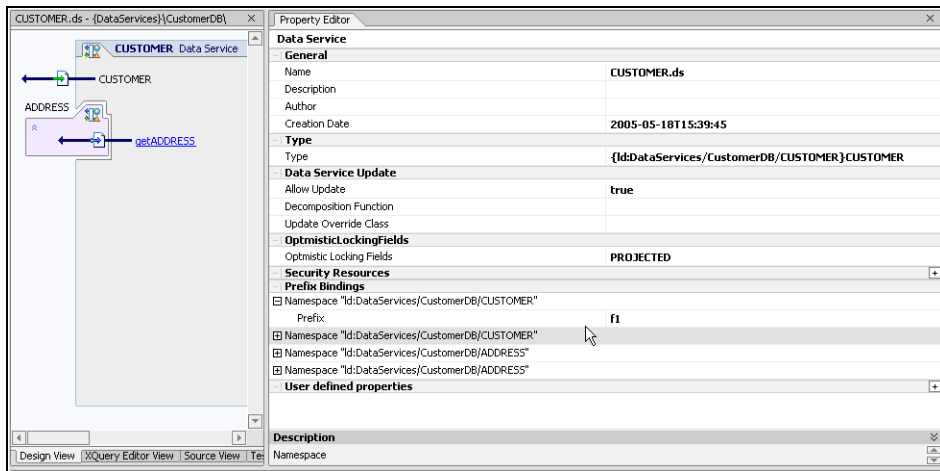
The SDO update mechanism for relational data uses an optimistic locking policy to avoid change conflicts. With optimistic locking, the data source is not locked after the SDO client acquires the data. Later, when an update is needed, the data in the source is compared to a copy of the data at a time when it was acquired. If there are discrepancies, the update is not committed.

Optimistic locking update policy is set for each data service. The following table lists the three optimistic locking update policy options.

Optimistic Locking Update Policy	Effect
Projected	<p>Projected is the default setting. It uses a 1-to-1 mapping of elements in the SDO data graph to the data source to verify the “updateability” of the data source.</p> <p>This is the most complete means of verifying that an update can be completed, however if many elements are involved updates will take longer due to the greater number of fields needing to be verified.</p>
Updated	<p>Only fields that have changed in your SDO data graph are used to verify the changed status of the data source.</p>
Selected Fields	<p>Selected fields are used to validate the changed status of the data source.</p>

For relational-based data service the Enable/Disable Optimistic Locking option becomes available for elements in its XML type when the optimistic locking property is set to Selected. (Optimistic locking policies are viewed and set through the Property Editor ([Figure 4-21](#))). For information on additional properties see “[Notable Design View Properties](#)” on page 4-40.

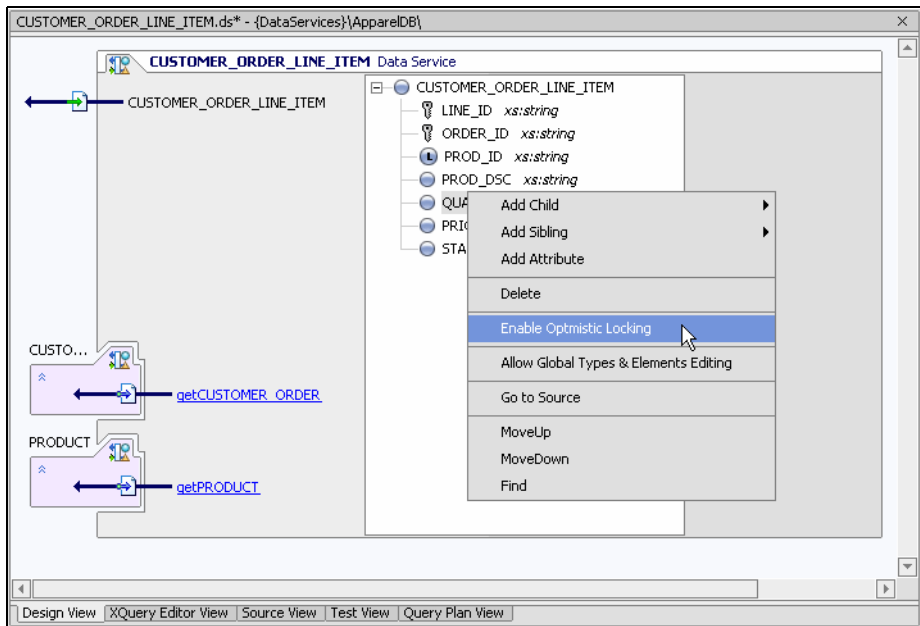
Figure 4-21 Data Service Allowing Updates and Optimistic Locking on Selected Fields



When active, the Selected Fields option allows you to validate optimistic locking logic prior to an update. Any number of fields can be selected through the right-click menu associated with the XML type. (If a complex element is selected, all its children are selected even though they are not so marked.)

When the Selected Fields option is picked, a right-click toggle option named Enable/Disable Optimistic Locking becomes available. Multiple elements can be selected.

Figure 4-22 Disabling Optimistic Locking Policy for a Field



In Figure 4-22 two fields are selected, PRODUCT_ID and QUANTITY.

These choices are reflected in the Source View pragma.

```
<optimisticLockingFields>
  <field name="PRODUCT_ID"/>
  <field name="QUANTITY"/>
</optimisticLockingFields>
```

For complete details on handling change conflicts based on optimistic locking policies see [“Enabling SDO Data Source Updates”](#) in the *Application Developer’s Guide*.

Adding Security Resources

Security resource settings are created at the data service level and activated at through the Data Services Platform Console. The steps involved are:

- Create as many security resources as are needed by your data service.
- Structure your query to support security resource validation.
- Assign security resources to your element through the DSP Console.

- Use Test View to validate your security policy settings.

An easy way to understand security settings is to create an example using the Shipping data service found in the DataServices/Demo/Java/Logical folder of RTLApp.

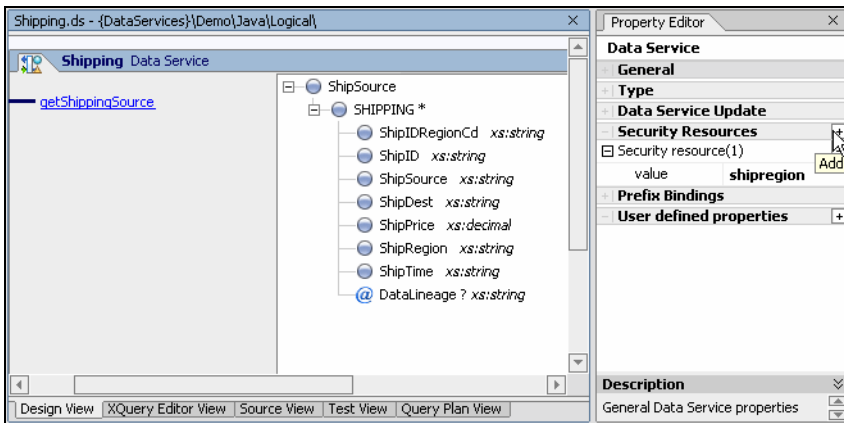
Goal. The goal is to restrict access for the East shipping region to the XML type’s ShipRegion string to a particular traffic monitor named Igor.

The following section describe the steps involved.

Create Necessary Security Resources

1. Open your data service.
2. Open the Property Editor.
3. Create a security resource by clicking the + on the Security Resource line. Any value (name) can be assigned to a security resource. In this case the name of an element in your XML type is used.

Figure 4-23 Create a Security Resource



This action add your new security resource (highlighted below) to the data service pragma in Source View.

```
(::pragma xds <x:xds targetType="ship:ShipSource"
xmlns:ship="http://Logical/ShipSource"
xmlns:x="urn:annotations.ld.bea.com">
  <creationDate>2005-11-01T15:50:28</creationDate>
  <userDefinedView/>
  <secureResources>
    <secureResource>shipregion</secureResource>
  </secureResources>
```

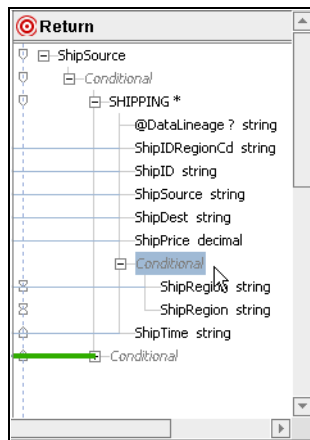
```
</x:xds>
::)
```

Structure Your Query To Support Security Resource Validation

In this section you use the XQuery Editor to attach your newly created security resource to the EAST group, ShipRegion element.

1. Click XQuery Editor View.
2. In the return type attach a security resource to the EAST group, ShipRegion element. Do this by right-clicking on the element to which you want to attach a security resource, then select the Make Conditional option.

Figure 4-24 Creating an If-Else Construct for East Group's ShipRegion String

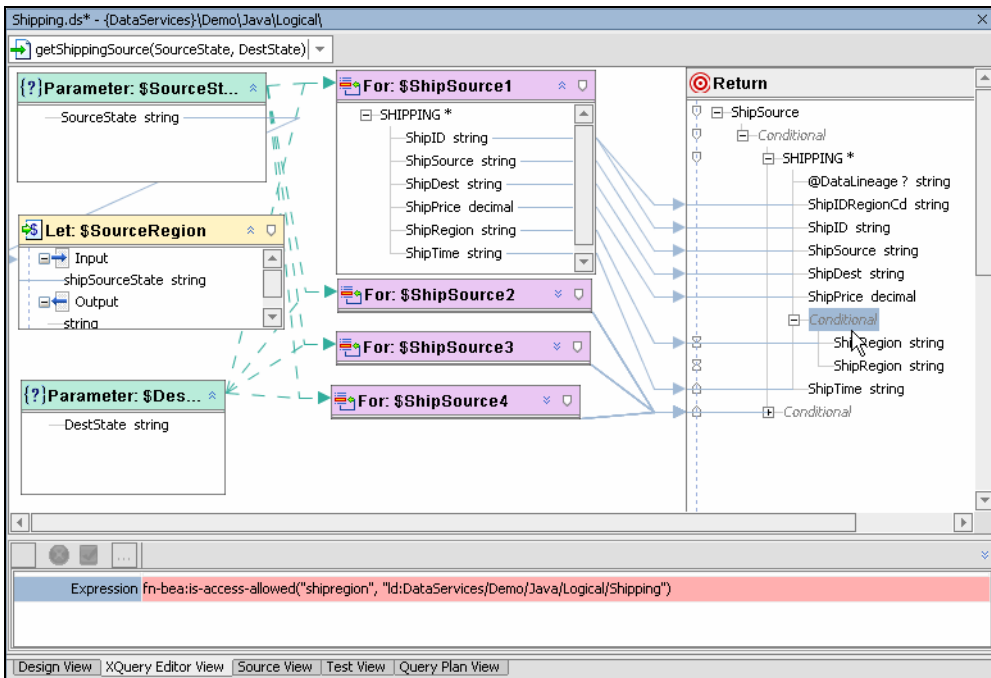


3. Associate your new conditional element with the built-in `fn-bea:is-access-allowed()` function by clicking on the element and dragging the function into the Expression editor. The function takes two parameters: a string and the name of a data service. In this case the string exactly matches your security resource name.

Note: (For details on built-in BEA functions see the DSP *XQuery Developer's Guide*. For details on editing expressions see “Transforming Data Using XQuery Functions” on page 6-38.)

4. Populate the function parameters by either entering the appropriate strings or dragging elements into the function placeholders.

Figure 4-25 Establishing Security Control for East Group’s ShipRegion Element



- The If-Else construct may now be read as “if access is allowed to the element return data, otherwise return nothing”. In many cases it is appropriate to return the fact that access is not allowed. This can be accomplished by setting the expression associated with the Else side of the conditional to “N/A” (not available).

In Source View your conditional is rendered as an XQuery if-else statement.

```

if (fn:upper-case($SourceRegion) eq 'EAST') then
(
  for $ShipSource1 in ns10:getShipSource1()/SHIPPING
  where $ShipSource1/ShipSource eq $SourceState
    and $ShipSource1/ShipDest eq $DestState
  return
  <SHIPPING DataLineage?="{ 'EAST Shipping Source' }">
    ...
    <ShipPrice>{fn:data($ShipSource1/ShipPrice)}</ShipPrice>
    {
      if (fn-bea:is-access-allowed("shipregion",
"Id:DataServices/Demo/Java/Logical/Shipping")) then

```



```

        <ShipRegion>{fn:data($ShipSource1/ShipRegion)}</ShipRegion>
      else
        <ShipRegion>{"N/A"}</ShipRegion>
    }
    <ShipTime>{fn:data($ShipSource1/ShipTime)}</ShipTime>
  </SHIPPING>

```

6. Build your project. This deploys your new security settings to the server.

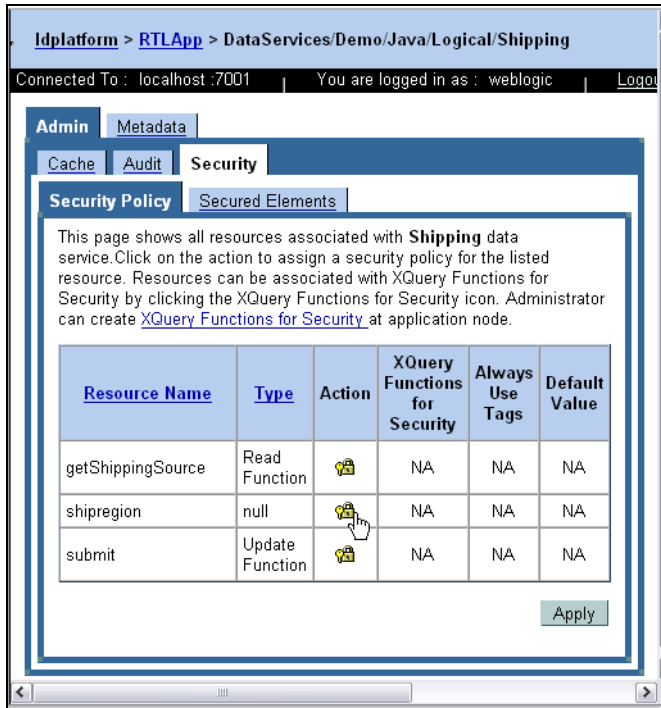
Assign Security Resources Through the DSP Console

The next steps involve the Data Services Platform Console (see [“Securing Data Service Platform Resources”](#) in the DSP *Administration Guide* for complete details).

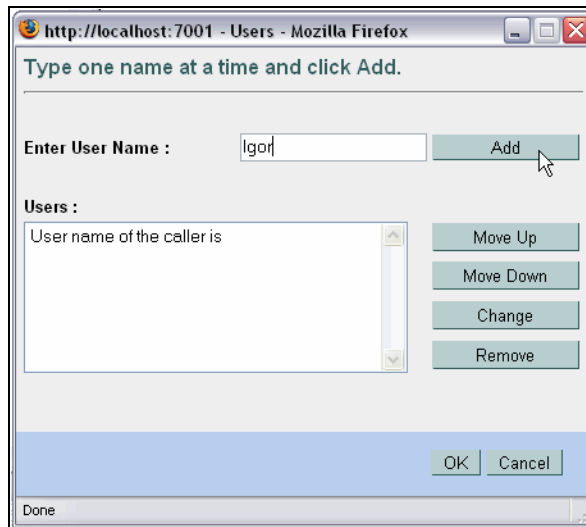
1. Sign into the DSP Console. For the RTLApp sample the user name and password are both ‘weblogic’.

Note: Unless a secured resource has been marked as available to user weblogic or some group that user weblogic is a member of, it will not be available.
2. Find the heading Search Metadata. Click on the Search Idplatform (the RTLApp sample domain server).
3. In the data service name field search for “shipping”.
4. In the Search Results click on the Shipping.ds name link to view the various administrative, caching, auditing, metadata search, and security options available to the data service.
5. Click the Security tab.

Figure 4-26 Security Policies Associated with the RTLApp's Shipping Data Service



6. Since you created a security resource named shipregion for your data service, it appears as an available resource name. Now security policies must be associated with the resource. Click the Action icon.
7. In the Administration Policies pane select the User name of the caller policy condition, then click Add.
8. A dialog box appears where you can enter the name of the user.

Figure 4-27 Associating a User Name with a Security Resource

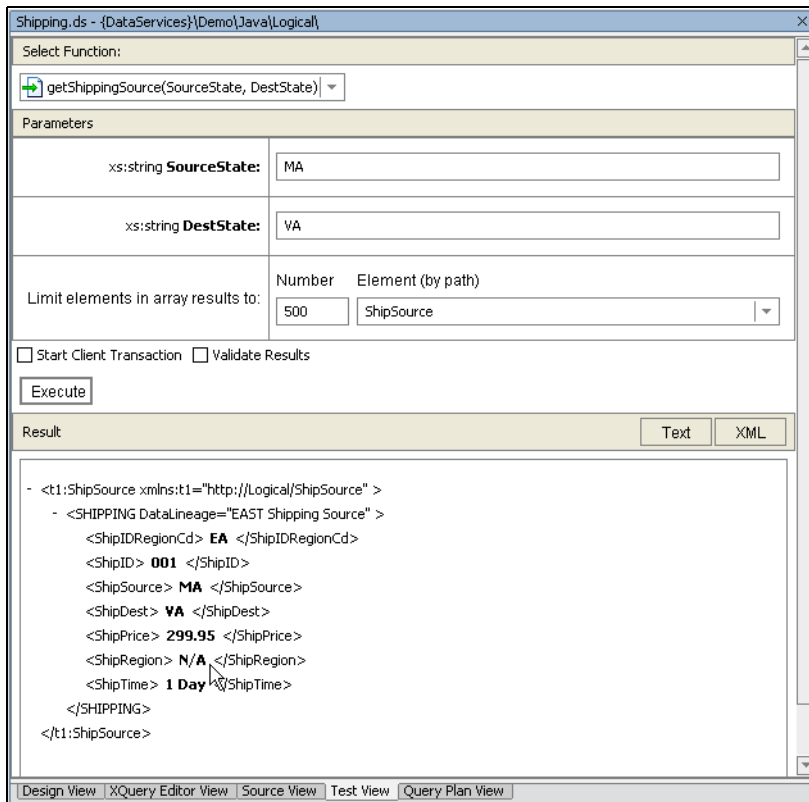
9. Click Apply.

Validating Security Policies Through Test View

Once security policies are established, they should be tested.

1. With the Shipping data service selected click the Test View tab.
2. The `getShippingSource()` function requires a source state and a destination state. (Valid states are shown in the `Examples.txt` source file which is located in the `Demo/Java/Logical` folder.)
3. Enter MA as the source state and VA as the destination, then click Execute.

Figure 4-28 Validating that the ShipRegion Element is Secured



4. Notice that rather than returning the region, the Else string N/A is returned. This is because the registered user of Test View is user weblogic, not user Igor.

Caching Functions

For each function in your data service, the Allow Caching option can be set to True or False. If False, results from executing your query function cannot be cached. If True, results from earlier invocations of your function can be cached if cache for that function is enabled through the Data Services Platform Console. In other words, in order to cache a function it must be Enabled:True in its data service and also enabled through the DSP Console. For details on enabling cache for a function as well as setting the cache's TTL (time-to-live) see the DSP [Administration Guide](#).

Caching Considerations

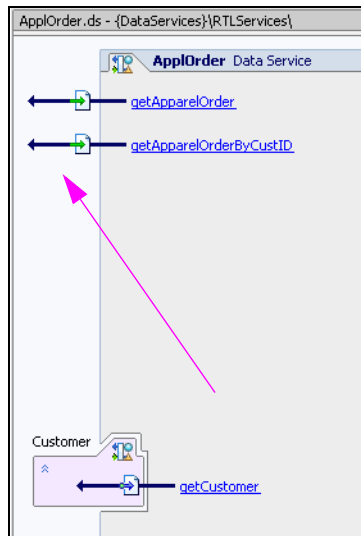
There are several things to keep in mind when considering whether to enable caching for a particular function:

- If the data accessed by your function is updated frequently, the function is not a good candidate for caching.
- Generally speaking, you should only enable cache to data service functions that have parameters. Since relational tables do not, by definition, have parameters, the cache for such tables should generally not be enabled.

Setting Caching Policy for a Function

To inspect or set the Allowed caching policy for a particular read function in your data service, click on the arrow to the left of the name of the function, then set its caching policy through the Properties Editor.

Figure 4-29 Click Arrow to the Left of a Function Name to Inspect or Set Its Caching Policy



You need to build your application in order for cache policy changes effective.

Note: When a cache policy of Enabled:False is set for a function it cannot be overridden through the Data Services Platform Console.

Notable Design View Properties

The following table identifies notable Data Services Platform Design View properties.

Table 4-30 Notable Design View Properties

Focus	Property	Settings	Comments
Data service	Name	Editable	Must end in .ds
	Description	Text	Optional
	Author	Text	Optional
	Creation Date	Non-editable	
	Type	URI to optional XML type	Also known as XML type.
	DS Update : Allow Update	True / False	Allows calling applications to execute the data service's update logic.
	DS Update : Decomposition Function	Selectable for logical data services with more than one read functions.	Identifies the function used for decomposition of the data service. In the case of physical data services the decomposition function is pre-defined by the source metadata. For logical data services, however, you can change the default decomposition function to another read function in your data service. For additional information on decomposition functions see “Leveraging Data Service Reusability” on page 9-15.
DS Update : Override Class	Optional and editable	Identifies a external Java class that provides custom update logic.	

Focus	Property	Settings	Comments
	Optimistic Locking Fields	Projected / Updated / Selected Fields	Applies only to relational-based data services.
	Security Resources	Any number of name:value pairs.	See “Adding Security Resources” on page 4-31.
	Prefix bindings	Any valid, non-conflicting prefix can be entered for namespaces defined in the data service.	See “Refactoring DSP Artifacts” on page 2-25.
	User-defined Properties	Optional and editable	Create any number of name/value pairs.
Data Service Read and Private Functions and Procedures	Name	Editable	
	Function type	Selectable (read or private)	Selectable for read function or private functions; not selectable for procedures.
	Cache enabled	True / False	Enables cache for the function.
	User Defined Properties	Optional and editable	Create any number of name/value pairs.
XML type	Root: Name	Editable	Typically same name as the data service without the file's extension.
	Root: Is Referenced	False	Read only. For the root element the Is Referenced property is always false as it is always a global element in the schema.
	Root: Type	<blank> or named type	Blank if the root element is an anonymous type; otherwise named type is shown

Focus	Property	Settings	Comments
	Element: Is Referenced	True / False	Read only. Identifies any elements that are imported into the current function. In source this appears as <code>ref="element"</code> .
	Element: Type	XML type	Examples: <code>xs:int</code> ; retailer: <code>CUSTOMER_VIEW</code>
	Element: Min Occurs	1, 0, or <i>n</i>	
	Element: Max Occurs	1, 0, or <i>n</i>	
	Element: Native Type	Data type	Available only for physical data. Example: <code>VARCHAR</code>
	Element: Native Size	Size of the data	Available only for physical data. Example: 10

Focus	Property	Settings	Comments
	Primary key: AutoNumber	<blank>, identity, sequence, or userComputed	<p>This and the Sequence Object Name option appear for elements representing primary keys in relational-based physical data services.</p> <p>Autonumber can be used to provide a value for a database primary key.</p> <ul style="list-style-type: none"> • Leaving the field blank means you will provide a value for the primary key. • The identity option pertains to IBM DB2, Sybase, SQL Server, and MySQL. In this case the database will provide a value for the primary key. • Sequence objects are available for DB2 and Oracle. You must provide a sequence object name. • User computed is a notational flag indicating that the primary key information has been provided to the database through your SDO custom update override class. <p>Note: It is not necessary to set this flag in order for the update override computed primary key logic to be used.</p>

Focus	Property	Settings	Comments
	Primary key: Sequence Object Name		If sequence is selected in the AutoNumber property (above), then the sequence object name must be supplied
Related Data Service	Role Name	Editable	Also changes the role name shown in a model diagram.
	Related Data Service	Path to the related data service	
	Min Occurs	1, 0, or <i>n</i>	
	Max Occurs	1, 0, or <i>n</i>	
	Opposite Role Name	Editable	Also changes the role name shown in the model diagram.
Relationship Read Function	Name	Editable	
	Cache	True / False	Enables cache for the function.
XML File Library (XFL)	Return type	Non-editable	Always navigation type.
	User Defined Properties	Editable	Create any number of name/value pairs.
	Name	Editable	Must end in <code>.xfl</code>
	Prefix bindings	Any valid, non-conflicting prefix can be entered for namespaces defined in the data service.	See “Refactoring DSP Artifacts” on page 2-25.

Modeling Data Services

Using BEA Aqualogic Data Services Platform (DSP), you can create and maintain models of your enterprise data services. Models describe data, relationship between data objects, data semantics, and consistency constraints.

Models also express relationships between physical data services, logical data services, or a combination. In DSP all model relationships are binary; each binary relationship is expressed in a model diagram as one or more lines between two data services.

You can use DSP model diagrams to:

- Obtain a high-level, visual view of data resources
- View the relationships between physical and logical data resources
- Facilitate the creation or modification of relationships between resources
- Quickly access or create a data service
- Modify a XML type of a data service

The following topics are covered in this chapter:

- [Model-Driven Data Services](#)
- [Building a Simple Model Diagram](#)
- [Building Data Service Relationships in Models](#)
- [Working with Model Diagrams](#)

- [How Changes to Data Services and Data Sources Can Impact Models](#)

Note: For more information on data service modeling concepts see “[Modeling and a Service-Oriented Architecture](#)” in the Data Services Platform *Concepts Guide*.

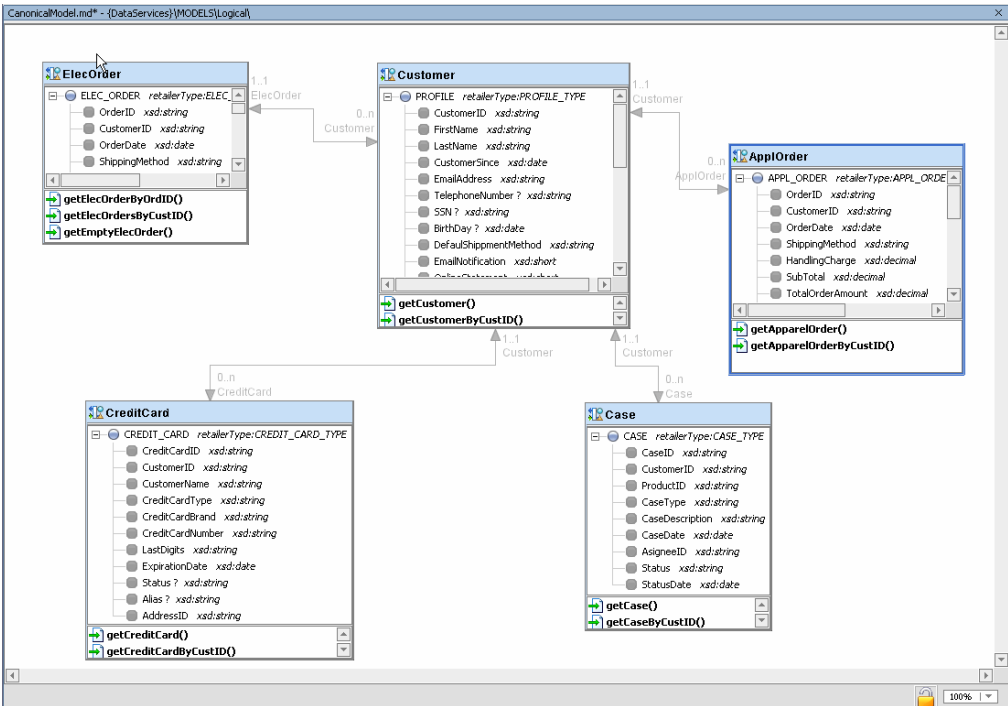
Also see in the Data Services Platform [Samples Tutorial Part I](#):

- Lesson 5: Modeling Data Services

Also see in the Data Services Platform [Samples Tutorial Part II](#):

- Lesson 20: Implementing Relationship Functions and Logical Modeling

Figure 5-1 Model Diagram of Physical Data Services



A model diagram is a graphical representation of a data model supported by DSP. In addition to showing collections of data services and relationships between data services, model diagrams also

identify role direction and cardinality information at each end of the relationship. 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.

Model-Driven Data Services

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 physical 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 in a modeler diagram.

In DSP model relationships are logical connections between two data services. The connections describe:

- The direction of the binary relationship (one- or two-way)
- The cardinality of the relationship (1-to-1, 1-to-many, 0-to-many, or many-to-many)
- A role name for each side of the relationship

Relationships can have one or more *navigation functions* that allows data associated with one data service (such as Customer) to potentially become a complex parameter for a related data service (such as Orders).

Some relationships — such as between relational data services — are automatically inferred through introspection of primary and foreign keys. See [“Importing Relational Table and View Metadata” on page 3-8](#) for details.

Additional relationships can be created 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 are automatically identified.
- Graphically, through gestures you make in your model diagram.
- Programmatically, through Source View of a data service.

Logical and Physical Data Models

Models can represent any combination of logical and physical data services.

Physical Data Models

Physical data services represent data that physically resides in the enterprise (see [Chapter 3](#), “[Obtaining Enterprise Metadata](#)”). The source may be from a relational database, a Web service, an XML data stream or document, a flat file such as a spreadsheet, or a Java file contain custom functions.

Logical Data Models

Logical data models are developed in DSP and are based on physical other logical data.

In other words, each *physical model entity* represents a single data source. *Logical data model entities* represent composite views of physical and/or logical models.

Rules Governing Model Diagrams

Model diagrams follow a set of rules:

- Each entity in the model has a title which is the data service local name (the fully-qualified name is visible as a mouse-over).
- Data services in models need not be associated with an XML type. However, if they are, the type is always displayed. For physical data services you have the option of displaying native schema types such as Integer(10).
- 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, one 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.

Note: 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.

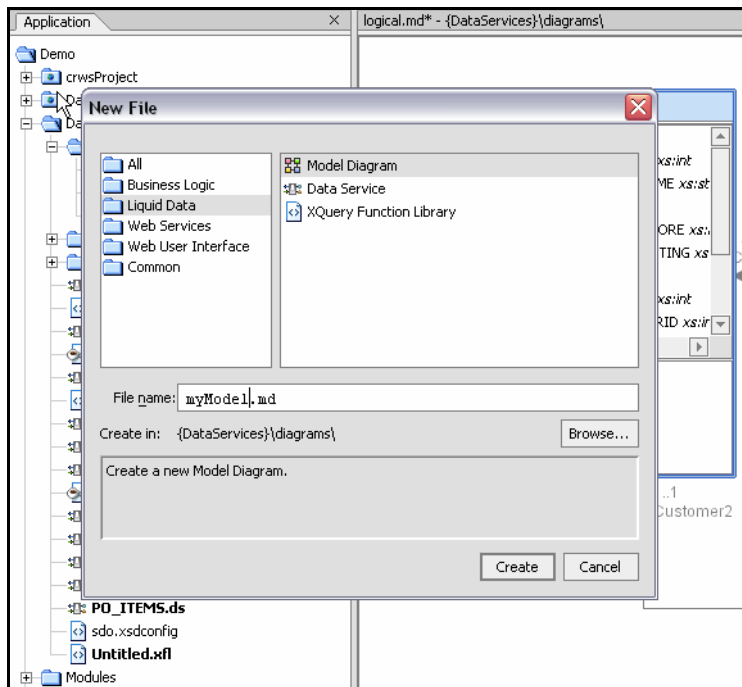
Building a Simple Model Diagram

You can create a data model by selecting a DSP-based project and then choosing:

File → New → Model Diagram

The following example describes how to create a model around physical data.

Figure 5-2 Creating a Data Model Using the File Menu



This example assumes that you are using the DSP demonstration program RTLApp.

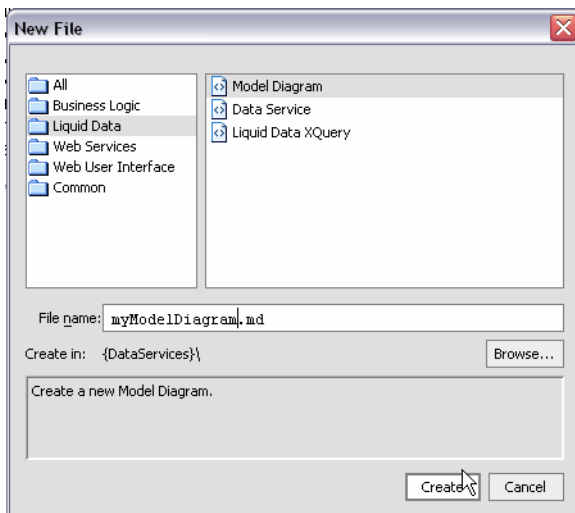
The data services used in the example in this chapter are PRODUCT, CUSTOMER_ORDER, and CUSTOMER_ORDER_LINE_ITEM. See [Chapter 3, “Obtaining Enterprise Metadata”](#) for details related to importing metadata.

Here are the steps required to create and populate a simple model:

1. First choose a name and physical location for your model. It can be created anywhere in your BEA WebLogic application. In the demonstration application provided with DSP, models are located in a MODELS folder.

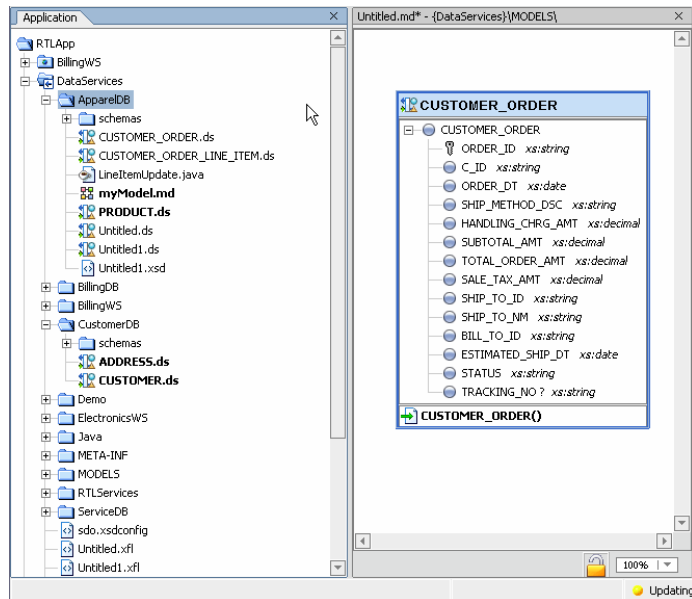
2. Right-click on your project and select New → Model Diagram.
3. Pick a location for your model and name it myModel Diagram.


Figure 5-3 Selecting a Data Service



4. Right-click in the work area of your new model and select Add Data Service.
5. From the dialog box select the CUSTOMER_ORDER data service in Data Services/ApparelDB.

Figure 5-4 Adding Data Services to a Data Model

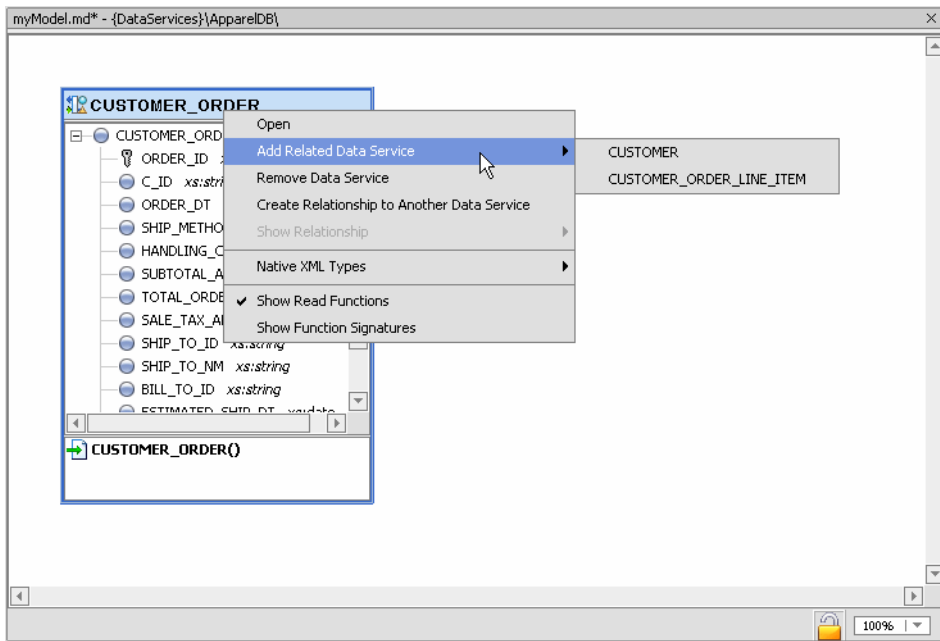


Since the data services in this example are representations of relational sources, a considerable amount of metadata is available. For example, primary keys are identified from the data; these are shown in data service type as a key icon ().

6. Right-click on the CUSTOMER_ORDER data service titlebar and choose the Add Related Services command.

In this case you will see that two relationship already exists: CUSTOMER and CUSTOMER_ORDER_LINE_ITEM ([Figure 5-5](#)).

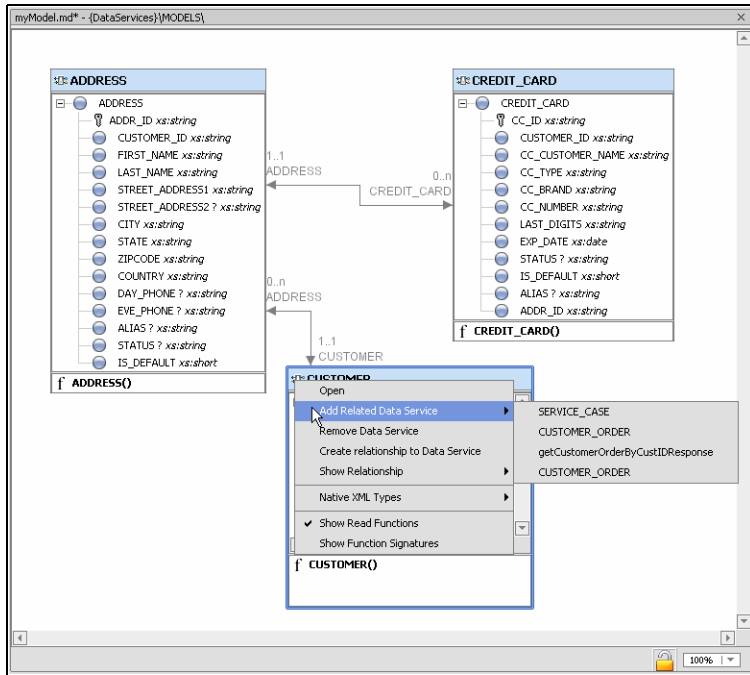
Figure 5-5 Adding Related Services



7. Mouse over to the related data service that you want to add to your model diagram. For this example perform this operation twice, adding both related data services to your model.

Once you have done this, you should automatically see the relationships between these three data services (Figure 5-6). (If not, try selecting the Show Relationship command for the Address data Service.)

Figure 5-6 Automatically Inferred Relationships Between Physical Data Sources



As described previously, relationship lines are graphical representations of *relationship declarations* and *navigation functions*.

There is a role at each end of a relationship. Initially, role names simply reflect their respective data service. [Table 5-7](#) details the model diagram’s services, roles, and cardinality of the model diagram, shown in [Figure 5-1](#).

Table 5-7 Relationship Declarations in Sample Model’s Data Services

data service	Role Name	Role Number	Opposite Role data service	Current Role	Minimum Occurrences	Maximum Occurrences
Address	Customer	1	customer.xds	Address	1	1
	CreditCard	2	credit_card.xds	Address	0	<i>n</i>
Credit_Card	Address	1	address.xds	Credit_card	1	1
Customer	Address	2	address.xds	Customer	0	<i>n</i>

Displaying Relationships Automatically

In the Application pane 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, any existing relationships to other data services in the model will be created automatically.

The relationships shown in the example are based on automatically created navigation functions found in the respective physical data services (see [Table 5-8](#)).

Table 5-8 Navigation Functions in a Model's Data Services

data service	Returns	Navigation Function
Address	Customer, Credit_Card	getCustomer()
Customer	Address	getAddress()

Generated Relationship Declarations in Source View

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. The Customer data service getAddress() function is show in [Listing 5-1](#).

Listing 5-1 Customer Data Service getAddress() Navigation Function

```
import schema namespace t2 = "ld:DataServices/CustomerDB/ADDRESS" at
"ld:DataServices/CustomerDB/schemas/ADDRESS.xsd";

(::pragma function <f:function xmlns:f="urn:annotations.ld.bea.com"
kind="navigate" roleName="ADDRESS"/>::)

declare function f1:getADDRESS($pk as element(t1:CUSTOMER)) as
element(t2:ADDRESS) *
{
  for $fk in f2:ADDRESS()
  where $pk/CUSTOMER_ID eq $fk/CUSTOMER_ID
  return $fk
};
```

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. This is shown in the Properties Editor (Figure 5-9).

Figure 5-9 Property Editor for New Model Diagram

Relationship: ADDRESS(CUSTOMER) - CUSTOMER(ADDRESS)	
Role (1)	
role-name	CUSTOMER
target-DS	ld:DataServices/CustomerDB/CUSTOMER.ds
min-occurs	1
max-occurs	1
Role (2)	
role-name	ADDRESS
target-DS	ld:DataServices/CustomerDB/ADDRESS.ds
min-occurs	0
max-occurs	n

Modeling Logical Data

The major difference between a logical model and a physical model is that the logical model contains representations of at least one logical data service, in addition to physical data services. In practice

there are no constraints between creating models that contain mixtures of logical or physical data services, including data services which are themselves composed of logical data services.

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. For details see [“Updating Data Source Metadata” on page 3-67](#).

Building Data Service Relationships in Models

In model diagrams, a relationship is created by the gesture of drawing a line from one data service to another (see [Figure 5-1](#)). 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 relationship has several editable properties:

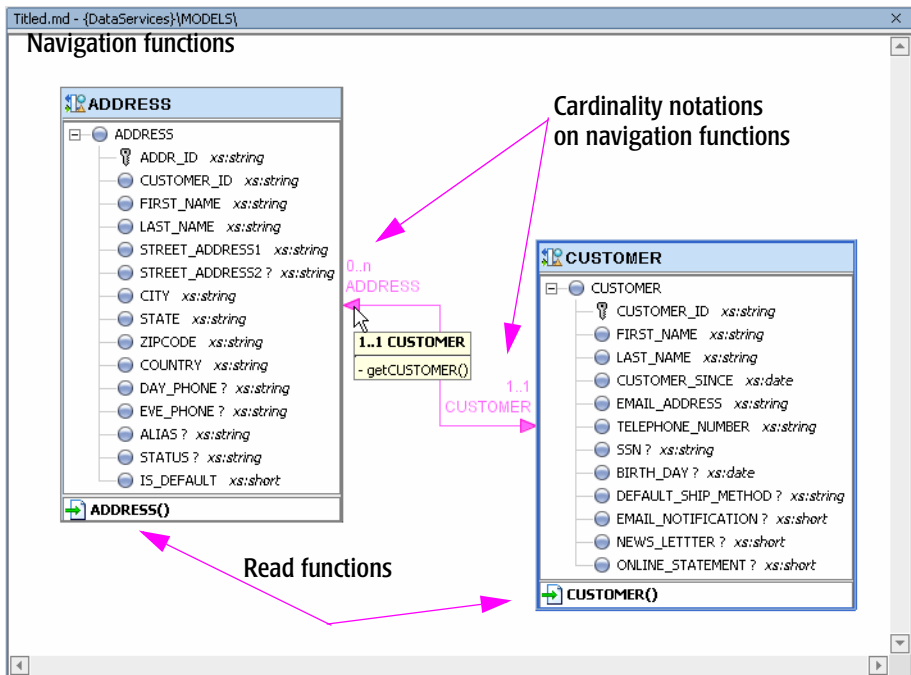
- **Cardinality.** Is the relationship zero-to-one (0:1 or 1:0) as in customer and promotion, one-to-one (1:1) as in customerID and custID, one-to-many (1: n) as in customers and orders, or many-to-many ($n:n$) as in customer orders and ordered items?
- **Direction.** Arrows indicate possible navigation 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 the binary relationship. They can be fully inspected in Source View for each data service. Navigation functions also appear as mouse-over text over each endpoint of the relationship line.

Direction, Role, and Relationships

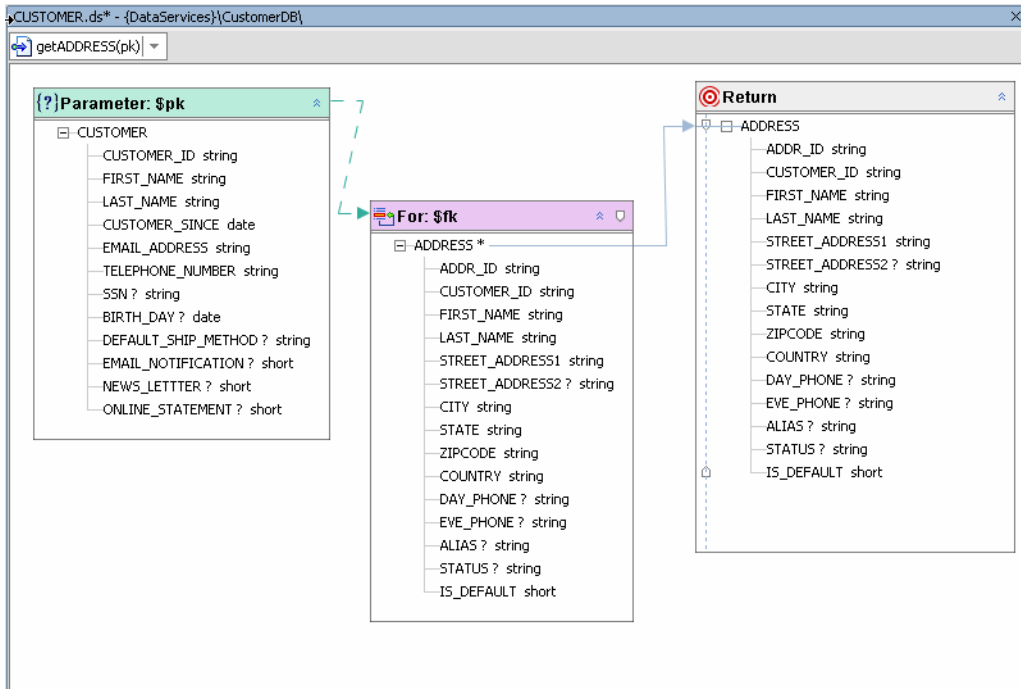
In a model diagram, each side of a relationship represents the role played by the adjacent data service. For example, in an ADDRESS: CUSTOMER relationship the end of the line near the customer is, by default, also called CUSTOMER. If you mouse over the role name, the opposite role name appears ([Figure 5-10](#)), as well as the name of the navigation function.

Figure 5-10 Model of Two Relational Data Services, ADDRESS and CUSTOMER



In the model diagram shown in [Figure 5-10](#) the ADDRESS role is accessed by CUSTOMER through its primary key, ADDR_ID. In the CUSTOMER data service the ADDRESS relationship has an automatically created function called getADDRESS(). Its role is to return address-type information about the holders of specific credit cards.

Figure 5-11 getAddress(pk) Function in the CUSTOMER Data Service



In the function shown in [Figure 5-11](#) the navigation function `getADDRESS(pk)` can take any `CUSTOMER` parameter input that includes a primary key `CUSTOMER_ID` and returns customer address information.

At the other end of the relationship in [Figure 5-10](#) is the `CUSTOMER` role, which supplies customer information to the `ADDRESS` data service also based on a unique customer ID.

In [Figure 5-10](#) notational arrows also identify cardinality notations. The `ADDRESS` role has a 0-to-1 cardinality with `CUSTOMER`, since your data source can have a customer without address information. The `ADDRESS` role has a 1-to-1 cardinality with `CUSTOMER`, since each `ADDRESS` must be identified with a single customer.

Cardinality notations can be modified in three places:

- Through your model diagram's Property Editor (see [“Model Diagram Properties” on page 5-23](#)).
- Through each data service Design View, using the Property Editor.
- Through Source View in each data service (not recommended).

Role Names

You can change role names to better express the relationship between two data services. This is particularly useful when there are multiple relationships between two data services.

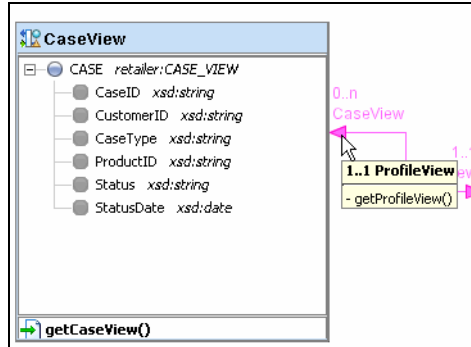
Take, for example: Customers and Orders. One relationship between these two data services would typically be 1: *n*, expressing two facts about the relationship:

- There is no limit to the number of orders a customer may have made.
- An order must be associated with one and only one customer.

By default, the role names would also be Customers and Orders. However you could change the role names to Supplies_Customer_Info and Orders_Array, respectively, to more precisely express the role of each side of the relationship.

A second relationship line could represent a different function, `getMostRecentOrder()`. This relationship would be 1:1 and the roles could be expressed as CustInfo and getOrder.

Figure 5-12 Mousing Over a Role Displays Its Navigation Function Name

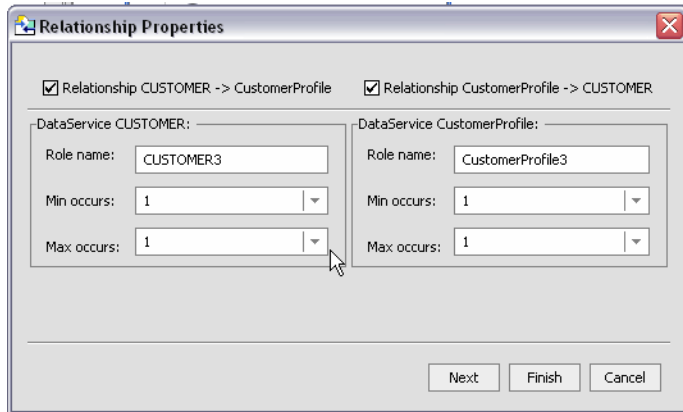


If you mouse over the end of a relationship line you will either see the navigation functions defined for that particular role (Figure 5-12) or a message indicating that no navigation functions have been defined.

Relationships

In a model diagram, drawing a line between two data services opens the Relationship Wizard.

Figure 5-13 First Dialog of Relationship Wizard



The wizard allows you to specify:

- Direction
- Role name
- Cardinality

Then, for each data service, you can additionally specify:

- Join conditions
- Parameters

When you are done you will have created a fully functional navigation function.

For an example and additional details see [“Adding a Relationship to Your Data Service”](#) on page 4-11.

With a few minor exceptions the Relationship wizard works the same when invoked in a model diagram as it does when you add a relationship to an existing data service.

Working with Model Diagrams

This section describes some of the common operations you will use when working with model diagrams.

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.

Table 5-14 Data Model Options

Scope	Command	Meaning
Data Model	Add Data Service	<p>Allows you to add one or several data services in your application to the current model diagram. The Add command brings up a file browser from which you can select a data service.</p> <p>Alternatively, you can drag data services from the Application pane into the model either individually or in groups (press the Ctrl key to select non-contiguous data services from your application).</p> <p>In the case of relational-based data services, dragging multiple data services into a model diagram at the same time will create relationships between the data services, if any exist. The relationships, of course, are based on primary/foreign key relationships that are available through imported metadata.</p> <p>Note: If a data service is already represented in your diagram, dragging will have no effect.</p>
	New Data Service	Allows you to create a new data service. After selecting a name and physical location for the data service (. ds) file using a browser, the service is created and placed on the diagram.
	Select All Nodes	Select all nodes in the model diagram.
	Generate Report	Creates either a Summary or Detail report describing the data services in the model, their bilateral relationships, and a description of each data service. See “Generating Reports on Your Models” on page 5-20 .
	Find Data Service	Locates a data service within your model. See “Locating Data Services in Large Model Diagrams” on page 5-19 for details.
Data Service	Open	Opens the currently selected data service in Design View (see “Creating a Data Service” on page 4-8). Alternatively, double-click on the data service representation.

Scope	Command	Meaning
	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 Data Service	Removes the selected data service from the model diagram. Alternatively, use the Delete key. Note: This operation does not affect the underlying 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” on page 4-13.)
	Show Relationship	Optionally displays/hides relationship lines associated with the currently selected data service. Click a relationship name in the sub-menu to select/deselect the display of its relationships.
	Show/Hide Native XML Types	Optionally displays/hides native types for elements representing physical objects associated with simple data types. Example: VARCHAR (25).
	Show/Hide Read Functions	Display/hides read functions associated with the data service.
	Show Function Signatures	Displays/hides full read function signatures such as: <code>getAddress() as element(Address)</code>
Relationship line	Remove Relationship	Removes the relationship from the diagram without affecting the underlying data service.
	Delete Relationship	Removes relationship notations in each respective data services and removes the relationship line from the model diagram.
	Show/Hide Role Name	Displays/hides the role name assigned to each side of the relationship.

Scope	Command	Meaning
	Show/Hide Cardinality	Displays/hides the cardinality of each side of the relationship. Only relationships between relational sources typically display cardinality.
XML type	Various	XML types can be edited in your model diagram. For important editing information see “Editing an XML Type” on page 4-24.

Creating Relationships in Model Diagrams

You can create additional relationship notations in model diagrams in several ways:

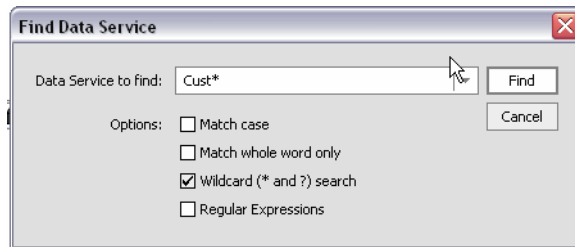
1. By drawing a line between two data services in your model diagram.
2. By right-clicking on a data service representation and selecting **Add Related Data Service**. Then select a data service from the sub-menu. The related data service will appear in the diagram along with a relationship line.
3. By selected a data service already in the model. Right-click on your data service and select **Create Relationship to Another Data Service**. Then, from the dropdown list in the resulting dialog, choose the data service to which you want to create a relationship. This will create a relationship line between the two data service representations.
4. By editing in Source View.

In the cases of options 1 and 2, above, the Relationship wizard will appear. The wizard is fully described in [“Adding a Relationship to Your Data Service” on page 4-11](#). Note that in the model diagram you do not have the option of changing the names of each side of the relationships since this has already been defined by the line connecting the two data services.

Locating Data Services in Large Model Diagrams

You can locate data services in your model diagram using the **Find Data Services** option, available from the right-click menu in your model diagram. Alternatively, use **Ctrl-F** when your model diagram is in focus.

Figure 5-15 Find Data Service Dialog Box



Options include the ability to:

- Match case
- Restrict search to whole words only
- Restrict the search to regular expressions

Wildcard character (?) and string (*) search is available.

Nodes matching the search criteria are highlighted and the model diagram view changes to show the first matching node.

Searches made during the current session can be retrieved using the drop-down combination listbox and entry field.

Generating Reports on Your Models

You can generate summary and detailed reports on the current model using the right-click Generate Report menu option, available from the title bar of your model. There are two types of reports: Summary and Detailed.

- **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
 - Owner (if any)
 - Comment (if any)
 - Date created
 - Date last modified

- **Detail 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 (known as 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

Creating a Model Report

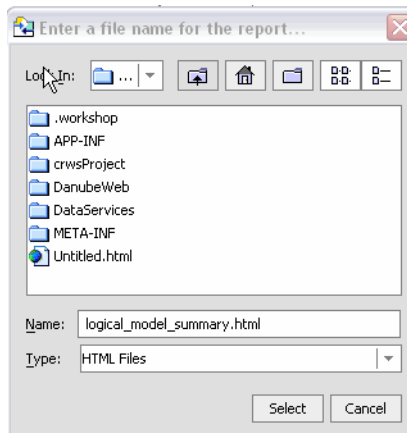
When you choose the Create a Model Report right-click option you are asked to select a name for the HTML document that is generated. By default, the name of the summary report is:

```
<model_name>_md_summary.html
```

and the name of the detail report is:

```
<model_name>_md_detail.html
```

Figure 5-16 Model Report Generator Dialog Box

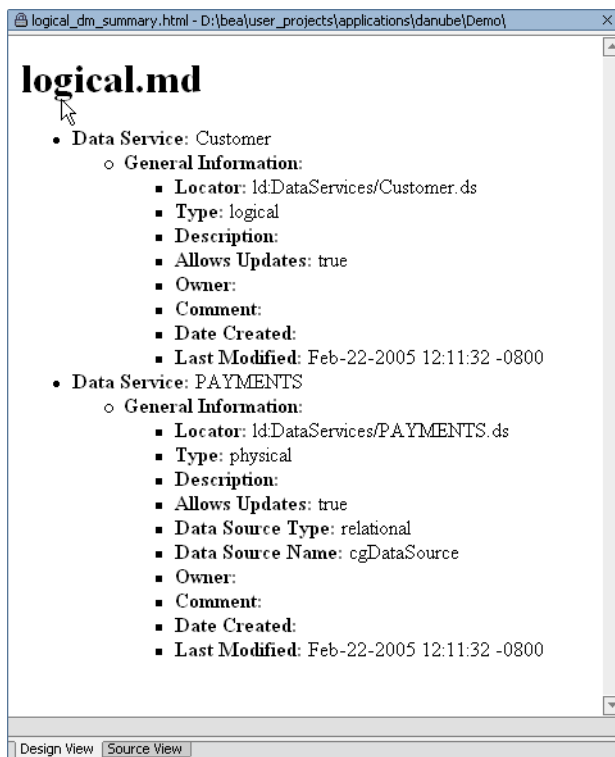


You can save the report to any location in your application ([Figure 5-16](#)) including to a new folder.

Model Report Format

The model report is in HTML format. When you initially run your report it opens in a WebLogic Workshop pane in HTML. A source tab is also available ([Figure 5-17](#)).

Figure 5-17 Sample Summary Model Report



Note: Print your report from any browser or application that supports HTML printing.

Zoom Mode

For larger models you can use a display-only zoom option, available in the lower right-hand corner of your model diagram (Figure 5-19). When in zoom mode an “lock” icon appears, indicating that Zoom mode is active and the model is read only.

Editing XML Types in Model Diagrams

You can edit any data service XML type represented in your model diagram. For XML type options see “Editing an XML Type” on page 4-24.

Model Diagram Properties

Properties both reflect and define relationships created in the model diagram. [Table 5-18](#) describes data model properties based on scope: data service, relationship, navigation functions, and XML type.

Table 5-18 Notable Data Model Properties

Scope	Property	Settings	Comments
Data Service			Properties described in “Managing Your Data Service” on page 4-26.
Relationship	<i>data service1(Role 1)</i> - <i>data service2(Role 2)</i>	Read only	Shows names of the related data services and their respective roles.
	Role (1)		Provides information on Role 1.
	role-name	Editable text	
	target data service	Read only	Name of data service1.
	min-occurs	Drop down, editable	Minimum occurrences can be blank, 0, 1, or <i>n</i> .
	max-occurs	Drop down, editable	Maximum occurrences can be blank, 0, 1, or <i>n</i> .
	Role (2)	See above.	Same settings as Role (1).
Navigation function	Name	Read only	
	Return Cardinality	Read only, 1 or *	Returns single type or an array
return type			See “Editing XML Types and Return Types” on page 2-16.

How Changes to Data Services and Data Sources Can Impact Models

A model diagram is dependent on its components including physical data, logical data, and relationships, all of which are subject to change outside the model itself.

Changes in a qualified name or deletion of a data service or changes in the underlying data can all cause a data model to become an incorrect representation of data services and their relationships.

A model diagram is revalidated when:

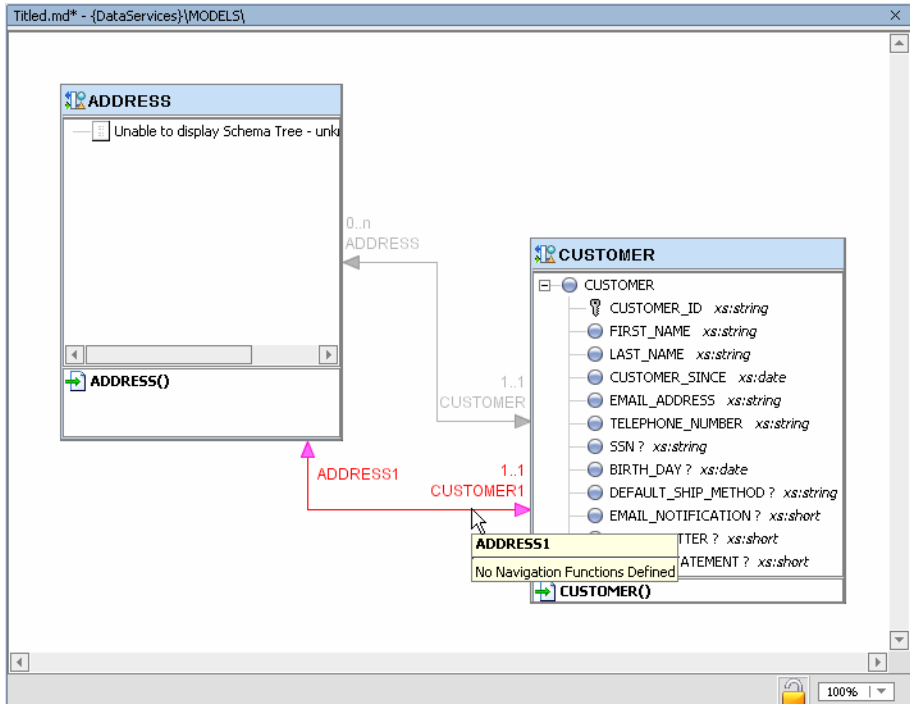
- it is opened or regains focus
- when the application is saved
- when metadata is updated

You can also use the Property Editor to correct a qualified name reference or to delete a stale reference. See [“Model Diagram Properties” on page 5-23](#) for details.

How Metadata Update Can Affect Models

Updating metadata will remove any manually created relationships between affected data services. In your model diagram this change is represented by the relationship line, appearing in red. In such cases, you will need to recreate the relationship with the newly updated data services.

Figure 5-19 Relationships Invalidated by Metadata Update Appear in Red



Modeling Data Services

Working with the XQuery Editor

BEA Aqualogic Data Services Platform (DSP) services provide a framework for creation and maintenance of functions that access and transform available data. You can use the XQuery Editor to create such functions.

A valid query function is always associated with a return type. In Source View a return type is described for each function. It typically matches the XML type — or schema — that defines the shape of your data service.

Once created, your query functions can be called by client applications. Details on the various methods of invoking DSP functions can be found in the Data Services Platform *Client Application Developer's Guide*.

You can also use the XQuery Editor to create standalone, ad hoc queries that can be run in Test View (see [Chapter 7, “Testing Query Functions and Viewing Query Plans”](#)).

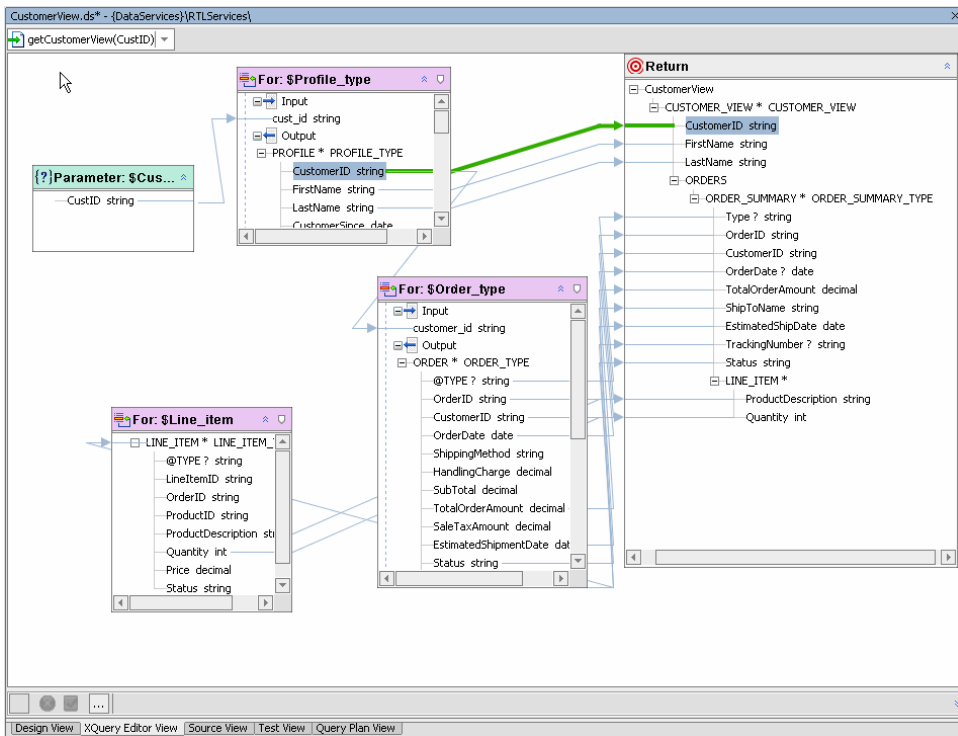
Topics discussed in this chapter include:

- [Role of the XQuery Editor](#)
- [Key Concepts of Query Function Building](#)
- [Managing Query Components](#)
- [Working With Data Representations and Return Type Elements](#)

Role of the XQuery Editor

Using the XQuery Editor you can create query functions using an intuitive, drag-and-drop approach. During the creation process you can easily move back and forth between the editor to Source View.

Figure 6-1 Sample Parameterized Function in the XQuery Editor



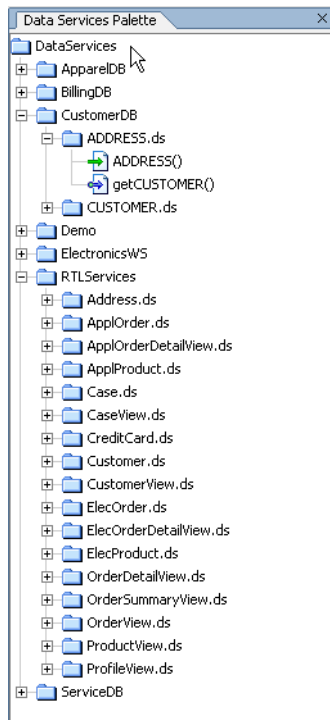
The XQuery Editor relies on data services functions for the metadata necessary to represent various types of data. (For detailed information on importing metadata see [Chapter 3, “Obtaining Enterprise Metadata”](#).)

Also see in the Data Services Platform [Samples Tutorial Part II:](#)

- Lesson 18: Building XQueries in XQuery Editor View
- Lesson 19: Building XQueries in Source View

A data service may represent a physical data source or it may represent logical data that has previously been created. Data service and custom XQuery library functions are both represented from the Data Service Palette (Figure 6-2), a WebLogic Workshop pane available when XQuery Editor View is active.

Figure 6-2 Data Service Functions Available to the RTL Sample Application



Notice in [Figure 6-2](#) that there are two different type of function representations: Functions represent by a straight (green) arrow are *read functions*, while functions represented by a more stylized (blue) arrow are *navigation functions*.

Essentially you create a query function by:

- Dragging in data representations from the Data Service Palette to the XQuery Editor work area.
- Identifying conditions, parameters, functions, and expressions that for your query.
- Associating elements with a return type.

As you work graphically you are automatically creating an XQuery in Source View.

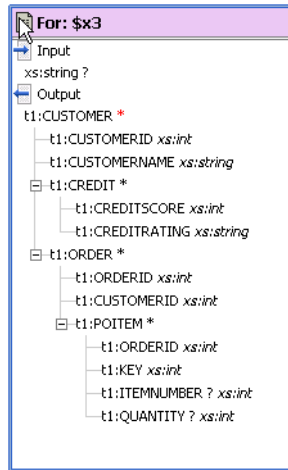
Once created, you can execute your function using Test View (see [Chapter 7, “Testing Query Functions and Viewing Query Plans”](#)). When you execute your query function, underlying data sources are accessed and the results appear. If you have appropriate permissions, data can be updated directly after the query is run.

Data Source Representations

Metadata representations of source are available to the XQuery Editor from the Data Service Palette. The Data Service Palette lists available data services and their read and relationship functions. Any such function can be dragged into the XQuery Editor work area where it will be transformed into a for clause.

Read functions and Web services often have input parameters. For example, the logical data service Customer (`customer.ds`) can be represented in the XQuery Editor by its read functions: `getCustomer()` and `getPaymentList()`. If you drag the `getCustomer()` item from the Data Service Palette to the XQuery Editor, the source representation shown in [Figure 6-3](#) appears in the work area.

Figure 6-3 Data Service Function From the Data Service Palette



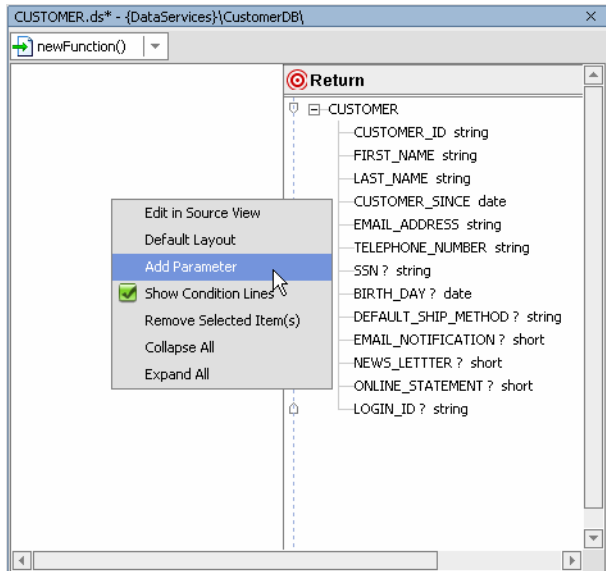
In some cases you may want to use a physical or logical data source representation several times in a query.

See [Chapter 3, “Obtaining Enterprise Metadata”](#) and [Chapter 4, “Designing Data Services”](#) for details on creating physical and logical data services.

XQuery Editor Options

When you create a new function in your data service and then click on the name of your new function ([Figure 6-1](#)), you will automatically be placed in the XQuery Editor. Alternatively, click the XQuery Editor View tab and select your function from the drop-down menu. Initially your XQuery will have only a return type, assuming that your data service is associated with an XML type (see [“Associating an XML Type”](#) on page 4-23).

Figure 6-4 Right-click Menu Options in the XQuery Editor



Several right-click menu options are available when you click in any unoccupied part of the work area.

Option	Meaning
Edit in Source View	Opens Source View to the section containing the currently selected function.
Default layout	The elements in the XQuery Editor are rearranged according to a pre-established formula including docking the return type to the right side of the work area.
Add Parameter	Adds a simple or complex parameter to your work area. Complex parameters require you to select a schema file and global type. See “Parameter Nodes” on page 6-20 .
Show Condition Lines	Hides/displays lines that identify conditions such as where clause predicates. By default condition lines are shown.
Remove Selected Item(s)	Deletes selected items from the work area.
Collapse All	Collapses all nodes in the work area including the return type.
Expand All	Expands all nodes in the work area.

Creating a New Data Service and Data Service Function

Creating a data service — as you will if you follow the steps in this section — is a good way to get the feel of what it is like to work with the XQuery Editor, as well as other aspects of data services. For example, through Source View you can quickly see how changes in the XQuery Editor are translated into XQuery code. Similarly, any changes you make in Source View will be immediately reflected in the XQuery Editor work area. (See [Chapter 8, “Working with XQuery Source.”](#))

The Goal

The goal of this exercise is to quickly create a logical data service from scratch, including creating an XML type for your data service, using the XQuery Editor. You create a logical data service by first building up a return type from several physical data services and then making that the type of your data service.

Note: The easiest way to change something you have done in the XQuery Editor is to use the Edit → Undo command (or Ctrl-Z). Since before saving your application you will be able to undo any number of previous steps, it is often preferable to use Undo rather than redrawing mappings, zone settings, or conditions, since these actions all modify the underlying source.

Setting Up Your Application

Using DSP sample data, the following steps illustrate one way to create a logical data service, including its return type.

Importing Your Metadata

In this section you will create a new application and DSP project and import the data source metadata sufficient to create the necessary physical data services.

1. Create a new Data Services; you can name it myLogical. (For details on creating DSP projects and applications see [“DSP-Based BEA WebLogic Projects” on page 2-2.](#))
1. Right-click on the automatically-created project entitled myLogicalDataServices.
2. Select Import Source Metadata. (If your Idplatform samples server is not already running you will need to start it before importing source metadata.)
3. Select Relational as the data source type.
4. Select all objects in cgDataSource from the drop-down list of available relational data sources.
5. Select the CUSTOMER table from the RTLCUSTOMER database.

6. Add it as a selected database object.
7. Repeat steps 5 and 6 for CUSTOMER_ORDER table in the RTLAPPLOMS database.
8. Click through the remaining options in the wizard to create two new data services.

Creating Your Logical Data Service

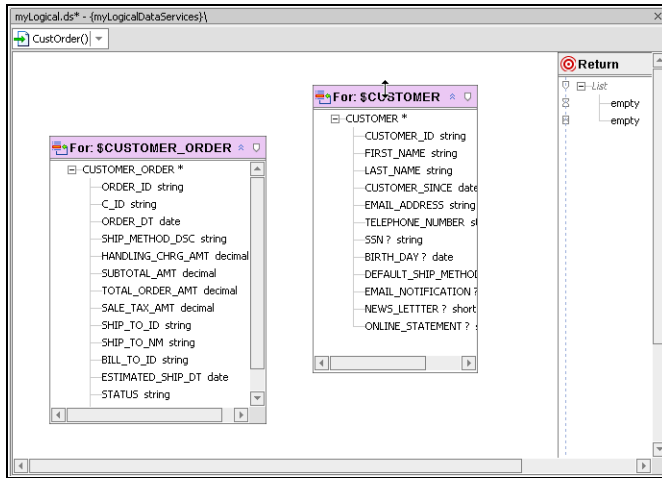
In this section you create a logical data service that provides client applications with the ability to retrieve customer-order information. In this section you will:

- Create a new data service and a new function.
- Add functions that represent source data (in this case customers and orders).
- Build up your return type using graphical gestures. This is where the master-detail arrangement of your returned data is defined.
- Modify return type zones to reflect nested for statements. This supports the nesting of all order details for a particular customer under that customer.
- Create your join conditions through a graphical gesture.

Here are the specific steps involved:

1. Right-click again on the myLogicalDataServices project and choose New → Data Service.
2. Name the data service myLogicalDS, then click Create. At this point your data service has no XML type.
3. Click on the titlebar of your new data service; select Add function. Name your new function CustOrder. Enter the XQuery Editor by clicking on the newly assigned name.
4. From your Data Service Palette drag the CUSTOMER() and CUSTOMER_ORDER() functions into the XQuery Editor work area (Figure 6-5).

Figure 6-5 XQuery Editor With Two Data Sources and an Empty Return Type



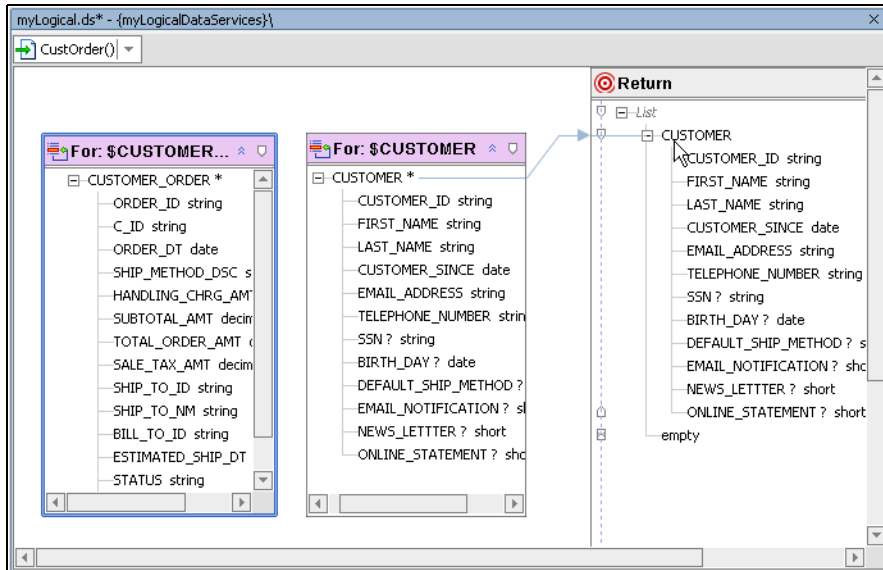
The existence of the two incomplete for clauses, \$CUSTOMER and \$CUSTOMER_ORDER, is accounted for by the return type's list of empty elements.

Next you need to populate the return type. In this case CUSTOMER_ORDER should be set up as a child of CUSTOMER so that information will be return in the following shape:

```
Customer1
..
  Order1
..
  Order2
..
Customer2
..
```

5. Holding down the Ctrl key map the CUSTOMER* element in the CUSTOMER for node to the topmost empty element in the return type.

Figure 6-6 Return Type After An Induced Mapping of the Customer For Node

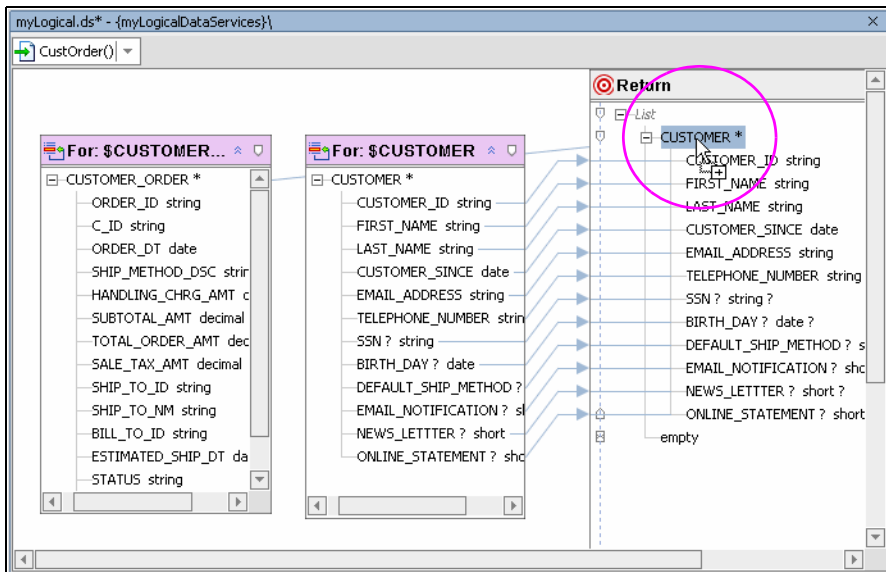


6. In your return type right-click on the new CUSTOMER root element and select Expand Complex Mapping. This maps all the elements in your CUSTOMER node to corresponding elements in your return type.

As this document must also list each customer's orders, you will need to create a second for statement. One way to do this is to simply add the CUSTOMER_ORDER type as a subordinate to CUSTOMER, as shown in the next step.

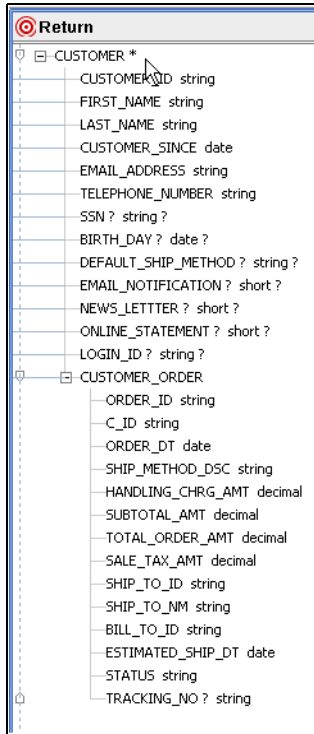
7. Holding down Shift+Ctrl keys select the root element in the \$CUSTOMER_ORDER for node and drag it over the CUSTOMER root element in your return type.

Figure 6-7 Append Mapping of the \$CUSTOMER_ORDER to the Return Type



The CUSTOMER_ORDER elements will appear as subordinate to CUSTOMER.

Figure 6-8 Subordinate Node Added to the Return Type



If you try to run a query at this point it will fail for several reasons:

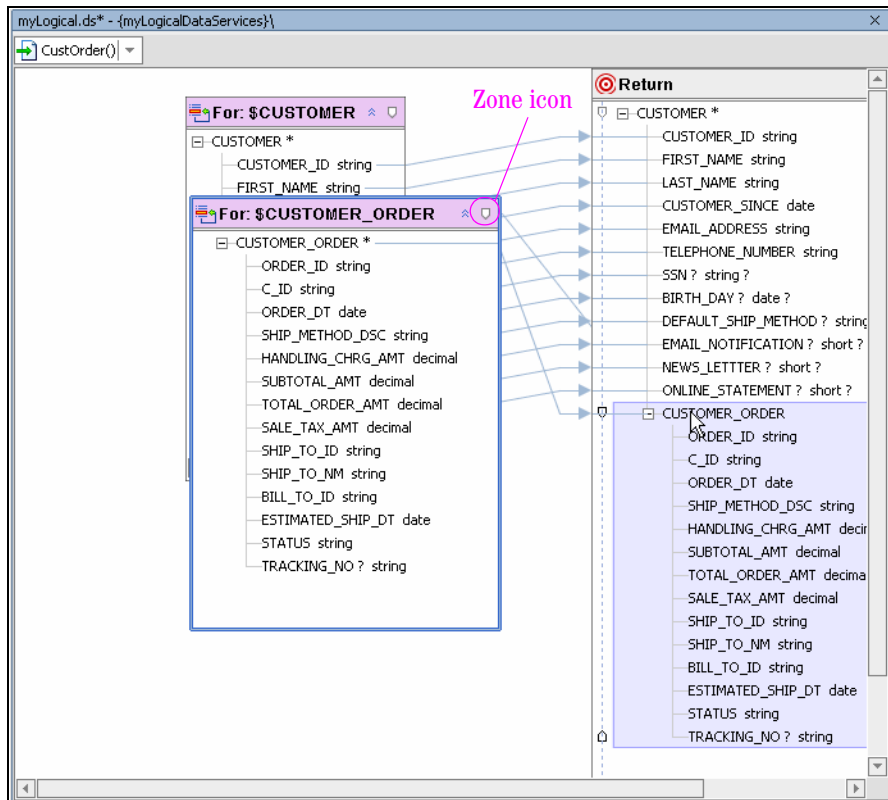
- Your data service has no associated XML type (schema).
- Your project (or application) needs to be build to create the proper SDO infrastructure.
- No where clause connecting the customer ID keys in the two data source representations has been created.
- The master-detail structure of the document has not been created.

Similarly, if you attempt to map source elements to CUSTOMER_ORDER, you will not be successful. This is because the implicit assumption behind the mapping of a complex element is that all the child elements are mapped to the return type.

These issues are resolved through the steps that follow.

- In your CUSTOMER_ORDER node select the *zone* icon (see Figure 6-9) and drag it over the CUSTOMER_ORDER element in your return type. (Notice that now when you mouse over the CUSTOMER_ORDER note, only the subordinate CUSTOMER_ORDER node is highlighted.) This action creates an inner zone in your return type which in source translates into an inner for clause for your query. An inner zone corresponds to the *detail* part of a relational *master-detail* ordering. (For more information on return type zones see “Setting Zones in Your Return Type” on page 6-50.)

Figure 6-9 Creating a Zone Supporting CUSTOMER_ORDER



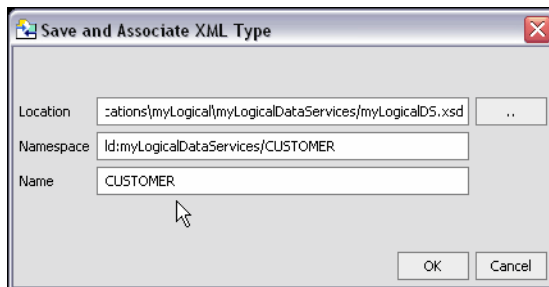
- Create a join between your two data source representations by dragging the CUSTOMER_ID element in the CUSTOMER node to the C_ID element in the CUSTOMER_ORDER node. A green line connecting the two elements appears.

Creating Your Data Service's XML Type, Building Your Application, and Running Your Query

The default name for your new schema matches the name of your data service; the default namespace is the *qualified name (qname)* of the root element of your return type.

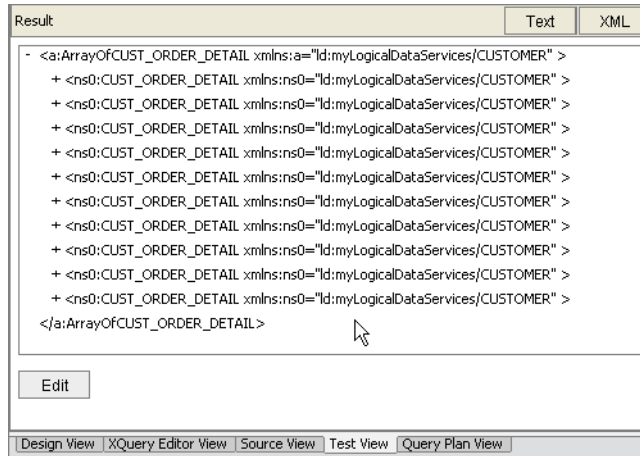
1. Click on the titlebar of your return type and select Save and Associate XML Type from the right-click menu. In order to complete this operation you need to provide the location of your new schema file, its namespace, and a name for the root element in your return type. In each case a default setting is provided, as shown in [Figure 6-10](#).

Figure 6-10 Save and Associate XML Type Dialog



2. Since the proposed qualified namespace of your new XML type is identical to the qualified name of your CUSTOMER data service, a type conflict will occur if you try to set your XML type to the return type. The solution is to modify either the namespace or the root name. Change the root name from CUSTOMER to CUST_ORDER_DETAIL. This will also change the root name of your return type and complete the association.
3. Build your project (or application).
4. Execute your query through Test View. Results should show customer orders nested for each customer. (See partial results in [Figure 6-11](#).)

Figure 6-11 Test Results



Although there are several ways to go about accomplishing the same task, it is also important to be aware that there were points along the way where an effort to build or deploy your application would not have been successful because the query or the return type was not fully formed. Thus the order in which steps are accomplished is often important.

Key Concepts of Query Function Building

The following terms and concepts are introduced in this section:

- [Data Sources](#)
- [Source Schemas and Return Types](#)
- [XQuery Editor Components](#)
- [The Distinct By node represents a single distinct by clause.](#)
- [Setting Expressions](#)
- [Mapping to Return Types](#)
- [Modifying a Return Type](#)

Data Sources

DSP supports multiple data sources including:

- RDBMS (relational database management systems)
- Web services
- Java functions
- Delimited files (such as spreadsheets)
- XML files

For details on importing data source metadata from these sources into DSP-based projects see [“Obtaining Enterprise Metadata.”](#)

Source Schemas and Return Types

The XQuery Editor uses XML schema representations as:

- **XML type.** An XML schema that describes the structure of a physical or logical data source.
- **Return type.** The return type of a function. In the XQuery Editor the return type contains information necessary to support customized queries in terms of the ordering of information returned from the query.

For more information see [“XML Types and Return Types” on page 4-7.](#)

XQuery Editor Components

Using the XQuery Editor, query functions can be built up graphically using a combination of graphical gestures and functions, including:

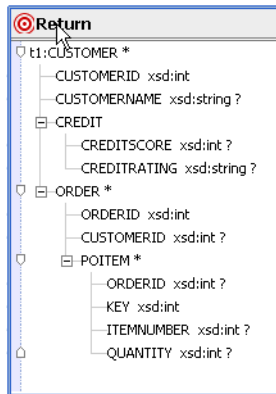
- Standard XQuery for and let clauses
- XQuery constructs such as where and order by
- XQuery extensions such as group by and if-then-else
- Standard and user-defined XQuery functions
- Physical and logical data source references

The following topics describe XQuery clauses as rendered in the XQuery Editor. (For information on the XQuery engine used by DSP and specific uses of XQuery in Source View see the Data Services Platform *XQuery Developer’s Guide*. This document also contains references to the most up-to-state XQuery W3C specifications.)

Return Type Node

Query functions always map to a single return type. If your data service is associated with a return type, that type will appear in the Return node.

Figure 6-12 Sample Return Type



The return type can be thought of as extending the XML type to in support of:

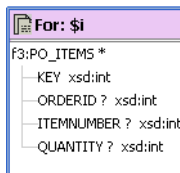
- If-then-else constructs using the right-click Conditional operation.
- Zones (see [“Setting Zones in Your Return Type”](#) on page 6-50).

When you click on a simple element in the return type, the expression on that element’s constructor appears.

For Clause Nodes

A for clause node represents a named XQuery for clause construct. For and let clause nodes are always based data service functions.

Figure 6-13 Sample For Statement Node



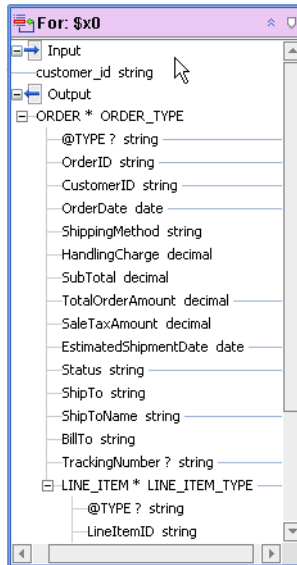
By default, whenever you add a data service to the XQuery Editor work area, it is represented in a for node. The for node typically represents looping over a query function using either:

- a variable reference
- an expression

Parameterized Input

A for node, representing a parameterized query function, provides both Input and Output sections. As you would expect, parameters are mapped to the Input elements while Output elements either serve as input to other nodes or to the return type.

Figure 6-14 Example of Parameterized For Node



For and Let Node Options

Several options are available when you right-click on the title of a for node.

Option	Meaning
Rename	Brings up a dialog which allows you to rename your node. Names cannot contain spaces.
Delete	Removes the node and any mappings in or out of the node from the work area.

Option	Meaning
Convert to let/for Clause	Changes the clause from a for to a let or from a let to a for. This operation is reversible.
Go to definition	Opens the data service that is represented by the node. The data service is opened to the current function in XQuery Editor View. However, if the function represents a physical data service (termed <i>external</i> in Source View), then the function definition in Source View appears. You can use the back arrow to return to your initial data service.
Relationship Functions	Relationship functions associated with the data service are listed. Selecting a relationship function allows your for or let node to serve as input for the relationship. See “Adding Relationship Functions to an Existing Data Service” on page 6-24 for an illustration and code sample.
View Source	Shows the source underlying the currently selected node.

Converting Between For and Let Clauses

For and let clauses (see [“Let Statement Nodes” on page 6-20](#)) have many interchangeable characteristics.

The following code shows the conversion of the `DataServices/RTLServices/Case/getCaseByCustID()` function expression from a for clause:

```
declare function ns1:getCaseByCustID($cust_id as xs:string) as
element(ns0:CASE)* {
  for $x0 in ns1:getCase()
  where $cust_id eq $x0/CustomerID
  return $x0
};
```

to a let clause:

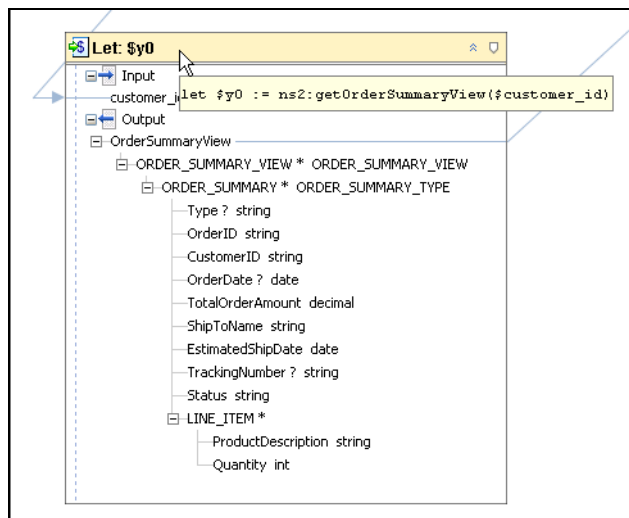
```
declare function ns1:getCaseByCustID($cust_id as xs:string) as
element(ns0:CASE)* {
  let $x0 := ns1:getCase()
  where $cust_id eq $x0/CustomerID
  return $x0
};
```

Let Statement Nodes

A let clause binds a sequence of elements (graphically contained in a node) to a variable that in turn becomes available to the FLWR expression.

Options available for use with for clauses are also available for let clauses. See [“For and Let Node Options”](#) on page 6-18.

Figure 6-15 Let Statement in the RTLServices/OrderSummaryView Data Service



When examining a let clause, you can read the assign string (:=) as the “*be bound to*”. For example, in the following let clause:

```
let $x := (1, 2, 3)
```

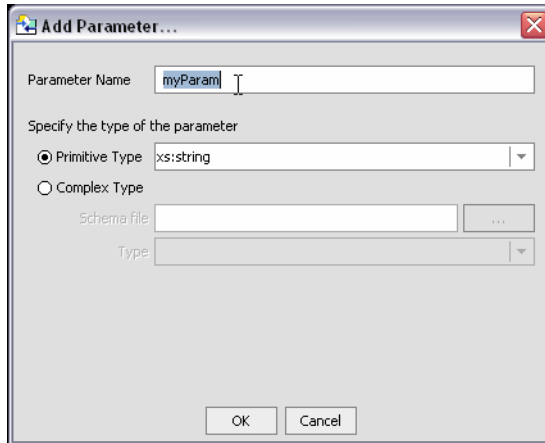
Can be read as "let the variable named *x* be bound to the sequence containing the items 1, 2, and 3."

See also [“Converting Between For and Let Clauses”](#) on page 6-19.

Parameter Nodes

Parameter nodes enable you to associate a parameter with a for or let clause. Parameter nodes are created in the XQuery Editor work area ([Figure 6-4](#)). Three right-click menu options are available: Rename, Delete, and View Source.

Figure 6-16 XQuery Editor Add Parameter Dialog Box

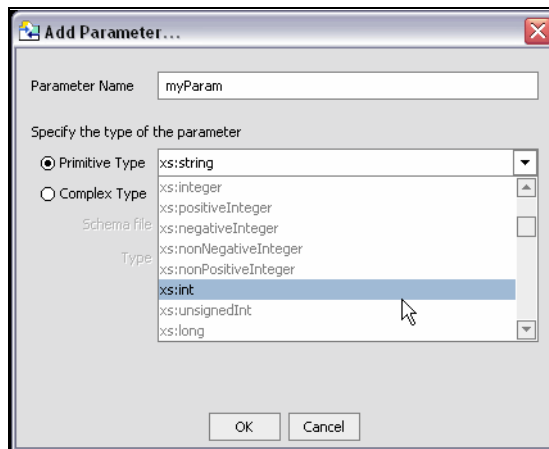


You can create parameters that range from simple data elements to elements of any complexity.

Adding a Parameter Requiring a Simple Type

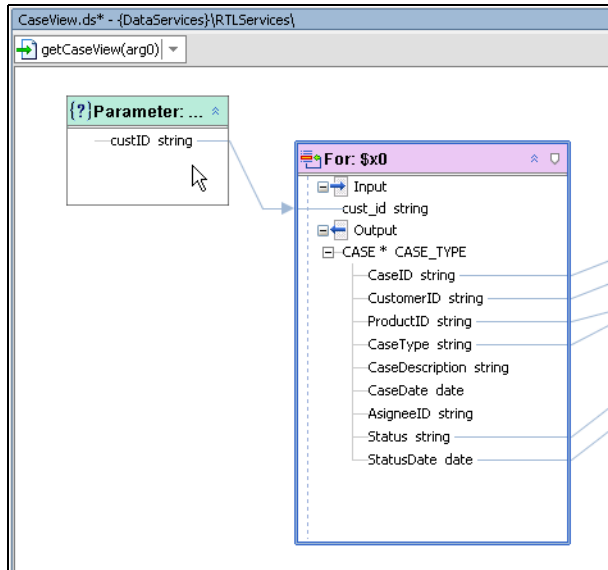
You can create a simple type parameter by selecting the type from the drop-down list and clicking Ok.

Figure 6-17 Setting a Simple Parameter Types



The act of *mapping* a parameter to a for or a let node containing an Input creates a parameterized query and also establishes a where condition. In [Figure 6-18](#) the customer_id string parameter is dragged over the element in the ADDRESS node which is to be associated with the parameter through a where clause.

Figure 6-18 Parameter Mapped to a For Node



The corresponding Source View code highlights the parameter:

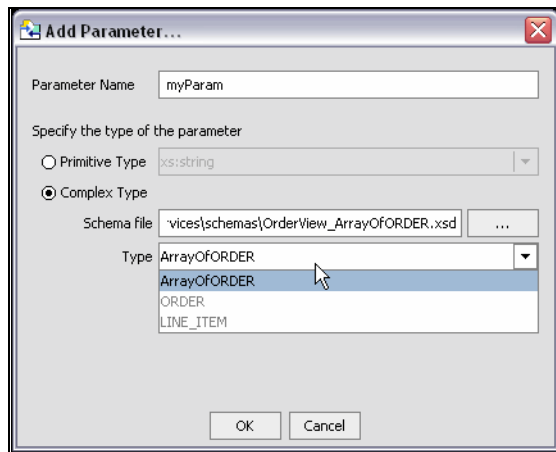
```
declare function ns5:getCaseView($custID as xs:string) as
element(ns6:CaseView) {
<ns6:CaseView>
{
  <CASE_VIEW>
  <CASES>{
    for $Case in ns7:getCaseByCustID($custID)
    return <CASE>
      <CaseID> {fn:data($Case/CaseID)} </CaseID>
      <CustomerID>{fn:data($Case/CustomerID)} </CustomerID>
      <CaseType> {fn:data($Case/CaseType)} </CaseType>
      <ProductID> {fn:data($Case/ProductID)} </ProductID>
      <Status> {fn:data($Case/Status)} </Status>
      <StatusDate> {fn:data($Case/StatusDate)} </StatusDate>
    </CASE>
  }
  </CASES>
  </CASE_VIEW>
}
</ns6:CaseView>
};
```

When you invoke your function from an application — or execute your function in Test View — you will supply a value for your parameter.

Adding a Complex Parameter

Complex parameters are established by identifying a schema and a global element. Some schemas have only one global element.

Figure 6-19 Setting a Complex Parameter Type

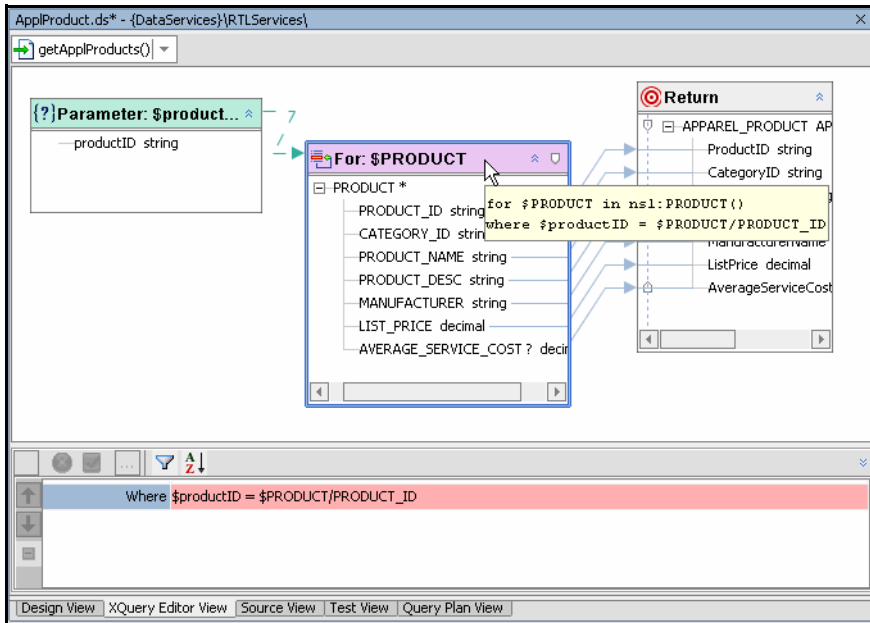


The resulting parameter can be associated with any for or let node. See also [“Parameterized Input”](#) on page 6-18.

Using the Parameter Dialog to Create a WHERE Clause

You can use the parameter dialog to create a where clause condition simply by dragging the simple or complex parameter over an element in a for or let clause. In [Figure 6-20](#) the newly created parameter productID is mapped to PRODUCT_ID. Since the \$PRODUCT for node is selected, the where clause is in scope.

Figure 6-20 Parameterized Where Clause



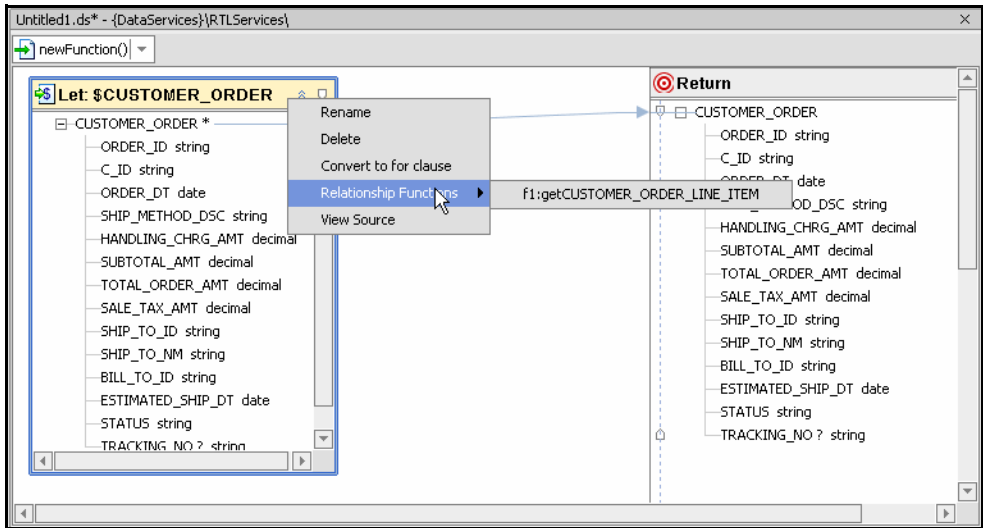
Adding Relationship Functions to an Existing Data Service

There are several ways to add relationship functions to existing data services. The recommended way is to use the right-click menu option available from for and let nodes, since this will create more appropriately nested clauses than simply dragging a relationship function from the Data Service Palette into the work area.

For example, if you want to create a logical data service that was a union of customer order and order line items, you could start with a customer order and add the related line item data.

Figure 6-21 takes the RTLApp DataServices/ApparelDB/CUSTOMER_ORDER() function and shows the process of adding the related getCUSTOMER_ORDER_LINE_ITEM() function.

Figure 6-21 Adding a Relationship Function



The function initially appears as:

```
declare function tns:newFunction() as element(ns30:CUSTOMER_ORDER9) * {
  for $CUSTOMER_ORDER in ns28:CUSTOMER_ORDER()
    return $CUSTOMER_ORDER
};
```

Adding the relationship function changes it to:

```
declare function tns:newFunction() as element(ns30:CUSTOMER_ORDER9) * {
  for $CUSTOMER_ORDER in ns28:CUSTOMER_ORDER()
    for $CUSTOMER_ORDER_LINE_ITEM in
      ns28:getCUSTOMER_ORDER_LINE_ITEM($CUSTOMER_ORDER)
    return $CUSTOMER_ORDER
};
```

To complete this simple example you would need to add elements from the related data service to your return type and complete your mappings, as well as any transformations.

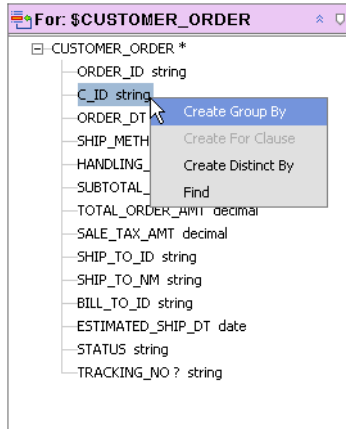
Group By Statement Nodes

The Group By node represents a single group by clause with zero or more grouping expressions. The top part of the Group By node defines variables available to the generated group by expression. The bottom part defines the grouping expression itself.

Group By expressions are often used with aggregation functions such as grouping customers by total sales. A for or let clause supports multiple group by elements.

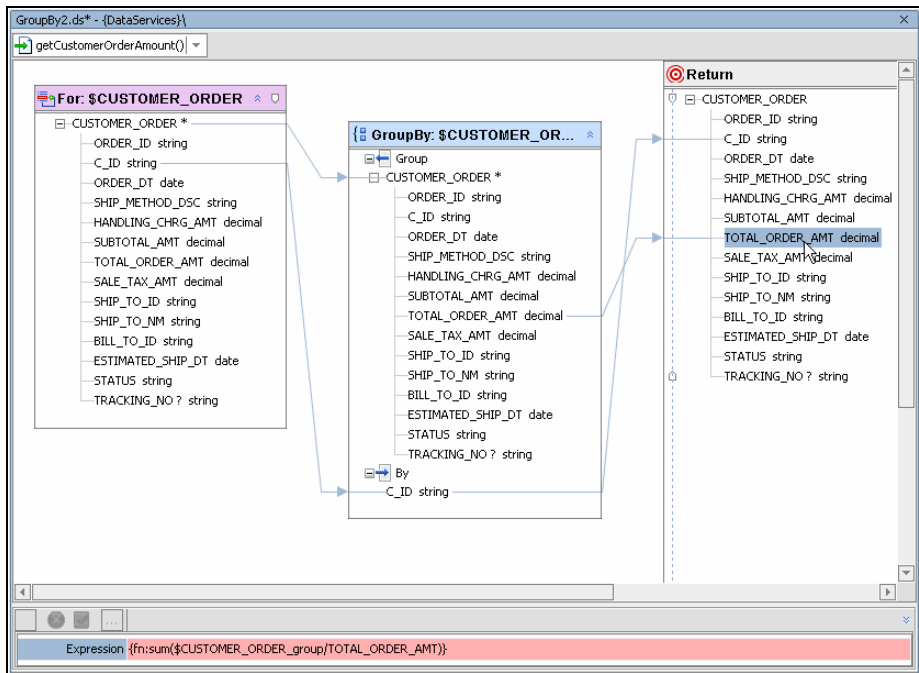
You can generate a Group By node by right-clicking on any element in a for or let node and selecting Create Group By from the right-click menu.

Figure 6-22 Creating a Group By Expression



In [Figure 6-22](#) output will be grouped by the C_ID (customer ID) element. Once a GroupBy node is created, mappings to target objects — such as the return type — are done through the new node.

Figure 6-23 Projecting Total Orders Grouped by Customer ID



The default name of the group by node will be a unique name based on the local name of the for/let node. Thus the `CUSTOMER_ORDER` for clause becomes the basis for `CUSTOMER_ORDER_group0`. Group By nodes cannot be renamed from the XQuery Editor.

As seen in [Figure 6-23](#), any node mappings are automatically transferred to the Group By node.

The resulting source is:

```

declare function tns:getCustomerOrderAmount() as
element(ns5:CUSTOMER_ORDER) * {
  for $CUSTOMER_ORDER in ns6:CUSTOMER_ORDER()
  group $CUSTOMER_ORDER as $CUSTOMER_ORDER_group by $CUSTOMER_ORDER/C_ID
  as $C_ID_group

  return
<ns5:CUSTOMER_ORDER>
  <ORDER_ID></ORDER_ID>
  <C_ID>{fn:data($C_ID_group)}</C_ID>
  <ORDER_DT></ORDER_DT>
  <SHIP_METHOD_DSC></SHIP_METHOD_DSC>
  <HANDLING_CHRG_AMT></HANDLING_CHRG_AMT>
  <SUBTOTAL_AMT></SUBTOTAL_AMT>

```

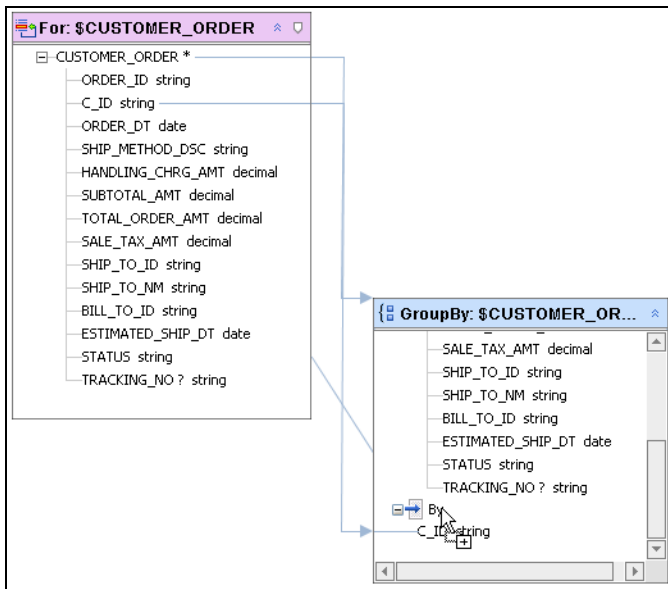
```
<TOTAL_ORDER_AMT>{ fn:sum ($CUSTOMER_ORDER_group/TOTAL_ORDER_AMT) }</TOTAL_ORDER_AMT>  
  <SALE_TAX_AMT></SALE_TAX_AMT>  
  <SHIP_TO_ID></SHIP_TO_ID>  
  <SHIP_TO_NM></SHIP_TO_NM>  
  <BILL_TO_ID></BILL_TO_ID>  
  <ESTIMATED_SHIP_DT></ESTIMATED_SHIP_DT>  
  <STATUS></STATUS>  
  <TRACKING_NO?></TRACKING_NO>  
</ns5:CUSTOMER_ORDER>  
};
```

If you delete a Group By node any mappings from the parent node will need to be redrawn.

Creating Multiple Group By Elements

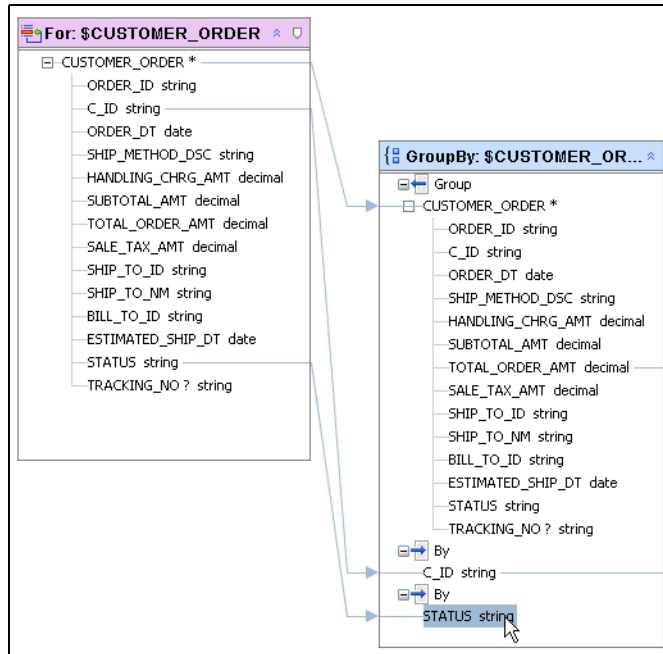
In the example you can add additional grouping expressions simply by dragging new elements over the “By” separator, (Figure 6-24).

Figure 6-24 Adding a Second Group By Element



The act of dragging the element over an existing group by expression, adds a second group by expression, as shown in Figure 6-25.

Figure 6-25 The New Group By Expression Element



The effect of adding the second group by in the above example is to group total orders by their status value.

```

<ORDER_ID/>
  <C_ID>CUSTOMER0</C_ID>
  <TOTAL_ORDER_AMT>1173.2</TOTAL_ORDER_AMT>
  <STATUS>CLOSED</STATUS>
</ns0:CUSTOMER_ORDER5>
<ns0:CUSTOMER_ORDER5
xmlns:ns0="ld:DataServices/ApparelDB/CUSTOMER_ORDER5">
  <ORDER_ID/>
  <C_ID>CUSTOMER0</C_ID>
  <TOTAL_ORDER_AMT>436.3</TOTAL_ORDER_AMT>
  <STATUS>OPEN</STATUS>
</ns0:CUSTOMER_ORDER5>
<ns0:CUSTOMER_ORDER5
xmlns:ns0="ld:DataServices/ApparelDB/CUSTOMER_ORDER5">
  <ORDER_ID/>

```

Using Multiple Group Nodes

You can create additional multiple Group By expressions to enable creation of logic such as:

*Group by A, then
Group by B*

In order to do this you need to introduce an additional for or let clause to establish the parent-child structure that will support the needed logic.

To creating a second-level group by:

1. Creating a child for clause. Right-click on the root element in your primary group by node and select Create For Clause.
2. Create a new zone (see [“Setting Zones in Your Return Type” on page 6-50](#)) in your return type. There are several ways to do this. One is to Add a Child Element.
3. Right-click on the new child element and select Mark as Zone.
4. Set the zone of your new for clause to the new child element. Do this by dragging the zone symbol over your new child element. You will know you have succeeded when the newChildElement displays an array symbol.

```
NewChildElement * empty
```

And, when you mouse over the new child element, it will be highlighted, indicating that it is a zone unto itself.

5. Create a group by right-clicking on the grouping element in the new for clause node.
6. Replace the new child element by dragging the group by element over the new child element while holding down the control key. This effectively overwrites the element with the group by expression.
7. Save and associate your new return type so that it become the XML type of your data service.

You can use Test View to verify your work.

Distinct By Statement Nodes

The Distinct By node represents a single distinct by clause.

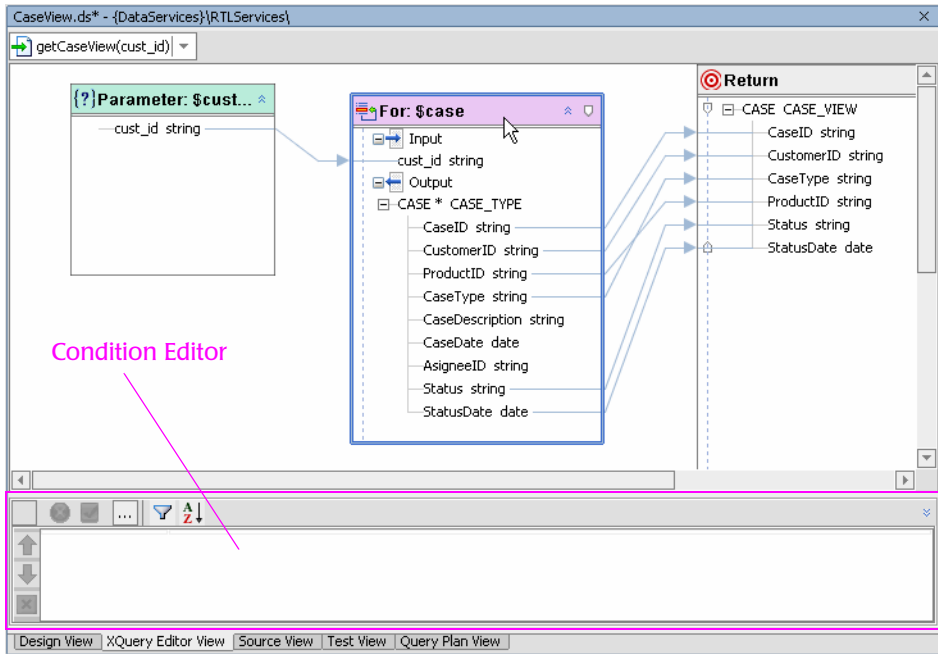
Distinct by is useful:

- When you what to return all distinct values for a particular element.
- When you want to perform functional operations on the result of a distinct by, such as the total number of distinct elements.

Setting Conditions

Several types of conditions can be graphically applied to for and let clauses. You can create these conditions using a multifunction editor that appears at the bottom of XQuery Editor work area. (Figure 6-26).

Figure 6-26 Multifunction Condition Editor



To add or modify constraints for a for or let node first select the node, then click anywhere in the multifunction editor. Everything but your selected expression will become unavailable, as indicated by the “grayed out” appearance of unselected objects.

Condition types are:

- where
- group by

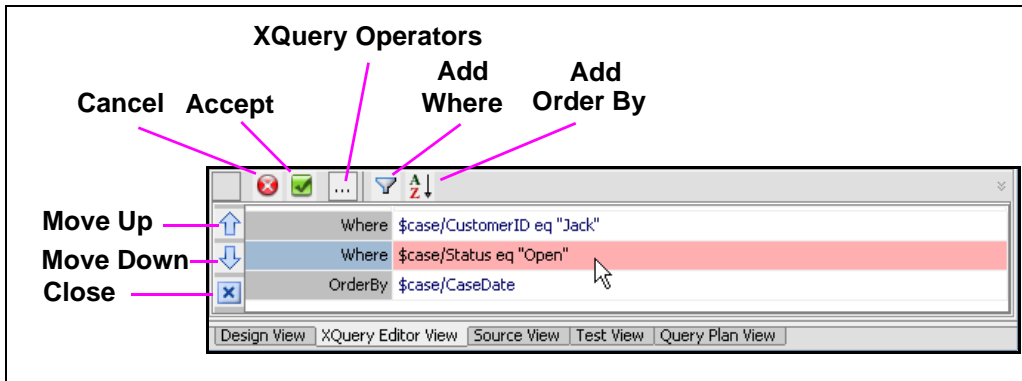
Figure 6-27 provides a closer look at the multifunction dialog which includes the ability to:

- Add any number of where or order by conditions.
- Edit a condition using the built-in line editor.

- Adjust the order in which the conditions are applied.
- Select XQuery operators from a drop-down list.
- Accept or cancel editing changes to a particular condition.
- Delete a where or order by clause.

Functions from the XQuery Function Palatte can be dragged into the multifunction box and then edited.

Figure 6-27 Detail of Multifunction Box



The Where Clause

The where clause places a condition on a for and/or let clause. A where clause can be any query expression, including another FLWR expression. The where clause typically filters the number of matches in a FLWR loop.

A common use of the where clause is to specify a *join* between two sources. For example, consider the following query:

```
<results>
{
for $x in (1, 2, 3), $y in (2, 3, 4)
where $x eq $y
return
    <matches>{$x}</matches>
}
</results>
```

The where clause in this query filters (or joins or *constraints*) the results that match two sequences specified in the for clause. In this case, the numbers 2 and 3 match, and the query returns the following results:

```
<results>
  <matches>2</matches>
  <matches>3</matches>
</results>
```

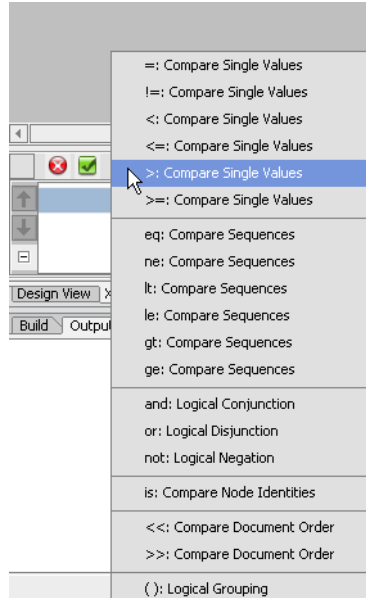
To effect this in the XQuery Editor you would select the for or let clause to which the where condition applies. Then, in the where condition field, you enter:

```
$x eq $y
```

You can type in the name of an element or drag it from the a node in the work area into the multifunction editor.

The eq XQuery operator can be entered directly or selected from the conditional pop-up list (Figure 6-28).

Figure 6-28 Conditional Operator Selection List



Here is a more complete example involving an XQuery function (see [“Using XQuery Functions” on page 6-35](#)). It involves finding all customers whose first name is Jack.

Using a Where Clause as a Filtering Device

The following example illustrates the use of a where clause in the multifunction editor:

1. Using the RTLApp sample application DataServices project create a new data service. Choose any name.
2. Select the Add Function option from the Data Service menu. Use any name.
3. Click on the new function name to enter the XQuery Editor.
4. From the Data Service Palette (View →Windows →Data Services Palette) select the CUSTOMER() function from the CustomerDB/CUSTOMER data service; drag it into the work area.
5. Associate the empty return type with the CUSTOMER elements by mapping the top node CUSTOMER element to the empty element in the return type while holding down the Control key (Ctrl-map).
6. In your return type select Expand Complex Mapping from the right-click menu associated with the top element in your return type (now CUSTOMER).
7. In the return type title select the Save and Associate right-click menu option associated with the return type title.

You can create a valid XML type (schema file) for your new data service by associating your return type with your data service. The name of the global element in your return type or the alias assigned to the namespace or both needs to be changed because there already is a schema named CUSTOMER that was based on the physical data source you started with. (For details on Save and Associate see [“Creating a New Data Service and Data Service Function” on page 6-7.](#))

If you change the name field CUSTOMER to CUSTOMER_WHERE and click Ok you will notice that the name of the complex element in your return type will change.

8. Click on the title bar of the Customer node. This highlights the multifunction editor.
9. Click on the Where clause icon ([Figure 6-31](#)). A field for the where clause appears.
10. Click on FIRST_NAME in the CUSTOMER node.
11. Add an equals operator following by “Jack” so that the entire clause appears as:

```
$CUSTOMER/FIRST_NAME eq "Jack"
```

Notice that the clause becomes red whenever your expression is invalid.
12. In Test View run your new function. Notice in your results that the where conditions are fulfilled.

See also [“Using the Parameter Dialog to Create a WHERE Clause”](#) on page 6-23.

The Order By Clause

The order by clause indicates output order for a given set of data.

Unless otherwise specified, the order data appears will follow the XML tree. This is known as the *document order*. The `order by` keyword indicates that the content should be sorted in ascending order by the identified element(s).

XQuery keywords such as descending are supported. For example, an XQuery can be written that orders the customers by last name in descending order:

```
for $customer in document('customers.xml')//customer
  order by last_name descending
return
  <customer>
    {$customer/first_name}
    {$customer/last_name}
  </customer>
...

```

In the XQuery Editor you would select the `for` or `let` clause to which the order by condition applies and in the order by condition field enter:

```
last_name descending
```

You can type in the name of an element or drag it from the work area into the multifunction editor [Figure 6-27](#).

Creating Join Conditions

Join conditions are represented as equality relationships in where clauses. Therefore you can create such an equality relationship by dragging and dropping the `eq` function onto a row in the Conditions tab and then selecting two source elements/attributes into the same row.

Using XQuery Functions

Data Services Platform contains a full set of built-in XQuery functions. Most XQuery functions in the XQuery Function Palette are standard XQuery functions supported by the W3C. However, there are several BEA-specific functions as well as several extensions to the language. (For details on the BEA implementation of the 1.0 XQuery engine see [XQuery Developer's Guide](#). For more detailed information on standard XQuery functions, see the W3C [XQuery 1.0 and XPath 2.0 Functions and Operators](#) specification.)

The functions available from the XQuery Functions palette help you create conditions around for and let clauses. XQuery functions can be used in several contexts:

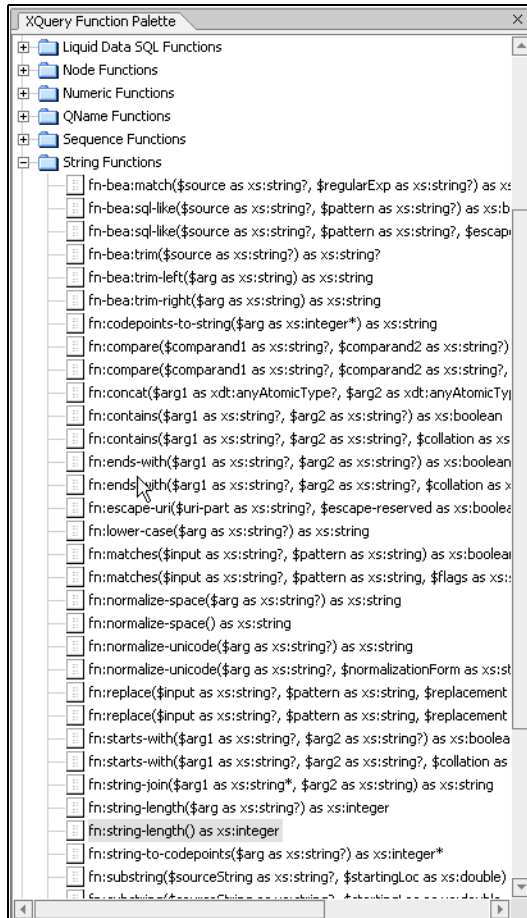
- In where clauses.
- As input to another XQuery function.
- Mapped to a target element.
- As input to a data service function.

Using XQuery Functions in Where Clauses

To create a where clause condition that filters customers a query returns you can follow these steps:

1. Follow steps 1-7 under [“Using XQuery Functions in Where Clauses” on page 6-36](#).
2. Click on the title bar of your CUSTOMER node.
3. Click the Where icon (see [“The Where Clause” on page 6-32](#)) to create a new where condition.
4. Open the XQuery Function palette (View →Windows →XQuery Function Palette).
5. Drag the fn:string-length() as xs:integer function from the XQuery Palette into the Where condition.

Figure 6-29 XQuery Palette



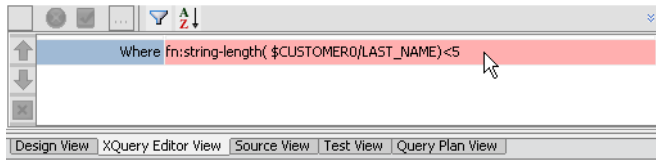
- Using the condition and expression built-in line editor highlight the function argument (\$arg).
- Click on \$CUSTOMER/LAST_NAME. The string appears as:

```
Where fn:string-length($CUSTOMER/LAST_NAME)
```

- Add <5 as the predicate so the string appears as:

```
Where fn:string-length($i/ORDERID)>5
```

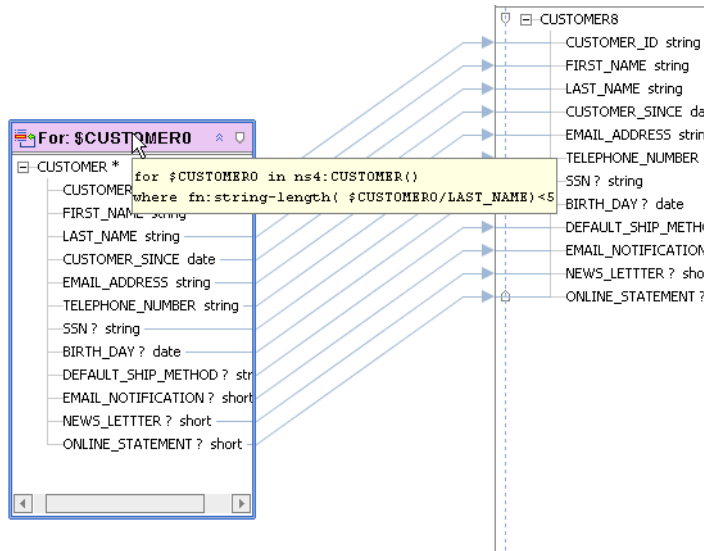
Figure 6-30 Editing an XQuery Function



9. Click the checkmark in the editor.

If you mouse over the title of your for clause, you can see that the condition has been associated with the fragment. You can also verify this change in source view.

Figure 6-31 Mouseover of Node Title Displays Its Conditions



When you run the function only records with LAST_NAMEs shorter than five characters will appear.

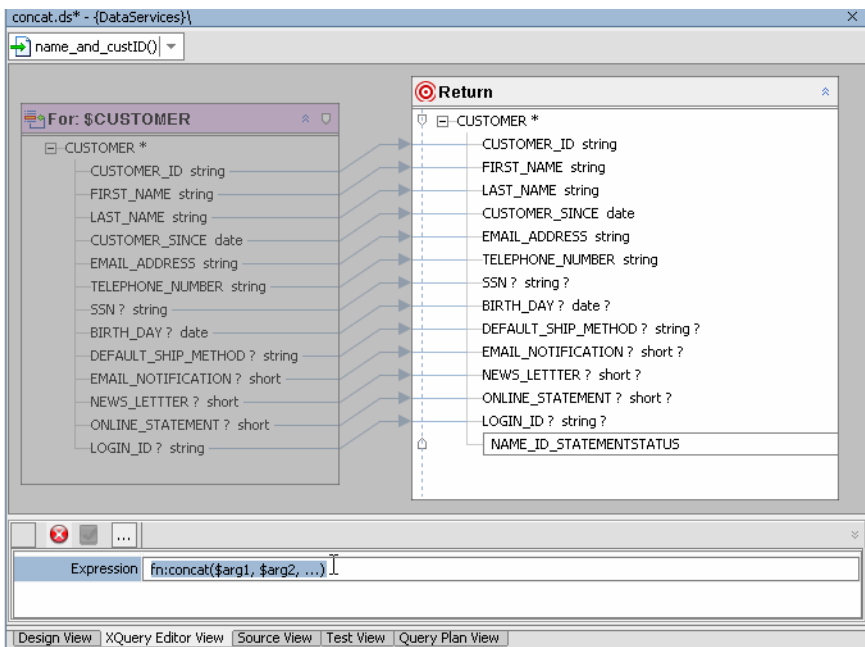
Automatic type casting generally ensures that input parameters used in functions and mappings are appropriate to the function in which they are used.

Transforming Data Using XQuery Functions

There are many transformational XQuery functions. In the following example the concat() function is used to quickly enrich data returned from a physical data service by adding functionality to the expression associated with an element returned by the function.

1. Follow steps 1-7 under “Using XQuery Functions in Where Clauses” on page 6-36.
2. Open the XQuery Function palette.
3. In the return type add a child element.
4. Rename it to NAME_ID_STATEMENTSTATUS. Initially no type is assigned to the element. This will be derived from the element or function mapped to it.
5. Click on your new element.
6. Then click in the expression field of the multifunction editor.
7. Drag the `fn:concat($arg1,$arg2,...)` function into the expression field.

Figure 6-32 Creating a Concatinated Element in the Return Type



8. Highlight the first argument (`$arg1`) and click on `FIRST_NAME` in the `$CUSTOMER` node. Similarly, select `$arg2` and click on `LAST_NAME`. Select the ellipses and then click on `ONLINE_STATEMENT`.

- The concat() function requires that all input parameters be of type string. Since ONLINE_STATEMENT of type short, it needs to be cast as a string. You can do this through the editor. Similarly, you can add spaces and a small legend to make the results more readable.

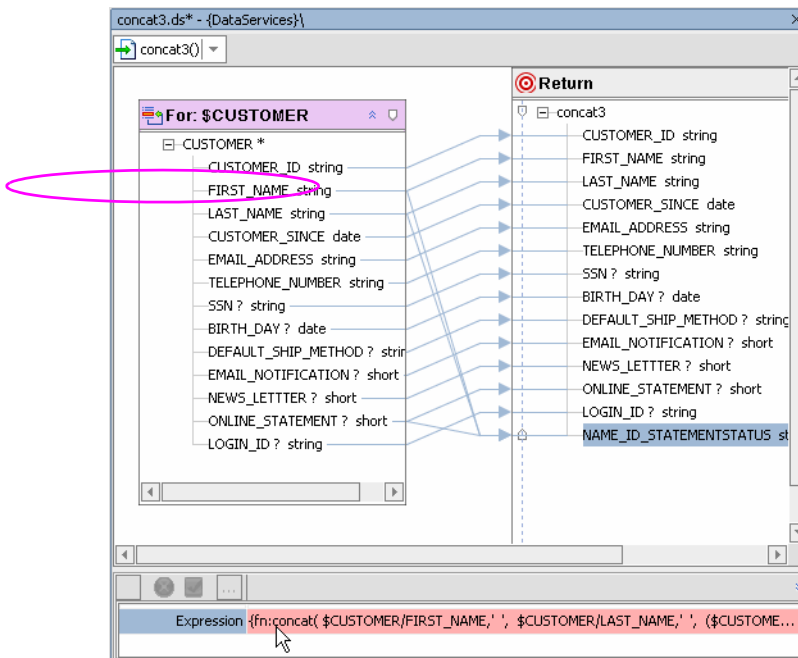
When your editing is complete, the XQuery function will appear as:

```
{fn:concat( $CUSTOMER/FIRST_NAME, ' ', $CUSTOMER/LAST_NAME, ' | ', ($CUSTOMER/ONLINE_STATEMENT cast as xs:string), ' (online=1; printed=0)')}
```

- Check the green box to accept your changes.
- Since you made changes to the return type schema, rebuild your application.

Notice also (Figure 6-33) how the new functionality is reflected in the source-to-target mapping.

Figure 6-33 XQuery Editor Work Area After Adding an Element Containing a Function to the Return Type



- When you run your program in Test View results will now include the following type of information:

```
<NAME_ID_STATEMENTSTATUS>
    Jack Black | 1 (online=1; printed=0)
</NAME_ID_STATEMENTSTATUS>
```

Setting Expressions

The Expression editor is most commonly used to edit return type expressions. For example, if the return type contains:

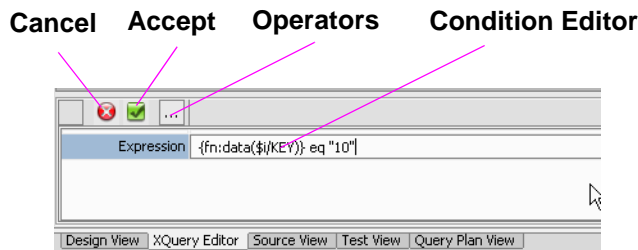
```
ORDERID xsd:int
```

The editor can be used to limit the scope of the expression to a single customer:

```
Expression>{fn:data($o/ORDERID)} eq "1001"
```

Prototypes of functions available from the XQuery Function Palette can be dragged into the editor or you can use the build-in line editor to enter them yourself.

Figure 6-34 Expression Editor



Operation of the Expression editor is similar to that for the multifunction box. When you select an element other workspace artifacts are grayed out. However, you can drag elements from any part of the work area into the Expression Editor.

XQuery operators are available as a drop down list, as shown in [Figure 6-28](#), or you can simply type them in.

Managing Query Components

If you think of selected *data elements* as nouns (what you want to work on), the *functions* as verbs (the action), then the *mapping* among the data elements creates a logical sentence that expresses the *query*.

Results and query performance can change significantly depending on how you:

- Map (or *project*) source data from one or more sources to the return type.
- Specify zones and other conditions (filter source data) and expressions (element level operations).

Although you can simply type in an XQuery and run it from Test View, the more common way to create a query is build it up through the following operations:

- Map simple or complex elements to the return type
- Define query parameters
- Transform information using built-in or custom functions
- Filter data using where or order by conditions
- Adjust expressions as required by business requirements

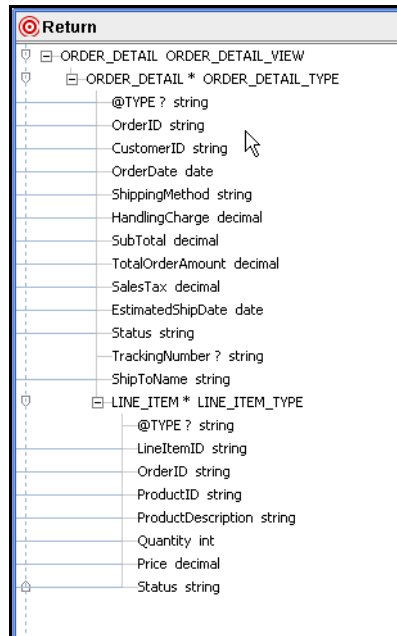
Note: Some operations are not deterministic. For example if a node has elements mapped to a return type, deleting the node before removing the mappings may create error conditions. Instead you can use Undo and then delete the mappings or you can make the necessary changes in Source View.

Working With Data Representations and Return Type Elements

Mapping elements involves establishing a visual relationship between data source elements and the return type or an intermediary node requiring input parameters.

There are two types of schema elements: *simple* and *complex*. *Complex elements* contain elements and/or attributes.

Figure 6-35 Expanded Schema Showing Complex and Simple Elements



To expand a complex element, click on the plus sign (+) to the left of its name. (If you double click on the name itself, you will enter edit mode.)

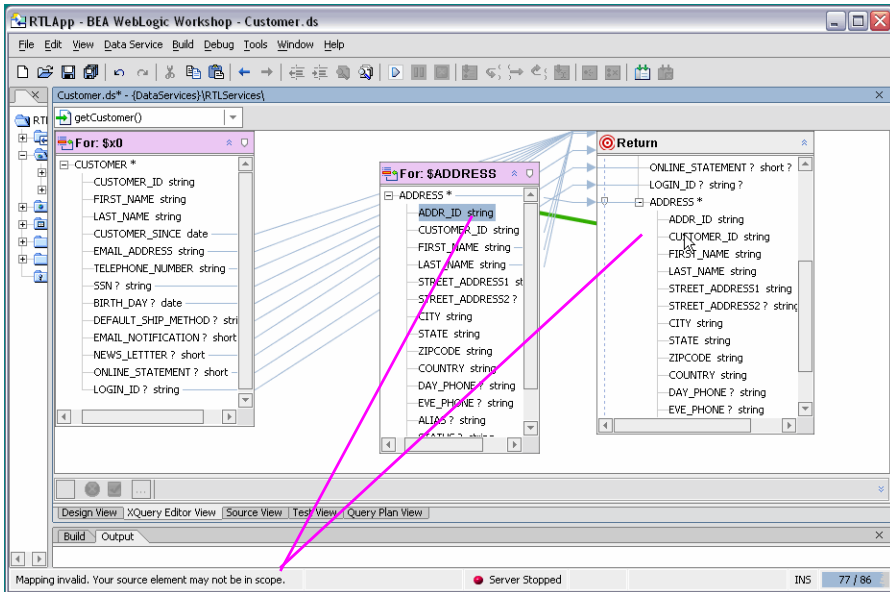
Mapping to Return Types

As shown in [“Creating a New Data Service and Data Service Function” on page 6-7](#), the XQuery Editor automatically generates queries based on graphical mappings into a return type.

The XQuery Editor supports two types of mappings: *value mappings* and *complex element mappings*. Value mappings map (assign) only the value of an element or attribute from a source to the value of its target element or attribute. Element mappings allow mapping source elements (simple or complex) to target.

In order to map an element to a return type, that element needs to be *in scope*. If the element you are attempting to map is not in scope, a message will appear indicated that the mapping is invalid (see [XX](#)). Invalid mappings occur whenever the underlying for or let statement would not be able to validly handle the association of the data element(s) with the return type schema.

Figure 6-36 Invalid Mapping Attempt Flagged by Alert



For more information on element scoping and other related issues see [“XML Types and Return Types” on page 4-7](#) and [“Editing XML Types and Return Types” on page 2-16](#).

Mapping Elements and Attributes to the Type

A questionmark symbol [?] next to an element name represents an optional element, meaning that it is not required by the query. Primary keys are never optional.

Complex Element Mappings to a Return Type

You can rapidly map complex elements from source to your return type. This known as an induced mapping is useful where all or part of the return type should match source representations.

There are many situations when you will find it convenient to map elements into your type, including:

- When you are creating a type from scratch.
- When you want elements individually mapped but it is easier to map complex elements, expand the mappings to include values, and then add or delete some mappings using right-click return type management commands.

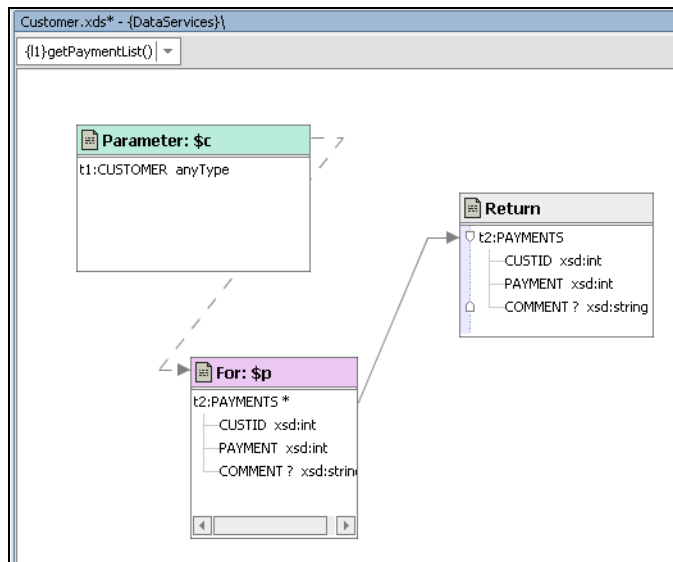
If the match is not exact, mapping a complex element to your return type will be appended.

There are several benefits of mapping or projecting elements:

- Manual one-to-one mapping of multiple elements is less often needed.
- The query is often easier to read.
- If the underlying structure of the complex element changes — an element is added, deleted, or an attribute is changed — the generated query does not change.

Figure 6-37 shows the results of the mapping of a complex element to a return type.

Figure 6-37 Example of Mapping of a Complex Element



Note: You cannot map multiple elements to a single target element.

Source-to-Target Mapping Options

Three source-to-return type gesture mappings are available — value mappings, overwrite mappings, and append mappings.

- **Value mappings.** Individual source node elements are individually mapped to simple elements or attributes in the return type. You can create a value or *simple mapping* by dragging and dropping elements from the source node to a corresponding target element in the return type. All elements may not need to be mapped, depending on the information you want in the XML document generated by your query function. However, special attention should be paid to

non-optional elements (those without adjacent question-marks (?)), since your query will fail if non-optional elements are not projected.

- **Overwrite mappings.** When you hold down the Ctrl key when mapping an element, the source element (and any children) will replace the target element (and any children). This gesture sometimes results in an *induced mapping*. An induced mapping occurs when a complex element in source is mapped to a comparable (exactly named) element in the return type. For example, you create an induced map when you drag and drop the CREDIT_CARD* element (root element in a source node) onto the CREDIT_CARD complex element in the return type.

The following code expresses the results of an induced mapping:

```
declare function tns:newFunction() as element(ns5:CREDIT_CARD)* {
    for $CREDIT_CARD in tns:getCreditCard()
    return
    $CREDIT_CARD
```

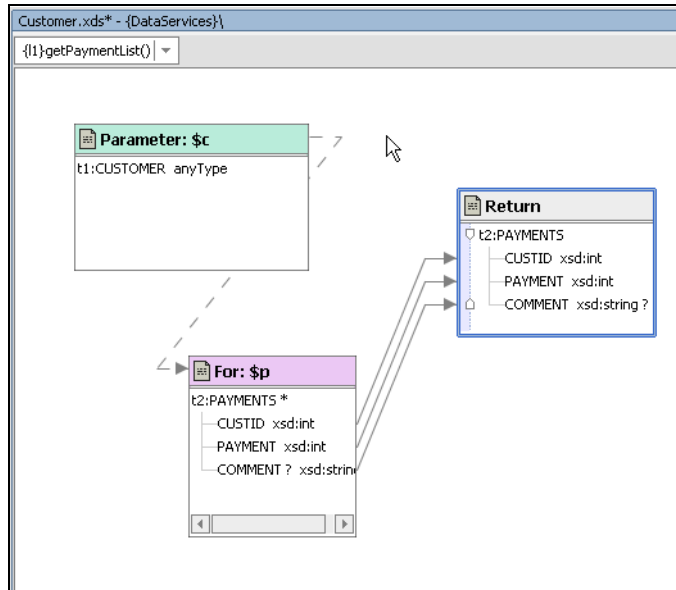
In many cases an induced mapping is insufficient either for further building your query function or running it. You can always expand an induced mapping by right-clicking on the element in the return type and selecting the only available option: Expand Complex Elements.

In the above case the source would be correspondingly modified:

```
declare function tns:newFunction() as element(ns5:CREDIT_CARD)* {
    for $CREDIT_CARD in tns:getCreditCard()
    return
    <ns5:CREDIT_CARD>
        <CreditCardID>{fn:data($CREDIT_CARD/CreditCardID)}</CreditCardID>
        <CustomerID>{fn:data($CREDIT_CARD/CustomerID)}</CustomerID>
        <CustomerName>{fn:data($CREDIT_CARD/CustomerName)}</CustomerName>
        <CreditCardType>{fn:data($CREDIT_CARD/CreditCardType)}</CreditCardType>
        <CreditCardBrand>{fn:data($CREDIT_CARD/CreditCardBrand)}</CreditCardBrand>

        <CreditCardNumber>{fn:data($CREDIT_CARD/CreditCardNumber)}</CreditCardNumber>
        <LastDigits>{fn:data($CREDIT_CARD/LastDigits)}</LastDigits>
        <ExpirationDate>{fn:data($CREDIT_CARD/ExpirationDate)}</ExpirationDate>
        <Status?>{fn:data($CREDIT_CARD/Status)}</Status>
        <Alias?>{fn:data($CREDIT_CARD/Alias)}</Alias>
        <AddressID>{fn:data($CREDIT_CARD/AddressID)}</AddressID>
    </ns5:CREDIT_CARD>
};
```

Figure 6-38 Expanded Complex Element



- **Append mappings.** You can append a source element and children (if any) to an element in a return type using the key combination of Ctrl+Shift. Click the source element, press the key combination of Ctrl+Shift and drag the element over an element in the return type. If the underlying element is highlighted, you can add the source as its child.

Note: Any changes you make to a return type should be propagated to your data services XML type using the Save and Associate XML Type right-click option, available from the return type titlebar.

Removing Mappings

You can delete mappings between elements by selecting the mapping line (link) and pressing Delete. Alternatively, use the Delete key.

Modifying a Return Type

The shape of the information returned by your query is determined by its return type. Using a combination of mapping techniques and return type options you can:

- Add or remove elements and attributes from your return type.
- Set up repeatable sections, known as zones.

You should only modify a return type if you intend to propagate the change to the data service's XML type using the Save and Associate XML Type command, described in [“Creating a New Data Service and Data Service Function” on page 6-7](#).

Modifying a Return Type

You can edit your return type by right-clicking any element. Editing options for a type in the XQuery Editor are somewhat different options described in [“Editing an XML Type” on page 4-24](#). For example, in a return type you can create zones automatically add for clauses to your query, allowing for a “master-detail” arrangement of results.

Warning: While it is possible to modify a return type and run a query in an ad hoc manner, problems will likely arise when your application calls a query with a mismatch between the return clause and the XML type of the data service.

[Table 6-39](#) describes notable return type editing options.

Table 6-39 Notable Return Type Options

Option	Meaning
Add Child Element	Creates a child element for the currently selected element.
Add Complex Child Element	Allows you to specify a schema and type for a new complex child element. By default, the type is the root element of the schema. If the schema has several global elements, however, you will first need to specify the element that you want to become the root.
Add Attribute	Creates an attribute for the selected element.
Make Conditional	<p>Inserts an element named Conditional above the currently selected element and clones the element (and children, if any).</p> <p>Conditional elements can be used in conjunction with if-then-else constructs. Transformational logic can then be developed through the XQuery Editor and mapped to the appropriate branch of the condition.</p>
Clone	<p>Duplicates the selected element (and children, if any) to the same level of the schema hierarchy.</p> <p>If you clone a simple element an unmapped, untyped element of the same name will be created.</p>

Option	Meaning
Mark as Zone / Remove Zone	Sets (or removes) a zone setting for the current element and its children (if any). If the elements are in a zone the query will return them in a master-detail arrangement. See “Setting Zones in Your Return Type” on page 6-50.
Delete	Deletes the selected element and any child elements or attributes.
Find	The Find dialog allows you to search for text strings in the return type with options to match case, match whole words only, use wildcards (*, ?), or regular expressions.

A special option, **Expand Complex Mappings**, is becomes available for use with Induced mappings. See [“Complex Element Mappings to a Return Type”](#) on page 6-44 for details.

There are several things to keep in mind when making changes to a return type:

- Changes to a return type should be propagated the your data service’s XML type.
- Changes to a return type through DSP components exist only in memory until you run the **File → Save All** command in WebLogic Workshop.
- Changes to a file using the **Save All** command cannot be reversed through **Undo**.

Adding a Complex Child Element

You can add a complex child element to a return type by selecting a schema and identifying a global element (a type). Complex child elements incorporate data service schemas (.xsd file) into the return type.

To add a complex global element to your return type:

1. Click on the element you want to be the parent of the complex element.
2. Right-click and select **Add Complex Child Element**.
3. In the dialog that appears navigate to the schema you want to use. If your schema only has one global element, then it will be automatically selected. Otherwise, you will need to pick which global element to use.

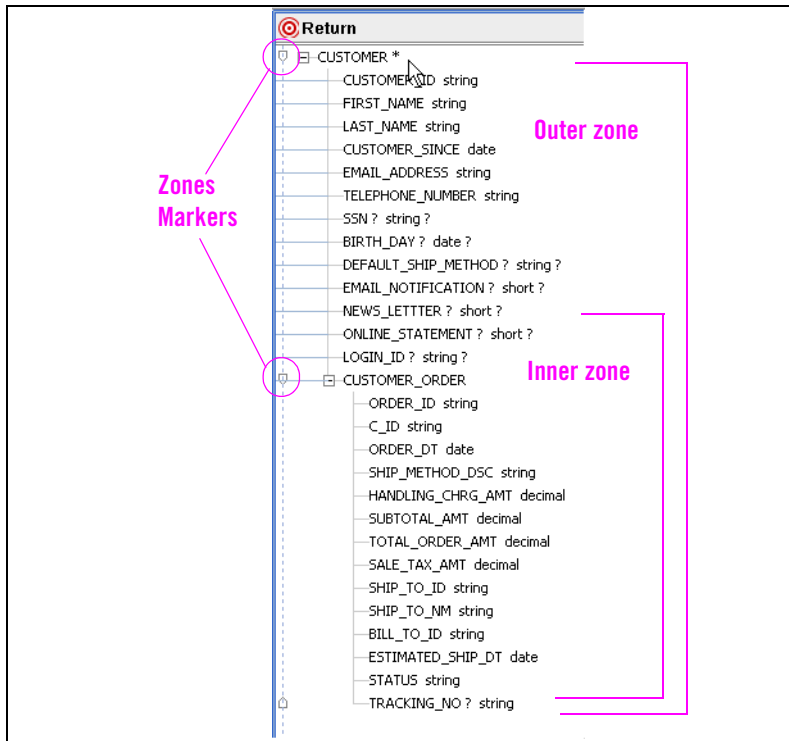
When you add a complex child element it will be place at the end of its peers in the return type.

Setting Zones in Your Return Type

In DSP return types zones identify how query results will be arranged. Adding or changes zones through the XQuery Editor is the same as adding or changing the order of subordinate for statements in Source View. (For a detailed example of building a logical data service that makes use of zones to create a nested master-detail arrangement of data see “[Creating a New Data Service and Data Service Function](#)” on page 6-7.

For example in [Figure 6-40](#) the CUSTOMER_ORDER elements for a particular customer will be grouped under that customer.

Figure 6-40 Sample Return Type With Two Zones



By default, return types have only a single zone. However, without additional zones elements simply repeat in their natural order. In the simple example shown in [Figure 6-40](#) this would mean that if a customer had more than one order, both the customer information and the order information would be repeated in your report until all matching orders had appeared.

The following slightly simplified XML illustrates a single-zone approach.

```

<CUSTOMERID>987655</CUSTOMERID>
<CUSTOMERNAME>Supermart</CUSTOMERNAME>
<ORDER>
  <ORDERID>632</ORDERID>
  <CUSTOMERID>987655</CUSTOMERID>
<CUSTOMERID>987655</CUSTOMERID>
<CUSTOMERNAME>Supermart</CUSTOMERNAME>
<ORDER>
  <ORDERID>888</ORDERID>
  <CUSTOMERID>987655</CUSTOMERID>
..

```

Notice the repetition of CUSTOMERNAME and CUSTOMERID.

XQuery source for a similar function clearly shows why this is:

```

for $CUSTOMER in ns0:CUSTOMER()
  for $CUSTOMER_ORDER in ns1:CUSTOMER_ORDER()
    where $CUSTOMER/CUSTOMER_ID = $CUSTOMER_ORDER/C_ID
return <ns2:CUSTOMER7>
  <CUSTOMER_ID>{fn:data($CUSTOMER/CUSTOMER_ID)}</CUSTOMER_ID>
  <LAST_NAME>{fn:data($CUSTOMER/LAST_NAME)}</LAST_NAME>

<ns1:CUSTOMER_ORDER>
  <ORDER_ID>{fn:data($CUSTOMER_ORDER/ORDER_ID)}</ORDER_ID>
  <C_ID>{fn:data($CUSTOMER_ORDER/C_ID)}</C_ID>

<TOTAL_ORDER_AMT>{fn:data($CUSTOMER_ORDER/TOTAL_ORDER_AMT)}</TOTAL_ORDE
R_AMT>
</ns1:CUSTOMER_ORDER>
</ns2:CUSTOMER7>

```

If you were, however, to create a repeatable zone around the CUSTOMER_ORDER element, a subordinate for clause will be introduced in Source View.

```

for $CUSTOMER in ns0:CUSTOMER()
return <ns2:CUSTOMER7>
  <CUSTOMER_ID>{fn:data($CUSTOMER/CUSTOMER_ID)}</CUSTOMER_ID>
  <LAST_NAME>{fn:data($CUSTOMER/LAST_NAME)}</LAST_NAME>{
for $CUSTOMER_ORDER in ns1:CUSTOMER_ORDER()
where $CUSTOMER/CUSTOMER_ID = $CUSTOMER_ORDER/ORDER_ID
return
<ns1:CUSTOMER_ORDER>
  <ORDER_ID>{fn:data($CUSTOMER_ORDER/ORDER_ID)}</ORDER_ID>
  <C_ID>{fn:data($CUSTOMER_ORDER/C_ID)}</C_ID>

<TOTAL_ORDER_AMT>{fn:data($CUSTOMER_ORDER/TOTAL_ORDER_AMT)}</TOTAL_ORDE
R_AMT>
</ns1:CUSTOMER_ORDER>

```

```
}  
</ns2:CUSTOMER7>
```

Specifically the highlighted where clause in the second code fragment mandates that all orders be collected under a single instance of customer.

To create a zone simply right-click on an element and select Mark as Zone. Once created, the zone will appear highlighted whenever you move your cursor into areas under its control ([Figure 6-40](#)).

Associating XQuery Editor Nodes With Zones

In XQuery for, let, and group by clauses can enclose other for, let, or group by clauses. Similarly, nodes representing these constructs can be associated with return type zones using the create zone icon in the titlebar of the node (see [Figure 6-9](#) in the XQuery Editor example at the beginning of this chapter). Simply drag the icon over an existing zone to associate the node with the zone.

To verify that the operation is successful mouse-over the zone icon after the association is complete. If successful, the appropriate zone will be highlighted (see [Figure 6-40](#)). Alternatively, look at Source View to verify that your operation has been successful or simply run your query under Test View.

Note: The order in which you create zones and other aspects of your XQuery in the XQuery Editor can be significant. For example, zones should be created before creating a where clause associating two nodes.

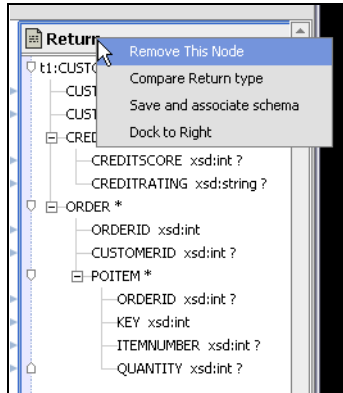
Removing Zones

To remove a zone, right-click on the parent element in the zone and select the Remove Zone option.

Validating and Saving Your Return Type

You can make changes in your function's return type and, optimally, bring your data service into conformance with the changes that you have made.

Figure 6-41 Return Type Management Options



Several right-click menu options are available for managing the return type, including:

- **Show Type Difference.** A toggle that displays or hides distinctions between your return type and the data service XML type. When activated Show Type Difference color coding is used to categorize differences between your return type and your data service's XML type.

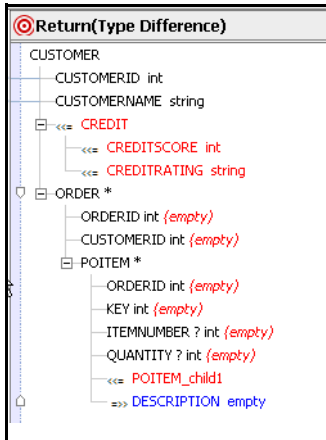
Color	Meaning
Black	Unchanged from XML type.
Red	Removed from the return type (but still present in the XML type).
Blue	In the return type but not the XML type.

Notice in the following example that two new child elements have been added to the return type.

Elements differences detected when comparing the return type with the content of the XML type are shown in red. This includes elements you have deleted from the return type as well as those you have added to the return type.

The addition of DESCRIPTION to the return type is shown in blue in [Figure 6-42](#).

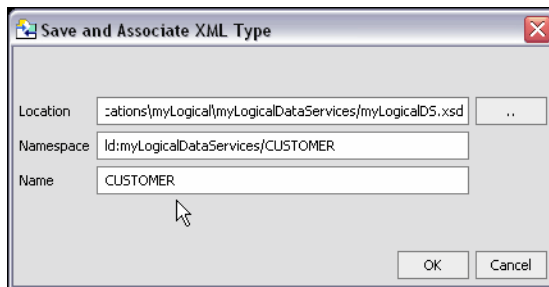
Figure 6-42 Return Type With a New Element



The arrows to the left of changed items indicate whether the change is originating locally in the return type (→) or in the data service's XML type (←).

- **Save and Associate Schema.** Provides a means for substituting the schema of a revised return type for the data service XML type. In order to change the data service XML type using this command you should not change the return type name.

Figure 6-43 Save and Associate Dialog



This command can also be used to save the revised return type to a schema, schema location, target Namespace, or root name that is different than that used by the containing data service.

When you are building a return type from data service functions it is sometimes necessary to change either the namespace or the root name prior to using the Save and Associate Schema command. This is because the qualified name of your return type will initially be the same as the function used to create the return type.

Other options include:

- Going to Design View and use the right-click menu Associate XML Type to change the schema associated with the data service (see [“Associating an XML Type” on page 4-23](#)). This will change the return type for all the read functions in your data service.
- Saving your data service to a new name using the Save As command. Then associate the new XML type. This is probably the better option if you have other data services that are dependent on the XML type.
- **Dock to Right.** A toggle that attaches/detaches the return type to the right edge of the work area.
- **View Source.** Shows your return type in its native XML format.

Working with the XQuery Editor

Testing Query Functions and Viewing Query Plans

You can use Test View to execute any data service read or relationship function for which data is available.

When you run a query in Test View results appear in an editable window in text or structured XML form. When updates are available for your data, you can immediately update your back-end data. Query results can also be used as complex parameters for other queries.

In creating support for query functions, BEA Aqualogic Data Services Platform (DSP) determines Test View options from your query function's signature. Several types of query function signatures are supported including queries with and without parameters, simple and complex parameters, and ad hoc queries.

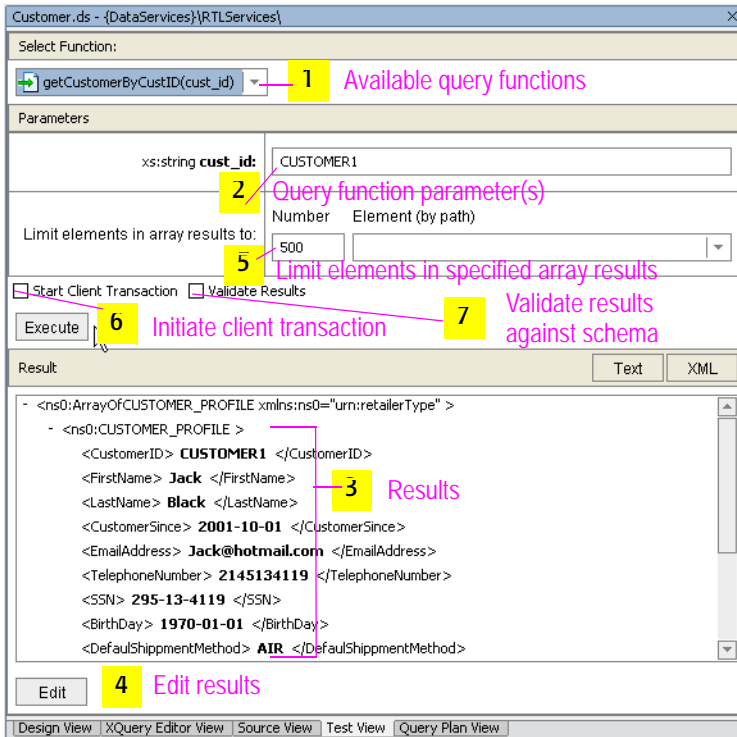
The following major topics are covered in this chapter:

- [Running Queries Using Test View](#)
- [Analyzing Queries Using Plan View](#)
- [Creating an Ad Hoc Query](#)

Running Queries Using Test View

In Test View you can select any read or navigation functions or procedures defined in your data service from a drop-down list.

Figure 7-1 Test View Options for a Function Accepting a Simple Parameter

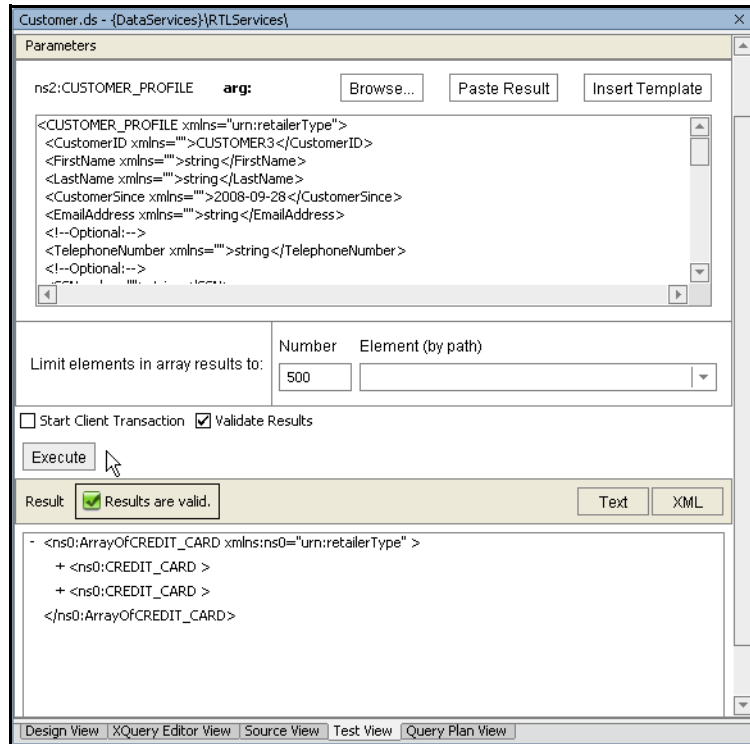


If the query accepts complex parameters, the parameter entry dialog automatically adjusts, as shown in Figure 7-2.

Also see in the Data Services Platform [Samples Tutorial Part II](#):

- Lesson 21: Running Ad Hoc Queries
- Lesson 26: Understanding the Query Plan

Figure 7-2 Function Accepting a Complex Parameter As Input



Using Test View

To use Test View, follow these steps:

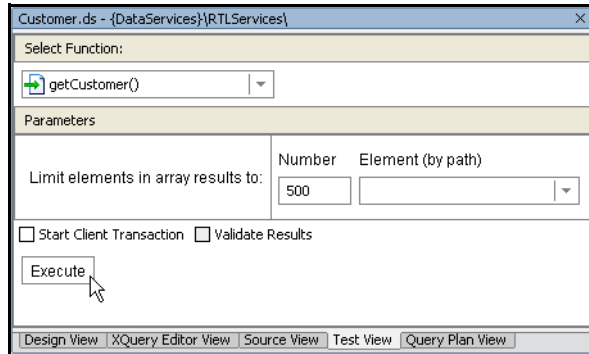
1. Select the Test View tab, then chose a function from the pulldown menu. The menu contains the read and navigation functions in your current data service, as well as the data service's procedures.
2. Enter parameters, if any.
3. Click on Execute to run the query and view the results.
4. If you have back-end data write permission, you can make changes in your data as well. Click on Edit Results and make any necessary changes. Then click Submit to update your data.

You can review your generated query in the Output window. See [“Auditing Query Performance” on page 7-13](#) for details.

Running a Query That Needs No Parameters

In the case of a query such as `getAllCustomers()`, no parameters are needed (Figure 7-3).

Figure 7-3 Query Without Parameters



When you click Execute the query will run.

Results are returned in text or XML form. Click on the + next to a complex element (in this case, a table representation) to see more detailed results.

Editing Results

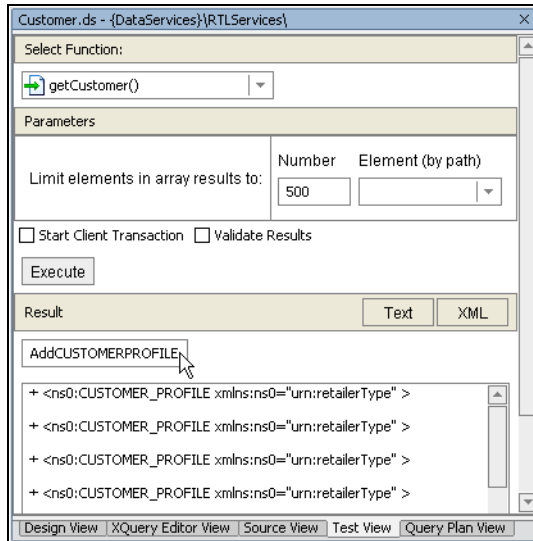
When you have appropriate update permissions — as is commonly the case with “sand box” testing — you can directly edit results using the Edit command (Figure 7-4).

Figure 7-4 Editing Query Results



You also have the option of adding a record once you are in Edit mode.

Figure 7-5 Adding a Record to a Data Set



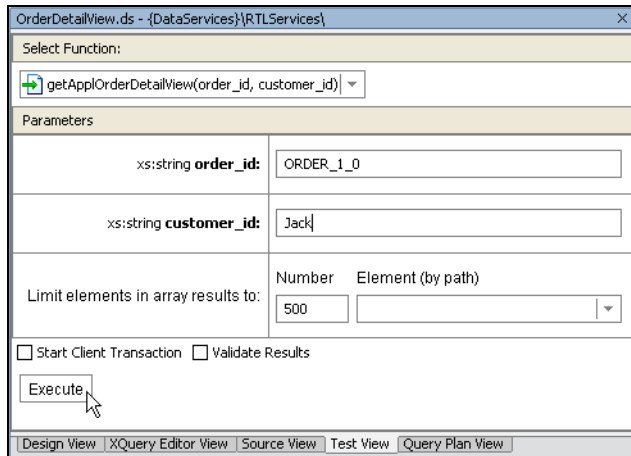
The structure for the root XML Type will be added to end of the data set. You will need to supply the content, of course. If you right-click on the root element of your new record, you can also add complex child elements.

When you are satisfied with the changes click Submit.

Running a Query Function With Simple Parameters

When your query requires one or multiple simple parameters, Test View display each parameter in its own field, identified by name and required type.

Figure 7-6 Function with Two Input Parameters



See [“Running a Query That Needs No Parameters” on page 7-4](#) for details on executing a query and editing and submitting results.

Testing a Query Function With Complex Parameters

Enterprise-scale queries often require a complex parameter type as input. For example, an inventory query may require a set of parameters which are based on a Web service supplying details of orders received. It is usually easier to just pass the entire object than to specify a large set of individual parameters.

When your query requires a complex parameter, the function will be listed with a parameter as in:

```
getProfileView(arg)
```

The `arg` parameter indicates that a complex parameter type is needed.

For such parameters Test View displays a box ([Figure 7-2](#)) into which you can:

- Paste the results of the most recently run query (assuming the results match the input requirements of the current function).
- Paste a template of the complex parameter into which you can enter necessary (that is, required primary key) values.
- Identify to an XML file to serve as input.
- Enter your complex parameter directly.

Using Prior Results as Input

For any given data service you can use results from a previously run query as input. This is particularly useful when invoking navigation functions, since navigation functions generally require complex parameters.

Note: When pasting prior results it's important to keep in mind that queries returning multiple results (arrays) cannot be input to functions looking for a single object as a parameter. For example, a function that gets orders for a particular customer is likely to return multiple orders. Those results cannot be used as input to a function that returns information about a particular customer.

The following steps show how results of a singleton query can be repurposed as input for a complex parameter.

1. Assume that you have first run a simple query, selecting information on a particular order. Then you want to get additional information on the customer who placed the order.

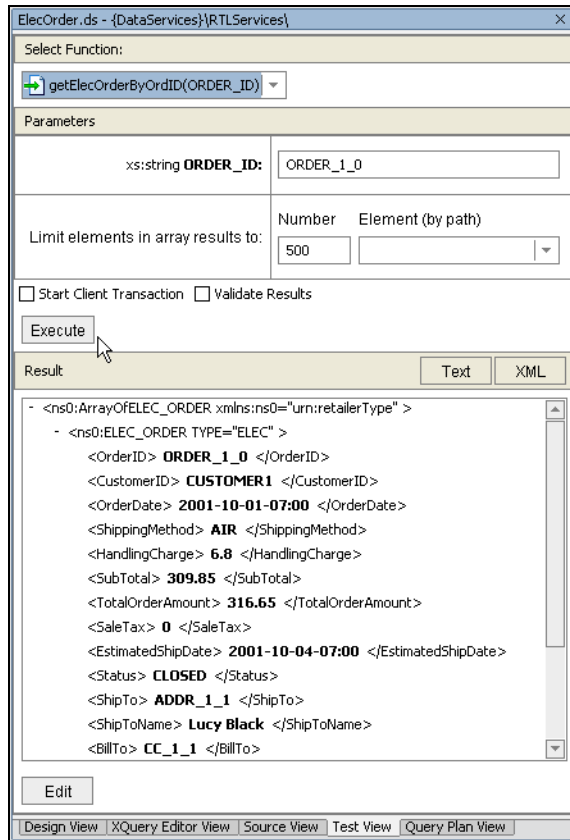
Results shown below contain elements called for by the function:

```
getElecOrderByOrdID (ORDER_ID)
```

located in the RTLServices/ElecOrder data service.

2. In the Test View parameter area supply a valid order ID such as ORDER_1_0.

Figure 7-7 Executing a simple parameterized query



3. Your results now contain the required customer ID. Select the getCustomer() relationship function from the dropdown list of available functions.
4. Click on the Paste Result button. Your previous results appear as an editable complex parameter in XML format (Figure 7-8).

Figure 7-8 Using Query Results in a New Query

The screenshot shows the Test View interface for a query named 'getCustomer(arg)'. The 'Parameters' section has 'ns18:ELEC_ORDER' selected. The results pane shows XML data for an 'ArrayOfELEC_ORDER' element, with the first element highlighted in blue. The interface includes buttons for 'Browse...', 'Paste Result', and 'Insert Template', and a section for limiting array results to 500 elements.

Note: Your results have been returned as a singleton element in an array (highlighted in blue in Figure 7-8). The array element needs to be removed before you can successfully execute your navigation function.

5. Edit your results to remove the `ArrayOfELEC_ORDER` element. The outermost elements of your XML document will change from:

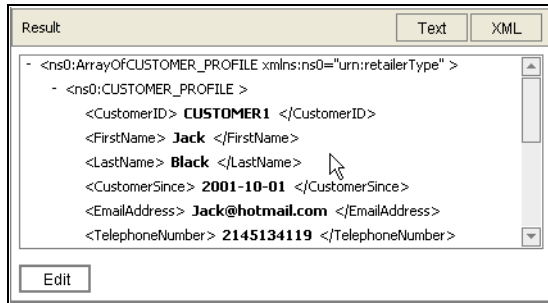
```
<ns0:ArrayOfELEC_ORDER xmlns:ns0="urn:retailerType">
  <ns0:ELEC_ORDER TYPE="ELEC">
    <OrderID>ORDER_1_0</OrderID>
    ...
  </ns0:ELEC_ORDER>
</ns0:ArrayOfELEC_ORDER>
```

to:

```
<ns0:ELEC_ORDER TYPE="ELEC" xmlns:ns0="urn:retailerType">
  <OrderID>ORDER_1_0</OrderID>
  ...
</ns0:ELEC_ORDER>
```

6. After making the necessary changes click **Execute**. Results of your new query are based on the Customer XML type appear (Figure 7-9).

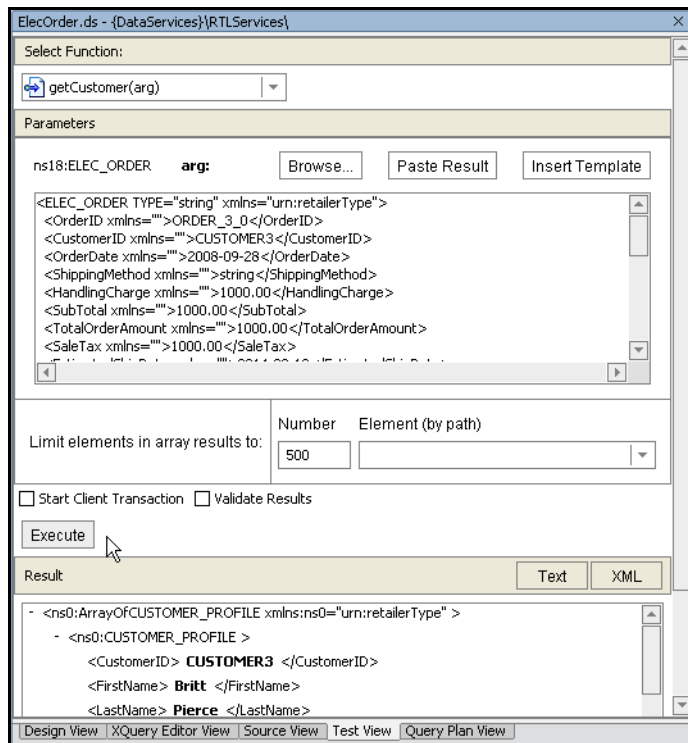
Figure 7-9 Complex Parameterized Query Results



Using the XML Type to Identify Input Parameters

You can automatically enter a template of the XML type of your data service. In [Figure 7-10](#), a customer ID (CUSTOMER3) and order ID (ORDER_3_0), are provided through the template. Results are also shown.

Figure 7-10 Using XML Type Template to Guide Data Input



Template parameters are useful when you know the key parameters required by your query.

See [“Running a Query That Needs No Parameters” on page 7-4](#) for details on executing a query and editing and submitting results.

Testing DSP Procedures

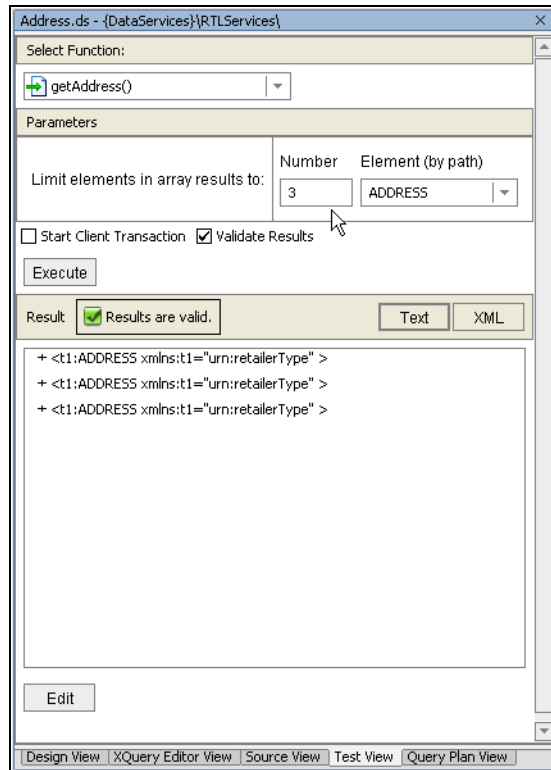
In Test View procedures are selected and run from the Select Functions drop-down list box in the same way that functions are selected. Running a procedure under Test View shows results only if the procedure returns data or a confirming message as to whether the operation was successful, for example.

Limiting Array Results

You can filter query results through Test View to *n* instances of a single element such as the first five of an array of 5,000 customers.

Figure 7-11 shows a function where the results for `RTLServices/Address/getAddress()` are limited to three Address elements. Without such a limitation, all customer records would be returned.

Figure 7-11 Limiting Elements in an Array Result



Starting Client Transaction Option

The Client Transaction Option supports functions that query more than multiple (two or more) relational sources using XA transaction drivers. By default this option is not selected, meaning that the `NotSupported` EJB transaction method is used. If the option is checked, the `Required` transaction mode will be used instead.

For general information on the subject see [“Transactions in EJB Applications”](#) WebLogic Server documentation.

Validating Results

Test View results are validated against the data service's schema file when the Validate Results checkbox (shown in [Figure 7-11](#)) is selected. When active the following conditions will be flagged as invalid:

- An illegal type mismatch between source elements and the return type. For example, if an element of type string is mapped to an element of type date, the query results are invalid since a string cannot be guaranteed to cast successfully to a date.
- An element or attribute that is required in the schema is removed from the return type.
- An element or attribute is added to the return type.

Invalid results are reported in the Output window. Such results can be addressed by correcting the return type or associating the return type with a new, corrected schema. See [“Validating and Saving Your Return Type” on page 6-52](#).

Notes: Whenever you attempt to edit results of a query, those results are re-validated. The criteria is the same as that used for the Validate Results option.

Results are validated by calling the XMLBean validate() method, currently documented at the following URL:

[http://edocs.bea.com/workshop/docs81/doc/en/workshop/java-class/com/bea/xml/Xm1Object.html#validate\(\)](http://edocs.bea.com/workshop/docs81/doc/en/workshop/java-class/com/bea/xml/Xm1Object.html#validate())

Disregarding a Running Query

An executing query can be ended through the Data Services Platform Console or by ending your server process. However, you can start a new query by changing your selection in Test View.

Auditing Query Performance

You can audit query performance by activating Audit for your application. This is a one-time operation which is accomplished through the Data Services Platform Console.

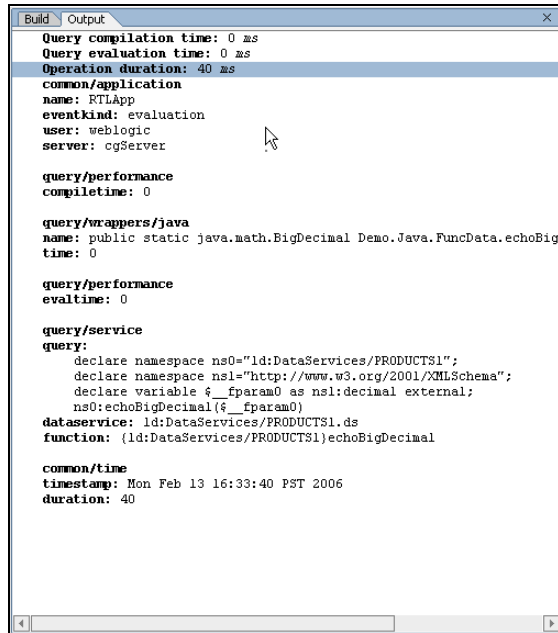
When a query function is invoked through Test View and DSP auditing is enabled, basic validation and performance information appears in the WebLogic Workshop Output window (View → Windows → Output). You can find the most recent query results at the bottom of the Output pane.

Note: For details on enabling auditing and tuning audit options see [“Audit and Log Information”](#) in the DSP *Administration Guide*.

Testing Query Functions and Viewing Query Plans

By default an audit includes such information as query compilation and execution time, user, server, and so forth (Figure 7-12).

Figure 7-12 Output Window Audit Results



```
Build Output
Query compilation time: 0 ms
Query evaluation time: 0 ms
Operation duration: 40 ms
common/application
name: RTLApp
eventkind: evaluation
user: weblogic
server: cgServer

query/performance
compiletime: 0

query/wrappers/java
name: public static java.math.BigDecimal Demo.Java.FuncData.echoBig
time: 0

query/performance
evaltime: 0

query/service
query:
  declare namespace ns0="ld:DataServices/PRODUCTS1";
  declare namespace ns1="http://www.w3.org/2001/XMLSchema";
  declare variable $__fparam0 as ns1:decimal external;
  ns0:echoBigDecimal($__fparam0)
dataservice: ld:DataServices/PRODUCTS1.ds
function: {ld:DataServices/PRODUCTS1}echoBigDecimal

common/time
timestamp: Mon Feb 13 16:33:40 PST 2006
duration: 40
```

Note: Query plan audit properties are not collected when a function is executed from Test View. This is because the function cache is not utilized for functions executed in Test View.

Analyzing Queries Using Plan View

Two types of information are available to help you analyze the design and performance of your query.

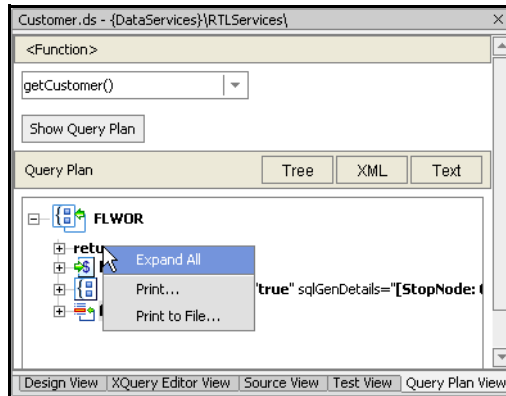
- Auditing information which appears in the Results window (see [Auditing Query Performance](#)).
- Query Plan View

Query Plan View helps in understanding how a query is designed. In addition to being able to view the plan, you can also print it (using the right-click menu option) or save it to a file in XML format.

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 (Figure 7-13).

Figure 7-13 Query Plan Right-Click Options



A query plan identifies the following query components:

- Joins
- Outer join
- Select statements
- Data sources
- Custom function calls
- Order-bys
- Remove duplicates
- Source access operator

Figure 7-14 Query Plan Fragment for RTLApp’s Customer Data Service getCustomerByCustID(cust_id) Function

The screenshot shows a query plan viewer for the function `getCustomerByCustID(cust_id)`. The interface includes a dropdown menu for the function name, a "Show Query Plan" button, and tabs for "Tree", "XML", and "Text". The "Tree" view is selected, showing a hierarchical structure of the query plan. Annotations point to various parts of the plan:

- Return type:** Points to the `<PROFILE>` element in the `return` block.
- View Options:** Points to the "Tree", "XML", and "Text" tabs.
- Temporary Traceable Variables:** Points to the `let $t62882` and `let $t62881` blocks.
- Source identification:** Points to the `relational source: cgDataSource` block.
- Join:** Points to the `LEFT OUTER JOIN` clause in the `FROM` clause.

The query plan structure is as follows:

```

<Function>
  getCustomerByCustID(cust_id)
  Show Query Plan
  Query Plan
    FLWOR
      return
        <PROFILE>
          <<CustomerID>> {t62879}
          <<FirstName>> {t62874}
          <<LastName>> {t62877}
          <<CustomerSince>> {t62878}
          <<EmailAddress>> {t62873}
          <<TelephoneNumber>> {t62869}
          <<SSN>> {t62870}
          <<BirthDay>> {t62876}
          <<DefaultShipmentMethod>> {t62875}
          <<EmailNotification>> {t62872}
          <<OnlineStatement>> {t62871}
        let $t62882
          FLWOR
            return
              <ADDRESS>
                <<AddressID>> {t62881/{t62831}}
                <<CustomerID>> {t62881/{t62846}}
                <<FirstName>> {t62881/{t62845}}
                <<LastName>> {t62881/{t62838}}
                <<StreetAddress_1>> {t62881/{t62833}}
                <<StreetAddress_2>> {t62881/{t62832}}
                <<City>> {t62881/{t62836}}
                <<State>> {t62881/{t62835}}
                <<ZipCode>> {t62881/{t62841}}
                <<Country>> {t62881/{t62839}}
                <<DayPhone>> {t62881/{t62843}}
                <<EveningPhone>> {t62881/{t62842}}
                <<Alias>> {t62881/{t62834}}
                <<Status>> {t62881/{t62844}}
                <<IsDefault>> {t62881/{t62840}}
            where
              fn:exists(
                {t62881/{t62837}}
              )
            for $t62881
            groupBy preclustered="true" stable="true"
            for $t62867
              relational source: cgDataSource :
                SELECT t1."BIRTH_DAY" AS c1, t1."CUSTOMER_ID" AS c2, t1."CUSTOMER_SINCE" AS c3,
                t1."DEFAULT_SHIP_METHOD" AS c4, t1."EMAIL_ADDRESS" AS c5, t1."EMAIL_NOTIFICATION" AS c6,
                t1."FIRST_NAME" AS c7, t1."LAST_NAME" AS c8, t1."ONLINE_STATEMENT" AS c9, t1."SSN" AS c10,
                t1."TELEPHONE_NUMBER" AS c11, t2."ADDR_ID" AS c12, t2."ALIAS" AS c13, t2."CITY" AS c14,
                t2."COUNTRY" AS c15, t2."CUSTOMER_ID" AS c16, t2."DAY_PHONE" AS c17, t2."EVE_PHONE" AS c18,
                t2."FIRST_NAME" AS c19, t2."IS_DEFAULT" AS c20, t2."LAST_NAME" AS c21, t2."STATE" AS c22,
                t2."STATUS" AS c23, t2."STREET_ADDRESS1" AS c24, t2."STREET_ADDRESS2" AS c25, t2."ZIPCODE" AS c26
              FROM "RTLCUSTOMER", "CUSTOMER" t1
              LEFT OUTER JOIN "RTLCUSTOMER", "ADDRESS" t2
              ON (t1."CUSTOMER_ID" = t2."CUSTOMER_ID")
              WHERE (t1."CUSTOMER_ID")
              ORDER BY t1."CUSTOMER_ID" ASC
    
```

There are several ways that a query plan can be viewed:

- **Tree view.** A collapsible graphical presentation of the query plan.
- **XML view.** A collapsible XML document view of the query plan.
- **Text view.** Presents the information as text.

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. The following table identifies the conditions associated with informational and warning messages.

Table 7-15 Informational and Warning Messages Associated With Query Plan Segments

Warning Message Type	Informational Message Type
<ul style="list-style-type: none"> • Typematch. Typematch issues that will be resolved by the compiler (may affect performance) 	<ul style="list-style-type: none"> • Audit. Auditing has been set for this particular function (will affect performance).
<ul style="list-style-type: none"> • No where clause. There is no predicate associated with the query function (will affect performance). 	<ul style="list-style-type: none"> • Cache. Function is cached (may enhance performance).
	<ul style="list-style-type: none"> • SQL pushdown generation details.

Printing Your Query Plan

A right-mouse option 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.

Analyzing a Sample Query

The following query is from the Data Services Platform RTLApp:

```
(RTLServices/OrderDetailView/getElecOrderDetailView(order_id,
customer_id)
```

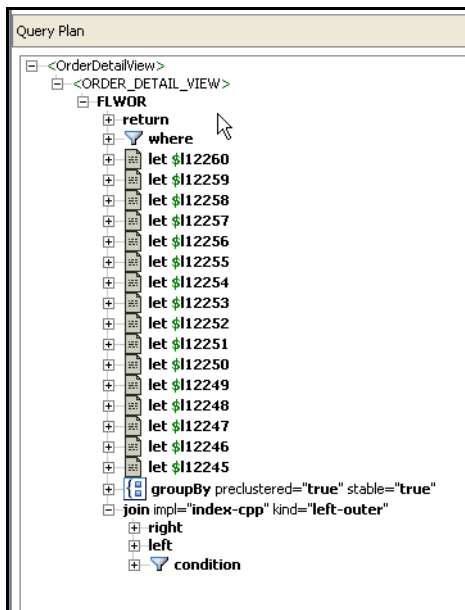
From the function signature you know that the query returns data related to order details after it is passed an order ID and a customer ID.

The following pseudocode describes 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*

A compressed version of the query plan is shown in [Figure 7-16](#).

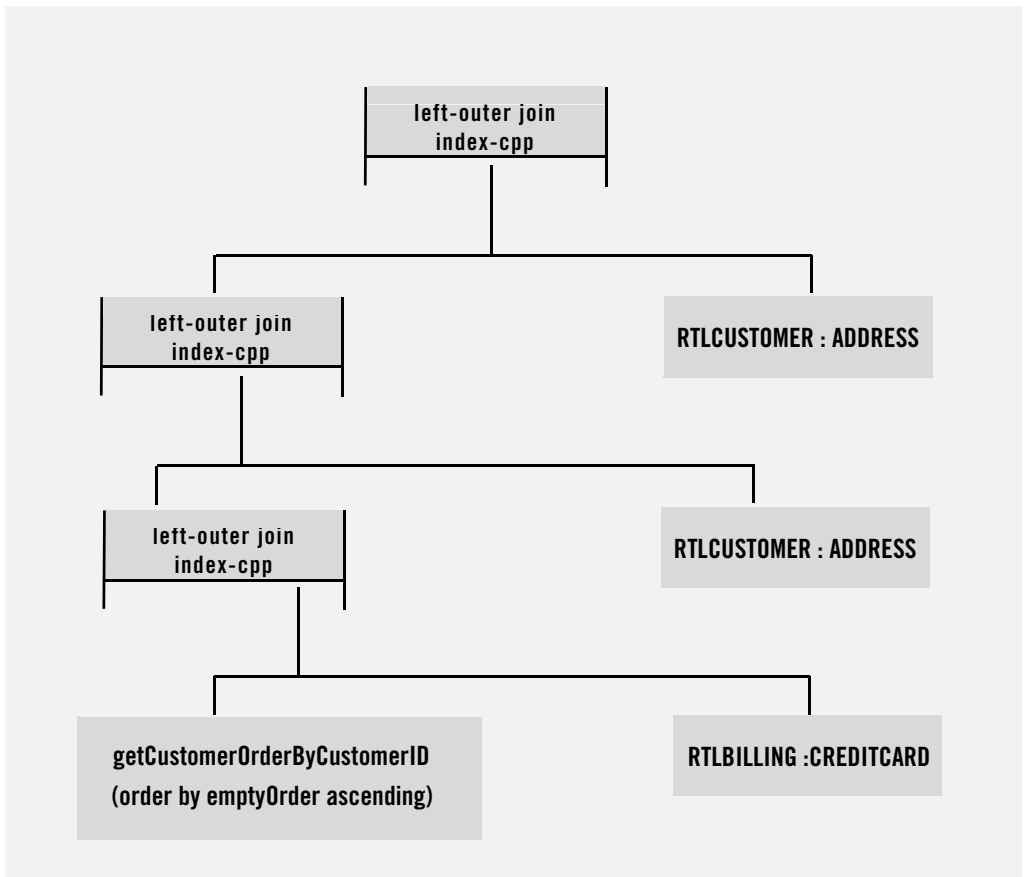
Figure 7-16 Query Plan for getElecOrderDetailView()



The let 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, as shown in [Figure 7-17](#).

Figure 7-17 Top Down Schematic of getElecOrderDetailView() Function



Working With Your Query Plan

Two options are available in Query Plan.

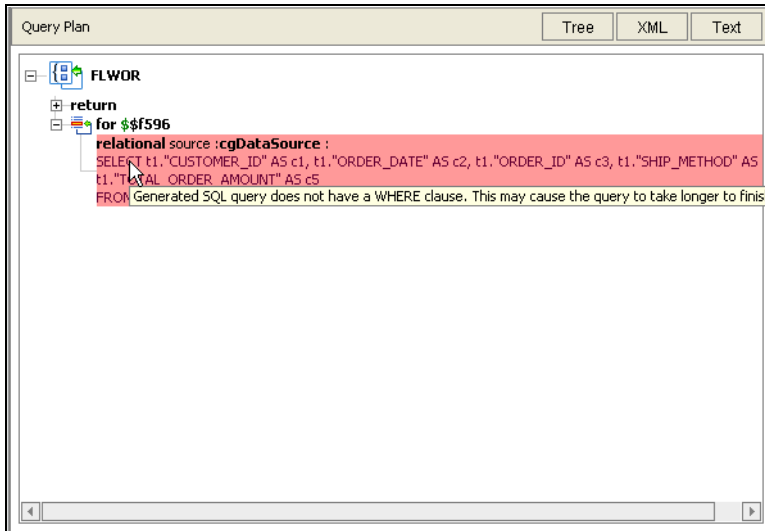
- **Expand All.** This right-click menu option expands the currently selected element and any children. If applied to the top-most element in the plan, all elements are expanded.

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.

Figure 7-18 Query Plan Viewer Flagging a For Statement with a Missing Where Clause

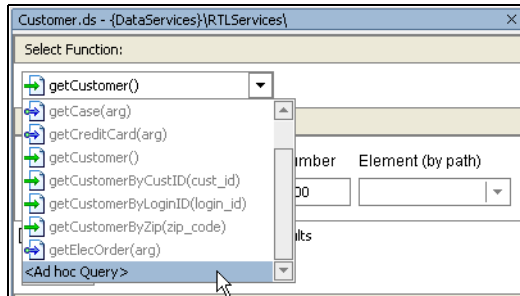


Simply mouse-over the highlighted section of the plan to view the information or warning.

Creating an Ad Hoc Query

It can be useful to quickly enter and test queries. You can do this through any data service's Test View. Simply pull down the list of available functions and select the ad hoc query option (shown in [Figure 7-19](#)).

Figure 7-19 Selecting Ad Hoc Query Option From Test View



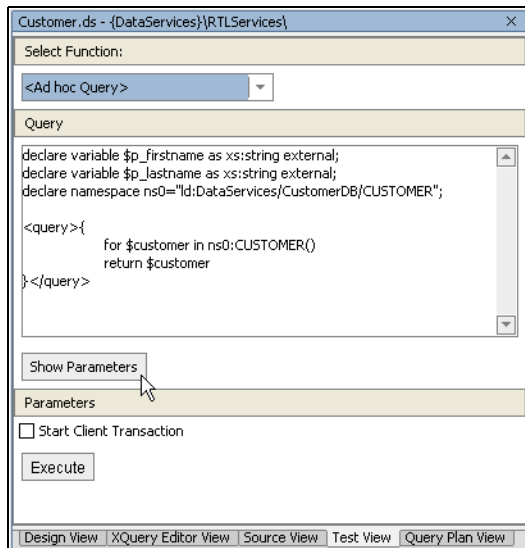
As the name implies, an ad hoc query is not limited to the currently selected data service. Any data service in the scope of the current application can be utilized.

Note: An ad hoc query remains available whenever the data service active when it was created is open to Test View. However, the ad hoc query is not visible in Source View and can only be saved by copying it to an external application.

Sample Ad Hoc Queries

In [Figure 7-20](#) a small query has been entered. Although the constructor function for the current data service was used (DataServices/CustomerDB/CUSTOMER), this was unnecessary.

Figure 7-20 Creating an Ad Hoc Query



Testing Query Functions and Viewing Query Plans

If your query requires simple or complex parameters, these can be exposed using the Show Parameters button.

A Results pane below the Execute button will contain the data returned by the query (if any).

In the RTLApp sample application you can copy the code in [Listing 7-1](#) into an ad hoc query pane. This query is designed to take several minutes to complete. It can also be used to experiment with monitoring and stopping executing queries through the Data Services Platform Console.

Note: In order to execute this query the applications Check Access Control option in Data Services Platform Console's General tab must be deselected. See DSP [Administration Guide](#) for details.

Listing 7-1 Sample Ad Hoc Query Executable From RTLApp's Test View

```
import schema namespace ns2="urn:retailerType" at
"ld:DataServices/RTLServices/schemas/CustomerProfile.xsd";

declare namespace ns9="ld:DataServices/RTLServices/Customer";

declare function ns9:getCustomerSlowly() as element(ns2:CUSTOMER_PROFILE)* {

    for $CUSTOMER_PROFILE in ns9:getCustomer(),
        $c1 in ns9:getCustomer() [CustomerID lt $CUSTOMER_PROFILE/CustomerID],
        $c2 in ns9:getCustomer() [CustomerID gt $CUSTOMER_PROFILE/CustomerID],
        $c3 in ns9:getCustomer() [CustomerID eq $CUSTOMER_PROFILE/CustomerID],
        $c4 in ns9:getCustomer() [CustomerID lt $CUSTOMER_PROFILE/CustomerID],
        $c5 in ns9:getCustomer() [CustomerID gt $CUSTOMER_PROFILE/CustomerID],
        $c6 in ns9:getCustomer() [CustomerID eq $CUSTOMER_PROFILE/CustomerID],
        $c7 in ns9:getCustomer() [CustomerID = $CUSTOMER_PROFILE/CustomerID],
        $c8 in ns9:getCustomer() [CustomerID != $CUSTOMER_PROFILE/CustomerID],
        $c9 in ns9:getCustomer() [CustomerID = $CUSTOMER_PROFILE/CustomerID],
        $c10 in ns9:getCustomer() [CustomerID !=

$CUSTOMER_PROFILE/CustomerID],
        $c11 in ns9:getCustomer() [CustomerID eq $CUSTOMER_PROFILE/CustomerID],
        $c12 in ns9:getCustomer() [CustomerID eq $CUSTOMER_PROFILE/CustomerID]
    return $CUSTOMER_PROFILE
};

ns9:getCustomerSlowly()
```

Once an ad hoc query has been entered, its query plan can be reviewed. See “[Analyzing Queries Using Plan View.](#)”

Also see in the Data Services Platform [Samples Tutorial Part II:](#)

- Lesson 21: Running Ad Hoc Queries

Testing Query Functions and Viewing Query Plans

Working with XQuery Source

This chapter describes BEA Aqualogic Data Services Platform (DSP) Source View. It includes the following topics:

- [What is Source View?](#)
- [Using Source View](#)

What is Source View?

The underlying XQuery source of a data service typically:

- References a schema as the data service's XML type
- Defines one or several read functions and, optionally, one or several relationship functions
- Declares namespaces for referenced services
- Contains various pragma directives to the XQuery engine

In addition, data services created from physical data sources contain metadata related to the physical sources. For example, data services based on relational data describe the XML field 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 more convenient or necessary to modifying source.

Also see in the Data Services Platform [Samples Tutorial Part II:](#)

- Lesson 28: Configuring Alternatives for Unavailable Data Sources

There are times when it may be preferable to develop or troubleshoot data services by working directly in source. The Source View tab 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 XQuery Editor; similarly, source is immediately updated when changes are made through the XQuery Editor View or Design View.

XQuery Support

Data Services Platform 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.

DSP supplements the base XQuery syntax with a set of elements and directives that appear in the 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 Data Services Platform [XQuery Developer's Guide](#).

Figure 8-1 Source View Showing Pragmas, Namespace Declarations, and a Function

```

Customer.xsd* - {DataServices}

(::pragma xds <x:xds targetType="t:CUSTOMER" xmlns:x="urn:annotations.ld.bea
<userDefinedView/>
<relationshipTarget roleName="payment"/><functionForDecomposition name="f:get
</x:xds::>)

declare namespace f1 = "ld:DataServices/CUSTOMERS:POINTBASE/WEBLOGIC";
declare namespace f2 = "ld:DataServices/PO_CUSTOMERS:POINTBASE/WEBLOGIC";
declare namespace f3 = "ld:DataServices/PO_ITEMS:POINTBASE/WEBLOGIC";
declare namespace l1 = "ld:DataServices/Customer";

import schema namespace t1 = "http://temp.openuri.org/schemas/Customer.xsd" a
import schema namespace crxsd = "http://www.openuri.org/" at "ld:DataServices
declare namespace f11 = "ld:DataServices/getCustomerCreditRatingResponse:crRa

(::pragma function <f:function xmlns:f="urn:annotations.ld.bea.com" kind="dat
declare function l1:getCustomer($arg0 as xs:int) as element(xsd:dat);

import schema namespace t3 = "ld:DataServices/schemas/PAYMENTList" at "ld:Da
declare namespace f5 = "ld:DataServices/PAYMENTS:POINTBASE/WEBLOGIC";
import schema namespace t2 = "ld:DataServices/PAYMENTS:POINTBASE/WEBLOGIC" at

(::pragma function <f:function kind="navigate" returns="payment" xmlns:f="ur:
declare function l1:getPaymentList($c as element(t1:CUSTOMER)) as element(t3:
  fn-bea:probe(<t3:PAYMENTList>
  {
    for $p in f5:PAYMENTS()
    where $p/CUSTID = 987654
    return
      <PAYMENTS>
      <CUSTID> {fn:data($p/CUSTID)} </CUSTID>
      <PAYMENT> {fn:data($p/PAYMENT)} </PAYMENT>
      <COMMENT> {fn:data($p/COMMENT)} </COMMENT>
      </PAYMENTS>
  }
  </t3:PAYMENTList>)
);

```

Using Source View

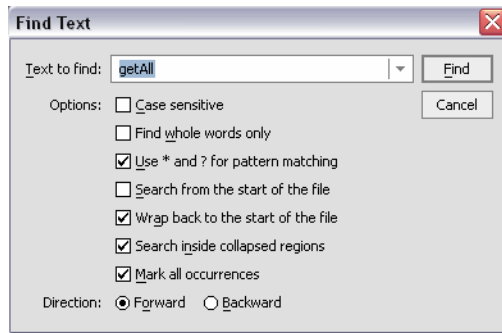
You can view a file in Source View by clicking the Source View tab. To open Source View to a particular query function, first select the function from Design View or XQuery Editor View, then click the Source View tab.

Finding Text

You can search for specific text strings in Source View using its open you can access file search using WebLogic Workshop's Edit → Find command option or <Ctrl-f>. Complete search and replace facilities are available including specifying case, whole words only, wildcard search patterns, and limited search. You also have the option to mark all occurrences of found strings.

Found items are highlighted in yellow. This makes it easy to trace the use of variables, for example.

Figure 8-2 Source View Search Dialog Box



Function Navigation

As a convenience you can quickly navigate to the data service represented by a particular function by clicking Ctrl while holding your mouse over a particular function call such as:

```
for $fk in f3:ADDRESS()
```

If you click the pop-up which repeats the function name, the data service that contains that function will open to that function.

Code Editing Features

WebLogic Workshop contains a rich code editing environment.

Color Coding

XQuery documents in Source View are color-coded to highlight the various elements of the source code. By default keywords are blue and bold, comments (including pragmas) are colored grey, and variables are colored magenta.

Figure 8-3 Color Coding in Source View

```

declare namespace f1 = "ld:TKAppLiquidDataApp/CUSTOMER";
import schema namespace t6 = "ld:TKAppLiquidDataApp/CUSTOMER" at "ld:TKAppLiquidDataApp/schemas/CUSTOMER.xsd";
(::pragma function <f:function xmlns:f="urn:annotations.ld.bea.com" kind="read" nativeName="CUSTOMER" nativeL
declare function f1:CUSTOMER() as schema-element(t6:CUSTOMER) * external;
declare namespace f2="ld:TKAppLiquidDataApp/ADDRESS";
import schema namespace t4 = "ld:TKAppLiquidDataApp/ADDRESS" at "ld:TKAppLiquidDataApp/schemas/ADDRESS.xsd";
(::pragma function <f:function xmlns:f="urn:annotations.ld.bea.com" kind="navigate" roleName="ADDRESS"/>:)
declare function f1:getADDRESSs($pk as element(t6:CUSTOMER)) as element(t4:ADDRESS)*
{
  for $fk in f2:ADDRESS()
  where $pk/CUSTOMER_ID eq $fk/CUSTOMER_ID
  return $fk
};

```

You can customize color coding through the Preferences dialog (Tools → Preferences).

Code Complete

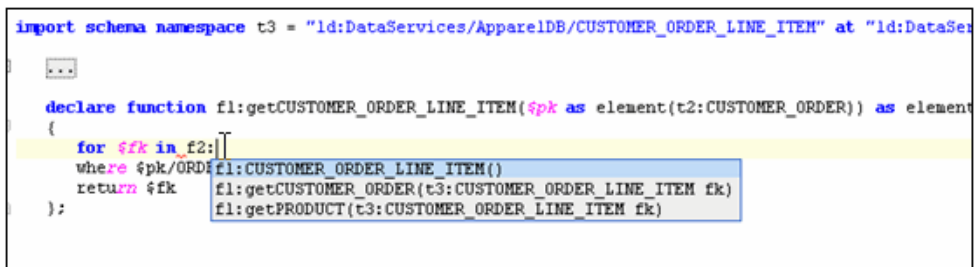
When working with Source View you can use WebLogic Workshop function completion feature.

Completing Functions

If you know the namespace prefix, you can activate the function-completion mechanism.

Function completion is invoked when you type a namespace prefix followed by a colon. The namespace prefix should be bound to a URI corresponding to a data service or XFL file. Alternatively, you can type the prefix followed by a colon followed by Ctrl-Space.

Figure 8-4 Function Completion Using Namespace Prefixes

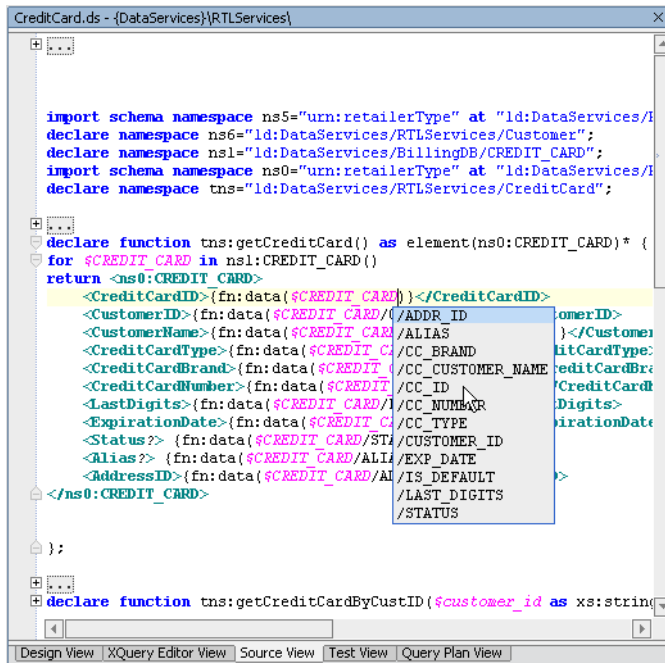


Completing Xpath Expressions

The function-completion facility can also be used to complete Xpath expressions:

1. Position your cursor at the end of the existing path expression.
2. Press the key combination of Ctrl-Space.
3. Select the appropriate element from the pop-up list (Figure 8-5).

Figure 8-5 XPath Code Completion in Source View



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 are flagged on Source View scroll bar (Figure 8-6). Clicking on the error mark takes the cursor to that line of code.

The actual code in question is underlined in red. Mouse-over the text to see the complete error message.

For additional information on editing the WebLogic Workshop properties configuration file see:

http://e-docs.bea.com/workshop/docs70/help/reference/configfiles/conWorkshop_propertiesConfigurationFile.html

Figure 8-6 Syntax Errors Are Flagged and Mouse-over Text Provides Details

```

ElecProduct.ds* - {DataServices}\RTLServices\
ERROR: ld:DataServices/RTLServices/ElecProduct.ds, line 11, column 1: {err}XP0003: Invalid
expression: expecting EOF, found 'declare'
declare namespace ns5="http://temp.openuri.org/SampleApp/Product.xsd";
declare namespace ns4="http://www.openuri.org/";
declare namespace ns3="ld:DataServices/ElectronicsWS/getProductListResponse";
import schema namespace ns2="urn:retailerType" at "ld:DataServices/RTLServices/schemas
declare namespace tns="ld:DataServices/RTLServices/ElecProduct";
declare function tns:getElecProducts() as element(ns2:ELEC_PRODUCT)* {
  for $getProductListResponse in ns3:getProductList(<ns4:getProductList></ns4:getPro
)/ns5:Products/ns5:PRODUCT
return <ns2:ELEC_PRODUCT>
  <ProductID>{fn:data($getProductListResponse/ns5:PRODUCT_ID)}</ProductID>
  <CategoryID>{fn:data($getProductListResponse/ns5:CATEGORY_ID)}</CategoryID>
  <ProductName>{fn:data($getProductListResponse/ns5:PRODUCT_NAME)}</ProductName>
  <ProductDescription>{fn:data($getProductListResponse/ns5:PRODUCT_DESC)}</ProductDe
  <ManufacturerName>{fn:data($getProductListResponse/ns5:MANUFACTURER)}</Manufacture
  <ListPrice>{fn:data($getProductListResponse/ns5:LIST_PRICE)}</ListPrice>
  {fn-bea:rename($getProductListResponse/ns5:AVERAGE_SERVICE_COST, <AverageServiceCo
</ns2:ELEC_PRODUCT>
};

```

If you would like Source View to provide code completion and error highlighting for additional classes, you can edit the `workshop.properties` file to add class files or JAR files to the `paths.classPath` property, then restart WebLogic Workshop.

Working with XQuery Source

Best Practices and Advanced Topics

This section contains general guidelines and patterns for creating a BEA Aqualogic Data Services Platform (DSP) services layer. The following topics are covered:

- [Using a Layered Data Integration and Transformation Approach](#)
- [Using Inverse Functions to Improve Performance During Updates](#)
- [Leveraging Data Service Reusability](#)
- [Modeling Relationships](#)

Using a Layered Data Integration and Transformation Approach

When planning a data service deployment, it is helpful to think of the data service layer in terms of an assembly line. In an assembly line, a product is built incrementally as it passes through a series of machines or assemblers that specialize in an aspect of the fabrication of the product.

Similarly, a well-designed data services layer transforms input (source data) into output (structured information) incrementally, through a series of small transformations. Such a design eases development and maintenance of the data services and increases the opportunity for reuse.

Note: Keep in mind that a multi-level data service implementation model described here is flattened when the data services are compiled for deployment. That is, adding a conceptual layers does not add overhead to the data integration work performed by the DSP deployment, and therefore does not affect performance.

By this design, distinct subsets of data services comprise sub-layers in the overall transformation layer. As data passes from layer to layer data is transformed from a more generalized state to a more application-specific state.

To further illustrate this design, consider a deployment with the following sublayers:

- **Raw data layer.** The first sublayer (that is, the first one to touch the raw data) is the physical data services layer. This layer exists in any DSP deployment, whether or not data is further transformed. The data services in this layer are generated when you import metadata for a data source. A physical data service and its XML type should not be modified other than to synchronize with the source data. See [“Updating Data Source Metadata” on page 3-67](#) for details on data synchronization.)
- **Data normalization.** The second sublayer of data services should normalize the data while retaining the data shape as imported. For example, it can change element names (that is, tag names) to make them consistent with other sources and make minor modifications to data values, for example, concatenating names or adjusting time values for a time zone or other cast-like operation.
- **Data integration.** Data services in the next sublayer can then use the normalized data to represent integrated business entities in the data domain, such as an a unified view of a customer. The data services can unify data sources, for example, or change the shape of the data in any way desired. Another way to look at this operation is as the creation of a virtual database from disparate data sources and other business logic.

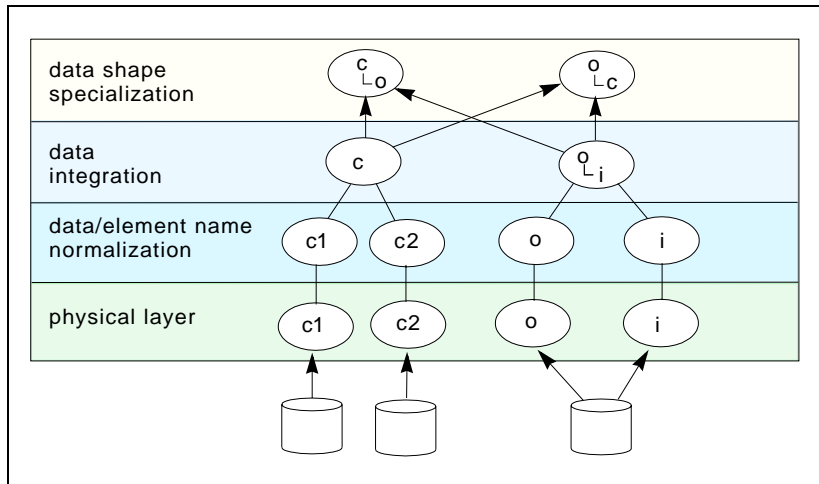
This sublayer does most of what might be called the integration work of the overall data services layer; it is where the integration logic and predicates and primary relationships are specified. (In small projects, this layer may be combined with the second sublayer. That is, it would contain data services that both normalize the data and define data shapes for the integration layer.)

- **Data specialization.** A final sublayer customizes information specifically for applications. This layer, which can be thought of as the extended services layer, tailors information in a way that makes sense to particular applications or types of applications, such as executive dashboards, sales portals, or HR applications. For example, it might specify nesting in its data shape a way that is useful for particular applications, such as having order items as a child of a customer item or, on the other hand, customers as a child of orders (as shown in [Figure 9-1](#)).

For very large database sources, instead of creating a single master data service, it is best to decide what a client application needs and build corresponding, minimal data services. The concept is to build client-specific data services from a manageable number of views that query a reasonable number of data sources, providing an abstraction from the lowest level and most common relationships while keeping the overall view reasonably simple. DSP also provides a metadata API that allows client

applications to discover relationships between data services at runtime, allowing applications to navigate the data services without the need for a master data service.

Figure 9-1 Layered Data Services Design Strategy



The most significant benefit of this approach is that it increases the opportunity for reuse within the overall data services layer. As shown in [Figure 9-1](#), once you have defined a single form of a business entity (such as a customer) in a data service dedicated to the task, you can have multiple application-specific data services use the information without having to repeat data normalization and integration tasks. An additional benefit is that it aids maintenance because there is a clear separation of concerns between the data service layers.

Using Inverse Functions to Improve Performance During Updates

When dealing with disparate data sources it is often necessary to normalize data during updates. Typical normalization includes simple type casting, currency, weights and measures, handling of composite keys, and text and numeric formatting.

While transformational functions are easy to create in XQuery, such functions do not automatically take advantage of the processing power of underlying sources. This becomes especially noticeable when large amounts of relational data are being manipulated.

You can use inverse functions to retain the benefits of high-performance data processing for your logical data.

Sample Inversible Data

Inverse functions are very useful in several types of commonly encountered situations, described in this section. For this topic you can assume underlying data sources with the following characteristics:

- A `US_EMPLOYEE` table containing information on U.S. employees including employee ID, first name, last name, social security number, hire date (in milliseconds post 1/1/1970), and salary in U.S. dollars.

ID	LNAME (string)	FNAME (string)	HIRED (long)	SALARY (int)
1	Smith	Victor	99500000000	120000
2	Davis	Michael	11000000000	95000

- A `UK_EMPLOYEE` table containing employee ID, full name, hired date, and salary in British sterling.

ID (int)	FULLNAME (string)	HIRED (long)	SALARY (int)
3	Jones, Paul	99000000000	60000
4	Williams, John	99100000000	55000

- Employee IDs are unique and normalized across the enterprise.

The `US_EMPLOYEE` and `UK_EMPLOYEE` tables are accessible through two functions in a logical data service: `US_EMPLOYEE()` and `UK_EMPLOYEE()`.

Considerations When Running Queries Against Logical Data

Here are several examples where running queries against logical data can result in noticeably degraded performance when compared with operations against the physical data itself:

- A logical data service has a `fullname()` function that concatenates `first_name` and `last_name` elements. Any attempts to sort by `fullname` would be penalized by the required retrieval of information on all customers, followed by local processing of the returned results.
- A `CUSTOMER` table contains a `customer_since` column of type `long`. You have built a `CustomerProfile` logical data service and created a Java transformational function that converts a simple (atomic) datatype from `long` to `xs:date`.

While a function collecting the names of customers entered after a particular date would succeed, the results would not be optimized. In other words, the processing required by the function would not take advantage of the underlying database's inherent processing power. If a large number of records were involved, the performance impact could be considerable.

Situations Where Inverse Functions Can Improve Performance

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

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

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

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

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

Additional Inverse Function Scenarios

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

- **Type mismatches.** A UK employees database stores date of hire as an integer number; the U.S. employees database stores hire dates in a `datetime()` format. You can convert the integer values to `timedate`, but then searching on hire date would require fetching every record in the database and sorting at the middleware layer. So, in addition, you could use inverse functions.
- **Data Normalization.** In order to avoid confusion of UK and U.S. employees, a data service function prepends a country code to the employee IDs of both groups. Again, sorting based on these values will be time consuming since the processing cannot be achieved on the backend without modifying the underlying data.
- **Data Conversion.** There are many cases where values need to be converted to their inverse based on established formulas. For example it could be requirement the application retrieve customers by date using the `xs:dateTime` rather than as a numeric. In this way users could supply date information in a variety of formats.

The data architect creates the following XQuery function:

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

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

Improving Performance Using Inverse Functions: an Example

Taking the first example in [“Considerations When Running Queries Against Logical Data” on page 9-4](#), it is clear that performance would be adversely affected when running the fullname() function against large data sets.

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

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

Often complimentary inverse functions are needed. For example, FahrenheitToCentigrade() and centigradeToFahrenheit() would be inverses of each other. Complimentary inverse functions are also needed to support fullname().

In addition to creating inverse functions, you also need to identify inverse functions as part of the metadata import process. The import process is described in [Chapter 3, “Obtaining Enterprise Metadata.”](#) The specific application of this process for inverse functions is described in [“Step 4: Configure Inverse Functions” on page 9-9](#).

Deconstructing Composite Keys

The RTLApp contains several examples of inverse functions. In the case of the `fullname()` function, custom Java code provides the underlying inverse function logic. The following actions were involved in creating this example:

- Make sure underlying data sources are available.
- Create the underlying Java functions.
- Import metadata based on those functions.
- Create additional XFL functions required to deconstruct the function written against the virtual data service database.
- Build your data service, including identifying inverse functions.

The following describes the detailed steps involved:

Step 1: Create the necessary programming logic

The string manipulation logic needed by the inverse function is in the following Java file in the RTLApp:

```
DataServices/Demo/InverseFunction/functions/LastNameFirstName.java
```

This file defines several straightforward string manipulation functions.

Listing 9-1 String Manipulation Functions in RTLApp's LastNameFirstName.java

```
package Demo.InverseFunction.functions;

public class LastNameFirstName
{
    public static String mkname(String ln, String fn) { return ln + ", " + fn; }

    public static String fname(String name) {
        return name.substring( name.indexOf(',') + 2);
    }

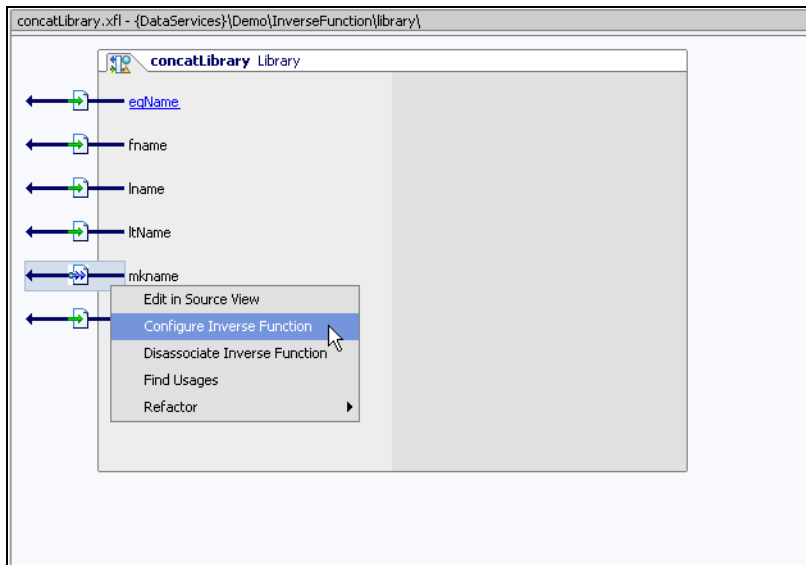
    public static String lname(String name) {
        int k = name.indexOf(',');
        return name.substring( 0, k );
    }
}
```

In [Listing 9-1](#) the function `mkname()` simply concatenates first and last name. The `fname()` and `lname()` functions deconstruct the resulting full name using the required comma in the `mkname` string as the marker identifying the separation between first and last names.

Step 2: Importing Java Function Metadata

After you have compiled your Java function you can import metadata from its class file, in this case `LastNameFirstName.class`. The resulting functions will be imported into an XML file library (XFL) named `concatLibrary.xfl`. [Figure 9-2](#) shows the resulting XFL as well as the right-click options available for the `mkname()` function.

Figure 9-2 Imported Metadata from the `LastNameFirstName.class`



Step 3: Add Functionality to Your XFL File

As is often the case, some additional programming logic is necessary. In this case two functions need to be added to the `concatLibrary` XFL file:

- A function — `precedesName()` — returns a Boolean based on a comparison of two names. First a determination is made as to whether the first `lname` (`x1`) precedes ("is less than") or is the same as ("is equal to") the second `lname` (`x2`). If the names are identical then a similar comparison is made between `fname`. The function returns `True` if conditions are fulfilled.

```
declare function f1:precedesName($x1 as xsd:string?, $x2 as xsd:string?) as
xsd:boolean? {
```

```

    f1:lname($x1) lt f1:lname($x2) or ( (f1:lname($x1) eq f1:lname($x2))
    and (f1:fname($x1) lt f1:fname($x2)) )
};

```

This function is necessary in order to retrieve an ordered list of names from an inverse function.

- A function — `eqName()` — comparing names and reporting through a Boolean whether the names are identical.

```

declare function f1:eqName($x1 as xsd:string?, $x2 as xsd:string?) as
xsd:boolean? {
    (f1:lname($x1) eq f1:lname($x2) and f1:fname($x1) eq f1:fname($x2))
};

```

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

To improve code readability by making a change to the `mkname()` function. Replace the `$x1` and `$x2` variables with `$lastName` and `$firstName`, respectively. When you are done the function appears as:

```

declare function f1:mkname($lastName as xsd:string?, $firstName as
xsd:string?) as xsd:string? external;

```

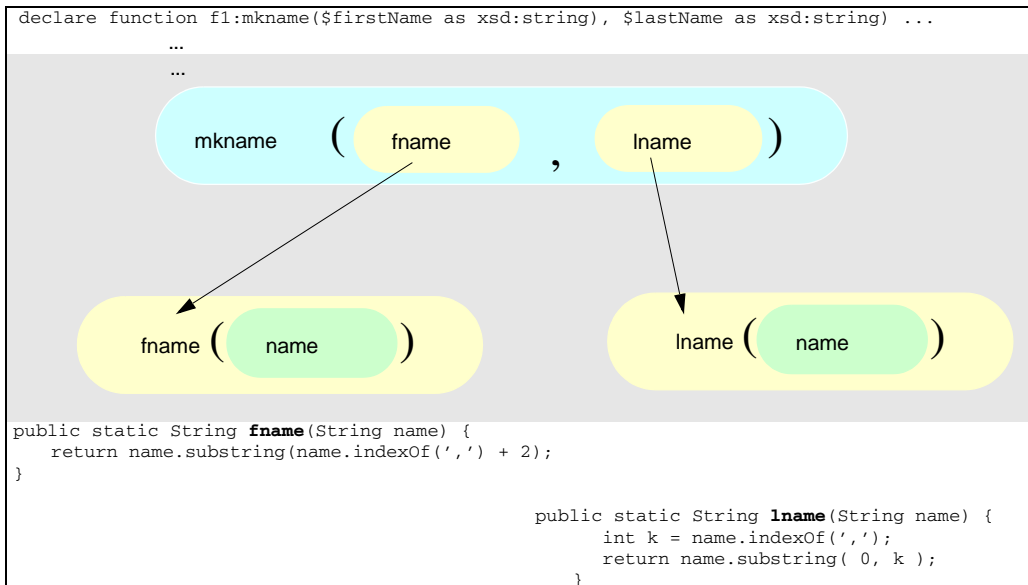
The benefits for doing this become apparent in the next step.

Step 4: Configure Inverse Functions

Since all the functions in `concatLibrary.xfl` have simple parameter types, you could create inverses for each. In this example you only need inverse functions to enable the XQuery engine to deconstruct the `mkname()` function into its component operations.

For each parameter in the `mkname()` function an inverse function is identified. A simplified view of the operation and relevant code can be seen in [Figure 9-3](#).

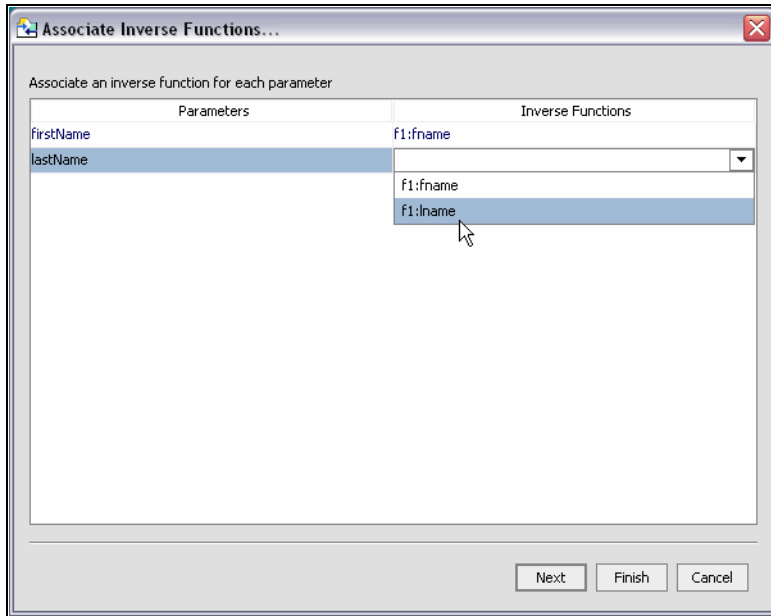
Figure 9-3 Inverse Functions Associated With mName Concatenation Function



In XFL Design View you can association the parameters of functions whose input and output types are atomic with inverse functions. To do this right-click on a function. The option Configure Inverse Function (shown in [Figure 9-2](#)) is available for functions that qualify.

[Figure 9-4](#) illustrates the association of parameters with inverse functions.

Figure 9-4 Configuring Inverse Functions for mName



Step 5: Configuring Conditions for Transformational Functions

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

Associating a particular conditional (such as "is greater-than") with a transformational function allows the XQuery engine to substitute such custom logic for a simple conditional.

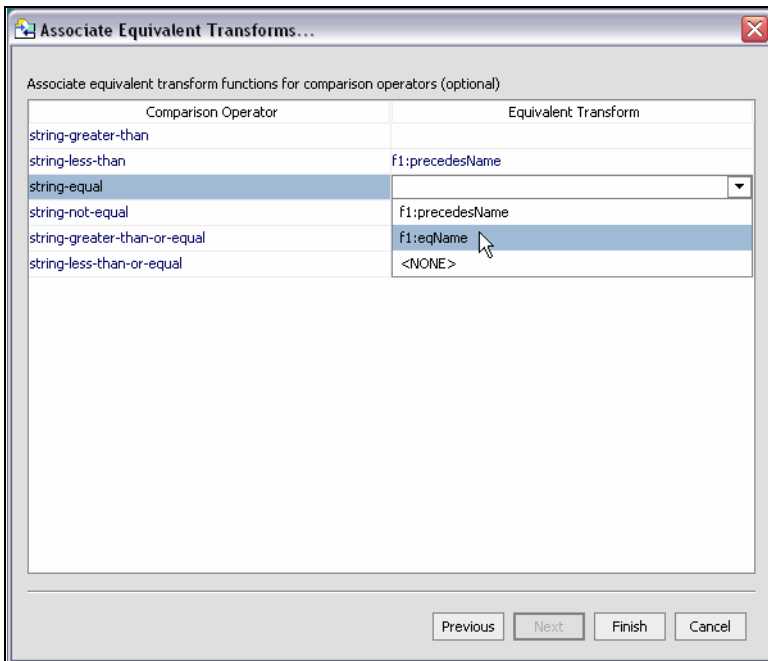
You can associate comparison operators with *transformational functions*. As is always the case with DSP, the original source of the function does not matter. It could be created in your data service, in an XFL, or externally in a Java or other routine. In the case of this example the transformational function, eqName(), is in an XFL file.

Figure 9-5 Conditional Operators That Can be Used for Equivalent Transforms

<code>string-greater-than (gt)</code>	<code>string-not-equal (ne)</code>
<code>string-less-than (lt)</code>	<code>string-greater-than-or-equal (ge)</code>
<code>string-equal (eq)</code>	<code>string-less-than-or-equal (le)</code>

The next step is to match comparison operators with an equivalent transform functions. Custom logic is needed to support pushdown operations in conjunction with comparison operations. In the current exercise the `string-less-than (lt)` operation is associated with the XFL `precedesName()` function; the `string-equal (eq)` operation is associated with the `eqName()` functions. When your query function encounters these operators, the corresponding custom logic is substituted.

Figure 9-6 Associating an Equivalent Transform With an Operator



Two equivalent transform functions were created in the `concatLibrary.xfl`. The first, `precedesName()`, tests names to make sure they are in ascending order. The second, `eqName()` simply compares two first names and two last names and makes sure they are identical.

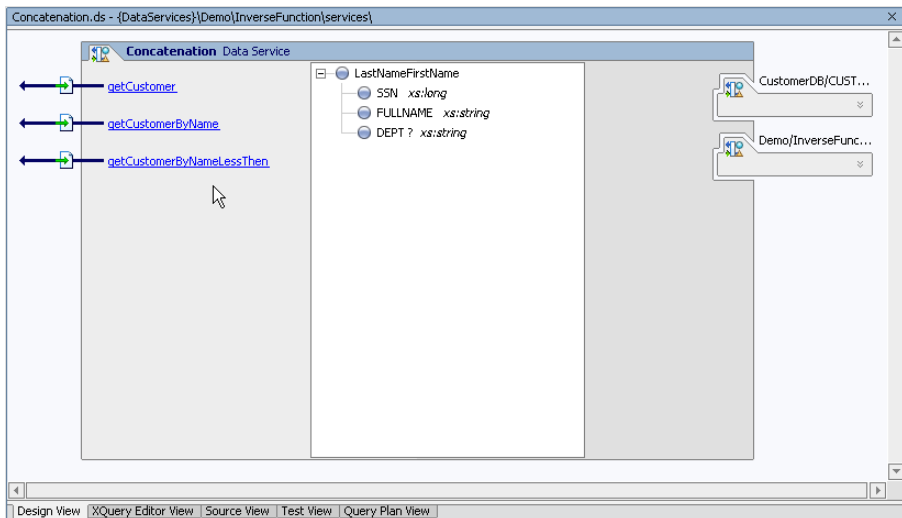
Step 6: Create Your Data Service

Now you are ready to create a data service that will contain functions such as `getCustomerByName()` and `getCustomerByNameLessThan()`. In reviewing available facilities, you have:

- Several custom Java functions which you added in the `concatLibrary` XFL file.
- XFL routines that you associated with conditional operators.

The data service, called `Concatenation`, uses a XML type associated with the `LastNameFirstName.xsd` schema.

Figure 9-7 Concatenation Data Service



This schema could have been created through the XQuery Editor, through the DSP schema editor, or through a third-party editing tool. (Notice also that one of the building blocks of your data service is the `concatLibrary` XFL.)

The familiar `getCustomer()` function operates somewhat differently in this example.

```
declare function tns:getCustomer() as element(ns0:LastNameFirstName)* {
  for $CUSTOMER in ns1:CUSTOMER()
  return
  <ns0:LastNameFirstName>
    <SSN> { fn:data ($CUSTOMER/SSN) } </SSN>

  <FULLNAME> { ns2:mkname ( fn:data ($CUSTOMER/LAST_NAME) , fn:data ($CUSTOMER/FIRST
_NAME) ) } </FULLNAME>
```

```

        <DEPT?></DEPT>
    </ns0:LastNameFirstName>

};

```

Using a U.S. social security number as the primary key, the routine relies on the Java-based `mkName()` function to retrieve first and last name from the data source and concatenate the results into a "fullname".

The `getCustomerByName()` routine takes `fullname` as input and returns `$LastNameFullName` and the associated social security number.

```

declare function tns:getCustomerByName($Name as xs:string) as
element(ns0:LastNameFirstName)* {
    for $LastNameFirstName in tns:getCustomer()
    where $LastNameFirstName/FULLNAME eq $Name
    return $LastNameFirstName
};

```

In the above code the equality (`eq`) test is evaluated by substituting the logic of the `concatLibrary eqName()` function.

The `getCustomerByNameLessThan()` routine uses the substitute condition logic available for the `lt` operator. First the routine.

```

declare function tns:getCustomerByNameLessThen($Name as xs:string) as
element(ns0:LastNameFirstName)* {
    for $LastNameFirstName in tns:getCustomer()
    where $Name lt $LastNameFirstName/FULLNAME
    return $LastNameFirstName
};

```

The logic of the less-than substitution can be derived from examining `LastNameFirstName.java` and the `concatLibrary`. The raw processing is containing in the Java file:

```

    public static boolean ltName(String name1, String name2) {
        String ln1 = lname(name1);
        String ln2 = lname(name2);
        return (ln1.compareTo(ln2)<0) || (ln1.equals(ln2) &&
fname(name1).compareTo(fname(name2))<0);
    }

```

The XFL function, `precedesName()` is:

```

declare function f1:precedesName($x1 as xsd:string?, $x2 as xsd:string?)
as xsd:boolean? {
    f1:lname($x1) lt f1:lname($x2) or ( (f1:lname($x1) eq f1:lname($x2))

```

```

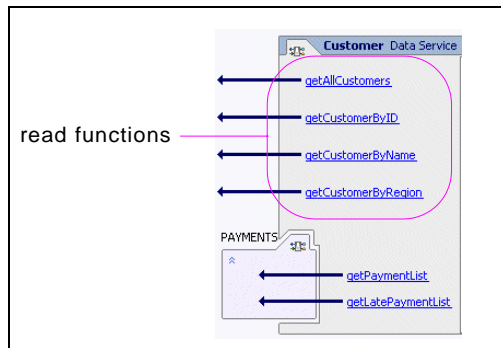
        and (f1:fname($x1) lt f1:fname($x2)) )
};

```

Leveraging Data Service Reusability

A typical design pattern within a logical data service is to have a single read function that defines the data shape without filtering conditions. The function may be declared private so that it can only be called by other functions within the same data service. Also, it is the only function containing integration logic. This is known as the *decomposition function*. By default the decomposition function is the first function listed in Design View of your logical data service. However you can, through the Properties Editor, set the decomposition function to be any public or private function in your data service. Additional functions, either in the same data service or in other data services, can use the private function to specify filtering criteria. [Figure 9-8](#) shows the design view of a data service exhibiting this pattern.

Figure 9-8 Customer Data Service functions



The following XQuery sample demonstrates the mechanics behind data service reuse. This function, `getCustomerByName()`, filters instances based on the customer name:

```

declare function l1:getCustomerByName($c_name as xs:string)
as element(t1:CUSTOMER) *
{
  for $c in l1:getAllCustomers()
  where $c/CUSTOMERNAME eq $c_name
  return $c
};

```

The `getAllCustomers()` function, in turn, would assemble the data shape for the returned data and provide join logic and transformation, as shown its return clause:

```
...
return
  <t1:CUSTOMER>
    <CUSTOMERID>{fn:data($c/CUSTOMERID)}</CUSTOMERID>
    <CUSTOMERNAME>{fn:data($c/CUSTOMERNAME)}</CUSTOMERNAME>
    {
      for $a in f2:ADDRESS()
      where $c/CUSTOMERID eq $a/CUSTOMERID
      return
        <ADDRESS>
          <STREET>{fn:data($a/RTL_STREET)}</STREET>
          <CITY>{fn:data($a/RTL_CITY)}</CITY>
          <STATE>{fn:data($a/RTL_STATE)}</STATE>
        </ADDRESS>
    }
  </t1:CUSTOMER>
```

Keep in mind that client application themselves can specify filtering conditions on a data service function call. Therefore, you as the data service designer can choose whether to have broadly defined data access functions (that is, without filter conditions), and let the client to apply filtering as desired, or narrowly by defining the criteria in the API.

Note: All functions whose bodies are some variation of a *flwor* (for-let-where-order-return) statement should be declared to return a plural rather than a singular result; for example:

```
element (purchase_order) *
```

rather than:

```
element (purchase_order)
```

applies to both read and navigation functions.

The reason for declaring returns to be plural is that the XQuery compiler wants to be sure that you indeed deliver the declared result at runtime. If it cannot determine that something is singular it inserts a runtime *typematch* operator in the query evaluation plan. You won't get the wrong result, but that operator will cause important pushdown-related optimizations (function unfolding) to be defeated.

Modeling Relationships

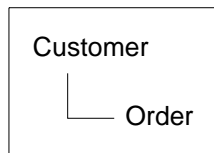
There are several ways to implement a logical relationship between distinct units of information with data services:

- Data shape containment
- Navigation functions

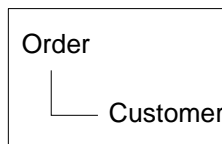
When containment is implemented in the data shape, it means that the XML data type of the data service is nested; that is, one element is the parent of another element. For example, in the following sample a customer element contains orders:

```
<customer>
  <customerId>...</customerId>
  <customerName>...</customerName>
  <orders>
    <order>...</order>
    <orderId>...</orderId>
  </orders>
</customer>
```

A diagram of this XML structure would be:

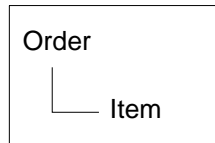


In this type of containment, the parent-child hierarchy between the customer and order is locked into the data shape. This nesting might make sense for most applications, particularly those oriented by customer. However, other applications may benefit from an orders-oriented view of the data. For example, an inventory application may prefer to work with the data in an orders-first fashion, with the customer as a child element of each order.



Conceptually, in this case it could also be said that an Order is not existence-dependent on a Customer. If a Customer record is deleted, it may not necessarily follow that the customer's order should be deleted as well.

Alternatively, other relationships do not require this type of hierarchical flexibility. In most cases, this also implies that the business entity's existence does depend on the existence of the parent. For example, consider an order that contains items.



In most logical data models, it would not make sense to have an item outside of the context of the order that contains it. When deleting an order, it is safe to say that composing order items would need to be deleted as well.

The choice when modeling such containment either through a relationship or through data shape nesting is informed by these considerations. When choosing whether to model containment either through data shape nesting or using relationships, it is recommended that:

- Existence-dependent entities are modeled as nested elements.
- Existence-independent entities are modeled as relationships.

By modeling independent entities with bi-directional relationships, data service users and designers can easily specialize the logical hierarchy between business entities as best suited for their applications.