AquaLogic Data Services Platform™ Tutorial: Part II

A Guide to Developing BEA AquaLogic Data Services Platform (DSP) Projects

Note: **This tutorial is based in large part on a guide originally developed for enterprises evaluating Data Services Platform for specific requirements. In some cases illustrations, directories, and paths reference Liquid Data, the previous name of the Data Services Platform.**

bea®

Contents

About This Document

Welcome to the *AquaLogic Data Services Platform (DSP) Samples Tutorial*. In this document, you are provided with step-by-step instructions that show how you can use DSP to solve the types of data integration problems frequently faced by Information Technology (IT) managers and staff. These issues include:

> What is the best way to normalize data drawn from widely divergent sources?
>
> Having normalized the data, can you access it, ideally through a single point of access?
>
> After you define a single point of access, can you develop reusable queries that are easily tested, stored, and retrieved?
>
> After you develop your query set, can you easily incorporate results into widely available applications?

Other questions may occur. Is the data-rich solution scalable? Is it reusable throughout the enterprise? Are the original data sources largely transparent to the application — or do they become an issue each time you want to make a minor adjustments to queries or underlying data sources?

Document Organization

This guide is organized into 35 lessons that illustrate several Data Services Platform capabilities:

> **Data service development.** In which you specify the query functions that DSP will use to access, aggregate, and transform distributed, disparate data into a unified view. In this stage, you also specify the XML type that defines the data view that will be available to client-side applications.
>
> **Data modeling.** In which you define a graphical representation of data resource relationships and functions.
>
> **Client-side development.** In which you define an environment for retrieving data results.

Each lesson in the tutorial consists of an overview plus "labs" that demonstrate DSP's capabilities on a topic-by-topic basis. Each lab is structured as a series of procedural steps that details the specific actions needed to complete that part of the demonstration.

The lessons are divided into two parts:

> **Part 1: Core Training.** Includes Lessons 1 through 16, which illustrate the DSP capabilities that are most commonly used.
>
> **Part 2: Power-User Training.** Includes Lessons 17 through 35; these illustrate DSP's more advanced capabilities.

Note: The lessons build on each other and must be completed in sequential order.

Technical Prerequisites

The lessons within this guide require a familiarity with the following topics: data integration and aggregation concepts, the BEA WebLogic® Platform™ (particularly WebLogic Server and WebLogic Workshop), Java, query concepts, and the environment in which you will install and use DSP.

For some lessons, an ability to understand XQuery is helpful.

System Requirements

To complete the lessons, your computer requires:

Server:	BEA WebLogic Server
Application:	AquaLogic Data Services Platform 2.01
Operating System:	Windows 2000 or Windows XP
Memory:	512 MB RAM minimum; 1 GB RAM recommended
Browser:	Internet Explorer 6 or higher

Data Sources Used Within These Lessons

The *Data Services Platform Samples Tutorial* builds data services that draw on a variety of underlying data sources. These data sources, which are provided with the product, are described in the following table:

Data Source Types and Sources Utilized by the DSP Samples Tutorial

Data Type	Data Source	Data
Relational	Customer Relationship Management (CRM) RTLCUSTOMER database	Customer and credit card data
Relational	Order Management System (OMS) RTLAPPLOMS database	Apparel product, order, and order line data
Relational	Order Management System (OMS) RTLELECOMS database	Electronics product, order, and order line data
Relational	RTLSERVICE database	Customer service data, organized in a single Service Case table
Web service	CreditRatingWS	Credit rating data
Stored procedure	GETCREDITRATING_SP	Customer credit rating information
Java function	Functions.DSML	Java function enabling LDAP access
Java function	Functions.excel_jcom	Excel spreadsheet data, via JCOM
Java function	Functions.CreditCardClient	Customer credit card information, via an XMLBean
XML files	ProductUNSPSC.xsd	Third-party product information
Flat file	Valuation.csv	Data received from an internal department that deals with customer scoring and valuation models

Related Information

In addition to the material covered in this guide, you may want to review the wealth of resources available at the BEA web site, WebLogic developer site, and third-party sites. Information at these sites includes datasheets, product brochures, customer testimonials, product documentation, code samples, white papers, and more.

For more information about Java and XQuery, refer to the following sources:

The Sun Microsystems, Inc. Java site at:

http://java.sun.com/

The World Wide Web Consortium XML Query section at:

http://www.w3.org/XML/Query

For more information about BEA products, refer to the following sources:

DSP documentation site at:

http://edocs.bea.com/aldsp/docs20/index.html

BEA e-docs documentation site at:

http://e-docs.bea.com/

BEA online community for WebLogic developers at:

http://dev2dev.bea.com

Part 2: Power-User Training

In the DSP *Samples Tutorial Part I* (Core Training), you were introduced to the features, functions, and tools necessary to build, cache, and secure data services within a DSP application. In Part 2, you will build upon that knowledge to:

Build queries in both XQuery Editor View and Source View.

Create models for logical data services.

Run ad hoc queries.

Use update overrides to perform custom data manipulations, update web services, and overwrite SQL updates.

Use the automatically generated Query Plan.

Re-use XQuery code.

Configure alternative sources for unavailable data sources.

Use SQL Exits to enable retrieving data from an SQL statement.

Enable fine-grained caching.

Enable element-level security.

Access data in stored procedures, Java functions, XML files, and flat files.

Create an XQuery function library.

Lesson 18 Building XQueries in XQuery Editor View

In concrete terms, a data service is simply a file that contains XML Query (XQuery) instructions for retrieving, aggregating, and transforming data. Essentially you create a query function by:

Integrating physical and logical data sources into the query.

Mapping data sources to the data service's Return type.

Creating XQuery statements that include conditions, parameters, functions, and expressions.

You can also modify the Return type, either within XQuery Editor View or using an external tool.

In this lesson, you will use XQuery Editor View to develop a variety of XQuery instructions.

Objectives

After completing this lesson, you will be able to:

Use the graphical XQuery Editor View to create parameterized, string, and date functions; outer joins, aggregate, and order by and constant expressions.

Use the XQuery Function Palette to add built-in XQuery functions to a query.

Overview

XQuery Editor View provides a graphical, drag-and-drop approach to constructing queries. Using XQuery Editor View, you can:

View and modify the data service's Return type, whose shape is defined by the data service's XML Type.

View, add, modify, and delete the function calls from other physical and logical data services that define which data source(s) will be queried.

View, add, and delete the source-to-target mappings that define which data will be made available to consuming applications.

View, add, modify, and delete the parameters, expressions, and conditions that define how the data will be processed.

Changes that you make in XQuery Editor View are immediately reflected in Source View. Similarly, changes you make in Source View will be immediately effective in XQuery Editor View.

Lab 18.1   Importing Schemas for Query Development

To simplify development time in this lesson you will use ready-made schemas that define a data service's Return type.

Objectives

In this lab, you will:

> Create a folder to organize all the queries that you will create in this lesson and the next.
>
> Import the schemas that you will use in those queries.

Instructions

1.   Create a new folder in the DataServices project folder, and name it MyQueries.

   a.   Right-click the MyQueries folder and choose Import.

   b.   Navigate to the <beahome>\weblogic81\samples\LiquidData\EvalGuide\MyQueries folder, select the schemas folder, and click Import. This will automatically create a folder named schemas, and appropriate .xsd files, within the MyQueries directory. These .xsd files will be used to determine the Return type for all queries developed in this lesson.

Lab 18.2   Creating Source-to-Target Mappings

Every function within a logical data service includes source-to-target mappings that define what results will be returned by the function. As described in Part I, there are several types of mappings:

> A *simple mapping* means that you are mapping simple source node elements to simple elements in the Return type one at a time. You can create a simple mapping by dragging and dropping any element from the source node to its corresponding target element in the Return type. Optional Return type elements do not need to be mapped; otherwise elements in the Return type need to be mapped in order for your query to run.
>
> An *induced mapping* means that a complex element is mapped to a complex element in the Return type. In this gesture the top level complex element in the Return type is ignored (source node name need not match). The editor automatically then maps any child elements (complex or simple) that are an exact match for source node elements.
>
> An *overwrite mapping* replaces a Result type element and all its children (if any) with the source node elements. As an example of the general steps needed to create an overwrite mapping, you would press <Ctrl>, then drag and drop the source node's complex element onto the corresponding element in the Result type. The entire source node's complex element is brought to the Result type, where it completely replaces the target element with the source element.
>
> An *append mapping* adds a simple or complex element (and any children or attributes) as a child of the specified element in the Return type. To create an append mapping, select the source element, then press <Ctrl>+<Shift> while dragging and dropping the source node's element onto the element in the Return type that you want to be the parent of the new element(s).
>
> Alternatively, if you simply want to add a child element to a Return type, you can drag a source element to a complex element in your Return type. The element will be added as a child of the complex element and mapped accordingly.

Objectives

In this lab, you will:

Create four types of mappings.

Review the results.

Instructions

1. Right-click the MyQueries folder, choose New → Data Service, and use CustomerInfo.ds in the Name field.

    1. In Design View, associate the CustomerInfo data service with the CUSTOMER.xsd schema. (The schema is located in the MyQueries\schemas folder.)

    2. Add a new function to the CustomerInfo data service and name it getAllCustomers.
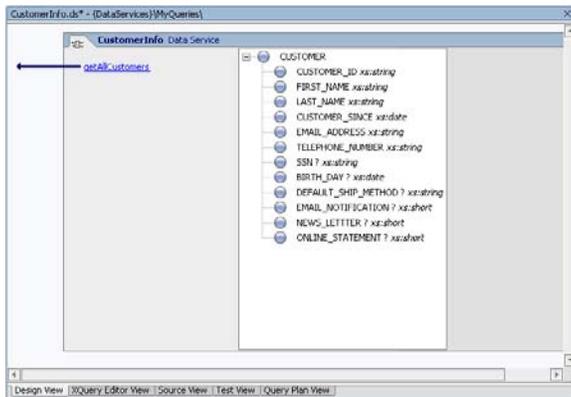
Figure 18-1   Design View of CustomerInfo Data Service

    3. Click the getAllCustomers() function to open XQuery Editor View.

    4. Add a for node to the work area by completing the following steps:

        a. In the Data Services Palette, open the CUSTOMER.ds folder, located in the DataServices\CustomerDB folder.

        b. Drag and drop CUSTOMER()into XQuery Editor View. This creates a For:$CUSTOMER source node.

    5. Create a simple mapping. Drag and drop each element in the CUSTOMER source node onto the corresponding element in the Return type.
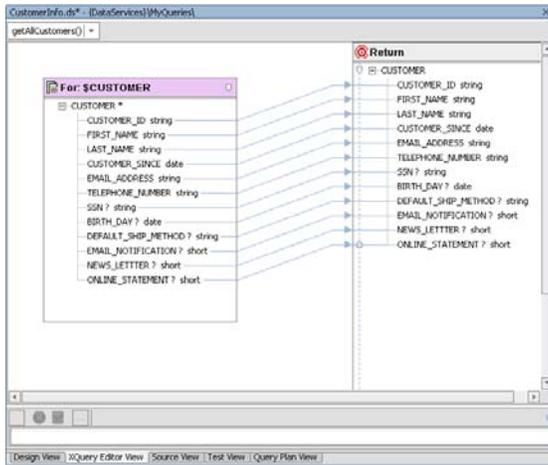
Figure 18-2　Simple Mapping

6. Create an induced mapping, by completing the following steps:

   a. Delete all the simple mappings. (Right-click a map line and select Delete from the pop-up menu.)

   b. Drag and drop the CUSTOMER* element (source node) onto the CUSTOMER element in the Return type.

Notice that the mappings are automatically generated for each element, since the source and target element names are the same.
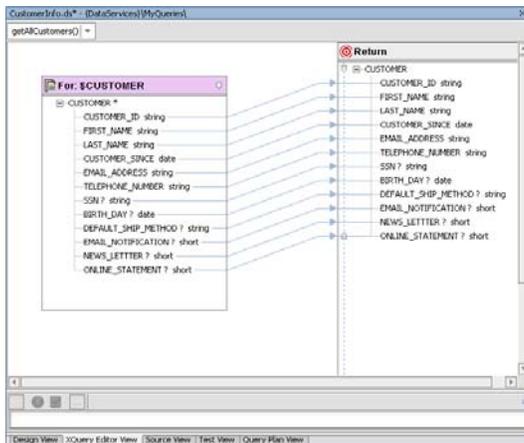
Figure 18-3　Induced Mapping

7. Create an overwrite mapping, by completing the following steps:

   a. In the Return type right-click the CUSTOMER element and choose Add Child Element.

   b. Double-click the NewChildElement, enter Addresses, and press Enter.

   c. In the Data Services Palette, open the ADDRESS.ds icon, which is located in the DataServices\CustomerDB folder.

   d. Drag and drop ADDRESS() into XQuery Editor View.

   e. Press Ctrl, and then drag and drop ADDRESS* element (source node) onto the Addresses element in the Return type.

Notice that the entire complex ADDRESS* element is brought to the target, where it overwrites the element, instead of adding it as a child.
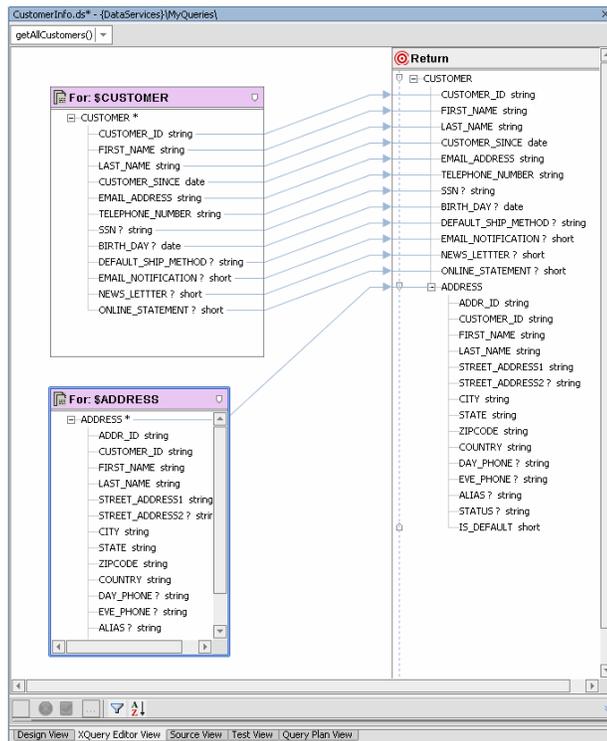
Figure 18-4   Overwrite Mapping

Lab 18.3   Creating a Basic Parameterized Function

A parameterized query lets you filter returned data based on specific criteria, such as a particular order number, customer name, or customer number.

Objectives

In this lab, you will:

>   Create a parameterized function that returns all orders for a particular customer.

>   Test the function.

>   Review the XQuery source code.

Instructions

In Design View: Add a new function to the CustomerInfo data service and name it getCustomerByName.

1.   Click getCustomerByName to open XQuery Editor View for that function.

2.   Add a for node, by completing the following steps:

>   a.   In the Data Services Palette, open the CUSTOMER.ds folder, which is located in the DataServices\CustomerDB folder.

b. Drag and drop CUSTOMER()into XQuery Editor View. This creates a For:$CUSTOMER source node.

3. Create an induced mapping. Drag and drop the CUSTOMER* element (source node) onto the CUSTOMER element in the Return type.

4. Add a parameter, by completing the following steps:

   a. Right-click on a empty spot in XQuery Editor View.

   b. Choose Add Parameter.

   c. Enter FirstName in the Parameter Name field.

   d. Select xs:string as the Primitive Type.

   e. Click OK. (Since the new parameter node may be placed behind the CUSTOMER node, you will need to move the nodes until all are visible.)

5. Add a where clause, by completing the following steps:

   a. Drag and drop the parameter's string element onto FIRST_NAME element (source node). Make sure that you release the mouse button when the FIRST_NAME element is highlighted. This action creates a filter for the FIRST_NAME element based on the parameter that is passed to the function.

   b. Confirm that the where clause is correctly set by clicking the $CUSTOMER source node's header. The Expression Editor will open and you should see the following where clause:

$FirstName = $CUSTOMER/FIRST_NAME
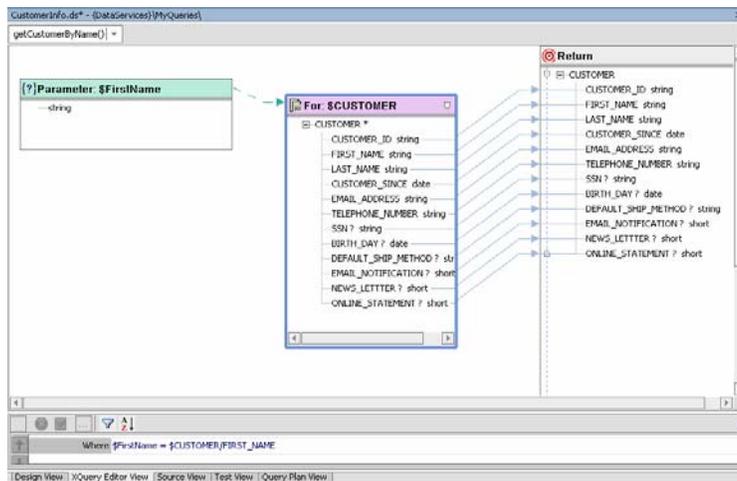
Figure 18-5   First Name Parameter and WHERE Clause

6. Add a second where clause, by completing the following steps:

   a. Add a new parameter, entering LastName, and selecting xs:string as the Primitive Type.

   b. Click the $CUSTOMER node's header. The Expression Editor opens.

   c. Triple-click inside the where field and place your cursor at the very end, after FIRST_NAME.

   d. Select the "and" logical conjunction from the pop-up operator list (the "..." icon). You can now define the where clause to filter data by last name.

   Note: An alternative method is to simply enter "and" in the field.

e.   Click the string element in the second parameter. The variable name $LastName appears at the end of the where clause.

f.   Choose eq: Compare Single Values from the popup operator list.

Note: An alternative method is to simply enter eq in the field.

g.   Click the LAST_NAME element in the For:$CUSTOMER node. You should see the following in the where clause field:

```
$FirstName = $CUSTOMER/FIRST_NAME and $LastName eq $CUSTOMER/LAST_NAME
```

h.   Click the green check button to accept the changes.
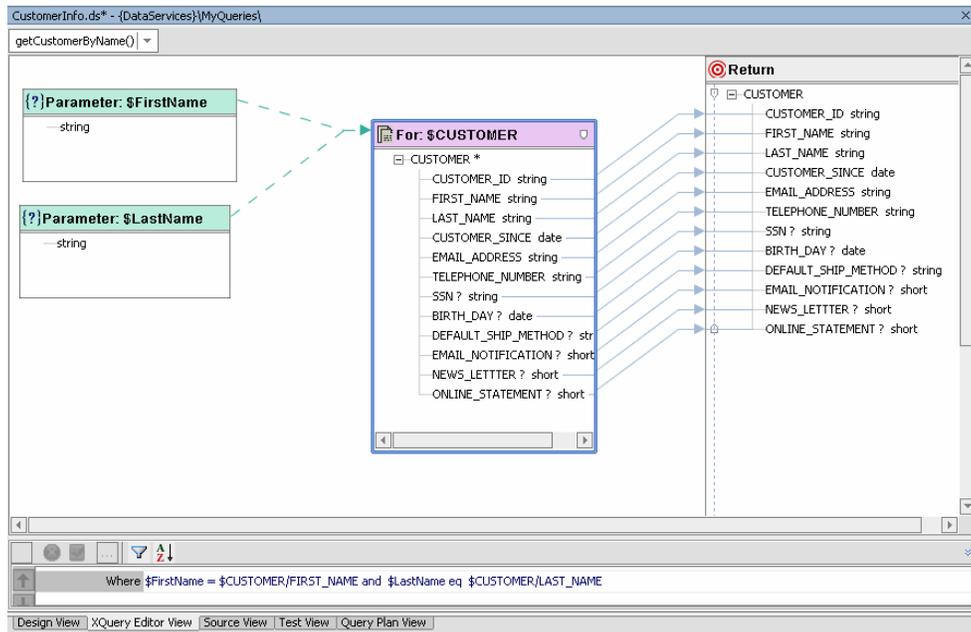
Figure 18-6   XQuery Editor View of Parameterized Query

7.   Test the function, by completing the following steps:

a.   Open CustomerInfo.ds in Test View.

b.   Select getCustomerByName(FirstName, LastName) from the drop-down list.

c.   Enter Jack in $FirstName field.

d.   Enter Black in the $LastName field.

e.   Click Execute.

Confirm the results, which should be as displayed in Figure 18-7.

Figure 18-7   Parameterized Query Results

8. Open CustomerInfo.ds in Source View to view the generated XQuery. The query should be
    similar to that displayed in Figure 18-8. (<u>Note:</u> The automatic namespace assignments may
    not match).
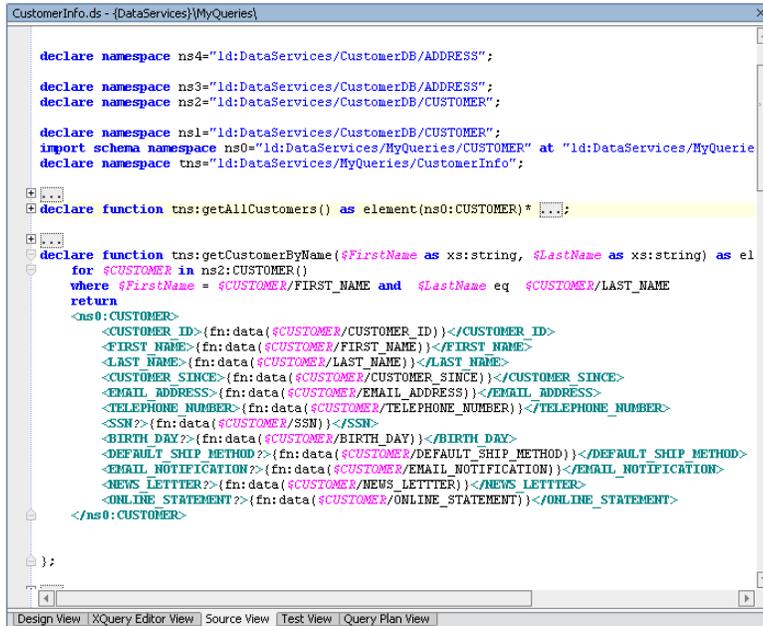
Figure 18-8   Parameterized Function Source Code

Lab 18.4 Creating a String Function with a Built-In XQuery Function

The XQuery language provides more than 100 functions. BEA provides some additional, special purpose functions. In this lab, you will build a query that uses the built-in XQuery startWith() function to create business logic sufficient to retrieve records based on an OR condition.

Objectives

In this lab, you will:

Create a string function that will find customers by their social security number.

Test the function.

Review the XQuery source code.

Instructions

1. Add a new function to the CustomerInfo data service and name it getCustomerBySSN.

2. Click getCustomerBySSN() to open XQuery Editor View to that function.

3. Add a for clause, by completing the following steps:

    a. In the Data Services Palette, open the CUSTOMER.ds folder, which is located in the DataServices\CustomerDB folder.

    b. Drag and drop CUSTOMER()into XQuery Editor View. This creates a For:$CUSTOMER node.

4. Click getCustomerBySSN() to open XQuery Editor View to that function.

5. Create an induced map. Drag and drop the CUSTOMER* element (source) onto the CUSTOMER element in the Return type.

6. Add a new parameter, entering SSN as the Parameter Name, and selecting xs:string as the Primitive Type.

7. Add a where clause that uses a built-in XQuery function, by completing the following steps:

    a. Click the $CUSTOMER node's header. The Expression Editor opens.

    b. Click the Add Where Clause icon.

    c. In XQuery Function Palette, expand the String Functions folder.

    d. Drag and drop the following function into the where clause field.

    ```
    fn:starts-with($arg1 as xs:string?, $arg2 as xs:string?) as xs:boolean
    ```

    e. Confirm that the where clause now includes the following built-in function:

    ```
    fn:starts-with($arg1, $arg2)
    ```

    f. Edit the where clause, so that it reads as follows:

    ```
    fn:starts-with($CUSTOMER/SSN, $SSN)
    ```

    g. Click the green check button to accept the changes.
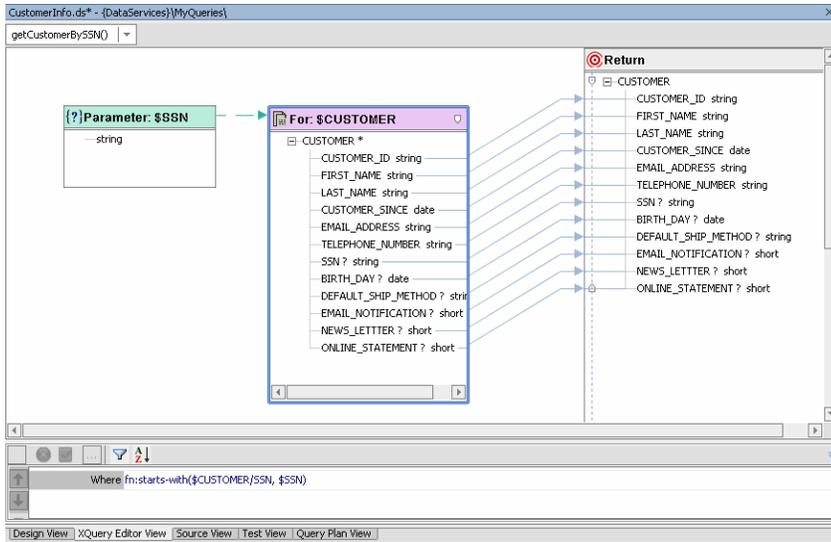
Figure 18-9   Built-In Function Where Clause

8.   Test the function, by completing the following steps:

   a.   Open CustomerInfo.ds in Test View.

   b.   Select getCustomerBySSN from the Function drop-down list.

   c.   Enter 647 in the xs:string SSN field.

   d.   Click Execute.

   e.   Confirm the results, which should be as displayed in Figure 18-10.

Figure 18-10   Built-In Function Test Results

9. Open CustomerInfo.ds in Source View to view the generated XQuery. The query should be similar to that displayed in Figure 18-11. (<u>Note:</u> The automatic namespace assignments may not match).

Figure 18-11   Source View of Built-In String Function

Lab 18.5   Creating a Date Function

A date function lets you retrieve data based on date parameters.

Objectives

In this lab, you will:

> Create a date function that will find customers by the year that they were born.
>
> Test the function.
>
> Review the XQuery source code.

Instructions

1. Add a new function to the CustomerInfo data service and name it getCustomerByBirthYear.

2. Click getCustomerByBirthYear() to open XQuery Editor View to that function.

3. Add a for clause, by completing the following steps:

    a. In the Data Services Palette, open the CUSTOMER.ds folder, which is located in the DataServices\CustomerDB folder.

    b. Drag and drop CUSTOMER()into XQuery Editor View. This creates a for node for the CUSTOMER() function.

4. Create an induced mapping. Drag and drop the CUSTOMER* element (source) onto the CUSTOMER element (Return).

5. Create a new parameter, enter BirthYear as the Parameter Name, and select xs:integer as the Primitive Type.

6. Add a where clause, by completing the following steps:

    a. Click the $CUSTOMER node's header. The Expression Editor opens.

    b. Click the Add Where Clause icon.

    c. In XQuery Function Palette, expand the Duration, Date, and Time Functions folder.

    d. Drag and drop the built-in following function into the where clause field.

    ```
    fn:year-from-date($arg as xs:date?) as xs:integer?
    ```

    e. Confirm that the where clause is as follows:

    ```
    fn:year-from-date($arg)
    ```

    f. Edit the built-in function, so that it reads as:

    ```
    fn:year-from-date($CUSTOMER/BIRTH_DAY) eq $BirthYear
    ```

    g. Click the green check button to accept the changes.
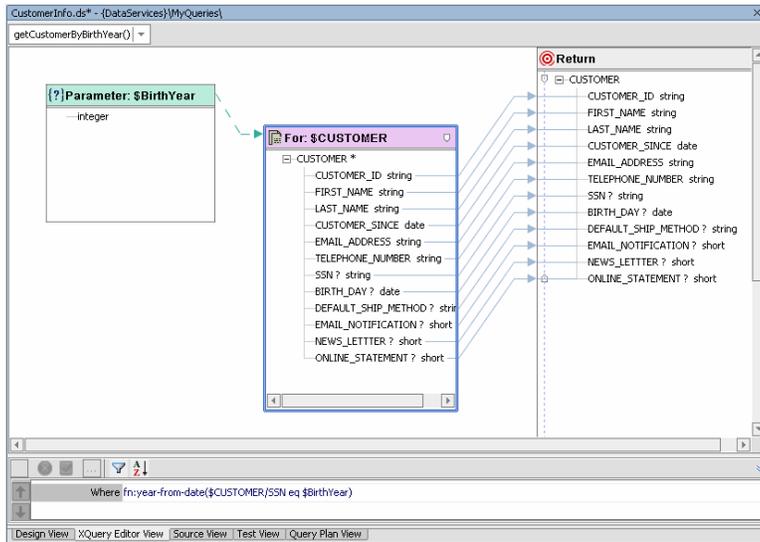
Figure 18-12   Where Clause Using a Built-In Date Function

7.   Test the function, by completing the following steps:

a.   Open CustomerInfo.ds in Test View.

b.   Select getCustomerByBirthYear() from the function drop-down list.

c.   Enter 1970 in the $arg0 field.

d.   Click Execute.

e.   Confirm the results, which should be as displayed in Figure 18-13. There should be five customer profiles returned.
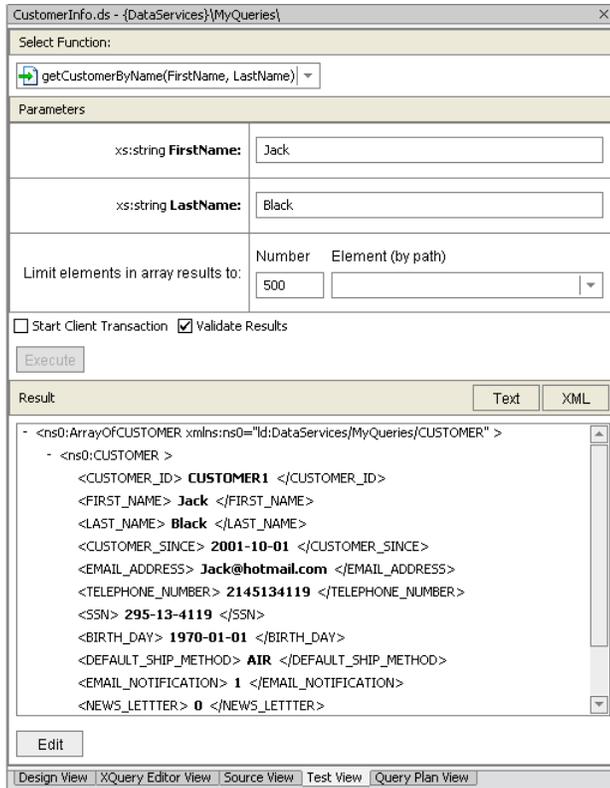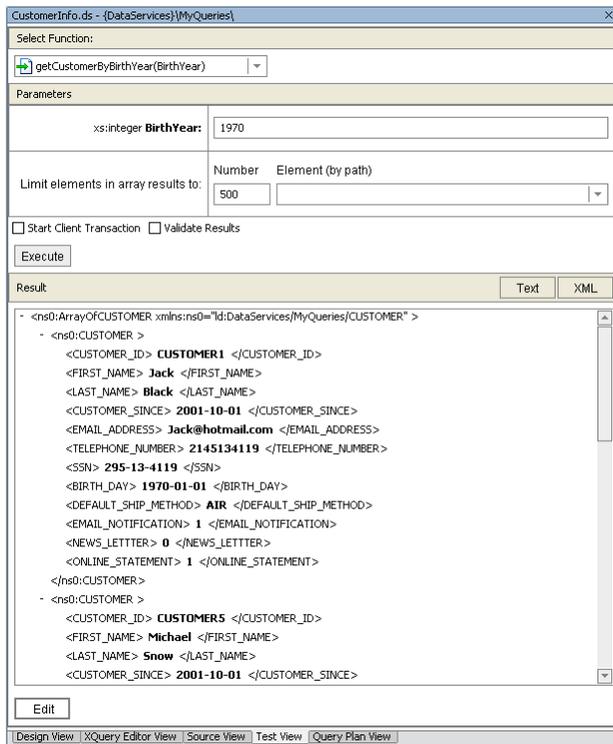

Figure 18-13   Date Function Test Results

8. Open CustomerInfo.ds in Source View to view the generated XQuery. The query should be similar to that displayed in Figure 18-14. (<u>Note:</u> The automatic namespace assignments may not match).


Figure 18-14   Date Function Source View

Lab 18.6   Creating Outer Joins and Order By Expressions

Outer joins return all records from one table even it doesn't contain values that match those in the other table. For example, an outer join of customers and orders reports all customers — even those without orders.

Objectives

In this lab, you will:

Create a function that:

o   Returns customer information and their addresses (there may be more than 1).

o   Nests address information inside customer information.

o   Orders customers by first name and last name, in ascending order.

o   Orders addresses by zip code, in descending order.

Test the function.

Review the XQuery source code.

Instructions

1. Add a new data service to the MyQueries folder and name it CustomerAddresses.

2. Associate the CustomerAddresses() data service with the CUSTOMERADDRESS.xsd schema. The schema is located in the MyQueries\schemas folder.

3. Add a new function to the CustomerAddresses data service and name it getCustomerAddresses.
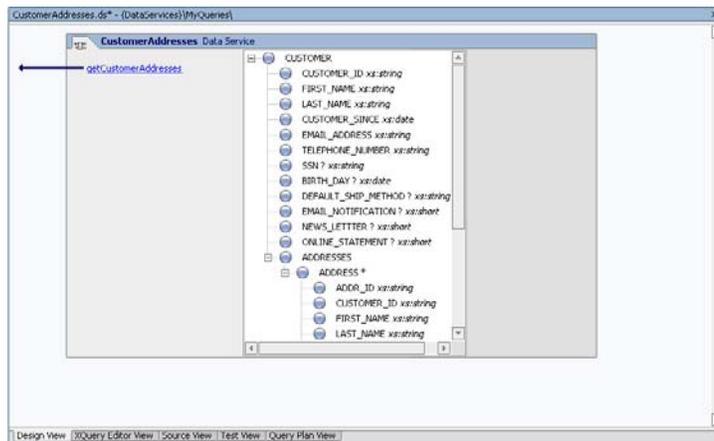
Figure 18-15   Design View of CustomerAddresses Data Service

4. Click getCustomerAddresses to open XQuery Editor View for that function.

5. Add two for nodes to the work area, by completing the following steps:

   a. In the Data Services Palette, expand the DataServices\CustomerDB folders.

   b. Open the CUSTOMER.ds folder (located in the CustomerDB folder), and then drag and drop CUSTOMER()into XQuery Editor View.

   c. Open the ADDRESS.ds folder (located in the CustomerDB folder), and then drag and drop ADDRESS()into XQuery Editor View.
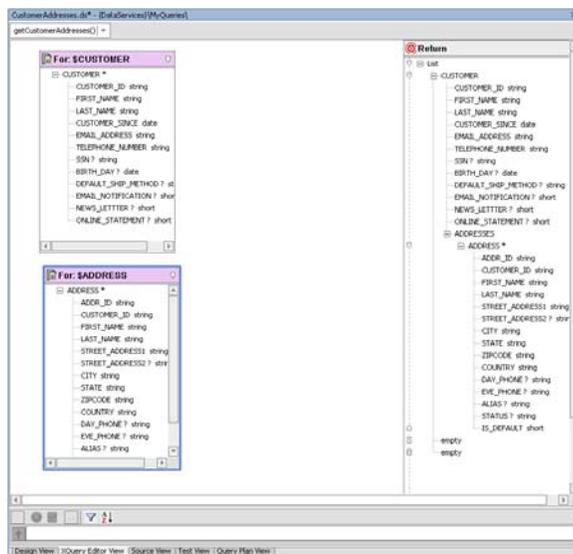
Figure 18-16   Source Nodes

6. Create an induced mapping for the CUSTOMER node. Drag and drop the CUSTOMER* element (source) onto the CUSTOMER element (Return).

7. Create an induced mapping for the ADDRESS node. Drag and drop the ADDRESS* element (source) onto the ADDRESS element (Return).

   Note: Do not drop the source element onto the ADDRESSES element.

8. Create a source node relationship. Drag and drop the CUSTOMER_ID element in the $CUSTOMER node onto the corresponding element in the $ADDRESS node.
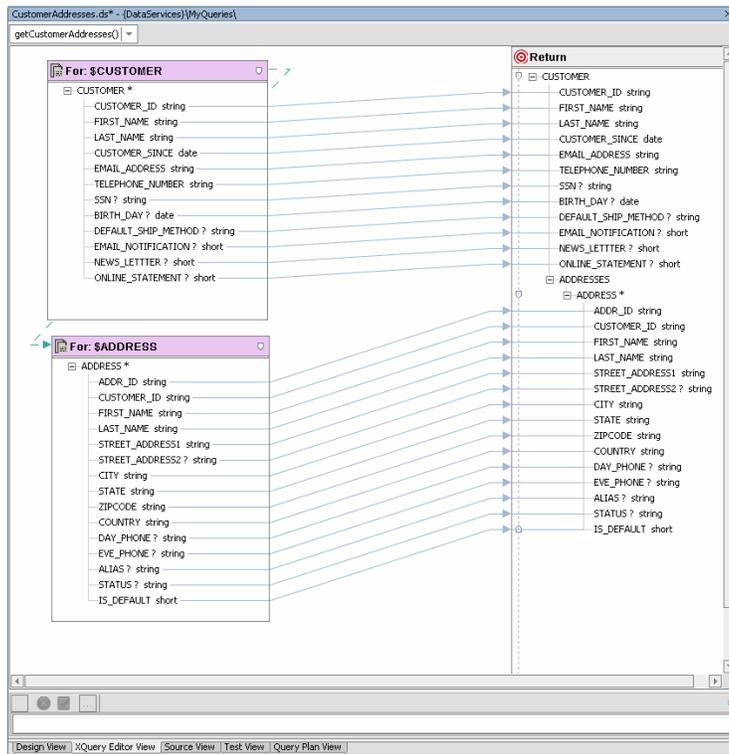
Figure 18-17   Mapped and Joined Source Nodes
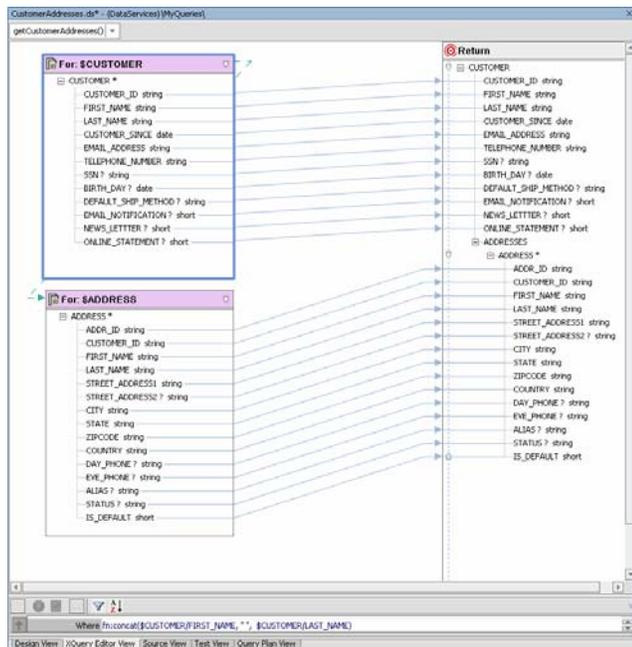
Figure 18-18   **Where** Clause

9. Add an OrderBy clause, by completing the following steps:

   a. Click the ADDRESS node's header. The Expression Editor opens.

   b. Click the Order By Clause icon.

c. Click inside the Order By Clause field.

d. Enter $ADDRESS/ZIPCODE descending in the field.

e. Click the green check button to accept the changes.
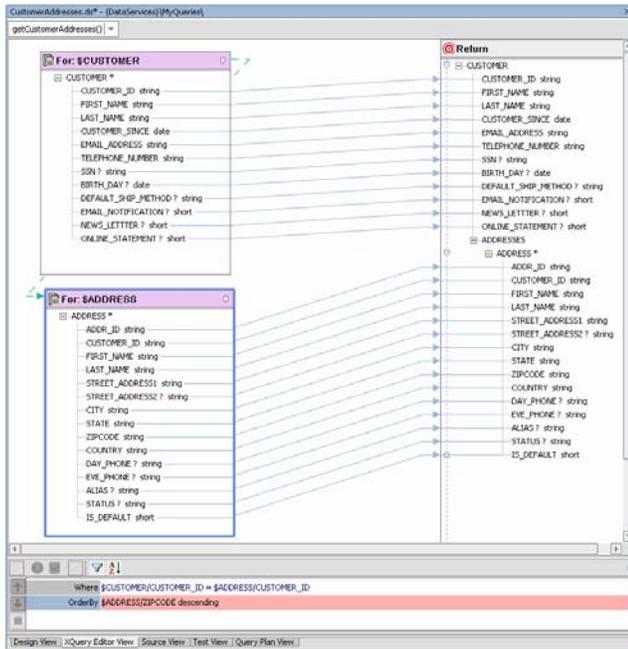
Figure 18-19   OrderBy Clause

10. Test the function, by completing the following steps:

a. Open CustomerAddresses.ds in Test View.

b. Select getCustomerAddresses() from the function drop-down list.

c. Click Execute.

d. Confirm the results. Addresses should be nested after the customer's information.
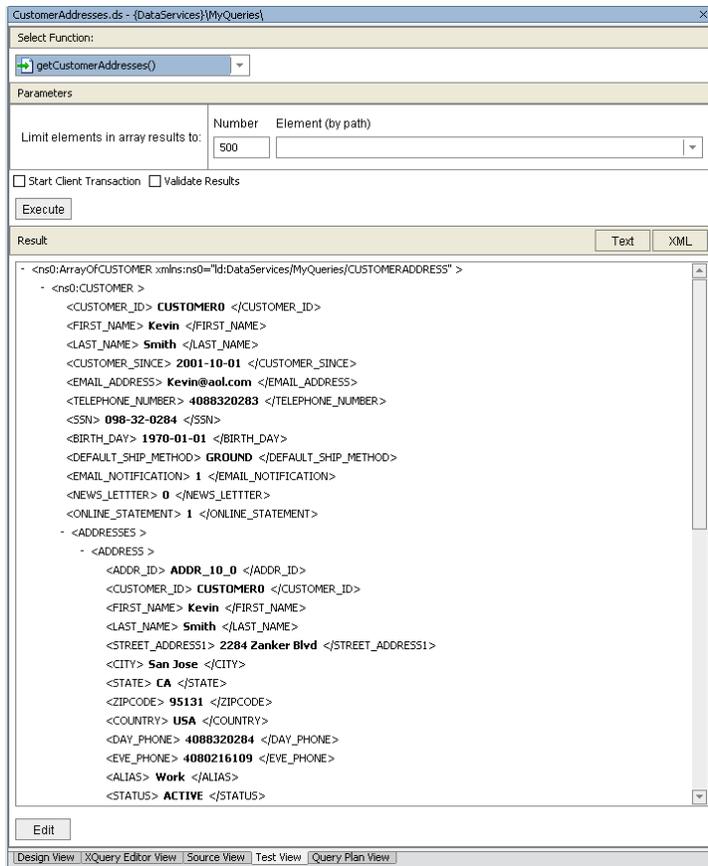
Figure 18-20   Order By Test Results

11. Open CustomerAddresses.ds in Source View to view the generated XQuery. (<u>Note:</u> The automatic namespace assignments may not match.)

```
CustomerAddresses.ds - {DataServices}\MyQueries\                                        ×
declare function tns:getCustomerAddresses() as element(ns5:CUSTOMER)* {
    for $CUSTOMER in ns6:CUSTOMER()
    return
    <ns5:CUSTOMER>
        <CUSTOMER_ID>{fn:data($CUSTOMER/CUSTOMER_ID)}</CUSTOMER_ID>
        <FIRST_NAME>{fn:data($CUSTOMER/FIRST_NAME)}</FIRST_NAME>
        <LAST_NAME>{fn:data($CUSTOMER/LAST_NAME)}</LAST_NAME>
        <CUSTOMER_SINCE>{fn:data($CUSTOMER/CUSTOMER_SINCE)}</CUSTOMER_SINCE>
        <EMAIL_ADDRESS>{fn:data($CUSTOMER/EMAIL_ADDRESS)}</EMAIL_ADDRESS>
        <TELEPHONE_NUMBER>{fn:data($CUSTOMER/TELEPHONE_NUMBER)}</TELEPHONE_NUMBER>
        <SSN?>{fn:data($CUSTOMER/SSN)}</SSN>
        <BIRTH_DAY>{fn:data($CUSTOMER/BIRTH_DAY)}</BIRTH_DAY>
        <DEFAULT_SHIP_METHOD?>{fn:data($CUSTOMER/DEFAULT_SHIP_METHOD)}</DEFAULT_SHIP_METHOD>
        <EMAIL_NOTIFICATION?>{fn:data($CUSTOMER/EMAIL_NOTIFICATION)}</EMAIL_NOTIFICATION>
        <NEWS_LETTTER?>{fn:data($CUSTOMER/NEWS_LETTTER)}</NEWS_LETTTER>
        <ONLINE_STATEMENT?>{fn:data($CUSTOMER/ONLINE_STATEMENT)}</ONLINE_STATEMENT>
        <ADDRESSES>
            {
                for $ADDRESS in ns8:ADDRESS()
                where $CUSTOMER/CUSTOMER_ID = $ADDRESS/CUSTOMER_ID
                order by $ADDRESS/ZIPCODE descending
                return
                <ADDRESS>
                    <ADDR_ID>{fn:data($ADDRESS/ADDR_ID)}</ADDR_ID>
                    <CUSTOMER_ID>{fn:data($ADDRESS/CUSTOMER_ID)}</CUSTOMER_ID>
                    <FIRST_NAME>{fn:data($ADDRESS/FIRST_NAME)}</FIRST_NAME>
                    <LAST_NAME>{fn:data($ADDRESS/LAST_NAME)}</LAST_NAME>
                    <STREET_ADDRESS1>{fn:data($ADDRESS/STREET_ADDRESS1)}</STREET_ADDRESS1>
                    <STREET_ADDRESS2?>{fn:data($ADDRESS/STREET_ADDRESS2)}</STREET_ADDRESS2>
                    <CITY>{fn:data($ADDRESS/CITY)}</CITY>
                    <STATE>{fn:data($ADDRESS/STATE)}</STATE>
                    <ZIPCODE>{fn:data($ADDRESS/ZIPCODE)}</ZIPCODE>
                    <COUNTRY>{fn:data($ADDRESS/COUNTRY)}</COUNTRY>
                    <DAY_PHONE?>{fn:data($ADDRESS/DAY_PHONE)}</DAY_PHONE>
                    <EVE_PHONE?>{fn:data($ADDRESS/EVE_PHONE)}</EVE_PHONE>
                    <ALIAS?>{fn:data($ADDRESS/ALIAS)}</ALIAS>
                    <STATUS?>{fn:data($ADDRESS/STATUS)}</STATUS>
                    <IS_DEFAULT>{fn:data($ADDRESS/IS_DEFAULT)}</IS_DEFAULT>
                </ADDRESS>
            }
        </ADDRESSES>
    </ns5:CUSTOMER>
Design View | XQuery Editor View | Source View | Test View | Query Plan View
```

Figure 18-21 **CustomerAddresses() Source View**

Lab 18.7   Creating Group By and Aggregate Expressions

Sometimes, you may want to group data according to particular data elements, such as grouping customers by state and country.

Objectives

In this lab, you will:

> Create a query using the group by operator and sum() function that generates a report of customers grouped by state and city, showing total sales by city.

> Test the function.

> Review the XQuery source code.

Instructions

1. Create a new data service in the MyQueries folder and name it CustomerOrders.

2. Associate the CustomerOrders data service with the CUSTOMER_ORDER.xsd schema. The schema is located in the MyQueries\schemas folder.

3. Create a new function and name it getCustomerOrderAmount.
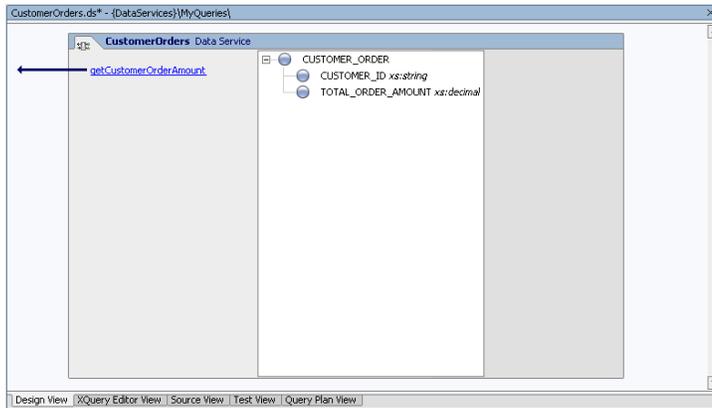
Figure 18-22   Design View of Customer Orders Data Service

4.   Click getCustomerOrderAmount to open XQuery Editor View for that function.

5.   Add a for node, by completing the following steps:

   a.   In the Data Services Palette, open the CUSTOMER_ORDER.ds folder, which is located in the DataServices\ApparelDB folder.

   b.   Drag and drop CUSTOMER_ORDER()into XQuery Editor View.

6.   Create a GroupBy clause, by completing the following steps:

   a.   Right-click the C_ID element in the $CUSTOMER_ORDER source node.

   b.   Choose Create Group By. A GroupBy node is created.

7.   Create a simple mapping. Drag and drop the TOTAL_ORDER_AMT from the Group section of the GroupBy node onto the corresponding element in the Return type.

8.   Create a simple mapping. Drag and drop the C_ID element in the By section of the GroupBy node to the corresponding element in the Return type.
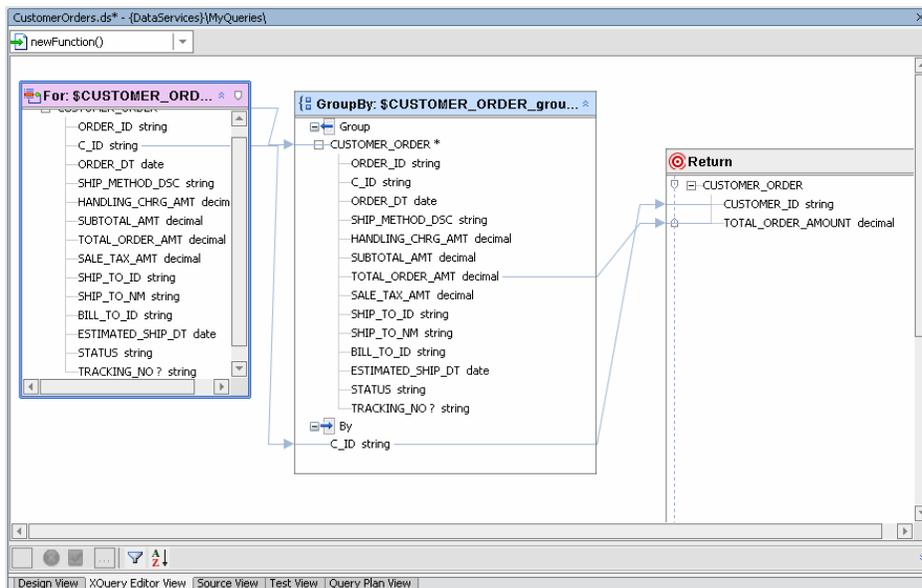
Figure 18-23   GroupBy Node Added and Mapped

Modify a Return expression, by completing the following steps:

a. Click the TOTAL_ORDER_AMOUNT, located in the Return node. The Expression Editor opens. Every element in a Return type has an underlying expression. In this case the expression is:

```
{fn:data($CUSTOMER_ORDER_group/TOTAL_ORDER_AMT)}}
```

b. Edit the expression so that it changes fn:data() to fn:sum(), as follows:

```
{fn:sum($CUSTOMER_ORDER_group/TOTAL_ORDER_AMT)}
```

c. Click the green check button to accept the changes.
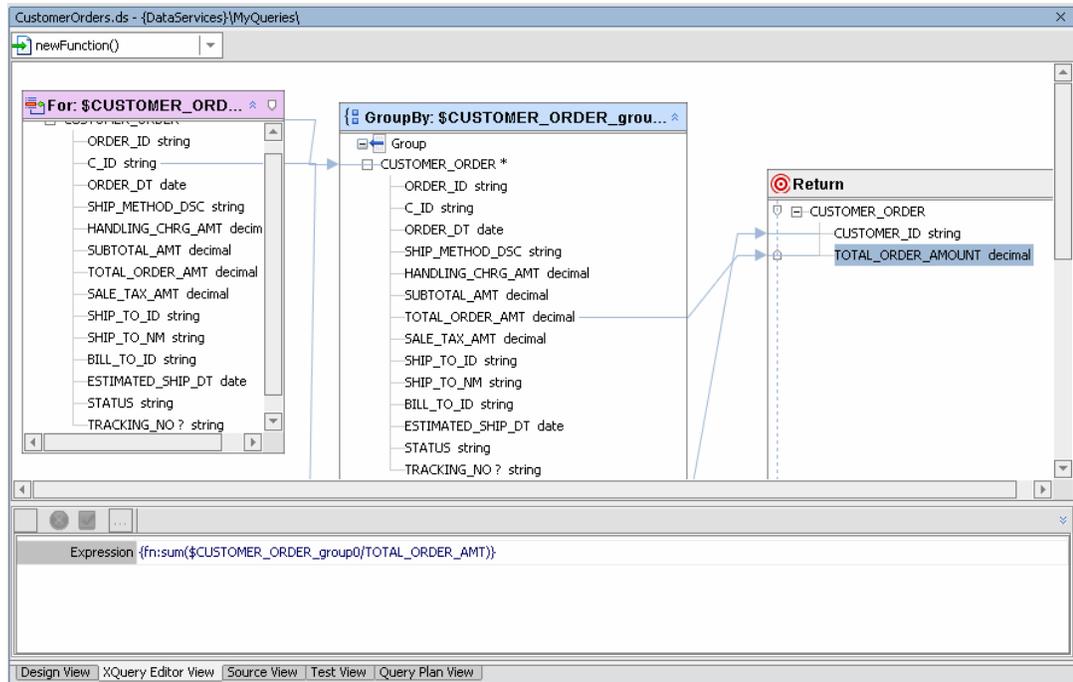
Figure 18-24   Aggregate Expression

9. Test the function, by completing the following steps:

a. Open CustomerOrders.ds in Test View.

b. Select getCustomerOrderAmount() from the Function drop-down list.

c. Click Execute.

d. Confirm the results.
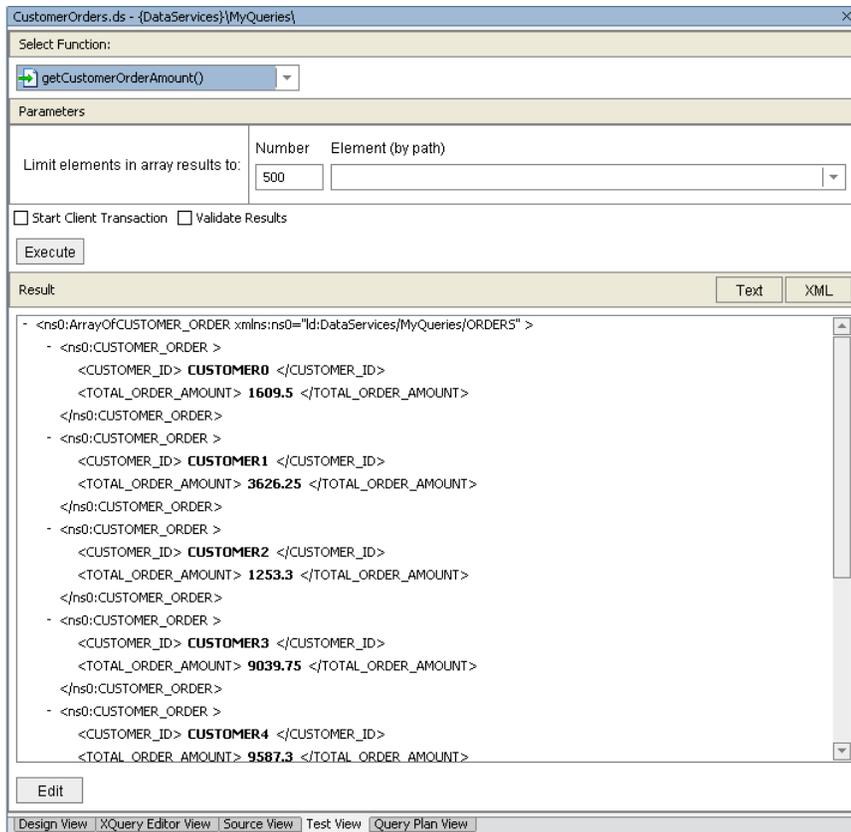
Figure 18-25   Aggregate Test Results

10. Open CustomerOrders.ds in Source View to view the generated XQuery. (<u>Note:</u> The automatic namespace assignments may not match that shown in the lab.)
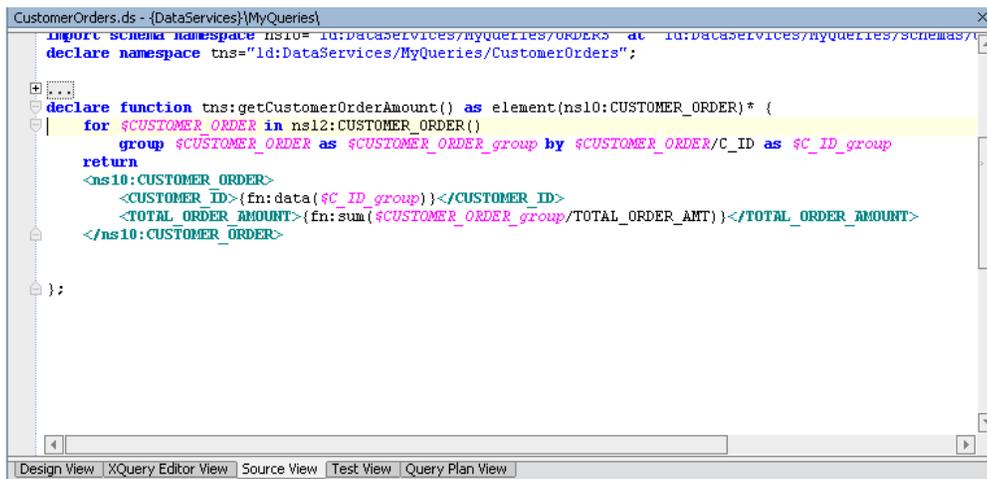
Figure 18-26   Source View of the CustomerOrders Data Service

Lab 18.8 Creating Constant Expressions

Creating a data service query that uses a constant expression enables a quick and easy way to locate specific information. For example, you can use a constant expression to identify all customers who ship by Ground method.

Objectives

In this lab, you will:

> Create a non-parameterized function that will return all customers whose default shipping method is GROUND.

> Test the function.

> View the XQuery source code.

Instructions

1. Add a new function to the CustomerInfo data service and name it getGroundCustomers.

2. Click the getGroundCustomers() function to open the XQuery Editor View.

3. Add a for node, by completing the following steps:

   a. In the Data Services Palette, open the CUSTOMER.ds folder, which is located in the DataServices\CustomerDB folder.

   b. Drag and drop CUSTOMER()into XQuery Editor View.

4. Create an induced mapping. Drag and drop the entire CUSTOMER* element (source node) onto the CUSTOMER element (Return).

5. Add a where clause, by completing the following steps:

   a. Click the CUSTOMER node's header. The Expression Editor opens.

   b. Click the Add Where Clause icon.

   c. Enter the following expression as a where clause:

      `$CUSTOMER/DEFAULT_SHIP_METHOD eq "GROUND"`

   d. Click the green check mark icon to accept the where clause for the customer object.
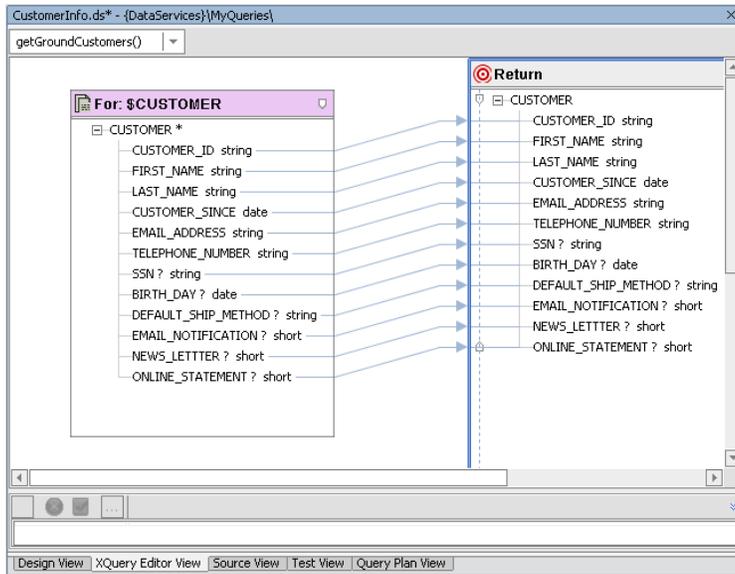
Figure 18-27   Constant Function with Default Expression

6.   Test the function. The results should be as displayed in Figure 18-28.
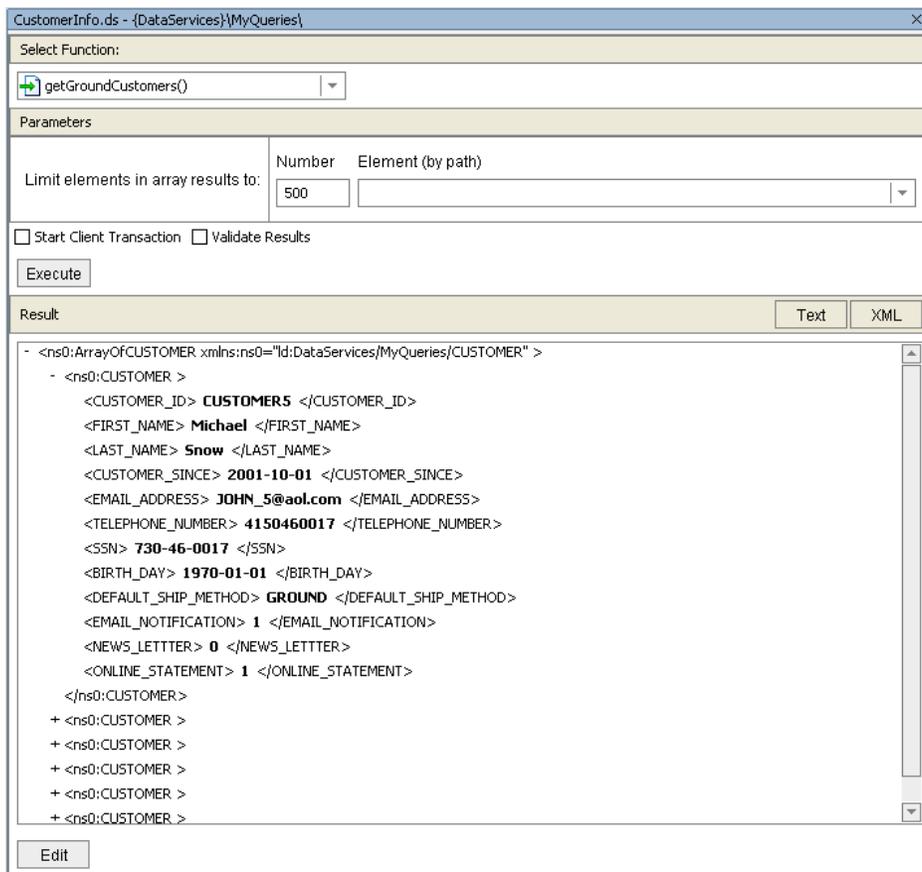
Figure 18-28   Test Results of a Constant Expression

7.   Open CustomerInfo.ds in Source View. The code should be as displayed in Figure 18-29.
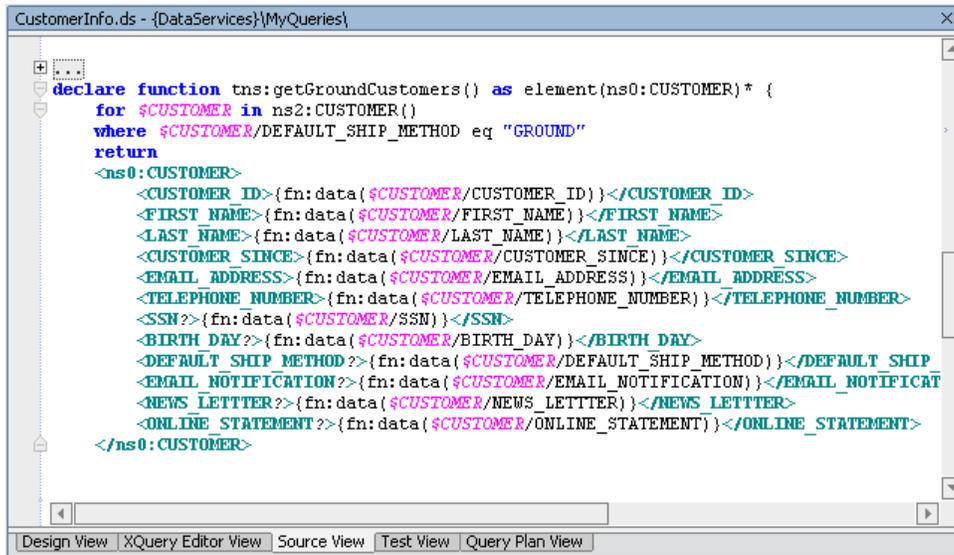
```
CustomerInfo.ds - {DataServices}\MyQueries\                                    ×

⊞ ...
declare function tns:getGroundCustomers() as element(ns0:CUSTOMER)* {
    for $CUSTOMER in ns2:CUSTOMER()
    where $CUSTOMER/DEFAULT_SHIP_METHOD eq "GROUND"
    return
    <ns0:CUSTOMER>
        <CUSTOMER_ID>{fn:data($CUSTOMER/CUSTOMER_ID)}</CUSTOMER_ID>
        <FIRST_NAME>{fn:data($CUSTOMER/FIRST_NAME)}</FIRST_NAME>
        <LAST_NAME>{fn:data($CUSTOMER/LAST_NAME)}</LAST_NAME>
        <CUSTOMER_SINCE>{fn:data($CUSTOMER/CUSTOMER_SINCE)}</CUSTOMER_SINCE>
        <EMAIL_ADDRESS>{fn:data($CUSTOMER/EMAIL_ADDRESS)}</EMAIL_ADDRESS>
        <TELEPHONE_NUMBER>{fn:data($CUSTOMER/TELEPHONE_NUMBER)}</TELEPHONE_NUMBER>
        <SSN?>{fn:data($CUSTOMER/SSN)}</SSN>
        <BIRTH_DAY?>{fn:data($CUSTOMER/BIRTH_DAY)}</BIRTH_DAY>
        <DEFAULT_SHIP_METHOD?>{fn:data($CUSTOMER/DEFAULT_SHIP_METHOD)}</DEFAULT_SHIP_
        <EMAIL_NOTIFICATION?>{fn:data($CUSTOMER/EMAIL_NOTIFICATION)}</EMAIL_NOTIFICAT
        <NEWS_LETTTER?>{fn:data($CUSTOMER/NEWS_LETTTER)}</NEWS_LETTTER>
        <ONLINE_STATEMENT?>{fn:data($CUSTOMER/ONLINE_STATEMENT)}</ONLINE_STATEMENT>
    </ns0:CUSTOMER>

 Design View | XQuery Editor View | Source View | Test View | Query Plan View
```

Figure 18-29   Source Code

Lesson Summary

In this lesson you learned how to:

> Use the graphical XQuery Editor View to create parameterized, string, and date functions; outer joins, aggregate, and order by and constant expressions.

> Use the XQuery Function Palette to add built-in XQuery functions to a query.

Lesson 19 Building XQueries in Source View

In the previous lesson, you built XQueries using XQuery Editor View. Sometimes, it is necessary to programmatically build a query or modify its code. In this lesson, you will learn how to use Source View to create and edit query functions.

Objectives

After completing this lesson, you will be able to:

> Use Source View to add, edit, or delete XQuery code that defines a data service's query functions.

> Compare the coded query with the XQuery Editor View.

> Test the results.

Overview

Source View lets you view and/or modify the data service's XQuery source code. In general, a data service is simply a file that contains XQuery code. Although DSP provides extensive visual design tools for developing a data service, sometimes you may need to work directly with XQuery syntax.

Two-way editing is supported — changes you make in Source View are reflected in XQuery Editor View, and vice versa. The source code is commented to help you edit the source correctly.

Figure 19-1   Source View

Source View Tools

Within Source View, you can use the XQuery Construct Palette, which lets you add any of several built-in generic FLWOR statements to the XQuery syntax. You can then customize the generic statement to match your particular needs.

To add a FLWOR construct, drag and drop the selected item into the appropriate *declare function* space.

If XQuery Construct Palette is not open, choose View → Windows → XQuery Construct Palette.
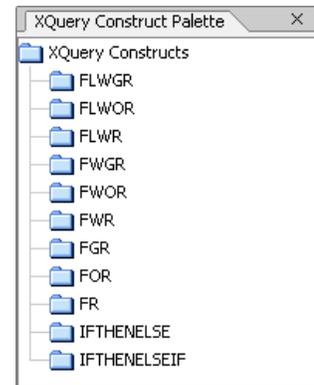
Lab 19.1 Creating a New XML Type

For each of the queries created in this lesson, you will define a function that returns results nested within the Return type. To enable that, you need to create a data service with an undefined XML type. By leaving the XML type's schema undefined, you can modify the Return type on an ad hoc basic, without a need to be concerned about synchronizing the XML and Return types.

Objectives

In this lab, you will:

>   Create a new data service, called XQueries.ds.

>   Create a new, but undefined, XML type.

Instructions

1. Create a new data service in the MyQueries folder and name it XQueries.

2. Create a new XML type by completing the following steps:

    a. Right-click the XQueries Data Service header.

    b. Select Create XML Type.

    c. Enter Results in the Return Type field.

        Note: Do not change the default settings for the Schema File and Target Namespace fields.

    d. Click OK.
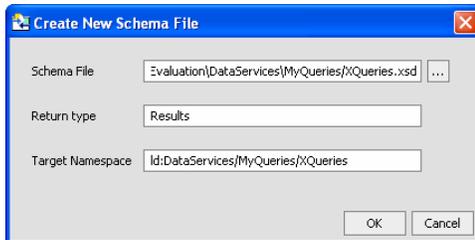
Figure 19-2   Create New XML Type

3. Confirm that the data service diagram is as displayed in Figure 19-3.
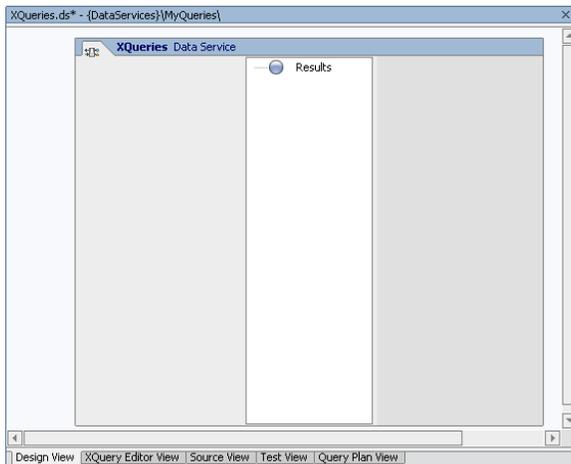
Figure 19-3   Design View: Undefined Results Type

Lab 19.2 Creating a Basic Parameterized XQuery

There are two basic types of queries: those without parameters and those with parameters. In the previous lesson, you used XQuery Editor View's graphical tools to define a query with parameters. In this lab, you will use Source Editor to programmatically define a parameterized query.

Objectives

In this lab, you will:

Build a query that retrieves customer information based on first and last names.

View the results in XQuery Editor View.

Test the function.

Instructions

Note: Namespaces may differ for your application.

1. Add a new function to XQueries.ds and name it getCustomerByName.

2. Open Source View.

3. Define the function declaration, by completing the following steps:

   a. Add the following parameter to the first parenthesis:

      ```
      $p_firstname as xs:string, $p_lastname as xs:string
      ```

   b. Remove the asterisk (*), since you want this function to only return a single result.

      The code should be similar to the following :

      ```
      declare function tns:getCustomerByName($p_firstname as xs:string,
      $p_lastname as xs:string) as element(ns0:Results) {
      ```

4. Click the + symbol next to the getCustomerByName() function. This opens the function body.

5. Split the <ns0:RESULTS/> element into open and end tags, with curly braces in between for the XQuery. The code should be as follows (ignore the error indicator):

   ```
   <ns0:Results>
   {
   }
   </ns0:Results>
   ```

6. Open XQuery Construct Palette.

7. Drag and drop the FWR construct between the curly braces. The code should be as follows:

   ```
   for $var in ()
   where true()
   return
     ()
   ```

8. Define the for clause by completing the following steps:

    a. Change the variable to $customer.

    b. In the Data Services Palette, expand the CustomerDB\CUSTOMER.ds folders.

    c. Drag and drop CUSTOMER() into the for clause's first empty parenthesis. The code should be similar to the following:

```
for $customer in (ns1:CUSTOMER())

where true()

return

()
```

9. Replace the where clause true() code with the following:

```
$customer/FIRST_NAME eq $p_firstname and $customer/LAST_NAME eq
$p_lastname
```

10. Set the return clause, by adding $customer between the parenthesis.

11. Confirm that the source code is as displayed in Figure 19-4; namespaces may be different for your application.

Figure 19-4   Parameterized Query Source Code

12. Build the DataServices project.

13. Open XQueries.ds in XQuery Editor View and review the graphical version of the XQuery code. It should be as displayed in Figure 19-5.
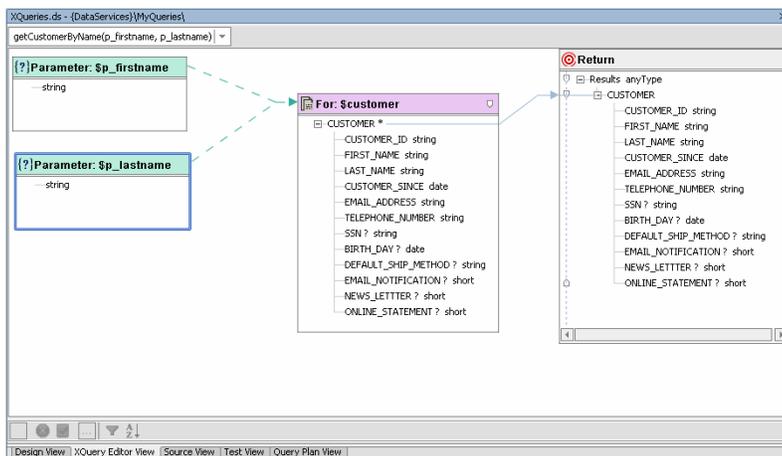
Figure 19-5   XQuery Editor View of Parameterized Function

14. Test the function, by completing the following steps:

    a.   Open XQueries.ds in Test View.

    b.   Select getCustomersByName from the Function drop-down list.

    c.   Enter the following parameters:

        Firstname: Jack

        Lastname: Black

    d.   Confirm the results.
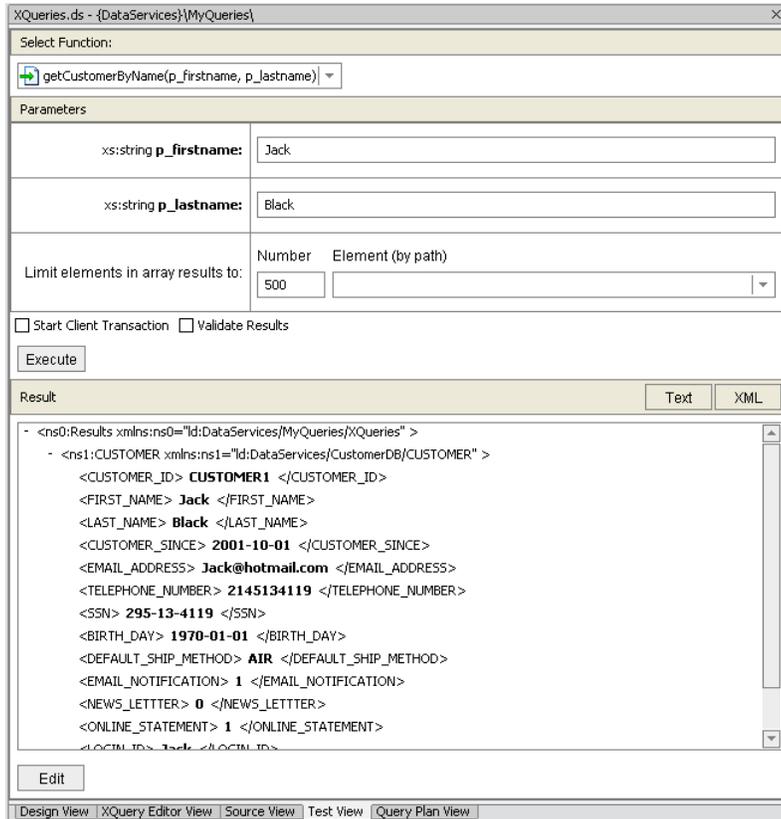
Figure 19-6   Test Results of a Parameterized Function

15. (*Optional*) Open CustomerInfo.ds in XQuery Editor View and compare the diagrams for the two data services.

XQuery Code Reference for a Parameterized Function

```
declare function tns:getCustomerByName($p_firstname as xs:string, $p_lastname as
xs:string) as element(ns0:Results) {
    <ns0:Results>
    {
        for $customer in (ns1:CUSTOMER())
        where ($customer/FIRST_NAME eq $p_firstname and $customer/LAST_NAME eq
$p_lastname)
        return
            ($customer)
    }
    </ns0:Results>
```

Lab 19.3 Creating a String Function

XQuery provides numerous string functions that can be incorporated into your business logic.

Objectives

In this lab, you will:

Create a startwith() function that retrieves customer information by name or SSN.

Test the function.

Instructions

1. Open XQueries.ds in Design View.

2. Click the getCustomerByName function's arrow, select Rename, and change the function name to getCustomerByNameOrSSN().

3. Open XQueries.ds in Source View.

4. Define the function declaration, by changing the parameter as follows:

   ```
   $fullname as xs:string, $ssn as xs:string
   ```

5. Replace the contents of the where clause with the following:

   ```
   fn:contains(fn:upper-case(fn:concat($customer/FIRST_NAME,"
   ",$customer/LAST_NAME)), fn:upper-case($fullname) ) or
   fn:starts-with($customer/SSN, $ssn)
   ```

Note: You can either type the code in or build the clause by using the following built-in functions, located in the XQuery Function Palette:

fn:concat	fn:starts-with
fn:contains	fn:upper-case

Note: The full name is created "on-the-spot" by concatenating FIRST_NAME and LAST_NAME elements to the local (XQuery engine internal) variable such as $p_name. Upper case is used to normalize names.

6. Leave the return clause as $customer so that all elements in the type are returned.

7. Confirm that the code is as follows (namespaces may be different for your application):

Figure 19-7   Source View of String Function

8.   Open XQueries.ds in XQuery Editor View.
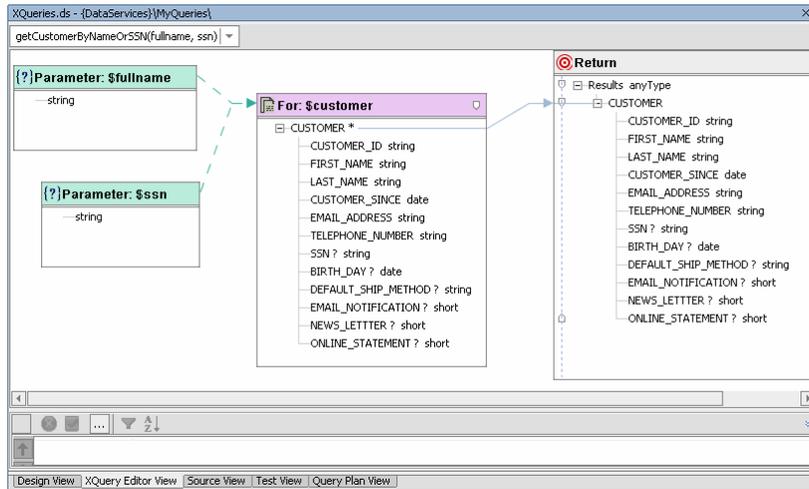
Figure 19-8   XQuery Editor View of String Function

9.   Test the query by completing the following steps:

    a.   Open XQueries.ds in Test View.

    b.   Enter a value in both Parameter fields. Neither field can be blank; however, because of the query logic, only one parameter needs to be matched.

    c.   Click Execute. The query should return results based on your keyword search parameters. See below for results in Test View and the underlying code.
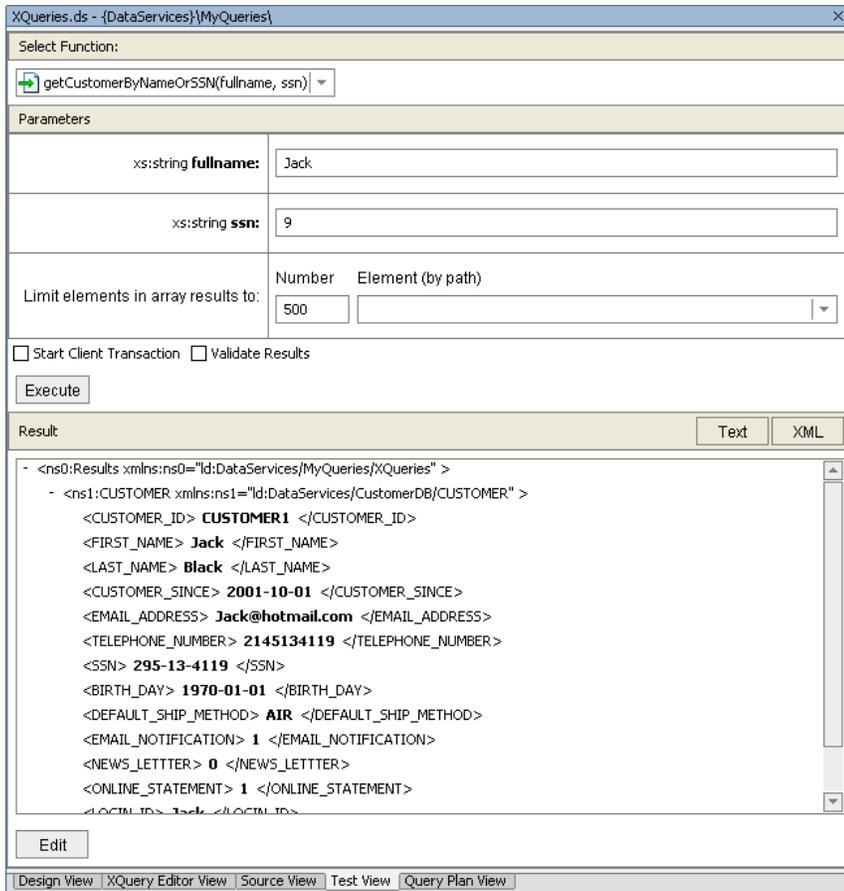
Figure 19-9   Test Results of String Function

XQuery Code Reference for a String Function

```
declare function tns:getCustomerByNameOrSSN($fullname as xs:string, $ssn as xs:string)
as element(ns0:Results) {
    <ns0:Results>
    {
        for $customer in (ns1:CUSTOMER())
        where ( fn:contains(fn:upper-case(fn:concat($customer/FIRST_NAME,"
            ",$customer/LAST_NAME)), fn:upper-case($fullname) ) or
            fn:starts-with($customer/SSN, $ssn) )
        return
            ($customer)
    }

    </ns0:Results>
```

Lab 19.4   Building an Outer Join and Using Order By

Outer joins allow you to get results from the joined objects even if the primary key is not represented in both objects. For example, an outer join of customers and orders reports all customers — even those without orders.

Objectives

In this lab, you will:

> Build a query that retrieves all customers and lists their addresses, if any.
>
> Shape the return data to include:
>
> o   All customers, even those without known addresses.
>
> o   Nest addresses with customers (there may be more than 1).
>
> o   Order customers by first name and last name.
>
> o   Order the addresses by zip code.
>
> Test the function.

Instructions

Note: Namespaces may differ for your application.

1. Add a new function to XQueries.ds and name it getCustomerAddresses.

2. Open XQueries.ds in Source View.

3. Define the function declaration by removing the asterisk (*). The code should be as:

   ```
   declare function tns:getCustomerAddresses() as element(ns0:Results) {
   ```

4. Click the + symbol next to the getCustomerAddresses() function. This opens the function body.

5. Split the <ns0:RESULTS/> element into open and end tags, with curly braces in between for the XQuery.

6. Open XQuery Construct Palette, and then drag and drop the FOR construct between the curly braces. The code should be as follows:

   ```
   for $var in ()
   order by ()
   return
   ()
   ```

7. Set the for clause, using a $customer variable that is associated with CUSTOMER() located in the CustomerDB\CUSTOMER.ds folder within the Data Services Palette.

   ```
   for $customer in (ns1:CUSTOMER())
   ```

8. Set the order by clause, by replacing the (), as follows:

   ```
   $customer/FIRST_NAME, $customer/LAST_NAME
   ```

9. Set the return clause, by replacing the (), as follows:

```
return
    <CUSTOMER>
        <FIRST_NAME>{fn:data($customer/FIRST_NAME) }</FIRST_NAME>
        <LAST_NAME>{fn:data($customer/LAST_NAME)}</LAST_NAME>
        {
          for $address in ()
          where ($address/CUSTOMER_ID eq $customer/CUSTOMER_ID)
          order by $address/ZIPCODE ascending
          return
          $address
        }
    </CUSTOMER>
```

Note: You can either type the code in, or use the XQuery Function Palette and XQuery Construct Palette to build up your query function.

10. Set the $address clause by associating it with ADDRESS(), which is located in CustomerDB\ADDRESS.ds folder within Data Services Palette.

```
for $address in (ns2:ADDRESS())
```

11. Confirm that the query is as shown in Figure 19-10; namespaces may be different for your application.

Figure 19-10   Source View of Outer View and Order By Function

12. Open XQueries.ds in XQuery Editor View.
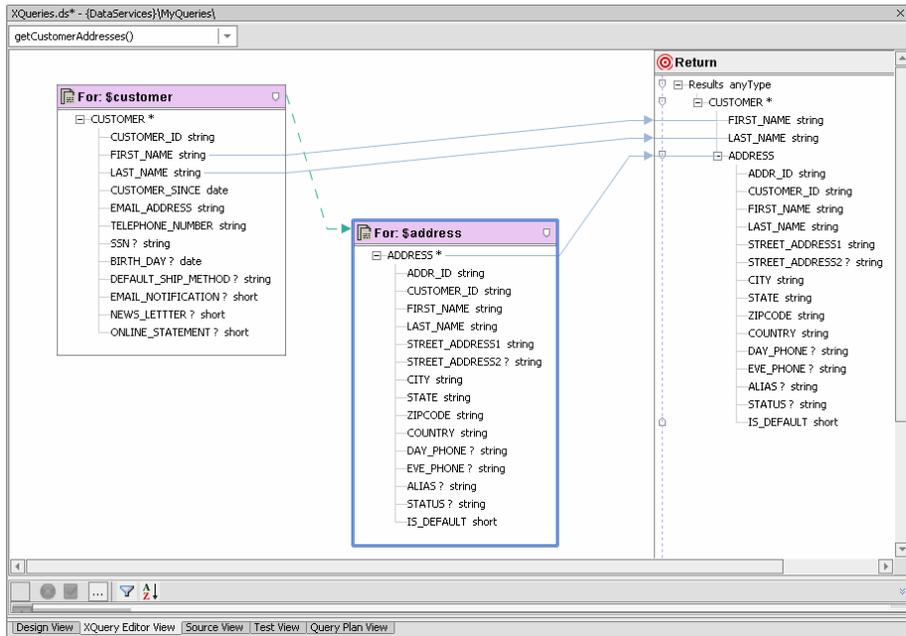
Figure 19-11   XQuery Editor View of Outer Join and Order By Function

13. Open XQueries.ds in Test View and test the query; no parameters are required. The XQuery function appears below.
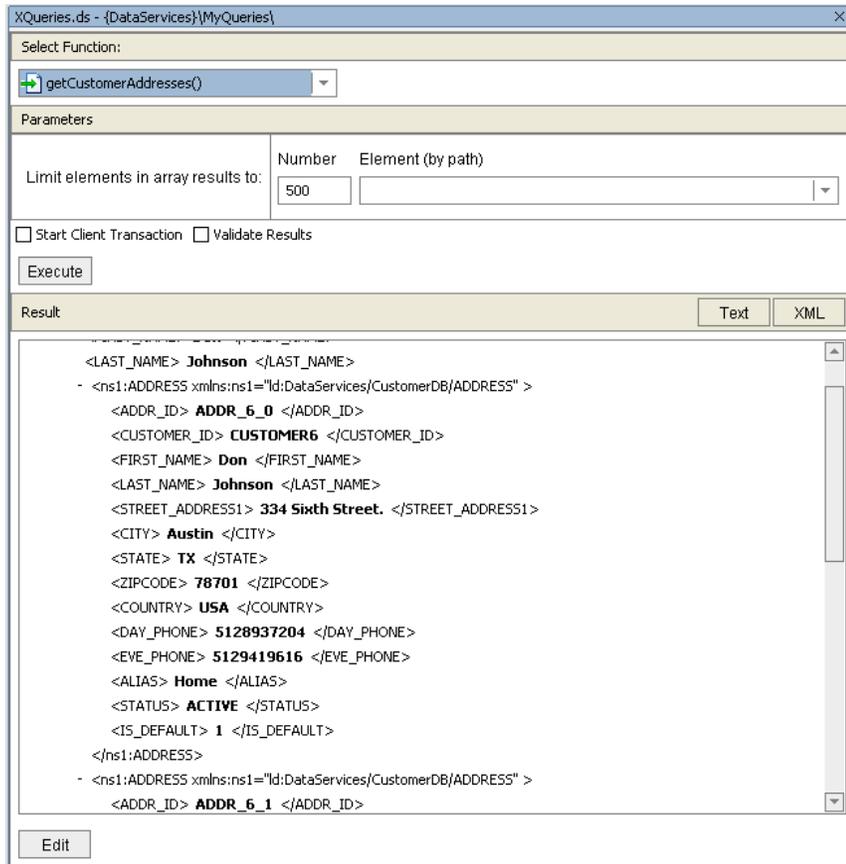
Figure 19-12   Test Results of Outer Join and Order By Function

XQuery Code Reference for an Outer Join and Order By Function

```
declare function tns:getCustomerAddresses() as element(ns0:Results) {
    <ns0:Results>
    {
    for $customer in (ns1:CUSTOMER())
    order by $customer/FIRST_NAME, $customer/LAST_NAME
    return
        <CUSTOMER>
              <FIRST_NAME>{ fn:data($customer/FIRST_NAME) }</FIRST_NAME>
              <LAST_NAME>{fn:data($customer/LAST_NAME)}</LAST_NAME>
              {
                  for $address in (ns2:ADDRESS())
                  where ($address/CUSTOMER_ID eq $customer/CUSTOMER_ID)
                  order by $address/ZIPCODE ascending
                  return
                  $address
              }
        </CUSTOMER>

    }
    </ns0:Results>
```

Lab 19.5 Creating an Inner Join and a Top *N*

Inner joins mandate that the only items that are returned are with a corresponding entry (such as a primary key in the relational world) in another data source. The following are introduced:

> let clauses.
>
> Nested for clauses.
>
> Concat() and subsequent() XQuery functions.

Objectives

In this lab, you will:

> Build a query that retrieves the top 10 customers who have placed orders with the company.
>
> Define the shape of the returned data to include:
>
> o Total order amount.
>
> o Items returned ordered by total order amount, from the highest to the lowest.
>
> o Customer's full name, order ID, and order amount.
>
> Test the function.

Instructions

Note: Namespaces may differ for your application.

1. Add a new function to XQueries.ds and name it getTop10Customers.

2. Open XQueries.ds in Source View.

3. Define the function declaration by removing the asterisk (*). The code should be as follows:

    ```
    declare function tns:getTop10Customers() as element(ns0:Results) {
    ```

4. Click the + symbol next to the getTop10Customers function. This opens the function body.

5. Add curly braces between the two tags.

---

6. After the opening curly brace, add the following let clause, which will hold the results of subsequent for clauses:

```
let $top10:=
```

7. Open XQuery Construct Palette, and then drag and drop the FWOR construct after the let clause. The code should be as follows:

```
for $var in ()

where true()

order by ()

return

 ()
```

8. Set the for clause using a $customer variable that is associated with CUSTOMER() located in the CustomerDB\CUSTOMER.ds folder within Data Services Palette.

```
for $customer in (ns1:CUSTOMER())
```

9. Create a second for clause, using a $order variable that is associated with CUSTOMER_ORDER() located in the ElectronicsDB\CUSTOMER_ORDER.ds folder within Data Services Palette.

```
for $order in (ns3:CUSTOMER_ORDER())
```

10. Set the where clause, by replacing the true() with the following code:

```
where ($customer/CUSTOMER_ID eq $order/CUSTOMER_ID)
```

11. Set the order by clause, by entering the following code in the ():

```
order by $order/TOTAL_ORDER_AMOUNT descending
```

12. Set the return clause, by entering the following code:

```
return

  <CUSTOMER>

    <CUSTOMER_NAME>

        {fn:concat($customer/FIRST_NAME," ", $customer/LAST_NAME)}

    </CUSTOMER_NAME>

    <ORDER_ID>{fn:data($order/ORDER_ID)}</ORDER_ID>

    <TOTAL_ORDERS>{fn:data($order/TOTAL_ORDER_AMOUNT)}</TOTAL_ORDERS>

  </CUSTOMER>

return fn:subsequence($top10, 1, 10)
```

Note: You can either type the code in, or use the XQuery Function Palette and XQuery Construct Palette to build up your query.

13. Confirm that the source code is similar to that displayed in Figure 19-13; namespaces may vary.
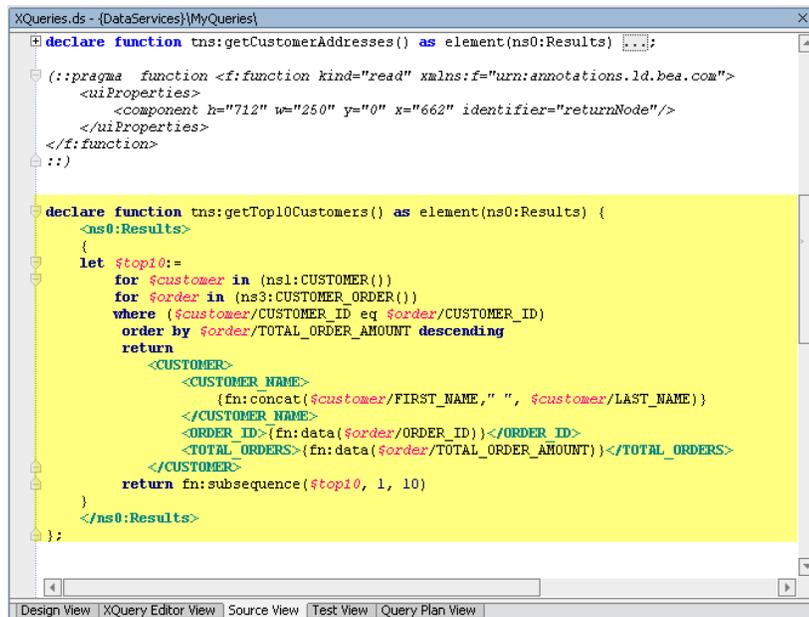
Figure 19-13   Source Code for Inner Join and Top N Function

14. Open XQueries.ds in XQuery Editor View.
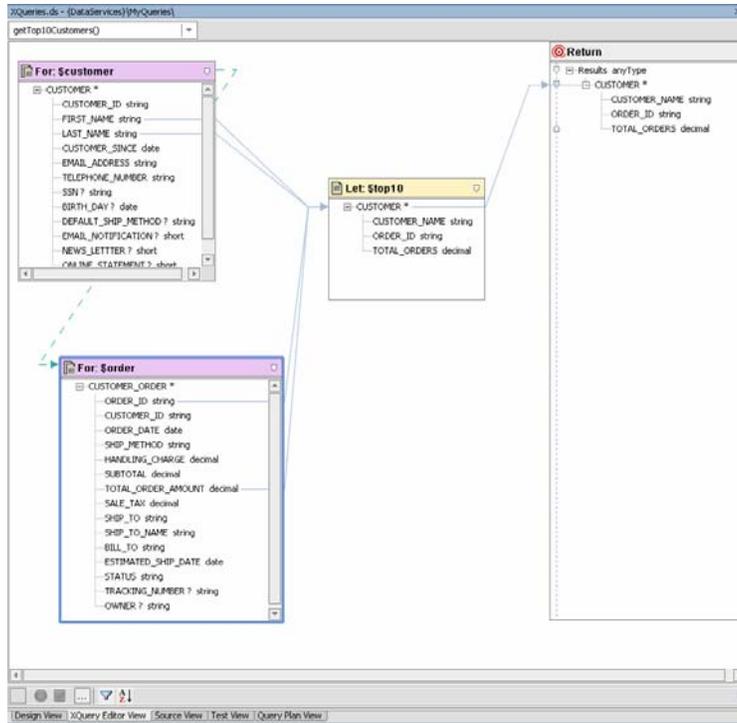
Figure 19-14   XQuery Editor View of Inner Join and Top N Function

15. Open XQueries.ds in Test View; no parameters are required to run your query. You should see a document containing the top 10 orders will appear, ordered by total amount. The XQuery function appears below.

Figure 19-15   Test View for Inner Join and Top N Function

XQuery Code Reference for Inner Join and Top N Function

```
declare function tns:getTop10Customers() as element(ns0:Results) {
    <ns0:Results>
    {
    let $top10:=
        for $customer in (ns1:CUSTOMER())
        for $order in (ns3:CUSTOMER_ORDER())
       where ($customer/CUSTOMER_ID eq $order/CUSTOMER_ID)
        order by $order/TOTAL_ORDER_AMOUNT descending
        return
          <CUSTOMER>
              <CUSTOMER_NAME>
                  {fn:concat($customer/FIRST_NAME," ", $customer/LAST_NAME)}
              </CUSTOMER_NAME>
              <ORDER_ID>{fn:data($order/ORDER_ID)}</ORDER_ID>
                  <TOTAL_ORDERS>{fn:data($order/TOTAL_ORDER_AMOUNT)}</TOTAL_ORDERS>
          </CUSTOMER>
        return fn:subsequence($top10, 1, 10)
    }
    </ns0:Results>
```

Lab 19.6   Creating a Multi-Level Group By

Retrieving customers grouped by states and cities is not only often needed; it is also a classic database exercise. The following are introduced:

> Group by clause.

> count() function.

Objectives

In this lab, you will:

> Create a query that determines the number of customers, by state and by city.

> Test the function.

Instructions

1. Add a function to XQueries.ds and name it getNumCustomersByState().

2. Open XQueries.ds in Source View.

3. Define the function declaration, by removing the asterisk *.

4. Click the + symbol next to the getNumCustomersByState() function.

5. Split the <ns0:Results/> element into open and end tags, with curly braces in between.

6. Open XQuery Construct Palette and then drag and drop the for-group-return (FGR) construct between the curly braces:

```
for $var in ()

group $var as $varGroup by () as $var2

return

   ()
```

7. Set the for and group clauses as follows:

```
for $address in ns2:ADDRESS()
group $address as $stateGroup by $address/STATE as $state
```

Note: Your source is invalid until you complete the next step.

8. Associate the for clause with ADDRESS() located in the CustomerDB\Address.ds folder within the Data Services Palette as follows:

```
for $address in ns2:ADDRESS()
```

9. Set the return clause, as follows:

```
   return

     <state>

        <name>{$state}</name>

        <number>{fn:count($stateGroup/CUSTOMER_ID)}</number>

   {
```

Note: The clause includes the fn:count() built-in function, available from the XQuery Function Palette.

10. Open XQuery Construct Palette and then drag and drop the FWGR construct after the return clause's open curlie brace:

```
for $address1 in ns2:ADDRESS()

where $address1/STATE eq $state

group $address1 as $cityGroup by $address1/CITY as $city

return

    <cities>

      <city>{$city}</city>

      <number>{fn:count($cityGroup/CUSTOMER_ID)}</number>

    </cities>

}

  </state>
```

11. Make sure that the namespace in the second for clause is the same as the namespace in the first for clause.

12. Confirm that the code is as displayed in Figure 19-16 (namespaces may be different for your application).
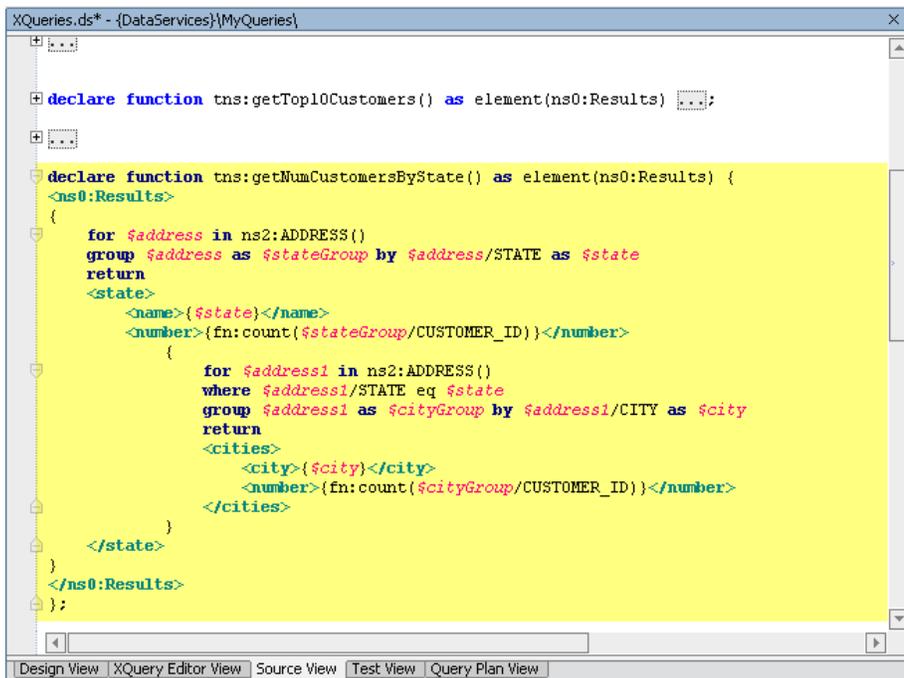
Figure 19-16   Source Code for Multi-Level Group By Function

13. Open XQueries.ds in XQuery Editor View.
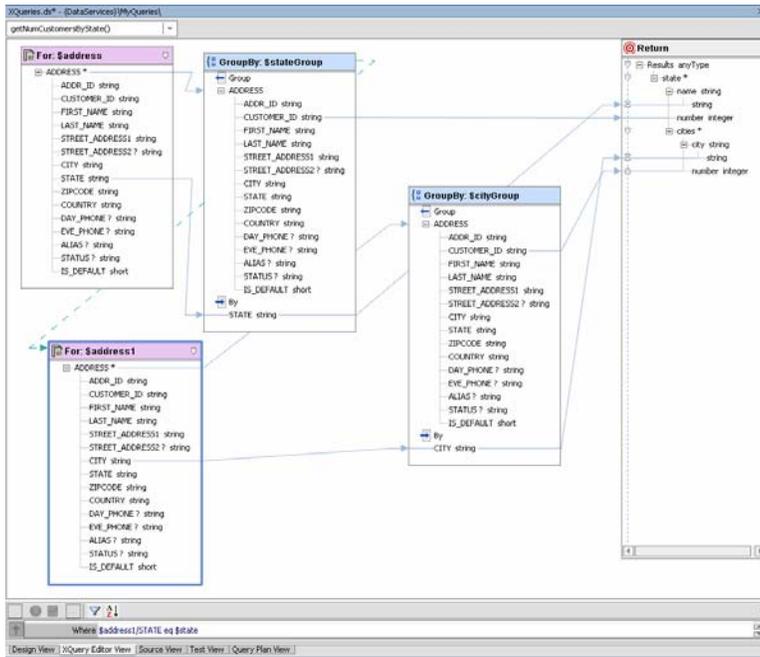
Figure 19-17   XQuery Editor View of Multi-Level Group By Function

14. Open XQueries.ds in Test View and test the function; no parameters are required. You should see the state name, followed by the number of customers residing in that state, followed by the city name and number of customers residing in that city. The underlying XQuery also appears below.

Figure 19-18   Test View of Multi-Level Group By Function

XQuery Code Reference for Multi-Level Group By Function

```
declare function tns:getNumCustomersByState() as element(ns0:Results) {
  <ns0:Results>
  {
    for $address in ns2:ADDRESS()
    group $address as $stateGroup by $address/STATE as $state
    return
    <state>
        <name>{$state}</name>
        <number>{fn:count($stateGroup/CUSTOMER_ID)}</number>
            {
                for $address1 in ns2:ADDRESS()
                where $address1/STATE eq $state
                group $address1 as $cityGroup by $address1/CITY as $city
                return
                <cities>
                    <city>{$city}</city>
                    <number>{fn:count($cityGroup/CUSTOMER_ID)}</number>
                </cities>
            }
    </state>
}
  </ns0:Results>
  };
```

Lab 19.7   Using If-Then-Else

This example shows how you can create switch-like conditions when building your query. The If-Then-Else concept is introduced.

Objectives

In this lab, you will:

> Create a function that returns different achievement levels as strings for a set of customers, based on their total order amount.

> Test the function.

Instructions

<u>Note:</u> Namespaces may differ for your application.

1.   Add a new function to XQueries.ds and name it getCustomerLevels.

2.   Open XQueries.ds in Source View.

3.   Define the function declaration, by removing the asterisk (*).

4.   Split the <ns0:Results/> element into open and end tags, with curly braces ({}) in between.

5.   Add a for clause, using a $customer variable that is associated with CUSTOMER() located in the CustomerDB\CUSTOMER.ds folder within Data Services Palette.

```
for $customer in ns1:CUSTOMER()
```

6.   Add a second for clause, using an $orders variable that is associated with CUSTOMER_ORDER() located in the ElectronicsDB\CUSTOMER_ORDER.ds folder within Data Services Palette.

```
for $orders in ns3:CUSTOMER_ORDER()
```

7.   Add where, let, and return clause code, placing it immediately after the second for clause:

```
where $customer/CUSTOMER_ID eq $orders/CUSTOMER_ID

  group $orders as $orderGroup by fn:concat($customer/FIRST_NAME,"
",$customer/LAST_NAME) as $customer_name

  let $sum := fn:sum($orderGroup/TOTAL_ORDER_AMOUNT)

  return

   <CUSTOMER_RATING>

       <CUSTOMER_ID>{$customer_name}</CUSTOMER_ID>

       <RATING> {

        if ($sum>=10000) then

        "GOLD"

         else if ($sum<5000) then

        "REGULAR"

         else

        "SILVER"

       }

       </RATING>

    </CUSTOMER_RATING>
```

8. Confirm that the code is as displayed in Figure 19-19; namespaces may be different in your application.

Figure 19-19   Source View of If-Then-Else Function

9. Open XQueries.ds in XQuery Editor View.

Figure 19-20   XQuery Editor View of If-Then-Else Function

10. Open XQueries.ds in Test View and test the function; no parameters are required. When you run the query you will see results organized according to the following levels of purchases:

Gold for total orders >= 10000

Silver for total orders >= 5000 and <10000

Regular for total orders below 5000

The customer's full name and level are also shown. The XQuery function appears below.
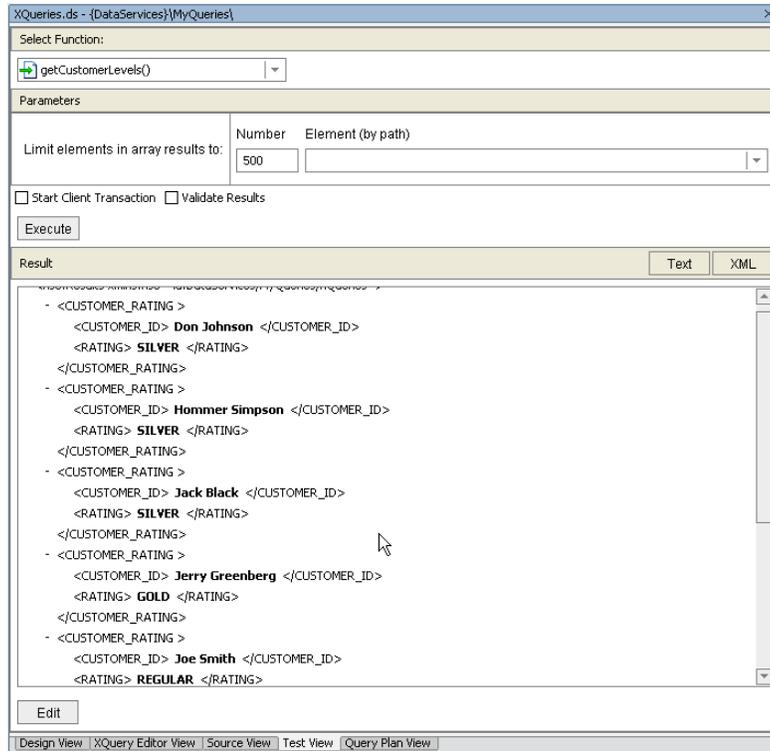
Figure 19-21   Test View of If-Then-Else Function

XQuery Code Reference for If-Then-Else Function

```
declare function tns:getCustomerLevels() as element(ns0:Results) {
  <ns0:Results>
  {
          for $customer in ns1:CUSTOMER()
          for $orders in ns3:CUSTOMER_ORDER()
          where $customer/CUSTOMER_ID eq $orders/CUSTOMER_ID
          group $orders as $orderGroup by fn:concat($customer/FIRST_NAME,"
",$customer/LAST_NAME) as $customer_name
          let $sum := fn:sum($orderGroup/TOTAL_ORDER_AMOUNT)
          return
                  <CUSTOMER_RATING>
                                  <CUSTOMER_ID>{$customer_name}</CUSTOMER_ID>
                                  <RATING> {
                                          if ($sum>=10000) then
                                          "GOLD"
                                                  else if ($sum<5000) then
                                          "REGULAR"
                                                  else
                                          "SILVER"
                                  }
                                  </RATING>
                          </CUSTOMER_RATING>
  }
  </ns0:Results>
  };
```

Lab 19.8   Creating a Union and Concatenation

This example demonstrates how to integrate data from two different data sources and present the results in a single report that lets you view the data source information as two separate variables.

Objectives

In this lab, you will:

> Create a function that gathers results from two order entry systems: RTLAPPLOMS and RTLELECOMS.

> Test the function.

Instructions

1. Add a new function to XQueries.ds and name it getCombinedOrders.

2. Open XQueries.ds in Source View.

3. Define the function declaration, by removing the asterisk * and adding the following parameter:

    ```
    $customer_id as xs:string
    ```

4. Split the <ns0:Results/> element into open and end tags, with curly braces ({}) in between.

5. Open XQuery Construct Palette and then drag and drop the FLWR construct between the curly braces.

6. Set the for clause using a $customer variable that is associated with CUSTOMER() located in the CustomerDB\CUSTOMER.ds folder within Data Services Palette.

    ```
     for $customer in ns1:CUSTOMER()
    ```

7. Set the let clause, using a $applOrder variable that is associated with CUSTOMER_ORDER(), which is located in the ApparelDB\CUSTOMER_ORDER.ds folder within Data Services Palette.

```
              let $applOrder:= for $order1 in ns4:CUSTOMER_ORDER()
```

8. Set the where clause as follows:

```
      where $customer/CUSTOMER_ID = $order1/C_ID
```

9. Set the return clause, as follows:

```
return

    $order1

let $elecOrder := for $order2 in ns3:CUSTOMER_ORDER()

    where ($order2/CUSTOMER_ID eq $customer/CUSTOMER_ID)

    return

        $order2

where ($customer/CUSTOMER_ID eq $customer_id)

return

 <CUSTOMER>

    {$customer}

        <Orders>

           {$applOrder, $elecOrder }

        </Orders>

 </CUSTOMER>
```

Note: ns3:CUSTOMER_ORDER() refers to CUSTOMER_ORDER.ds in ElectronicsDB folder

10. Confirm that the code is as displayed in Figure 19-22; the namespaces may be vary in your application.
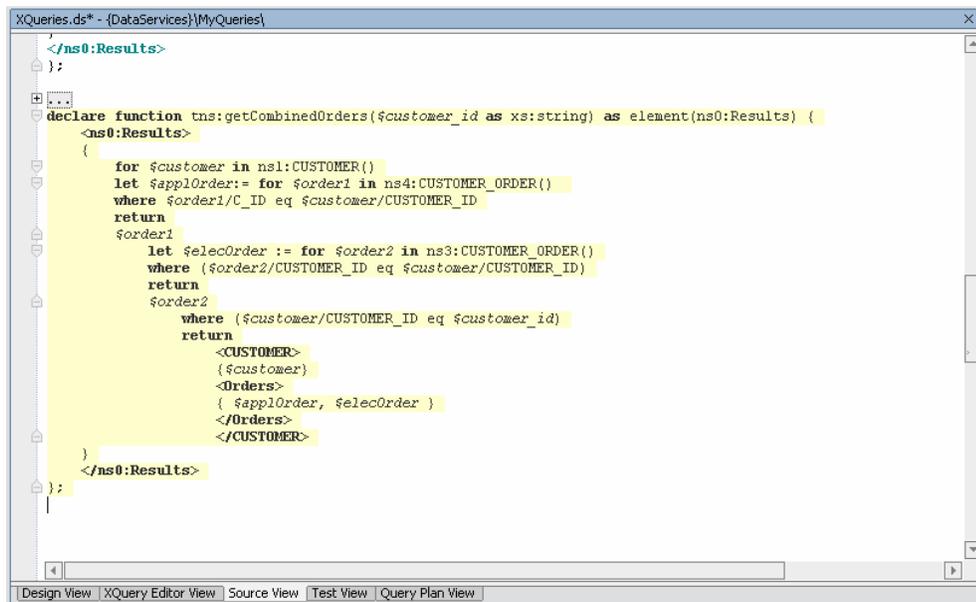
Figure 19-22   Source View for Union and Concatenation Function

11. Open XQueries.ds in XQuery Editor View.
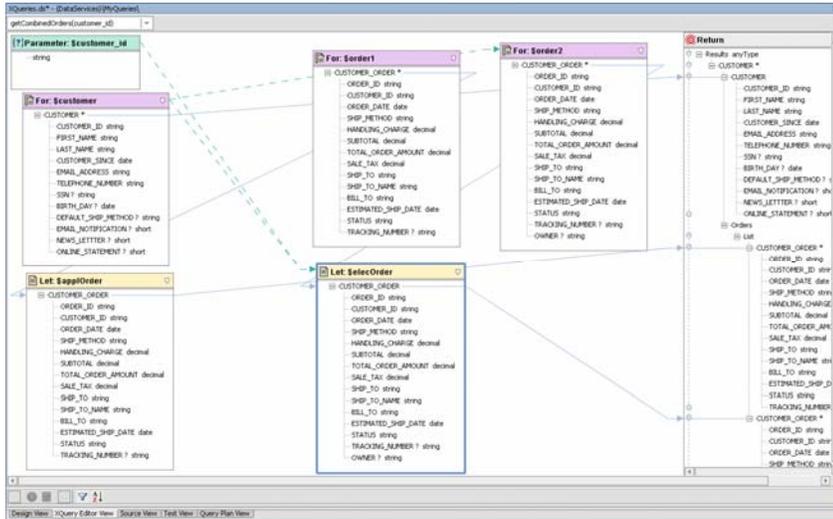
Figure 19-23   XQuery Editor View of Union and Concatenation Function

12. Open XQueries.ds in Test View, and then test the getCombinedOrders() function using CUSTOMER3 as the parameter. The XQuery function appears below.

Figure 19-24   Test View of Union and Concatenation Function

XQuery Reference Code for Union and Concatenation Function

```
declare function tns:getCombinedOrders($customer_id as xs:string) as
element(ns0:Results) {

    <ns0:Results>
```

```
{
    for $customer in ns1:CUSTOMER()
    let $applOrder:= for $order1 in ns4:CUSTOMER_ORDER()
        where ($order1/C_ID eq $customer/CUSTOMER_ID)
        return
        $order1
    let $elecOrder := for $order2 in ns3:CUSTOMER_ORDER()
        where ($order2/CUSTOMER_ID eq $customer/CUSTOMER_ID)
        return
        $order2
    where ($customer/CUSTOMER_ID eq $customer_id)
    return
        <CUSTOMER>
            {$customer}
            <Orders>
                { $applOrder, $elecOrder }
            </Orders>
        </CUSTOMER>
}
</ns0:Results>
};
```

Lesson Summary

In this lesson you, learned how to:

Use Source View to add, edit, or delete XQuery code that defines a data service's query functions.

Compare the coded query with the XQuery Editor View.

Lesson 20 Implementing Relationship Functions and Logical Modeling

Relationship functions return data combined from two or more data services. For example, by creating a relationship between the Address and Customer data services, you can obtain the address for a given customer. Or by creating a relationship between the Customer and Order Management data services, you can receive data that identifies all orders returned by a particular customer.

Model diagrams are used to view a selected set of data services and the relationships between them. The model shows the basic structure of the data returned by the data service. The main purpose of the diagram is to help you envision meaningful subsets of your enterprise data relationships, but it can also be used to define new artifacts or edit existing artifacts.

Logical modeling is an extension of the physical modeling that you learned about in Lesson 5. There are three labs in this lesson, which are to be completed in sequential order. The labs in this lesson are dependent on the work completed in the previous lessons.

Objectives

After completing this lesson, you will be able to:

Create model diagrams for a logical data service.

Define relationships between data services.

View and implement multiple relationship functions.

Test multiple relationship functions.

Overview

To help you get from a complex, distributed physical data landscape to a more holistic view of enterprise information, DSP supports a visual, model-driven approach to developing data services. Modeling provides a graphical representation of the data resources in your environment, providing a bird's-eye view of a large system or giving you a way to create "zoomed" views of enterprise areas. In a model diagram data services appear as boxes, while relationships appear as annotated lines connection the data service representations. A relationship is only visible if both end points are also on the diagram.

The result is real-time access to externally persisted data through a logical data model.

Lab 20.1 Implementing and Testing a Relationship Function

The getCustomer_Order() function that is intended to return customer order information for a specific customer. However, to accomplish that you need to add the ApparelDB data service's CUSTOMER_ORDER as a source schema, and then create a relationship with the target schema.

Objectives

In this lab you will:

Implement a relationship function, using XQuery Editor View to define the return data service, by:

   o   Identifying the data source.

   o   Creating an overwrite map between source and target elements.

   o   Creating a simple map between a parameter and a source element.

Test the relationship function created as a result of the mappings.

Instructions

1. Open CUSTOMER.ds in XQuery Editor View. (The file is located in the DataServices\CustomerDB folder.)

2. Select getCustomer_Order(arg) from the Function drop-down list.
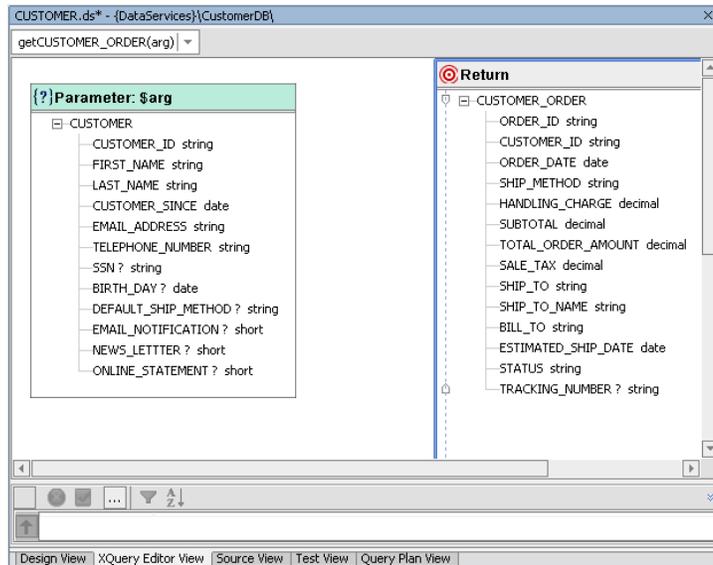
Figure 20-1   XQuery Editor View of getCustomer_Order Function

3. In Data Services Palette, expand the ApparelDB and CUSTOMER_ORDER.ds folders.

4. Drag and drop CUSTOMER_ORDER() into XQuery Editor View.

5. In XQuery Editor View, create an overwrite mapping between the CUSTOMER_ORDER source and Return elements by completing the following steps:

   a. Press Ctrl.

   b. Drag and drop the source node's CUSTOMER_ORDER* element onto the Return type's CUSTOMER_ORDER element.

6. Drag and drop the parameter's CUSTOMER_ID element onto the source node's C_ID element. Confirm that the getCustomer_Order function is as displayed in Figure 20-2.
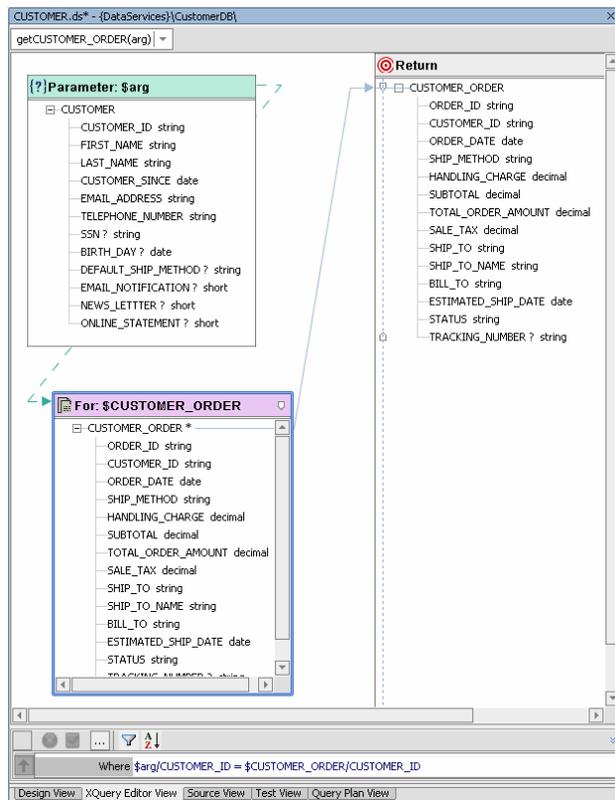
Figure 20-2 Joined and Mapped Function

7. Save your work and then build the DataServices project.

8. Open CUSTOMER.ds in Test View and run a test by completing the following steps:

Select getCUSTOMER_ORDER(arg) from the Function drop-down list.

Click Browse, navigate to, and open the <beahome>\weblogic81\samples\LiquidData\EvalGuide directory.

Select the customer.xml file.
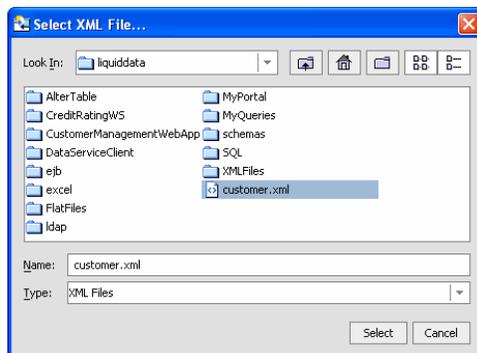
Figure 20-3 Select XML File

9. Click Select. The contents of the file are inserted into the Parameters field.
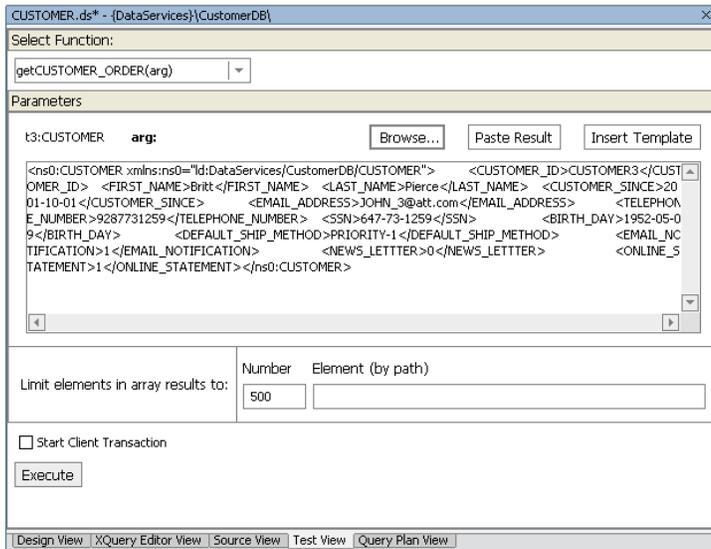
Figure 20-4   Select XML File

10. Click Execute. The order information for CUSTOMER3 should appear.
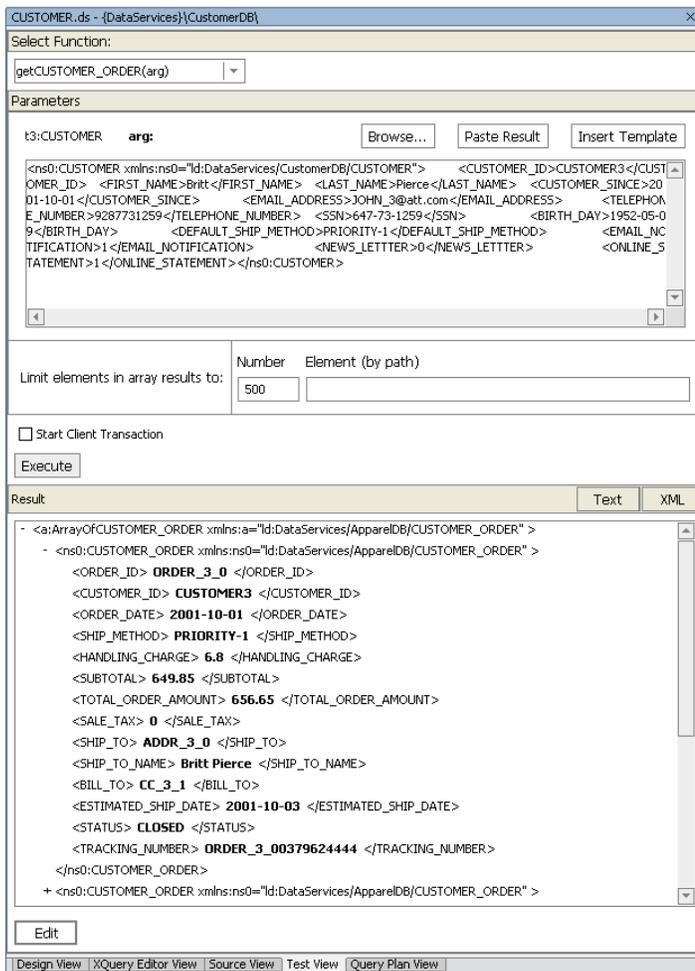
Figure 20-5   Relationship Test Results

Lab 20.2 Creating a Model Diagram for Logical Data Services

Model diagrams display the basic structure of the data returned by a data service. A model diagram lets you view a selected set of data services and the relationships between them. The main purpose of the diagram is to help you envision meaningful subsets of the model, but it can also be used to define new artifacts or edit existing artifacts.

Objectives

In this lab, you will:

> Import a schema that provides a logical and unified representation of two separate physical data sources.

> Create a basic model diagram by adding data services to the imported logical data service.

> Create relationship functions between the modeled data services.

Instructions

1. Import the OrderManagement schema into the DataServices project folder by completing the following steps:

   Right-click the DataServices project folder.

   a. Choose Import.

   b. Navigate to and open the <beahome>\weblogic81\samples\LiquidData\EvalGuide directory.

   c. Select the OrderManagement folder.

   d. Click Import. A new folder, OrderManagement, is created in the DataServices project. The imported schema contains logical representations of the two Order Management Systems (Apparel and Electronics), which make the two systems appear as if they are a single Order Management System.

2. Create a sub-folder within the Models folder by completing the following steps:

   a. Right-click the MODELS folder, located in the DataServices folder.

   b. Choose New → Folder.

   c. Enter Logical in the Name field.

   d. Click OK.

3. Create a new logical model diagram by completing the following steps:

   a. Right-click the Logical folder.

   b. Choose New → Model Diagram.

   c. Enter OrderManagement_Logical_Model.md in the Name field.

   d. Click Create.

4. Create a model for the OrderManagement data services by completing the following steps:

   a. Expand the CustomerManagement, OrderManagement, and ServiceDB folders.

   b. Drag and drop the following .ds files into the model:

Data Service File	Located In:
`customerProfile.ds`	`CustomerManagement`
`address.ds`	`OrderManagement`
`customer.ds`	`OrderManagement`
`customerOrder.ds`	`OrderManagement`
`customerOrderLineItem.ds`	`OrderManagement`
`orders.ds`	`OrderManagement`
`product.ds`	`OrderManagement`
`service_case.ds`	`ServiceDB`

Your model diagram should be similar to that displayed in Figure 20-6. Notice that relationships between data services already exist. These relationships were generated during the Import Source Metadata process, and are based on the foreign key relationship defined in the underlying relational data.
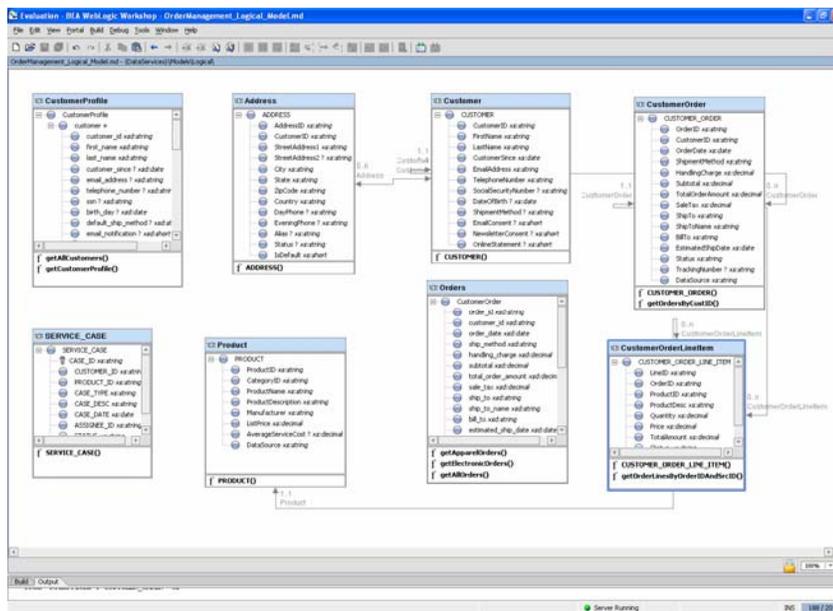
Figure 20-6   Model Diagram for Logical Data Services

5. Create a relationship between the CustomerProfile and ADDRESS data services by completing the following steps:

    a. Drag and drop the customer_id element (CustomerProfile) onto the CustomerID element (Address).

    b. Click Finish in the Relationship Properties window.

6. Create a relationship between CustomerProfile and SERVICE_CASE data services by completing the following steps:

    a. Drag and drop the customer_id (CustomerProfile) onto the CUSTOMER_ID element (SERVICE_CASE).

    b. Click Finish in the Relationship Properties window.
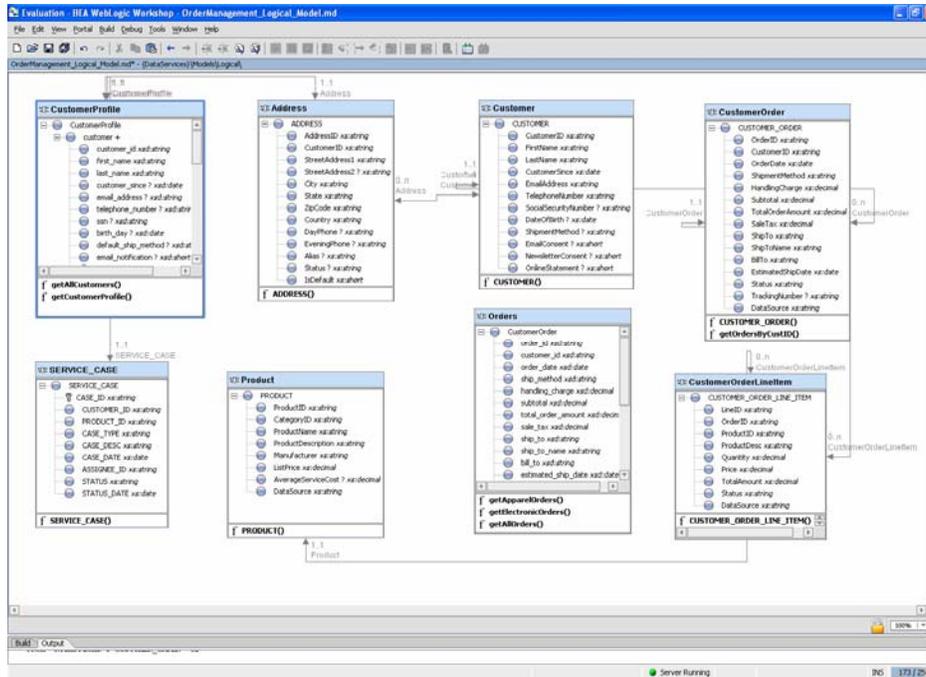
Figure 20-7   New Relationships Defined

7. Open CustomerProfile.ds in Design View. You should see two new relationship functions,
    getAddress1 (which navigates to the Address logical data service, located in OrderManagement)
    and getSERVICE_CASE1 (which navigates to the SERVICE_CASE physical data service,
    located in ServiceDB).
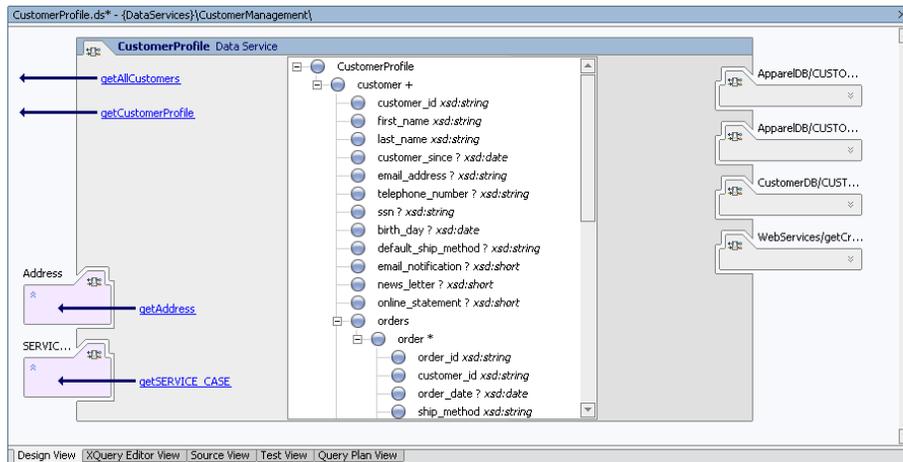
Figure 20-8   New Functions

8. Save your work.

Lesson 21 Running Ad Hoc Queries

Sometimes it is necessary to execute a query on functions associated with an application that is already deployed. Rather than take the application offline to create a new query, DSP provides the PreparedExpression class, which lets you create and run ad hoc queries on deployed applications.

Objectives

After completing this lesson, you will be able to:

Create an ad hoc query from within a DSP application.

Run an ad hoc query.

Overview

DSP includes a PreparedExpression class that lets you build an ad hoc query using remote data sources, and then execute it via the Mediator API or DSP Control. Using the methods within the PreparedExpression class, you can build queries on top of existing XDS functions belonging to applications already deployed on an active local or remote server domain.

The process for running an ad hoc query is as follows:

1. Create a StringBuffer to hold the query.

2. Create an instance of the PreparedExpression class, using the prepareExpression method.

3. Create parameters for the ad hoc query, using the bind<DataType> methods.

4. Submit the query and review the results, using the Mediator API or DSP Control.

Lab 21.1 Creating an Instance of the PreparedExpression Class

The first steps in creating an ad hoc query are to instantiate a StringBuffer and the PreparedExpression class. For the latter instance, you use the prepareExpression method of the DataServiceFactory class, which accepts three parameters:

InitialContext

Application Name

XQuery String

For example:

```
PreparedExpression pe = DataServiceFactory.prepareExpression(
getInitialContext(),
"Evaluation",
xquery.toString()
      );
```

Objectives

In this lab, you will:

Build a StringBuffer instance to hold the ad hoc query.

Create an instance of the PreparedExpression class.

Instructions

1. Create a new Java project in the Evaluation application, and name it AdHocClient.

2. Create a new Java class in the AdHocClient project, and name it AdHocQuery.

3. Open AdHocQuery.java.

4. Import the following Java classes:

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

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

import com.bea.xml.XmlObject;

import javax.naming.InitialContext;

import javax.naming.NamingException;

import javax.xml.namespace.QName;

import weblogic.jndi.Environment;
```

Note: You can also import the necessary Java classes by first adding the code specified below, and then pressing Alt + Enter.

5. Specify the initial context for the query, by adding the following code after the first curly brace:

```
public static InitialContext getInitialContext() throws NamingException {
    Environment env = new Environment();
    env.setProviderUrl("t3://localhost:7001");
    env.setInitialContextFactory("weblogic.jndi.WLInitialContextFactory");
    env.setSecurityPrincipal("weblogic");
    env.setSecurityCredentials("weblogic");
    return new InitialContext(env.getInitialContext().getEnvironment());
}
```

6. Add the main argument, by adding the following code after the initial context:

```
public static void main (String args[]) {
    System.out.println("==================== Ad Hoc Client ====================");
    try {
      } catch (Exception e) {
      e.printStackTrace();
    }
}
```

7. Build a StringBuffer instance to hold your query. For example, add the following code after the line: `try {`:

```
StringBuffer xquery = new StringBuffer();

xquery.append("declare variable $p_firstname as xs:string external;
\n");
xquery.append("declare variable $p_lastname as xs:string external;
\n");

xquery.append("declare namespace
ns1=\"ld:DataServices/MyQueries/XQueries\"; \n");
xquery.append("declare namespace
ns0=\"ld:DataServices/CustomerDB/CUSTOMER\"; \n\n");

xquery.append("<ns1:RESULTS>                                  \n");
xquery.append("{                                              \n");
xquery.append("   for $customer in ns0:CUSTOMER()             \n");
xquery.append("   where ($customer/FIRST_NAME eq $p_firstname \n");
xquery.append("       and $customer/LAST_NAME eq $p_lastname) \n");
xquery.append("   return                                      \n");
xquery.append("      $customer                                \n");
xquery.append(" }                                             \n");
xquery.append("</ns1:RESULTS>                                 \n");
```

8. Use the prepareExpression method of the Mediator API's DataServiceFactory class to create an instance of the PreparedExpression class, by adding the following code:

```
PreparedExpression pe = DataServiceFactory.prepareExpression(
getInitialContext(), "Evaluation",  xquery.toString());
```

Lab 21.2   Defining Ad Hoc Query Parameters

After you create an instance of the PreparedExpression class, you need to specify the parameters that will be passed when the ad hoc query is submitted. To pass parameters, you use one or more bind<DataType> methods, such as bindString and bindInt.

Objectives

In this lab, you will:

> Use the bind<DataType> methods of the PreparedExpression instance to pass parameters.

> Invoke the query.

> Display the query's XML results.

Instructions

1. Pass parameters by using the bindString method of the PreparedExpression instance. For example, add the following code to the AdHocQuery.java file:

    ```
    pe.bindString(new QName("p_firstname"), "Jack");

    pe.bindString(new QName("p_lastname"), "Black");
    ```

2. Invoke the executeQuery method to return the query results in an XmlObject.

    ```
    XmlObject obj = pe.executeQuery();
    ```

3. Enter the code necessary to return the XmlObject and display the XML. For example:

    ```
    System.out.println(obj.toString());
    ```

Lab 21.3   Testing the Ad Hoc Query

You are now ready to test the ad hoc query, which is set to return information for Jack Black.

Objectives

In this lab, you will:

> Build the AdHocClient project.

> Run the AdHocQuery.java

Instructions

1. Build the AdHocClient project.

2. In the AdHocQuery.java application, click the Start icon (or press Ctrl + F5).

3. Confirm that you can retrieve customer profile information for Jack Black.


```
Build  Output                                                         ×
Trying to create process and attach to 1981...
D:\bea\jdk142_05\bin\javaw.exe -Xdebug -Xnoagent -Djava.compiler=NONE
Process started
Attached successfully.
==================== Data Service Client ====================
<ns0:RESULTS xmlns:ns0="ld:DataServices/MyQueries/XQueries">
   <ns1:CUSTOMER xmlns:ns1="ld:DataServices/CustomerDB/CUSTOMER">
     <CUSTOMER_ID>Jack</CUSTOMER_ID>
     <FIRST_NAME>Jack</FIRST_NAME>
     <LAST_NAME>Black</LAST_NAME>
     <CUSTOMER_SINCE>2001-10-01</CUSTOMER_SINCE>
     <EMAIL_ADDRESS>Jack@hotmail.com</EMAIL_ADDRESS>
     <TELEPHONE_NUMBER>2145134119</TELEPHONE_NUMBER>
     <SSN>295-13-4119</SSN>
     <BIRTH_DAY>1970-01-01</BIRTH_DAY>
     <DEFAULT_SHIP_METHOD>AIR</DEFAULT_SHIP_METHOD>
     <EMAIL_NOTIFICATION>1</EMAIL_NOTIFICATION>
     <NEWS_LETTTER>0</NEWS_LETTTER>
     <ONLINE_STATEMENT>1</ONLINE_STATEMENT>
   </ns1:CUSTOMER>
</ns0:RESULTS>

Debugging Finished
```

Code Reference for an Ad Hoc Query

```java
import com.bea.ld.dsmediator.client.DataServiceFactory;
   import com.bea.ld.dsmediator.client.PreparedExpression;
   import com.bea.xml.XmlObject;
   import javax.naming.InitialContext;
   import javax.naming.NamingException;
   import javax.xml.namespace.QName;
   import weblogic.jndi.Environment;

   public class AdHocQuery
   {
   public static InitialContext getInitialContext() throws NamingException {
           Environment env = new Environment();
                   env.setProviderUrl("t3://localhost:7001");
           env.setInitialContextFactory("weblogic.jndi.WLInitialContextFactory");
                   env.setSecurityPrincipal("weblogic");
                   env.setSecurityCredentials("weblogic");
                   return new
InitialContext(env.getInitialContext().getEnvironment());
   }
     public static void main (String args[]) {
           System.out.println("==================== Ad Hoc Client
====================");
           try {
                   StringBuffer xquery = new StringBuffer();

xquery.append("declare variable $p_firstname as xs:string external; \n");
xquery.append("declare variable $p_lastname as xs:string external;  \n");

xquery.append("declare namespace ns1=\"ld:DataServices/MyQueries/XQueries\"; \n");
xquery.append("declare namespace ns0=\"ld:DataServices/CustomerDB/CUSTOMER\"; \n\n");

xquery.append("<ns1:RESULTS>                                        \n");
xquery.append("{                                                    \n");
xquery.append("    for $customer in ns0:CUSTOMER()                  \n");
xquery.append("    where ($customer/FIRST_NAME eq $p_firstname      \n");
xquery.append("       and $customer/LAST_NAME eq $p_lastname)       \n");
xquery.append("    return                                           \n");
xquery.append("        $customer                                    \n");
xquery.append(" }                                                   \n");
xquery.append("</ns1:RESULTS>                                       \n");

PreparedExpression pe = DataServiceFactory.prepareExpression(getInitialContext(),
"Evaluation",  xquery.toString());
pe.bindString(new QName("p_firstname"), "Jack");
pe.bindString(new QName("p_lastname"), "Black");
XmlObject results = pe.executeQuery();
System.out.println(results);

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

Lesson Summary

In this lesson, you learned how to:

Create a StringBuffer instance to hold the ad hoc query.

Create an instance of the PreparedExpression class, using the prepareExpression method of the Mediator API's DataServiceFactory class.

Create parameters for the ad hoc query, using the bindString method of the PreparedExpression class.

Submit the query and review the results, using the Mediator API.

Review the XML output.

Lesson 22 Creating Data Services Based on SQL Statements

The SQL-Exit feature lets developers re-use SQL statements that are currently available in the source system. These user-defined SQL statements are bound in XQuery as external functions, in the same manner as all DSP sources.

Objectives

After completing this lesson, you will be able to:

Create data service based on a user-defined SQL statement.

Use that data service to retrieve customer and address information together.

Overview

Configuring the SQL-exit data source involves the following steps:

1. Create the .xsd schema that describes the SQL results.

2. Create the data service, including annotations, describing the result set.

3. Associate an XML Type for the data service to the schema previously created.

When a user-defined SQL statement is used within other functions, the DSP engine will bind the SQL statement as a sub-query in a new SQL statement. To disable this functionality, the metadata property isSubquery, stored in the function's pragma, can be set to value false.

Lab 22.1 Creating a Data Service from a User-Defined SQL Statement

The SQL statement that will be used to create a new data service involves a join between the CUSTOMER and ADDRESS data services. You need to manually add all the necessary metadata to the new data service, before this query can execute. To do so, you will use metadata previously imported from the CUSTOMER and ADDRESS tables.

Objectives

In this lab, you will:

Import an SQL statement as source metadata for a physical data service.

Generate a new data service.

Instructions

1. Open the SQL_Statement.txt file, located in the
   <beahome>\weblogic81\samples\LiquidData\EvalGuide folder.

2. Copy the text within the file. The text is:

   ```
   select "A"."CUSTOMER_ID", "A"."FIRST_NAME", "A"."LAST_NAME",
   "B"."ADDR_ID", "B"."CITY", "B"."STATE", "B"."ZIPCODE", "B"."COUNTRY"
   from "RTLCUSTOMER"."CUSTOMER" "A", "RTLCUSTOMER"."ADDRESS" "B" where
   "A"."CUSTOMER_ID" = "B"."CUSTOMER_ID" AND "B"."STATE" = ?
   ```

3. Create a new folder in the DataServices project and name it SQL. You will use this folder to store a new data service based on user-defined SQL statements.

---

4. Right-click the SQL folder and select Import Source Metadata.

5. Select Relational from the Data Source Type and click Next.

6. Select the SQL statement radio button and click Next. The SQL Statement page opens.

7. Paste the copied text into the SQL Statement field.

8. Select VARCHAR from the Type column for Position 1 and click Next. The Summary page opens.

9. Rename the data service to MySQL.
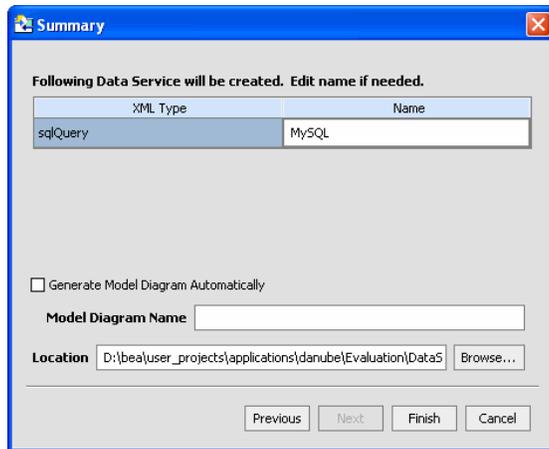
Figure 22-1   Summary for SQL-Based Data Service

10. Click Finish. The MySQL data service and associated schema files are added to the SQL folder.

Lab 22.2   Testing Your SQL Data Service

You are now ready to test whether the MySQL data service can retrieve all customers who reside in California.

Objectives

In this lab, you will:

> Test the MySQL data service.

> View the results.

Instructions

1. Open MySQL.ds in Test View.

2. Select MySQL(x1) from the Function drop-down list.

3. In the parameter box enter CA

4. Click Execute. The result set will show customer and address information for the state of California.

Figure 22-2   Test Results for an SQL-Based Data Service

Lesson Summary

In this lesson, you learned how to:

Manually create a data service out of an SQL statement.

Test the SQL-based data service.

Lesson 23 Performing Custom Data Manipulation Using Update Override

DSP permits customized updates through the use of the *update override* feature. The update override logic, which is triggered prior to submitting data, can be used for custom data manipulation, update overrides, logging, debugging, or other custom logic needs.

In this lesson, you will write an update override that computes total orders, based on the quantity and price of each order.

Objectives

After completing this lesson, you will be able to:

> Write customized data manipulation through an update override.

> Associate an update override with a data service.

Overview

An update override, which you assign to a data service, performs custom logic prior to submitting data. The update override is a Java class that implements the com.bea.sdo.mediator.UpdateOverride class. Using that class's performChange (DataGraph graph) method, a Data Graph instance of the current data service is returned. The Data Graph can then be manipulated in via the update override logic.

For example, you can get the CustomerProfileDocument DataObject through the data graph

```
(CustomerProfileDocument) graph.getRootObject();
```

You could also get the Change Logging summary through graph.getChangeSummary()

On return of the Data Graph, the following conditions apply:

> Return true: Proceed with the rest of update.

> Return false: Stop the update.

> Throw Exception: Rollback.

Lab 23.1   Creating an Update Override

An update override enables custom manipulation of data within data service.

Objectives

In this lab, you will:

> Create a new Java class that will serve as the basis for an update override.
>
> Import and implement an update override class.
>
> Implement the performChange method.
>
> Write customized update logic.

Instructions

1. Create a new Java class by completing the following steps:

    a. Right-click the CustomerManagement folder, located in the DataServices folder.

    b. Choose New → Java Class.

    c. Enter CustomerProfileExit in the File Name field.

    d. Click Create.

2. Build the DataServices project.

3. Open the CustomerProfileExit.java file.

4. Import and implement the update override, by completing the following steps:

    a. Import the update override by entering the following code:

    ```
    import com.bea.ld.dsmediator.update.UpdateOverride;
    ```

    b. Implement the update override by modifying the public class CustomerProfileExit code, as follows:

    ```
    public class CustomerProfileExit implements UpdateOverride
    ```

    c. Press Alt + Enter, and then click OK to add the performChange(DataGraph) signature.

    d. Implement the performChange(DataGraph graph) method by modifying the code to read as follows:

    ```
    public boolean performChange(DataGraph graph)
    ```

    The DataGraph passed in the argument contains the current SDO instance with all changes, including the change summary.

5. Access the update override by casting the root object of the data graph to your SDO. Add the following code, after the opening braces:

    ```
    CustomerProfileDocument customerDocument =
                (CustomerProfileDocument) graph.getRootObject();
    ```

6. Press Alt+Enter. With this CustomerProfileDocument instance, you can get and set values that will be applied to the SDO before it is submitted.

---

7. Write update logic to compute the total order amount, based on the sum of each order item's quantity multiplied by its price (sum of price*qty). You can use this to get the total of each item's quantity*price and to set the total order amount to this value. (<u>Note:</u> Use BigDecimals for computations.) For example:

```
Order[] orders =
customerDocument.getCustomerProfile().getCustomerArray(0).getOrders()
.getOrderArray();


for (int x=0; x<orders.length; x++) {

BigDecimal total = new BigDecimal(0);

OrderLine[] items = orders[x].getOrderLineArray();

for (int y=0; y < items.length; y++) {

total =
total.add(items[y].getQuantity().multiply(items[y].getPrice()));

}

orders[x].setTotalOrderAmount(total);

}
```

8. Press Alt + Enter, for all flagged items.

9. Enter the code necessary to return the results. For example:

```
System.out.println(">>> CustomerProfile.ds Exit completed");

return true;

}

}
```

10. Confirm that your code is as displayed in Figure 23-1.

11. Build DataServices project.
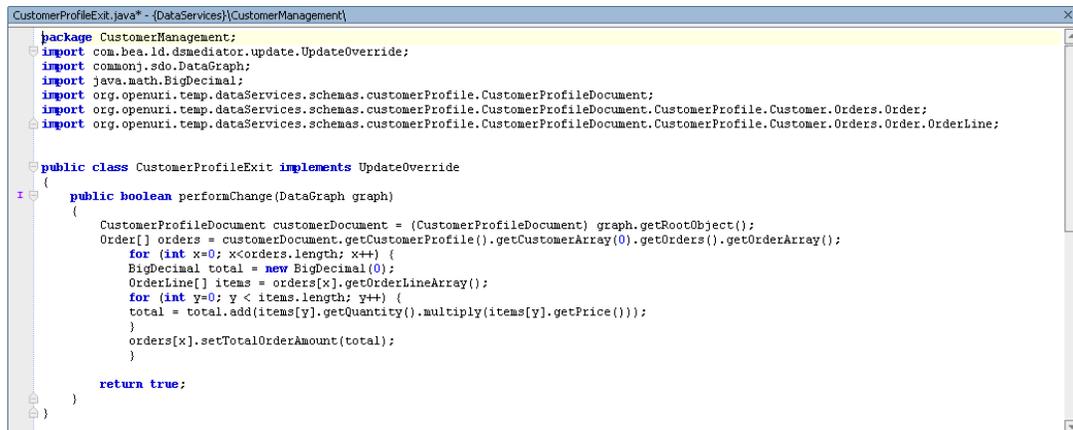

```
CustomerProfileExit.java* - {DataServices}\CustomerManagement\                                                                                    ×
    package CustomerManagement;
    import com.bea.ld.dsmediator.update.UpdateOverride;
    import commonj.sdo.DataGraph;
    import java.math.BigDecimal;
    import org.openuri.temp.dataServices.schemas.customerProfile.CustomerProfileDocument;
    import org.openuri.temp.dataServices.schemas.customerProfile.CustomerProfileDocument.CustomerProfile.Customer.Orders.Order;
    import org.openuri.temp.dataServices.schemas.customerProfile.CustomerProfileDocument.CustomerProfile.Customer.Orders.Order.OrderLine;


    public class CustomerProfileExit implements UpdateOverride
    {
        public boolean performChange(DataGraph graph)
        {
            CustomerProfileDocument customerDocument = (CustomerProfileDocument) graph.getRootObject();
            Order[] orders = customerDocument.getCustomerProfile().getCustomerArray(0).getOrders().getOrderArray();
                for (int x=0; x<orders.length; x++) {
                BigDecimal total = new BigDecimal(0);
                OrderLine[] items = orders[x].getOrderLineArray();
                for (int y=0; y < items.length; y++) {
                total = total.add(items[y].getQuantity().multiply(items[y].getPrice()));
                }
                orders[x].setTotalOrderAmount(total);
                }

            return true;
        }
    }
```

Figure 23-1   Update Override Code

Lab 23.2   Associating an Update Override to a Logical Data Service

Before you can use the update override, you must associate it with a specific data service.

Objectives

In this lab you will:

> Use the Property Editor to associate an update override with a specific data service.
>
> Build the data service to include the update override.

Instructions

1. Open the CustomerProfile data service in Design View.

2. Click the CustomerProfile header to activate the Property Editor. (If the Property Editor is not open, press Alt + 6.)

3. Click the update override class field.

4. Navigate to the DataServices.jar\CustomerManagement folder.

5. Select CustomerProfileExit.class and click Open. The update override class field is now populated with the CustomerManagement.CustomerProfileExit.

6. Build the DataServices project.

Lab 23.3   Testing the Update Override

As with any other data service, you should test the update override to ensure that it works properly.

Objectives

In this lab you will:

> Change order information from within your CustomerManagementWebApp application.
>
> Confirm update override results.

Instructions

1. Open CustomerPageFlowController.jpf, which is located in the CustomerManagementWebApp\CustomerPageFlow folder.

2. Click the Start icon. The Workshop Test Browser opens.

3. Enter CUSTOMER3 in the CUSTOMER ID field and click Submit.

    Note: It may take a few seconds before the information is returned.

4. Change the order information by adding, modifying or deleting order lines.

5. Click Submit All Changes.

6. Click Back to return to the CUSTOMER ID page.

7. Enter CUSTOMER3 in the CUSTOMER ID field and click Submit.

8. Confirm if the updated total order information was computed.

Update Override Reference Code

```
package CustomerManagement;

import com.bea.ld.dsmediator.update.UpdateOverride;

import commonj.sdo.DataGraph;

import java.math.BigDecimal;

import org.openuri.temp.dataServices.schemas.customerProfile.CustomerProfileDocument;

import org.openuri.temp.dataServices.schemas.customerProfile.CustomerProfileDocument.CustomerProfile.C

import
·g.openuri.temp.dataServices.schemas.customerProfile.CustomerProfileDocument.CustomerProfile.Customer.Or


public class CustomerProfileExit implements UpdateOverride
{
 public boolean performChange(DataGraph graph)
 {
     CustomerProfileDocument customerDocument = (CustomerProfileDocument) graph.getRootObject();

     Order[] orders = customerDocument.getCustomerProfile().getCustomerArray(0).getOrders().getOrderAr

         for (int x=0; x<orders.length; x++) {
         BigDecimal total = new BigDecimal(0);
         OrderLine[] items = orders[x].getOrderLineArray();
         for (int y=0; y < items.length; y++) {
         total = total.add(items[y].getQuantity().multiply(items[y].getPrice()));
         }
         orders[x].setTotalOrderAmount(total);
         }
     return true;
 }
}
```

Lesson Summary

In this lesson, you learned how to:

> Create an update override for a logical data service.
>
> Write logic in the update override to access the XML bean and perform custom data manipulation prior to submitting.
>
> Associate an update override to the data service.

Lesson 24 Updating Web Services Using Update Override

You can also use update overrides to update a Web service.

Objectives

After completing this lesson, you will be able to:

Write an update override function for performing manual updates.

View your results.

Overview

Unlike relational data sources, Web service updates are not automated, since DSP is unable to determine how to decompose a read function into a corresponding write. To enable DSP to perform the necessary writes, you must create an update override for the physical data service, and then implement the necessary writes in that update override. For example:

```
public class CreditRatingExit implements UpdateOverride {
    public boolean performChange(DataGraph datagraph){

    // don't do anything if there are no changes
        ChangeSummary cs = datagraph.getChangeSummary();
        if (cs.getChangedDataObjects().size()==0)
            return true;

    // get changed values from SDO
        GetCreditRatingResponseDocument creditRating =
(GetCreditRatingResponseDocument) datagraph.getRootObject();
        int newRating =
creditRating.getGetCreditRatingResponse().getGetCreditRatingResult().getRating();
        String customerId =
creditRating.getGetCreditRatingResponse().getGetCreditRatingResult().getCustomerId();

    // update CreditRating web service
        try {
            CreditRatingDBTestSoap ratingWS = new
CreditRatingDBTest_Impl().getCreditRatingDBTestSoap();
            CreditRating rating = new CreditRating(newRating,customerId);
            ratingWS.setCreditRating(rating);
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
        System.out.println("WEB SERVICE EXIT COMPLETE!");
        return true;
    }
}
```

Lab 24.1   Creating an Update Override for a Physical Data Service

WebLogic's clientgen utility generates a Web Service-specific client .jar file that client applications can use to invoke Web Services. You simply need to specify the WSDL URI, the name and location of the client .jar file to generate and a package structure. Clientgen is available as an ant task as well as a Java application that can be invoked from the command line.

For more information on clientgen see:

> http://e-docs.bea.com/wls/docs81/webserv/anttasks.html

Objectives

In this lab, you will:

Edit the WebLogic clientgen command to point to your WebLogic Server.

Run the clientgen utility.

Add the generated client .jar file to your application Library.

Instructions

Set WebLogic's clientgen command line utility to generate a web service client .jar file by completing the following steps:

1. Edit the setenv.cmd (located in the <beahome>\weblogic81\samples\LiquidData\EvalGuide folder) to point to your WebLogic Server installation. This will set the environment for running clientgen. For example:

   ```
   call d:\bea\weblogic81\server\bin\setWLSEnv.cmd

   set CLASSPATH=d:\bea\weblogic81\server\lib\webservices.jar;%CLASSPATH%

   echo %CLASSPATH%
   ```

2. Open a command prompt.

3. Navigate to the <*beahome*>\weblogic81\samples\LiquidData\EvalGuide folder.

4. Run setenv.cmd.

5. Run clientgen.cmd to generate CreditRatingWSClient.jar.

6. In WebLogic Workshop add CreditRatingWSClient.jar to your application's Libraries folder. (The .jar file should be located in the <beahome>\weblogic81\samples\LiquidData\EvalGuide folder.)

Lab 24.2 Writing Web Service Update Logic in the Update Override

You now should set the update override class to the CreditRatingExit. This will let you get any updated credit rating information, invoke the CreditRating web service, and pass in the new value.

Objectives

In this lab, you will:

Import the CreditRatingExit.java file into the WebServices folder.

Set the update override class to the CreditRatingExit.

Instructions

1. Right-click the WebServices folder, located in the DataServices folder.

2. Choose Import.

3. Navigate to the <beahome>\weblogic81\samples\LiquidData\EvalGuide folder and select CreditRatingExit.java.

4. Click Import.

5. Build the DataServices project.

6. Open getCreditRatingResponse.ds in Design View. (The file is located in the WebServices folder.)

7. In the Property Editor, set the update override class by selecting CreditRatingExit from the DataServices\WebServices folder.

8. Build the DataServices project.

Lab 24.3 Testing the Update Override

You are now ready to test whether the update override functions correctly.

Objectives

In this lab, you will:

Change a customer's credit rating.

View the results.

Instructions

1. Open CreditRatingDBTest.jws, located in the CreditRatingWS folder.

2. Click the Start icon. The Workshop Test Browser opens.

3. Enter CUSTOMER3 in the customer_id field and click getCreditRating.

4. Click the Test XML tab.

5. Copy the SOAP body for the getCreditRating function.

```
<getCreditRating xmlns="http://www.openuri.org/">

  <!--Optional:-->

  <customer_id>string</customer_id>
```

```
</getCreditRating>
```

6. Close the Workshop Test Browser.

7. Open getCreditRatingResponse.ds in Test View.

8. Paste the SOAP body into the Parameter field.

9. Change <customer_id>string</customer_id> to <customer_id>CUSTOMER3</customer_id>.

10. Click Execute.

11. Click Edit and modify the credit rating. The update override is functioning correctly if you can update the credit rating.
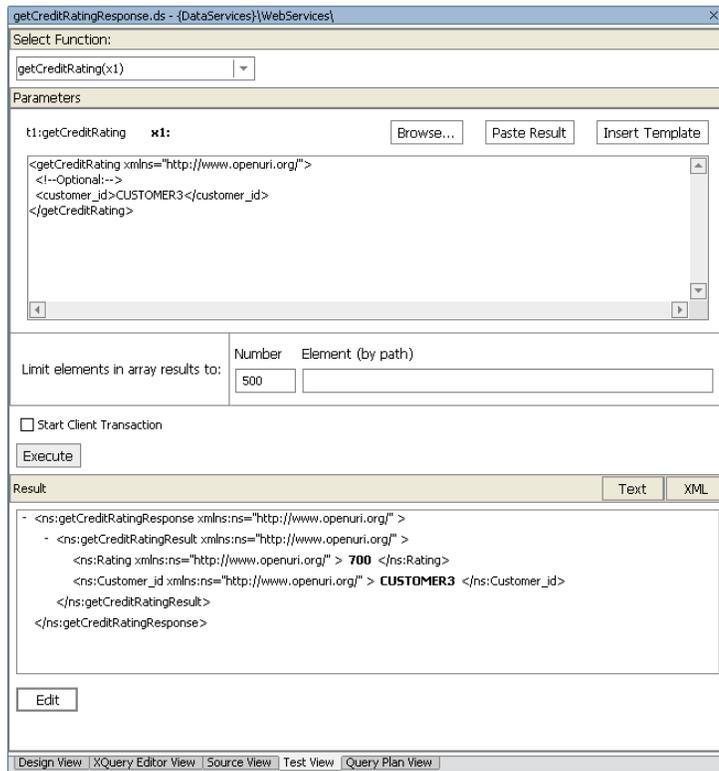
Figure 24-1   Test View of Update Override for a Web Service

Lab 24.4 Checking for Change Requirements

You can now use the web service to perform update overrides.

Objectives

In this lab you will:

> Change credit rating information from within your CustomerManagementWebApp application.

> Confirm update override results.

Instructions

1. Open CustomerPageFlowController.jpf, which is located in the CustomerManagementWebApp folder. The Workshop Test Browser opens.

2. Click the Start icon.

3. Enter CUSTOMER3 in the CUSTOMER ID field and click Submit.

4. Click Update Profile, change the credit rating information, click Submit, and then click Submit All Changes.

5. Confirm if the credit rating was updated, by clicking Back, entering CUSTOMER3 in the CUSTOMER ID field, and clicking Submit.
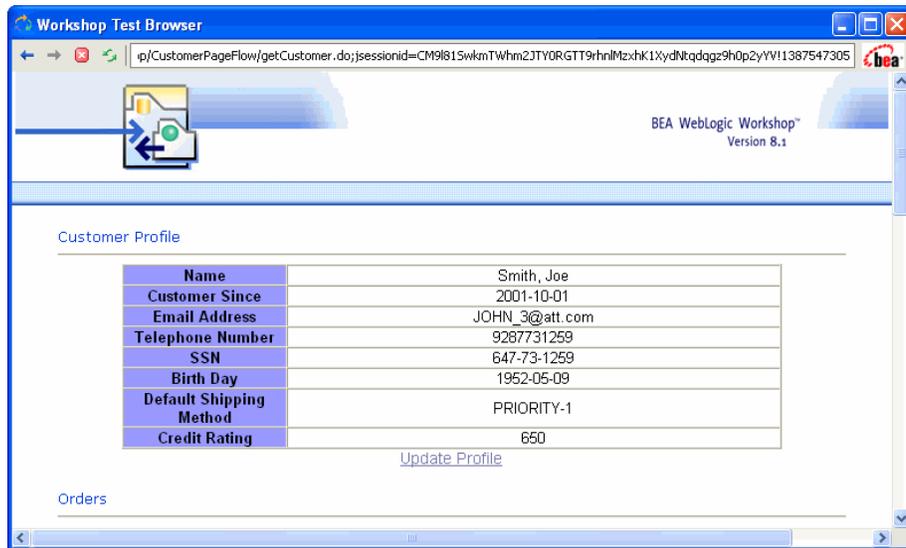
Figure 24-2   Workshop Test Browser View of Update Override Functionality

Lesson Summary

In this lesson, you learned how to:

> Create an update override for a physical data service (Web service)

> Associate the update override with a Web service client and write logic to invoke Web service update operations.

> Use the change summary to check whether there are changes needing to be written.

Lesson 25 Overriding SQL Updates Using Update Overrides

So far you have completed a few lessons on how update override functionality can be used for custom data manipulation and web service updates.

In this lesson you will learn how custom SQL updates can be used for performing manual updates to a relational source (table, view, stored procedure, or SQL Exit), using update overrides and JDBC.

Objectives

After completing this lesson, you will be able to:

Add update functionality to a previously created update override.

Write an update override for performing manual updates to a relational source (table, view, stored procedure, or update override) via JDBC.

Create an update override for a physical data service.

Setup the update override to be a JDBC client and write logic to update the database table.

Overview

Update overrides are useful in situations where you need to perform some custom updates or create a custom query.

In this particular case, since the previous update override lacks update functionality, you can add an update statement to the override.

Lab 25.1   Adding SQL Update Statements to an Update Override File

You can add SQL update statements to an update override file, thereby enabling custom data manipulations in relational databases.

Objectives

In this lab, you will:

Import the Java folder, which contains the MySQLExit.java file.

Add SQL update statements to the Java file.

Instructions

1. Right-click the SQL folder located in DataServices project, choose Import, and select the Java folder from the <beahome>\weblogic81\samples\LiquidData\EvalGuide folder.

2. Click Import and verify that the Java folder is added to the SQL folder.

3. Open MySQLExit.java, located in the DataServices\SQL\Java folder.

4. Locate the line "Type in your UPDATE SQL statements here".

5. Enter the two following SQL statements and store them into updateStr and updateStr1 respectively:

```
"UPDATE RTLCUSTOMER.CUSTOMER SET FIRST_NAME=?, LAST_NAME=? WHERE
CUSTOMER_ID=?";
```

```
"UPDATE RTLCUSTOMER.ADDRESS SET CITY=?, STATE=?, ZIPCODE=?, COUNTRY=?
WHERE ADDR_ID=?";
```

Your code should look like the following:

```
String updateStr = "UPDATE RTLCUSTOMER.CUSTOMER SET FIRST_NAME=?,
LAST_NAME=? WHERE CUSTOMER_ID=?";
```

```
String updateStr1 = "UPDATE RTLCUSTOMER.ADDRESS SET CITY=?, STATE=?,
ZIPCODE=?, COUNTRY=? WHERE ADDR_ID=?";
```


6.   Save MySQLExit.java and close the file.

7.   Build DataServices project.

Lab 25.2   Associating an SQL-Based Data Service and Update Override

You must now set the update override class to the MySQLExit. This will let you get any updated changes and pass the new value.

Objectives

In this lab, you will:

Associate the update override class with the MySQLExit.

Confirm the settings in the Property Editor.

Instructions

1.   Open MySQL.ds in Design View. (The file is located in the DataServices\SQL folder.

2.   Click the MySQL Data Service header. The Property Editor opens.

3.   In the Property Editor, set the update override class by selecting MySQLExit from the DataServices\SQL\Java folder.

4.   Save the MySQL.ds file.

5.   Build your DataServices project.

Lab 25.3   Testing Updates

You are now ready to test whether the update override functions correctly.

Objectives

In this lab, you will:

Test the update override, by using the MySQL data service to make changes to the underlying relational data source.

View the results.

Instructions

1. Open MySQL.ds in Test View.

2. Select MySQL(x1) from the Function drop-down list, enter CA, and click Execute.

3. Click Edit.

4. Test if updates are getting propagated to the database, by completing the following steps:

   a. Select any Customer node.

   b. Modify City and Zip Code elements.

   c. Click Submit to issue the update override commit command and propagate changes to the database.

5. Select MySQL(x1) from the function drop-down list, enter CA, and click Execute to confirm that your database is updated.

Lesson Summary

In this lesson, you learned how to:

Create an update override for a physical data service.

Setup the update override to be a JDBC client and write logic to update the database table.

Lesson 26 Understanding Query Plans

A query plan contains detailed, functional-level information about an XQuery. Reviewing the Query Plan is the first step in troubleshooting a data service function's performance bottlenecks, since it lets you view the query's construction.

Objectives

After completing this lesson, you will be able to:

Examine a query plan in three different views: tree, XML, and text.

Locate the SQL statement created to retrieve data from the underlying database.

Locate XML elements.

Overview

The most common reason for viewing a query plan is to review the SQL statement generated by the DSP query engine. However, the query plan also displays the following information for the physical data sources to be called during the query:

Physical Data Source	Information Provided
Relational	Data source name, actual SQL calls, and join parameters.
Web Services	Data source name, operation(s) called, and join parameters.
Custom Functions	Function name and join parameters.
XML and Delimited Files	Filename.

In addition, the following information is displayed for all functions:

Number of invocations.

Order in which the data source calls are made.

Compilation time.

Areas where calls are made in parallel.

Areas where there are Cartesian joins.

Areas where join algorithms are used, including parameter passing and index joins.

Any calls to a middle-tier cache.

Lab 26.1   Viewing the Query Plan

A query plan is generated for each data service function, when a DSP project is built.

Objectives

In this lab, you will:

> Get the query plan for the getCustomerProfile() function.

> View the results in tree, XML, and text views.

Instructions

1. Open XQueries.ds in Query Plan View.

2. Select getTop10Customers() from the function drop down list.

3. Click show query plan. The query plan opens in tree view, as displayed in Figure 26-1.
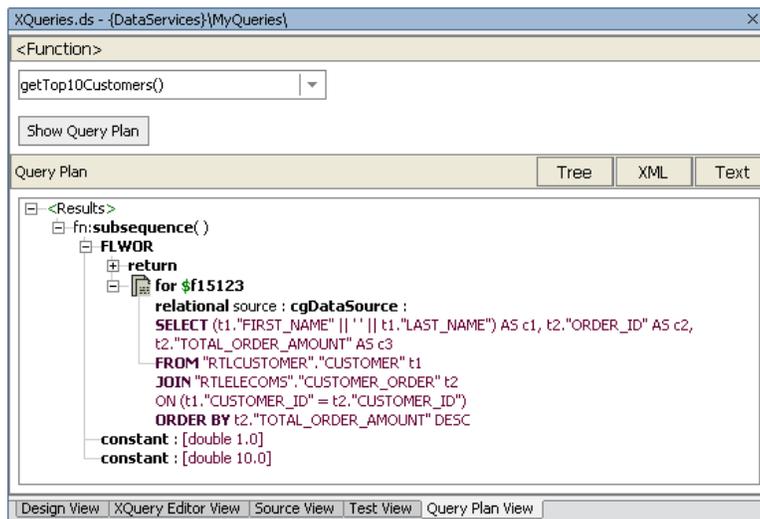
Figure 26-1   Query Plan as a Tree Structure

4. Click the XML button to view the Query Plan as an XML document.
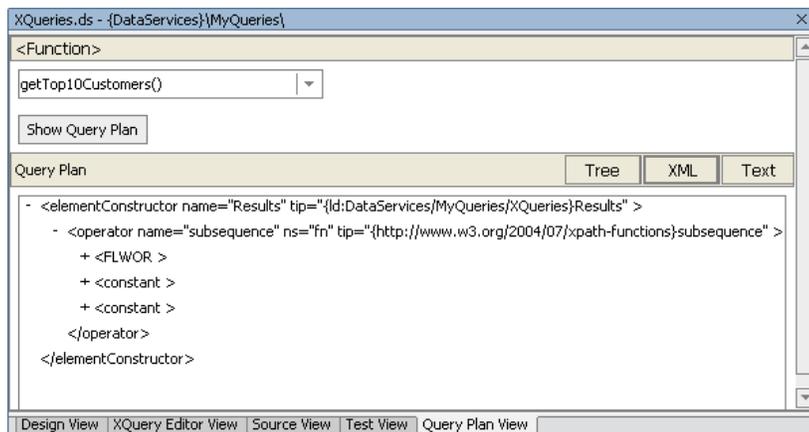
Figure 26-2   Query Plan as an XML Document

5. Click the Text button to view the Query Plan as a text document.

```
XQueries.ds - {DataServices}\MyQueries\                                          ✕
<Function>

getTop10Customers()                        ▼

Show Query Plan

Query Plan                                           Tree     XML     Text

<elementConstructor name="Results" tip="{ld:DataServices/MyQueries/XQueries}Results">
  <operator name="subsequence" ns="fn" tip="{http://www.w3.org/2004/07/xpath-functions}subsequence">
    <FLWOR>
      <return>
        <elementConstructor name="CUSTOMER">
          <elementConstructor field="c1" from="f15123" name="CUSTOMER_NAME">
          </elementConstructor>
          <elementConstructor field="c2" from="f15123" name="ORDER_ID">
          </elementConstructor>
          <elementConstructor field="c3" from="f15123" name="TOTAL_ORDERS">
          </elementConstructor>
        </elementConstructor>
      </return>
      <for name="f15123">
        <source kind="relational" name="cgDataSource" ns="fn-bea" tip="cgDataSource">
          [#cdata-section: SELECT (t1."FIRST_NAME" || ' ' || t1."LAST_NAME") AS c1, t2."ORDER_ID" AS c2,
   t2."TOTAL_ORDER_AMOUNT" AS c3
FROM "RTLCUSTOMER"."CUSTOMER" t1
JOIN "RTLELECOMS"."CUSTOMER_ORDER" t2
ON (t1."CUSTOMER_ID" = t2."CUSTOMER_ID")
ORDER BY t2."TOTAL_ORDER_AMOUNT" DESC]
        </source>
      </for>
    </FLWOR>
    <constant>
      [#cdata-section: [double 1.0]]
    </constant>
    <constant>
      [#cdata-section: [double 10.0]]
    </constant>
  </operator>
</elementConstructor>

Design View | XQuery Editor View | Source View | Test View | Query Plan View
```

Figure 26-3   Query Plan as a Text Document

Lab 26.2   Locating the SQL Statement in a Query Plan

SQL statements are generated for functions that call relational databases.

Objectives

In this lab, you will:

> Locate an SQL statement within the query.

> Review the contents of the SQL statement.

Instructions

1. Open the Query Plan as an XML document.

2. Expand the FLWOR nodes until you see the #cdata-section. This is the SQL statement for the query.
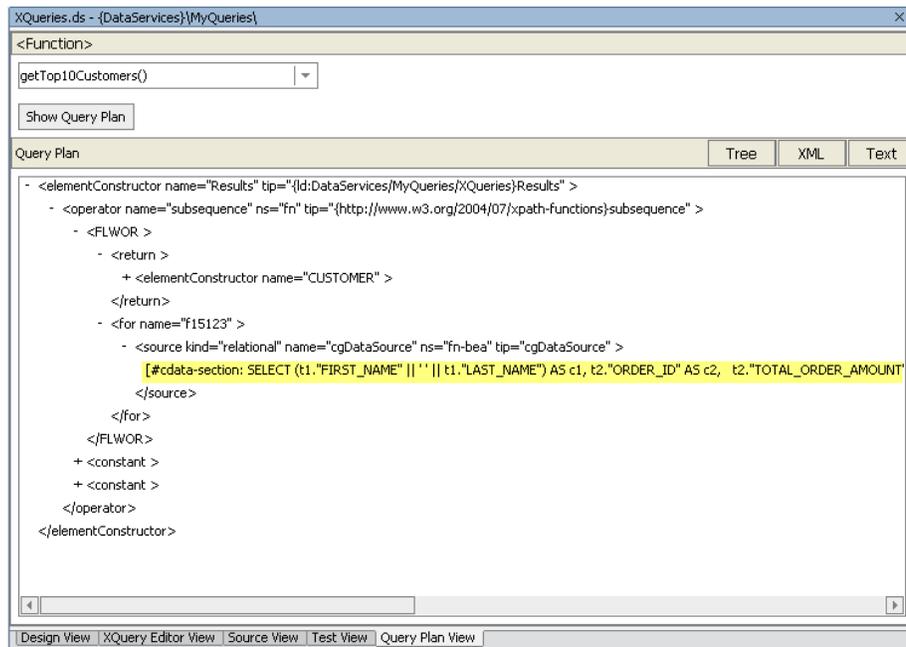
Figure 26-4   Query Plan View of SQL Statements

3. Scroll through the SQL statement. As a demonstration of DSP's push-down framework, notice that the SQL statement contains the "order by" clause. This means that the database will do the majority of work, since it is already optimized to do such sorting operations.

As a reminder, this function retrieves customer and order amount information. In addition, the result set is ordered in descending order by order amount.

Lab 26.3   Locating XML Elements

XML elements identify the data that will be returned by the query function. Each XML element is identified with a QName.

Objectives

In this lab, you will:

> Locate all XML elements within the query.
>
> Review the contents of the XML element lines.

Instructions

1. In Query Plan View, expand the return node.

2. Notice all the XML elements that will be returned when the function is executed.
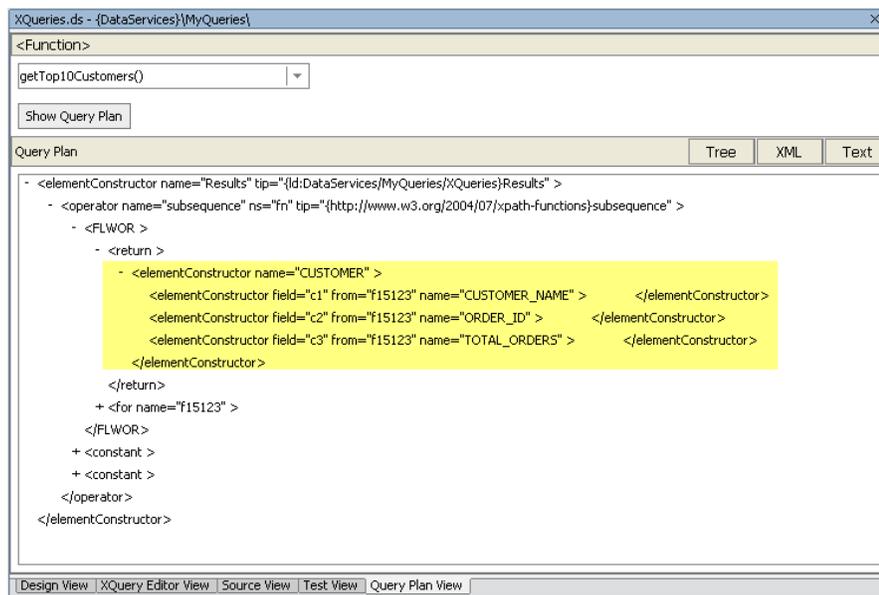
Figure 26-5   Query Plan View of XML Elements

Lesson Summary

In this lesson, you learned how to:

> Examine a query plan as tree, XML, and text documents.
>
> Locate the SQL statement that was created to retrieve data from the underlying database.
>
> Locate XML elements.

Lesson 27 Reusing XQuery Code through Vertical View Unfolding

DSP enables powerful data service code reusability.

Objectives

After completing this lesson, you will be able to:

> Re-use code.

> Unfold vertical file view.

Overview

DSP enables powerful data service code reusability. You can develop your logic once, and then re-use it later when building other data services. This feature is called *view unfolding*.

In addition to code reuse, DSP is smart enough to optimize your output and only query sources and elements that you request in your data service (*vertical view unfolding*).

Lab 27.1 Unfolding Vertical View

You will reuse the CustomerProfile data service previously built to retrieve Customer Order information. The CustomerProfile data service is built from three different tables in the underlying PointBase database: CUSTOMER, CUSTOMER_ORDER and CUSTOMER_ORDER_LINE_ITEM.

Objectives

In this lab, you will:

> Import the CustomerOrder data service into the CustomerManagement folder.

> Import the CustomerOrder.xsd, and then associate the schema with the CustomerOrder data service.

> Implement a query function, and define its conditions.

Instructions

1. Import CustomerOrder.ds into the DataServices\CustomerManagement folder. (The file is located in the <beahome>\weblogic81\samples\LiquidData\EvalGuide folder.)

2. Import CustomerOrder.xsd into the DataServices\CustomerManagement\schemas folder. (The file is also located in the <beahome>\weblogic81\samples\LiquidData\EvalGuide folder.)

3. Implement the getCustomerOrder() function in the CustomerOrder data service, by completing the following steps:

   a. Open CustomerOrder.ds in XQuery Editor View.

   b. In Data Services Palette, drag and drop getAllCustomers() into XQuery Editor View. (The method call is located in the folder:

      `DataServices\CustomerManagement\CustomerProfile`

4. Set the conditions for the function, by completing the following steps:

a. Select the Customer* element. This will activate the Expression Editor and make visible the ns2:getAllCustomers() expression. You will use the Expression Editor to scope the data returned in the getAllCustomers() function.
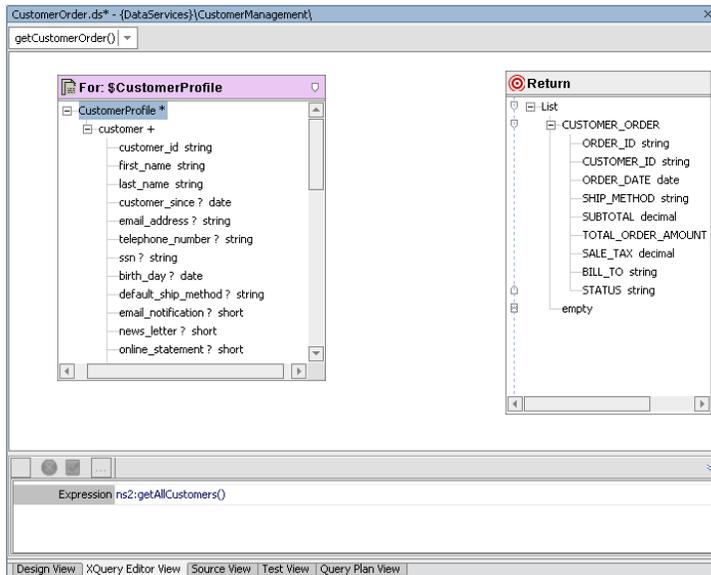
Figure 27-1   Default Expression

b. Triple-click the Expression field.

c. Modify the expression by adding the following code:

```
ns2:getAllCustomers()/customer/orders/order
```

d. Click the green checkmark icon to accept the changes. The CustomerProfile* element changes to the order* element, and the For:$CustomerProfile schema now includes the order elements.
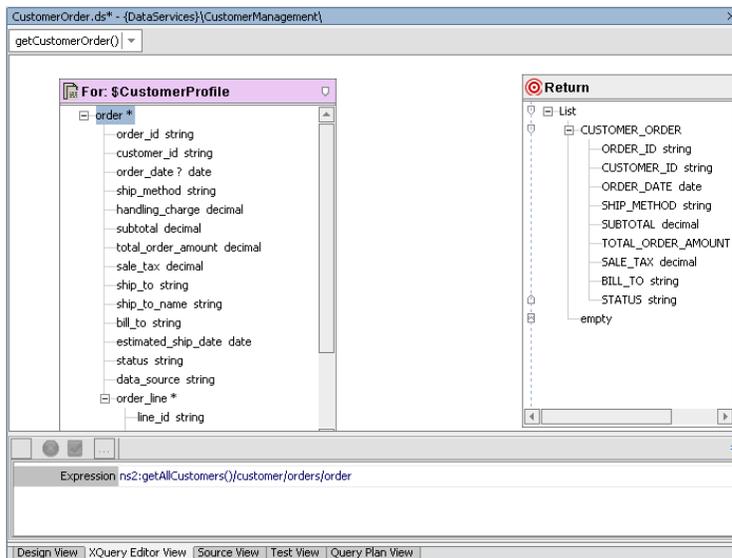
Figure 27-2   Modified Expression

5. Create a simple mapping: Drag and drop all order* elements (source node) to the corresponding CUSTOMER_ORDER elements in the Return type.
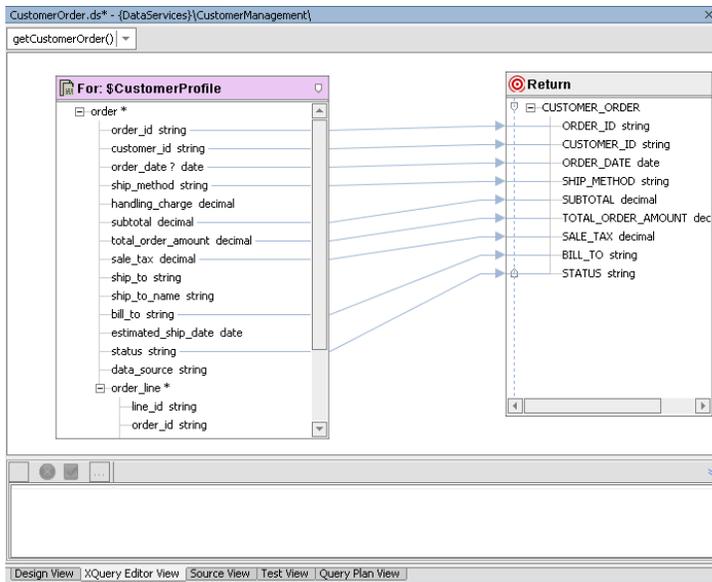
Figure 27-3   XQuery Editor View—Mappings

6.   Save the data service file.

7.   Open CustomerOrders.ds in Source View and notice that the function is using the CustomerProfile file as its data source.
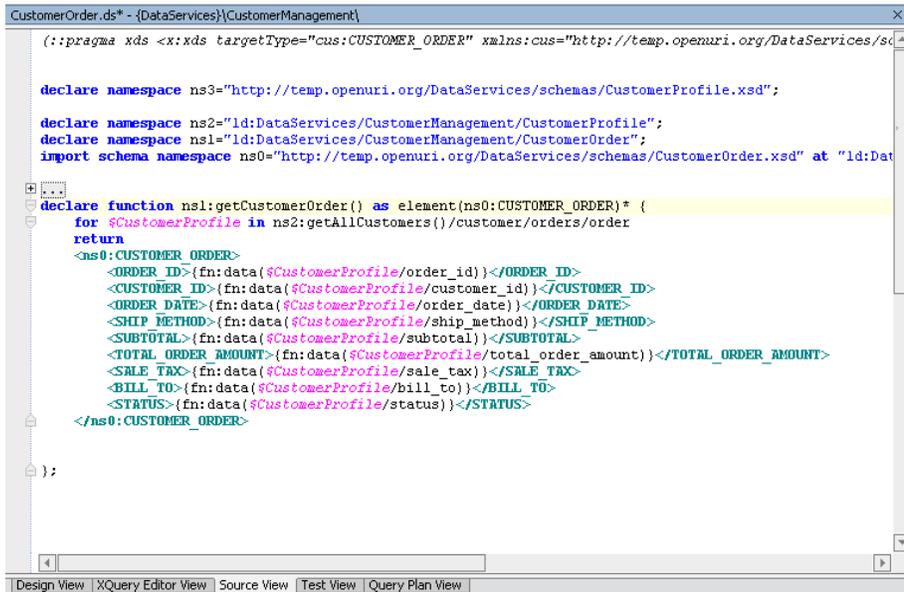
Figure 27-4   Source View of Vertical File Unfolding Function

Lab 27.2   Testing a Vertical File Unfolding

Testing a vertical file unfolding is similar to testing any other data service function.

Objectives

In this lab, you will:

> Test the CustomerOrder data service.

> Review the results.

Instructions

1. Open CustomerOrders.ds in Test View.

2. Select getCustomerOrder() from the function drop-down list.

3. Click Execute.

4. Confirm that you can retrieve customer order information.
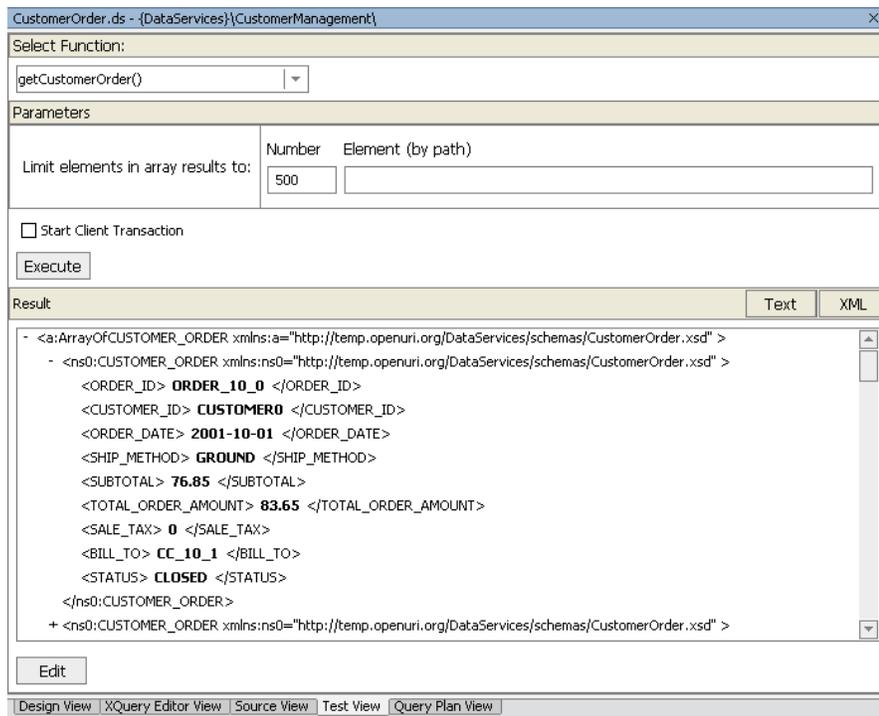
Figure 27-5   Vertical File Unfolding Test Results

Lesson Summary

In this lesson, you learned how to:

> Build a data service based on another data service (view unfolding)

> Re-use code (vertical file unfolding).

Lesson 28 Configuring Alternatives for Unavailable Data Sources

Sometimes a particular data source is either temporarily unavailable or very slow to send a response back to a consuming application. In such cases, you need to be able to run an alternative data source. DSP enables you create an alternative data source that will be called if the primary data source does not respond within a specified timeframe.

Objectives

After completing this lesson, you will be able to:

> Invoke, configure, and test an alternative data source.
>
> Use the fn-bea:timeout() function for configuring alternative sources.
>
> Review WebLogic Server output.

Overview

Enabling an alternative data source is implemented by calling the fn-bea:timeout() function. The syntax for the function is as follows:

```
fn-bea:timeout($seq as item()*, $millis as xs:int, $alt as item()*)
as item()*
```

where:

> $seq is the primary expression.
>
> $millis in the timeout in milliseconds.
>
> $alt is the alternate expression.

To implement this functionality, the return types of both the primary and alternative expression should be available when the project is compiled. This ensures that the function's return type is correctly inferred. In other words, the source metadata must be available at compile time, since the alternative source function provides only runtime failover capability.

Lab 28.1   Setting the Demonstration Conditions

You will import a slow web service into your application, thereby enabling the demonstration of configuring alternatives for unavailable data sources.

Objectives

In this lab, you will:

> Import and test a "slow" web for demonstration purposes.

> Create a physical data service that is based on an alternative data source.

Instructions

1. Right-click the Evaluation folder and then import the <beahome>\weblogic81\samples\LiquidData\EvalGuide\CreditWS file as a Web Service Project. This will import a simple web service that does nothing but sleep for 3 seconds. Click 'Yes' when asked for "Files required for Web Services are not in the project. Do you wish to add them?"

2. Build the CreditWS project.

3. Test the slow web service by completing the following steps:

   a. Open the NewCreditReport.jws, located in the CreditWS folder.

   b. Click the Start icon (or press Ctrl + F5). The Workshop test browser opens.

   c. Enter CUSTOMER3 in the cid field and click NewLookupCredit.

   d. Confirm that you can get credit rating information.
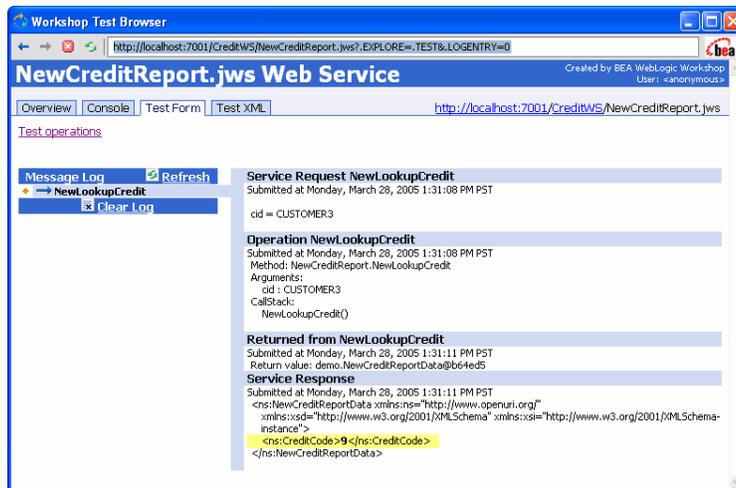
Figure 28-1   Test Browser View of the Slow Web Service

4. Create a physical data service for the slow web service, by completing the following steps:

   a. Select the Overview tab in the Workshop Test Browser.

   b. Click Complete WSDL.

   c. Copy the WSDL URI, which you will use to import an alternative data service. The URI typically is:

   ```
   http://localhost:7001/CreditWS/NewCreditReport.jws?WSDL=
   ```

---

d. In the Application pane of WebLogic Workshop, right-click the WebServices folder (located in DataServices).

e. Choose Import Source Metadata.

f. Select Web Service from the Data Source Type drop-down list and click Next.

g. Paste the WSDL URI into the URI field, then click Next.

h. Expand the folders and select the NewLookupCredit operation.

i. Click Add to populate the Selected Web Service Operations pane and click Next.

j. Review the Summary information and click Finish.

5. Check the Application pane. There should be a new physical data service called NewLookupCreditResponse.ds.

6. Open NewLookupCreditResponse.ds in Design View. There should be a function called NewLookupCredit.
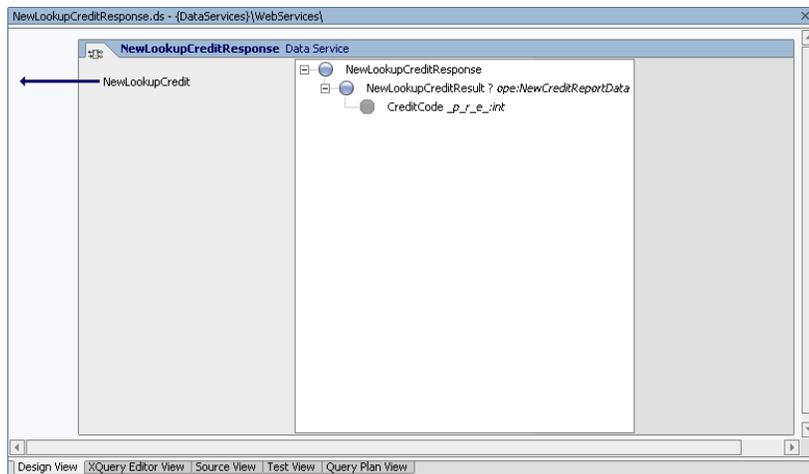
Figure 28-2   Design View of Web Service-Based Data Service

Lab 28.2 Configuring Alternative Sources

Since the CreditWS web service is slow, you need to configure an alternative source to obtain the credit rating information in a timely manner.

Objectives

In this lab, you will:

Configure an alternative data source.

Use the fn:bea:timeout() function.

Instructions

1. Open CustomerProfile.ds in Source View. (The file is located in the DataServices\CustomerManagement folder.

2. Add the following code to the namespace declaration:

```
declare namespace
ws3="ld:DataServices/WebServices/NewLookupCreditResponse";

declare namespace ws4 = "http://www.openuri.org/";
```

3. Locate the getAllCustomers() function.

4. Locate the following entry:

```
{
for $rating  in ws1:getCreditRating(
    <ws2:getCreditRating>
        <ws2:customer_id>{data($CUSTOMER/CUSTOMER_ID)}</ws2:customer_id>
    </ws2:getCreditRating> )
return
    <creditrating>
        <rating>{data($rating/ws2:getCreditRatingResult/ws2:Rating)}</rating>

    <customer_id>{data($rating/ws2:getCreditRatingResult/ws2:Customer_id)}</cu
    stomer_id>
     </creditrating>
}
```

5. Replace that entry with the following code:

```
{
<creditrating>
  <rating>
    {
    fn-bea:timeout(
        data(
            ws3:NewLookupCredit(
                <ws4:NewLookupCredit>
                    <ws4:cid>{data($CUSTOMER/CUSTOMER_ID)}</ws4:cid>
                </ws4:NewLookupCredit>
                )/ws4:NewLookupCreditResult/ws4:CreditCode
        )
        , 2000,
        data(
            ws1:getCreditRating(
                <ws2:getCreditRating>

  <ws2:customer_id>{data($CUSTOMER/CUSTOMER_ID)}</ws2:customer_id>
                </ws2:getCreditRating>
  )/ws2:getCreditRatingResult/ws2:Rating)
        )
    }
  </rating>
  <customer_id>{data($CUSTOMER/CUSTOMER_ID)}</customer_id>
</creditrating>
}
```

Note the use of the fn-bea:timeout() function.
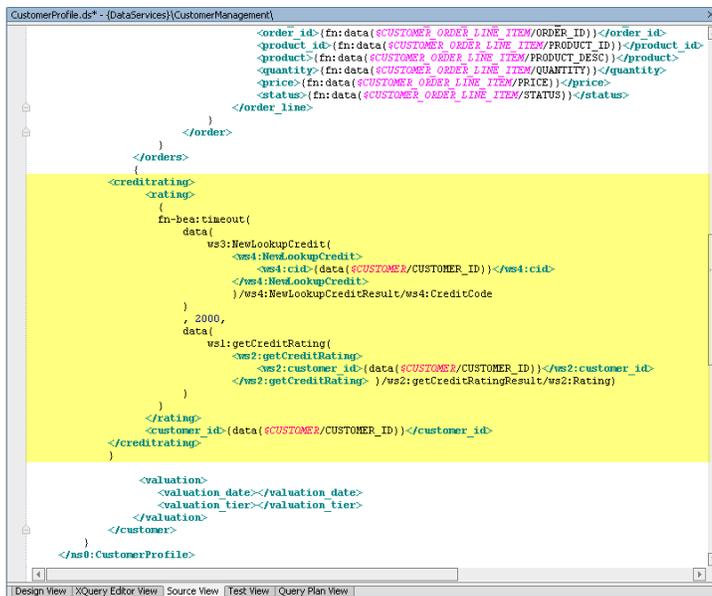
Figure 28-3   Source View of Configured Alternative

Lab 28.3   Testing an Alternative Source

Testing getAllCustomers function will let you confirm that the query is retrieving data from the alternative source, rather than the CreditWS.

Objectives

In this lab, you will:

Test the CustomerProfile data service, using the getAllCustomers function.

Review the results in the Output window.

Instructions

1. Build the DataServices project.

2. Open CustomerProfile.ds located in CustomerManagement folder in Test View.

3. Select getAllCustomers from the function drop-down list.

4. Click Execute.

5. Open the Output window, scroll to the bottom, and then confirm that the following is happening for each customer profile:

WebLogic Server calls the CreditWS web service.

The server waits 155501 ms, and then calls the CreditRating web service.


```
Output                                                                                    ×

  DataSource name: ld:EvaluationDataServices/WebServices/NewLookupCreditResponse  Invocations: 0  Time: 0ms
  Statement: NewLookupCredit

  DataSource name: ld:EvaluationDataServices/WebServices/getCreditRatingResponse  Invocations: 10  Time: 2057ms
  Statement: getCreditRating
```
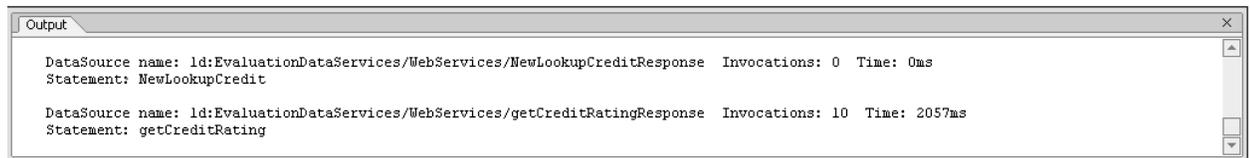
Figure 28-4   Output Window

The invocation of the first Web service NewLookupCreditResponse fails due to the thread itself timing out. Since this Web service has failed it will not be invoked again. Instead, the alternate web service is invoked.

Lesson Summary

In this lesson, you learned how to:

Invoke, configure, and test an alternative data source.

Use the fn-bea:timeout() function for configuring alternative sources.

Review WebLogic Server output.

Lesson 29 Enabling Fine-Grained Caching

Fine-grained caching lets you cache a data subset, such as information that does not frequently change. Fine-grained caching is at the function level, since a function's role is to retrieve specific information.

Objectives

After completing this lesson, you will be able to:

Define a cache policy for the slow credit rating web service.

Testing caching performance.

Overview

DSP provides a flexible caching mechanism to manage caching of data service functions. In Part 1, you learned how to cache a function in a logical data service. However, there are situations where you may want to cache only a sub-set of information available in a particular logical data service. For example, the CustomerProfile data service includes information about each customer's profile and order information. The profile information does not change often, whereas order information constantly changes. In this situation users would like to cache the profile information for a given customer but retrieve the most recent order information from the operational system.

By defining different caching policies for the underlying customer and order physical data services, you can cache only the CUSTOMER physical data service. As a result, any request made to the logical CustomerProfile data service will be partly answered from the DSP Cache for customer information and partly answered from the operational system for order information.

Lab 29.1 Enabling Function-Level Caching for a Physical Data Service

Caching of a function in an underlying data service provides you with the ability to cache a sub-set of data within a data service function.

Objectives

In this lab, you will:

Enable application-level caching and function-level caching.

Instructions

1. Login to the DSP Console (http://localhost:7001/ldconsole/), using the following credentials:
   - o User Name = weblogic
   - o Password = weblogic
2. Using the + icon, expand the ldplatform directory. (Note: If you click the ldplatform name, the Application List page opens. This is *not* the page you want for this lesson.)
3. Click Evaluation. The Administration Control's General page opens.
4. In the Cache section, select Enable Cache.
5. Select cgDataSource from the Cache data source name drop-down list.
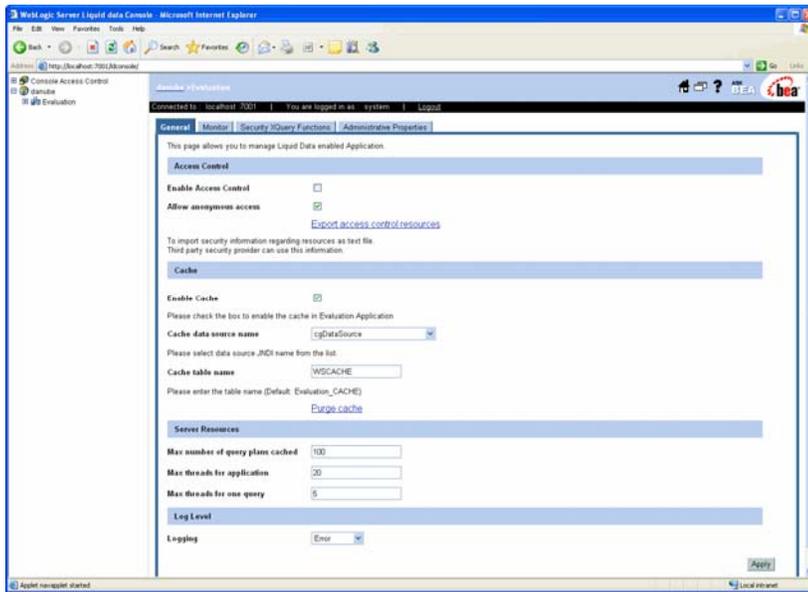6. Enter WSCACHE in the Cache table name field.

---

7. Click Apply.

Figure 29-1 Enable Caching

8. Expand the Evaluation folder and navigate to the getCreditRatingResponse data service, located in the DataServices\WebServices\ folder.

9. For the getCreditRating() function, set a caching policy by completing the following steps:

   a. Select Enable Cache.

   b. Enter 300 in the TTL field.

   c. Click Apply.
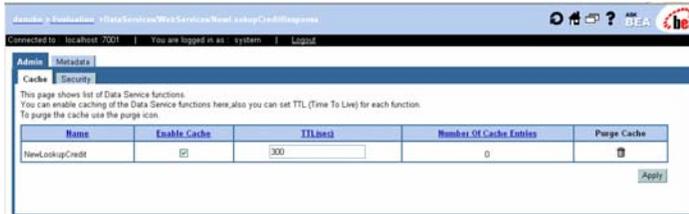
Figure 29-2   Enable Function-Level Caching

Lab 29.2   Testing the Caching Policy

You are now ready to test your new fine-grained caching policy.

Objectives

In this lab, you will:

> Test the function-level caching policy.

> Determine whether the cache was populated.

Instructions

1. In WebLogic Workshop, execute a test query by completing the following steps:

   a. Open CustomerProfile.ds in Test View. (The file is located in the CustomerManagement folder.)

   b. Select getCustomerProfile(CustomerID) from the function drop-down list.

   c. Enter CUSTOMER3 in the Parameter field.

   d. Click Execute.

   e. In the Output window, note the number of invocations and the times for the NewLookupCreditResponse and getCreditRatingResponse data sources.

2. In the PointBase Console, check whether the cache database table was populated by completing the following steps:

   a. Start the PointBase Console, using the following command in a command prompt window:

      `<beahome>\weblogic81\common\bin\startPointBaseConsole.cmd`

   b. Use the following configuration to connect to your local PointBase database:

      o Driver: com.pointbase.jdbc.jdbcUniversalDriver

      o URL: jdbc:pointbase:server://localhost:9093/workshop

      o User: weblogic

      o Password: weblogic

   c. Click OK.

d.   Enter the SQL command: SELECT * FROM WSCACHE

e.   Click Execute to check whether the cache was populated.
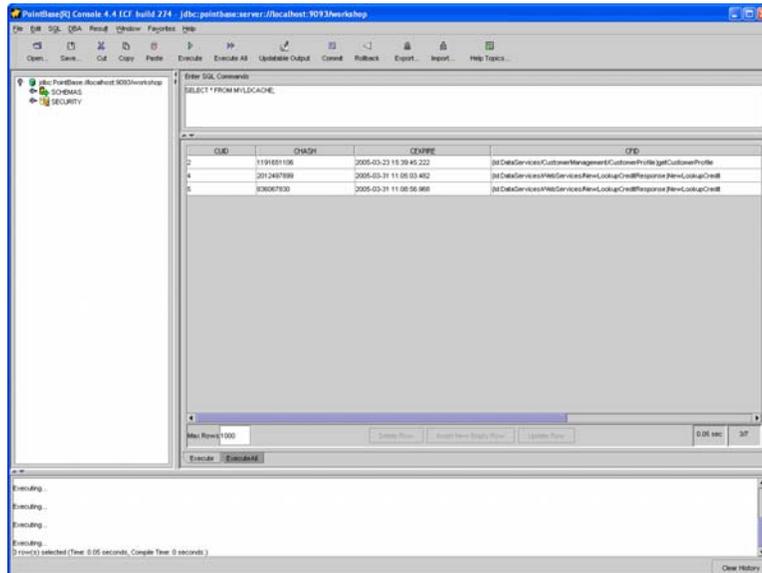
Figure 29-3   PointBase Console Cache Information

Lab 29.3   Testing Performance Impact

The next step is to determine whether the caching policy improves query performance.

Objectives

In this lab, you will:

Execute a data service test.

Determine whether the caching policy improved query performance time.

Instructions

1.   In WebLogic Workshop, execute a test query by completing the following steps:

a.   Open CustomerProfile.ds in Test View. (The file is located in the CustomerManagement folder.)

b.   Select getCustomerProfile() from the function drop-down list.

c.   Enter CUSTOMER3 in the Parameter field.

d.   Click Execute.

2.   Confirm the following performance results in the Output window:

a.   Confirm that the slow web service (NewLookupCreditRatingResponse) was never invoked due to alternate path execution.

b.   Determine whether caching the Web service helped to reduce the query execution time.

Lesson Summary

In this lesson, you learned how to:

Enable the cache for a physical data service function and define the cache's TTL.

Determine the performance impact of the physical data service cache on a function in a logical data service by checking the query response time and whether the physical data service (original data source) was invoked.

Lesson 30 Creating XQuery Filters to Implement Conditional-Logic Security

Data Services Platform can enable security based on the results of conditional logic.

Objectives

After completing this lesson, you will be able to:

Activate security XQuery functions.

Write security XQuery functions.

Overview

Conditional logic can be used to establish very specific security restrictions. For example, users in an employee role can see only orders less than $1,000, while users in a manager role can see all orders, regardless of total amount. This feature is implemented through the DSP Console.

Lab 30.2   Creating User Groups

The first step in setting conditional-logic security is establishing security groups.

Objectives

In this lab, you will:

Create new user groups.

Assign user accounts to user groups.

Instructions

1. Login to the WebLogic Server Console (http://localhost:7001/console/), using the following credentials:

    User Name = weblogic

    Password = weblogic

2. Create two new user groups by completing the following steps:

    a.   Choose Security → Realms → myrealm → Groups.

    b.   Select Configure a New Group.

    c.   Enter LD_Emp in the Name field.

    d.   (*Optional*) Enter "Employee Group" in the Description field.

    e.   Click Apply.

    f.   Repeat steps 2b through 2e to create a new group for LD_Mgr.
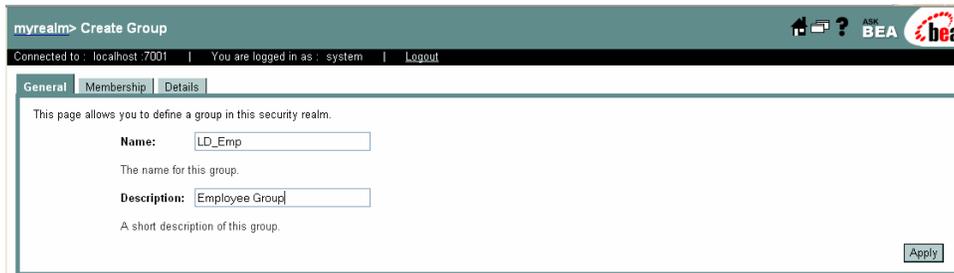
Figure 30-1   Configuring a New User Group

3. Create a new user in the LD_Emp group, by completing the following steps:

    a. Choose Security → Realms → myrealm → Users.

    b. Click Bob in the User column. The User page for Bob opens.

Figure 30-2   User Page for Bob

    c. Click the Groups tab. The Groups page opens.

    d. Select LD_Emp from the Possible Groups pane.

    e. Click the arrow (→) to add the group to the Current Groups pane.
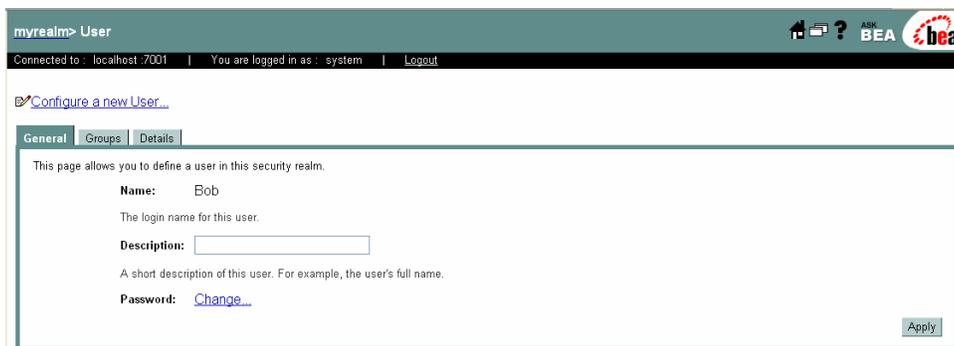
    f. Click Apply.
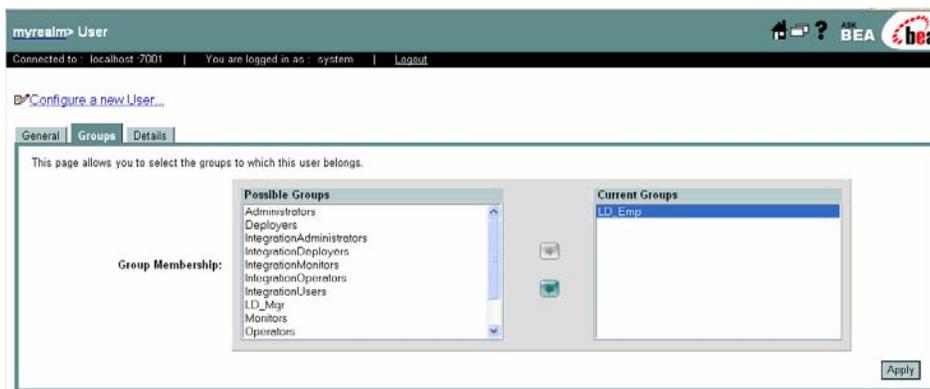
Figure 30-3   Group Assignment Page for Bob

4. Create a new user in the LD_Mgr group, by completing the following steps:

    a. Choose Security → Realms → myrealm → users.

    b. Click Joe in the User column. The User page for Joe opens.

    c. Click the Groups tab. The Groups page opens.

    d. Select LD_Mgr from the Possible Groups pane.

    e. Click the arrow (→) to add the group to the Current Groups pane.

    f. Click Apply.

Lab 30.3 Writing the XQuery Security Function

You can specify a security function using XQuery syntax.

Objectives

In this lab, you will:

> Set security access control.

> Set a security XQuery function.

Instructions

1. Login to the DSP Console (`http://localhost:7001/ldconsole/`), using the following credentials:

    A. User Name = weblogic

    B. Password = weblogic

2. Click the Security tab.

3. Using the plus (+) icon, expand the ldplatform directory. (<u>Note:</u> If you click the ldplatform name, the Application List page opens. You do *not* want this page for this lesson.)

4. Click Evaluation. The Administration Control's General page opens.

5. Select Check Access Control.

6. Select Allow Default Anonymous Access.
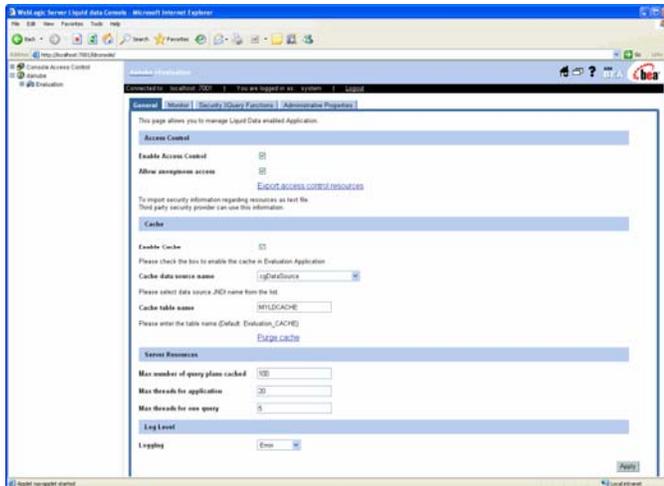
Figure 30-4   Setting Access Control

7. Select Security Xquery Functions and enter the following function.

    Note: Namespaces may be different for your application.

    ```
    declare namespace demo="lib:mydemo";

    declare namespace
    itemns="http://temp.openuri.org/DataServices/schemas/CustomerProfile.xsd" ;

    declare function demo:secureCustomer($ssn as xs:string) as xs:boolean {

    if (fn-bea:is-user-in-group("LD_Mgr")) then fn:true()

    else fn:false()

    };
    ```

8. Click Apply. You should now have the following:
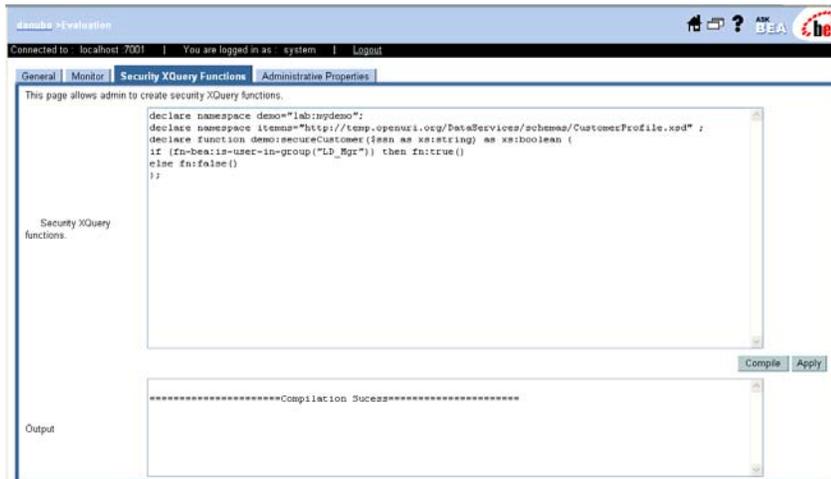
Figure 30-5   Specifying Security XQuery Function Code

Lab 30.4   Activating the XQuery Security Function

The next step in setting an XQuery security function is to set security at the element level.

Objectives

In this lab, you will:

Secure data source elements.

Set a security policy.

Instructions

1. In the DSP Console expand the Evaluation folder and navigate to the CustomerProfile data service, located in the DataServices\CustomerManagement folder.

2. Click Security Policy.

3. Click the icon in the Security XQuery function column for the CustomerProfile/customer/ssn resource. The QName window opens.
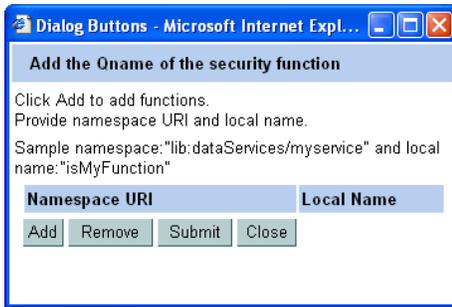
Figure 30-6   QName Window

4. Set the Namespace URI and Local Name, by completing the following steps:

   a. Click Add and enter the following values:

      o   Namespace URI: lib:mydemo;

      o   Local Name: secureCustomer

   b. Click Submit.

   c. Click Close.
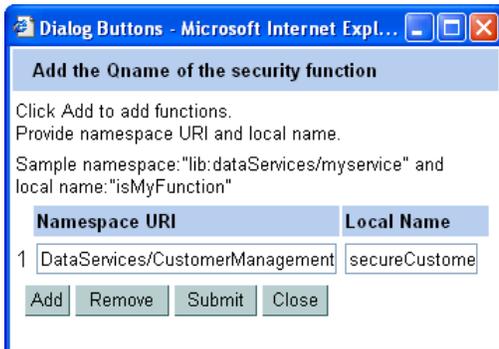
Figure 30-7   QName Information

Lab 30.5 Testing the XQuery Security Function

Using the security credentials for Bob and Joe, you can now test the XQuery security function.

Objectives

In this lab, you will:

> Test access control, using two different user logins.
>
> View the results.

Instructions

1. Set the login properties to Bob and run a test, by completing the following steps:

    a. In the DSP-enabled Workshop application, choose Tools → Application Properties → WebLogic Server.

    b. Select Use Credentials Below.

    c. Enter "Bob" and "password" in the Use Credentials Below fields.

    d. Click OK.

    e. Open CustomerProfile.ds in Test View. (The file is located in the CustomerManagement folder.)

    f. Select getAllCustomers from the function drop-down list.

    g. Click Execute. All customer data, *except SSNs*, should be returned.

    Note: In order to deploy from WorkShop User/Group you should have permission to deploy applications.

2. Change the login properties to Joe and run a test. All customer data, including SSNs, should be returned.

3. In the DSP Console expand the Evaluation folder and navigate to the CustomerProfile data service, located in the DataServices\CustomerManagement folder.

4. Click Security Policy.

5. Click the icon in the Security XQuery function column for the CustomerProfile/customer/ssn resource. The QName window opens.

6. Click Remove, click Submit, and then click Close to remove the following:

    > Namespace URI: lib:mydemo;
    >
    > Local Name: secureCustomer

Important: You must remove the Namespace/Local Name information before you can proceed with the following lessons.

Lesson Summary

In this lesson, you learned how to:

> Establish security based on XQuery functions.
>
> Write security XQuery functions.

Lesson 31 Accessing Data in Stored Procedures

Enterprise databases utilize stored procedures to improve query performance, manage and schedule data operations, enhance security, and so forth. 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.

You can import stored procedure metadata from any relational data available to the BEA WebLogic Server. DSP then uses that metadata to generate a physical data service that you can then use in logical data services.

Objectives

After completing this lesson, you will be able to:

> Import stored procedures as a Java project within an application.
>
> Import stored procedure metadata into a data service.

Overview

Imported stored procedure metadata is quite similar to imported metadata for relational tables and views. Stored procedure metadata generally contains:

> A data service file with a pragma that describe the parameters of the stored procedure.
>
> A schema file with the same primary name as the procedure name.

Note: If a stored procedure includes only one return value and the value is either simple type or a row set that is mapping to an existing schema, no schema file is created.

Handling Stored Procedure Row Sets

A row set type is a complex type, whose name can include:

> The *parameter name*, if there is an input/output or output only parameter.
>
> An *assigned name* such as RETURN_VALUE, if there is a return value.
>
> The *referenced element name* (result rowsets) in a user-specified schema.

The row set type contains a repeatable element sequence (for example, called CUSTOMER) with the fields of the row set.

Notes:

> All row set-type definitions must conform to the structure in the stored procedure itself. In some cases the Metadata Import Wizard will be able to automatically detect the structure of a row set and create an element structure. However, if the structure cannot be determined, you will need to provide it through the wizard.
>
> Each database vendor approaches stored procedures differently. Refer to your database documentation for details on managing stored procedures.
>
> XQuery support limitations are, in general, due to JDBC driver limitations.

DSP does not support rowset as an input parameter.

Lab 31.1   Importing a Stored Procedure into the Application

The first step in demonstrating DSP's ability to access data through a stored procedure is to import the procedure into the application.

Objectives

In this lab, you will:

Import stored procedures as a Java project.

Test the results.

Instructions

1. Import storedprocs as a Java project, adding it to the Evaluation application. (The project is located in the <beahome>\weblogic81\samples\LiquidData\EvalGuide folder.)

2. Build the storedprocs project. The storedprocs.jar file will be added to the Libraries folder.

3. Shutdown the PointBase database, by stopping WebLogic Server. (Stopping WebLogic Server calls the PointBase shutdown script.)

4. Open the startPointBase.cmd in a text editor such as Notepad. The file is located in <*BEAHOME*>\weblogic81\common\bin.

5. Add the complete path of the storedprocs.jar file to the startPointBase.cmd script. For example, add the line:

```
set CLASSPATH=<beahome>\user_porjects\applications\Evaluation\APP-
INF\lib\storedprocs.jar;%CLASSPATH%
```

Notes:

For reference, the modified startPointBase.cmd is included in the samples\liquiddata folder.

The CLASSPATH depends on your WebLogic Server installation. User can copy correct path from the Output window of Workshop.

6. Start WebLogic Server, which in turn starts the PointBase database.

7. Run CreditRatingStoredProcedure.java to define the stored procedures in PointBase.

8. Click OK at the pop-up message.

9. Confirm that the stored procedure executed, by reviewing the contents in the Output window. You should see the credit rating for CUSTOMER3.

Note: Your credit rating may be different, based on the changes that you made in Lab 24.3.
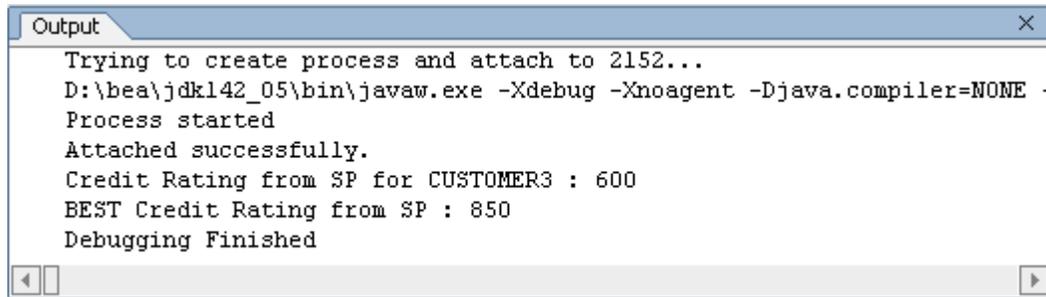
```
Output                                                                    ×
    Trying to create process and attach to 2152...
    D:\bea\jdk142_05\bin\javaw.exe -Xdebug -Xnoagent -Djava.compiler=NONE -
    Process started
    Attached successfully.
    Credit Rating from SP for CUSTOMER3 : 600
    BEST Credit Rating from SP : 850
    Debugging Finished
```

Figure 31-1   Output Window View of Stored Procedures Compilation

Lab 31.2   Importing Stored Procedure Metadata into a Data Service

Importing a stored procedure's source metadata enables the generation of a stored procedure data service.

Objectives

In this lab, you will:

> Import source metadata into a new data service.
>
> Test the stored procedure data service.

Instructions

1. Create a new folder in the DataServices project and name it StoredProcedures.

2. Import stored procedures metadata, by completing the following steps:

   a. Right-click the StoredProcedures folder.

   b. Choose Import Source Metadata.

   c. Select Relational from the Data Source Type drop-down list, then click Next.

   d. Select cgDataSource from the Data Source drop-down list, then click Next.

   e. Expand the WEBLOGIC\Procedures folders.

   f. Select GETCREDITRATING_SP, click Add, and click Next.

   g. Accept the default settings displayed in the Configure Procedure window, then click Next.

   h. Accept the default settings displayed in the Summary window and click Finish.

3. Build the DataServices project.

4. In the Application pane, confirm that there is a new data service, GETCREDITRATING_SP.ds, located in the StoredProcedures folder.

5. Test the data service, by completing the following steps:

   a. Open GETCREDITRATING_SP.ds in Test View.

   b. Select GETCREDITRATING_SP(x1) from the Function drop-down list.

   c. Enter CUSTOMER3 in the Parameter field.

   d. Click Execute. You should see the credit rating for Customer3

6. Review the results.

Lesson Summary

In this lesson, you learned how to:

> Import stored procedures into an application.

> Import stored procedure source metadata into a data service.

Lesson 32 Accessing Data with Java Functions

A Java function is another form of metadata that DSP can use as a data source. This is perhaps the most powerful one, since this allows DSP to basically utilize any data source that can be accessed from Java, such as Enterprise Java Beans, JMS/messaging applications, LDAP and other directory services, text/binary files that can be read through Java I/O, and even DCOM-based applications like Microsoft Excel.

In this lesson, you will access three data sources through Java functions:

WebLogic's embedded LDAP, by importing a Directory Service Markup Language (DSML)-based Java application as a Java function.

Data in a Microsoft Excel spreadsheet, by importing a Java application that uses JCOM to access the MS Excel spreadsheet.

An Enterprise Java Bean, that returns customer credit card information using a Java function.

Objectives

After completing this lesson, you will be able to:

Write Java functions and access them from data services.

Overview

When you use DSP's Import Source Metadata feature to import user-defined Java functions, the functions are introspected to create the necessary method signatures and parameter metadata. At the same time, a prologue is created that defines the function's signatures and relevant schema type for complex elements such as Java classes and arrays.

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

Java primitive types and single-dimension arrays, such as Boolean, byte, and char.

XMLBean classes corresponding to global elements, complex types, and arrays. The classes generated by XMLBeans can be used as parameters or Return types. The advantage of using XMLBean-generated classes is that you do not need to define a schema for the references complex type or element.

The Metadata Import Wizard supports marshalling and unmarshalling that converts Java token iterators into XML, and vice versa. For example, you start with a Java function, getListGivenMixed, defined as follows:

```
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 Metadata Import Wizard, the following XML-based metadata is generated:

```
(::pragma xds <x:xds xmlns:x="urn:annotations.ld.bea.com"

targetType="t:float" xmlns:t="http://www.w3.org/2001/XMLSchema">

<javaFunction

classpath="D:\jf\build\jar\jfTest.jar;D:\jf\xbeanTests\xbeangen\

Customer.jar;D:\wls82\weblogic81\server\lib\xbean.jar"

class="jfTest.Customer"/>

</x:xds>::)

declare namespace f1 = "ld:javaFunc/float";

(::pragma function <f:function xmlns:f="urn:annotations.ld.bea.com"

kind="datasource" access="public">

<params>

<param nativeType="[F"/>

<param nativeType="int"/>

</params>

</f:function>::)

declare function f1:getListGivenMixed($x1 as xsd:float*, $x2 as
xsd:int)

as xsd:float* external;
```

The corresponding XQuery for the imported Java function would be as follows:

```
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
```

Note: To ensure successful importation and usage within DSP, the Java function should be static functions and its package and class names should be defined in its namespace. DSP recognizes the Java method name as the XQuery function name qualified with the Java function namespace.

For detailed information about using Java functions within DSP see the *Data Services Developer's Guide*.

Lab 32.1   Accessing Data Using WebLogic's Embedded LDAP Function

DSP enables access to data services, using WebLogic's embedded LDAP function. You will learn how to use this functionality by importing a Directory Service Markup Language (DSML)-based Java application as a Java function.

Objectives

In this lab, you will:

> Set the LDAP security credential for WebLogic's Embedded LDAP.
>
> Create a new user account.
>
> Import JAR files and Java applications that will be used to generate a data service.
>
> Test the data service.

Instructions

1. In the DataServices project, create a folder and name it Functions. This is where you will place the Java functions that you want to import.

2. Set the LDAP security credential for WebLogic's Embedded LDAP, by completing the following steps:

    a. Open the WebLogic Server Console from your browser:

        http://localhost:7001/console.

    b. Login using the following credentials:

        > User Name = weblogic
        >
        > Password = weblogic

    c. Select the Security folder, located under the ldplatform domain.

    d. Select Embedded LDAP.

    e. Enter security in the Credential and Confirm Credential fields.

    f. Click Apply. This allows access to the WebLogic Server LDAP.
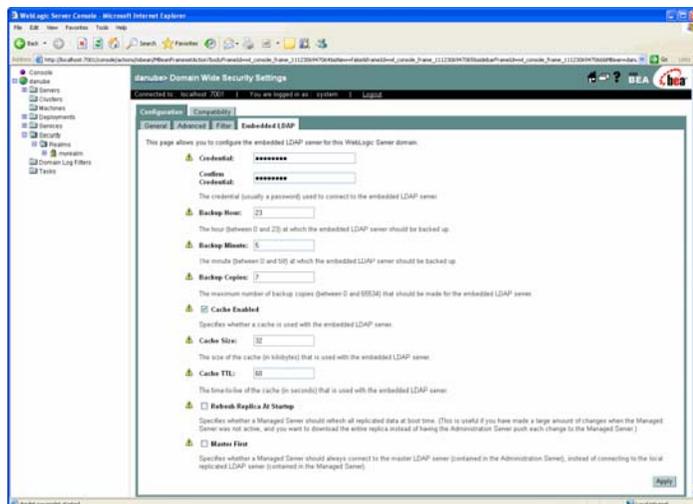
Figure 32-1   Setting LDAP Access Credentials

3. You will need to restart the WebLogic Server now as change to this property does not take effect until the Server is restarted.

4. Create a new user, by completing the following steps:

    a. Expand the Security → Realms → myrealm → Users folders.

    b. Click Configure a New User, using your name and a password of your choice.

    c. Click Apply.

5. In WebLogic Workshop right-click the Libraries folder and import all the JAR files located in the samples\liqiddata\EvalGuide\ldap\lib folder into the Libraries folder in Workshop.

6. Right-click the Functions folder and import DSML.java from the samples\liquiddata folder.

7. Build the DataServices project.

8. Import the Java function metadata for the DSML Java application into the Functions folder by completing the following steps:

    a. Right click the functions project and choose Import Source Metadata.

    b. Select Java Function for the Data Source Type and click Next.

    c. Enter Functions.DSML in the Class Name field and click Next.

    d. Select the callDSML function, click Add, and then click Next.

    e. Accept the default settings in the Summary window and click Finish.

    The dsml.ds file and schemas folder are added to the Functions folder.

9. Build the DataServices project.

10. Test the DSML data service by completing the following steps:

    a. Open dsml.ds in Test View.

    b. Select callDSML() from the Function drop-down list.

    c. Enter the following arguments (for more information on LDAP arguments and access, see http://dev2dev.bea.com/codelibrary/code/ld_ldap.jsp):

Description	Argument
LDAP URL	`ldap://localhost:7001`
Principal (Directory Manager)	`cn=Admin`
Credentials (Password)	`security`
JNDI (true: use JNDI to access LDAP; false: use native LDAP connection	`jndi`
Base domain name to search	`dc=ldplatform`
Filter used to search	`cn=<your user name>`

    d. Click Execute.

    e. View the results.

Lab 32.2   Accessing Excel Spreadsheet Data Using JCOM

Data in a Microsoft Excel spreadsheet can be accessed through JCOM.

Objectives

In this lab, you will:

> Import JAR and Java files appropriate that will be used to generate a data service for using JCOM.
>
> Test the results.

Instructions

1. Right-click the Libraries folder and import all the JAR files located in the samples\liqiddata\EvalGuide\excel\lib folder.

2. Right-click the Functions folder and import excel_jcom.java from the <beahome>\weblogic81\samples\LiquidData\EvalGuide folder.

3. Build the DataServices project.

4. Import the Java function metadata for the Excel JCOM Java application into the Functions folder, by completing the following steps:

   a. Right click the Functions project and choose Import Source Metadata.

   b. Select Java Function for the Data Source Type and click Next.

   c. Enter Functions.excel_jcom in the Class Name field and click Next.

   d. Select the getExcel function, click Add, and then click Next.

   e. Accept the default settings in the Summary window and click Finish. The excel.ds and associated schema files are added to the Functions folder.

5. Build the DataServices project.

6. Test the Excel data service, by completing the following steps:

   a. Open excel.ds in Test View.

   b. Enter the following arguments:

Description	Argument
XLS File Name	`<beahome>\weblogic81\samples\LiquidData\EvalGuide\excel\ test.xls`
Worksheet Name	`Customers`

> Review the results.

For more information on Excel access:
http://dev2dev.bea.com/codelibrary/code/liquiddata_Excel.jsp

Lab 32.3   (*Optional*) Accessing Data Using an Enterprise Java Bean

Create an Enterprise Java Bean that returns customer credit card information using a Java function.

Objectives

In this lab, you will:

> Import the schemas needed to define an EJB-based data service.
>
> Generate an EJB-based data service.
>
> Test the results.

Instructions

1. Create a Schemas Project, by completing the following steps:

    a. Right-click the Evaluation application folder and import the Schemas folder as a Schema Project. (The folder is located in the <beahome>\weblogic81\samples\LiquidData\EvalGuide\ejb folder.)

       This schema will be used for the EJB results, which returns an XML document containing credit card information for a customer.

    b. Build the Schemas project.

2. Create an EJB Project, by completing the following steps:

    a. Right-click the Evaluation application folder and import the EJB folder as an EJB Project. (The folder is located in the <beahome>\weblogic81\samples\LiquidData\EvalGuide\ejb folder.) This contains:

       > a container-managed entity bean that maps to the credit card database table and
       >
       > a stateless session bean that invokes the entity bean finder method returning a list of credit cards for a given customer in the shape of the CREDIT_CARDS XML schema.

    b. Build the EJB project.

3. Create a Java project, by completing the following steps:

    a. Right-click the Evaluation application folder and import the EJBClient folder as a Java Project. (The folder is located in <beahome>\weblogic81\samples\LiquidData\EvalGuide\ejb folder. This project contains the Java client that connects remotely to the stateless session bean. This will be used as the custom function.

    b. Build the EJBClient project.

4. Run CreditCardClient.java, which is located in the EJBClient project folder. (<u>Note:</u> Click OK for the pop-up message.)

    A list of credit cards for CUSTOMER3 should display in the Output window.

5. Drag and drop the CreditCardClient.java into the Functions folder.

6. Build the DataServices project.

7. Import the Java function metadata for the EJB Client into the DataServices project by completing the following steps.

    a. Right-click the Functions folder and select Import Source Metadata.

    b. Select Java Function as the Data Source Type and click Next.

    c. Enter Functions.CreditCardClient as the Class Name and click Next.

d. Select getCreditCards, click Add, and then click Next.

e. Accept the default settings in the Summary window and click Finish. The CREDIT_CARDS.ds file is added to the Functions folder.

(Note: Do not confuse this data service with the CREDIT_CARD.ds created from the relationship database.)

f. Build the DataServices project.

8. Test the getcreditCard() function within the CREDIT_CARDS data service. Use CUSTOMER3 as the argument. Confirm that you can retrieve credit card information for Britt Pierce.

Lesson Summary

In this lesson, you learned how to import the following sources as Java functions:

WebLogic's embedded LDAP through a Directory Service Markup Language (DSML)-based Java application

Data in a Microsoft Excel spreadsheet through a Java application that uses JCOM to access the MS Excel spreadsheet.

An Enterprise Java Bean that returns customer credit card information.

Lesson 33 Accessing Data in XML Files

XML documents are a convenient means for handling hierarchical data. DSP enables the creation of data services that read data stored in XML files.

Objectives

After completing this lesson, you will be able to:

Import XML metadata and query XML files.

Confirm that the results conform to the XML file specifications.

Overview

Contents of an XML file can be turned into a data service and used as a data source.

In this lab you will create a data service that queries data stored in an XML file. The XML file contains UNSPSC product category received from third-party vendor.

Lab 33.1   Importing XML Metadata and XML Schema Definition

Importing XML metadata ad schema definitions is similar to importing relational and web service metadata, with some differences.

Objectives

In this lab, you will:

Import XML metadata.

Associate a schema and XML source file with the data service.

Generate a data service that reads XML data for the UNSPSC product category.

Instructions

1. Import the XMLFiles folder into the DataServices project. (The folder is located in the <beahome>\weblogic81\samples\LiquidData\EvalGuide folder.)

2. Right click the XMLFiles folder and select Import Source Metadata.

3. Select XML Data from the Data Source Type drop-down list, then click Next.
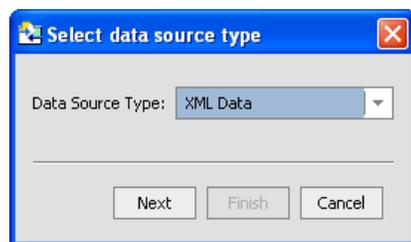
Figure 33-1   Import XML Data
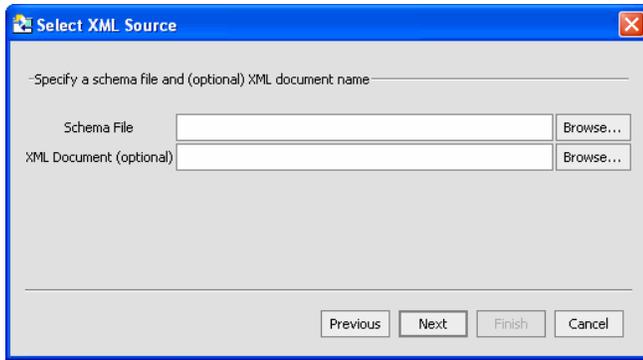
The Select XML Source window opens.

Figure 33-2   Select XML Source Window

4. Associate a schema file with the data service, by completing the following steps:

    a. Click Browse, next to the Schema File field. The XMLFiles directory opens in the Select Schema Files window.

    b. Expand the Schemas folder.
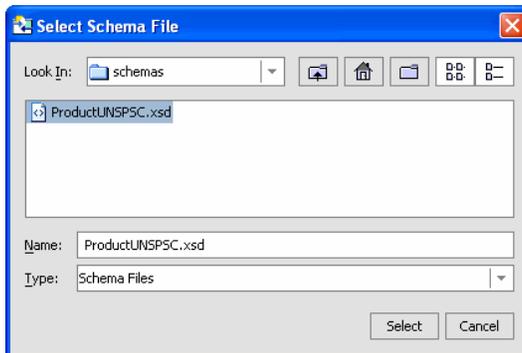
    c. Select ProductUNSPSC.xsd and click Select.

Figure 33-3   Select Schema File

5. Associate the XML Document with the data service, by completing the following steps:

    a. Click Browse, next to the XML Document field. The XMLFiles directory opens in the Select XML Source File window.

    b. Select unspsc.xml and click Select.
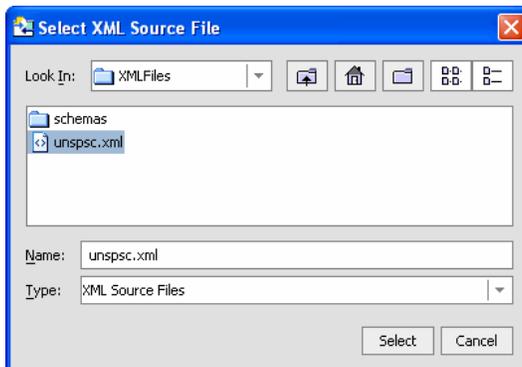
Figure 33-4   Select XML Source File

The Select XML Source window is now populated with file information.
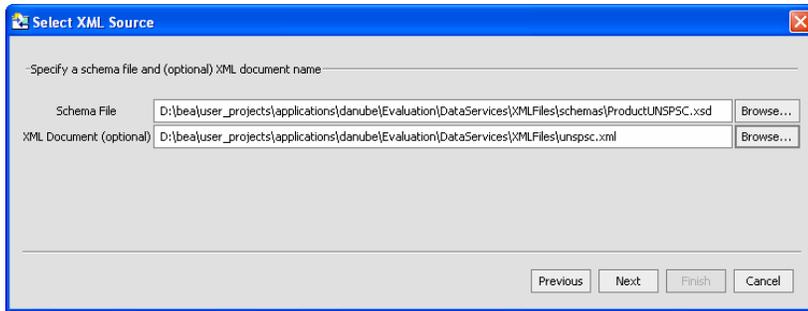
Figure 33-5   Populated Select XML Source Window

6. Click Next. The Summary window opens.
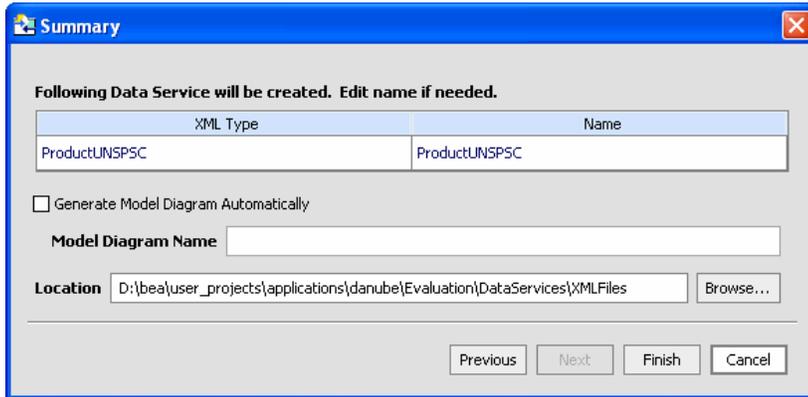
Figure 33-6   Summary Window

The Summary information includes the following details:

> *XML Type,* for XML objects whose source metadata will be imported.
>
> *Name,* for each data service that will be generated from the source metadata. (Any name conflicts appear in red; you can modify any data service name to correct an error condition or to change to a different project-unique name.)
>
> *Location,* where the generated data service(s) will reside.

7. Click Finish. A new data service, called ProductUNSPSC.ds, is created in the DataServices\XMLFiles folder.

Lab 33.2   Testing the XML Data Service

After creating an XML data service, you need to confirm that the service is able to return data, based on the associated XML source file.

Objectives

In this lab, you will:

> Build the DataService project.
>
> Execute the productUNSPSC() function.
>
> Compare the test results with the unspsc.xml file.

Instructions

1. Open ProductUNSPSC.ds in Test View.

2. Test the data service by completing the following steps:

   a. Select productUNSPSC() from the Function drop-down list.

   b. Click Execute.

   c. Confirm that you can retrieve data, as displayed in Figure 33-7.
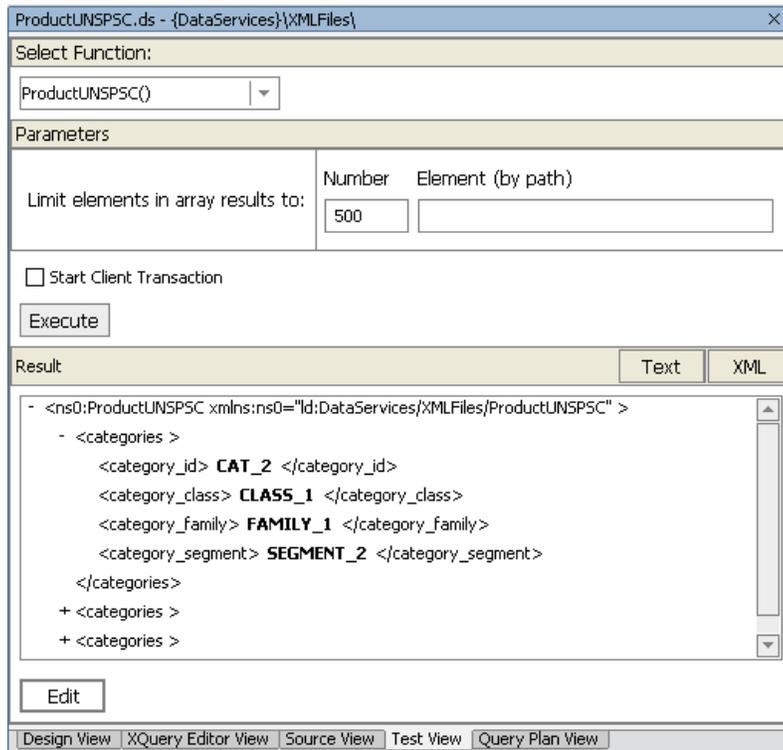
Figure 33-7   XML Data Service Test Results

3. In the Application pane expand the XMLFiles folder and open the unspsc.xml file.

4. Confirm that the test results conform to the specifications in the XML file.

Figure 33-8   XML Elements

Lesson Summary

In this lesson, you learned how to:

Access data in an XML file.

Confirm that the results conform with the contents of the XML file.

Lesson 34 Accessing Data in Flat Files

Flat files, such as spreadsheets, offer a highly adaptable means of storing and manipulating data, especially data that needs to be quickly changed. Flat files are simply treated as another data source that DSP can use to generate metadata and create a data service.

Objectives

After completing this lesson, you will be able to:

Create a data service that can access data stored in a flat file.

Associate the flat file data service with a logical data service.

Overview

Flat files, such as spreadsheets, often support a text format called CSV or Comma Separated Values. Such file formats typically have a .csv extension.

Lab 34.1 Importing Flat File Metadata

The flat file must be in a DSP project, before a data service can be generated. As part of the import process, you must provide a schema name, a file name, or both.

Objectives

In this lab, you will:

Create a data service that queries data stored in a flat file. The flat file contains customer valuation data received from an internal department that deals with customer scoring and valuation models. The file contains the following fields:

Customer_id

Valuation_date

Valuation_score

Instructions

1. Right-click the DataServices folder and import the FlatFiles folder, which is located in the <beahome>\weblogic81\samples\LiquidData\EvalGuide folder.

2. Import source metadata by completing the following steps:

    a. Right-click the FlatFiles folder and select Import Source Metadata.

    b. Select Delimited Data from the Data Source Type drop-down list, then click Next.

    c. Ignore the Schema field.

    d. Click Browse, next to the Delimited Source field.

    e. Select Valuation.csv and click Select.

    f. Confirm that the Has Header checkbox is enabled.

By selecting this option, you specify that the header data, which is usually located in the first row of the spreadsheet, will not be treated as data within the generated data service.

g. Confirm that the Delimited radio button is enabled. By enabling this option, you specify that the data is separated by a specific character, rather than a fixed width such as 10 spaces.

h. Confirm that a comma (,) is in the Delimiter field. If data is delimited, then you must specify what character is used to delimit the data. Although the default is a comma, any ASCII character is supported.

i. Click Next. A summary window opens.

j. Click Finish. A new data service called Valuation.ds is created in the DataServices\FlatFiles folder.

3. Open the Valuation.ds file in Design View.

4. Open Valuation.ds in Design View and confirm that there is a Valuation function. This function will retrieve all data from the flat file.
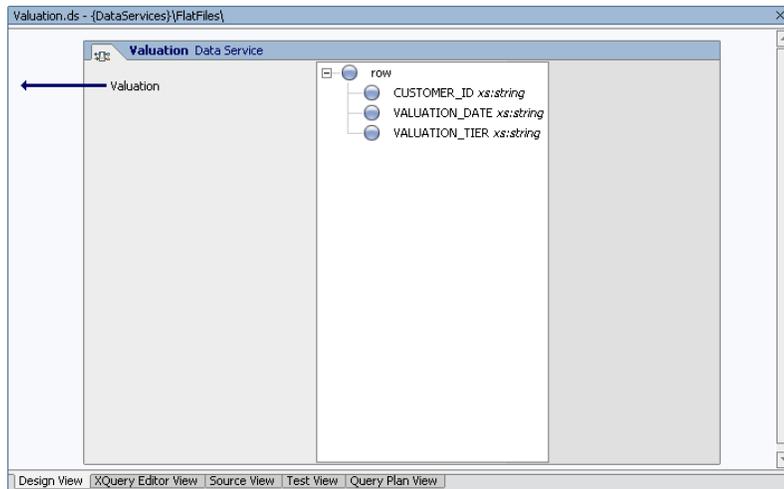

Figure 34-1   Design View of the Data Service Based on a Flat File

Lab 34.2 Testing Your Flat File Data Service

After creating the data service, you need to confirm that the service is able to return data, based on the associated delimited source file.

Objectives

In this lab, you will:

> Build the DataService project.

> Execute the Valuation function.

Instructions

1. Right-click the DataServices folder.

2. Choose Build DataServices.

3. Open Valuation.ds in Test View.

4. Test the data service by completing the following steps:

    a. Select Valuation() from the Function drop-down list.

    b. Click Execute.

5. Confirm that you can retrieve data, as displayed in Figure 34-2. Notice that the return element is introspected. That is based on the header information in the Valuation.csv file.
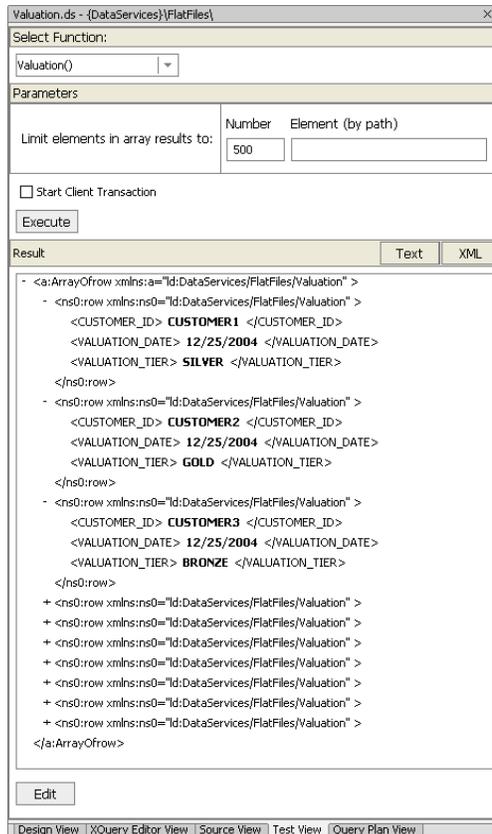

Figure 34-2   Test Results—Flat File Data Service

Lab 34.3   Integrating Flat File Valuation with a Logical Data Service

At this point, you are able to pull data from the flat file. However, integrating the flat file data service into a logical data service lets you retrieve multiple sources of information.

Objectives

In this lab, you will:

> Modify a function to retrieve data from a flat file physical data service.

> View the results in both XQuery Editor View and Source View.

Instructions

1. Open CustomerProfile.ds under DataServices/CustomerManagement/CustomerProfile in XQuery Editor View.

2. Select getAllCustomers() from the Function drop-down list.

3. In the Data Services Palette, expand the FlatFiles and Valuation.ds folders.

4. Drag and drop Valuation() into XQuery Editor View.

5. Create a simple mapping by dragging and dropping the VALUATION_DATE and VALUATION_TIER elements (valuation node) onto the corresponding elements in the Return type.

6. Create a join. Drag and drop the CUSTOMER_ID element (Customer node) onto the corresponding element in the Valuation node. The final layout should be similar to that shown in Figure 34-3:
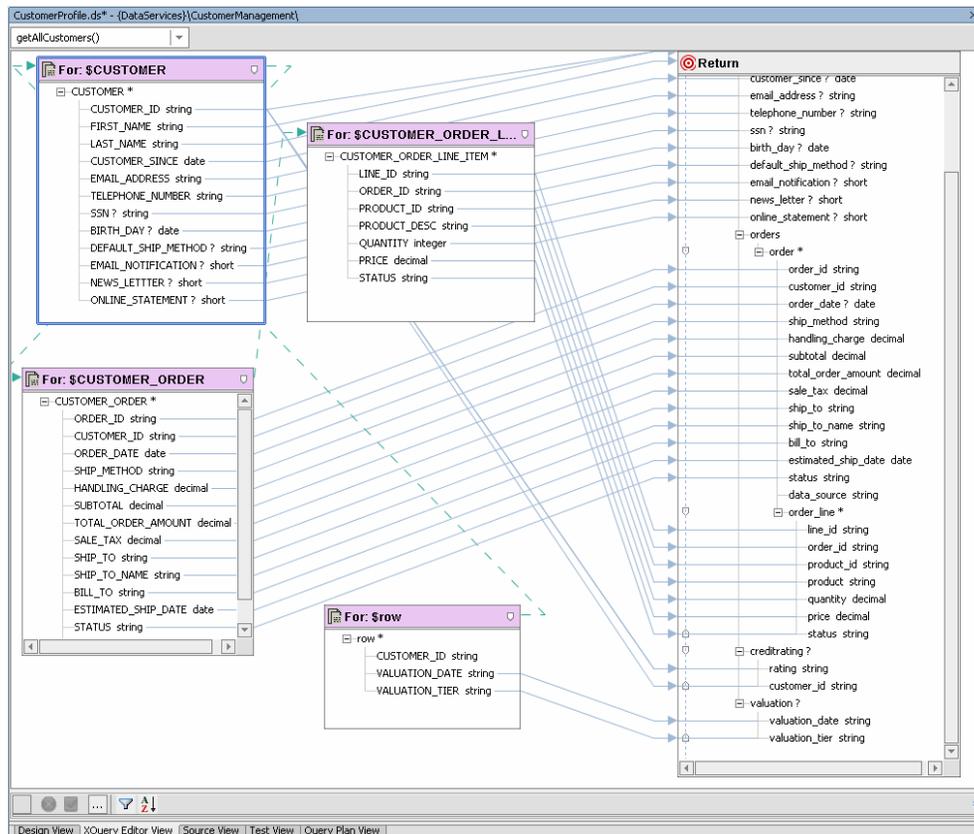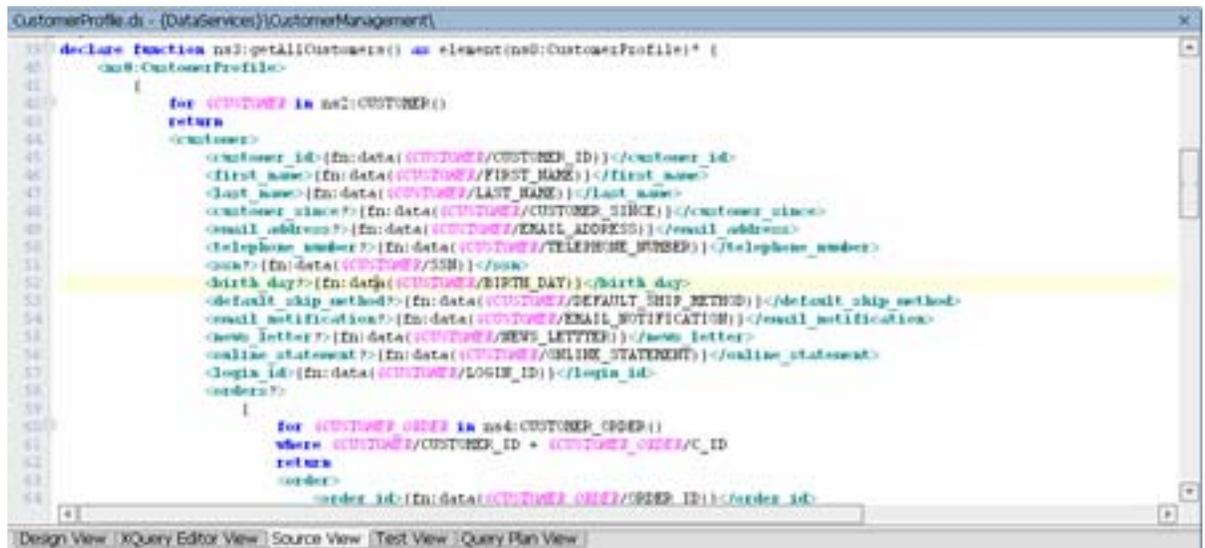
Figure 34-3   XQuery Editor View of Flat File Data Service Integrated with Logical Data Service

7.   Open CustomerProfile.ds in Source View and confirm that the following mapping have been created:

Figure 34-4   Source View of Flat File Data Service Integrated with Logical Data Service

Lab 34.4   Testing an Integrated Flat File Data Service

Testing the function lets you confirm that the data is correctly retrieved.

Objectives

In this lab, you will:

Test the getAllCustomers function.

View the results.

Instructions

1.   Open CustomerProfile.ds in Test View.

2.   Select getAllCustomers() from the Function drop-down list.

3.   Click Execute.

4.   Confirm that you can retrieve valuation information.

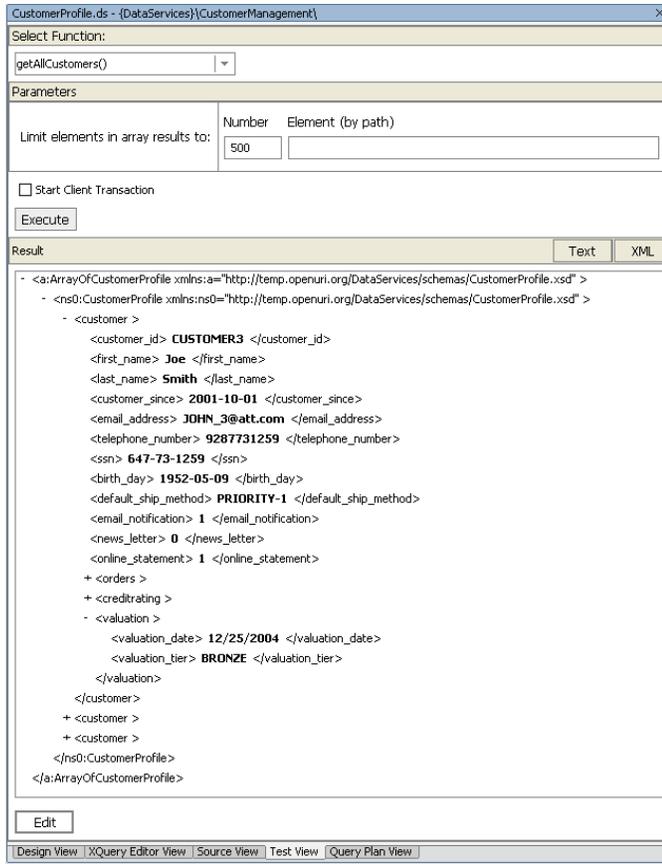Figure 34-5   Test View of Integrated Flat File Data Service

5. (*Optional*) Use the getCustomerProfile function, enter CUSTOMER3 in the Parameter field, and click Execute.

Lesson Summary

In this lesson, you learned how to:

    Import a CSV file containing valuation information.

    Create a flat file physical data service.

    Integrate the flat file physical data service with a logical data service.

Lesson 35 Creating an XQuery Function Library

In any DSP project you can create XQuery libraries containing functions which can be used by any data service in your application. An XQuery function library is ideal for containing transformation and other types of functions without the overhead of having to build a data service. An XQuery function library can also be used to hold security functions which, in turn, can be used by any data service.

Objectives

After completing this lesson, you will be able to:

Create and use XFL functions.

View the results.

Overview

An XQuery Function Library (XFL) contains user functions that return discrete values, such as string, integer, or calendar. These functions are useful for data manipulation at query execution time.

Lab 35.1 Creating an XQuery Function Library

In this lesson, you will "encrypt" a customer's SSN to hide its value. As part of this process you will be modifying the getCustomerProfile() query function.

Objectives

In this lab, you will:

Import a Java file into the DataServices project.

Import source metadata.

Test the function

Instructions

1. Create a new folder in the DataServices project and name it xfl.

2. Import protectSSN.java into the XFL folder. (The file is located in samples\liquiddata folder.)

3. Build the DataServices project.

4. Import source metadata into the xfl folder by completing the following steps:

    a. Right-click the xfl folder and choose Import Source Metadata.

    b. Select Java Function from the Data Source Type drop-down list and click Next.

    c. Enter xfl.protectSSN in the Class Name field and click Next.

Figure 35-1   Selecting the Java File

        d.   Select the protectSSN function, and then click Add.
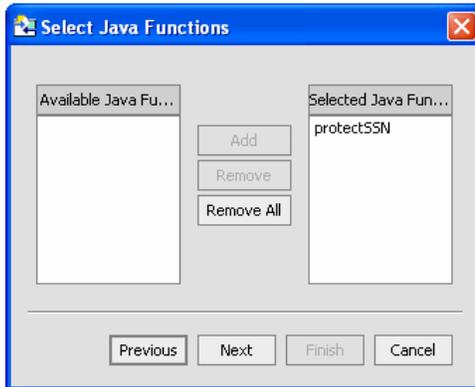
Figure 35-2   Selecting the Java Function

        e.   Click Next. The Summary window opens.
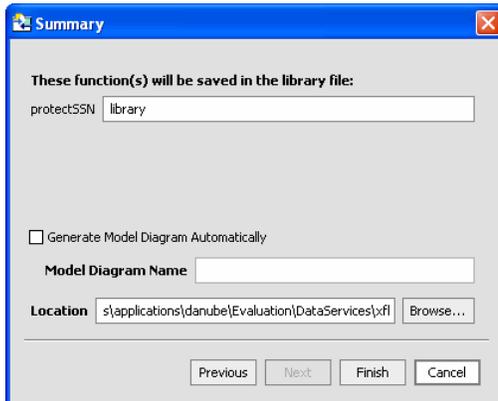
Figure 35-3   Imported Java Metadata Summary

        f.   Click Finish.

5.   Test the function, by completing the following steps:

        a.   Open library.xfl in Test View.

        b.   Select protectSSN from the Function drop-down list.

        c.   Insert any number in the Parameter field; for example, 3.

        d.   Click Execute. The test should return 999-99-9999, regardless of the input parameter.
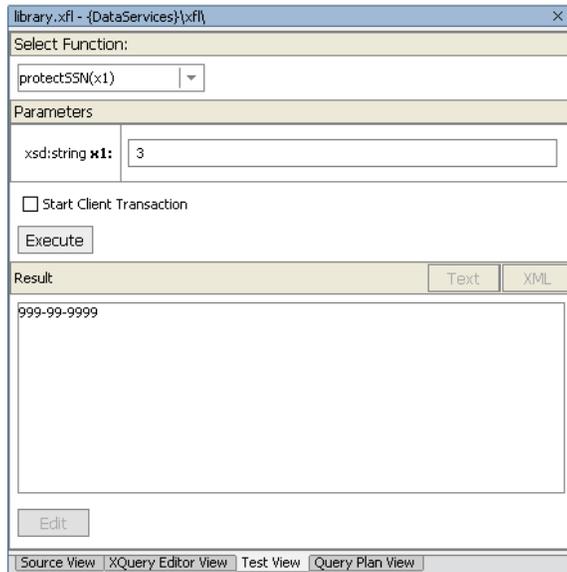
Figure 35-4   XQuery Function Library Test

Lab 35.2   Using the XQuery Function Library in an XQuery

Adding an XQuery Function Library file to an XQuery.

Objectives

In this lab, you will:

> Add the protectSSN.xfl file to an XQuery.
>
> Test the query.
>
> View the results.

Instructions

1. Build the DataServices project.

2. Test the getCustomerProfile() function without the protectSSN function by completing the following steps:

    a. Open CustomerProfile.ds in Test View.

    b. Select getCustomerProfile from the function drop-down list.

    c. Enter CUSTOMER3 in the Parameter field.

    d. Confirm that the query returns a valid SSN.

3. Set SSN protection, by completing the following steps:

    a. Open CustomerProfile.ds in Source View.

    b. Expand the getAllCustomers node.

    c. Locate the SSN return code within the getAllCustomers() function. It should be as follows:

        <ssn?>{fn:data($CUSTOMER/SSN)}</ssn>

    d. In Data Services Palette, expand the xfl and library.xfl folders.

e.   Drag and drop protectSSN() to the SSN return value.

f.   Modify the remaining code, so that it is as follows:

```
{ssn?{ns9:protectSSN($CUSTOMER/SSN)}</ssn>)}
```

Note: <a> is the renamed element. You can use any name for the element, but for the sake of clarity, we used the simple <a> name.

4.   Test the getCustomerProfile() function with the protectSSN function, by completing the following steps:

a.   Open CustomerProfile.ds in Test View.

b.   Select getCustomerProfile from the function drop-down list.

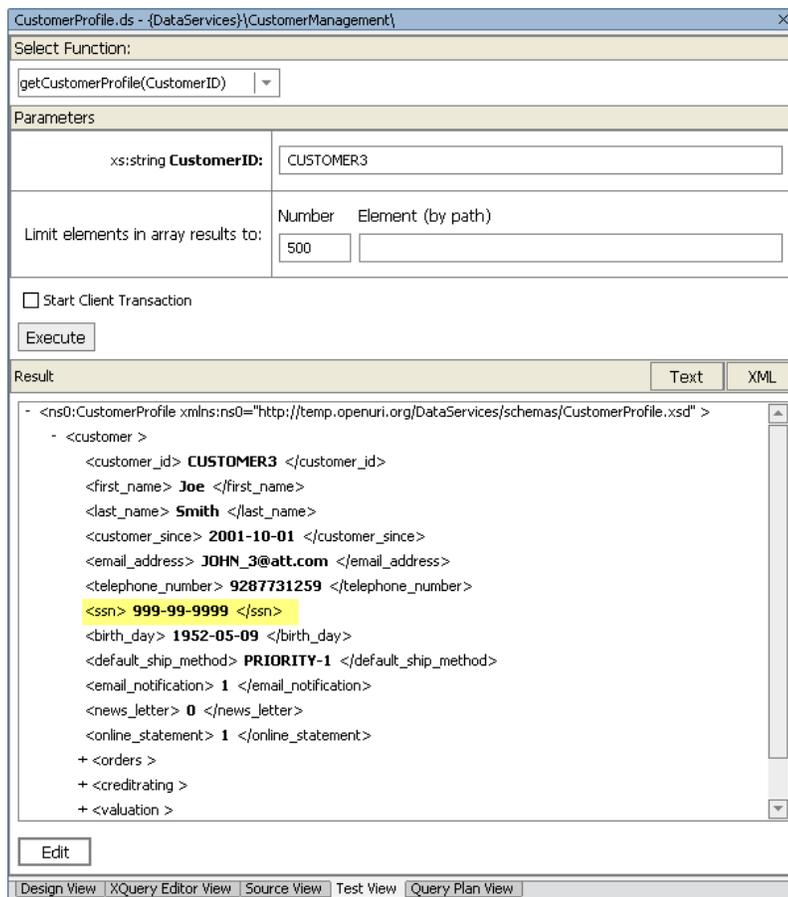c.   Enter CUSTOMER3 in the parameter field. The query should return an invalid social security number.

Figure 35-5   Test View of Protected SSN

Lesson Summary

In this lesson, you learned how to:

Create a XFL function.

Use the XFL function within a query.

Glossary

ad-hoc query. A hand-coded or generated query that is passes to Data Services Platform on the fly, rather than stored in the DSP repository.

administration console. A web-based administration tool that an administrator uses to configure and monitor WebLogic Servers. DSP provides a console to help manage instances of Data Services Platform.

application. A collection of all resources and components deployed as a unit to an instance of WebLogic Server. The application contains one or more projects, which in turn contain the folders and files that make up your application. Only one application can be open at a time.

cache. The location where DSP stores information about commonly executed stored queries for subsequent, efficient retrieval, thereby enhancing overall system performance. DSP provides query plan cache and result set cache.

cache policy. In the result set cache, configuration settings determine when the cached results expire for individual stored queries.

data model. A visual representation of data resources.

data object. In SDO, a complex type that holds atomic values and references to other data objects.

data service. A modeled object that describes a data shape and functions used to retrieve and update the data, as well as functions to navigate to other related data services.

data service mediator. The SDO mediator that uses data services to retrieve and update data.

data service update. The engine responsible for handling submits of changes to SDOs

data source. Any structured, semi-structured, or unstructured information that can be queried. The types of data sources that DSP can query include relational databases, Web services, flat files (delimited and fixed width), XML files, Java functions, application views via web applications (business-level interfaces to the data in packaged applications such as Siebel, PeopleSoft, or SAP), data views (dynamic results of DSP queries).

data source schema. An XML schema that defines the content, semantics, and physical structure of a data source.

function. A uniquely named portion of an XQuery that performs a specific action. In the case of DSP the function would typically query physical or logical data.

java server page (JSP). A J2EE component that extends the Servlet class, and allows for rapid server-side development of HTML interfaces that can be co-mingled with Java.

logical data service. A data service that integrates data from multiple physical and/or logical data services.

mapping. The process of connecting data source schemas to a target (result) schema.

metadata. Descriptors about a data service's information, format, meaning, and lineage.

physical data service. The leaf-level data services that expose external data. For relational sources, this would be a data service representing tables or stored procedures. For functional sources, this would be the functions that are considered to be the initial source of data operated on by XQuery.

project. Groups related files within an application.

query. In DSP an XQuery function that retrieves data from a data source. Functions define what tasks the query will perform, while expressions define what data to extract.

query operation. Operation that a query performs, such as a join, aggregation, union, or minus.

query plan. A compiled query. Before a query is run, DSP compiles the XQuery code into an executable query plan. When the query executes, the query plan is sent to the data source for processing.

repository. File-based metadata maintained in a DSP project.

result set. The data returned from an executed query. There are two types of result sets: intermediate result sets are temporary result sets that the query processor generates while processing an analytical query; final result sets are returned to the client application that requested the query in the form of XML data.

return type. A type of XML schema that defines the shape of data returned by a query.

schema. A model for representing the data types, structure, and relationships of data sets and queries.

security. Set of mechanisms available to prevent access to, corruption of, or theft of data. DSP extends the WebLogic Server compatibility security mechanisms to define groups, users, and access control to DSP resources.

service data object (SDO). Defines a Java-based programming architecture and API for data access.

simple object access protocol (SOAP). An extensible, platform-independent, XML-based protocol that allows disparate applications to exchange messages over the Web. SOAP can be used to invoke methods on servers, Web services, application components, and objects in a distributed, heterogeneous environment. SOAP-based Web services are one of the data sources DSP supports.

source schema. XML schema that describes the shape (structure and legal elements) of the source data — that is, the data to be queried. The DSP-enabled server runs queries against source data and returns query results in the form of the source schema.

stored query. A query that has been saved to the DSP repository. There is a performance benefit to using a stored query because its query plan is always cached in memory, optionally along with query result. With an ad-hoc query, however, the query plan and result are not cached. In addition, caching of query results for a stored query is configurable through the Cache tab on the DSP node in the Administration Console.

structured query language (SQL). The standard, structured language used for communicating with relational databases. Database programmers use SQL queries to retrieve information and modify information in relational databases. In order to be able to access different types of data sources dynamically, DSP employs the XML-based XQuery language as a layer on top of platform-dependent query systems such as SQL.

target schema. See return type.

weblogic server. The platform upon which DSP is built.

weblogic workshop. The IDE in which DSP runs as an application.

web service. Business functionality made available by one company, usually through an Internet connection, for use by another company or software program. Web services are a type of service that can be shared by, and used as components of, distributed Web-based applications. Web services communicate with clients (both end-user applications or other Web services) through XML messages that are transmitted by standard Internet protocols, such as HTTP. Web services endorse standards-based distributed computing. Currently, popular Web Service standards are Simple Object Access Protocol (SOAP), Web services description language (WSDL), and Universal Description, Discovery, and Integration (UDDI).

web services description language (WSDL). Specification for an XML-based grammar that defines and describes a Web service. A WSDL is necessary if two different online systems need to communicate without human intervention.

xml schema. A structured model for describing the structure, content, and semantics of XML documents based on custom rules. Unlike DTDs, XML schemas are written in XML data syntax and provide more support for standard data types and other data-specific features. When metadata about a data source is obtained, it is stored in an XML schema in the DSP repository.

xquery. An XML query language, which represents a query as an expression which is used to query relational, semi-structured, and structured data.

xsd. An abbreviation for XML Schema Definition. An XSD file describes the contents, semantics, and structure of data within an XML document.