BEA WebLogic® Integration

Tutorial: Building Your First Data Transformation

Version 10.2
Document Revised: March 2008

Contents

CHAPTER 1

Tutorial: Building Your First Data Transformation

Data transformation is the mapping and conversion of data from one format to another. For example, XML data can be transformed from XML data valid to one XML Schema to another XML document valid to a different XML Schema. Other examples include the data transformation from non-XML data to XML data. This tutorial introduces the basics of building a data transformation by describing how to create and test a XML-to-XML data transformation using BEA WorkSpace Studio.

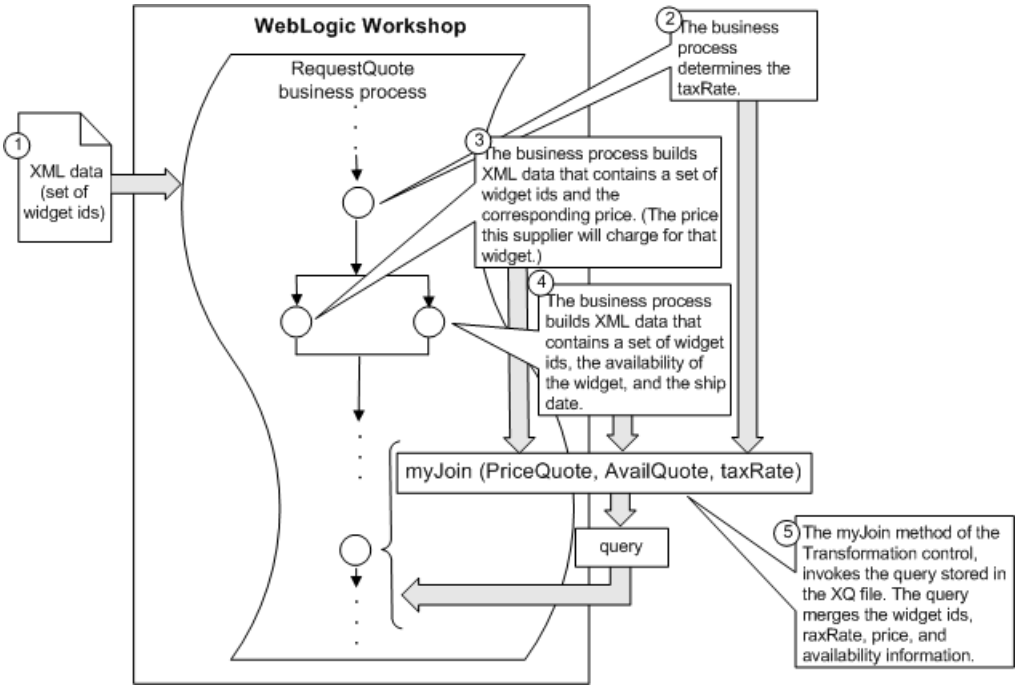
In WebLogic Integration business processes, a data transformation transforms data using queries (written in the XQuery language). This tutorial describes the steps for building a query in the XQuery language—a language defined by the World Wide Web Consortium (W3C) that provides a vendor independent language for the query and retrieval of XML data.

To learn about the XQuery language, see the XQuery 1.0: An XML Query Language Specification -W3C Recommendation 23 January 2007 .

The data transformation created in this tutorial is invoked in the RequestQuote business process. This business process is created to meet the business needs of an enterprise. The enterprise starts the business process as a result of receiving a Request for Quote from clients, checks the enterprise's inventory and pricing systems to determine whether the order can be filled, and sends a quote for the requested items to the client. To learn more about creating business processes and the RequestQuote business process, see *Tutorial: Building Your First Business Process*.

The following figure shows the flow of data in the RequestQuote business process of the Tutorial Process application.

Figure 1-1 Representing Flow of Data



The purpose of the RequestQuote business process is to provide price and availability information for a set of widgets. The flow of the data through the RequestQuote business process is represented by the following steps:

1. The business process receives the set of widget IDs.
2. The business process determines the tax rate for the shipment and puts the result in the `taxRate` float business process variable.
3. The business process gets the price of each of the requested widgets from a source and places the resulting XML data into the `priceQuote` business process variable. (This XML data is valid to the XML Schema in the `PriceQuote.xsd` file.)
4. The business process gets information about availability for the widgets from another source and places the resulting XML data into the `availQuote` business process variable. (This XML data is valid to the XML Schema in the `AvailQuote.xsd` file.)

5. The business process invokes the **Combine Price and Avail Quotes** node. The **Combine Price and Avail Quotes** node calls the `myJoin` Transformation method stored in the Transformation file called `MyTutorialJoin.java` file. The business process passes the values of the `priceQuote`, `availQuote`, and `taxRate` business process variables to the `myJoin` method. The `myJoin` method invokes the query written in the XQuery language and stored in the `myJoin.xq` file. The query merges all the price, availability, and tax rate information into a single set of XML data and returns the result as the return value of the `myJoin` method. The data returned from this `myJoin` method is valid to the XML Schema in the `Quote.xsd` file. After the `myJoin` method is invoked, the **Combine Price and Avail Quotes** node assigns the resulting XML data to the `Quote` business process variable.

Tutorial Goals

The tutorial provides steps to create and test a transformation using the graphical environment provided in BEA Workshop for WebLogic Platform. Specifically, in this tutorial you will create the following:

- The **MyTutorialJoin** Transformation file.
- The `myJoin` Transformation method in the **MyTutorialJoin** Transformation file.
- The query invoked by the `myJoin` Transformation method. This query is stored in the XQ file called `myJoin.xq`.

Steps in This Tutorial

Follow the steps in this tutorial to create and test a data transformation. Specifically, the steps include:

Chapter 2, "Step 1: Getting Started"
Describes how to load the prepackaged Tutorial Process Application.

Chapter 3, "Step 2: Building the Transformation"
Provides a step-by-step procedure to create and select source and target types for a Transformation method.

Chapter 4, "Step 3: Mapping Elements and Attributes"
Provides a step-by-step procedure to create mappings between source and target elements and attributes in a Transformation method.

Chapter 5, "Step 4: Mapping Repeating Elements—Creating a Join"
Provides a step-by-step procedure to add a join between repeating elements to the Transformation method.

CHAPTER 2

Step 1: Getting Started

The Business Process and Data Transformation Tutorials both use a prepackaged Tutorial Process application. The prepackaged **Tutorial: Request Quote Process Application** contains all the business process, XML, XML Schema, Transformation, and XQ files, required to run the tutorial business processes and transformations.

The RequestQuote business process in the Tutorial Process application invokes a transformation stored in the `TutorialJoin.java` and `join.xq` files. The steps in this Tutorial explain the procedures to create the same transformation that is prepackaged in the `TutorialJoin.java` and `join.xq` files of the **Tutorial: Request Quote Process Application**. (You can use the transformation in the `TutorialJoin.java` and `join.xq` files as a reference.) Name the Transformation file `MyTutorialJoin.java` and the XQ file that contains the query: `myJoin.xq`.

After completing the steps in this Tutorial, modify the RequestQuote business process to invoke the transformation created in this tutorial. In addition, run the RequestQuote business process which will invoke the transformation, as described in Step 12: Run the Request Quote Business Process of the Business Process Tutorial.

Note: If you followed the steps described in the Business Process Tutorial, you have already created an application and can skip the "To Load The Tutorial Process Application" on page 2-2 task.

The task in this step include:

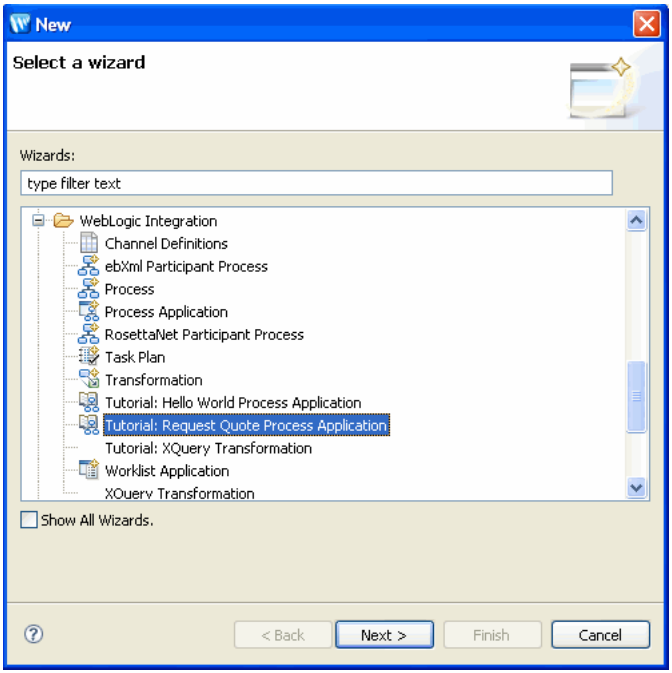
- To Load The Tutorial Process Application

To Load The Tutorial Process Application

In this task, you load the prepackaged **Tutorial: Process Application**.

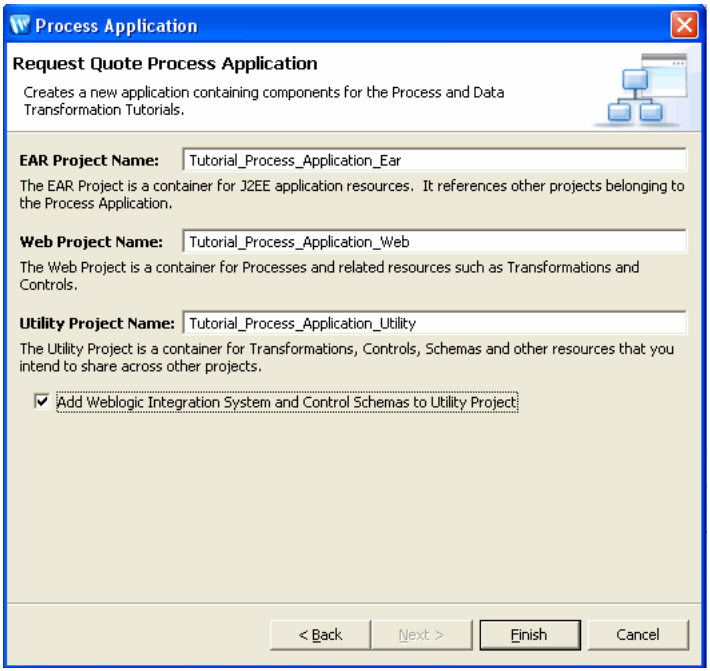
1. From the **BEA WorkSpace Studio** menu, click **File** > **New** > **Other**. The **Select a Wizard** dialog box is displayed.
2. Expand **WebLogic Integration**, and select **Tutorial:Request Quote Process Application**, and click **Next**.

Figure 2-1 Select a Wizard dialog box



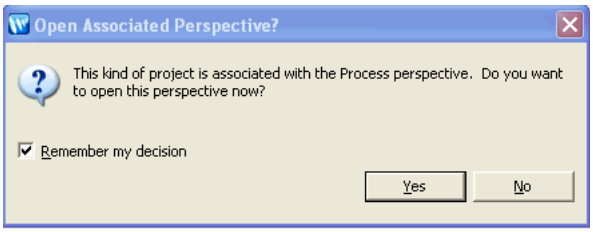
3. In the **Request Quote Process Application** dialog box, type the following:
   a. In the **Ear Project Name** field, enter `Tutorial_Process_Application_Ear`.
   b. In the **Web Project Name** field, enter `Tutorial_Process_Application_Web`.
   c. In the **Utility Project Name** field, enter `Tutorial_Process_Application_Utility`.

Figure 2-2 Process Application dialog box



4. Select the **Add WebLogic Integration System and Control Schemas to Utility Project** checkbox.
5. Click **Finish**.
6. In the displayed **Open Associated Perspective?** dialog box, click **Yes** to switch from Workshop Perspective to Process Perspective. Select the **Remember my decision** checkbox.

Figure 2-3 Open Perspective Confirmation

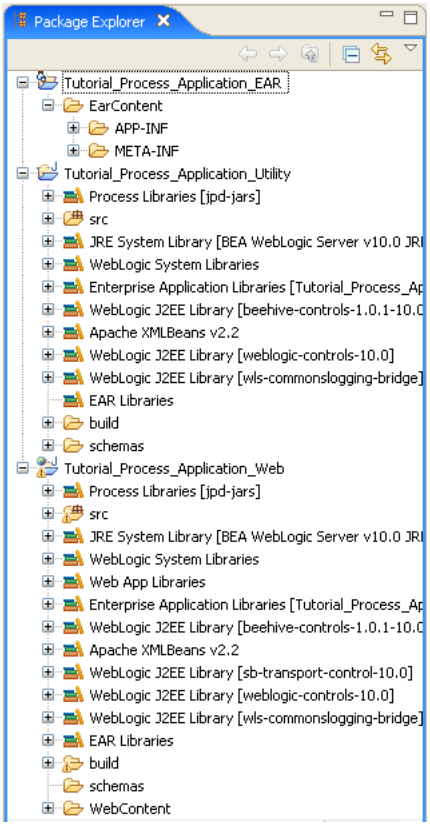


Note: **J2EE** is the default perspective of the **BEA WorkSpace Studio**. The Process perspective contains all the required views like **Node Palette**, **Data Palette**, and so on.

Similarly, XQueryTransformation perspective contains views pertaining to XQuery Transformation like **Expression Functions, Expression Variables**, **Target Expression**, and **Constraints**.

7. The Tutorial Process Application is created and displayed in the **Package Explorer** pane.

Figure 2-4 Package Explorer Pane



The **Package Explorer** pane displays the files and resources available in the application:

Ear Project — An EAR project is the central point of application. An EAR project contains JAR files that are shared by the projects in the enterprise application. This project

contain links to all of the projects in the application. The project files are used by BEA WorkSpace Studio to test and deploy enterprise applications that contain multiple projects. The EAR Project files are used to create EAR (Enterprise Archive) files.

Web Project— A project with WebLogic Integration process facet added to it. Every application contains one or more projects. Projects represent WebLogic Server applications. In other words, when you create a project, you are creating a Web application. (The name of your project is included in the URL that clients use to access your application.)

Utility Project—A project that contains the XML Schemas and the Message Broker channel file used in the application.

Web Applications are J2EE deployment units that define a collection of Web resources such as business processes, Web services, JSPs, servlets, HTML pages, and can define references to external resources such as EJBs.

requestquote—contains the business processes, transformation, xq files

- **FileQuote.java**—A File control used by your Request for Quote business process to write a file to the file system.
- **PriceAvailTransformations.java—**Contains data transformations used in `RequestQuote.java`.
- **RequestQuote.java**—The completed business process. (The tutorial walks you through rebuilding this business process. It is provided for reference, and allows you to run and test the business process before you start rebuilding it.)
- **RequestQuoteTransformation.java** and **TutorialJoin.java**—Contains data transformations used in `RequestQuote.java`.

XQ files—An XQ file is created for each transformation method on a transformation file. XQ files contain the queries (written in the XQuery language) called by the transformation files in your project.

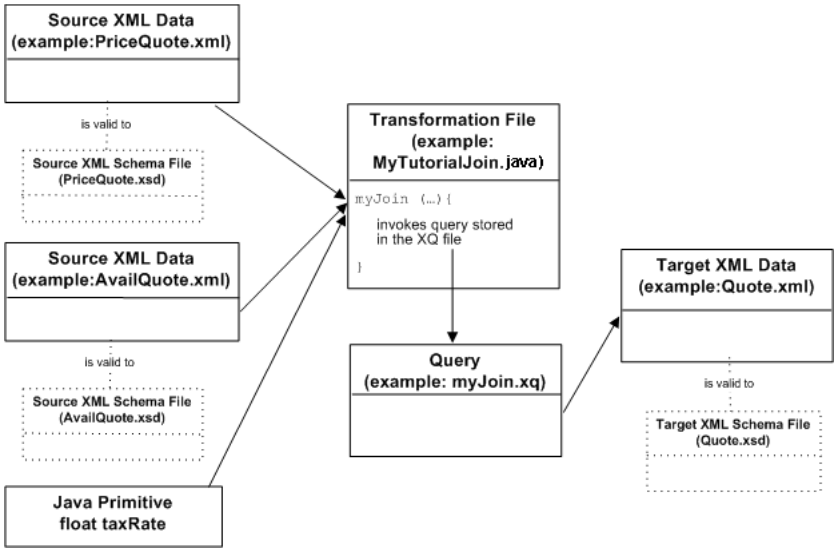
requestquote.services folder contains services with which your business process interacts. The **services** folder includes Web services, Web Service controls, business processes and Process controls.

testxml folder contains XML files which you can use to test the completed business process.

Step 2: Building the Transformation

In this step, you create a transformation that contains the mapping of different source (input) types to a single target (output) type. Specifically, this tutorial provides the steps for transforming a Java primitive and two sets of XML data (valid to two different schemas) to a single set of XML data valid to a third schema, as shown in the following figure.

Figure 3-1 Mapping between source and target types



The RequestQuote business process takes as input a set of widget IDs and returns the price and availability of these widget IDs.

The source parameters to the `myJoin` Transformation method include the following:

- XML data valid to the `PriceQuote.xsd` file. The RequestQuote business process of the Tutorial Process application builds a piece of XML data that is valid to the `PriceQuote.xsd` XML Schema and stores it in a business process variable called `priceQuote`. This piece of XML data contains a set of widget IDs and their price.
- XML data valid to the `AvailQuote.xsd` file. The RequestQuote business process of the Tutorial Process application builds a piece of XML data that is valid to the `AvailQuote.xsd` XML Schema and stores it in a business process variable called `availQuote`. This piece of XML data contains a set of widget IDs, a boolean that represents if the widget is available, and the ship date.
- A Java primitive of type float called `taxRate`.

The `myJoin` Transformation method takes these source parameters and invokes a query which merges the price, availability, and tax rate information into one piece of XML data valid to the XML Schema in the `Quote.xsd` file.

The tasks in this step include:

- To Create MyTutorialJoin.java
- To Add a Transformation method to MyTutorialJoin
- To Select the Source Types
- To Select the Target Type

To Create MyTutorialJoin.java

In this task, you create a Transformation file called `MyTutorialJoin.java`. In addition, you create a Transformation method in the Transformation file. During run time, the business process will call this method to invoke the transformation.

1. In the **Package Explorer** pane, right-click the `src > requestquote` folder and from the drop-down menu, select **New** > **Transformation**.
2. The **New Transformation** dialog box is displayed.
3. In the **Name** field, enter `MyTutorialJoin`.
4. In the **New Transformation** dialog box, click **Finish**.

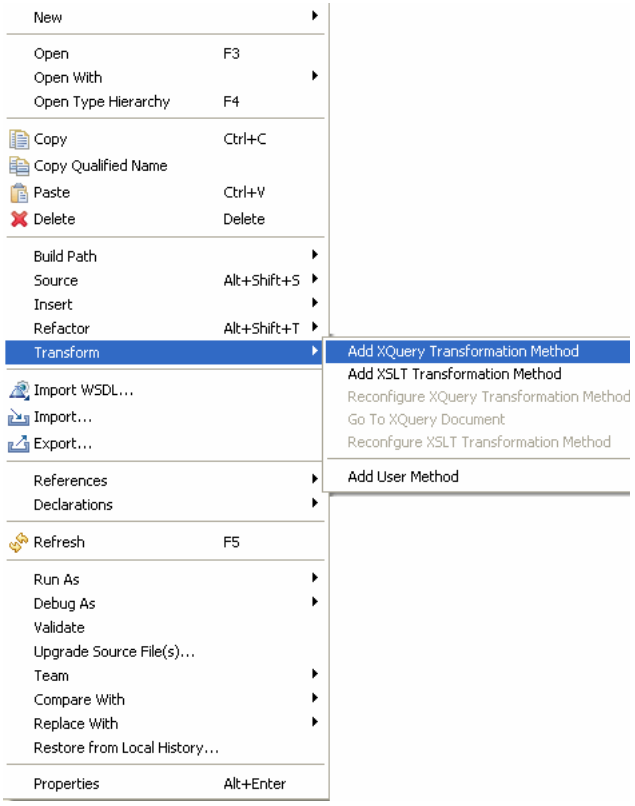
The `MytutorialJoin.java` is created under the `src > requestquote` folder.

Note: To alternatively create Transformation, click the drop down arrow in the **Data Palette** view, select **Integration Controls**, and then **Transformation**.

To Add a Transformation method to MyTutorialJoin

1. In the **Package Explorer** pane, double click on `MyTutorialJoin.java`.
2. Right-click in the `MyTutorialJoin.java` **Source** pane that is displayed.
3. From the popup menu, select **Transform**, and then **Add XQuery Transformation Method**.

Figure 3-2 Transform Pop-up Menu



4. In the **New XQuery Transformation Method** dialog box, type the values for **Transformation Method Name**, and **XQuery File Name** fields. For example, Type `myJoin` as the Transformation Method Name, and `myJoin.xq` as the XQuery File Name. You can also accept the default values, and click **Next**.

5. Select the source (input) types for the transformation from the **Source Types** dialog box. The available source types are XML, Non-XML, and Java. See To Select the Source Types.

6. click **Next**

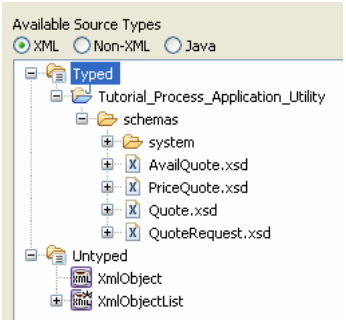
7. Select a target (output) type for the transformation from the **Target Types** dialog box. See, To Select the Target Type.

To Select the Source Types

In this task, you select the source types for the transformation in the **Source Types** dialog box of **New XQuery Transformation** wizard. Source types are the input data types for the transformation—the data types that are transformed to the target data type.

1. In the **Available Source Types** pane, the application XSD files are displayed, as shown in following figure.

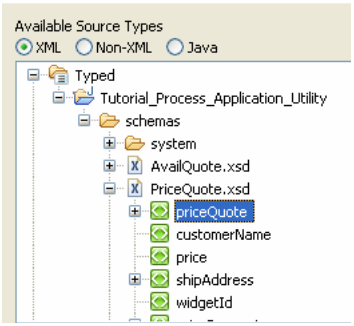
Figure 3-3 Available Source Types pane



Note: If these files are not listed, you probably have not loaded the **Tutorial: Process Application**. For instructions on loading this application, see "To Load The Tutorial Process Application" on page 2-2.

2. In the **Available Source Types** pane, expand **schemas** and **PriceQuote.xsd** folder, then select the **priceQuote** element, as shown in the following figure.

Figure 3-4 XML-Types

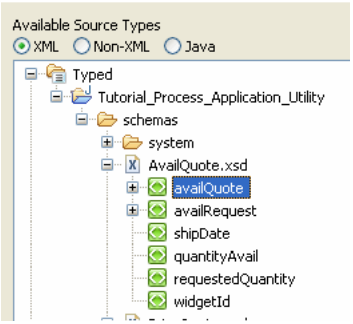


3. Click **Add**.

   The elements and attributes that make up the **priceQuote** element are displayed in the **Selected Source Types** pane.

4. In the **Available Source Types** pane, expand **AvailQuote.xsd** folder, then select the **availQuote** element.

Figure 3-5 Available Source Types - XML Options



5. Click **Add**.

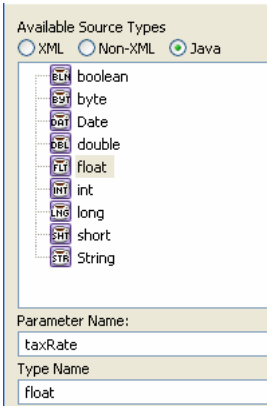
   The elements and attributes that make up the **availQuote** element are displayed in the **Selected Source Types** pane.

6. In the **Available Source Types** pane, select the **Java** option.

   The available Java Types are displayed in the **Available Source Types** pane.

7. In the **Available Source Types** pane, select the **float** node, as shown in the following figure.

Figure 3-6 Select float node from Available Source Types



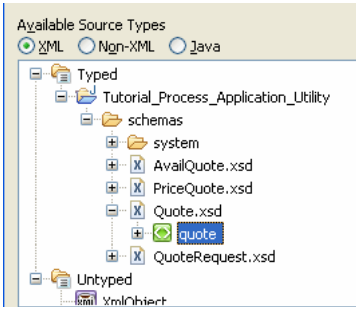
8. Type `taxRate` as the **Parameter Name**.
9. Click **Add**
10. Click **Next.**

To Select the Target Type

In this task, you select a target type for the transformation in the **Target Types** dialog box of **New XQuery Transformation**.

1. In the **Available Target Types** pane of the **Target Types** dialog box, the **PriceQuote.xsd, AvailQuote.xsd, Quote.xsd,** and **QuoteRequest.xsd** files are listed.
2. In the **Available Target Types** pane, expand **Schemas** and **Quote.xsd** folder, then select the **quote** element, as shown in the following figure.

Figure 3-7 Available Target Types Pane



3. Click **Add**.

   The elements and attributes that make up the **quote** element are displayed in the **Selected Target Types** pane.

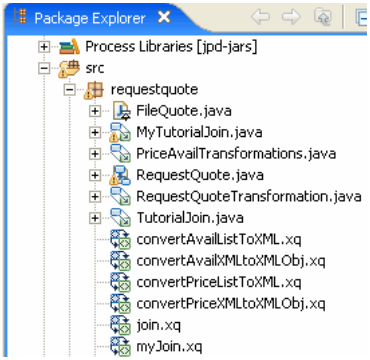
4. Click **Finish**.

   The file: `myJoin.xq` is created and displayed in the **Design** view.

   The `myJoin` Transformation method is added to **MyTutorialJoin** Transformation file. The `myJoin` method contains the three source parameters selected from priceQuote.xsd, availQuote.xsd, and the float java type.

   In the **Package Explorer** pane, representations of the **MyTutorialJoin.java** and **myJoin.xq** files are displayed as shown in the following figure.

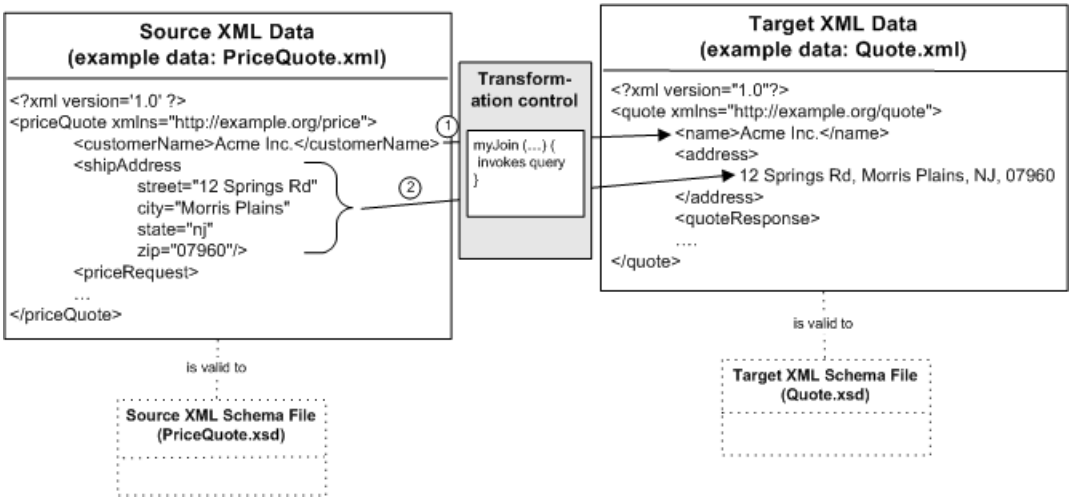
Figure 3-8 MyTutorialJoin.Java file in Package Explorer Pane



Step 3: Mapping Elements and Attributes

In this step, you map source nodes to target nodes. The following figure shows the mapping of example XML data.

Figure 4-1 Mapping Example



In the preceding figure, the source XML data has a different format than the target XML data. When building a query invoked by a Transformation method, you map the source nodes to target nodes as represented by the arrows. During run time, the transformation uses the mappings to convert the data from the source format to the target format. For example, the arrow labeled 1

represents the transformation of the `priceQuote/customerName` element to the `quote/name` element.

The mapping of the address data, is a more complex transformation, as represented by the arrow labeled 2 in the preceding figure. To transform the address information, all the attributes of the `shipAddress` element (`street`, `city`, `state`, and `zip`) must be converted to a single string XML element called `address`.

The source XML data is valid to a different XML Schema than the target XML data. As shown in the preceding figure, the example source XML document called `PriceQuote.xml` is valid to the XML Schema in the `PriceQuote.xsd` file. Additionally, the example source XML document called `Quote.xml` is valid to the XML Schema in the `Quote.xsd` file.

Note: The `PriceQuote.xml`, `AvailQuote.xml`, `QuoteRequest.xml` files are located in the `Tutorial_Process_ApplicationWeb/requestquote/testxml` directory of the application.

Note: Figure 4-1 shows one source data type (`priceQuote`). This is just one of the three sources to the `myJoin` method as described in "Step 2: Building the Transformation" on page 3-1. In this step, the mappings between the XML Schema in the `PriceQuote.xsd` file to the XML Schema in the `Quote.xsd` file are discussed. In the "Step 4: Mapping Repeating Elements—Creating a Join" on page 5-1, mappings between the other source types (`AvailQuote.xsd` and `taxRate`) are discussed.

The `PriceQuote.xml`, `AvailQuote.xml`, `QuoteRequest_a.xml`, `QuoteRequest.xml`, and `Quote.xml` files are provided as examples and are not used by the business process during run time. During run time, the business process constructs the source XML data, and passes it to the transformation as described in the Introduction of this tutorial.

Complete the following tasks to create, alter, and test mappings between the source and target data:

- To Map a Node From a Source to a Target
- To Map Attributes of an Element to Single Element
- To View and Save the Generated Simple Query
- To Test a Simple Query
- To Edit and Retest the Simple Query
- To Add an XQuery Function Call to the Query

To Map a Node From a Source to a Target

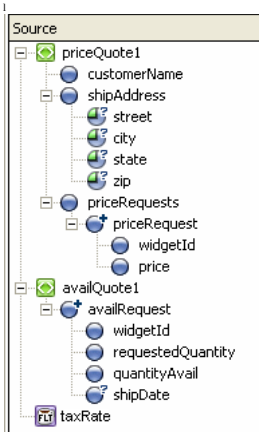
In this step, you map the XML string element called **customerName** from the source (`PriceQuote.xsd`) to the XML string element called **name** in target (`Quote.xsd`).

1. View `myJoin.xq` in the **Design** view:

   a. In the **Navigator** pane, browse to the **src/requestquote** folder, and double click on **myJoin.xq**.

   The **Design** view displays the a graphical representation of the selected sources in the **Source** pane, as shown in the following figure.

Figure 4-2 Graphical Representation of Design View



Note: If the **priceQuote1**, **availQuote1**, and **taxRate** nodes are not displayed in your **Source** pane, follow the instructions in "To Select the Source Types" on page 3-4.

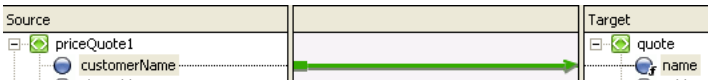
The nodes displayed in the **Source** pane correspond to source parameters of the `myJoin` method of the **MyTutorialJoin** Transformation file. The signature of the `myJoin` method from the `MyTutorialJoin.java` file is shown in the following Java code segment:

```
public abstract org.example.quote.QuoteDocument
myJoin(org.example.price.PriceQuoteDocument priceQuote1,
org.example.avail.AvailQuoteDocument availQuote1, float taxRate);
```

2. From the **Source** pane of the **myJoin** XQ file, drag-and-drop the **priceQuote1**/**customerName** node onto the **quote**/**name** node in the **Target** pane.

   A solid line appears between the two elements. This solid line represents a data link between the two nodes—a link that converts the value of the source node directly to the value of the target.

Figure 4-3 Link between two elements



This link corresponds to the mapping represented with an arrow (labeled with the number 1) in Figure 4-1.

To Map Attributes of an Element to Single Element

In this step, you will map multiple attributes of one element to another single element.

The XML **priceQuote1**/**shipAddress** element contains the following attributes:

- street
- city
- state
- zip

All these attributes will be mapped to the single XML **quote**/**address** element of type string. This mapping is represented by the arrow labeled 2 in Figure 4-1.

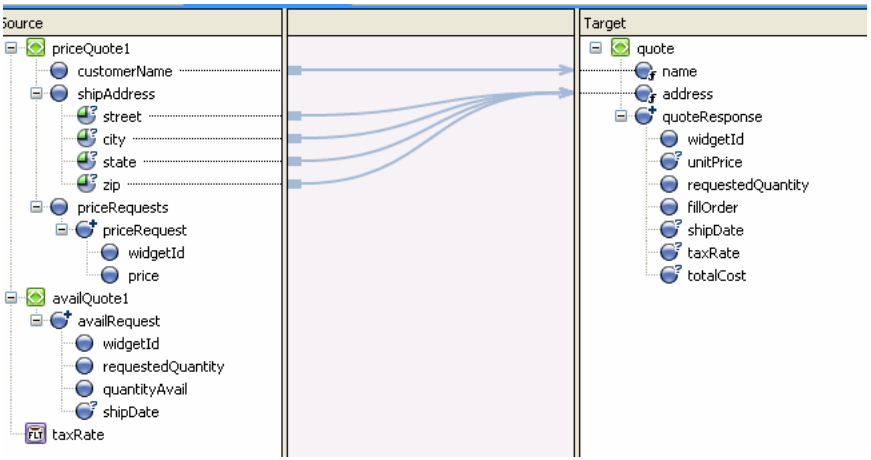
Link the multiple **shipAddress** attributes from the **Source** pane to the **Target** pane with drag-and-drop operations, as described in the following procedure:

1. In the **Source** pane, select the **street** attribute of **priceQuote1**/**shipAddress** node.
2. Drag-and-drop the Street attribute from the **Source** pane to the **quote**/**address** node in the **Target** pane.
3. Similarly select each of the individual attributes, drag-and-drop them from the **Source** pane to the **quote**/**address** node in the **Target** pane.

   Note: Note: You can also use the Shift key to select groups of nodes.

   Four new links are displayed, as shown in the following figure.

Figure 4-4 Create Links



The links labeled with numbers in Figure 4-4, correspond to the mappings represented as arrows (label number 2) in Figure 4-1.

To View and Save the Generated Simple Query

A query (in the XQuery language) is generated when you create mapping links from source elements and attributes to target elements and attributes.

1. Select the **Source** view of the `myJoin.xq` file.

   The generated query is displayed as shown

Figure 4-5 Source view of the myjoin.xq file

```
<ns2:quote>
    <name>{ data($priceQuote1/ns0:customerName) }</name>
    <address>{ concat($priceQuote1/ns0:shipAddress/@street ,
                      $priceQuote1/ns0:shipAddress/@city ,
                      $priceQuote1/ns0:shipAddress/@state ,
                      $priceQuote1/ns0:shipAddress/@zip) }
    </address>
</ns2:quote>
```

Note: The XQuery code labeled with numbers in Figure 4-1 correspond to the numbered mappings and links in Figure 4-4, respectively.

2. Save all the files in this application. From the **BEA WorkSpace Studio** menu bar, choose **File** > **Save All**. You can also save all the files by entering **Ctrl**+**Shift**+**S**.

Note: Pressing **Ctrl+S** saves just the active file

To Test a Simple Query

This section describes the steps necessary to test the query generated in the preceding section. In this section, you will enter source XML data, run that data against the query, and view the resulting target XML data.

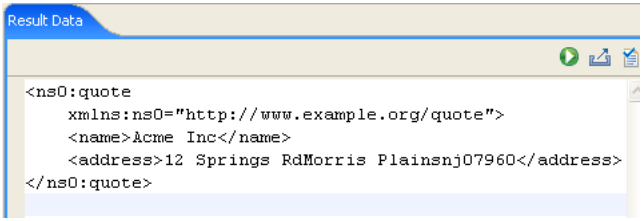
1. Select the **Test** tab of `myJoin.xq` file.
2. Import `PriceQuote.xml` as source data for the transformation:
   a. From the **Source Variable** drop-down menu in the **Source Data** pane, select **`priceQuote1`**.
   b. Click the Import icon.

      The **Import File** dialog box is displayed.
   c. Double-click the **src** folder.
   d. Double-click the **testxml** folder.
   e. Click the `PriceQuote.xml` file.
   f. Click **Open**.

   A graphical representation of the `PriceQuote.xml` file is displayed in the **Source Data** pane.
3. In the **Result Data** pane, click the Test XQuery icon.

   The source XML data in one format is transformed by the query to XML in the target format is displayed in the **Result Data** pane, as shown in the following figure.

Figure 4-6 Result Data Pane



To learn more about the transformation occurring in the query including a walk through of the generated XQuery code, see Understanding the Transformation.

Note: In the XML document, the string: `quot` is the namespace prefix for the following namespace URI: `xmlns:quot="http://www.example.org/quote"`. To learn more about namespace declarations and how this XML data was generated, see Understanding the Transformation.

To Edit and Retest the Simple Query

This section provides the steps for editing the generated query to add a delimiter between the street, city, state, and zip code fields of the `address` element.

1. Select the **Design** tab of `myJoin.xq` file.
2. Select the link between the **zip** attribute of the **priceQuote1**/**shipAddress** node and the **quote/address** node.
3. In the **General Expression** pane of the **Target Expression** tab, add the argument: `","` between the address attribute parameters of the `concat` function to delineate between the different address fields, as shown in the following listing:

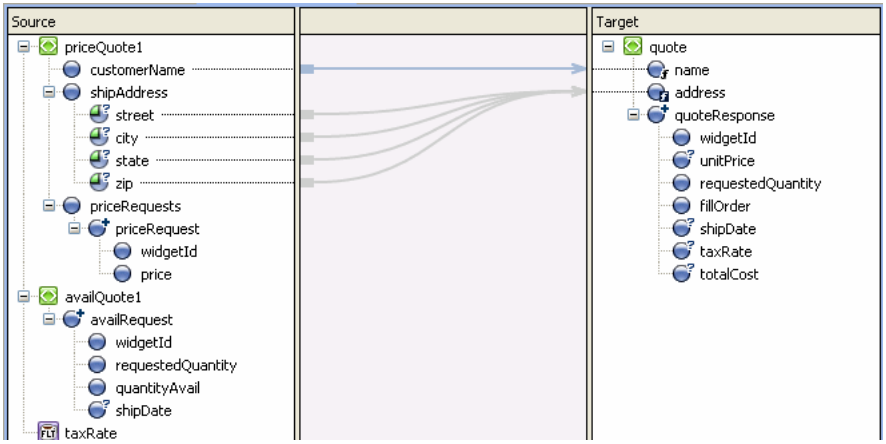
```
concat($priceQuote1/ns0:shipAddress/@street,",",
$priceQuote1/ns0:shipAddress/@city,",",
$priceQuote1/ns0:shipAddress/@state,",",
$priceQuote1/ns0:shipAddress/@zip)
```

4. Click **Apply**.

   The updated map is displayed as shown in the following figure.

   In the proceeding step, you modified the links between `shipAddress` attributes and the `address` element in the query, which causes these links to change from direct data links (represented as blue lines) to implied links (represented as light gray lines) as show in the following figure. The mapper parses the XQuery code and determines that there are implied links between the target and source elements.

Figure 4-7 Mapping Between Source and Target Panes



5. Select the **Test** tab of `myJoin.xq` file and in the **Result Data** pane click **Test**.

   In the **Result Data** pane, the resulting XML data is displayed.

   The street, city, state, and zip code fields of the `address` element will be delineated by a commas, as shown in the following listing:

   ```
   <address>12 Springs Rd,Morris Plains,nj,07960</address>
   ```

To Add an XQuery Function Call to the Query

This section provides steps for converting the `state` field to uppercase by calling a standard W3C XQuery function from the query.

1. Select the **Design** tab of `myJoin.xq` file.
2. Switch to **XQuery Transformation** perspective to view the **Target Expression** pane. To switch to **XQuery Transformation** perspective, select **Window** > **Open Perspective** > **XQuery Transformation** from the **BEA WorkSpace Studio** menu.
3. Select the link between the `state` attribute of the `shipAddress` element and the `quote/address` element.

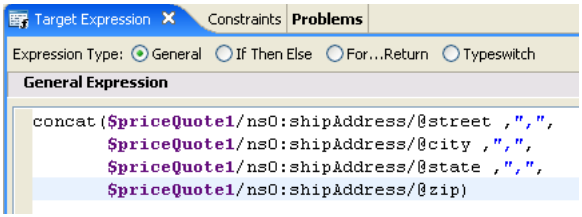
   In the **Source** pane, the `state` attribute link gets highlighted in green.

   In the **General Expression** pane of the **Target Expression** tab, the call to the `concat` function is displayed.

4. In the **General Expression** pane, find the following text:

   ```
   $priceQuote1/ns0:shipAddress/@state
   ```

Figure 4-8 Target Expression Pane - 1

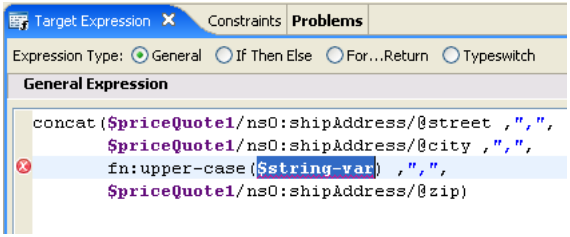


5. From the XQuery Functions folder, in the Expressions Functions palette expand **String Functions**.

6. Select the **upper-case** function, and drag-and-drop it over the `$priceQuote1/ns0:shipAddress/@state` attribute in the **General Expression** pane.

   The following is displayed in the **General Expression** pane, as shown in the following figure.

Figure 4-9 Target Expression Pane - 2

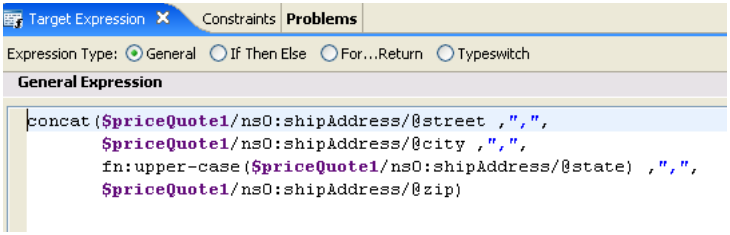


   Leave `$string-var` selected in the **General Expression** pane as shown in the preceding figure.

7. In the **Source** pane, select the **priceQuote1/shipAddress/state** node and drag-and-drop it over the `$string-var` parameter of the **General Expression** pane.

   In **General Expression** pane the following is displayed, as shown in the following figure.

Figure 4-10 Target Expression Pane - 3



8. Click **Apply** and save myJoin.xq file.
9. Select the **Test** tab.
10. Click the Test XQuery icon.

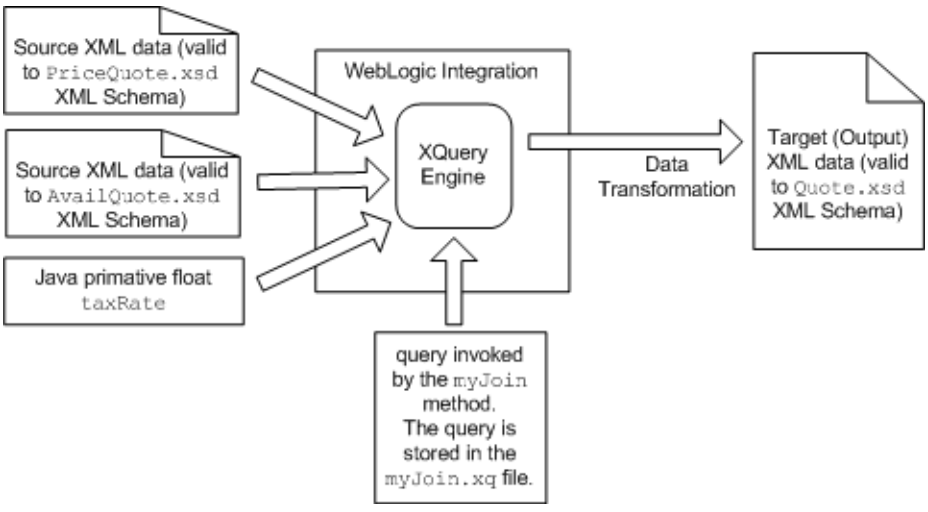
    In the XML **Source** tab of the **Result Data** pane, the state is displayed in uppercase characters, as shown in the following listing:

    ```
    <address>12 Springs Rd,Morris Plains,NJ,07960</address>
    ```

Step 4: Mapping Repeating Elements—Creating a Join

In this step, you will add additional mappings to the existing query. In the previous sections, you mapped some data from the source type defined by the `PriceQuote.xsd` XML Schema to the target type defined by the `Quote.xsd` XML Schema. In this section, you will map additional data from the source types (defined by the `PriceQuote.xsd` XML Schema, the `AvailQuote.xsd` XML Schema, and the Java float primitive: `taxRate`) to the target type (defined by the `Quote.xsd` XML Schema) as shown in the following figure.

Figure 5-1 Adding Data from Source Types



Mappings created in this section will create a join between repeating elements in the source and target XML Schemas. Complete the following tasks to create, test, and alter the join:

- Create a User-Defined Java Method to Invoke From the Join Query
- To Join Two Sets of Repeating Elements
- Add Links to Populate the quoteResponse Element
- Call the calculateTotalPrice User Method From the Query
- To View the Generated Query
- To Test the Query
- Create an Instance of the MyTutorialJoin Control
- Edit the Node That Invokes the Transformation
- To Run the Business Process

Create a User-Defined Java Method to Invoke From the Join Query

In this task, you will create a user-defined Java method in the **MyTutorialJoin** Transformation file that calculates the total price of the widgets requested including tax. In "Call the calculateTotalPrice User Method From the Query" on page 5-9, you will change the query to invoke this method.

1. In the **Navigator** pane, browse to the **src/requestquote** folder, and select `MyTutorialJoin.java`.
2. Right-click on the Source of the **MyTutorialJoin** Transformation .
3. From the shortcut menu, select **Transform** > **Add User Method**.

   A User method is created in the **MyTutorialJoin** Transformation file.
4. A new method called `newusermethod1` will be added in the Source. Rename this method to `calculateTotalPrice`.
5. Edit the MyTutorialJoin Transformation file and replace the following generated `calculateTotalPrice` Java method:

```
 public java.lang.String calculateTotalPrice() {
return "";
}
```

With the following `calculateTotalPrice` Java method:

```
public static float calculateTotalPrice(float taxRate, int quantity,
float price, boolean fillOrder)
{
float totalTax, costNoTax, totalCost;
if (fillOrder)
{
   // Calculate the total tax
   totalTax = taxRate * quantity * price;
   // Calculate the total cost without tax
   costNoTax = quantity * price;
   // Add the tax and the cost to get the total cost
   totalCost = totalTax + costNoTax;
}
else
{
    totalCost = 0;
}
return totalCost;
   }
```

Note: Ensure you modify the return type of the `calculateTotalPrice` function from `String` to `float`.

6. Save all the files in this application. From the **BEA WorkSpace Studio** menu bar, choose **File** > **Save All**.

   Note: XQuery Mapper executes the user defined java method used in XQuery only if the java method is static.

To Join Two Sets of Repeating Elements

1. View `myJoin.xq` in the **Design** view:

   a. In the **Navigator** pane, double-click `Tutorial_Process_Application_Web\src\requestquote\myJoin.xq` and select the **Design**.

2. In the **Source** pane, collapse the **shipAddress** node.

3. From the **Source** pane, drag-and-drop the **priceQuote1\priceRequests\priceRequest** node onto the **quote\quoteResponse** node in the **Target** pane.

   These nodes are both repeating nodes. A repeating node means more than one instance of this node can be specified. The + symbol to the right of the node indicates these nodes are repeating nodes.

   Warning: You must select the **priceRequest** node and not the **priceRequests** node.

   A dashed line linking the two repeating nodes is displayed, as shown in the following figure.

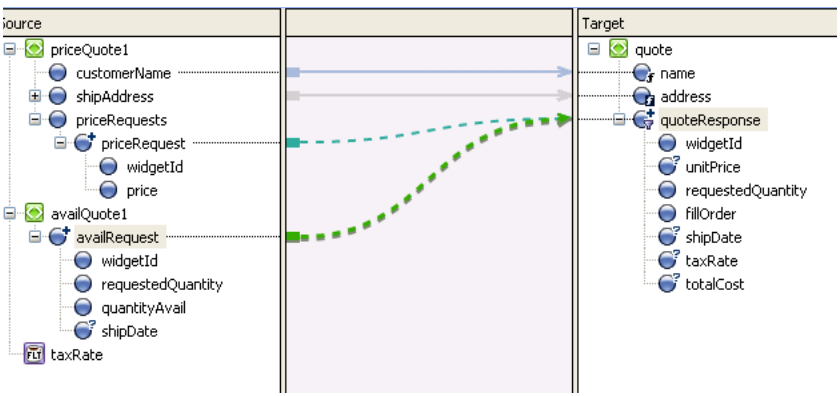
   The dashed line with short dashes represents a structural link—a link between two parent structures that does not map data directly.

   To learn more about XML repeating nodes, see "Understanding XML Repeating Nodes" on page 6-5.

4. From the **Source** pane, drag-and-drop the **availQuote1\availRequest** node onto the **quote\quoteResponse** node in the **Target** pane.

   A dashed line linking the two repeating elements is displayed, as shown in the following figure.

Figure 5-2 Joining Two Sets of Repeating Elements



5. Select the **Source** tab to view the changes to the query.

   The following query is displayed in **Source**:

```
declare namespace xf =
"http://tempuri.org/Tutorial_Project_Web/src/requestquote/myJoin/";

declare namespace ns0 = "http://www.example.org/price";
```

```
declare namespace ns1 = "http://www.example.org/avail";
declare namespace ns2 = "http://www.example.org/quote";
declare function xf:myJoin($priceQuote1 as element(ns0:priceQuote),
    $availQuote1 as element(ns1:availQuote),
    $taxRate as xs:float)
    as element(ns2:quote) {
        <ns2:quote>
            <name>{ data($priceQuote1/ns0:customerName) }</name>
           <address>{ concat($priceQuote1/ns0:shipAddress/@street ,",",
$priceQuote1/ns0:shipAddress/@city ,",",
fn:upper-case($priceQuote1/ns0:shipAddress/@state) ,
",",$priceQuote1/ns0:shipAddress/@zip) }</address>
            {
for
$priceRequest in $priceQuote1/ns0:priceRequests/ns0:priceRequest,
$availRequest in $availQuote1/ns1:availRequest
return
                    <quoteResponse/>
            }
        </ns2:quote>
};

declare variable $priceQuote1 as element(ns0:priceQuote) external;
declare variable $availQuote1 as element(ns1:availQuote) external;
declare variable $taxRate as xs:float external;

xf:myJoin($priceQuote1,
    $availQuote1,
    $taxRate)
```

In the preceding query, there are no data links between the children of the repeating nodes, so the `quoteResponse` element is empty. (The string: `<quoteResponse/>` is an empty node.)

The structural links between the repeating nodes generates the `for` loop which is shown in bold in the preceding query listing. This XQuery `for` loop iterates through the set of `priceRequest` and `availReqest` repeating elements. For example, if the source XML data to this query contains three instances of the `priceRequest` element and three instances of the `availRequest` element, the `for` loop would execute a total of nine times with the following combinations:

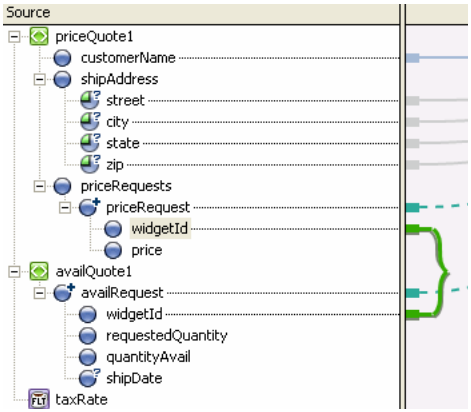
- The first instance of the `priceRequest` element with the first instance of `availRequest` element.
- The first instance of the `priceRequest` element with the second instance of `availRequest` element.
- The first instance of the `priceRequest` element with the third instance of `availRequest` element.
- The second instance of the `priceRequest` element with the first instance of `availRequest` element.
- The second instance of the `priceRequest` element with the second instance of `availRequest` element.
- The second instance of the `priceRequest` element with the third instance of `availRequest` element.
- The third instance of the `priceRequest` element with the first instance of `availRequest` element.
- The third instance of the `priceRequest` element with the second instance of `availRequest` element.
- The third instance of the `priceRequest` element with the third instance of `availRequest` element.

For some transformations, you may want the query to generate all the possible combinations but for others, you may want to constrain the combinations as described in the following steps.

6. Select the **Design** tab.
7. From the **Source** pane, drag-and-drop the **priceQuote1/priceRequests/priceRequest/widgetId** node onto the **availQuote1/availRequest/widgetId** node. Both of these elements are in the **Source** pane.

   A line between the two **widgetId** nodes is displayed, as shown in the following figure.

Figure 5-3 Link Between Two WidgetId Nodes



8. View the changes to the query by clicking the **Source** tab.

   The following query is displayed:

```
declare namespace xf =
"http://tempuri.org/Tutorial_Process_Application_Web/src/requestquote/m
yJoin/";

declare namespace ns0 = "http://www.example.org/price";
declare namespace ns1 = "http://www.example.org/avail";
declare namespace ns2 = "http://www.example.org/quote";
declare function xf:myJoin($priceQuote1 as element(ns0:priceQuote),
   $availQuote1 as element(ns1:availQuote),
   $taxRate as xs:float)
   as element(ns2:quote) {
<ns2:quote>

                <name>{ data($priceQuote1/ns0:customerName) }</name>
               <address>{ concat($priceQuote1/ns0:shipAddress/@street
   ,",",               $priceQuote1/ns0:shipAddress/@city ,",",
                 fn:upper-case($priceQuote1/ns0:shipAddress/@state) ,
   ",",               $priceQuote1/ns0:shipAddress/@zip) }</address>

               {

   for $priceRequest in $priceQuote1/ns0:priceRequests/ns0:priceRequest,

    $availRequest in $availQuote1/ns1:availRequest

where

data($priceRequest/ns0:widgetId) = data($availRequest/ns1:widgetId)

return

                    <quoteResponse/>
```

```
            }
        </ns2:quote>
};


declare variable $priceQuote1 as element(ns0:priceQuote) external;
declare variable $availQuote1 as element(ns1:availQuote) external;
declare variable $taxRate as xs:float external;

xf:myJoin($priceQuote1,
    $availQuote1,
    $taxRate)
```

The link between the **widgetId** nodes generates the `where` clause in the `for` loop, as shown in bold in the preceding query listing. This `where` clause constrains or limits the output of the `for` loop. Specifically, the `where` clause specifies that if the expression in the `where` clause is true, the `for` loop will output the contents of the `return`. For this example, if the **widgetId** of the `availRequest` element is equal to the **widgetId** of the `priceQuest` element, the following XML data is returned:

```
<quoteResponse/>
```

An empty `quoteReponse` element isn't very useful. In the following task: "Add Links to Populate the quoteResponse Element" on page 5-8, you will add data links that will populate the `quoteResponse` element.

Add Links to Populate the quoteResponse Element

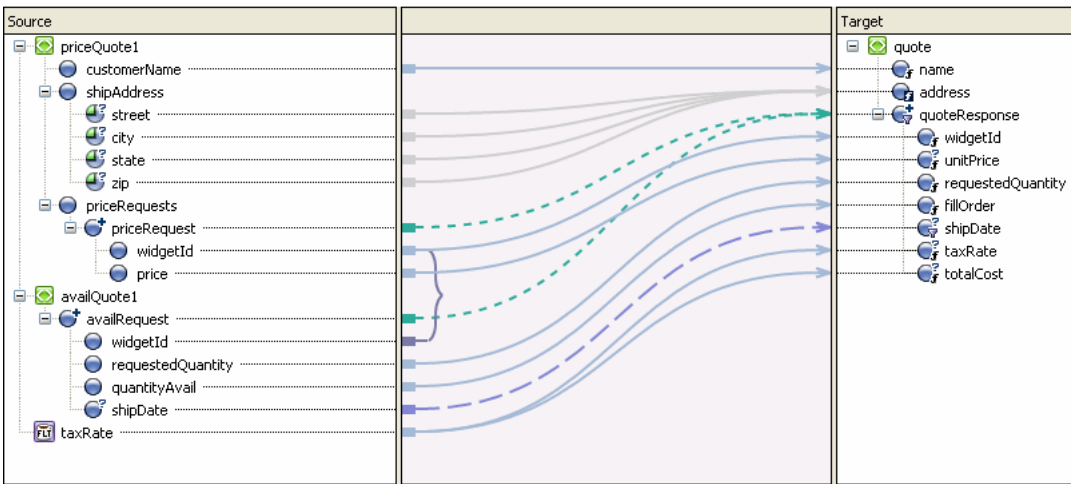
1. Select the **Design** tab.
2. From the **Source** pane, drag-and-drop the **priceQuote1/priceRequests/priceRequest/widgetId** node onto the **quote/quoteResponse/widgetId** node in the **Target** pane.
3. From the **Source** pane, drag-and-drop the **priceQuote1/priceRequests/priceRequest/price** node onto the **quote/quoteResponse/unitPrice** node in the **Target** pane.
4. From the **Source** pane, drag-and-drop the **availQuote1/availRequest/requestedQuantity** node onto the **quote/quoteResponse/requestedQuantity** node in the **Target** pane.
5. From the **Source** pane, drag-and-drop the **availQuote1/availRequest/quantityAvail** node onto the **quote/quoteResponse/fillOrder** node in the **Target** pane.
6. From the **Source** pane, drag-and-drop the **availQuote1/availRequest/shipDate** node onto the **quote/quoteResponse/shipDate** node in the **Target** pane.

7. From the **Source** pane, drag-and-drop the **taxRate** Java primitive onto the **quote**/**quoteResponse**/**taxRate** node in the **Target** pane.
8. From the **Source** pane, drag-and-drop the **taxRate** Java primitive onto the **quote**/**quoteResponse**/**totalCost** node in the **Target** pane.

   Note: In the next section, to calculate the total cost of the order, you will edit the link between the **taxRate** Java primitive and the **quote**/**quoteResponse**/**totalCost** node.

   In the **Design** view, the following links are displayed, as shown in the following figure.

Figure 5-4 Linking Source and Target Panes



9. Save all the files in this application. From the **BEA WorkSpace Studio** menu bar, choose **File** > **Save All**.

Call the calculateTotalPrice User Method From the Query

1. Select the **Design** tab.
2. Select the link between the **taxRate** Java primitive and the **quote/quoteResponse/totalCost** node.
3. In the **Expression Functions** Palette, find the **User Functions** folder.
4. In the **User Functions** folder, select the **calculateTotalPrice** function, and drag-and-drop it into the **General Expression** pane.

5. In the **Source** pane, select the **taxRate** node and drag-and-drop it onto the **$float-var** parameter of the **General Expression** pane.

   In the **General Expression** pane, the default argument: **$float-var** is replaced with the **$taxRate** argument and the next argument becomes selected.

   Select `$int-var` in the **General Expression** pane.

6. In the **Source** pane, select **availQuote1/availRequest/requestedQuantity** and drag-and-drop it onto the selected **$int-var** argument in the **General Expression** pane.

   In the **General Expression** pane, the default argument: **$int-var** is replaced with the **$availRequest/ns1:requestedQuantity** argument and the next argument becomes selected.

   Select `$float-var` in the **General Expression** pane.

7. In the **Source** pane, select **priceQuote1/priceRequests/priceRequest/price** and drag-and-drop it onto the selected **$float-var** argument in the **General Expression** pane.

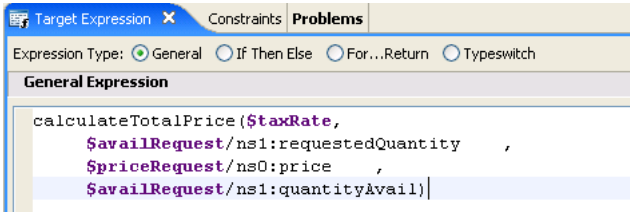
   In the **General Expression** pane, the default argument: **$float-var** is replaced with the **$priceRequest/ns0:price** argument and the next argument becomes selected.

   Select `$boolean-var` in the **General Expression** pane.

8. In the **Source** pane, select **availQuote1/availRequest/quantityAvail** and drag-and-drop it onto the selected **$boolean-var** argument in the **General Expression** pane.

   In the **General Expression** pane, the default argument: **$boolean-var** is replaced with the **$availRequest/ns1:quantityAvail** argument, as shown in Figure 5-5.

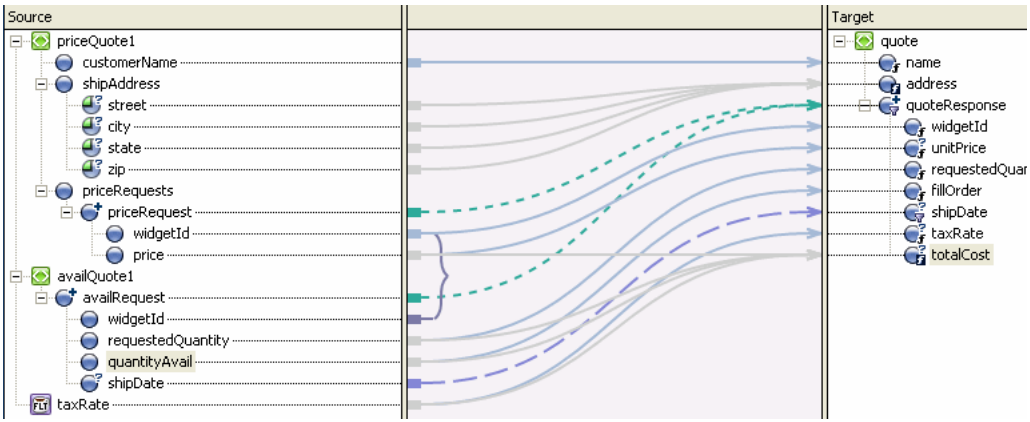
Figure 5-5 General Expression pane



9. Click **Apply**.

   In the **Design** view, the following is displayed, as shown in Figure 5-6.

Figure 5-6 Link Between Source and Target Panes



10. Save all the files in this application. From **the BEA WorkSpace Studio** menu bar, choose **File** > **Save All**.

To View the Generated Query

1. Select the **Source** tab.

   The following query is displayed in **Source** view:

```
declare namespace xf =
"http://tempuri.org/Tutorial_Process_Application_Web/src/requestquote/m
yJoin/";
declare namespace ns0 = "http://www.example.org/price";
declare namespace ns1 = "http://www.example.org/avail";
declare namespace ns2 = "http://www.example.org/quote";
declare function xf:myJoin($priceQuote1 as element(ns0:priceQuote),
    $availQuote1 as element(ns1:availQuote),
    $taxRate as xs:float) as element(ns2:quote)
    {
      <ns2:quote>
{
<name>{ data($priceQuote1/ns0:customerName) }</name>

           <address>{ concat($priceQuote1/ns0:shipAddress/@street ,",",
$priceQuote1/ns0:shipAddress/@city ,",",
fn:upper-case($priceQuote1/ns0:shipAddress/@state) , ",",
$priceQuote1/ns0:shipAddress/@zip) }</address>

             {

                 for $priceRequest in
                 $priceQuote1/ns0:priceRequests/ns0:priceRequest,
```

```
                    $availRequest in $availQuote1/ns1:availRequest
                where data($priceRequest/ns0:widgetId) =
                data($availRequest/ns1:widgetId)

                return
            <quoteResponse>
                <widgetId>{ data($priceRequest/ns0:widgetId) }</widgetId>

                <unitPrice>{ data($priceRequest/ns0:price) }</unitPrice>

                <requestedQuantity>{
                data($availRequest/ns1:requestedQuantity)
}</requestedQuantity>
<fillOrder>{ data($availRequest/ns1:quantityAvail) }</fillOrder>


                {
                    for $shipDate in $availRequest/ns1:shipDate
                    return
                        <shipDate>{ data($shipDate) }</shipDate>
                }
                <taxRate>{ $taxRate }</taxRate>
                <totalCost>{ calculateTotalPrice($taxRate,
                    $availRequest/ns1:requestedQuantity,
                    $priceRequest/ns0:price,
                    $availRequest/ns1:quantityAvail) }</totalCost>
            </quoteResponse>
}
</ns2:quote>
```

The links added in the preceding task generate the additional XQuery source code listed between the `<quoteResponse>` and `</quoteResponse>` tags highlighted in bold in the preceding query listing.

To Test the Query

1. Select the **Test View** tab.

   There are three import parameters to the myJoin Transformation method: $priceQuote1, $availQuote1, and $taxRate. In the task: To Test a Simple Query, you imported `PriceQuote.xml` as source data for the `$priceQuote1` parameter.

2. Import `AvailQuote.xml` for the source parameter: `$availQuote1`:

   a. From the drop-down menu in the **Source Data** pane, select **availQuote1**.

   b. Click **Import...**

   The **Import File** to Test dialog box is displayed.

c. Double-click the **src** folder.

d. Double-click the **testxml** folder.

e. Double-click the **AvailQuote.xml** file.

A graphical representation of the `AvailQuote.xml` file appears in the **Source Data** pane.

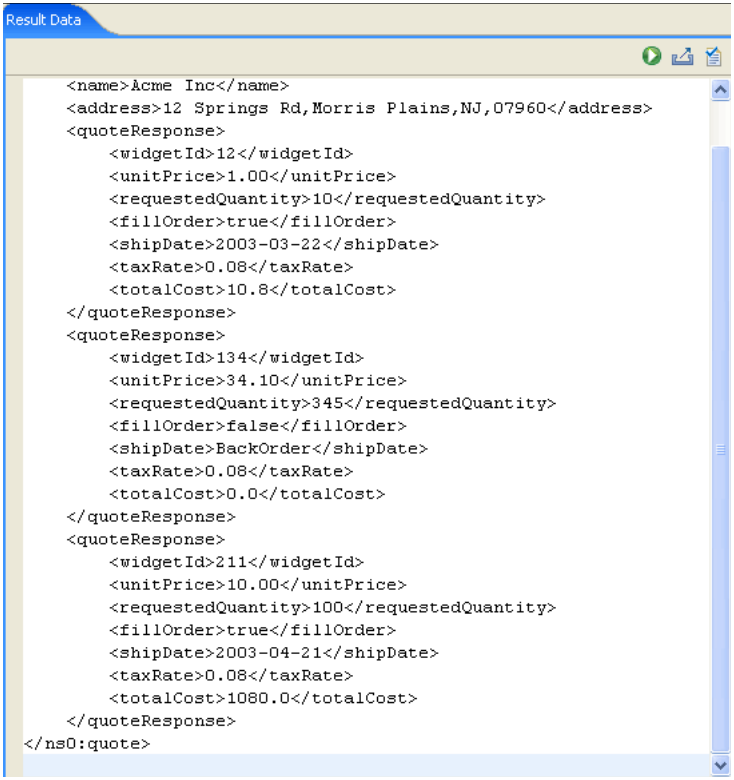
3. From the drop-down menu in the Source Data pane, select **taxRate** .

4. In the Node Value field of the **taxRate** node, double-click the existing value, enter "0.08", and then press Enter on your keyboard.

5. In the Result Data, pane click **Test**.

   The query is run with the test XML data. A graphical representation of the resulting XML data is shown in the XML Design View of the Result Data pane, as shown in Figure 5-7.

Figure 5-7 Result Data Pane



This query joins the two sets of source repeating elements (availRequest and priceRequest) to a single repeating element (quoteResponse).

6. To check that the resulting XML data from the query is valid against the associated XML Schema, in the Result Data pane click **Validate**.

   The Output tab will show whether the XML data is valid to the target XML Schema. In this example, the resulting XML data is checked against the XML Schema in the `Quote.xsd` file.

Create an Instance of the MyTutorialJoin Control

In this task, you create an instance of the `MyTutorialJoin.java` control.

Note: Switch from XQuery Transformation perspective to Process perspective to create a control. Click **Window > Open Perspective > Other > Process Perspective**.

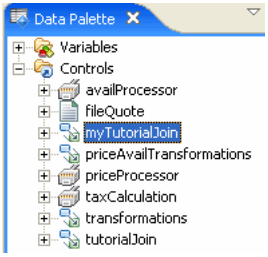
1. View the RequestQuote business process in the **Design** view:

   a. In the **Package Explorer** pane, navigate to `Tutorial_Process_Application_Web\src\requestquote\RequestQuote.java` and double-click the **RequestQuote.java.**

   b. Drag-and-drop the **myTutorialJoin.java** transformation file from the **src/requestquote** folder to the **Controls** folder in the **DataPalette**.

   An instance called **MyTutorialJoin.java** is created in your project and displayed in the **Controls** pane as shown by the following figure:

Figure 5-8 Instance of MyTutorialJoin.java



Edit the Node That Invokes the Transformation

In this task, you edit the **Combine Price and Avail Quotes** node in the RequestQuote business process and change the instance that gets invoked by this node from an instance of the `TutorialJoin.java` to an instance of the `MyTutorialJoin.java`. Additionally, you change the design of the **Combine Price and Avail Quotes** node to call the `myJoin()` method on the

MyTutorialJoin control. The `myJoin()` method combines the data returned to your business process from different systems creating a single XML response document (quote) that is subsequently returned to the business process's client.

1. In the RequestQuote business process, double-click the **Combine Price and Avail Quotes** node to open its node builder.

   The node builder opens on the **General Settings** pane.

2. From the drop-down menu in the **Control** field select **myTutorialJoin**.

3. Select `QuoteDocument myJoin()` from the **Method** field.

4. Click **Send Data** to open the second pane of the node builder.

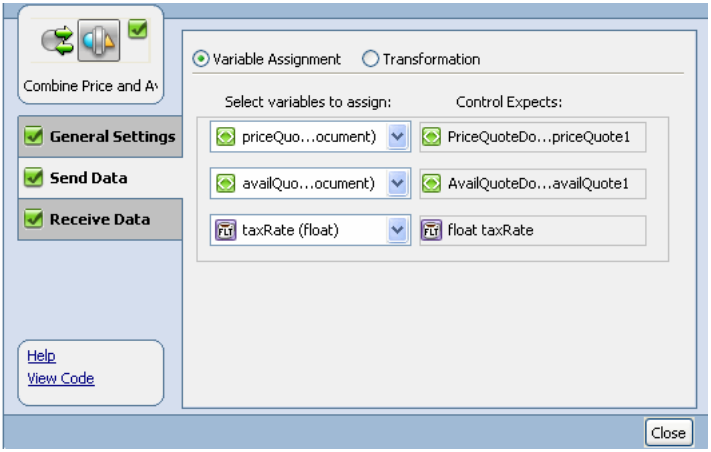
   The **Select variables to assign** fields are populated with default variables. The data types match the data type expected in the source parameters to the `myJoin`() method as shown in the following list:

   priceQuote holds the price quote data, which is returned from the PriceProcessor service in the **For Each** loop in your business process.

   availQuote holds the availability quote data, which is returned from the AvailProcessor service in the **For Each** loop in your business process.

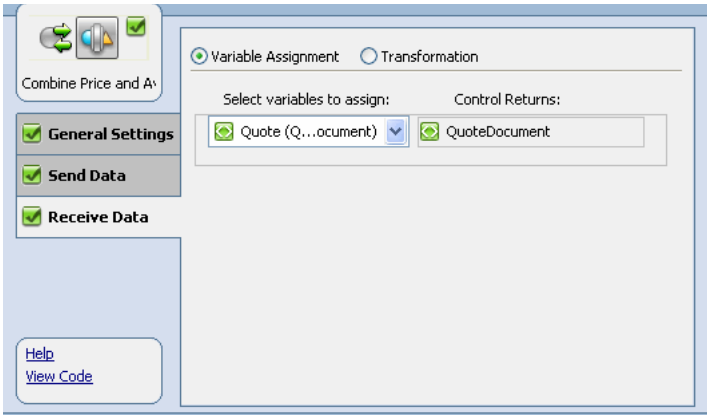
   taxRate holds the rate of sales tax applied to the quote, based on the shipping address, which is returned to your business process from the taxCalculation service.

   The **Control Expects** fields are populated with the data type expected by the `myJoin()` method on the **MyTutorialJoin** control, as shown in the following figure.



5. Click **Receive Data** to open the third pane of the node builder.

   On the **Receive Data** tab, the **Select variables to assign** field is populated with the default variable: **Quote**. The data type matches the data type expected in the target parameter to the `myJoin`() method. The **Control Returns** field is populated with the data type returned by the `myJoin`() method: **QuoteDocument**, as shown in the following figure.



6. In the node builder, click **Close** save your specifications and close the node builder.

7. Save all the files in this application, including the RequestQuote business process. From the **BEA WorkSpace Studio** menu bar, choose **File > Save All**.

To Run the Business Process

Earlier in this tutorial, you entered the XML data that is run against the query. During run time, the business process builds the XML data and passes it to the query that was built in this tutorial. To run the business process and invoke the query, follow the instructions in Step 12: Run the RequestQuote Business Process in the *Tutorial: Building Your First Business Process*.

Understanding the Concepts

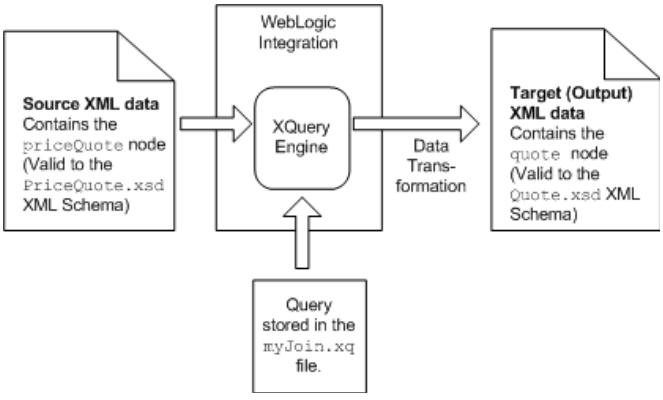
This section is optional and provides detailed conceptual information about the following topics:

- Understanding the Transformation
- Understanding XML Repeating Nodes

Understanding the Transformation

The transformation occurring in the query built in "Step 3: Mapping Elements and Attributes" on page 4-1 is shown in the following figure:

Figure 6-1 Mapping Elements and Attributes



The query generated in "To Map Attributes of an Element to Single Element" on page 4-4 is shown in the following listing:

```
1. (:: pragma bea:dtfFile-class type="requestquote.MyTutorialJoin" ::)
2. declare namespace xf =
"http://tempuri.org/Tutorial_Process_Application_Web/src/requestquote/m
yJoin/";
   3. declare namespace ns0 = "http://www.example.org/price";
   4. declare namespace ns1 = "http://www.example.org/avail";
   5. declare namespace ns2 = "http://www.example.org/quote";


   6. declare function xf:myJoin($priceQuote1 as
   element(ns0:priceQuote),
   7. $availQuote1 as element(ns1:availQuote),
   8. $taxRate as xs:float)
9. as element(ns2:quote) {
10. <ns2:quote>
   11. <name>{ data($priceQuote1/ns0:customerName) }</name>
   12. <address>{ concat($priceQuote1/ns0:shipAddress/@street ,
   $priceQuote1/ns0:shipAddress/@city ,
   $priceQuote1/ns0:shipAddress/@state ,
   $priceQuote1/ns0:shipAddress/@zip) }</address>
13. </ns2:quote>
14. };


   15. declare variable $priceQuote1 as element(ns0:priceQuote) external;
   16. declare variable $availQuote1 as element(ns1:availQuote) external;
   17. declare variable $taxRate as xs:float external;


   18. xf:myJoin($priceQuote1,$availQuote1,$taxRate)
```

Note: The line numbers are provided are for your reference.

The second line of this query represents the namespace of the XQuery function. The namespace declarations are part of the query prolog. For each namespace in the source and target XML Schema, the mapper generates a namespace declaration. For example, the mapper generates the

namespace declaration: `ns0` for the namespace URI (`http://www.example.org/price`) defined in the XML Schema of the `PriceQuote.xsd` file. Namespaces are used to uniquely distinguish elements in XML Schema from elements in another XML Schema.

The following steps describe the transformation that occurs when the source XML data is run against the preceding query:

1. The tenth line of the query is shown in the following listing:

   ```
   <ns2:quote>
   ```

   This line of the query becomes the first line of the XML output, as shown in the following listing:

   ```
   <quot:quote xmlns:quot="http://www.example.org/quote">
   ```

   During the transformation, the namespace prefix for the `quote` element changes. In the query, the namespace prefix associated with `http://www.example.org/quote` namespace URI is `ns2`. However, in the resulting XML data, the namespace prefix generated for the `http://www.example.org/quote` namespace URI is `quot`. This namespace declaration is highlighted in bold in the preceding listing.

2. The eleventh line of the query is shown in the following listing:

   ```
   <name>{data($priceQuoteDoc/ns0:customerName)}</name>
   ```

   This line of the query transforms the `customerName` element of the `priceQuote` element to the `name` element of the `quote` element.

   The following steps describe the transformation that occurs on this line of XQuery code:

   a. The `<name>` and `</name>` tags transform directly to XML output.

   b. Characters between curly braces `{}` are interpreted in a special way by the XQuery engine. That is, characters surrounded by curly braces are not transformed directly into XML. Specifically, in this example, the curly braces surrounding the data method specify that the `data` function of the XQuery language should be executed.

      The `data` function returns the value of the passed in XML node. For this example, the argument to the `data` function is the following XPath expression: `$priceQuoteDoc/ns0:customerName`. The `$priceQuoteDoc` variable contains the contents of the `priceQuote` element, including its subelements. This XPath expression returns the `customerName` node of the `priceQuote` element. (The `/` XPath operator delineates parent nodes from child nodes.)

      The XQuery `data` function takes `customerName` node and returns the value of the node, the string: `Acme Inc`. This string is placed between the `<name>` and `</name>`

tags resulting in the following line of output XML data, as shown in the following listing:

```
<name>Acme Inc</name>
```

3. The twelfth line in the query is shown in the following listing:

```
<address>
                {
                    concat($priceQuote1/ns1:shipAddress/@street ,",",
                    $priceQuote1/ns1:shipAddress/@city ,",",
                  fn:upper-case($priceQuote1/ns1:shipAddress/@state) ,",",
                    $priceQuote1/ns1:shipAddress/@zip)
                }
</address>
```

The following steps describe the transformation that occurs on this line of XQuery code.

a. The `<address>` and `</address>` tags transform directly to XML output.

b. Characters between curly braces `{}` are interpreted in a special way by the XQuery engine. That is, characters surrounded by curly braces are not transformed directly into XML. Specifically, in this example, the curly braces surrounding the data method specify that the `data` function of the XQuery language should be executed.

c. The `concat` function takes the values of all its arguments, concatenates these values together, and returns them as a string. For this example, the `concat` function takes the values of the all the XPath expressions and concatenates them together in one address string. Additionally, all the arguments in this `concat` function are XPath expressions that return the value of specified attribute, as shown in Table 6-1.

Table 6-1 Arguments: XPath Expressions

The Following XPath Expression	Returns	The String
`$priceQuote1/ns0:shipAddress/@ns0:street`	The value of the `street` attribute of the `shipAddress` element.	`12 Springs Rd`
`$priceQuote1/ns0:shipAddress/@ns0:city`	The value of the `city` attribute of the `shipAddress` element	`Morris Plains`

Table 6-1 Arguments: XPath Expressions

`$priceQuote1/ns0:shipAddress/@ns0:state`	The value of the `state` attribute of the `shipAddress` element.	`nj`
`$priceQuote1/ns0:shipAddress/@ns0:zip`	The value of the `zip` attribute of the `shipAddress` element.	`07960`

The return string of the `concat` function is placed between the `<address>` and `<address>` tags resulting in the following line of XML data, as shown in the following listing:

```
<address>12 Springs RdMorris Plainsnj07960</address>
```

4. The last line of the query is shown in the following listing:

```
</ns2:quote>
```

The last line of the query becomes the last line of the XML output, as shown in the following listing:

```
</quot:quote>
```

The resulting `address` element has no delimiter between the street, city, state, and zip code fields, making the address difficult to read and parse. For instructions on adding delimiters to this query, return to "To Edit and Retest the Simple Query" on page 4-7 in the main section of this tutorial.

Understanding XML Repeating Nodes

A repeating node means that more than one instance of this node can be specified. For example, in the following XML data there are three instances of the **priceRequest** node, as shown in the following listing:

```
<?xml version="1.0"?>
<priceQuote xmlns="http://www.example.org/price">
    <customerName>Acme Inc</customerName>
    <shipAddress street="12 Springs Rd" city="Morris Plains" state="nj"
zip="07960"/>
    <priceRequests>
        <priceRequest>
            <widgetId>12</widgetId>
            <price>1.00</price>
        </priceRequest>
        <priceRequest>
```

```
            <widgetId>134</widgetId>
            <price>34.10</price>
        </priceRequest>
        <priceRequest>
            <widgetId>211</widgetId>
            <price>10.00</price>
        </priceRequest>
    </priceRequests>
</priceQuote>
```

A segment of the XML Schema for the preceding XML data is shown in the following listing:

```
<?xml version="1.0"?>
<xsd:schema . . . >
. . .
    <xsd:element name="widgetId" type="xsd:integer"/>
    <xsd:element name="price" type="xsd:float"/>
    <xsd:element name="priceRequest">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="pri:widgetId"/>
                <xsd:element ref="pri:price"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="priceRequests">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="pri:priceRequest" minOccurs="1"
maxOccurs="10"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
. . .
    <xsd:element name="priceQuote">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="pri:customerName" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="pri:shipAddress" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="pri:priceRequests"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

The `minOccurs="1"` and `maxOccurs="10"` settings, in the definition of the `priceRequest` element (highlighted in bold in the preceding listing), specify that there can be one to ten instances of the `priceRequest` element. This defines `priceQuote` as a repeating element.

To View the Full listing of the XML Schema, Open the PriceQuote.xsd file

1. In the **Package Explorer** pane, expand the expand the **Tutorial_Process_Application_Utility/schemas** folder.
2. Double-click the **PriceQuote.xsd** icon.

   The `PriceQuote.xsd` file is displayed.
3. Return to the **Design** view of the `myJoin.xq` file:

   a. In the **Package Explorer** pane, double-click `Tutorial_Process_Application_Web\src\requestquote\myJoin.xq`

   b. Select the **Design** tab.