Using Microsoft .NET in WebSphere Message Broker V8: Part 1: Using the .NETCompute node sample

Matthew Golby-Kirk
Ben Thompson

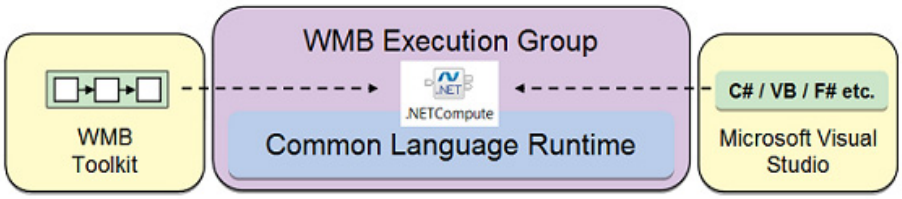
December 21, 2011

This series of four tutorials demonstrates the new support for Microsoft .NET in WebSphere Message Broker V8. Part 1 shows you how to use the .NETCompute node to filter, modify, and create messages, and provides a sample scenario along with explanatory C# code snippets. Readers should be familiar with either Microsoft .NET or WebSphere Message Broker but need not be familiar with both.

View more content in this series

Before you start

IBM® WebSphere® Message Broker V8 (hereafter called Message Broker) provides the capability to integrate with existing Microsoft® .NET® Framework (hereafter called .NET) applications. You can do this integration by wiring the new Message Broker .NETCompute node into a message flow, or by calling a .NET application from an ESQL Compute node.



About this tutorial series

This series of four tutorials shows you how to use the new Message Broker .NETCompute node integration capability. Each tutorial shows you how to create C# code in Microsoft Visual Studio 2010 using an embedded template, which is provided by an installation of the WebSphere Message Broker Toolkit. The four tutorials explore the following topics:

1. **Using the .NETCompute node sample**
2. Using the .NETCompute node to integrate with Microsoft Word
3. Using the .NETCompute node to integrate with Microsoft Excel
4. Using the .NETCompute node for exception handling

About this tutorial

Youtube tutorial: Integrating Microsoft .NET code in a WebSphere Message Broker message flow

This five-minute youtube tutorial shows you how simple it is to use WebSphere Message Broker V8 to build a message flow that includes Microsoft .NET code. Microsoft Visual Studio is used to build .NET code in C#, which is then integrated into a message flow using Message Broker and an HTTP RESTful interface.

This .NETCompute Node sample filters, modifies, and transforms messages using code written in C#. You can use the .NETCompute node on Microsoft Windows® brokers to construct output messages and interact with the Microsoft .NET framework (.NET) or Component Object Model (COM) applications.

WebSphere Message Broker enables you to host and run .NET code inside an execution group. The new .NETCompute node routes or transforms messages by using any Common Language Runtime (CLR) compliant .NET programming language, such as C#, Visual Basic (VB), F#, or C ++/Common Language Infrastructure (CLI). This tutorial describes the new .NET API provided by WebSphere Message Broker, which enables .NET developers to interact with Message Broker's logical tree.

Prerequisites and system requirements

This tutorial is written for WebSphere Message Broker programmers who want to learn about the new .NETCompute node, and for .NET programmers who want to learn about using WebSphere Message Broker. If you have a general familiarity with C# or with Message Broker, then you should find the tutorial relatively easy to complete.

To build and execute the example in this tutorial, you will need:

- A Windows installation that includes Microsoft .NET Framework V4
- WebSphere Message Broker (Toolkit and Runtime) V8
- Microsoft Visual Studio 2010 (Express Edition or Professional Edition) to write and build the required C# code

Sample files

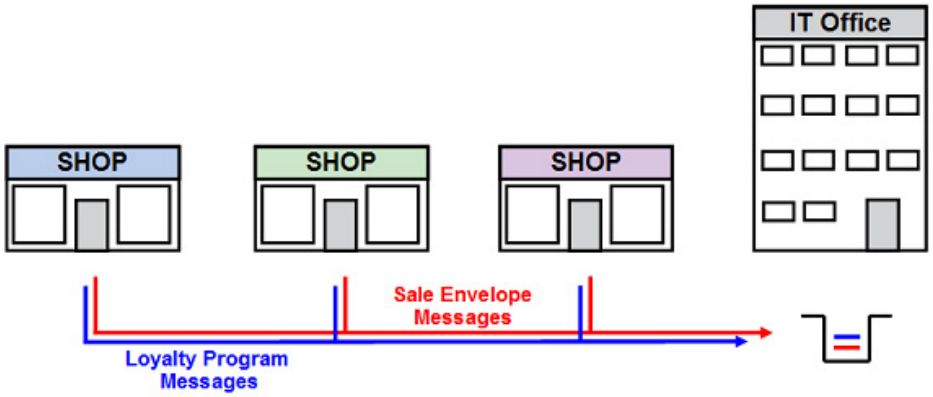
You can import the message flow and test data for this tutorial from the Message Broker Samples Gallery, which is available as part of your Message Broker installation. No further downloads are required in order to complete this tutorial.

Introduction

The message flow for the tutorial is imported from the Message Broker Samples Gallery. Instructions below then explain how to create the C# code required by the .NETCompute node, and how to do deployment and testing. This tutorial provides more detailed explanations for the sample than in the Message Broker documentation, and also uses a configurable service in conjunction with the AppDomainName property of the .NETCompute node to reflect real-life production use.

Scenario description

A retail company has several stores at separate locations within a city. The stores complete sales transactions throughout the day, and each transaction generates an XML message that is routed to an input queue at a central IT office:

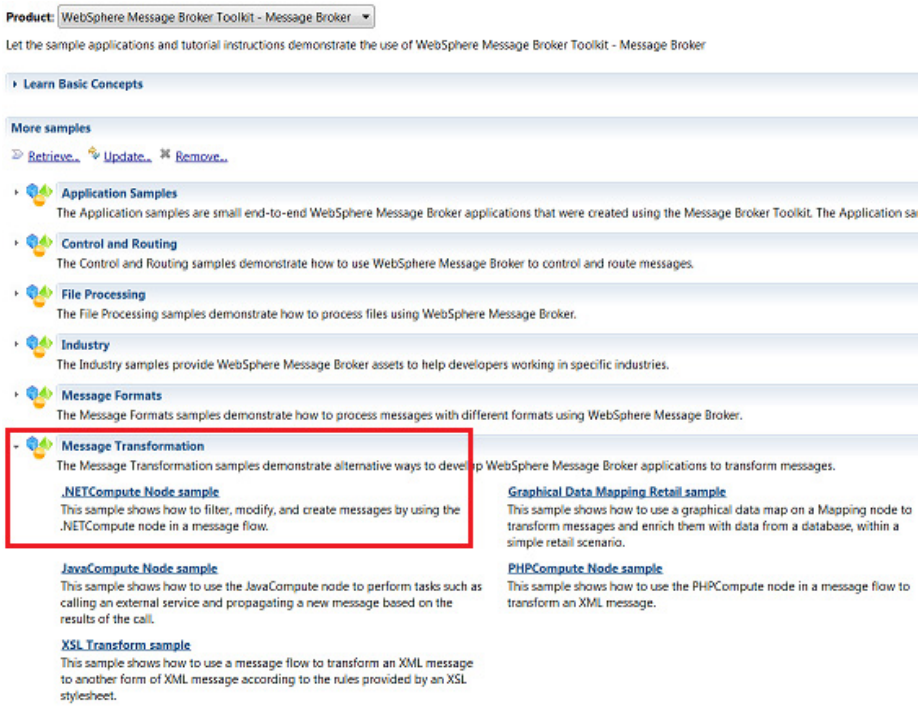


The stores are rolling out a customer loyalty program. For every customer who registers in the loyalty program, an XML message in a different format containing the customer's personal details is sent to the same input queue. The company has decided to use Message Broker to process the messages. The routing and transformations used in the solution demonstrate the capabilities of the Message Broker .NETCompute node.

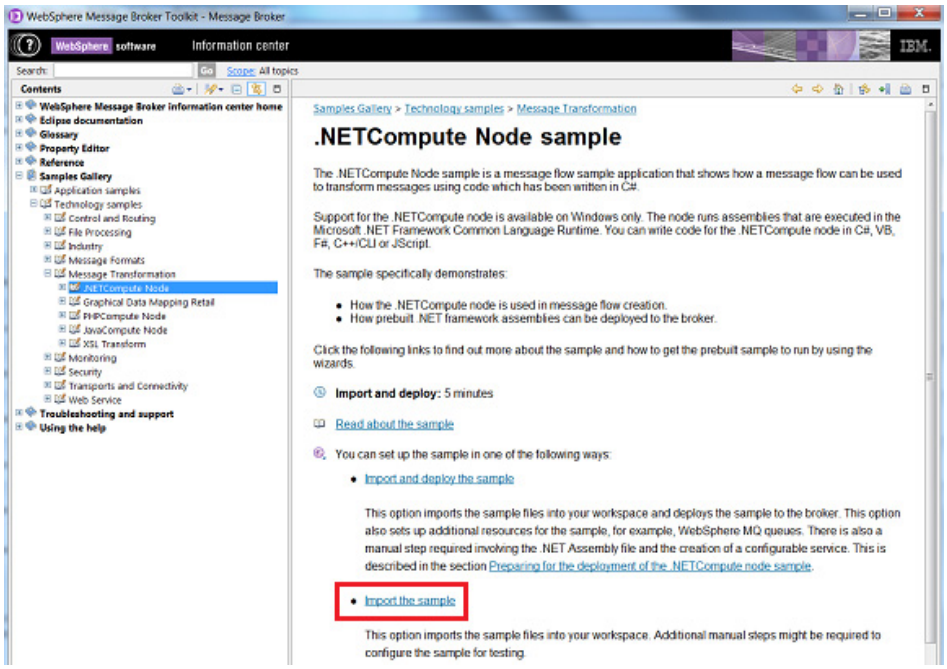
Importing the .NETCompute sample message flow

1. If the Message Broker Toolkit has not yet been started, select **Start => Programs => IBM WebSphere Message Broker Toolkit => IBM WebSphere Message Broker Toolkit 8.0 => WebSphere Message Broker Toolkit 8.0**. You will be asked for the location of a workspace -- use `C:\student\DOTNET\lab_sample\workspace`.
2. Navigate to the Samples Gallery of the WebSphere Message Broker Toolkit from the Help menu: Select **Help => Samples and Tutorials => WebSphere Message Broker Toolkit – Message Broker**. When the Samples and Tutorials page opens, scroll down and you should see a section named Message Transformation, highlighted with the red box below.

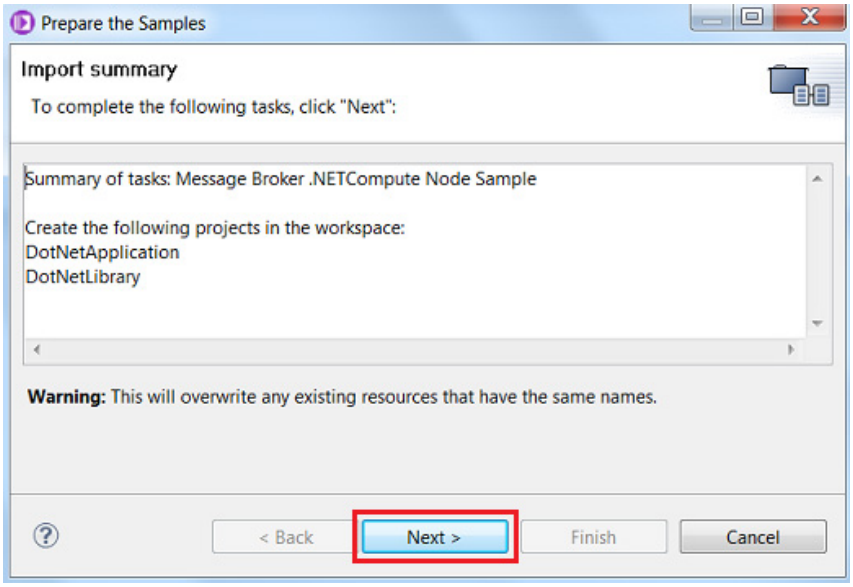
Expand the Message Transformation section and click on the **.NETCompute Node sample**:



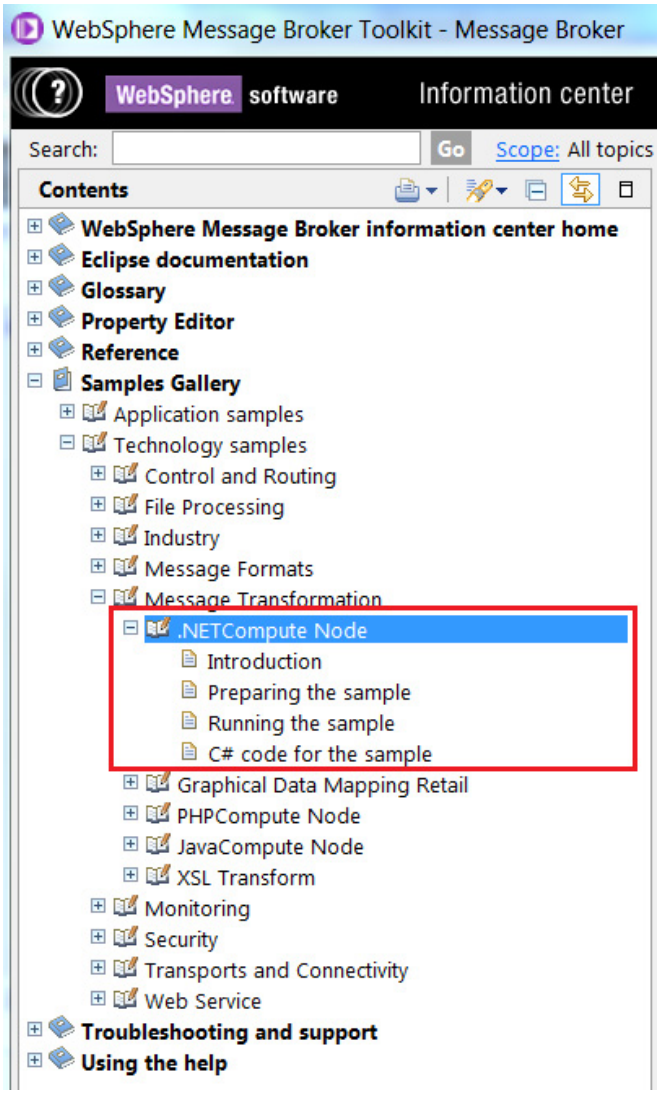
3. A window opens and displays the introduction to the .NETCompute node sample, as shown below. Read the page and click on **Import the sample** to import the sample files into your workspace in the Broker Toolkit:



4. The "Prepare the Samples" wizard will launch. Click **Next**, wait a few seconds for projects to be imported, click **Next** again, and the final page of the wizard should indicate that the import has been successful. Click **Finish** and you will be returned to the Samples window:



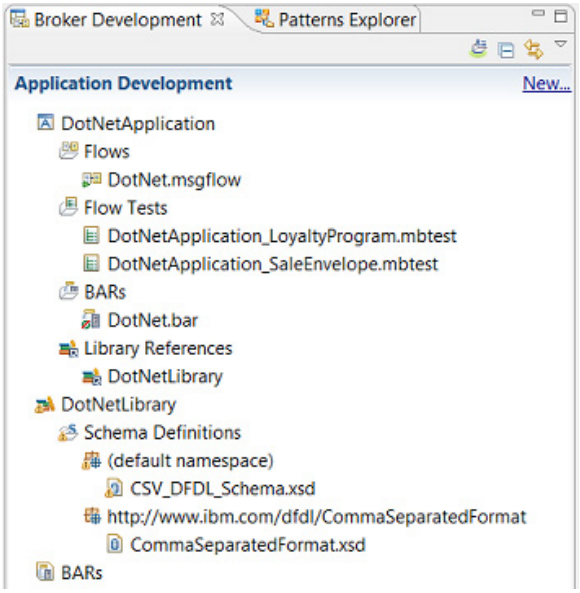
5. At this point, if you are experienced with Message Broker, you may want to expand the sample menus and explore the rest of the sample on your own. You will see topics named "Introduction," "Preparing the sample," "Running the sample," and "C# code for the sample" to guide you. Alternatively, the following sections will walk you through the C# development in more detail.



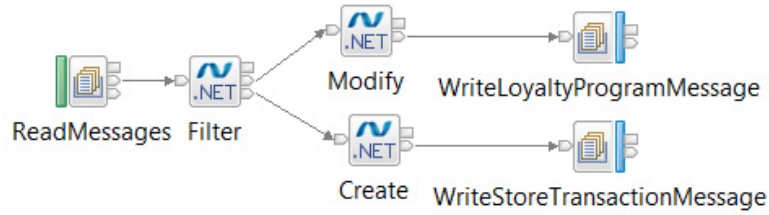
The next section of the tutorial explores the imported Message Broker artifacts.

Exploring the message flow

1. Minimize the window with the documentation for the sample, and return to the Message Broker Toolkit. You should see that the import has created some projects, as shown below. Expand the projects and you will see that they contain several prepared files:

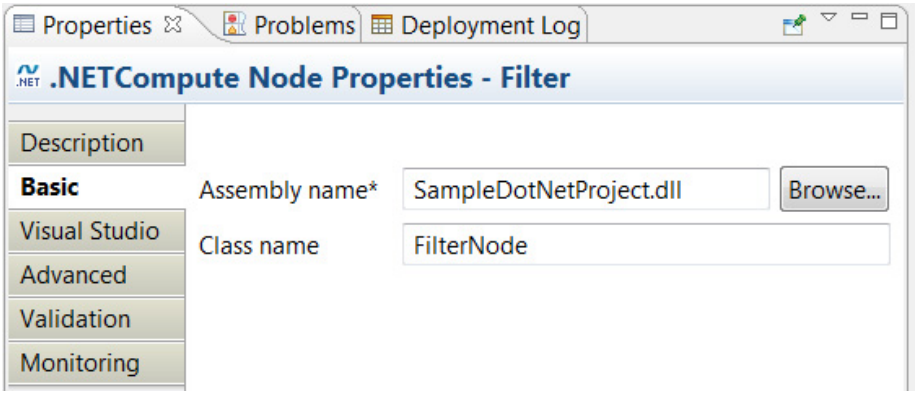


The application project named `DotNetApplication` contains a single message flow (DotNet.msgflow), two files with the extension `.mbtest`, which are used to send test messages through the message flow using the built-in Broker Test Client, and the Broker Archive (BAR) file `DotNet.bar`, which contains compiled copies of the resources that you will deploy to the runtime broker below. There is also a library project named `DotNetLibrary` that contains DFDL schema definitions for a comma separated style of output message. This library will be used for one of the output branches in the message flow. Next, examine the message flow `DotNet.msgflow`, which should already be open:



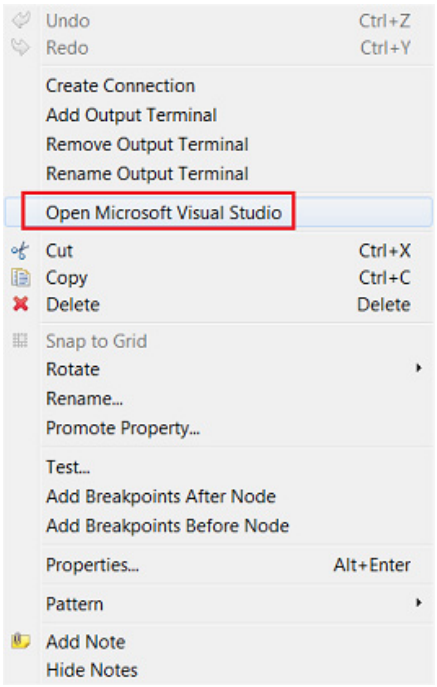
- The ReadMessages MQInput node takes messages from an input queue named DOTNET.IN.
- The Filter .NETCompute node routes each message down one of the two flow branches, depending on its format.
- The Modify .NETCompute node adds some XML elements to the message.
- The Create .NETCompute node transforms the input message into a new output message, which uses a comma separated format. Both branches of the message flow result in a message being written to the same output queue, named DOTNET.OUT.
- The WriteLoyaltyProgramMessage node has the output queue name of DOTNET.OUT hard-coded as its queue name property
- The WriteStoreTransactionMessage node has its output queue controlled dynamically using a DestinationList, which is set up by the preceding Create node. The sample chooses to set the DestinationList to point at the queue DOTNET.OUT as well.

2. Examine the properties of the Filter .NETCompute node:



When you create a .NETCompute node, you associate it with a .Net assembly file. By default, the sample flow has been configured with the name of an assembly file `SampleDotNetProject.dll`.

3. You will create a C# Project, add C# source code, and then build it in order to create this assembly file in subsequent steps, then return to the configuration of the message flow after you have created this transformation code in Microsoft Visual Studio. Right click on the **Filter node** and select **Open Microsoft Visual Studio**:

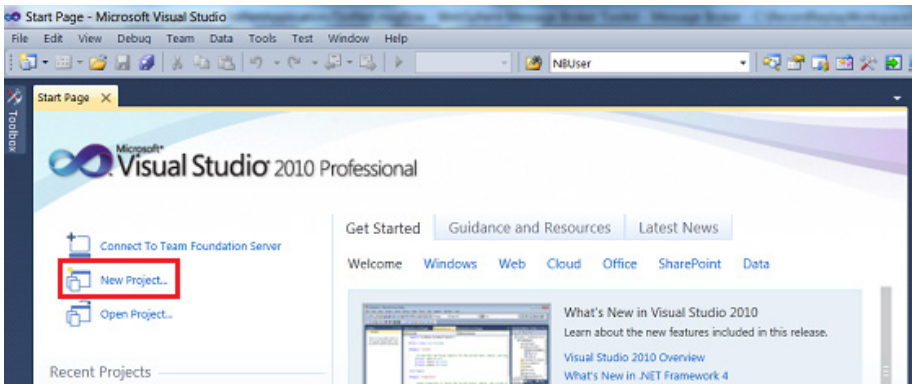


Once a .NETCompute node has been associated with a particular Microsoft Visual Studio solution using the Node property on the Visual Studio tab, it will launch with the solution files open. You are yet to create the .NET solution, so the Visual Studio opens with its splash page showing. The next section explains how to write the C# code.

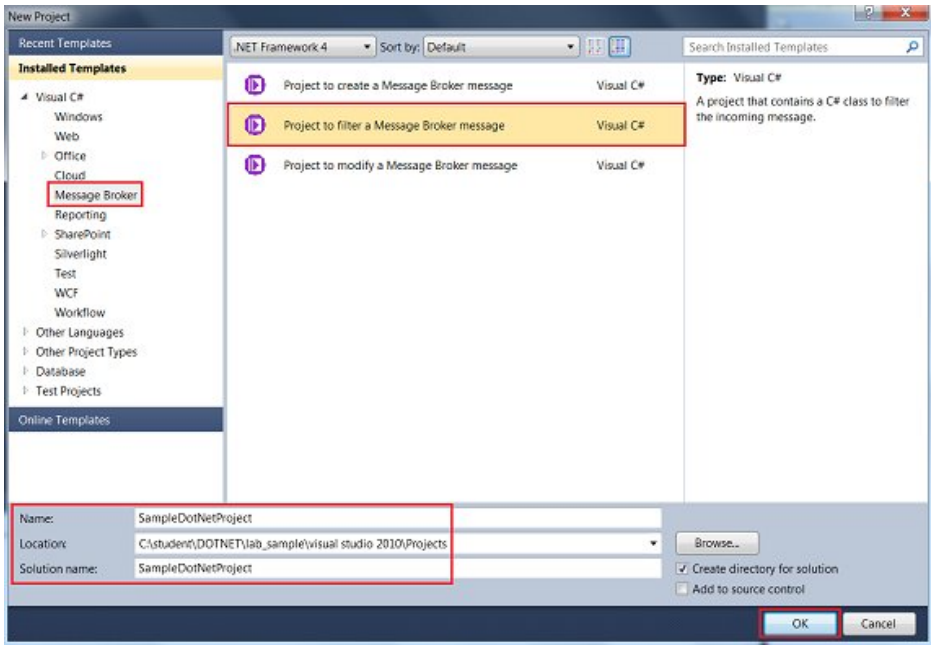
Creating the .NET solution in Microsoft Visual Studio

The following images were taken with Microsoft Visual Studio Professional Edition, but you can also use Microsoft Visual Studio Express Edition.

1. Once Microsoft Visual Studio has launched, you will see the Start Page below. Select **New Project**, as highlighted in the red box:

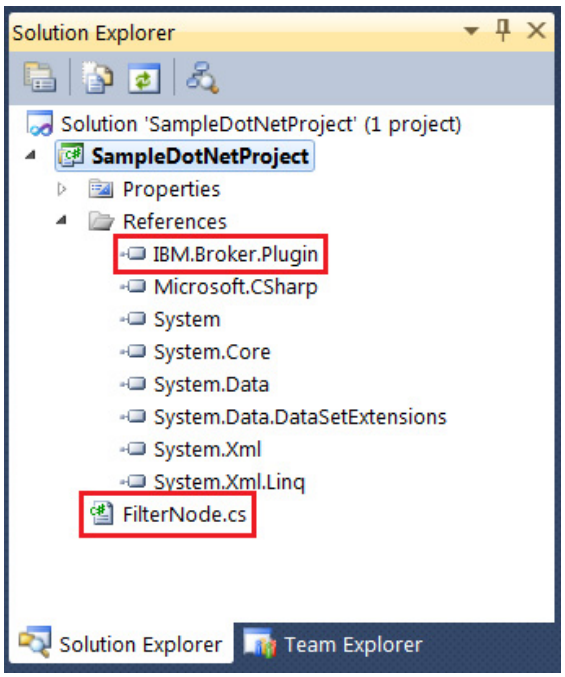


2. The New Project wizard offers you the three types of Project templates. Select the one named **Project to filter a Message Broker message**. Specify the properties at the bottom of the window as follows and Click **OK**:
   Name = `SampleDotNetProject`
   Location = `C:\student\DOTNET\lab_sample\visual studio 2010\Projects`
   Solution Name = `SampleDotNetProject`

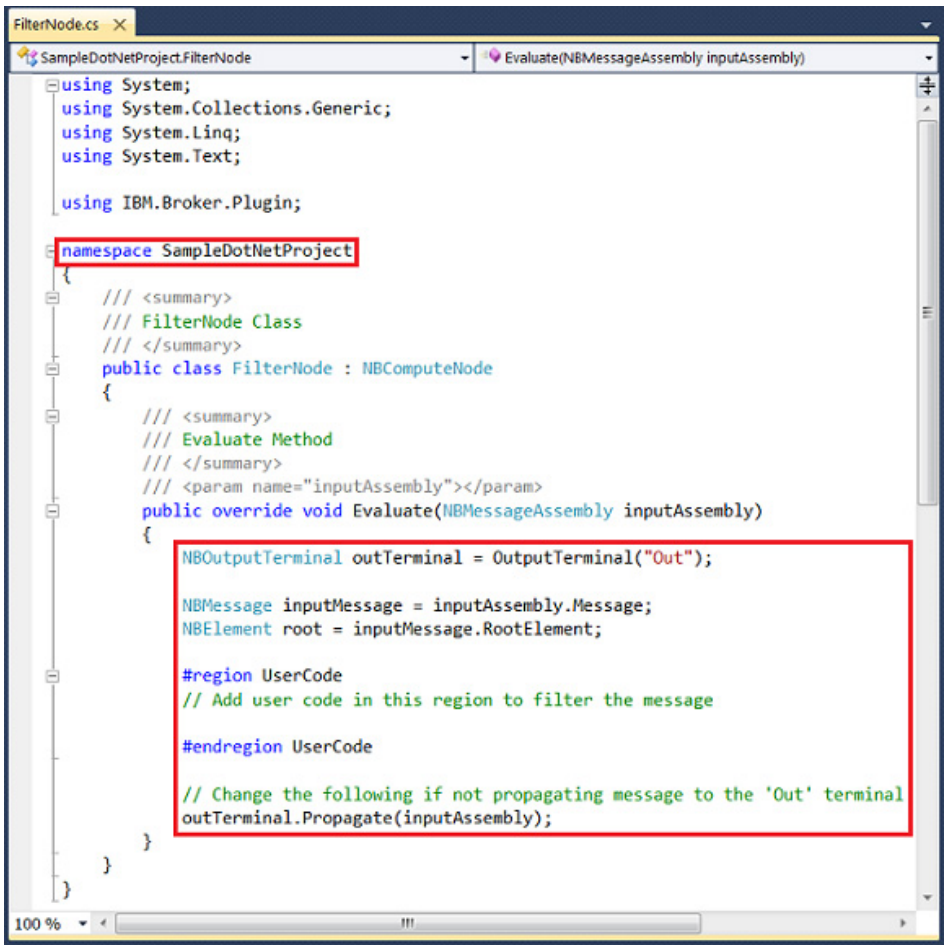


   If the WebSphere Message Broker Toolkit is installed after Microsoft Visual Studio, then the Message Broker project templates will be automatically installed ready for you to use. However, if the Broker Toolkit is installed first, then you will need to manually install the templates by executing the file `IBM.Broker.DotNet.vsix` and stepping through the wizard. If you have used the default installation location, you will find this file at `C:\Program Files (x86)\IBM\WMBT800\wmbt`.

3. Once the project is created, expand the **Solution Explorer view** at top right. You should see that the `FilterNode.cs` file has been created to contain the C# class, and that a reference to `IBM.Broker.Plugin` has been added. This assembly contains the API provided by Message Broker to transform messages in a .NETCompute node.



4. In the main window, `FilterNode.cs` has been created with a few lines of template code, which you will add to in the next step. The code is created within a namespace (highlighted below with a red box) that corresponds to the name of the project, SampleDotNetProject. The main entry point for .NET code executed in a .NETCompute is the Evaluate method, which contains a UserCode region where a flow developer typically adds their code. When the Propagate method is invoked, the message assembly leaves the .NETCompute node down the nominated terminal. When you edit this code in the next step, you will change the Evaluate method so that it creates an extra output terminal, and invokes the Propagate method against a chosen output terminal, depending on the content of the message data.



5. Replace the contents of the Evaluate method in `FilterNode.cs` (shown by the red box in the above image) with the sample code provided below. Remember to delete the existing call to the Propagate method from the template that was generated in the previous step. This call is the line of code `outTerminal.Propagate(inputAssembly);` Also, when you do the copy, be sure to include the definitions for the Alternate and Failure terminals, which appear in the first few lines of the code shown below:

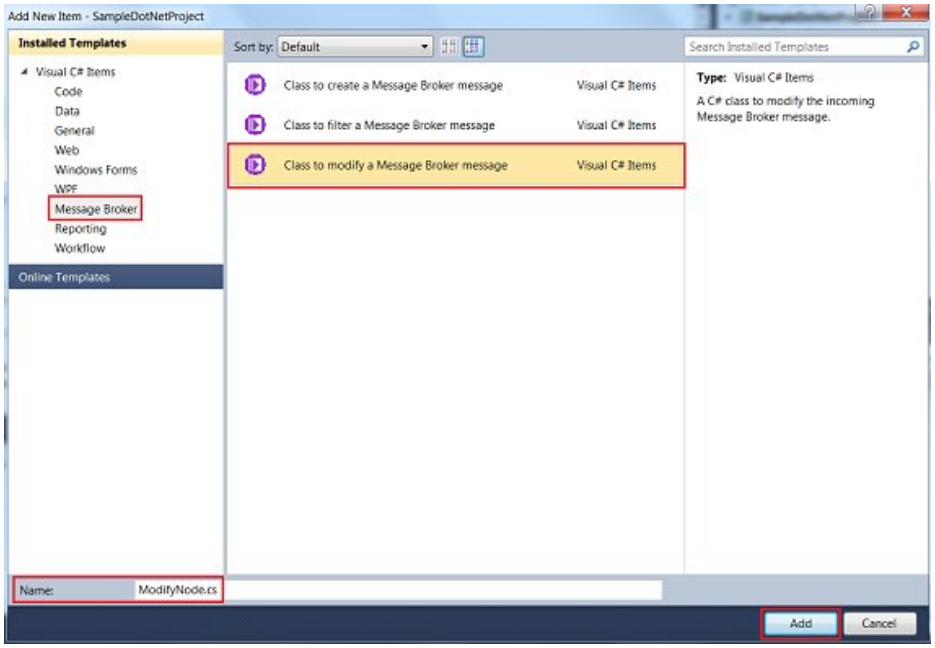
Listing 1. Sample code for Evaluate method of FilterNode.cs

```
NBOutputTerminal outTerminal = OutputTerminal("Out");
NBOutputTerminal altTerminal = OutputTerminal("Alternate");
NBOutputTerminal failureTerminal = OutputTerminal("Failure");

NBMessage inputMessage = inputAssembly.Message;
NBElement root = inputMessage.RootElement;

#region UserCode
     // Add user code in this region to filter the message
     // The following expression deliberately uses LastChild in
     // preference to FirstChild in case an XML Declaration is present!
     switch(root[NBParsers.XMLNSC.ParserName].LastChild.Name)
     {
          case "LoyaltyProgram":
               outTerminal.Propagate(inputAssembly);
               break;
          case "SaleEnvelope":
               altTerminal.Propagate(inputAssembly);
```

```
                break;
           default:
                failureTerminal.Propagate(inputAssembly);
                break;
      }
#endregion UserCode
```

Save the edited file `FilterNode.cs`" Press **Ctrl-S** or use the option from the **File** menu.

6. Add a new class to modify a Message Broker message from Solution Explorer: Right click the **Project** level of the hierarchy (SampleDotNetProject) and select **Add => Class**. In the resulting Add New Item dialog, select **Class to modify a Message Broker message**, and make sure that you specify the name `ModifyNode.cs` by default, the wizard suggests a name of `ModifyNode1.cs`). Click **Add**:



7. Edit the `ModifyNode.cs` file using the code below to populate the UserCode region of the template:

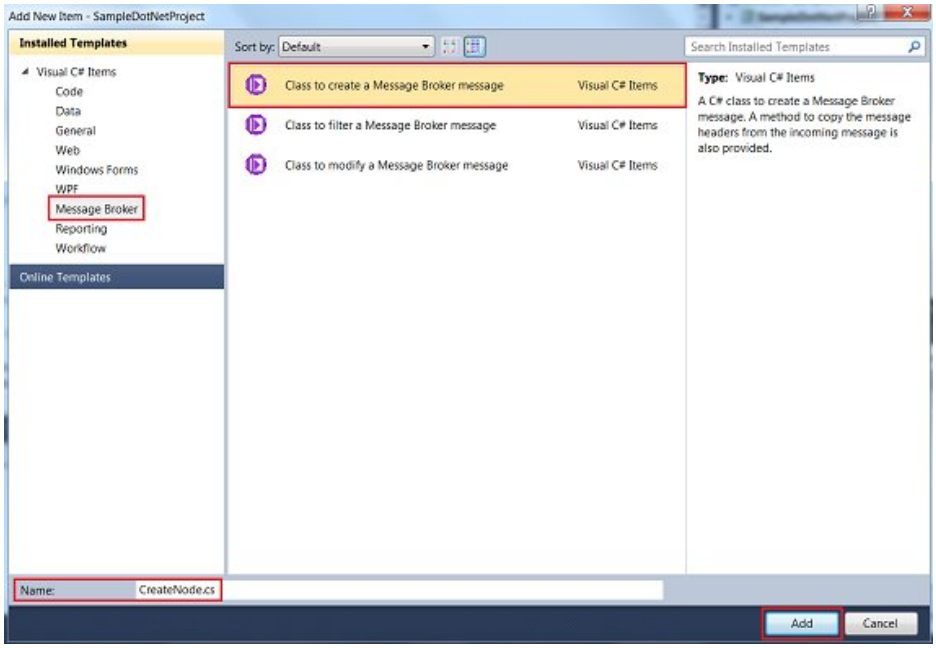
Listing 2. Sample code for UserCode region of ModifyNode.cs

```
#region UserCode
     NBElement xmlRoot = outputRoot[NBParsers.XMLNSC.ParserName];
     NBElement xmlDecl = xmlRoot[NBParsers.XMLNSC.XmlDeclaration, "XmlDeclaration"];
     if (xmlDecl == null)
     {
          // Create an XML Declaration if required
          NBParsers.XMLNSC.CreateXmlDeclaration(xmlRoot, "1.0", "UTF-8", "yes");
     }
     string notarget = "";
     string ns = "http://www.example.org/store";
     NBElement storeDetails = xmlRoot[notarget,"LoyaltyProgram"][ns, "StoreDetails"];
     string storeName = "";
     string storeStreet = "";
     string storeTown = "Happyville";
     switch ((string)storeDetails[ns, "StoreID"])
     {
          case "001":
               storeName = "Broker Brothers Central";
               storeStreet = "Exuberant Avenue";
```

```
                break;
           case "002":
                storeName = "Broker Brothers Mall";
                storeStreet = "Enthusiastic Crescent";
                break;
           case "003":
                storeName = "Broker Brothers District";
                storeStreet = "Peaceful Road";
                break;
      }
      storeDetails.CreateLastChild(ns, "StoreName", storeName);
      storeDetails.CreateLastChild(ns, "StoreStreet", storeStreet);
      storeDetails.CreateLastChild(ns, "StoreTown", storeTown);
#endregion UserCode
```

Save the edited file `ModifyNode.cs`" Press Ctrl-S or use the option from the **File** menu.

8. Add a new class to create a Message Broker message using the Solution Explorer: Right-click the **Project** level of the hierarchy (SampleDotNetProject) and select **Add => Class**. In the Add New Item dialog, select **Class to create a Message Broker message**, and make sure that you specify the name `CreateNode.cs` by default, the wizard suggests a name of `CreateNode1.cs`). Click **Add**:



9. Edit the `CreateNode.cs` file using the code below to populate the UserCode region of the template:

Listing 3. Sample code for UserCode region of CreateNode.cs

```
#region UserCode
     outputRoot["Properties"]["MessageSet"].SetValue("DotNetLibrary");
     outputRoot["Properties"]["MessageType"].SetValue("File");
     outputRoot.CreateLastChildUsingNewParser(NBParsers.DFDL.ParserName);
     NBElement File =
          outputRoot[NBParsers.DFDL.ParserName].CreateFirstChild(null, "File");
     NBElement inxmlRoot = inputRoot[NBParsers.XMLNSC.ParserName];
     IEnumerable<NBElement> invoices =
          inxmlRoot["SaleEnvelope"]["SaleList"].Children("Invoice");
     foreach (NBElement invoice in invoices)
     {
          TransformInvoice(File, invoice);
     }
     // Define Local Environment override to dynamically control the MQOutput node
     NBElement outLE = outAssembly.LocalEnvironment.RootElement;
     NBElement mqLE =
          outLE.CreateFirstChild(null, "Destination").CreateFirstChild(null, "MQ");
     mqLE = mqLE.CreateFirstChild(null, "DestinationData");
     mqLE.CreateFirstChild(null, "queueName", "DOTNET.OUT");
#endregion UserCode
```

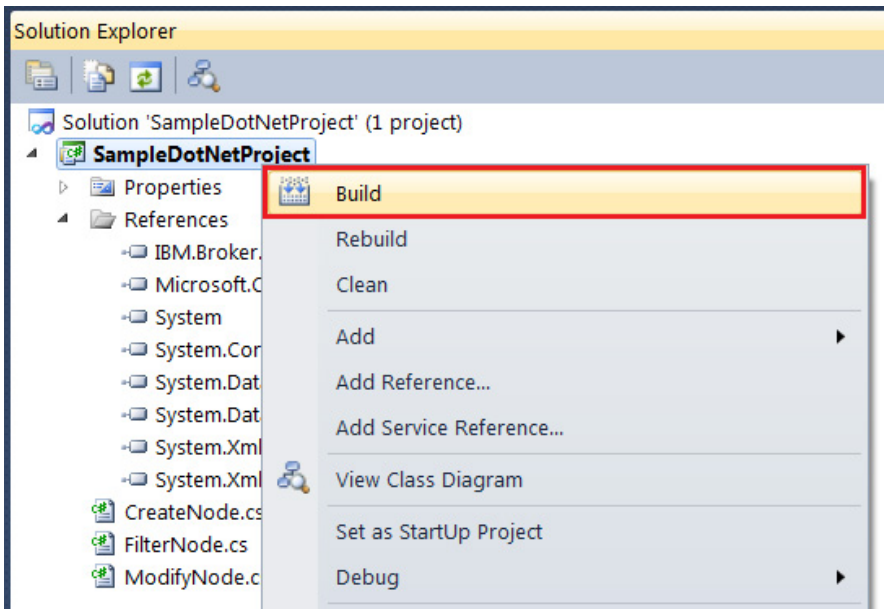
10. Add a new method named TransformInvoice to the `CreateNode.cs` file using the code provided below. Be sure to copy it to the correct position in the file. In the hierarchy of the file, this method should be a sibling of the CopyMessageHeaders method.

Listing 4. Sample code for the TransformInvoice method of CreateNode.cs

```
private static void TransformInvoice(NBElement outFileEl, NBElement inInvEl)
{
     // This method creates a structure based on
     // the Invoice Element in the input message
     IEnumerable<NBElement> items = inInvEl.Children("Item");
     foreach (NBElement item in items)
     {
          NBElement record = outFileEl.CreateLastChild(null, "Record");
          string notgt = "";
          record.CreateLastChild(notgt, "Code1", (string)item["Code", 0]);
          record.CreateLastChild(notgt, "Code2", (string)item["Code", 1]);
          record.CreateLastChild(notgt, "Code3", (string)item["Code", 2]);
          record.CreateLastChild(notgt, "Description", (string)item["Description"]);
          record.CreateLastChild(notgt, "Category", (string)item["Category"]);
          record.CreateLastChild(notgt, "Price", (decimal)item["Price"]);
          record.CreateLastChild(notgt, "Quantity", (Int32)item["Quantity"]);
     }
}
```

You can leave the CopyMessageHeaders method (which was provided as part of the template when you added `CreateNode.cs`) unchanged. Save the edited file `CreateNode.cs`: Press Ctrl-S or use the option from the **File** menu.

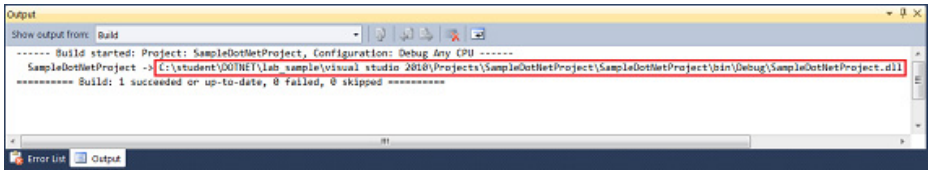
11. From the Solution Explorer, right-click on the Solution and select **Build** (or use the F6 shortcut):



How to open the Output window in Microsoft Visual Studio

If the Output window is not visible and you are using Microsoft Visual Studio Professional Edition, you can open it using **Debug => Windows => Output**. If the Output window is not visible and you are using Microsoft Visual Studio Express Edition, you can open it using **View => Output**.

12. The Output window shows you where the built assembly file has been saved on your file system. Depending on how you have Microsoft Visual Studio configured, the Build Output window may not be immediately visible, in which case you should follow the instructions in the sidebar to open the Output window.



If you have used the default naming suggested throughout the tutorial, then you should find that the assembly file has been saved at:

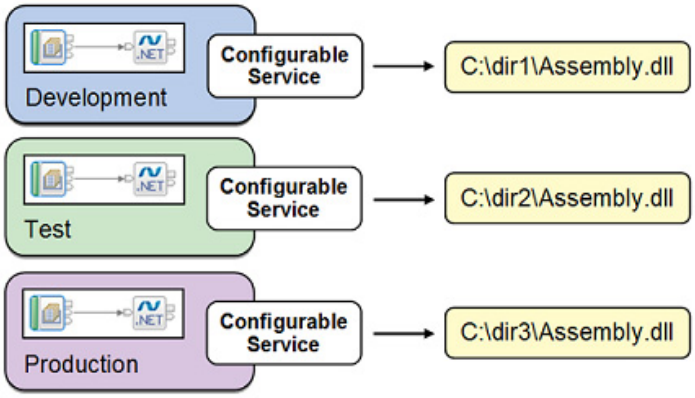
```
C:\student\DOTNET\lab_sample\visual studio 2010\Projects\SampleDotNetProject\
SampleDotNetProject\bin\Debug\SampleDotNetProject.dll
```

The next section of the tutorial shows you how to unite the message flow development and the C# assembly and deploy to Message Broker.

Deploying to Message Broker

Having built an assembly file from the C# code, it is possible to drag and drop the assembly file from a Windows Explorer window directly onto a .NETCompute node in a message flow in order to associate the node with the code. This technique results in a hard-coded absolute location for the assembly, and it is useful when developing, testing, and hot swapping the .NET code

that the Broker is executing. However, for production situations, a better approach is to define a Message Broker Configurable Service that specifies a .NETCompute node to locate the assembly file. This method is much more dynamic and better suited when moving a deployment between environments during development, test, and production. The diagram below shows that the same message flow can be used in multiple environments, with the configurable service in each environment defining the location of the assembly file, which may be at a different location on the file system for each environment.

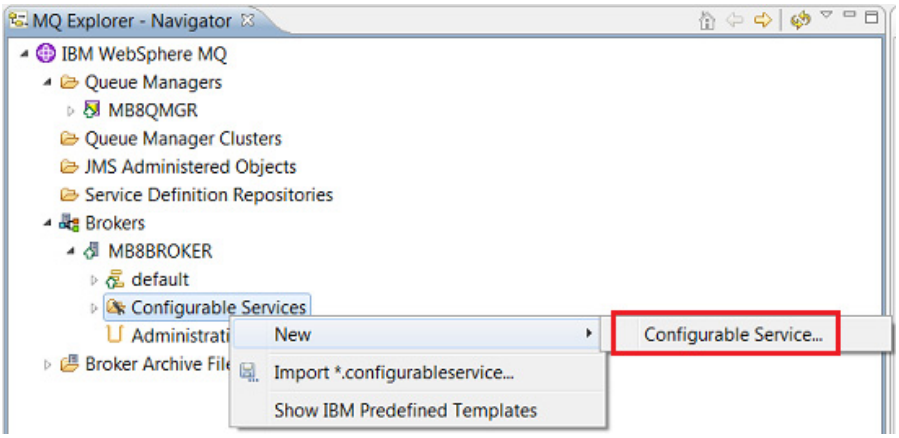


Use this approach and define a configurable service using the following steps:

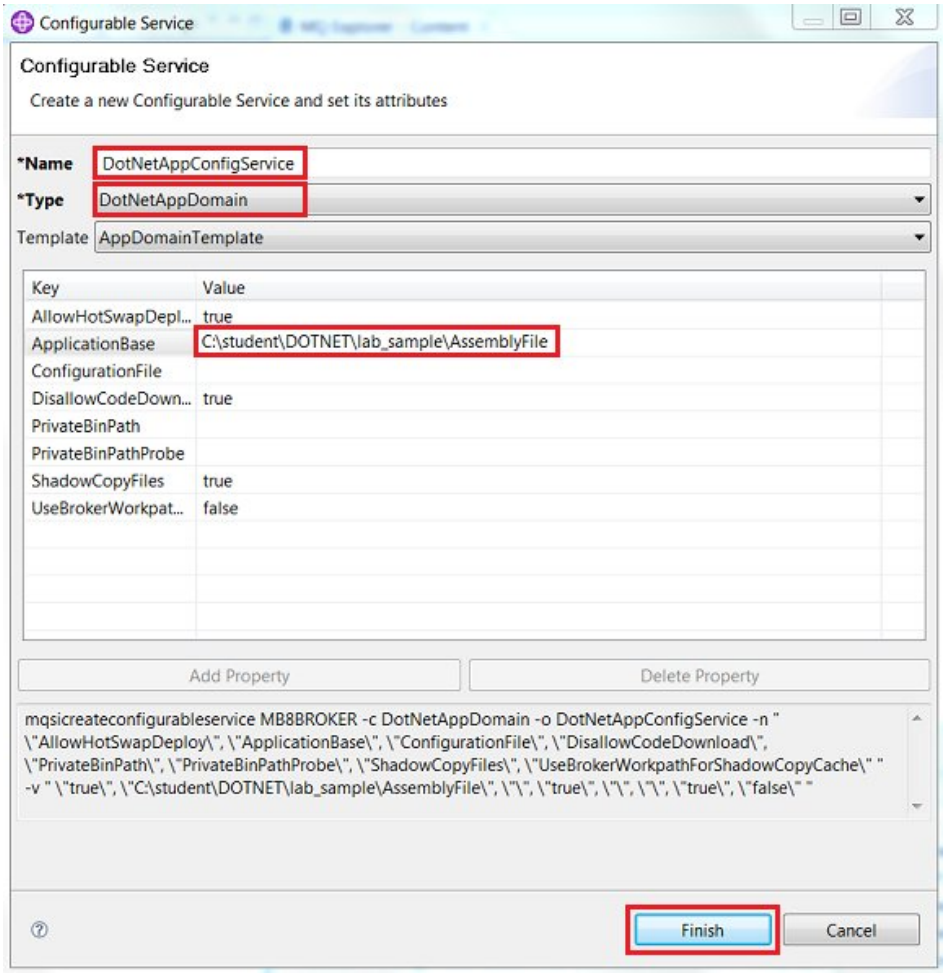
1. Using a Windows Explorer window, make a copy of the assembly and debug files, which are named `SampleDotNetProject.dll` and `SampleDotNetProject.pdb`:

```
Copy both files from the directory:
C:\student\DOTNET\lab_sample\visual studio 2010\Projects\
SampleDotNetProject\SampleDotNetProject\bin\Debug

To the directory:
C:\student\DOTNET\lab_sample\AssemblyFile
```

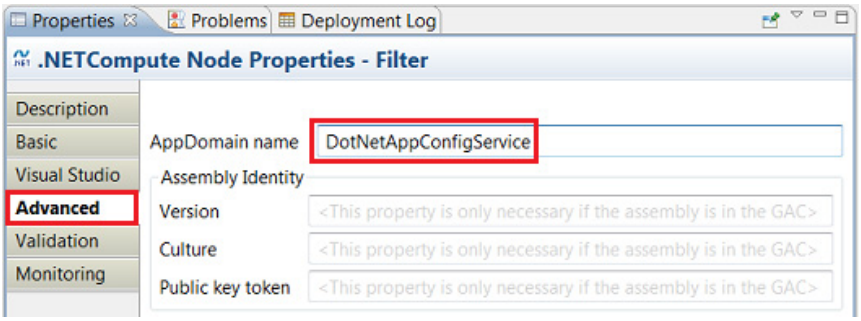
2. Define the required configurable service for the message flow using Message Broker Explorer, which is a graphical user interface for administering your brokers based on the Eclipse platform. Message Broker Explorer is an extension to WebSphere MQ Explorer. Open Message Broker Explorer: Select **Start => IBM WebSphere Message Broker 8.0.0.0 => Message Broker Explorer**. Right-click the **Configurable Services** folder for the runtime broker you are using and select **New => Configurable Service**.



3. Set the following parameters:
   Name: **DotNetAppConfigService**
   Type: **DotNetDomain**
   ApplicationBase: **C:\student\DOTNET\lab_sample\AssemblyFile**
   Click **Finish**.



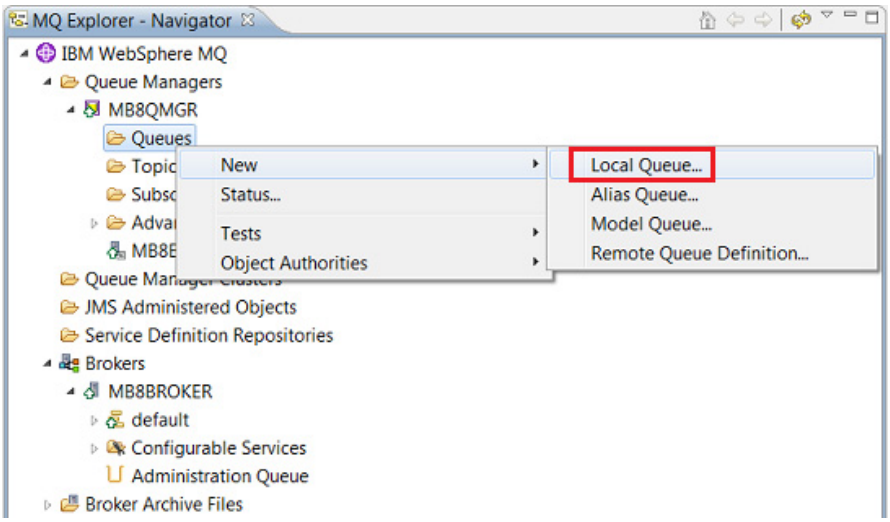
4. Return to the WebSphere Message Broker Toolkit and navigate to the properties of the .NETCompute node named Filter. Switch to the **Advanced** tab and set the AppDomain name to be `DotNetAppConfigService`:



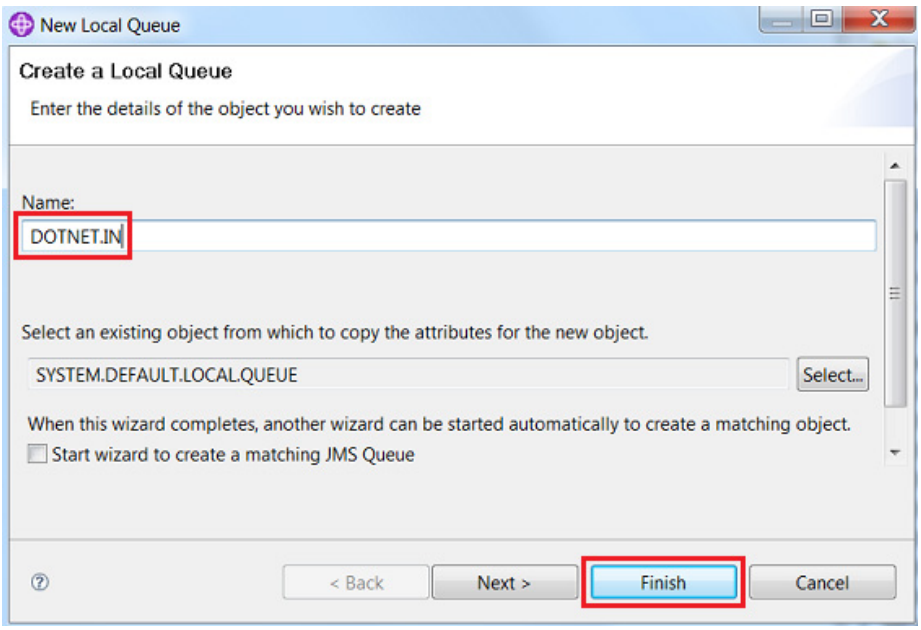
5. Repeat the last step for the other two .NETCompute nodes in the message flow, named Modify and Create. Each of these nodes uses a class defined inside the same assembly,

and each of the nodes will locate the assembly at runtime using the configurable service you defined above.

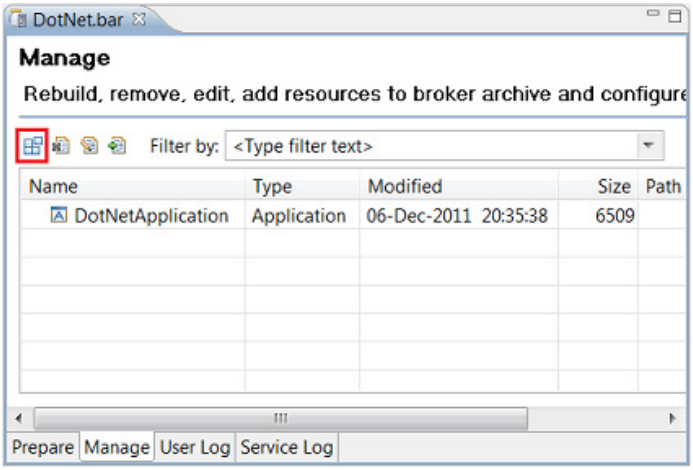
6. The rest of the tutorial assumes that you have created a queue manager named MB8QMGR and a runtime broker named MB8BROKER, which are known as the *Default Configuration*. For more information, see Creating the Default Configuration in the Message Broker information center.
7. The message flow requires the creation of two MQ queues, DOTNET.IN and DOTNET.OUT. Return to Message Broker Explorer and right-click the **Queues** folder underneath your Queue Manager (in the screenshot below the queue manager is named MB8QMGR) and choose **New => Local Queue**:



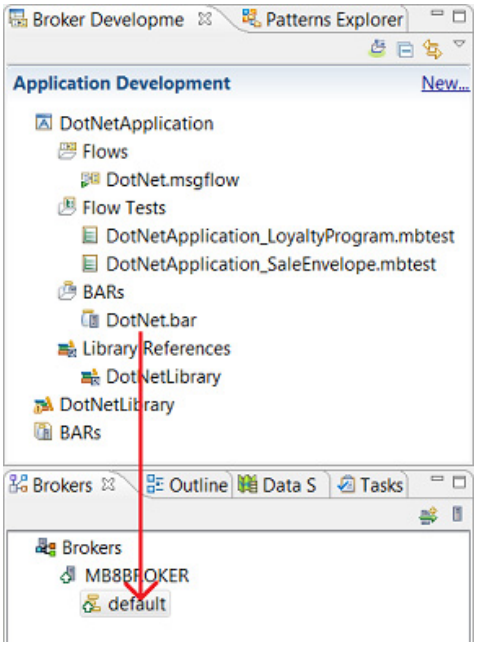
8. Specify the Name `DOTNET.IN` and click **Finish**:



9. Repeat the last two steps to create a queue named `DOTNET.OUT`.
10. Return to Message Broker Toolkit and open the BAR file named DotNet.bar, located inside DotNetApplication. Click **Rebuild and Save**:



11. Deploy the BAR file DotNet.bar by dragging and dropping it onto the execution group:



The next section of the tutorial shows you how to test the scenario.

Testing the scenario

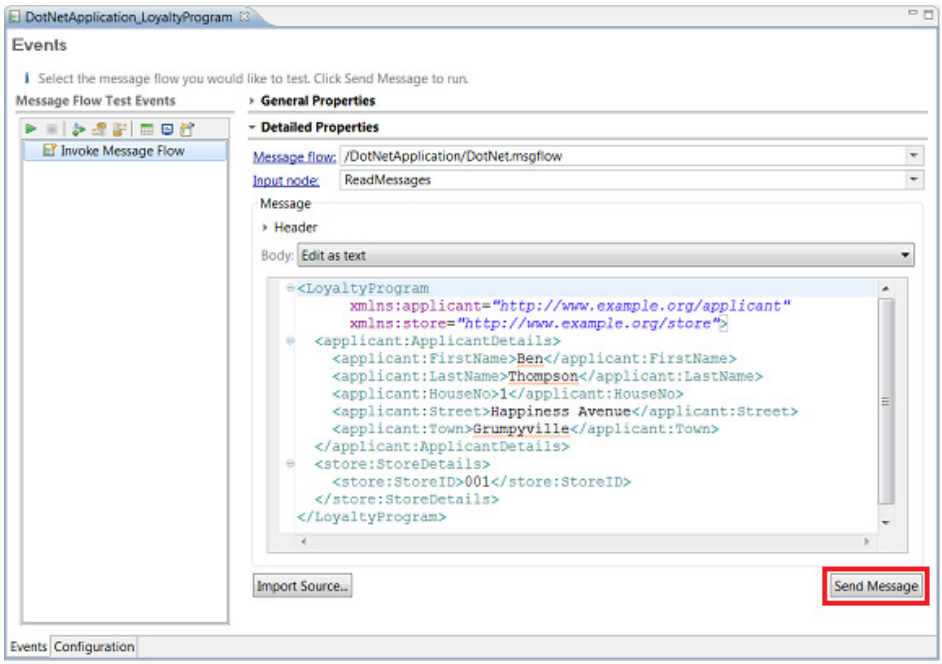
This final section of the tutorial shows you how to test the entire scenario directly from the Message Broker Toolkit using its built-in Test Client.

1. In order to test the top branch of the message flow, within the DotNetApplication, expand the **Flow Tests** folder and open the file named **DotNetApplication_LoyaltyProgram.mbtest**. The input data contained in the test file is shown below :

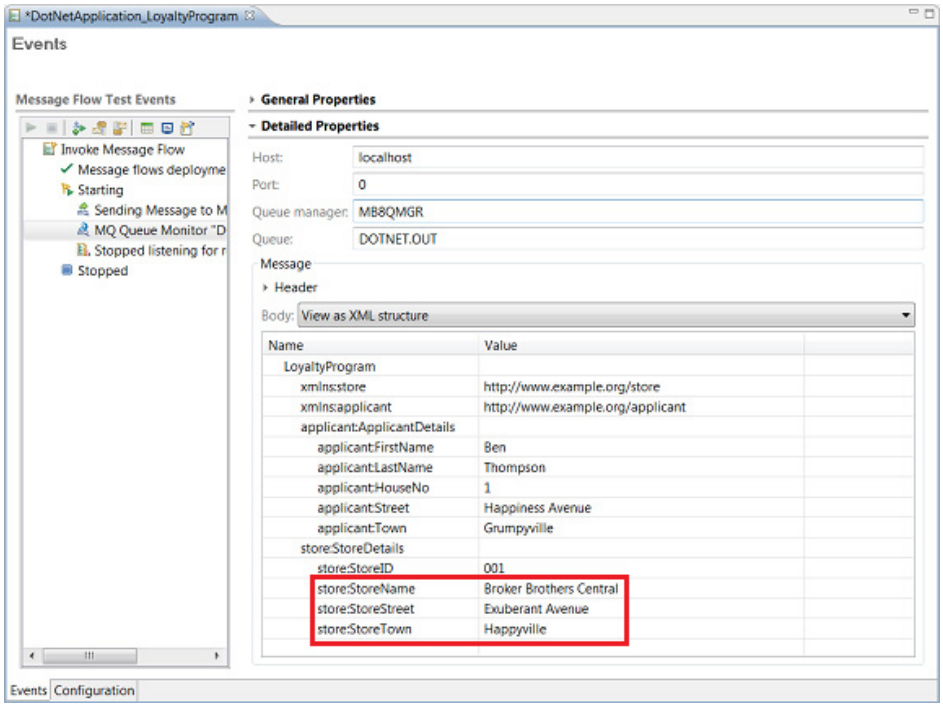
Listing 5. LoyaltyProgram message sample input data

```
<LoyaltyProgram
      xmlns:applicant="http://www.example.org/applicant"
      xmlns:store="http://www.example.org/store">
  <applicant:ApplicantDetails>
    <applicant:FirstName>Ben</applicant:FirstName>
    <applicant:LastName>Thompson</applicant:LastName>
    <applicant:HouseNo>1</applicant:HouseNo>
    <applicant:Street>Happiness Avenue</applicant:Street>
    <applicant:Town>Grumpyville</applicant:Town>
  </applicant:ApplicantDetails>
  <store:StoreDetails>
    <store:StoreID>001</store:StoreID>
  </store:StoreDetails>
</LoyaltyProgram>
```

Click the **Send Message** button in the bottom right corner:



2. Once the test has run, you should see that the StoreDetails section of the message has been enriched with a StoreName, StoreStreet, and StoreTown, as shown below in the red box:



Here is a listing of the output message:

Listing 6. LoyaltyProgram message sample output data

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<LoyaltyProgram
      xmlns:applicant="http://www.example.org/applicant"
      xmlns:store="http://www.example.org/store">
  <applicant:ApplicantDetails>
    <applicant:FirstName>Ben</applicant:FirstName>
    <applicant:LastName>Thompson</applicant:LastName>
    <applicant:HouseNo>1</applicant:HouseNo>
    <applicant:Street>Happiness Avenue</applicant:Street>
    <applicant:Town>Grumpyville</applicant:Town>
  </applicant:ApplicantDetails>
  <store:StoreDetails>
    <store:StoreID>001</store:StoreID>
    <store:StoreName>Broker Brothers Central</store:StoreName>
    <store:StoreStreet>Exuberant Avenue</store:StoreStreet>
    <store:StoreTown>Happyville</store:StoreTown>
  </store:StoreDetails>
</LoyaltyProgram>
```

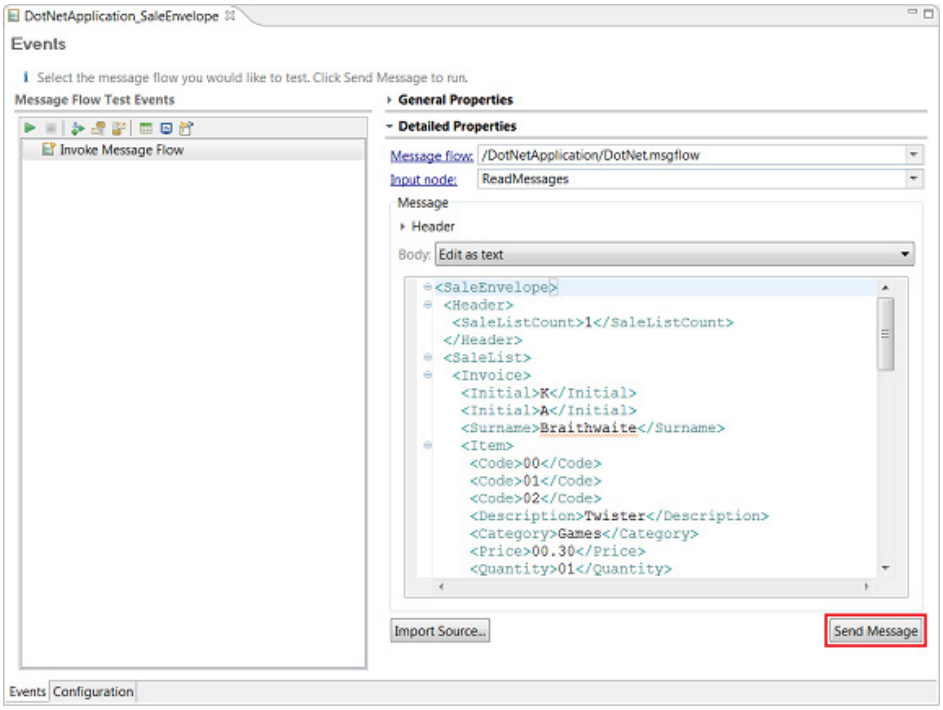
3. In order to test the bottom branch of the message flow, within the DotNetApplication, expand the **Flow Tests** folder and open the file **DotNetApplication_SaleEnvelope.mbtest**. The input data contained in the test file is shown below:

Listing 7. SaleEnvelope message sample input data

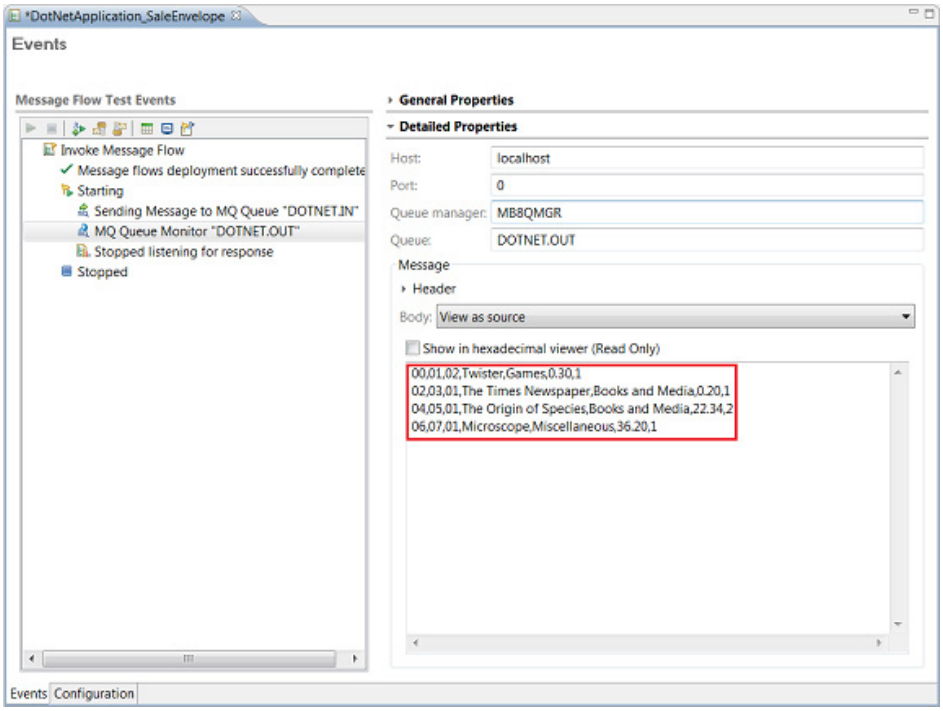
```
<SaleEnvelope>
 <Header>
  <SaleListCount>1</SaleListCount>
 </Header>
 <SaleList>
  <Invoice>
   <Initial>K</Initial>
   <Initial>A</Initial>
```

```
   <Surname>Braithwaite</Surname>
   <Item>
    <Code>00</Code>
    <Code>01</Code>
    <Code>02</Code>
    <Description>Twister</Description>
    <Category>Games</Category>
    <Price>00.30</Price>
    <Quantity>01</Quantity>
   </Item>
   <Item>
    <Code>02</Code>
    <Code>03</Code>
    <Code>01</Code>
    <Description>The Times Newspaper</Description>
    <Category>Books and Media</Category>
    <Price>00.20</Price>
    <Quantity>01</Quantity>
   </Item>
   <Balance>00.50</Balance>
   <Currency>Sterling</Currency>
  </Invoice>
  <Invoice>
   <Initial>T</Initial>
   <Initial>J</Initial>
   <Surname>Dunnwin</Surname>
   <Item>
    <Code>04</Code>
    <Code>05</Code>
    <Code>01</Code>
    <Description>The Origin of Species</Description>
    <Category>Books and Media</Category>
    <Price>22.34</Price>
    <Quantity>02</Quantity>
   </Item>
   <Item>
    <Code>06</Code>
    <Code>07</Code>
    <Code>01</Code>
    <Description>Microscope</Description>
    <Category>Miscellaneous</Category>
    <Price>36.20</Price>
    <Quantity>01</Quantity>
   </Item>
   <Balance>81.84</Balance>
   <Currency>Euros</Currency>
  </Invoice>
 </SaleList>
 <Trailer>
  <CompletionTime>12.00.00</CompletionTime>
 </Trailer>
</SaleEnvelope>
```

Click the **Send Message** button in the bottom right corner:



4. Once the test has run, you should see the output message displayed with a comma-separated format, as shown below in the red box:



Here is a listing of the output message:

Listing 8. SaleEnvelope message sample output data

```
00,01,02,Twister,Games,0.30,1
02,03,01,The Times Newspaper,Books and Media,0.20,1
04,05,01,The Origin of Species,Books and Media,22.34,2
06,07,01,Microscope,Miscellaneous,36.20,1
```

This is the end of the task steps for Part 1 of this tutorial series. You can continue on to the resource links and author information by clicking **Next** below, or you can go on to Part 2 of the tutorial series.