

Hands-On Lab

Part 1: Introduction to the AppFabric Service Bus

Lab version: 2.0.0

Last updated: 3/31/2011



Contents

[Overview 3](#_Toc289349796)

[Getting Started: Creating a Service Bus Namespace 6](#_Toc289349797)

[Task 1 – Creating your Service Bus Namespace 6](#_Toc289349798)

[Exercise 1: Getting a Basic Client and Service Working 9](#_Toc289349799)

[Task 1 – Creating the WCF Service Project and Service Contracts 10](#_Toc289349800)

[Task 2 – Hosting the WCF Service through the AppFabric Service Bus 20](#_Toc289349801)

[Task 3 – Configuring the WCF Service 26](#_Toc289349802)

[Task 4 – Creating the WCF Client Project 27](#_Toc289349803)

[Task 5 – Consuming the WCF Service Hosted through the App Fabric Service Bus 32](#_Toc289349804)

[Verification 36](#_Toc289349805)

[Exercise 2: Connecting a WCF Service in IIS 7.5 to the Service Bus 40](#_Toc289349806)

[Task 1 – Creating a Virtual Directory For The Application 41](#_Toc289349807)

[Task 2 – Configuring the WCF Service to be discoverable 43](#_Toc289349808)

[Task 3 – Enabling Windows Server App Fabric Auto-Start for the WCF Calculator Service 49](#_Toc289349809)

[Verification 57](#_Toc289349810)

[Exercise 3: Multicast Messaging through the Service Bus 61](#_Toc289349811)

[Task 1 – Defining the Service Contract Operations 62](#_Toc289349812)

[Task 2 – Subscribing The Worker Roles To Receive Messages 69](#_Toc289349813)

[Task 3 – Setting up the Management Console to Send Messages 80](#_Toc289349814)

[Verification 91](#_Toc289349815)

[Summary 94](#_Toc289349816)

Overview

* 1. A common problem when developing connected, peer-to-peer applications is to enable connections with applications through network devices like firewalls and **network address translators** (NATs).
  2. Network devices like these typically allow applications to initiate outbound network connections but prevent them from accepting inbound network connections.
  3. To establish a connection through these types of network devices, application developers often end up writing custom logic that deals with establishing connections across network topologies. Because there are many different types of network topologies, this logic often becomes very complex.
  4. The **AppFabric** **Service Bus** provides the communication infrastructure that protects developers from having to create the complex code necessary to achieve seamless connectivity. It allows you to expose a service to the Internet from behind your firewall or NAT. Several common application messaging patterns are supported allowing bi-directional, peer-to-peer communication, publish, subscribe and storing and forwarding of messages.

#### Facts

* + The Service Bus is built on **Windows Communication Foundation** (WCF) and uses standard Internet protocols.
  + The service is cloud-based. The technical aim of the Service Bus is to simplify connectivity between services and clients, even if those servers and clients exist behind different firewalls and NAT routers.
  + Building applications using the Service Bus does not require a significant footprint within an organization and leads to more rapid deployments while keeping the local infrastructure intact.
  + As a set of hosted integration technologies, the Service Bus reduces the cost involved in building, acquiring, and maintaining systems that might have to be built to do the same things.

# Objectives

* 1. In this hands-on lab, you will learn how to:
  + Expose and consume a service through theService Bus**.**
  + Implement the **Multicast** message pattern through the Service Bus**.**
  + Use X.509 certificates for message confidentiality.
  + Connect a WCF Service in IIS 7.5 to the Service Bus to enable service automatic activation.

# Prerequisites

* 1. You must have the following items to complete this lab:
  + [Microsoft Visual Studio 2010](http://www.microsoft.com/visualstudio/en-us/products/2010-editions)
  + [Microsoft.NET Framework 4](http://go.microsoft.com/fwlink/?LinkId=186916)
  + [Windows Azure platform AppFabric SDK V1.0](http://go.microsoft.com/fwlink/?LinkID=129448)
  + [Microsoft Internet Information Services 7.5](http://technet.microsoft.com/en-us/library/cc725762.aspx)
  + [IIS Manager for Remote Administration](http://www.iis.net/download/iismanager)
  + [Windows Server AppFabric](http://go.microsoft.com/fwlink/?LinkID=169435)
  + [Windows Azure Tools for Microsoft Visual Studio 1.4](http://go.microsoft.com/fwlink/?LinkID=128752)

# Setup

* 1. For convenience, much of the code used in this hands-on lab is available as Visual Studio code snippets. To check the prerequisites of the lab and install the code snippets:
  2. Open a Windows Explorer window and browse to the lab’s **Source\Setup** folder.
  3. Double-click the **Dependencies.dep** file in this folder to launch the Dependency Checker tool and install any missing prerequisites and the Visual Studio code snippets.
  4. If the User Account Control dialog is shown, confirm the action to proceed.
     1. **Note:** This process may require elevation. The *.dep* extension is associated with the Dependency Checker tool during its installation. For additional information about the setup procedure and how to install the Dependency Checker tool, refer to the **Setup.docx** document in the **Assets** folder of the training kit.

# Using the Code Snippets

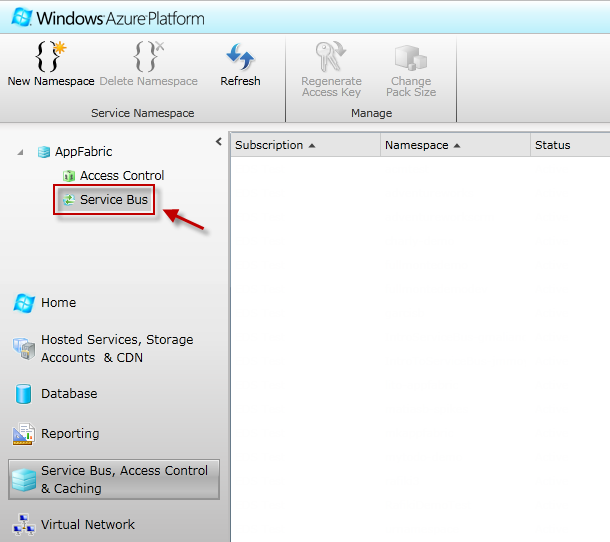
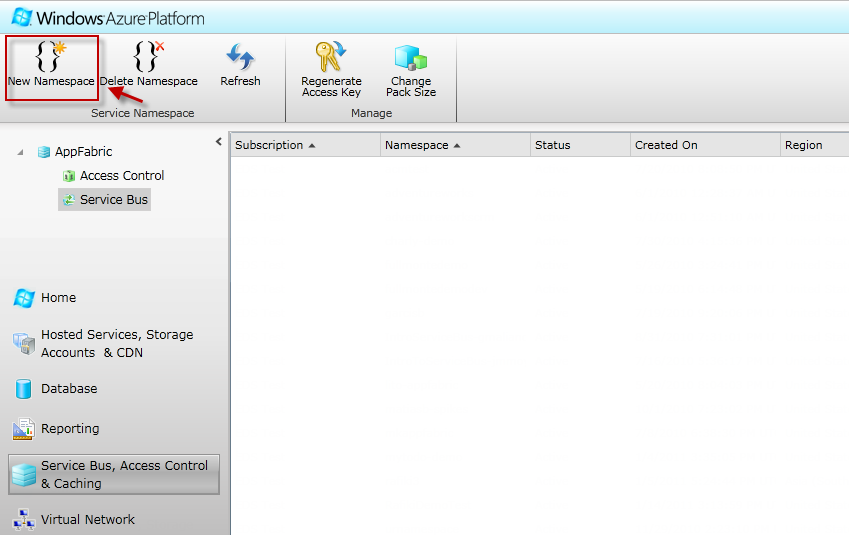
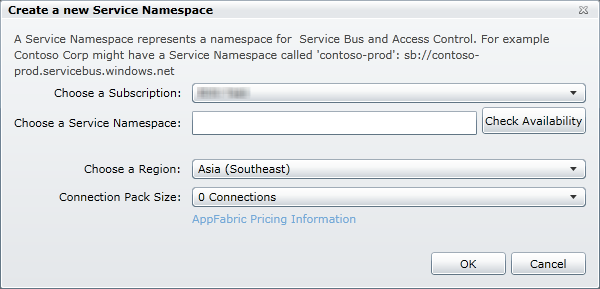
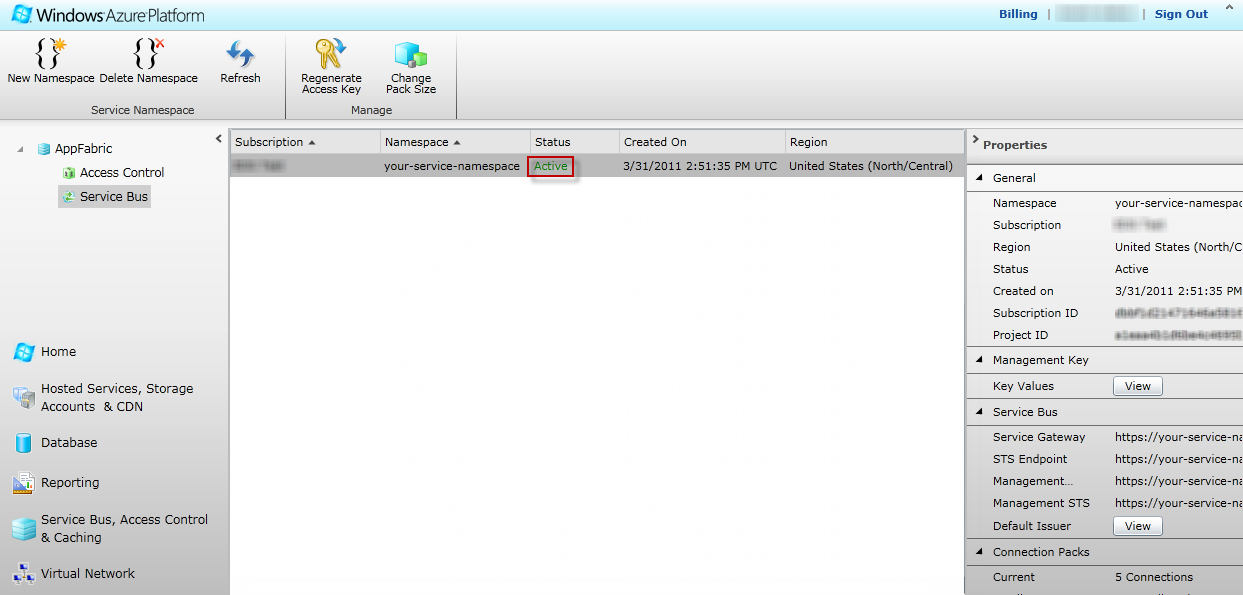
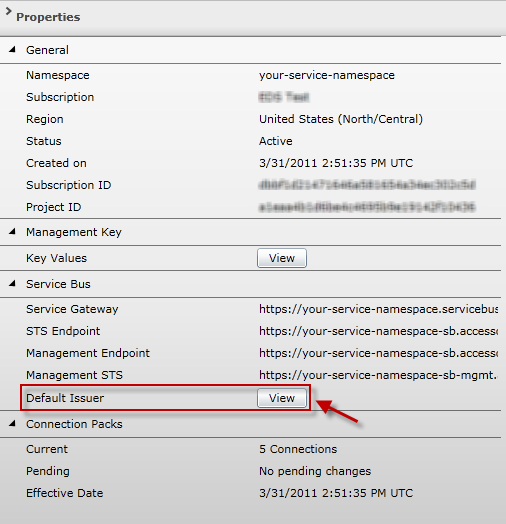
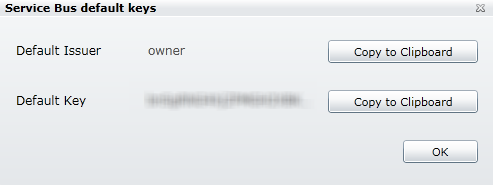
* 1. Throughout the lab document, you will be instructed to insert code blocks. For your convenience, most of that code is provided as Visual Studio Code Snippets, which you can use from within Visual Studio 2010 to avoid having to add it manually.
  2. If you are not familiar with the Visual Studio Code Snippets, and want to learn how to use them, you can refer to the **Setup.docx** document in the **Assets** folder of the training kit, which contains a section describing how to use them.

# Exercises

* 1. This hands-on Lab includes the following exercises:
  2. Getting a Basic Client and Service Working
  3. Connecting a WCF Service in IIS 7.5 to the Service Bus
  4. Multicast Messaging through theService Bus
  5. Estimated time to complete this lab: **60 minutes**.
  6. **Note:** When you first start Visual Studio, you must select one of the predefined settings collections. Every predefined collection is designed to match a particular development style and determines window layouts, editor behavior, IntelliSense code snippets, and dialog box options. The procedures in this lab describe the actions necessary to accomplish a given task in Visual Studio when using the **General Development Settings** collection. If you choose a different settings collection for your development environment, there may be differences in these procedures that you need to take into account.

Getting Started: Creating a Service Bus Namespace

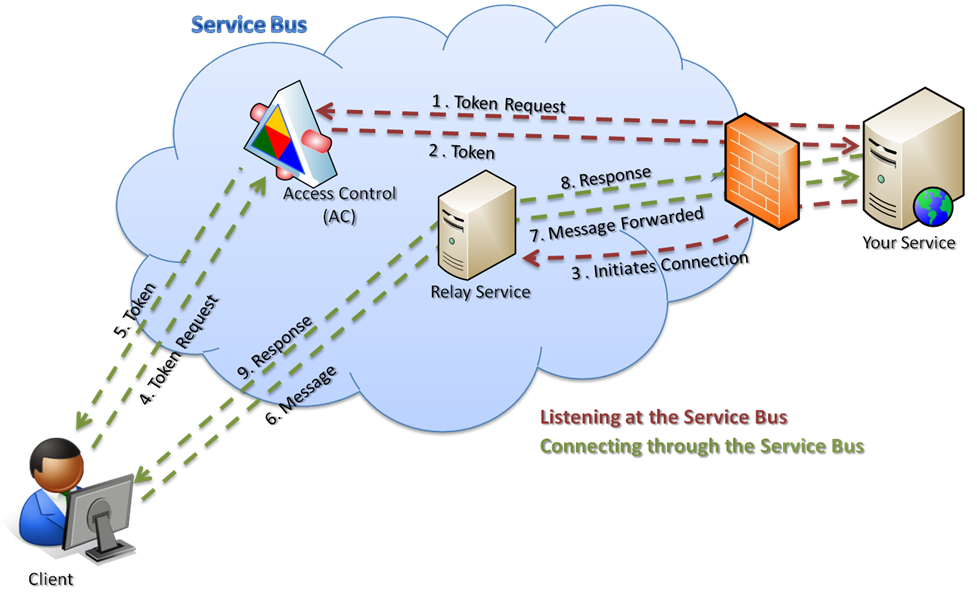
* 1. To follow this lab and complete all the exercises you first need to create an AppFabric Service Bus Namespace. Once you have created , it can be used for **all** of the AppFabric labs and for your own projects as well.

1. Task 1 – Creating your Service Bus Namespace
   1. In this task, you will create a new AppFabric Service Bus Namespace.
   2. Navigate to the [Windows Azure platform portal](https://windows.azure.com/). You will be prompted for your Windows Live ID credentials if you are not already signed in.
   3. Click **Service Bus, Access Control & Caching** link in the left pane, and then select “**Service Bus**” item under AppFabric element.
      1. 
      2. Figure
      3. Configuring AppFabric Service bus
   4. Add a Service Namespace. A service namespace provides an application boundary for each application exposed through the Service Bus and is used to construct Service Bus endpoints for the application. To add a service namespace, click **New Namespace** button on the upper ribbon bar.
      1. 
      2. Figure
      3. Creating a New Namespace
   5. Choose the Subscription, enter a namefor your **Service Namespace**, select a **Region** for your service to run in, **select a Connection Pack Size** and click the **OK** button. Make sure to validate the availability of the name first. Service names must be globally unique as they are hosted in the cloud and accessible by whomever you decide to grant access.
      1. 
      2. Figure
      3. Creating a new Service Namespace
      4. Please be patient while your service is activated. It can take a few minutes while it is provisioned.
      5. **Note:** You may have to refresh the browser to show the service is active.
   6. Once the namespace is active, click its name in the list of available namespaces to display the Service Namespace information page.
      1. 
      2. Figure 4
      3. Summary page listing available service namespaces
   7. In the **Properties** right pane, locate the **Service Bus** section and click the Default Issuer **View** button.
      1. 
      2. Figure
      3. Summary page listing available service namespaces
   8. Record the value shown for **Default Issuer** and **Default Key,** and click **OK**. You will need these values later on to authenticate using the Access Control.
      1. 
      2. Figure 6
      3. Service Bus default keys
   9. You have now created a new AppFabric namespace for this hands-on lab. To sign in at any time, simply navigate to the [Windows Azure platform AppFabric portal](http://go.microsoft.com/fwlink/?LinkID=129428), click **Sign In** and provide your Live ID credentials.

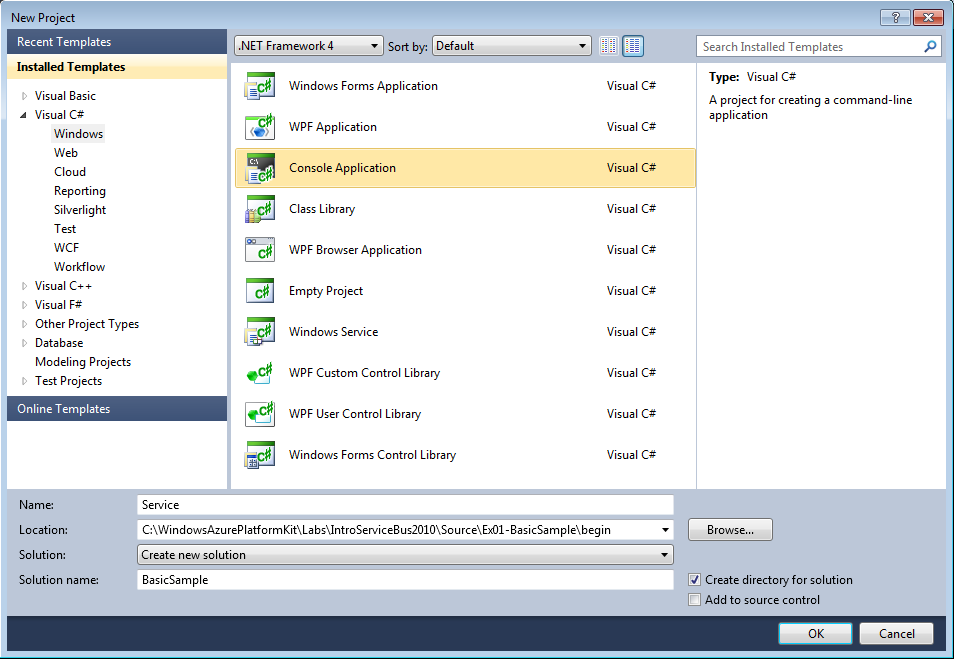
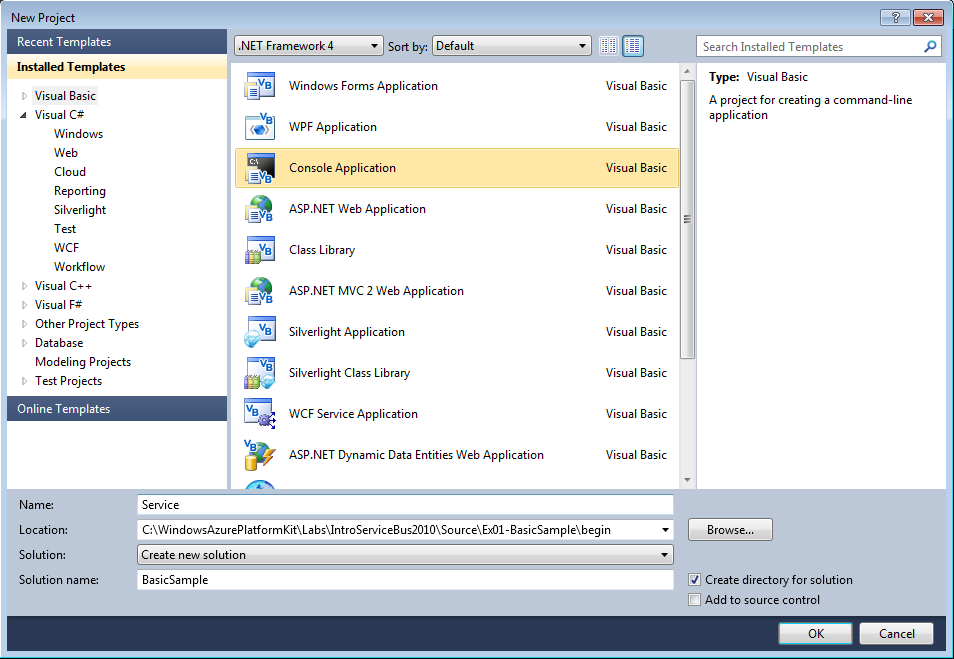
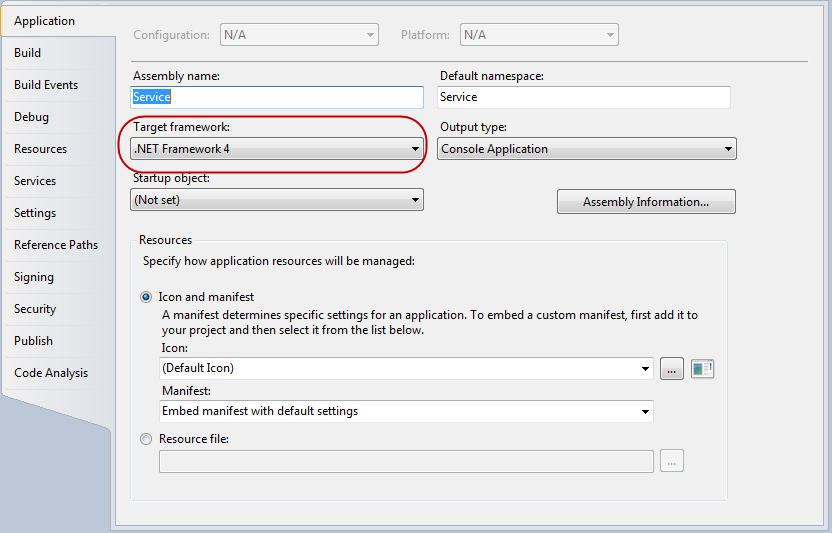
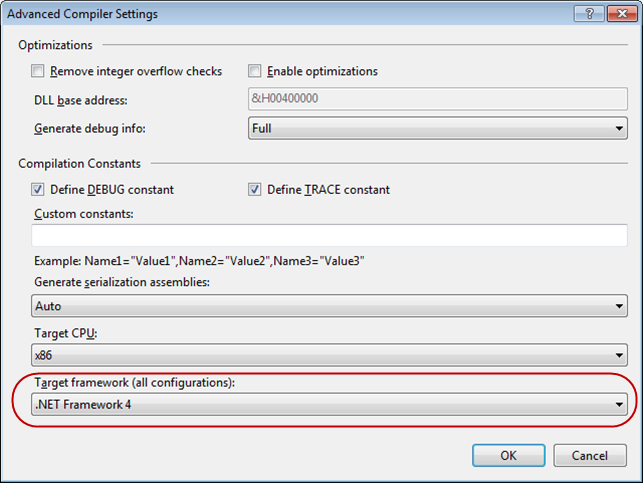
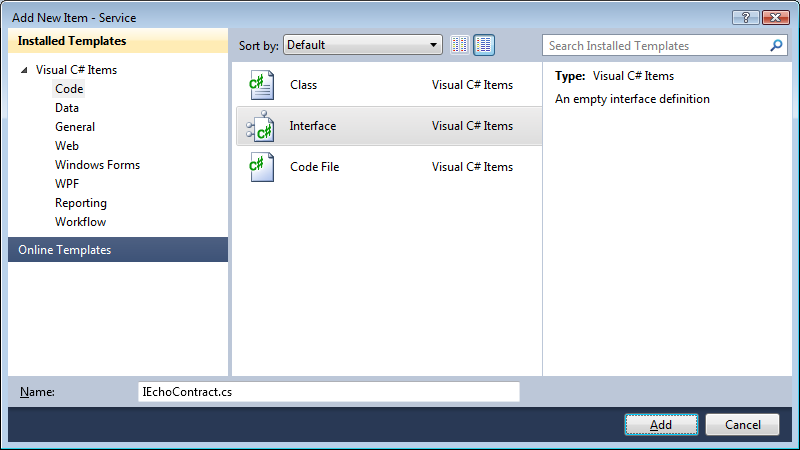
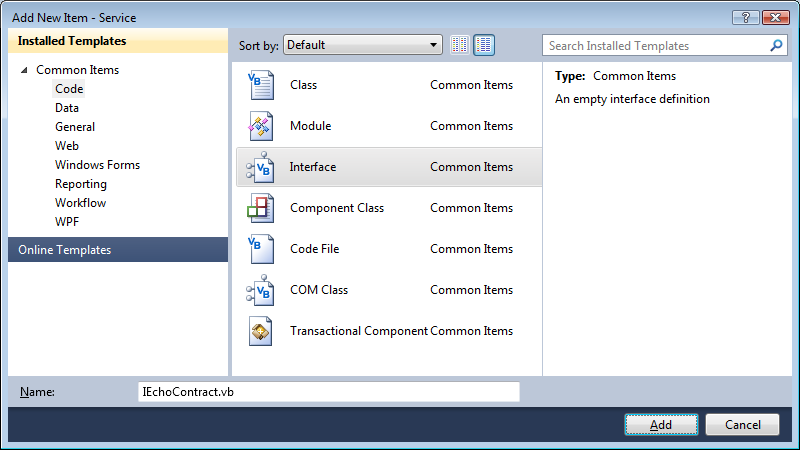
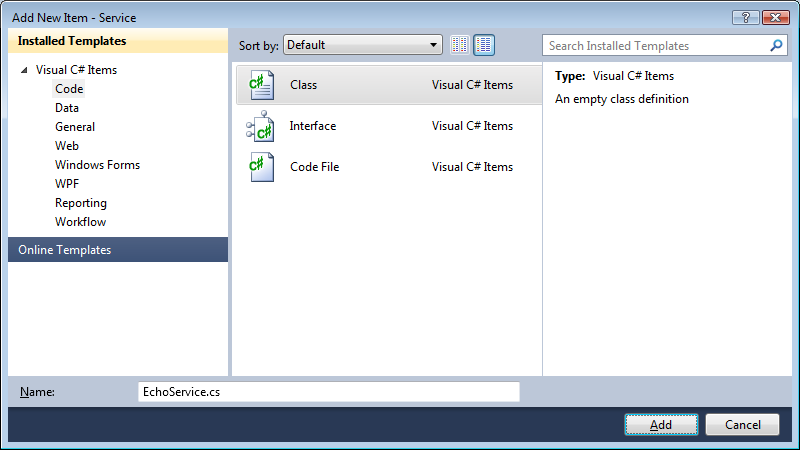
Exercise 1: Getting a Basic Client and Service Working

* 1. In this exercise, you learn how to expose and consume a simple WCF service through the Service Bus**.**
  2. TheService Bus simplifies many of the communication challenges by providing a universal naming convention for services along with a mechanism for direct/indirect communication between two nodes independent of network topology and configuration.

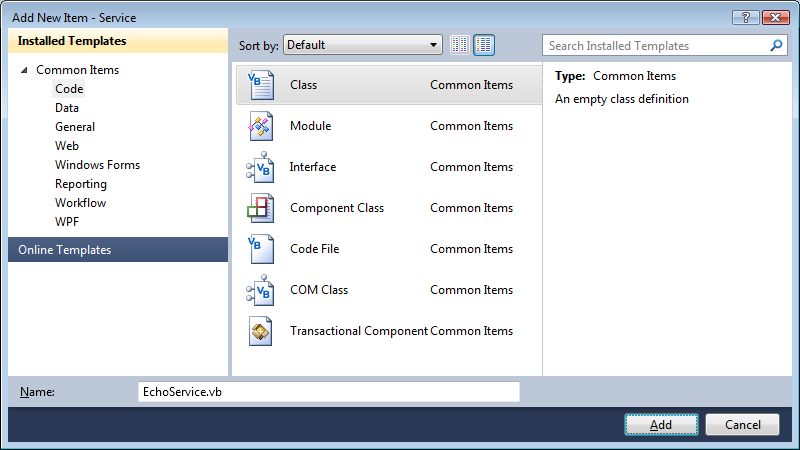
TheService Bus allows a Windows Communication Foundation-based (WCF) application to listen to a public network address, even if the application is located behind a NAT or network firewall. This functionality can be used to help applications communicate with each other, regardless of network specific structure. Through the use of theService Bus as an intermediary you can interconnect different applications without the need to write and maintain complex logic and code to traverse networks.

* 1. Another core part of the solution is the AppFabric Access Control, a publicly accessible identity and authentication management service. Access Control (AC) is a service designed to secure REST web services allowing them to integrate with both simple clients and enterprise identity technologies using a single code base. Access Control is also used to secure the Service Bus endpoints. To accomplish this task, AC receives an issuer name and an issuer secret (32-bit key). If they are correctly authenticated, it issues a security token (called Simple Web Token) that is then trusted by the service itself. All messages sent to and via the Service Bus must contain a valid security token.
  2. 
  3. Figure 7
  4. Service Bus message flow in relayed mode
  5. This exercise will expose several Service Bus features such as:
  + Access Control authentication methods
  + Communication between client and service across theService Bus
  1. **Note:** To verify that each step is performed correctly, it is recommended to build the solution at the end of each task.

Task 1 – Creating the WCF Service Project and Service Contracts

* 1. In this task, you create a WCF Service project that is then hosted and exposed through theService Bus.
  2. Open Microsoft Visual Studio 2010 from **Start | All Programs | Microsoft Visual Studio 2010 | Microsoft Visual Studio 2010**.
  3. In the **File** menu, point to **New**, and then click **Project**.
  4. In the **New Project** dialog, expand the language of your preference (Visual C# or Visual Basic) in the installed templates list and select **Windows**. Make sure that **.NET Framework 4** is selected as the target framework and choose the **Console Application** template. Set the project name to **Service**, the solution name to **BasicSample** and the location to **Ex01-BasicSample\begin** in the **Source** folder of this lab. Click **OK** to create the application project.
     1. 
     2. Figure 8
     3. Creating a new project (C#)
     4. 
     5. Figure 9
     6. Creating a new project (Visual Basic)
  5. Change the **Target Framework** to **.NET Framework 4** to allow the Service Bus interaction. To do this, in **Solution Explorer**, right-click the **Service** project and select **Properties.** 
     1. For **Visual C#** projects: In the **Properties** window, select the **Application** tab and choose .NET **Framework 4** in the **Target Framework** option. If prompted to confirm the target framework change, click **Yes** to proceed.
     2. For **Visual Basic** projects: In the **Properties** window, select the **Compile** tab, click **Advanced Compile Options**,choose **.NET** **Framework 4** in the **Target Framework** option, and click OK. If prompted to confirm the target framework change, click **Yes** to proceed.
     3. 
     4. Figure 10
     5. Choosing the target framework for C# projects
     6. 
     7. Figure 11
     8. Choosing the target framework for Visual Basic projects
  6. Add a reference to the **System.ServiceModel.dll** assembly. To do this, in **Solution Explorer**, right-click the **Service** project and select **Add Reference**. In the **Add Reference** dialog box, select the **.NET** tab and choose the **System**.**ServiceModel** assembly. Click **OK** to add the reference.
     1. **Note:** TheService Bus is built on Windows Communication Foundation; **System.ServiceModel.dll** is one of WCF's core assemblies.
  7. Now, add a reference to the Service Buscore assembly. Again, right-click the **Service** project in **Solution Explorer**, select **Add Reference** to open the **Add Reference** dialog, select the **.NET** tab, choose **Microsoft.ServiceBus.dll** for .NET Framework 4.0 and click **OK**.
     1. **Note:** If you cannot find the **Microsoft.ServiceBus** assembly in the **.NET** tab, use the **Browse** tab to locate this assembly inside the **%ProgramFiles%\Windows Azure AppFabric SDK\V1.0\Assemblies\NET4.0** folder.
  8. Create the WCF service contract. In **Solution Explorer**, right-click the **Service** project**,** point to **Add** and select **New** **Item**. In the **Add New Item** dialog, choose the **Code** category and select the **Interface** template. Change the **Name** to **IEchoContract.cs** (for Visual C# projects) or **IEchoContract.vb** (for Visual Basic projects) and click **Add** to create the interface.
     1. 
     2. Figure 12
     3. Creating the IEchoContract interface (C#)
     4. 
     5. Figure 13
     6. Creating the IEchoContract interface (Visual Basic)
  9. At the top of the **IEchoContract.cs** file (for Visual C# projects) or the **IEchoContract.vb** file (for Visual Basic projects), add the following namespace directive.
     1. C#
     2. **using System.ServiceModel;**
     3. Visual Basic
     4. **Imports System.ServiceModel**
  10. Make the interface **public** and apply the **ServiceContract** attribute to the **IEchoContract** interface to indicate that the interface defines a service contract in the WCF application. To do this, replace the current interface definition with the following code.
      1. **Note:** In Visual Basic, the template for a new interface already declares the interface as Public.
      2. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Echo Contract - CS*)
      3. C#
      4. **[ServiceContract(Name = "EchoContract", Namespace = "http://samples.microsoft.com/ServiceModel/Relay/")]**
      5. **public interface IEchoContract**
      6. **{**
      7. **}**
      8. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Echo Contract - VB*)
      9. Visual Basic
      10. **<ServiceContract(Name:="EchoContract", Namespace:="http://samples.microsoft.com/ServiceModel/Relay/")> \_**
      11. **Public Interface IEchoContract**
      12. **End Interface**
  11. Declare the **Echo** operation contract to indicate that the method defines an operation that is part of a service contract. To do this, paste the following code inside the interface definition.
      1. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Echo Operation Contract - CS*)
      2. C#
      3. public interface IEchoContract
      4. {
      5. **[OperationContract]**
      6. **string Echo(string text);**
      7. }
      8. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Echo Operation Contract - VB*)
      9. Visual Basic
      10. Public Interface IEchoContract
      11. **<OperationContract()> \_**
      12. **Function Echo(ByVal text As String) As String**
      13. End Interface
  12. Create the **EchoService** class to implement the **IEchoContract** interface. To do this, in **Solution Explorer**, right-click the **Service** project**,** point to **Add** and select **New** **Item**. In the **Add** **New** **Item** dialog box, choose the **Class** template. Change the **Name** to **EchoService.cs** (for Visual C# projects) or **EchoService.vb** (for Visual Basic projects) and then click **Add**.
      1. 
      2. Figure 14

Creating the EchoService class (C#)

* + 1. 
    2. Figure 15
    3. Creating the EchoService class (Visual Basic)
  1. At the top of the **EchoService** class, add the following namespace directive.
     1. C#
     2. **using System.ServiceModel;**
     3. Visual Basic
     4. **Imports System.ServiceModel**
  2. Add an inheritance declaration to make the **EchoService** class implement the **IEchoContract** interface and add the **public** access modifier.
     1. **Note:** In Visual Basic, the template for a new class already declares the class as Public.
     2. C#
     3. **public** class EchoService : **IEchoContract**
     4. **{**
     5. **}**
     6. Visual Basic
     7. Public Class EchoService
     8. **Implements IEchoContract**
     9. End Class
  3. Add a **ServiceBehavior** attribute to the **EchoService** class to indicate that the class implements a WCF service. To do this, paste the following code above the class definition.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Echo Service Behavior - CS*)
     2. C#
     3. **[ServiceBehavior(Name = "EchoService", Namespace = "http://samples.microsoft.com/ServiceModel/Relay/")]**
     4. public class EchoService : IEchoContract
     5. {
     6. }
     7. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Echo Service Behavior - VB*)
     8. Visual Basic
     9. **<ServiceBehavior(Name:="EchoService", Namespace:="http://samples.microsoft.com/ServiceModel/Relay/")> \_**
     10. Public Class EchoService
     11. Implements IEchoContract
     12. End Class
  4. Implement the **Echo** method of the **IEchoContract** interface. To do this, paste the following code inside the class definition.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Echo Method - CS*)
     2. C#
     3. public class EchoService : IEchoContract
     4. {
     5. **public string Echo(string text)**
     6. **{**
     7. **Console.WriteLine("Echoing: {0}", text);**
     8. **return text;**
     9. **}**
     10. }
     11. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Echo Method - VB*)
     12. Visual Basic
     13. Public Class EchoService
     14. Implements IEchoContract
     15. **Public Function Echo(ByVal text As String) As String Implements IEchoContract.Echo**
     16. **Console.WriteLine("Echoing: {0}", text)**
     17. **Return text**
     18. **End Function**
     19. End Class

Task 2 – Hosting the WCF Service through the AppFabric Service Bus

* 1. In this task, you add code to expose the **EchoService** service through theService Bus. The service will then stay listening for incoming messages at a designated URI.
  2. To communicate across theService Bus, WCF extensibility points have been used to define a new set of bindings available for use with the Service Bus. A WCF binding determines what kind of communication will be exposed or used by a service endpoint. In this lab, you use a binding named **NetTcpRelayBinding**, which is one of the bindings introduced with the Service Bus. On the service or listener side, the **NetTcpRelayBinding** enables you to expose an endpoint that is accessible through a global URI supported by theService Bus. When an endpoint using **NetTcpRelayBinding** is opened, its URI is registered within the Service Busand from that moment, any authorized client will be able to send messages to that global URI using a similarly configured client. TheService Bus then becomes the conduit for communication between those two endpoints.
  3. Open the **Program.cs** file (for Visual C# projects) or **Module1.vb** file (for Visual Basic projects) in the **Service** project and declare the following namespaces below the existing namespace directives.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Namespaces - CS*)
     2. C#
     3. **using System.ServiceModel;**
     4. **using System.ServiceModel.Description;**
     5. **using Microsoft.ServiceBus;**
     6. **using Microsoft.ServiceBus.Description;**
     7. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Namespaces - VB*)
     8. Visual Basic
     9. **Imports System.ServiceModel**
     10. **Imports System.ServiceModel.Description**
     11. **Imports Microsoft.ServiceBus**
     12. **Imports Microsoft.ServiceBus.Description**
  4. Next, paste the following code inside the **Main** method. This will prompt the user to provide a **service namespace domain**, an **issuer name**, and an **issuer secret**.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Main Header - CS*)
     2. C#
     3. static void Main(string[] args)
     4. {
     5. **Console.Title** = "**Service**";
     6. **Console.Write("Your Service Namespace Domain (e.g. sb://<YOUR-NAMESPACE>.servicebus.windows.net/): ");**
     7. **string serviceNamespaceDomain = Console.ReadLine();**
     8. **Console.Write("Your Issuer Name: ");**
     9. **string issuerName = Console.ReadLine();**
     10. **Console.Write("Your Issuer Secret: ");**
     11. **string issuerSecret = Console.ReadLine();**
     12. }
     13. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Main Header - VB*)
     14. Visual Basic
     15. Sub Main()
     16. **Console.Title = "Service"**
     17. **Console.Write("Your Service Namespace Domain (e.g. sb://<YOUR-NAMESPACE>.servicebus.windows.net/): ")**
     18. **Dim serviceNamespaceDomain As String = Console.ReadLine()**
     19. **Console.Write("Your Issuer Name: ")**
     20. **Dim issuerName As String = Console.ReadLine()**
     21. **Console.Write("Your Issuer Secret: ")**
     22. **Dim issuerSecret As String = Console.ReadLine()**
     23. End Sub
  5. Create the URI for the service based on the service namespace; this is the address where the service will be hosted. To do this, add the following code (shown in **bold**).
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Create Uri - CS*)
     2. C#
     3. static void Main(string[] args)
     4. {
     6. **// create the service URI based on the service namespace**
     7. **Uri address = ServiceBusEnvironment.CreateServiceUri("sb",**
     8. **serviceNamespaceDomain, "EchoService");**
     9. }
     10. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Create Uri - VB*)
     11. Visual Basic
     12. Sub Main()

     15. **' create the service URI based on the service namespace**
     16. **Dim address As Uri = ServiceBusEnvironment.CreateServiceUri("sb", serviceNamespaceDomain, "EchoService")**
     17. End Sub
     18. **Note:** The URI points to the address where the service will be hosted. The generated URI will have the following format: *sb://<serviceNamespace>.servicebus.windows.net/EchoService/*
     19. Notice that to create the URI, you are passing "*sb*" as the scheme, followed by the **Service Namespace** identity. “EchoService” is being used as part of the URI to host the service.
  6. Create a **TransportClientEndpointBehavior** to specify which type of credentials should be used to authenticate the service with the Service Bus. Specify a **CredentialType** of type **SharedSecret**,which relies on the **issuer name** and **issuer key** obtained when you signed up for AppFabric in the Getting Started section.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Create Credential object for the Endpoint -CS*)
     2. C#
     3. static void Main(string[] args)
     4. {
     5. // ...
     7. // create the service URI based on the service namespace
     8. Uri address = ServiceBusEnvironment.CreateServiceUri("sb", serviceNamespaceDomain, "EchoService");
     9. **// create the credential object for the endpoint**
     10. **TransportClientEndpointBehavior sharedSecretServiceBusCredential = new TransportClientEndpointBehavior();**
     11. **sharedSecretServiceBusCredential.CredentialType = TransportClientCredentialType.SharedSecret;**
     12. **sharedSecretServiceBusCredential.Credentials.SharedSecret.IssuerName = issuerName;**
     13. **sharedSecretServiceBusCredential.Credentials.SharedSecret.IssuerSecret = issuerSecret;**
     14. }
     15. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Create Credential object for the Endpoint -VB*)
     16. Visual Basic
     17. Sub Main()
     18. ' ...
     19. ' create the service URI based on the service namespace
     20. Dim address As Uri = ServiceBusEnvironment.CreateServiceUri("sb", serviceNamespaceDomain, "EchoService")
     21. **' create the credential object for the endpoint**
     22. **Dim sharedSecretServiceBusCredential As New TransportClientEndpointBehavior()**
     23. **sharedSecretServiceBusCredential.CredentialType = TransportClientCredentialType.SharedSecret**
     24. **sharedSecretServiceBusCredential.Credentials.SharedSecret.IssuerName = issuerName**
     25. **sharedSecretServiceBusCredential.Credentials.SharedSecret.IssuerSecret = issuerSecret**
     26. End Sub
     27. **Note:** Authentication and authorization are both performed by the AppFabric Access Control. To make these steps simple, the **Microsoft.ServiceBus** namespace contains a set of transport client credential helpers that automatically handle the acquisition of the required security tokens. The **transportClientEndpointBehavior** allows you to specify the Credential Type to be used to authenticate the service. In this case, you are using **SharedSecret**, which relies on the **issuer name** and **issuer key** for authentication with the Service Bus.
     28. As well as **SharedSecret**, there are other credential types:
     29. **- SimpleWebToken:** The client authenticates with the Service Bus using a *SimpleWebToken* credential (the *SharedSecretCredential.ComputeSimpleWebTokenString(string issuerName, string issuerSecret)* method computes the string that is expected by TransportClientEndpointBehavior).

**- Unauthenticated:** No client credential provided. In the service configuration file, the security configuration for all of the relay bindings contains a *relayClientAuthenticationType* setting that controls whether or not a client is required to authenticate when accessing the HTTP service. Setting this to “None” means that access is granted to anyone consuming the service. A typical scenario for unauthenticated services is exposing RSS feeds.

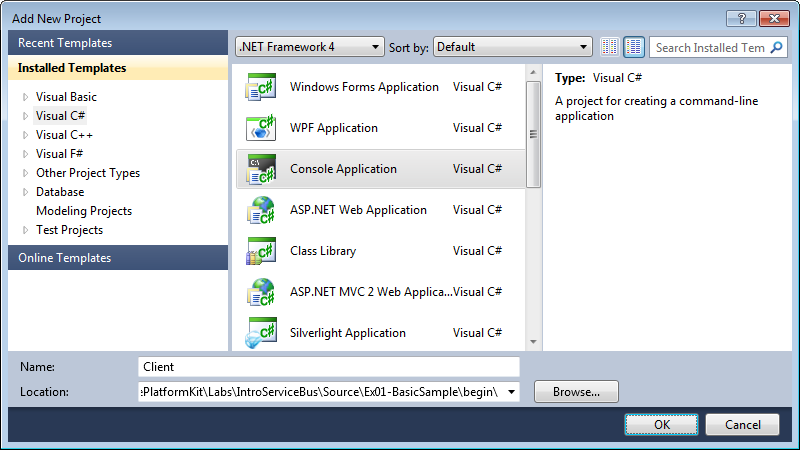
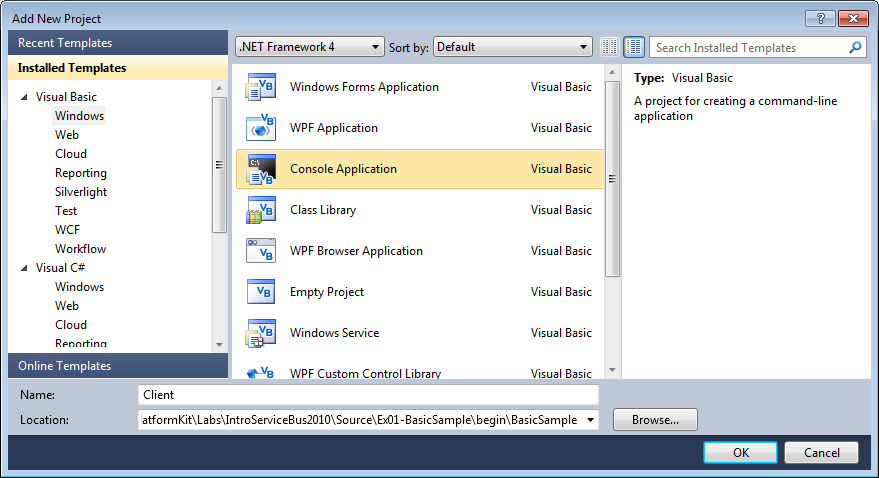
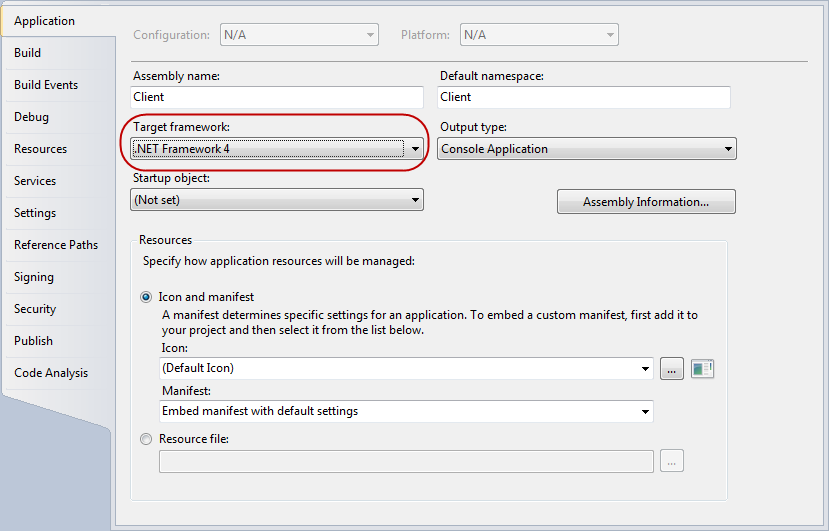
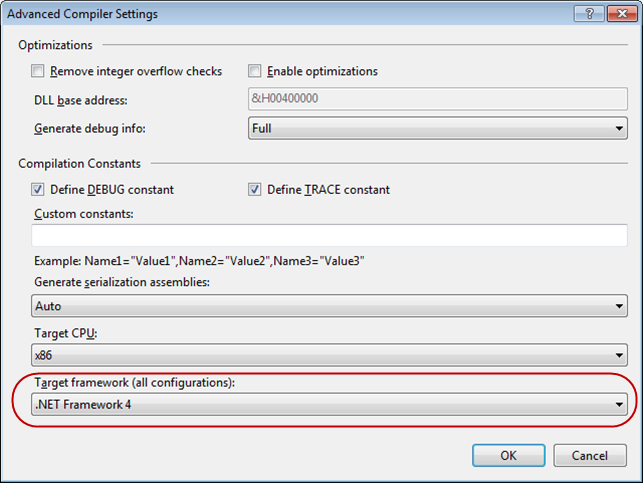
* 1. Create a **ServiceHost** object to host the **EchoService** as a WCF Service**.** Paste the following code below the one inserted previously.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Create Service Host - CS*)
     2. C#
     3. static void Main(string[] args)
     4. {
     5. // ...
     7. **// create the service host reading the configuration**
     8. **ServiceHost host = new ServiceHost(typeof(EchoService), address);**
     9. }
     10. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Create Service Host - VB*)
     11. Visual Basic
     12. Sub Main()
     13. ' ...
     15. **' create the service host reading the configuration**
     16. **Dim host As New ServiceHost(GetType(EchoService), address)**
     17. End Sub
  2. To associate a listener endpoint with its Service Bus credentials, you must add a **TransportClientEndpointBehavior** behavior to the respective endpoint’s behavior collection. If your service exposes multiple endpoints through the relay, you can add the same behavior instance to each endpoint. Paste the following code below the one inserted previously in order to add the behavior for all endpoints.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Add Service Bus Credentials to Endpoints - CS*)
     2. C#
     3. static void Main(string[] args)
     4. {
     5. // ...
     7. // create the service host reading the configuration
     8. ServiceHost host = new ServiceHost(typeof(EchoService), address);
     9. **// create the ServiceRegistrySettings behavior for the endpoint**
     10. **IEndpointBehavior serviceRegistrySettings = new ServiceRegistrySettings(DiscoveryType.Public);**
     11. **// add the Service Bus credentials to all endpoints specified in configuration**
     12. **foreach (ServiceEndpoint endpoint in host.Description.Endpoints)**
     13. **{**
     14. **endpoint.Behaviors.Add(sharedSecretServiceBusCredential);**
     15. **}**
     16. }
     17. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Add Service Bus Credentials to Endpoints - VB*)
     18. Visual Basic
     19. Sub Main()
     20. ' ...
     22. ' create the service host reading the configuration
     23. Dim host As New ServiceHost(GetType(EchoService), address)
     24. **' add the Service Bus credentials to all endpoints specified in configuration**
     25. **For Each endpoint As ServiceEndpoint In host.Description.Endpoints**
     26. **endpoint.Behaviors.Add(sharedSecretServiceBusCredential)**
     27. **Next endpoint**
     28. End Sub
  3. Add the code to open the host and have it listen at the relay service until the user presses any key. To do this, add the following code to method **Main**.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Open Service Host - CS*)
     2. C#
     3. static void Main(string[] args)
     4. {
     5. // ...
     7. **// open the service**
     8. **host.Open();**
     9. **Console.WriteLine("Service address: " + address);**
     10. **Console.WriteLine("Press [Enter] to exit");**
     11. **Console.ReadLine();**
     12. **// close the service**
     13. **host.Close();**
     14. }
     15. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Open Service Host - VB*)
     16. Visual Basic
     17. Sub Main()
     19. ' ...
     20. **' open the service**
     21. **host.Open()**
     22. **Console.WriteLine("Service address: " & address.ToString())**
     23. **Console.WriteLine("Press [Enter] to exit")**
     24. **Console.ReadLine()**
     25. **' close the service**
     26. **host.Close()**
     27. End Sub
     28. **Note:** When you call **ServiceHost.Open**, the **NetTcpRelayBinding** causes WCF to communicate with theService Bus in order to authenticate the service identity and to authorize it to listen at the specified URI.

Task 3 – Configuring the WCF Service

* 1. In this task, you will configure the WCF Service.
  2. In the **app.config** file, add the following system.serviceModel configuration section to define the **EchoService**.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex01 Service Model - XML*)
     2. XML
     3. <configuration>
     4. **<system.serviceModel>**
     5. **<services>**
     6. **<service name="Service.EchoService">**
     7. **</service>**
     8. **</services>**
     9. **</system.serviceModel>**
     10. ...
     11. </configuration>
  3. Add an endpoint definition to the service. To do this, add the following code (shown in **bold**) inside the service configuration.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex01 Service Endpoint 1 - XML*)
     2. XML
     3. <!-- Application Service -->
     4. <service name="Service.EchoService">
     5. **<endpoint contract="Service.IEchoContract"**
     6. **binding="netTcpRelayBinding" />**
     7. </service>
     8. **Note:** The endpoints are configured to use a binding of type **netTcpRelayBinding**.

Task 4 – Creating the WCF Client Project

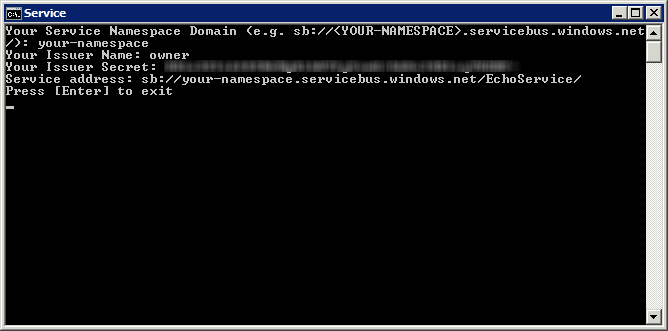
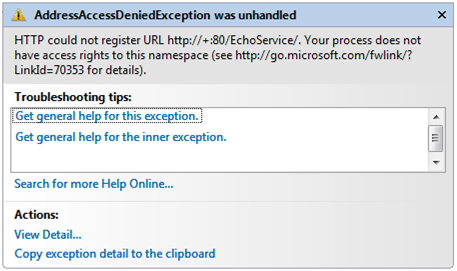
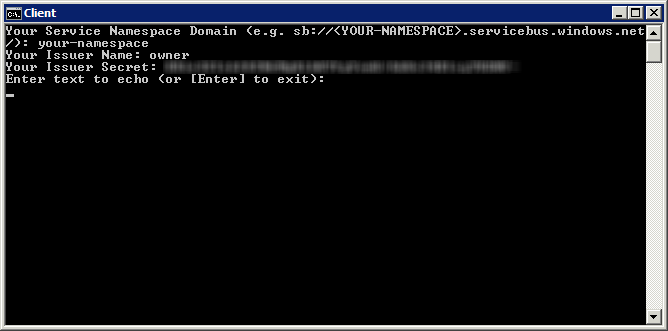
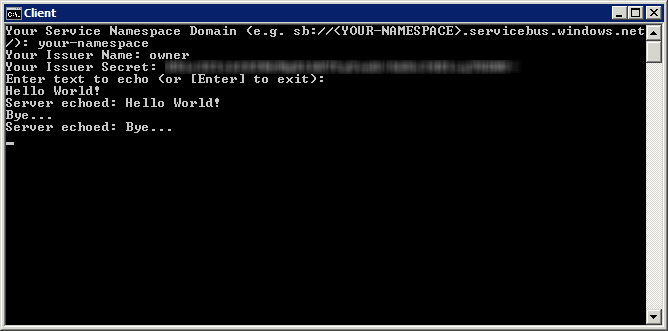
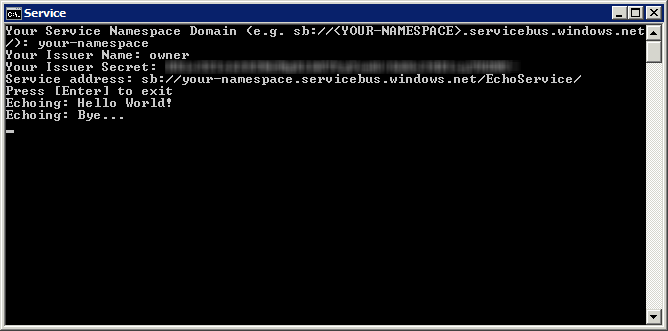
In this task, you create the project for the client WCF application and define the proxy to communicate with the service.

* 1. Create the **Client** project. To do this, in the **File** menu, point to **Add** and then select **New Project**. In the **New Project** dialog, select the **Console Application** template, ensure that the target framework is set to **.NET Framework 4**, set the project name to **Client** and click **OK**.
     1. 
     2. Figure 16
     3. Creating the client project (C#)
     4. 
     5. Figure 17
     6. Creating the client project (Visual Basic)
  2. Change the **Target Framework** to **.NET Framework 4** to allow the Service Bus interaction. To do this, in **Solution Explorer**, right-click the **Client** project and select **Properties.** 
     1. For **Visual C#** projects: In the **Properties** window, select the **Application** tab and choose **.NET** **Framework 4** in the **Target Framework** option. If prompted to confirm the target framework change, click **Yes** to proceed.
     2. For **Visual Basic** projects: In the **Properties** window, select the **Compile** tab, click **Advance Compile Options** , choose **.NET Framework 4** in the **Target Framework** option, and click **OK**. If prompted to confirm the target framework change, click **Yes** to proceed
     3. 
     4. Figure 18
     5. Choosing the target framework for C# projects
     6. 
     7. Figure 19
     8. Choosing the target framework for Visual Basic projects
  3. Add a reference to the **ServiceModel** assembly. To do this, in **Solution Explorer**, right-click the **Client** project and select **Add Reference**. In the **Add Reference** dialog, choose the **.NET** tab, select the **System**.**ServiceModel** assembly. Click **OK** to add the reference.
  4. Now, add a reference to the Service Bus core assembly. Again, right-click the **Client** project in **Solution Explorer**, select **Add Reference** to open the **Add Reference** dialog, select the **.NET** tab, choose **Microsoft.ServiceBus** and click **OK**.
     1. **Note:** If you cannot find the **Microsoft.ServiceBus** assembly in the .NET tab, use the **Browse** tab to locate this assembly inside the **%ProgramFiles%\Windows Azure AppFabric SDK\V1.0\Assemblies\NET4.0** folder.
  5. Create the Echo client proxy. To do this, in **Solution Explorer**, right-click the **Client** project, point to **Add** and select **New Item**. In the **Add New Item** dialog box, choose **Code** in the categories list, select the **Interface** template, set the **Name** to **EchoProxy.cs** (for Visual C# projects) or **EchoProxy.vb** (for Visual Basic projects)and click **Add**.
  6. In the **EchoProxy.cs** file (for Visual C# projects) or the **EchoProxy.vb** file (for Visual Basic projects), add the following namespace directive.
     1. C#
     2. **using System.ServiceModel;**
     3. Visual Basic
     4. **Imports System.ServiceModel**
  7. Define the interface. To do this, replace the current interface definition (**EchoProxy**) with the following code (shown in **bold**).
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Echo Proxy Contract - CS*)
     2. C#
     3. **[ServiceContract(Name = "EchoContract", Namespace = "http://samples.microsoft.com/ServiceModel/Relay/")]**
     4. **public interface IEchoContract**
     5. **{**
     6. **}**
     7. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Echo Proxy Contract - VB*)
     8. Visual Basic
     9. **<ServiceContract(Name:="EchoContract", Namespace:="http://samples.microsoft.com/ServiceModel/Relay/")> \_**
     10. **Public Interface IEchoContract**
     11. **End Interface**
  8. Define the **Echo** operation contract. To do this, paste the following code inside the **IEchoContract** class.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex01 OperationContract IEchoContract* -CS)
     2. C#
     3. [ServiceContract(Name = "EchoContract", Namespace = "http://samples.microsoft.com/ServiceModel/Relay/")]
     4. public interface IEchoContract
     5. {
     6. **[OperationContract]**
     7. **string Echo(string text);**
     8. }
     9. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex01 OperationContract IEchoContract* -VB)
     10. Visual Basic
     11. <ServiceContract(Name:="EchoContract", Namespace:="http://samples.microsoft.com/ServiceModel/Relay/")> \_
     12. Public Interface IEchoContract
     13. **<OperationContract()> \_**
     14. **Function Echo(ByVal text As String) As String**
     15. End Interface

Task 5 – Consuming the WCF Service Hosted through the App Fabric Service Bus

* 1. In this task, you create the service proxy and connect to the hosted service.
  2. In **Solution Explorer**, locate the **Client** project, double-click **Program.cs** (for Visual C# projects) or **Module1.vb** file (for Visual Basic projects) inside it and paste the following code below the existing namespace directives at the top of the file.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 - Ex01 Namespaces Program -CS)
     2. C#
     3. **using System.ServiceModel;**
     4. **using Microsoft.ServiceBus;**
     5. (Code Snippet - Introduction to Service Bus Lab Part 1 - Ex01 Namespaces Module1 -VB)
     6. Visual Basic
     7. **Imports System.ServiceModel**
     8. **Imports Microsoft.ServiceBus**
  3. Add code to let you enter the service namespace and credentials and then create a channel to the service. The authentication type used is **SharedSecret**; the same configuration that was used in step 4 of task 2. To do this, paste the following code inside method **Main**.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Client - CS*)
     2. C#
     3. static void Main(string[] args)
     4. {
     5. **Console.Title** = "**Client**";
     6. **Console.Write("Your Service Namespace Domain (e.g. sb://<YOUR-NAMESPACE>.servicebus.windows.net/): ");**
     7. **string serviceNamespaceDomain = Console.ReadLine();**
     8. **Console.Write("Your Issuer Name: ");**
     9. **string issuerName = Console.ReadLine();**
     10. **Console.Write("Your Issuer Secret: ");**
     11. **string issuerSecret = Console.ReadLine();**
     12. **// create the service URI based on the service namespace**
     13. **Uri serviceUri = ServiceBusEnvironment.CreateServiceUri("sb", serviceNamespaceDomain, "EchoService");**
     14. **// create the credentials object for the endpoint**
     15. **TransportClientEndpointBehavior sharedSecretServiceBusCredential = new TransportClientEndpointBehavior();**
     16. **sharedSecretServiceBusCredential.CredentialType = TransportClientCredentialType.SharedSecret;**
     17. **sharedSecretServiceBusCredential.Credentials.SharedSecret.IssuerName = issuerName;**
     18. **sharedSecretServiceBusCredential.Credentials.SharedSecret.IssuerSecret = issuerSecret;**
     19. **// create the channel factory loading the configuration**
     20. **ChannelFactory<IEchoContract> channelFactory = new ChannelFactory<IEchoContract>("RelayEndpoint", new EndpointAddress(serviceUri));**
     21. **// apply the Service Bus credentials**
     22. **channelFactory.Endpoint.Behaviors.Add(sharedSecretServiceBusCredential);**
     23. **// create and open the client channel**
     24. **IEchoContract channel = channelFactory.CreateChannel();**
     25. **((ICommunicationObject)channel).Open();**
     26. }
     27. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Client - VB*)
     28. Visual Basic
     29. Sub Main()
     30. **Console.Title = "Client"**
     31. **Console.Write("Your Service Namespace Domain (e.g. sb://<YOUR-NAMESPACE>.servicebus.windows.net/): ")**
     32. **Dim serviceNamespaceDomain As String = Console.ReadLine()**
     33. **Console.Write("Your Issuer Name: ")**
     34. **Dim issuerName As String = Console.ReadLine()**
     35. **Console.Write("Your Issuer Secret: ")**
     36. **Dim issuerSecret As String = Console.ReadLine()**
     37. **' create the service URI based on the service namespace**
     38. **Dim serviceUri As Uri = ServiceBusEnvironment.CreateServiceUri("sb", serviceNamespaceDomain, "EchoService")**
     39. **' create the credentials object for the endpoint**
     40. **Dim sharedSecretServiceBusCredential As New TransportClientEndpointBehavior()**
     41. **sharedSecretServiceBusCredential.CredentialType = TransportClientCredentialType.SharedSecret**
     42. **sharedSecretServiceBusCredential.Credentials.SharedSecret.IssuerName = issuerName**
     43. **sharedSecretServiceBusCredential.Credentials.SharedSecret.IssuerSecret = issuerSecret**
     44. **' create the channel factory loading the configuration**
     45. **Dim channelFactory As New ChannelFactory(Of IEchoContract)("RelayEndpoint", New EndpointAddress(serviceUri))**
     46. **' apply the Service Bus credentials**
     47. **channelFactory.Endpoint.Behaviors.Add(sharedSecretServiceBusCredential)**
     48. **' create and open the client channel**
     49. **Dim channel As IEchoContract = channelFactory.CreateChannel()**
     50. **CType(channel, ICommunicationObject).Open()**
     51. End Sub
  4. Add code to request the user to enter a message to be sent to the service and then proceed to send it (by calling **EchoChannel**.**Echo**). If the user enters an empty string, the application ends.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Client User Interaction - CS*)
     2. C#
     3. static void Main(string[] args)
     4. {
     5. ...
     6. ((ICommunicationObject)channel).Open();
     7. **Console.WriteLine("Enter text to echo (or [Enter] to exit):");**
     8. **string input = Console.ReadLine();**
     9. **while (input != String.Empty)**
     10. **{**
     11. **try**
     12. **{**
     13. **Console.WriteLine("Server echoed: {0}", channel.Echo(input));**
     14. **}**
     15. **catch (Exception e)**
     16. **{**
     17. **Console.WriteLine("Error: " + e.Message);**
     18. **}**
     19. **input = Console.ReadLine();**
     20. **}**
     21. }
     22. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Client User Interaction - VB*)
     23. Visual Basic
     24. Sub Main()
     25. ...
     26. ' create and open the client channel
     27. Dim channel As IEchoContract = channelFactory.CreateChannel()
     28. CType(channel, ICommunicationObject).Open()
     29. **Console.WriteLine("Enter text to echo (or [Enter] to exit):")**
     30. **Dim input As String = Console.ReadLine()**
     31. **Do While input <> String.Empty**
     32. **Try**
     33. **Console.WriteLine("Server echoed: {0}", channel.Echo(input))**
     34. **Catch e As Exception**
     35. **Console.WriteLine("Error: " & e.Message)**
     36. **End Try**
     37. **input = Console.ReadLine()**
     38. **Loop**
     39. End Sub
     40. **Note:** When the relay service receives an incoming message from an authorized sender (**Client**), it simply relays the message to an instance of the **EchoService** running locally. **EchoService** creates an application response and sends it back to the relay service.
     41. The relay service completes the message flow by sending the response to the client containing the **EchoService** response message.
  5. Add code to close both the **Channel** and the **ChannelFactory**.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Close Channel - CS*)
     2. C#
     3. static void Main(string[] args)
     4. {
     5. ...
     6. while (input != String.Empty)
     7. {
     8. ...
     9. }
     10. **((ICommunicationObject)channel).Close();**
     11. **channelFactory.Close();**
     12. }
     13. (Code Snippet - Introduction to Service Bus Lab Part 1 *- Ex01 Close Channel - VB*)
     14. Visual Basic
     15. Sub Main()
     16. ...
     17. Do While input <> String.Empty
     18. ...
     19. Loop
     20. **CType(channel, ICommunicationObject).Close()**
     21. **channelFactory.Close()**
     22. End Sub
  6. Add an endpoint and binding configuration for the proxy used in the client side. To do this, paste the following code in **app.config**.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex01 Service Endpoint 2- XML*)
     2. XML
     3. <configuration>
     4. **<system.serviceModel>**
     5. **<client>**
     6. **<!-- Application Endpoint -->**
     7. **<endpoint name="RelayEndpoint"**
     8. **contract="Client.IEchoContract"**
     9. **binding="netTcpRelayBinding"/>**
     10. **</client>**
     11. **</system.serviceModel>**
     12. ...
     13. </configuration>

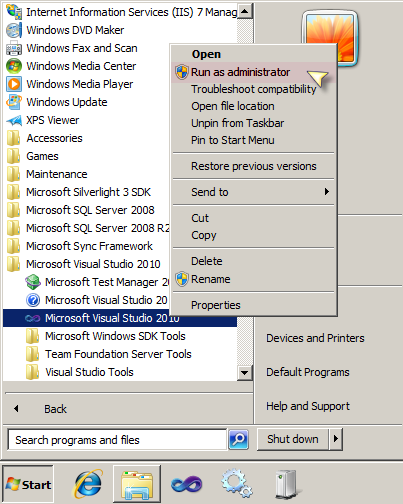
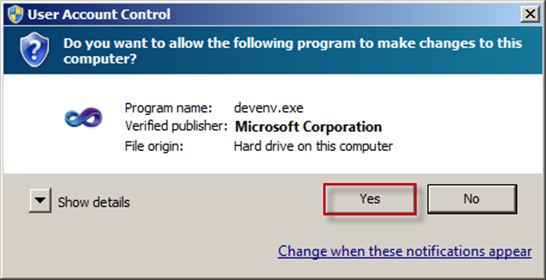
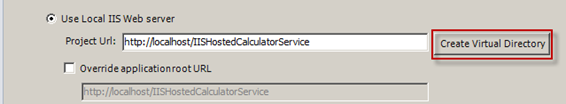
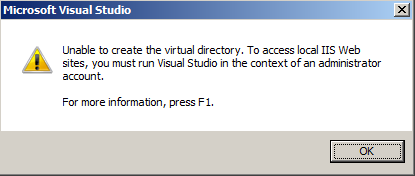
Verification

* 1. In order to verify that you have performed every step in the exercise correctly, proceed as follows:
  2. **Note:** You require an AppFabric project and a service namespace to complete this exercise. If you have not already done so, complete the Getting Started section.
  3. Launch the service. To do this, right-click the **Service** project, point to **Debug** and select **Start New Instance**.
  4. Enter your **Service Namespace Domain**; this is the service namespaceyou entered when provisioning your account at the portal Web Site. After that, enter your **Default Issuer Name** and **Default Issuer Key** for the service namespace; these are the name and key generated for the service namespace at the portal Web Site that you recorded during the Getting Started section.
     1. 
     2. Figure 20
     3. Starting the service application
     4. **Note:** HTTP services require administrative privileges in order to register a URL. For instance, if you see the following error message:
     5. 
     6. It means that the namespace where the service is being hosted has not been reserved yet. To solve this, stop debugging, run the following command with administrative privileges, and then launch the service again:
     7. **netsh http add urlacl url=http://+:80/EchoService/ user=DOMAIN\user**
     8. Where **DOMAIN\user** is the user that will run the **Service** console application.
     9. **Notice that you could also workaround this by launching Visual Studio as an administrator.**
  5. Start the WCF **Client**. To do this, in **Solution Explorer**, right-click the **Client** project, point to **Debug** and select **Start** **new** **instance**.
  6. Once again, enter the **Service Namespace Domain**, **Default Issuer Name** and **Default Issuer Key** of your service namespace.
     1. 
     2. Figure 21
     3. Starting the client application
     4. **Note:** In this hands-on-lab, both the client and the service use the same credentials; therefore, no additional steps are required to grant the client access to listen or to send a message to the service through the Service Bus. By default, a credential is granted *Listen*, *Send*, and *Manage* privileges within its own Service Bus service namespace. If the client is using a different credential than the service, a specific Access Control rule is needed for a client credential to gain *Listen*, *Send* or *Manage* privileges in the service namespace for the service.
  7. Type some messages at the **Client** console. Messages will be sent to the **Service** via the Service Bus.
     1. 
     2. Figure 22
     3. Messages sent to the hosted service
  8. The **Service** receives the messages via the relay service and outputs them to the console window.
     1. 
     2. Figure 23
     3. Service receives and outputs the messages

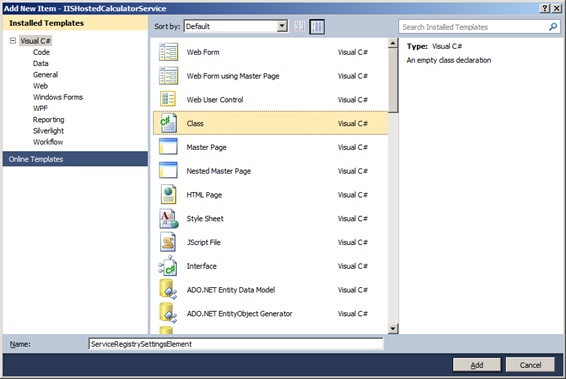
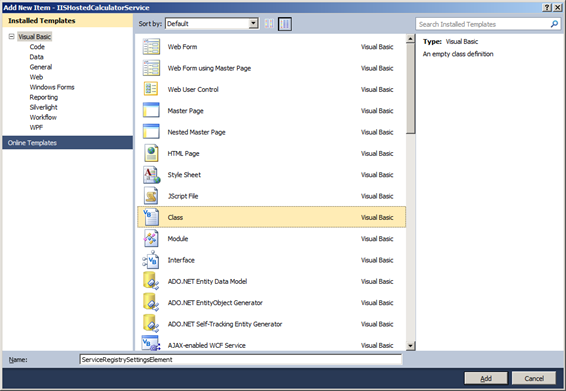
Exercise 2: Connecting a WCF Service in IIS 7.5 to the Service Bus

* 1. In this exercise you will connect a WCF Service hosted in IIS 7.5 to the Service Bus. Additionally, you will configure the service so it automatically activates once IIS 7.5 starts running. This configuration provides a solution to an important challenge when hosting a WCF service using Service Bus bindings.
  2. In order for the WCF service to start receiving messages from the Service Bus in the cloud (the Relay Service), the service opens an outbound port and creates a bidirectional socket for communication. It connects to the Service Bus, authenticates itself, and starts listening to calls from the relay service before the client sends its requests. IIS\WAS, on the other hand, relies on message-based activation and will launch the host only after the first request comes in.
  3. Consequently, until the first message is received by IIS the service will never establish a connection to the Service Bus; but with no connection to the Service Bus, it will never receive a message.
  4. During the exercise, you will use the Windows App Fabric Auto-Start feature to get the service up and running as soon as the application that it belongs to is started and before the service receives the first WCF message from the client, thus overcoming this limitation.

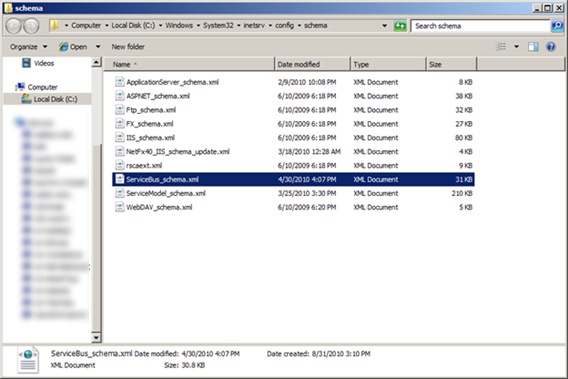
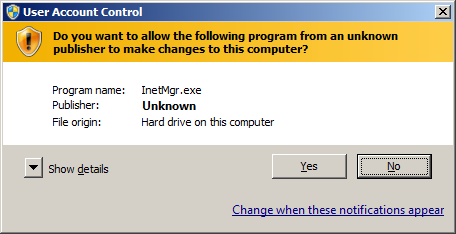
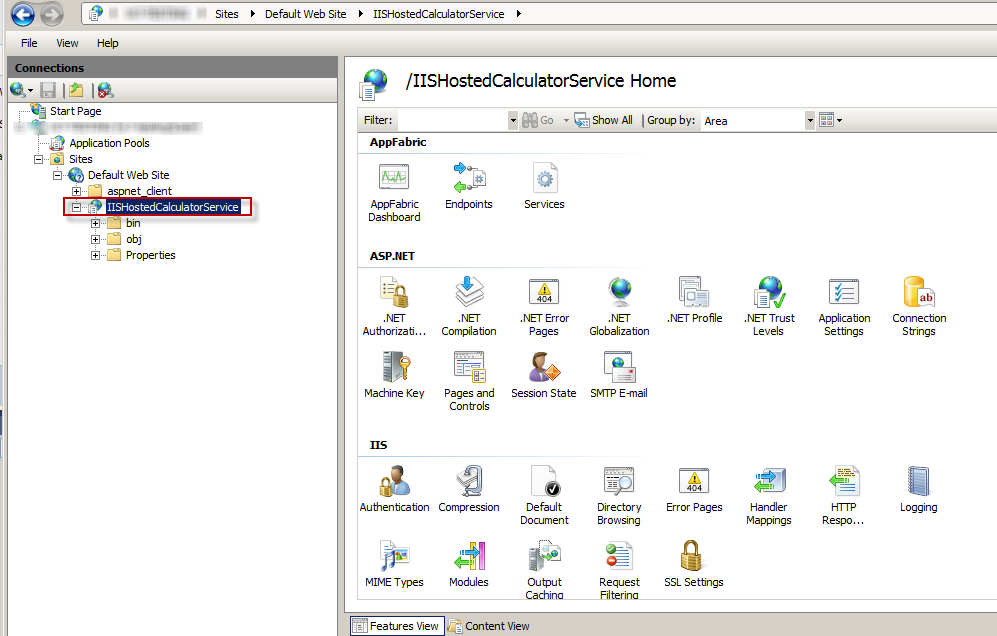
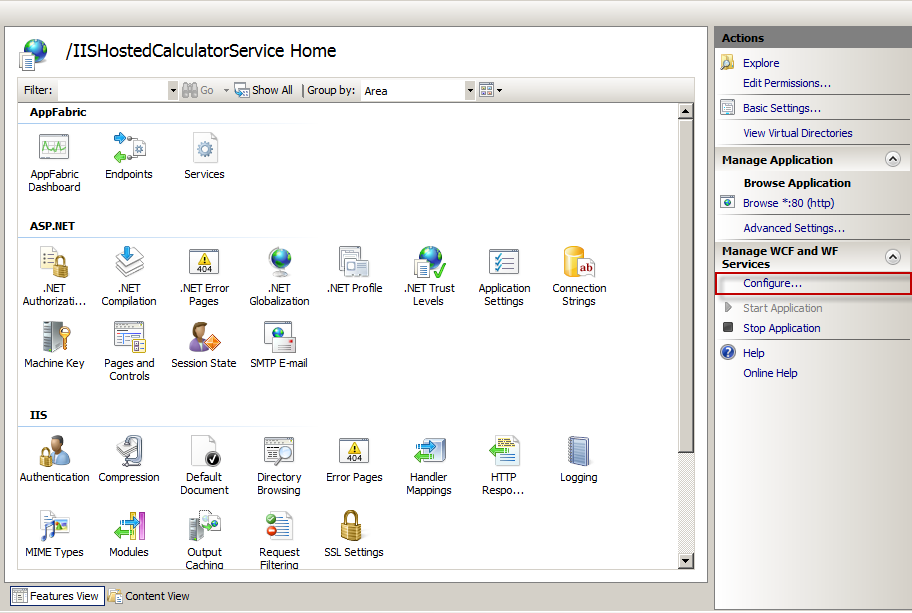
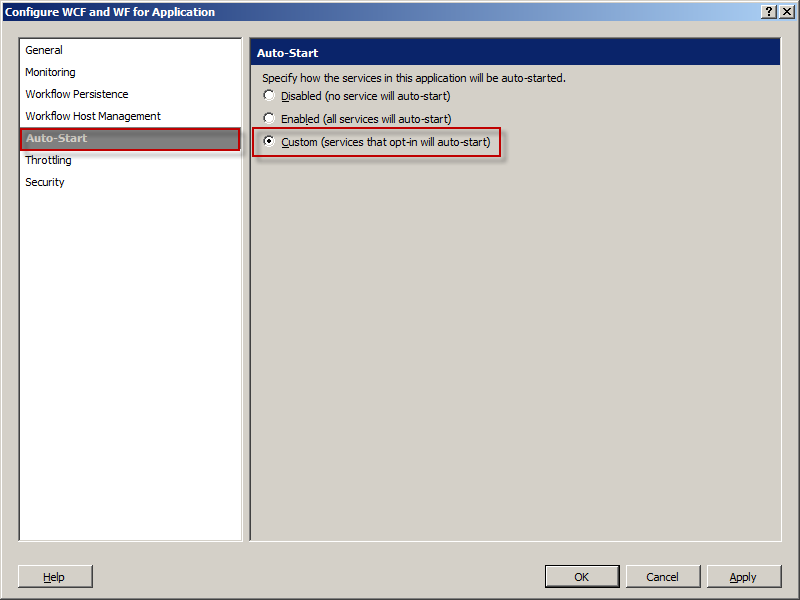
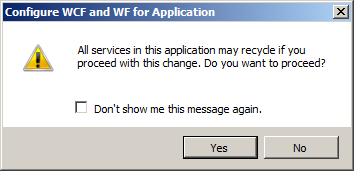
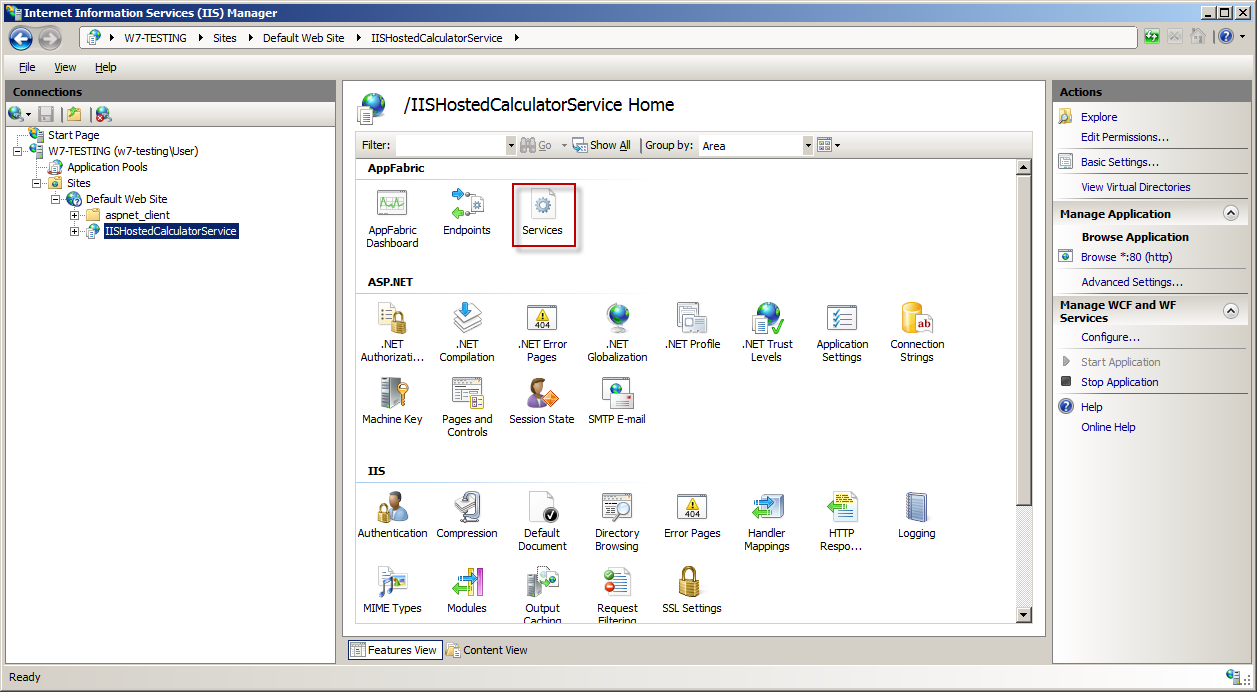
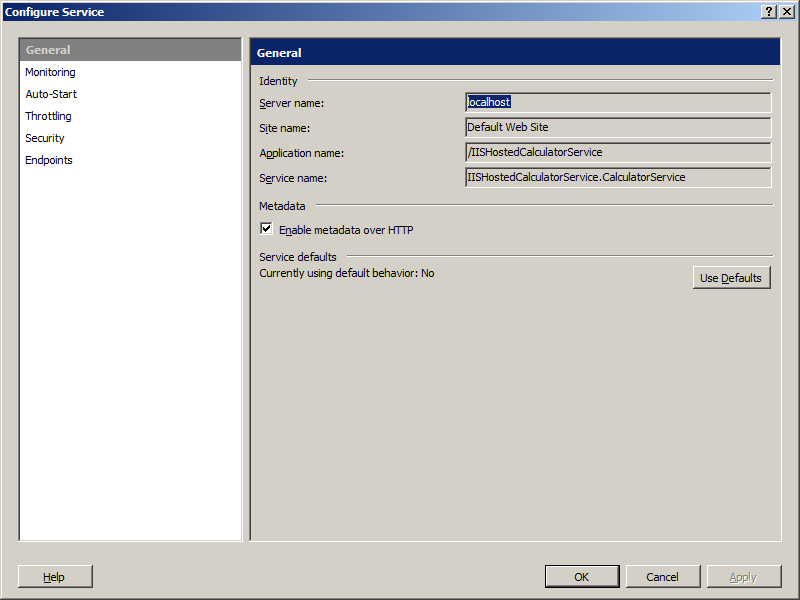
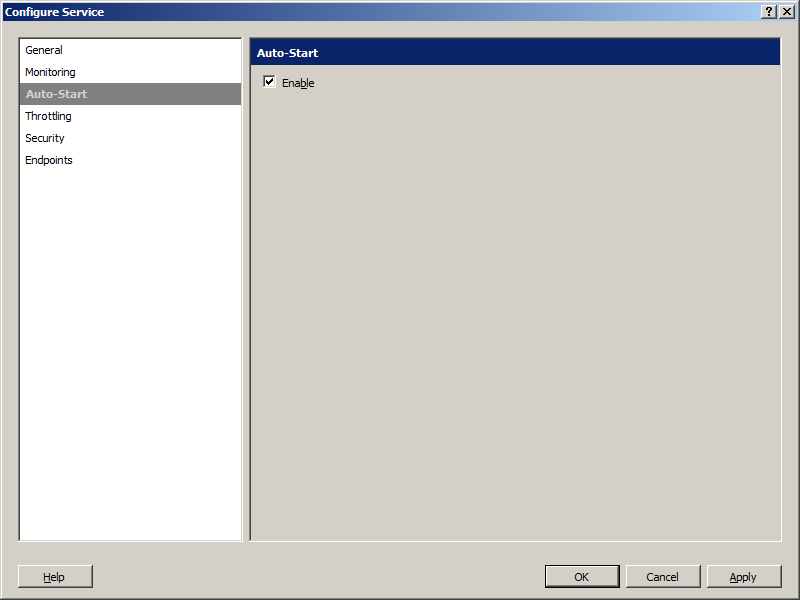
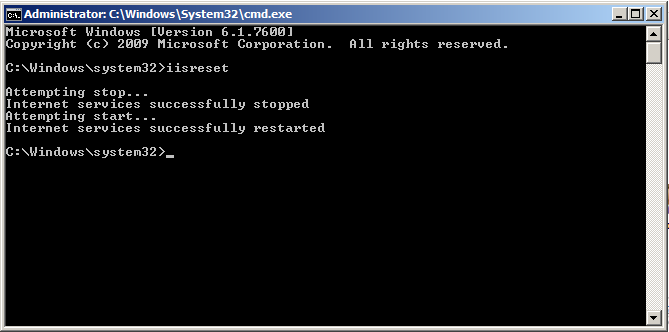
Task 1 – Creating a Virtual Directory For The Application

* 1. In this task you create virtual directory for the web site which hosts the WCF service.
  2. Open Microsoft Visual Studio 2010 with elevated permissions. To do this, go to **Start | All Programs | Microsoft Visual Studio 2010,** right-click the **Microsoft Visual Studio 2010** option and select **Run as Administrator**. This is shown in the following figure.
     1. 
     2. Figure 24
     3. Running Visual Studio as administrator from the Start menu
  3. Click **Yes** in the **User** **Account** **Control** dialog. Visual Studio will open with Administrator privileges.
     1. 
     2. Figure 25
     3. Clicking Yes in the User Account Control dialog opens VS with Administrator privileges
  4. In Visual Studio, open the **IISHostedCalculatorService**.**sln** solution file from Ex02-IISHostedService\begin\{CS|VB} in the Source folder of the lab.
  5. In **Solution** **Explorer**, right-click the **IISHostedCalculatorService** project and select **Properties**.
  6. Update the server to be used by the **IISHostedCalculatorService** web site. To do this, in the **Web** tab in the **Properties** window, select the **Use Local IIS Web Server** option.
  7. Click **Create Virtual Directory** to create a virtual directory for the Web Site in IIS.
     1. 
     2. Figure 26
     3. Creating a Virtual Directory in IIS for the Web Site
     4. **Note:** If you are not running Visual Studio with elevation you will receive an error message similar to the one shown in the following figure.
     5. 
     6. Figure 27
     7. Error message when trying to create a virtual directory without elevated permissions
  8. In Visual Studio, save all changes by clicking the **Save** **All** button.

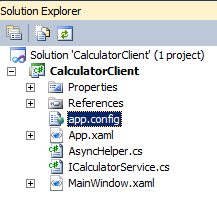
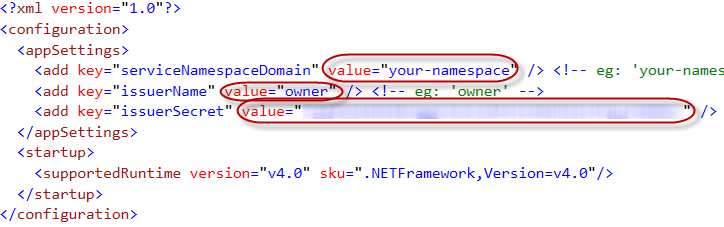
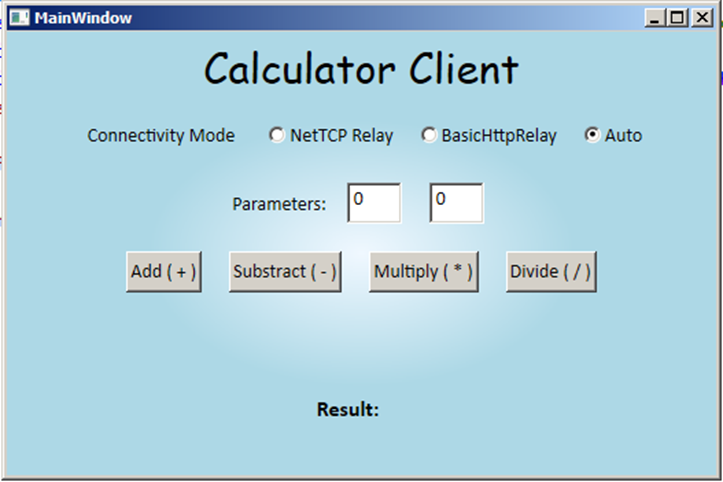
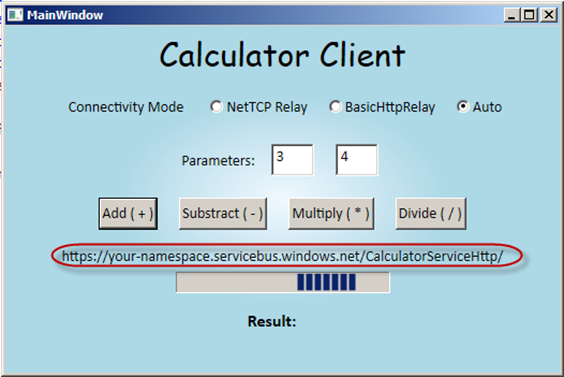
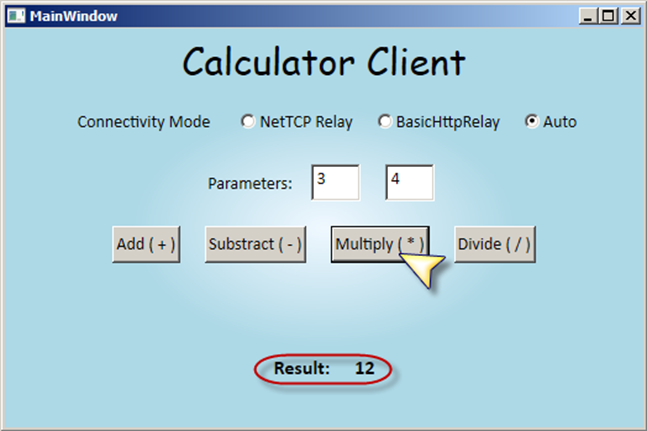
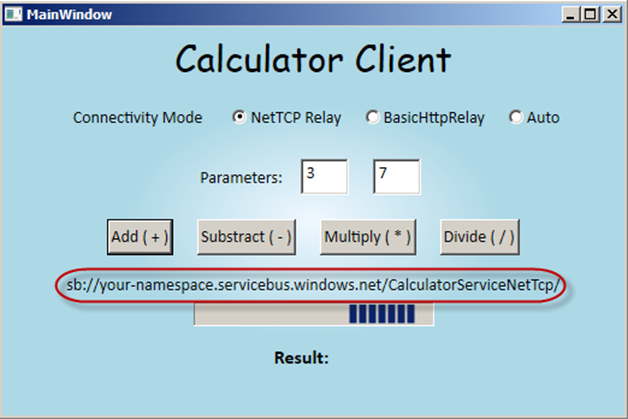
Task 2 – Configuring the WCF Service to be discoverable

* 1. In this task you will update the service's configuration to be able to expose it to the Service Bus. To do this, you will create a custom configuration element that will enable you to set the discoverability policy to "Public" in the configuration file.
  2. Add a reference to the **Microsoft.ServiceBus.dll** assembly. To do this, right-click the **IISHostedCalculatorService** project in **Solution Explorer**, select **Add Reference** to open the **Add Reference** dialog, select the **.NET** tab, choose **Microsoft.ServiceBus.dll** for .NET Framework 4.0 and click **OK**.
     1. **Note:** If you cannot find the assembly in the .NET tab, use the Browse tab to locate it inside the C:\%Program Files%\Windows Azure AppFabric SDK\V1.0\Assemblies\NET4.0 folder.
  3. In **Solution** **Explorer**, right-click the **IISHostedCalculatorService**,point to **Add** and click **Class**. The **Add** **New** **Item** dialog appears.
  4. Name the class **ServiceRegistrySettingsElement**, as shown in the following figure.
     1. 
     2. Figure 28
     3. Adding the ServiceRegistrySettingsElement class to the IISHostedCalculatorService project (C#)
     4. 
     5. Figure 29
     6. Adding the ServiceRegistrySettingsElement class to the IISHostedCalculatorService project (Visual Basic)
  5. Click **Add**. In the following steps you will make this class accessible as a **BehaviorExtension** configuration element. The reason for doing this is that there is not out-of-the-box way to set the discoverability policy to "Public" in the configuration file. This custom **BehaviorExtension** element will allow you to achieve this functionality.
  6. At the top of the **ServiceRegistrySettingsElement.cs** file (for Visual C# projects) or the **ServiceRegistrySettingsElement.vb** file (for Visual Basic projects), add the following namespace directives.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 - Ex02 ServiceRegistrySettingsElement Namespaces -CS)
     2. C#
     3. **using System.ServiceModel.Configuration;**
     4. **using Microsoft.ServiceBus;**
     5. **using System.Configuration;**
     6. (Code Snippet - Introduction to Service Bus Lab Part 1 - Ex02 ServiceRegistrySettingsElement Namespaces - VB)
     7. Visual Basic
     8. **Imports System.ServiceModel.Configuration**
     9. **Imports Microsoft.ServiceBus**
     10. **Imports System.Configuration**
  7. Update the class' definition as shown in the following code sample. The class inherits from the BehaviorExtensionElement class, which enables the user to customize service or endpoint behaviors. The properties marked with the ConfigurationProperty attributes will be accessible from the configuration file.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 - Ex02 ServiceRegistrySettingsElement Definition - CS)
     2. C#
     3. **public class ServiceRegistrySettingsElement : BehaviorExtensionElement**
     4. **{**
     5. **private const string displayNameId = "displayName";**
     6. **private const string discoveryModeId = "discoveryMode";**
     7. **public override Type BehaviorType**
     8. **{**
     9. **get { return typeof(ServiceRegistrySettings); }**
     10. **}**
     11. **protected override object CreateBehavior()**
     12. **{**
     13. **return new ServiceRegistrySettings()**
     14. **{**
     15. **DiscoveryMode = this.DiscoveryMode,**
     16. **};**
     17. **}**
     18. **[ConfigurationProperty(discoveryModeId, DefaultValue = DiscoveryType.Private)]**
     19. **public DiscoveryType DiscoveryMode**
     20. **{**
     21. **get { return (DiscoveryType)this[discoveryModeId]; }**
     22. **set { this[discoveryModeId] = value; }**
     23. **}**
     24. **public string DisplayName**
     25. **{**
     26. **get { return (string)this[displayNameId]; }**
     27. **set { this[displayNameId] = value; }**
     28. **}**
     29. **}**
     30. (Code Snippet - Introduction to Service Bus Lab Part 1 - Ex02 ServiceRegistrySettingsElement Definition - VB)
     31. Visual Basic
     32. **Public Class ServiceRegistrySettingsElement**
     33. **Inherits BehaviorExtensionElement**
     34. **Private Const displayNameId As String = "displayName"**
     35. **Private Const discoveryModeId As String = "discoveryMode"**
     36. **Public Overrides ReadOnly Property BehaviorType() As Type**
     37. **Get**
     38. **Return GetType(ServiceRegistrySettings)**
     39. **End Get**
     40. **End Property**
     41. **Protected Overrides Function CreateBehavior() As Object**
     42. **Return New ServiceRegistrySettings() With { \_**
     43. **.DiscoveryMode = Me.DiscoveryMode, \_**
     44. **.DisplayName = Me.DisplayName \_**
     45. **}**
     46. **End Function**
     47. **<ConfigurationProperty(discoveryModeId, DefaultValue:=DiscoveryType.Public)> \_**
     48. **Public Property DiscoveryMode() As DiscoveryType**
     49. **Get**
     50. **Return DirectCast(Me(discoveryModeId), DiscoveryType)**
     51. **End Get**
     52. **Set(ByVal value As DiscoveryType)**
     53. **Me(discoveryModeId) = value**
     54. **End Set**
     55. **End Property**
     56. **<ConfigurationProperty(displayNameId)> \_**
     57. **Public Property DisplayName() As String**
     58. **Get**
     59. **Return DirectCast(Me(displayNameId), String)**
     60. **End Get**
     61. **Set(ByVal value As String)**
     62. **Me(displayNameId) = value**
     63. **End Set**
     64. **End Property**
     65. **End Class**
  8. In Visual Studio, save all files by clicking the **Save** **All** button.
  9. In **Solution** **Explorer**, double-click the **Web**.**config** file to open it in the Visual Studio editor.
  10. Replace the <system.servicemodel> element and its children with the following configuration. Make sure to replace the [YOUR-SERVICE-NAMESPACE-DOMAIN], [YOUR-ISSUER-NAME] and [YOUR-ISSUER-KEY] placeholders with the correct values. Note that the code highlighted in **bold** uses the **ServiceRegistrySettingsElement** previously created.
      1. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex02 Service Model - XML*)
      2. XML
      3. **<system.serviceModel>**
      4. **<services>**
      5. **<clear />**
      6. **<service behaviorConfiguration="MyServiceTypeBehavior" name="IISHostedCalculatorService.CalculatorService">**
      7. **<endpoint address="http://localhost/IISHostedCalculatorService/CalculatorService.svc/LocalCalculatorService"**
      8. **binding="basicHttpBinding"**
      9. **bindingConfiguration="BasicHttpConfig"**
      10. **name="Basic"**
      11. **contract="IISHostedCalculatorService.ICalculatorService" />**
      12. **<endpoint address="https://[YOUR-SERVICE-NAMESPACE-DOMAIN].servicebus.windows.net/CalculatorServiceHttp/"**
      13. **behaviorConfiguration="sharedSecretClientCredentials"**
      14. **binding="basicHttpRelayBinding"**
      15. **bindingConfiguration="HttpRelayEndpointConfig"**
      16. **name="RelayEndpoint"**
      17. **contract="IISHostedCalculatorService.ICalculatorService" />**
      18. **<endpoint address="sb://[YOUR-SERVICE-NAMESPACE-DOMAIN].servicebus.windows.net/CalculatorServiceNetTcp/"**
      19. **behaviorConfiguration="sharedSecretClientCredentials"**
      20. **binding="netTcpRelayBinding"**
      21. **bindingConfiguration="NetTcpRelayEndpointConfig"**
      22. **name="RelayEndpoint"**
      23. **contract="IISHostedCalculatorService.ICalculatorService" />**
      24. **</service>**
      25. **</services>**
      26. **<bindings>**
      27. **<basicHttpBinding>**
      28. **<binding name="BasicHttpConfig" />**
      29. **</basicHttpBinding>**
      30. **<!--service bus binding-->**
      31. **<basicHttpRelayBinding>**
      32. **<binding name="HttpRelayEndpointConfig">**
      33. **<security relayClientAuthenticationType="RelayAccessToken" />**
      34. **</binding>**
      35. **</basicHttpRelayBinding>**
      36. **<netTcpRelayBinding>**
      37. **<binding name="NetTcpRelayEndpointConfig">**
      38. **<security relayClientAuthenticationType="RelayAccessToken" />**
      39. **</binding>**
      40. **</netTcpRelayBinding>**
      41. **</bindings>**
      42. **<behaviors>**
      43. **<endpointBehaviors>**
      44. **<behavior name="sharedSecretClientCredentials">**
      45. **<transportClientEndpointBehavior credentialType="SharedSecret">**
      46. **<clientCredentials>**
      47. **<sharedSecret issuerName="[YOUR-ISSUER-NAME]" issuerSecret="[YOUR-ISSUER-KEY]" />**
      48. **</clientCredentials>**
      49. **</transportClientEndpointBehavior>**
      50. **<ServiceRegistrySettings discoveryMode="Public"/>**
      51. **</behavior>**
      52. **</endpointBehaviors>**
      53. **<serviceBehaviors>**
      54. **<behavior name="MyServiceTypeBehavior">**
      55. **<serviceMetadata httpGetEnabled="true" />**
      56. **<serviceDebug includeExceptionDetailInFaults="true" />**
      57. **</behavior>**
      58. **</serviceBehaviors>**
      59. **</behaviors>**
      60. **<extensions>**
      61. **<behaviorExtensions>**
      62. **<add name="ServiceRegistrySettings" type="IISHostedCalculatorService.ServiceRegistrySettingsElement, IISHostedCalculatorService, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />**
      63. **</behaviorExtensions>**
      64. **</extensions>**
      65. **</system.serviceModel>**
  11. Build the solution.

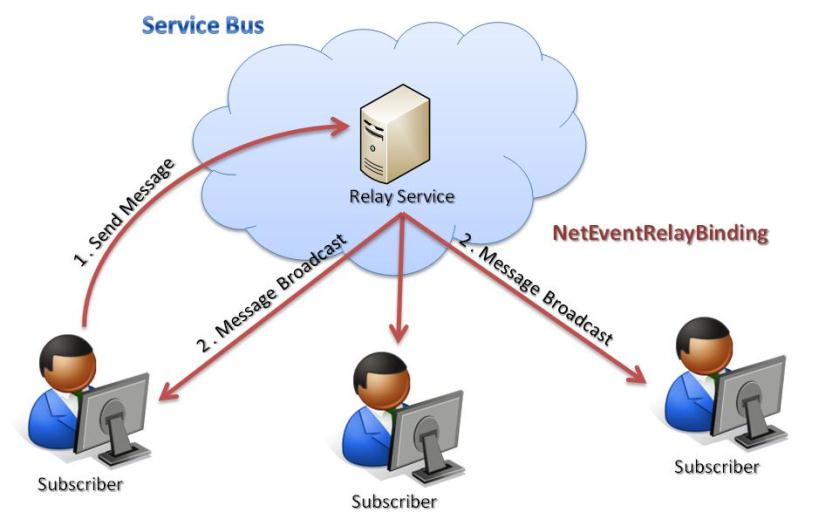
Task 3 – Enabling Windows Server App Fabric Auto-Start for the WCF Calculator Service

* 1. In this task you will enable the **Auto-Start** feature provided by **Windows Server App Fabric** for the **WCF Calculator Service** hosted by the **IISHostedCalculatorService** site. As explained previously, you activate **Auto-Start** to get the service up and running as soon as the application that it belongs to is started. This is done to make up for the following facts:
  + Until the first message is received by IIS, the service will never establish a connection to the Service Bus.
  + With no connection to the Service Bus, the service will never receive a message.
  1. Additionally, as **Windows Server AppFabric** uses the **Microsoft.WebAdministration** (MWA) API to read and write configuration files, you will provide a custom MWA schema for some of the elements from the **Web.config** file to be correctly recognized. This will give you the possibility to enable the Auto-Start feature from the Internet Information Services (IIS) Manager.
  2. Copy the **ServiceBus\_schema.xml** file located under the **Source\Assets** folder of this lab to the **%SystemRoot%\System32\inetsrv\config\schema** directory. This file contains the schema that will be used by the **Windows Server App Fabric** to read the configuration file for your service.
     1. 
     2. Figure 30
     3. ServiceBus\_schema.xml file copied to the %SystemRoot%\System32\inetsrv\config\schema directory
     4. **Note:** If you want to know what elements are supported by the **ServiceBus\_schema.xml** schema file, you can open it as you would open any other XML file.
  3. Open Internet Information Services (IIS) Manager from **Start | All Programs | Internet Information Services (IIS) 7 Manager**. The **User** **Account** **Control** dialog appears, as shown in the following figure.
     1. 
     2. Figure 31
     3. User Account Control dialog when opening the IIS Manager
  4. Click **Yes**.
  5. In the **Connections** tree view, select the **IISHostedCalculatorService** site, as shown in the following figure.
     1. 
     2. Figure 32
     3. Selecting the IISHostedCalculatorService site
  6. In the **Actions** menu, click **Configure** under the **Manage WCF and WF services** submenu. This is shown in the following figure. The **Configure WCF and WF for Application** dialog appears.
     1. 
     2. Figure 33
     3. Configuring the WCF Calculator service
  7. In the list, select **Auto-Start**, and then select the **Custom** option. Selecting the custom options allows you manually determine those services that will be auto-started. The outcome configuration is shown in the following figure.
     1. 
     2. Figure 34
     3. Selecting the Custom Auto-Start option
  8. Click **Apply**. A warning message explaining that all services in the application might recycle appears.
     1. 
     2. Figure 35
     3. Warning about Application services being recycled
  9. Click **Yes**.
  10. In the **Configure WCF and WF for Application** dialog, click **OK**.
  11. Double-click the **Services** option, which is highlighted in the following picture. The content in the center pane will be replaced with the list of services from your application.
      1. 
      2. Figure 36
      3. Services option
  12. Double-click the **IISHostedCalculatorService.CalculatorService**. The **Configuration** **Service** dialog appears.
      1. 
      2. Figure 37
      3. Configuration Service dialog
  13. In the list, select **Auto-Start**, and select the **Enable** option. This is done to enable Auto-Start for the service.
      1. 
      2. Figure 38
      3. Enabling Auto-Start for the IISHostedCalculatorService.CalculatorService
  14. Click **Apply**. In the warning message that appears which explains that all services in the application might recycle, click **Yes**.
  15. In the **Configuration** **Service** dialog, click **OK**.
  16. Restart Internet Information Services. To do this, Open a **Command Prompt** as **administrator**, and type **iisreset**.
      1. 
      2. Figure 39
      3. Restarting Internet Information Services

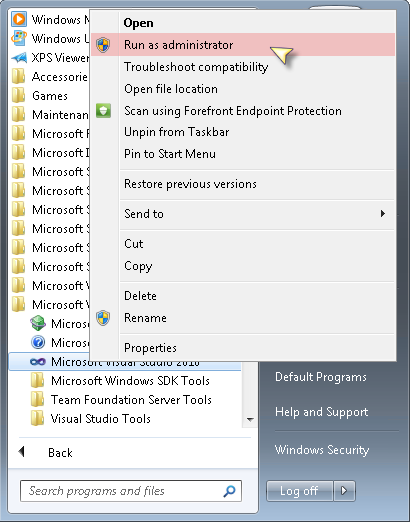
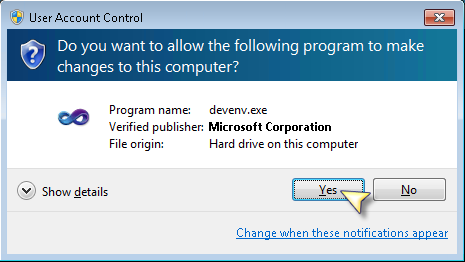
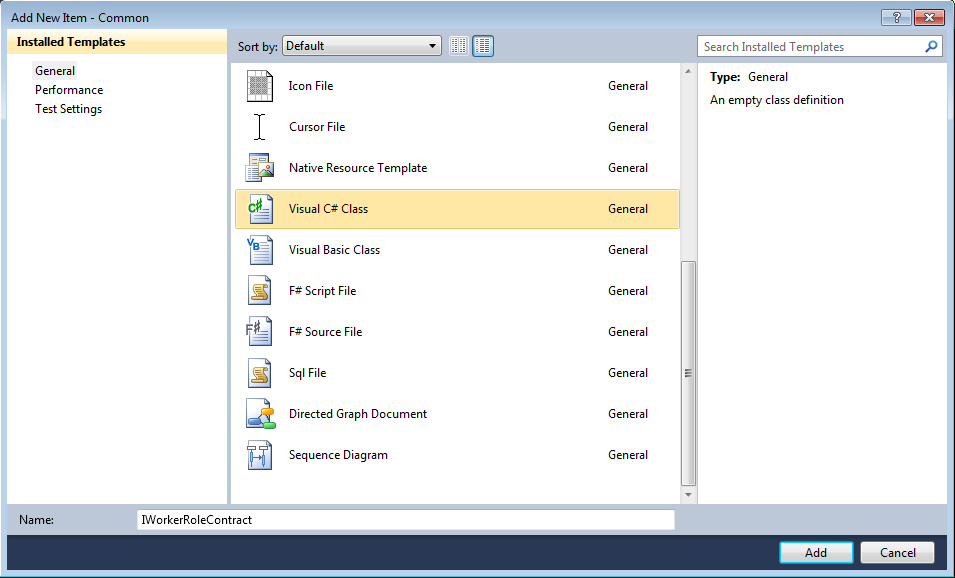
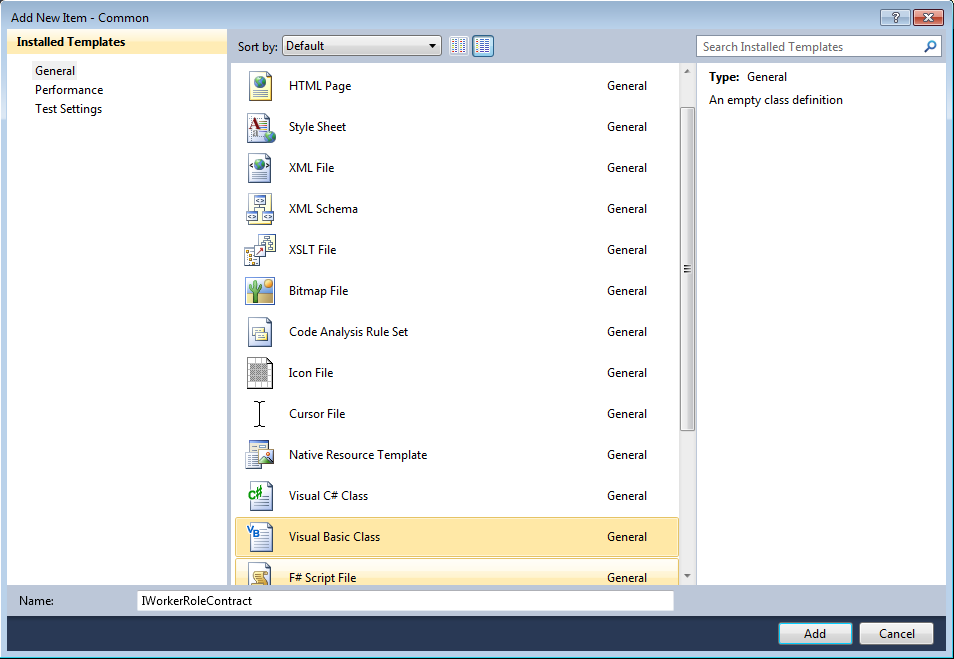
Verification

* 1. In order to verify that you have performed every step in the exercise correctly, perform the steps below.
  2. In this verification you will use an already developed WPF client that consumes the CalculatorService via the service bus. You will first configure the client with the data where the service is published in the service bus. Afterwards, you will verify that all operations work as expected.
  3. **Note:** You require an AppFabric project and a service namespace to complete this exercise. If you have not already done so, follow the Getting Started section.
  4. In Visual Studio, open the **IISHostedCalculatorService.sln** solution file in the Source folder of the lab.
  5. Right click the **CalculatorService.svc** file from the **IISHostedCalculatorService** project, and select View in Brower.
  6. In Visual Studio, open the **CalculatorClient.sln** solution file from Assets\CalculatorClient in the Source folder of the lab.
  7. Open the application configuration file to setup the service bus data. To do this, double click the **app.config** file from the Solution Explorer.
     1. 
     2. Figure 40
     3. Application Configuration
  8. In the **appSettings** section of the app.config file, locate the **serviceNamespaceDomain** setting and replace [**YOUR-SERVICE-NAMESPACE-DOMAIN**] with the domain service namespace that you registered at the portal.
  9. Locate the **issuerName** and **issuerSecret** settings and replace [**YOUR-ISSUER-NAME**] and [**YOUR-ISSUER-KEY**] with the **Default** **Issuer Name** and **Default** **Issuer Key** for the service namespace; these are the name and key generated for the service namespace at the portal Web Site that you recorded during the Getting Started section. The figure below show an example of the app.config file after replacing the setting value.
     1. 
     2. Figure 41
     3. App.config
  10. Press **F5** in Visual Studio to run the CalculatorClient application. The CalculatorClient application will open.
      1. 
      2. Figure 42
      3. Calculator Client application
  11. Type some numeric values in the parameters text boxes, and click on the **Add ( + )** operation button. This button will execute the Add operation of the CalculatorService via the Service Bus. You will see the service URI below the buttons. Notice that https protocol is used by default.
      1. 
      2. Figure 43
      3. Service Executing
  12. When the service finishes executing you will see the result at the bottom of the application. You can verify that all service operations are working by clicking on the other operation buttons (**Substract ( - ), Multiply ( \* )** and **Divide ( / )**).
      1. 
      2. Figure 44
      3. Service Result
  13. On the **Connectivity Mode** section, check the **NetTCPRelay** option and click any of the operation buttons. This will cause the Calculator Client to use the TCP protocol to communicate with the service bus. You will notice that the service URI will change.
      1. 
      2. Figure 45
      3. TCP Protocol

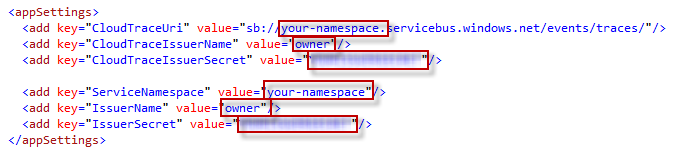
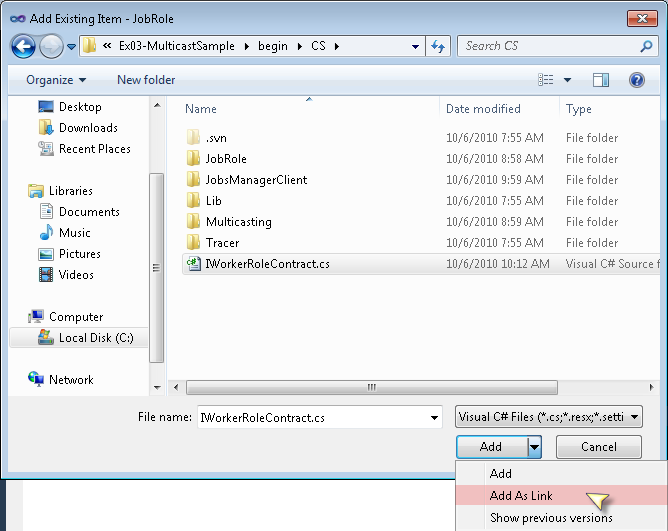
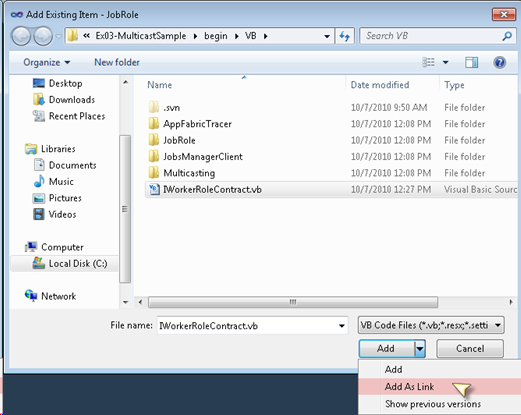
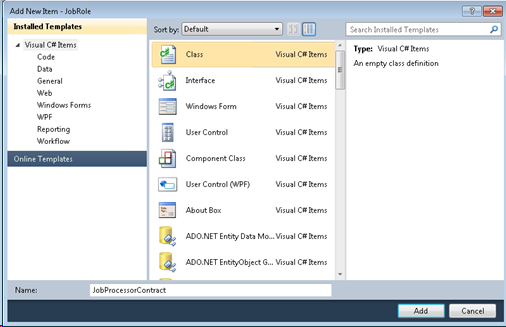
Exercise 3: Multicast Messaging through the Service Bus

* 1. In this exercise, you build a worker role management application that demonstrates multicast messaging through theService Bus. Multicast, otherwise known as simple publish/subscribe, enables multiple listeners and senders at a single URI. Every actor is both a listener and a sender at the same time.
  2. To implement a multicast messaging pattern, the Service Bus provides a binding named **netEventRelayBinding** that you can use for this purpose. This binding enables a publish/subscribe communication pattern over WCF, which is not available from any of the built-in WCF bindings. The **netEventRelayBinding** binding allows multiple applications to subscribe to messages sent to an endpoint. Any messages sent to that endpoint are received by all applications.
  3. TheService Bus implements the **netEventRelayBinding** by allowing multiple listeners to register themselves at a URI specified by the first sender. Each service host that has an endpoint listening on that same URI implicitly becomes a subscriber. When a client sends a message to that URI, the message is delivered to all service endpoints currently subscribed. In this scenario, the back-end system plays the role of the WCF client and broadcasts (publishes) a message to all subscribers.
  4. 
  5. Figure 46
  6. Multicasting using the netEventRelayBinding
  7. In this exercise, you will create a WPF client application which will be used to send messages to a set of worker roles. To do this, you will need to establish a WCF contract that will define the set of messages the worker roles will process and that the WPF client will send to the roles. In this process, you will also establish a connection for each of the components (the worker roles and the WPF client) to the service bus, and configure them to listen to a single URI.
  8. This exercise will expose several Service Bus features such as:
  + Service Bus **netEventRelayBinding**
  1. **Note:** This exercise could be executed with a single solution. However, it is interesting to run it using two different machines and two different AppFabric accounts. Also, enabling tracing and message logging allows you to understand how the relay works.

Task 1 – Defining the Service Contract Operations

* 1. In this task, you define the contract operations that will be shared among the worker roles and the worker role manager client.
  2. Open Microsoft Visual Studio 2010 with elevated permissions. To do this, go to **Start | All Programs | Microsoft Visual Studio 2010,** right-click the **Microsoft Visual Studio 2010** option and select **Run as Administrator**. This is shown in the following figure.
     1. 
     2. Figure 47
     3. Running Visual Studio as administrator from the Start menu
  3. Click **Yes** in the **User** **Account** **Control** dialog. Visual Studio will open with Administrator privileges.
     1. 
     2. Figure 48
     3. Clicking Yes in the User Account Control dialog opens VS with Administrator privileges
  4. In Visual Studio, open the **MulticastSample**.**sln** solution file from Ex03-MulticastSample\begin\{CS|VB} in the Source folder of the lab.
  5. Create the interface that specifies the contract to interact with worker roles. To do this, in Solution Explorer, right-click the **Common** folder, point to **Add**, and select **New** **Item**. The Add New Item dialog appears.
  6. Select the **Visual C# Class** template if you are working withC#or the **Visual Basic Class** template if you are working in Visual Basic.
     1. 
     2. Figure 49
     3. Creating the IWorkerRoleContract interface (C#)
     4. 
     5. Figure 50
     6. Creating the IWorkerRoleContract interface (Visual Basic)
  7. Change the **Name** to **IWorkerRoleContract** and click **Add**.
  8. Remove the class's constructor (for Visual C# projects), update the type's definition to interface, and update the interface's name to **IWorkerRoleContract**. The outcome is shown in the following code.
     1. C#
     2. public interface IWorkerRoleContract
     3. {
     5. }
     6. Visual Basic
     7. Public Interface IWorkerRoleContract
     8. End Interface
  9. At the top of the file for the IWorkerRoleContract interface, add the following namespace directive.
     1. C#
     2. using System.ServiceModel;
     3. Visual Basic
     4. Imports System.ServiceModel
  10. (For C# projects) Wrap the interface and namespace directives with the **JobSample**.**Common** namespace declaration, as shown in the following code.
      1. C#
      2. **namespace JobSample.Common**
      3. {
      4. using System;
      5. using System.ServiceModel;
      6. public interface IWorkerRoleContract
      7. {
      8. }
      9. }
  11. Define the service contract operations, by adding the following methods to the interface's body. These operations will be used to signal worker roles that they should start/stop processing jobs.
      1. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 WorkerRoleContract Operations - CS*)
      2. C#
      3. public interface IWorkerRoleContract
      4. {
      5. **[OperationContract(IsOneWay = true)]**
      6. **void StartProcessingJobs();**
      7. **[OperationContract(IsOneWay = true)]**
      8. **void StopProcessingJobs();**
      9. }
      10. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 WorkerRoleContract Operations - VB*)
      11. Visual Basic
      12. Public Interface IWorkerRoleContract
      13. **<OperationContract(IsOneWay:=True)> \_**
      14. **Sub StartProcessingJobs()**
      15. **<OperationContract(IsOneWay:=True)> \_**
      16. **Sub StopProcessingJobs()**
      17. End Interface
      18. **Note:** The **netEventRelayBinding**, which you will use for client and service endpoints, requires contracts to expose one way operations only. Therefore, all operations must be marked with *“IsOneWay=true”* to indicate that the operation only consists of a single input message and has no associated output message.
  12. Apply the **ServiceContract** attribute to the **IWorkerRoleContract** interface to indicate that the interface defines a service contract in the application. To do this, add the highlighted code on top of the interface's definition.
      1. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 WorkerRoleContract ServiceContract Attribute - CS*)
      2. C#
      3. **[ServiceContract(Name = "IWorkerRoleContract", Namespace = "http://samples.microsoft.com/ServiceModel/Relay/")]**
      4. public interface IWorkerRoleContract
      5. {
      6. ...
      7. }
      8. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 WorkerRoleContract ServiceContract Attribute - VB*)
      9. Visual Basic
      10. **<ServiceContract(Name:="IWorkerRoleContract", [Namespace]:="http://samples.microsoft.com/ServiceModel/Relay/")> \_**
      11. Public Interface IWorkerRoleContract
      12. ...
      13. End Interface
  13. At the bottom of the **JobSample**.**Common** namespace scope, add an interface definition as shown in the following code fragment. This interface will be used to connect to the multicast relay.
      1. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 IWorkerRoleChannel Interface - CS*)
      2. C#
      3. **public interface IWorkerRoleChannel : IWorkerRoleContract, IClientChannel { }**
      4. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 IWorkerRoleChannel Interface - VB*)
      5. Visual Basic
      6. **Public Interface IWorkerRoleChannel**
      7. **Inherits IWorkerRoleContract**
      8. **Inherits IClientChannel**
      9. **End Interface**
  14. Having performed all steps, the code in the **IWorkerRoleContract.cs** file should be the following one.
  15. C#
  16. namespace JobSample.Common
  17. {
  18. using System;
  19. using System.ServiceModel;
  20. [ServiceContract(Name = "IWorkerRoleContract", Namespace = "http://samples.microsoft.com/ServiceModel/Relay/")]
  21. public interface IWorkerRoleContract
  22. {
  23. [OperationContract(IsOneWay = true)]
  24. void StartProcessingJobs();
  25. [OperationContract(IsOneWay = true)]
  26. void StopProcessingJobs();
  27. }
  28. public interface IWorkerRoleChannel : IWorkerRoleContract, IClientChannel { }
  29. }
  30. Visual Basic
  31. Imports Microsoft.VisualBasic
  32. Imports System.ServiceModel
  33. <ServiceContract(Name:="IWorkerRoleContract", [Namespace]:="http://samples.microsoft.com/ServiceModel/Relay/")> \_
  34. Public Interface IWorkerRoleContract
  35. <OperationContract(IsOneWay:=True)> \_
  36. Sub StartProcessingJobs()
  37. <OperationContract(IsOneWay:=True)> \_
  38. Sub StopProcessingJobs()
  39. End Interface
  40. Public Interface IWorkerRoleChannel
  41. Inherits IWorkerRoleContract
  42. Inherits IClientChannel
  43. End Interface

Task 2 – Subscribing The Worker Roles To Receive Messages

* 1. In this task you will provide the necessary configuration and provide an implementation for the worker role contract. This will enable worker roles to be subscribed to messages sent through the multicast relay and perform actions when receiving those messages.
  2. In the Solution Explorer, double-click the **App.config** file under the **Common** folder.
  3. Replace the **[YOUR-SERVICE-NAMESPACE-DOMAIN]**, **[YOUR-ISSUER-NAME]** and [**YOUR-ISSUER-KEY]** placeholders with the correct values. These are the Service Namespace, Default Issuer Name and Default Issuer Key respectively. The figure below shows an example of the app.config file after replacing the values.
     1. 
     2. Figure 51
     3. Replacing the Azure account values in the configuration file
  4. Create a new binding configuration for the **netEventRelayBinding**. To do this, place the following code (shown in **bold**) inside the <bindings> element.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 Worker Roles NetEventRelayBinding - XML*)
     2. XML
     3. <bindings>
     4. **<netEventRelayBinding>**
     5. **<binding name="default" />**
     6. **</netEventRelayBinding>**
     7. </bindings>
  5. Provide the configuration for the client endpoint. To do this, add the highlighted code in the following sample below the </bindings> closing tag. Note that this endpoint will use the **netEventRelayBinding**, and the **IWorkerRoleContract** interface you created in the first task of this exercise.
     1. For Visual C# use the following fragment.
     2. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 Worker Roles Client Endpoint C# - XML*)
     3. XML
     4. ...
     5. </bindings>
     6. **<client>**
     7. **<endpoint name="RelayEndpoint"**
     8. **contract="JobSample.Common.IWorkerRoleContract"**
     9. **binding="netEventRelayBinding"**
     10. **bindingConfiguration="default"**
     11. **address="http://AddressToBeReplacedInCode/" />**
     12. **</client>**
     13. For Visual Basic use the following fragment.
     14. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 Worker Roles Client Endpoint VB - XML*)
     15. XML
     16. ...
     17. </bindings>
     18. **<client>**
     19. **<endpoint name="RelayEndpoint"**
     20. **contract="JobRole.IWorkerRoleContract"**
     21. **binding="netEventRelayBinding"**
     22. **bindingConfiguration="default"**
     23. **address="http://AddressToBeReplacedInCode/" />**
     24. **</client>**
  6. Add **IWorkerRoleContract** interface as a link in the **JobRole** project. To do this:
     1. Right-click the **JobRole** project, point to **Add** and click **Existing** **Item**. The **Add** **Existing** **Item** dialog appears.
     2. Browse to the Source\Ex03-MulticastSample\begin\{CS|VB} folder of this Lab and select the **IWorkerRoleContract**.**cs** file (for Visual C# projects) or the **IWorkerRoleContract**.**vb** file (for Visual Basic projects).
     3. Click the arrow next to the **Add** button and select **Add as Link** from the drop down list. This is shown in the following figure.
        + 1. 
          2. Figure 52
          3. Adding the IWorkerRoleContract.cs interface file as a link (C#)
          4. 
          5. Figure 53
          6. Adding the IWorkerRoleContract.cs interface file as a link (Visual Basic)
  7. Right-click the **JobRole** project, point to **Add** and select **Class**. The **Add** **New** **Item** dialog appears with the **Class** item template selected by default.
  8. Name the class **JobProcessorContract**, and click **OK**. This class will be used to provide an implementation for the **IWorkerRoleContract** interface. The following figure shows the dialog before confirming the new item's creation.
     1. 
     2. Figure 54
     3. Adding the JobProcessorContract class (C#)
  9. At the top of the file for the JobProcessorContract class, add the following namespace directives.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 - Ex03 JobProcessorContract Namespace Directives - CS)
     2. C#
     3. **using JobSample.Common;**
     4. **using System.ServiceModel;**
     5. (Code Snippet - Introduction to Service Bus Lab Part 1 - Ex03 *JobProcessorContract Namespace Directives - VB*)
     6. Visual Basic
     7. **Imports System.ServiceModel**
  10. Implement the **IWorkerRoleContract** interface and add the public access modifier to the **JobProcessorContract** class' declaration, as shown in the following code.
      1. C#
      2. **public** class JobProcessorContract : **IWorkerRoleContract**
      3. {

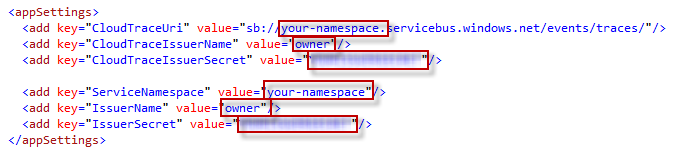
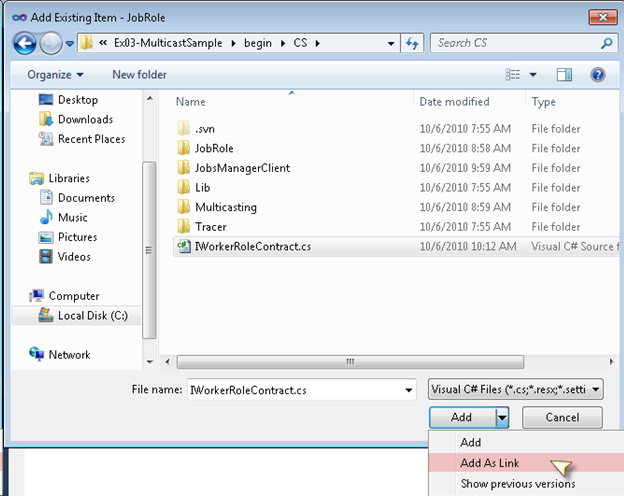
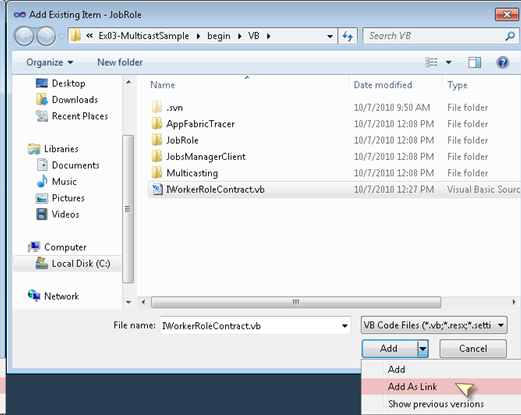
}

* + 1. Visual Basic
    2. Public Class JobProcessorContract
    3. **Implements IWorkerRoleContract**
    4. End Class
  1. Provide the implementations highlighted in the following code fragment for the methods declared in the **IWorkerRoleContract** interface.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 JobProcessorContract Implementation - CS*)
     2. C#
     3. public class JobProcessorContract : IWorkerRoleContract
     4. {
     5. **public void StartProcessingJobs()**
     6. **{**
     7. **JobProccesor.Instance.StartProcessingJobs();**
     8. **}**
     9. **public void StopProcessingJobs()**
     10. **{**
     11. **JobProccesor.Instance.StopProcessingJobs();**
     12. **}**
     13. }
     14. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 JobProcessorContract Implementation - VB*)
     15. Visual Basic
     16. Public Class JobProcessorContract
     17. Implements IWorkerRoleContract
     18. **Public Sub StartProcessingJobs() Implements IWorkerRoleContract.StartProcessingJobs**
     19. **JobProccesor.Instance.StartProcessingJobs()**
     20. **End Sub**
     21. **Public Sub StopProcessingJobs() Implements IWorkerRoleContract.StopProcessingJobs**
     22. **JobProccesor.Instance.StopProcessingJobs()**
     23. **End Sub**
     24. End Class
  2. Add a reference to the Service Bus core assembly. To do this, right-click the **JobRole** project in Solution Explorer, select **Add** **Reference** to open the **Add** **Reference** dialog, select the .**NET** tab, choose **Microsoft**.**ServiceBus**.**dll** for .NET Framework 4.0 and click **OK**.
     1. **Note:** If you cannot find the **Microsoft**.**ServiceBus** assembly in the .**NET** tab, use the **Browse** tab to locate this assembly inside the **%ProgramFiles%\Windows Azure AppFabric SDK\V1.0\Assemblies\NET4.0** folder.
  3. In Solution Explorer, double-click the **WorkerRole**.**cs** file (for Visual C# projects) or the **WorkerRole**.**vb** file (for Visual Basic projects).
  4. At the top of the file for the WorkerRole class, add the following namespace directives.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 WorkerRole Namespace Directives - CS*)
     2. C#
     3. **using System;**
     4. **using JobSample.Common;**
     5. **using Microsoft.ServiceBus;**
     6. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 WorkerRole Namespace Directives - VB*)
     7. Visual Basic
     8. **Imports System**
     9. **Imports Microsoft.ServiceBus**
  5. Add the following private fields to the **WorkerRole** class.
     1. C#
     2. public class WorkerRole : RoleEntryPoint
     3. {
     4. **private ServiceHost host;**
     5. **private IWorkerRoleChannel channel;**
     6. public override void Run()
     7. {
     8. Trace.WriteLine(string.Format("{0}: Started", RoleEnvironment.CurrentRoleInstance.Id), "Information");
     9. this.ConnectToMulticastRelay();
     10. // start listening to job requests
     11. JobProccesor.Instance.Start();
     12. }
     13. ...
     14. }
     15. For Visual Basic use the following fragment.
     16. Visual Basic
     17. Public Class WorkerRole
     18. Inherits RoleEntryPoint
     19. **Private host As ServiceHost**
     20. **Private channel As IWorkerRoleChannel**
     21. Public Overrides Sub Run()
     22. Trace.WriteLine(String.Format("{0}: Started", RoleEnvironment.CurrentRoleInstance.Id), "Information")
     23. Me.ConnectToMulticastRelay()
     24. ' start listening to job requests
     25. JobProccesor.Instance.Start()
     26. End Sub
     27. ...
     28. End Class
  6. Add the highlighted code from the following fragment inside the body of the **ConnectToMulticastRelay** method.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 ConnectToMulticastRelay Implementation - CS*)
     2. C#
     3. private void ConnectToMulticastRelay()
     4. {
     5. // Get Connection info from config file
     6. string serviceNamespace = ConfigurationManager.AppSettings["ServiceNamespace"];
     7. string issuerName = ConfigurationManager.AppSettings["IssuerName"];
     8. string issuerSecret = ConfigurationManager.AppSettings["IssuerSecret"];
     9. **//establish connection**
     10. **// create the credentials object for the endpoint**
     11. **TransportClientEndpointBehavior relayCredentials = new TransportClientEndpointBehavior();**
     12. **relayCredentials.CredentialType = TransportClientCredentialType.SharedSecret;**
     13. **relayCredentials.Credentials.SharedSecret.IssuerName = issuerName;**
     14. **relayCredentials.Credentials.SharedSecret.IssuerSecret = issuerSecret;**
     15. **// create the service URI based on the service namespace**
     16. **Uri serviceAddress = ServiceBusEnvironment.CreateServiceUri("sb", serviceNamespace, "/JobProcessor/");**
     17. **// create service host**
     18. **this.host = new ServiceHost(typeof(JobProcessorContract), serviceAddress);**
     19. **this.host.Description.Endpoints[0].Behaviors.Add(relayCredentials);**
     20. **this.host.Open();**
     21. **// create the channel factory loading the configuration**
     22. **ChannelFactory<IWorkerRoleChannel> channelFactory = new ChannelFactory<IWorkerRoleChannel>("RelayEndpoint", new EndpointAddress(serviceAddress));**
     23. **// apply the Service Bus credentials**
     24. **channelFactory.Endpoint.Behaviors.Add(relayCredentials);**
     25. **// create and open the client channel**
     26. **this.channel = channelFactory.CreateChannel();**
     27. **this.channel.Open();**
     28. Trace.WriteLine(string.Format("{0}: Connected",RoleEnvironment.CurrentRoleInstance.Id), "Information");
     29. }
     30. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 ConnectToMulticastRelay Implementation - VB*)
     31. Visual Basic
     32. Private Sub ConnectToMulticastRelay()
     33. ' Get connection info from the config file
     34. Dim serviceNamespace As String = ConfigurationManager.AppSettings("ServiceNamespace")
     35. Dim issuerName As String = ConfigurationManager.AppSettings("IssuerName")
     36. Dim issuerSecret As String = ConfigurationManager.AppSettings("IssuerSecret")
     37. **'establish connection**
     38. **' create the credentials object for the endpoint**
     39. **Dim relayCredentials As New TransportClientEndpointBehavior()**
     40. **relayCredentials.CredentialType = TransportClientCredentialType.SharedSecret**
     41. **relayCredentials.Credentials.SharedSecret.IssuerName = issuerName**
     42. **relayCredentials.Credentials.SharedSecret.IssuerSecret = issuerSecret**
     43. **' create the service URI based on the service namespace**
     44. **Dim serviceAddress As Uri = ServiceBusEnvironment.CreateServiceUri("sb", serviceNamespace, "/JobProcessor/")**
     45. **' create service host**
     46. **Me.host = New ServiceHost(GetType(JobProcessorContract), serviceAddress)**
     47. **Me.host.Description.Endpoints(0).Behaviors.Add(relayCredentials)**
     48. **Me.host.Open()**
     49. **' create the channel factory loading the configuration**
     50. **Dim channelFactory As New ChannelFactory(Of IWorkerRoleChannel)("RelayEndpoint", New EndpointAddress(serviceAddress))**
     51. **' apply the Service Bus credentials**
     52. **channelFactory.Endpoint.Behaviors.Add(relayCredentials)**
     53. **' create and open the client channel**
     54. **Me.channel = channelFactory.CreateChannel()**
     55. **Me.channel.Open()**
     56. Trace.WriteLine(String.Format("{0}: Connected", RoleEnvironment.CurrentRoleInstance.Id), "Information")
     57. End Sub
  7. In the Solution Explorer, double-click the **App**.**config** file under the **Common** folder.
  8. Provide the configuration for the service and its endpoint. To do this, add the highlighted code in the following sample below the </client> closing tag. Note that this endpoint will use the **netEventRelayBinding**, and the implementation you created for the **IWorkerRoleContract** (JobProcessorContract class)you created in this task.
     1. For Visual C# use the following fragment.
     2. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 Worker Roles Service Endpoint C# - XML*)
     3. XML
     4. ...
     5. </client>
     6. **<services>**
     7. **<service name="JobRole.JobProcessorContract">**
     8. **<endpoint name="RelayEndpoint"**
     9. **contract="JobSample.Common.IWorkerRoleContract"**
     10. **binding="netEventRelayBinding"**
     11. **bindingConfiguration="default"**
     12. **address="" />**
     13. **</service>**
     14. **</services>**
     15. </system.serviceModel>

...

* + 1. For Visual Basic use the following fragment.
    2. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 Worker Roles Service Endpoint VB - XML*)
    3. XML
    4. ...
    5. </client>
    6. **<services>**
    7. **<service name="JobRole.JobProcessorContract">**
    8. **<endpoint name="RelayEndpoint"**
    9. **contract="JobRole.IWorkerRoleContract"**
    10. **binding="netEventRelayBinding"**
    11. **bindingConfiguration="default"**
    12. **address="" />**
    13. **</service>**
    14. **</services>**
    15. </system.serviceModel>
  1. Note that the configuration file has an entry to add a trace listener. This trace listener is used to trace messages from the worker roles.  
     1. XML
     2. <system.diagnostics>
     3. <trace>
     4. <listeners>
     5. <add name="AppFabricTracer" type="Tools.AppFabricTracer.CloudTraceListener, Tools.AppFabricTracer" />
     6. </listeners>
     7. </trace>
     8. </system.diagnostics>
     9. For Visual Basic, the segment included is the following.
     10. XML
     11. <system.diagnostics>
     12. <trace>
     13. <listeners>
     14. <add name="AppFabricTracer" type=" AppFabricTracer.Tools.AppFabricTracer.CloudTraceListener, AppFabricTracer" />
     15. </listeners>
     16. </trace>
     17. </system.diagnostics>

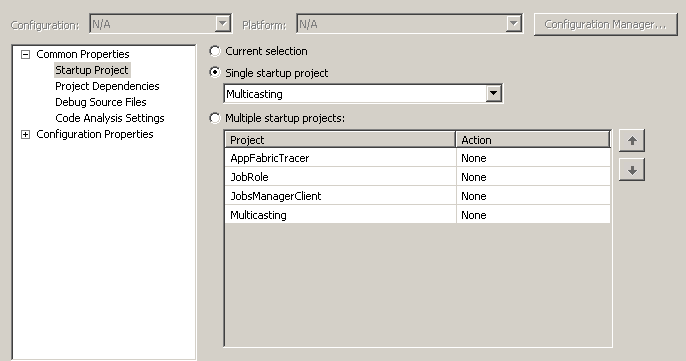
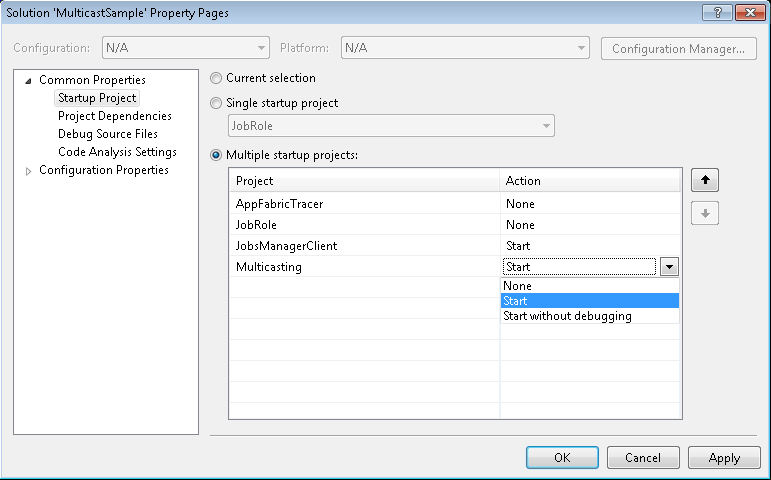
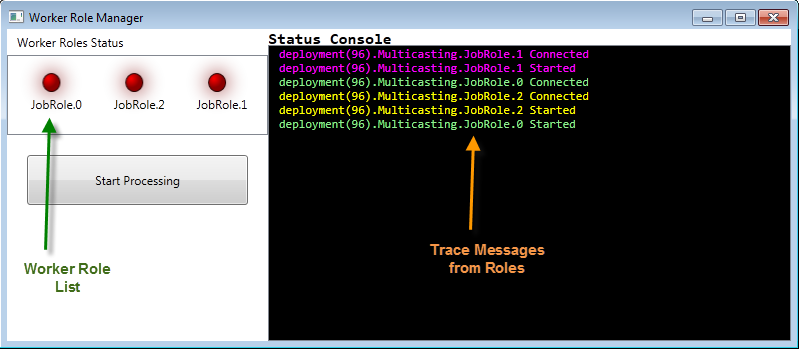
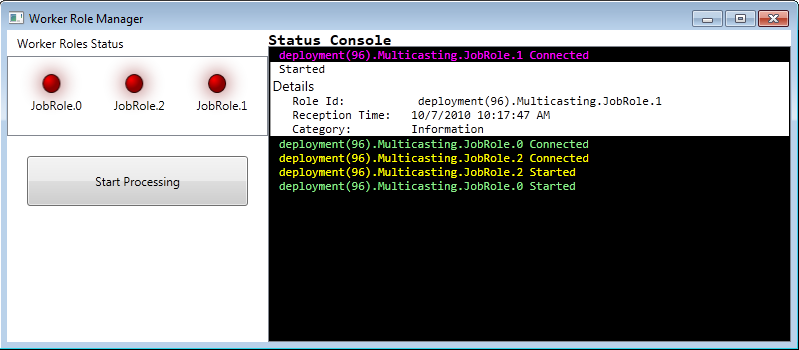
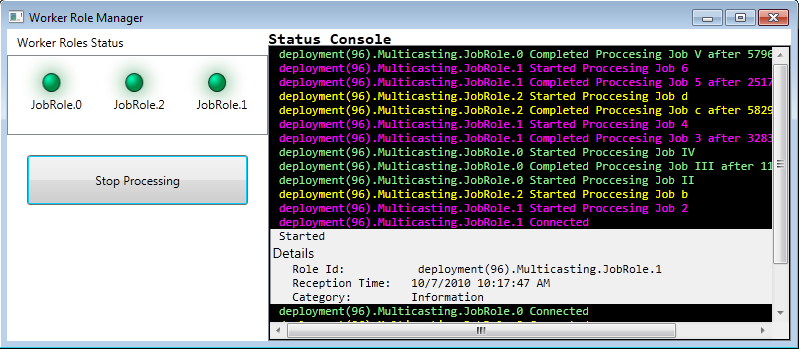
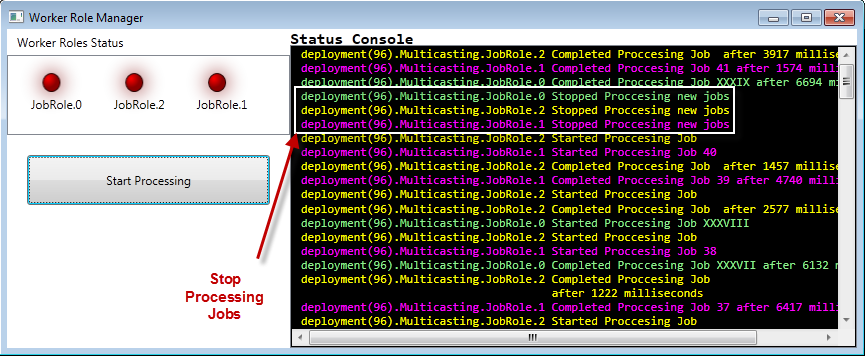
Task 3 – Setting up the Management Console to Send Messages

* 1. In this task you will provide the necessary configuration and provide an implementation for the worker role contract. Additionally, you will customize the manager console's logic to start/stop processing jobs when a command is executed.
  2. In the Solution Explorer, double-click the **App**.**config** file under the **JobsManagerClient** project.
  3. Replace the **[YOUR-SERVICE-NAMESPACE-DOMAIN]**, **[YOUR-ISSUER-NAME]** and **[YOUR-ISSUER-KEY]** placeholders with the correct values. These are the Service Namespace, Default Issuer Name and Default Issuer Key respectively. The figure below shows an example of the app.config file after replacing the values.
     1. 
     2. Figure 55
     3. Replacing the Azure account values in the configuration file
  4. Create a new binding configuration for the **netEventRelayBinding**. To do this, place the following code (shown in **bold**) inside the <bindings> element.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 WPF Client NetEventRelayBinding - XML*)
     2. XML
     3. <bindings>
     4. **<netEventRelayBinding>**
     5. **<binding name="default"/>**
     6. **</netEventRelayBinding>**
     7. </bindings>
  5. Provide the configuration for the client endpoint. To do this, add the highlighted code in the following sample below the </bindings> closing tag. Note that this endpoint will use the **netEventRelayBinding**, and the **IWorkerRoleContract** interface you created in the first task of this exercise.
     1. For Visual C# use the following fragment.
     2. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 WPF Client Client Endpoint C# - XML*)
     3. XML
     4. ...
     5. </bindings>
     6. **<client>**
     7. **<endpoint name="RelayEndpoint" contract="JobSample.Common.IWorkerRoleContract" binding="netEventRelayBinding" bindingConfiguration="default" address="http://AddressToBeReplacedInCode/"/>**
     8. **</client>**
     9. ...
     10. For Visual Basic use the following fragment.
     11. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 WPF Client Client Endpoint VB - XML*)
     12. XML
     13. ...
     14. </bindings>
     15. **<client>**
     16. **<endpoint name="RelayEndpoint" contract="JobsManagerClient.IWorkerRoleContract" binding="netEventRelayBinding" bindingConfiguration="default" address="http://AddressToBeReplacedInCode/"/>**
     17. **</client>**
     18. ...
  6. Add the **IWorkerRoleContract** interface as a link in the **JobsManagerClient** project. To do this:
     1. Right-click the **JobsManagerClient** project, point to **Add** and click **Existing** **Item**. The **Add** **Existing** **Item** dialog appears.
     2. Browse to the Source\Ex03-MulticastMessaging\begin\{CS|VB} folder of this Lab and select the **IWorkerRoleContract**.**cs** file (for Visual C# projects) or the **IWorkerRoleContract**.**vb** file (for Visual Basic projects).
     3. Click the arrow next to the **Add** button and select **Add as Link** from the drop down list. This is shown in the following figure.
        + 1. 
          2. Figure 56
          3. Adding the IWorkerRoleContract.cs interface file as a link (C#)
          4. 
          5. Figure 57
          6. Adding the IWorkerRoleContract.cs interface file as a link (VB)
  7. Right-click the **JobsManagerClient** project, point to **Add** and select **Class**. The **Add** **New** **Item** dialog appears with the Class item template selected by default.
  8. Name the class **JobManager**, and click **OK**. This class will be used to provide an implementation for the **IWorkerRoleContract** interface.
  9. At the top of the file for the **JobManager** class, add the following namespace directives.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 JobManager Namespace Directives - CS*)
     2. C#
     3. **using System.ServiceModel;**
     4. **using JobSample.Common;**
     5. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 JobManager Namespace Directives - VB*)
     6. Visual Basic
     7. **Imports System.ServiceModel**
  10. Update the class' definition as shown in the following code fragment.
      1. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 JobManager - CS*)
      2. C#
      3. **[ServiceBehavior(Name = "JobManager", Namespace = "http://samples.microsoft.com/ServiceModel/Relay/")]**
      4. **public class JobManager : IWorkerRoleContract**
      5. **{**
      6. **public void StartProcessingJobs()**
      7. **{**
      8. **// WPF should do nothing when receiving messages from the SB**
      9. **}**
      10. **public void StopProcessingJobs()**
      11. **{**
      12. **// WPF should do nothing when receiving messages from the SB**
      13. **}**
      14. **}**
      15. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 JobManager - VB*)
      16. Visual Basic
      17. **<ServiceBehavior(Name:="JobManager", Namespace:="http://samples.microsoft.com/ServiceModel/Relay/")> \_**
      18. **Public Class JobManager**
      19. **Implements IWorkerRoleContract**
      20. **Public Sub StartProcessingJobs() Implements IWorkerRoleContract.StartProcessingJobs**
      21. **' WPF should do nothing when receiving messages from the SB**
      22. **End Sub**
      23. **Public Sub StopProcessingJobs() Implements IWorkerRoleContract.StopProcessingJobs**
      24. **' WPF should do nothing when receiving messages from the SB**
      25. **End Sub**
      26. **End Class**
      27. **Note:** Both methods provided by the contract are empty because the WPF client does not need to do anything when it receives the multicast message.
  11. In the Solution Explorer, double-click the **App**.**config** file under the **JobsManagerClient** project.
  12. Provide the configuration for the service and its endpoint. To do this, add the highlighted code in the following sample below the </client> closing tag. Note that this endpoint will use the **netEventRelayBinding**, and the implementation you created for the **IWorkerRoleContract** (JobManager class) you created in this task.
      1. For Visual C# use the following fragment.
      2. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 WPF Client Service Endpoint C# - XML*)
      3. XML
      4. ...
      5. </client>
      6. **<services>**
      7. **<service name="JobsManagerClient.JobManager">**
      8. **<endpoint name="RelayEndpoint" contract="JobSample.Common.IWorkerRoleContract" binding="netEventRelayBinding" bindingConfiguration="default" address=""/>**
      9. **</service>**
      10. **</services>**
      11. </system.serviceModel>
      12. For Visual Basic use the following fragment.
      13. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 WPF Client Service Endpoint VB - XML*)
      14. XML
      15. ...
      16. </client>
      17. **<services>**
      18. **<service name="JobsManagerClient.JobManager">**
      19. **<endpoint name="RelayEndpoint" contract="JobsManagerClient.IWorkerRoleContract" binding="netEventRelayBinding" bindingConfiguration="default" address=""/>**
      20. **</service>**
      21. **</services>**
      22. </system.serviceModel>
  13. In the Solution Explorer, double-click the **ManagerWindowViewModel**.**cs** file (for Visual C# projects) or the **ManagerWindowViewModel**.**vb** file (for Visual Basic projects) under the **ViewModels** folder of the **JobsManagerClient** project.
  14. Add a reference to the Service Buscore assembly. To do this, right-click the **JobsManagerClient** project in **Solution Explorer**, select **Add Reference** to open the **Add Reference** dialog, select the **.NET** tab, choose **Microsoft.ServiceBus.dll** for .NET Framework 4.0 and click **OK**.
      1. **Note:** If you cannot find the **Microsoft.ServiceBus** assembly in the **.NET** tab, use the **Browse** tab to locate this assembly inside the **%ProgramFiles%\Windows Azure AppFabric SDK\V1.0\Assemblies\NET4.0** folder.
  15. At the top of the file, add the following namespace directives.
      1. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 ManagerWindowViewModel Namespace Directives - CS*)
      2. C#
      3. **using System.ServiceModel;**
      4. **using JobSample.Common;**
      5. **using Microsoft.ServiceBus;**
      6. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 ManagerWindowViewModel Namespace Directives - VB*)
      7. Visual Basic
      8. **Imports System.ServiceModel**
      9. **Imports Microsoft.ServiceBus**
  16. Declare the highlighted instance attributes from the following code fragment, at the top of the class' body.
      1. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 ManagerWindowViewModel Instance Attributes - CS*)
      2. C#
      3. public class ManagerWindowViewModel : IDisposable, INotifyPropertyChanged
      4. {
      5. **private ServiceHost host;**
      6. **private ChannelFactory<IWorkerRoleChannel> channelFactory;**
      7. **private IWorkerRoleChannel channel;**
      8. private ObservableWorkerRoleMessageListener traceListener;
      9. private bool isRunning;
      10. ...
      11. }
      12. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 ManagerWindowViewModel Instance Attributes - VB*)
      13. Visual Basic
      14. Public Class ManagerWindowViewModel
      15. Implements IDisposable
      16. Implements INotifyPropertyChanged
      17. **Private host As ServiceHost**
      18. **Private channelFactory As ChannelFactory(Of IWorkerRoleChannel)**
      19. **Private channel As IWorkerRoleChannel**
      20. ...
      21. End Class
  17. Add code to the **ConnectToMulticastRelay** method to be able to establish the connection to the multicast relay. The following code fragment shows the necessary code in **bold**.
      1. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 ManagerWindowViewModel ConnectToMulticastRelay - CS*)
      2. C#
      3. private void ConnectToMulticastRelay()
      4. {
      5. // Get connection info from the config file
      6. string serviceNamespace = ConfigurationManager.AppSettings["ServiceNamespace"];
      7. string issuerName = ConfigurationManager.AppSettings["IssuerName"];
      8. string issuerSecret = ConfigurationManager.AppSettings["IssuerSecret"];
      9. // Establish connection
      10. **// create the credentials object for the endpoint**
      11. **TransportClientEndpointBehavior relayCredentials = new TransportClientEndpointBehavior();**
      12. **relayCredentials.CredentialType = TransportClientCredentialType.SharedSecret;**
      13. **relayCredentials.Credentials.SharedSecret.IssuerName = issuerName;**
      14. **relayCredentials.Credentials.SharedSecret.IssuerSecret = issuerSecret;**
      16. **// create the service URI based on the service namespace**
      17. **Uri serviceAddress = ServiceBusEnvironment.CreateServiceUri("sb", serviceNamespace, "/JobProcessor/");**
      18. **// create service host**
      19. **host = new ServiceHost(typeof(JobManager), serviceAddress);**
      20. **host.Description.Endpoints[0].Behaviors.Add(relayCredentials);**
      21. **host.Open();**
      22. **// create the channel factory loading the configuration**
      23. **channelFactory = new ChannelFactory<IWorkerRoleChannel>("RelayEndpoint", new EndpointAddress(serviceAddress));**
      24. **// apply the Service Bus credentials**
      25. **channelFactory.Endpoint.Behaviors.Add(relayCredentials);**
      27. **// create and open the client channel**
      28. **channel = channelFactory.CreateChannel();**
      29. **channel.Open();**
      30. Trace.WriteLine("Worker Role Manager: Connected", "Information");
      31. this.IsRunning = false;
      32. }
      33. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 ManagerWindowViewModel ConnectToMulticastRelay - VB*)
      34. Visual Basic
      35. Private Sub ConnectToMulticastRelay()
      36. ' Get connection info from the config file
      37. Dim serviceNamespace As String = ConfigurationManager.AppSettings("ServiceNamespace")
      38. Dim issuerName As String = ConfigurationManager.AppSettings("IssuerName")
      39. Dim issuerSecret As String = ConfigurationManager.AppSettings("IssuerSecret")
      40. 'establish connection
      41. **' create the credentials object for the endpoint**
      42. **Dim relayCredentials As New TransportClientEndpointBehavior()**
      43. **relayCredentials.CredentialType = TransportClientCredentialType.SharedSecret**
      44. **relayCredentials.Credentials.SharedSecret.IssuerName = issuerName**
      45. **relayCredentials.Credentials.SharedSecret.IssuerSecret = issuerSecret**
      46. **' create the service URI based on the service namespace**
      47. **Dim serviceAddress As Uri = ServiceBusEnvironment.CreateServiceUri("sb", serviceNamespace, "/JobProcessor/")**
      48. **' create service host**
      49. **host = New ServiceHost(GetType(JobManager), serviceAddress)**
      50. **host.Description.Endpoints(0).Behaviors.Add(relayCredentials)**
      51. **host.Open()**
      52. **' create the channel factory loading the configuration**
      53. **channelFactory = New ChannelFactory(Of IWorkerRoleChannel)("RelayEndpoint", New EndpointAddress(serviceAddress))**
      54. **' apply the Service Bus credentials**
      55. **channelFactory.Endpoint.Behaviors.Add(relayCredentials)**
      56. **' create and open the client channel**
      57. **channel = channelFactory.CreateChannel()**
      58. **channel.Open()**
      59. Trace.WriteLine(String.Format("Worker Role Manager: Connected", "Information"))
      60. Me.IsRunning = False
      61. End Sub
  18. Update the **StartProcessingCommand** and **StopProcessingCommand** methods to make them start/stop processing jobs respectively. This is shown in the following code fragments.
      1. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 ManagerWindowViewModel StartProcessingCommand - CS*)
      2. C#
      3. private void StartProcessingCommand(object parameter)
      4. {
      5. // start processing jobs
      6. **this.channel.StartProcessingJobs();**
      7. this.ProcessingCommand.ExecuteMethod = this.StopProcessingCommand;
      8. this.IsRunning = true;
      9. }
      10. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 ManagerWindowViewModel StartProcessingCommand - VB*)
      11. Visual Basic
      12. Private Sub StartProcessingCommand(ByVal parameter As Object)
      13. ' start processing jobs
      14. **Me.channel.StartProcessingJobs()**
      15. Me.ProcessingCommand.ExecuteMethod = AddressOf Me.StopProcessingCommand
      16. Me.IsRunning = True
      17. End Sub
      18. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 ManagerWindowViewModel StopProcessingCommand - CS*)
      19. C#
      20. private void StopProcessingCommand(object parameter)
      21. {
      22. // stop processing jobs
      23. **this.channel.StopProcessingJobs();**
      24. this.ProcessingCommand.ExecuteMethod = this.StartProcessingCommand;
      25. this.IsRunning = false;
      26. }
      27. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 ManagerWindowViewModel StopProcessingCommand - VB*)
      28. Visual Basic
      29. Private Sub StopProcessingCommand(ByVal parameter As Object)

' stop processing jobs

* + 1. **Me.channel.StopProcessingJobs()**
    2. Me.ProcessingCommand.ExecuteMethod = AddressOf Me.StartProcessingCommand
    3. Me.IsRunning = False
    4. End Sub
  1. Update the **Dispose** method to close the channel factory, channel and host objects that are created when connecting to the multicast relay. The following code shows how to achieve this.
     1. (Code Snippet - Introduction to Service Bus Lab Part 1 - *Ex03 ManagerWindowViewModel Dispose - CS*)
     2. C#
     3. public void Dispose()
     4. {
     5. **channel.Close();**
     6. **channelFactory.Close();**
     7. **host.Close();**
     8. }
     9. (Code Snippet - Introduction to Service Bus Lab Part 1 - Ex03 ManagerWindowViewModel Dispose - VB)
     10. Visual Basic
     11. Public Sub Dispose() Implements IDisposable.Dispose
     12. **channel.Close()**
     13. **channelFactory.Close()**
     14. **host.Close()**
     15. End Sub

Verification

* 1. In order to verify that you have performed every step in the exercise correctly, perform the following steps.
  2. In the Solution Explorer, right-click the **MulticastSample** solution file and select **Properties**.
  3. In the left pane tree view, select the **Startup Project** node, under **Common Properties**. The following figure shows the default configuration.
     1. 
     2. Figure 58
     3. Default Startup Project configuration
  4. Select the **Multiple startup projects** option.
  5. Change the action of the **JobsManagerClient** and **Multicasting** projects to **Start**. To do this, click the drop down button and select **Start**, as shown in the following figure.
     1. 
     2. Figure 59
     3. Changing the startup action for the projects
  6. Click **OK**.
  7. In the Solution Explorer, double-click the **ServiceConfiguration.cscfg** file under the **Multicasting** project.
  8. Note that the number of role instances to be run is three.
     1. XML
     2. <Role name="JobRole">
     3. **<Instances count="3" />**
     4. <ConfigurationSettings>
     5. <Setting name="Microsoft.WindowsAzure.Plugins.Diagnostics.ConnectionString" value="UseDevelopmentStorage=false" />
     6. </ConfigurationSettings>
     7. </Role>
  9. In Visual Studio, press **F5** to run the role management application. The **Worker Role Manager** window appears. It displays a list of running worker roles with their status and a trace console of messages received from those worker roles, as depicted in the following figure. Note that each worker role's messages are displayed with a different color.
     1. 
     2. Figure 60
     3. Worker Role Manager information
  10. Select a trace message from the **Status Console**. Note that the list item expands to show more information about that particular message.
      1. 
      2. Figure 61
      3. Viewing the details for a particular message
  11. Click **Start Processing**. Note that messages from the different worker roles will be displayed in the status console. Additionally, the status light for each worker role will turn green as long as the role is processing jobs. This is shown in the following figure.
      1. 
      2. Figure 62
      3. Worker Role Manager console showing status
  12. Click **Stop Processing**. Note that the jobs being processed when the signal to stop processing is received by the worker roles are completed, as it can be seen in the messages received after the processing stops. Additionally, when each worker role stops processing, its light turns red.
      1. 
      2. Figure 63
      3. Display when orders processing stops
  13. Close the **Worker** **Role** **Manager** window.

Summary

* 1. By completing this hands-on lab, you’ve learnt to create a service namespace domain and to host and consume your services using the AppFabric Service Bus. Additionally, you’ve seen how to connect a WCF Service hosted in IIS7.5 to the Service Bus and how to automatically activate the service. Finally, you’ve learnt to implement a multicast messaging service, using the netEventRelayBinding binding.