Hands-On Lab

Part 2: Introduction to the AppFabric Service Bus

Lab version: 2.0.0
Last updated: 12/15/2010
# CONTENTS

OVERVIEW ................................................................................................................................. 3

GETTING STARTED: CREATING A SERVICE PROJECT .......................................................... 6
 Task 1 – Creating your AppFabric Project ................................................................................. 6

EXERCISE 1: BINDINGS, CONNECTION MODES AND MEX ENDPOINTS ....................... 11
 Task 1 – Using Hybrid Connection Mode .............................................................................. 12
 Verification .............................................................................................................................. 16
 Task 2 – Changing the AppFabric Service Bus Binding .......................................................... 20
 Verification .............................................................................................................................. 23
 Task 3 – Exposing a Metadata Endpoint .............................................................................. 25
 Verification .............................................................................................................................. 31

EXERCISE 2: REST-BASED ACCESS AND BINARY DATA ................................................. 35
 Task 1 – Creating the Service Contracts .............................................................................. 35
 Task 2 – Configuring the Service Application ..................................................................... 37
 Verification .............................................................................................................................. 40

EXERCISE 3: MESSAGE BUFFERS ...................................................................................... 42
 Task 1 – Creating the Order Peek ...................................................................................... 43
 Task 2 – Creating the Order Submitter ................................................................................ 49
 Task 3 – Creating the Order Processor ................................................................................ 53
 Verification .............................................................................................................................. 57

SUMMARY ................................................................................................................................. 61
Overview

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).

Network devices like these typically allow applications to initiate outbound network connections but prevent them from accepting inbound network connections.

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.

The Service 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. Through the use of the Service Bus as an intermediary you can interconnect different applications without the need to write and maintain complex logic and code to traverse networks.

![Service Bus Diagram](image)

**Figure 1**

*Service Bus message flow in relayed mode*

This is the second part of the Introduction to the AppFabric Service Bus lab. In part 1 the basic Service Bus concepts have been explained and you have learnt how to get a basic service and client working via the Service Bus, connecting a WCF Service in IIS 7.5 to the Service Bus and creating a Multicast...
Messaging service. In this second part, more advanced topics will be explained and you will learn how to customize a service endpoint, create REST-based services and use the AppFabric message buffers.

Objectives

In this hands-on lab, you will learn how to:

- Service Bus bindings options.
- Expose a MEX endpoint through the Service Bus.
- Expose a REST Service through the Service Bus.
- Expose binary data through the Service Bus.
- Explore the Message Buffer Service Bus capabilities.

Prerequisites

You must have the following items to complete this lab:

- Microsoft Visual Studio 2010
- Microsoft .NET Framework 4
- Windows Azure platform AppFabric SDK V1.0

Setup

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:

1. Open a Windows Explorer window and browse to the lab’s Source\Setup folder.
2. 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.
3. If the User Account Control dialog is shown, confirm the action to proceed.

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

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.

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

This hands-on Lab includes the following exercises:

1. Bindings, Connection Modes and MEX Endpoints
2. REST-Based Access and Binary Data
3. Message Buffers

Estimated time to complete this lab: 60 minutes.

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 Project

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

Task 1 – Creating your AppFabric Project

In this task, you will create a new AppFabric project.

1. Navigate to the Windows Azure platform AppFabric portal. You will be prompted for your Windows Live ID credentials if you are not already signed in.

2. Create a new project. Type a Project Name, such as your company name or your name, accept the Terms of Use and click the OK button.

Figure 2

Creating a new Azure AppFabric project
3. Add a service namespace to your project. 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 the Project Name link for your new project to view its information page and then click Add Service Namespace.

![AppFabric project information page]

**Figure 3**
*AppFabric project information page*

4. Enter a name for your Service Namespace, select a Region for your service to run in, and click the Create 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.
Please be patient while your service is activated. It can take a few minutes while it is provisioned.

**Note:** You may have to refresh the browser to show the service is active.

5. Once the namespace is active, click its name in the list of available namespaces to display the Service Namespace information page.
Figure 5  
*Project summary page listing available service namespaces*  

6. In the **Service Namespace** information page, locate the Service Bus section and record the value shown for **Default Issuer Name** and **Default Issuer Key**. You will need these values later on to authenticate using the Access Control.
You have now created a new AppFabric project and defined a namespace for this hands-on lab. To sign in at any time, simply navigate to the Windows Azure platform AppFabric portal, click Sign In and provide your Live ID credentials. Clicking the AppFabric tab on the left will list the projects associated with your account.
My Projects

These are all the projects you have created or for which you have been designated as a Service Administrator by your Account Owner. Click on one project to see the list of services under it.

<table>
<thead>
<tr>
<th>Project Name</th>
<th>Account Owner</th>
<th>Service Administrator</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>YOUR-PROJECT</td>
<td></td>
<td></td>
<td>Active</td>
</tr>
</tbody>
</table>

If this is your first project, click on the project line to get started and begin adding services to the project.

Figure 7

My Projects page showing available AppFabric projects

Exercise 1: Bindings, Connection Modes and MEX Endpoints

In this exercise, you modify a simple application that demonstrates service-client communication to try different connection modes and bindings that can be used to communicate sender and receivers across the Service Bus.

The Service Bus allows the usage of numerous bindings and two different connection modes between senders and receivers that determine how the connection to the Service Bus is made. The different bindings and connection modes offer a great deal of flexibility for specific communication scenarios.

The following table describes the connection modes that will be shown during this exercise:

<table>
<thead>
<tr>
<th>Connection Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relayed</td>
<td>Configures the binding to use the relay to exchange messages between the client and the service. <strong>This is the default connection mode.</strong></td>
</tr>
<tr>
<td>Hybrid</td>
<td>Hybrid connection capabilities - After establishing a relayed connection, it switches automatically to a direct, non-relayed connection whenever possible. A direct connection mode between the client and the service is an optimization of the Relayed mode because it attempts to bridge a direct connection between the sender and receiver, if possible.</td>
</tr>
</tbody>
</table>

In addition, you learn how to expose a metadata endpoint through the Service Bus.
This exercise will expose several Service Bus features such as:

- Service Bus connection modes: **Hybrid** and **Relayed**
- Service Bus bindings
- Exposing metadata exchange (MEX) endpoints through the Service Bus

**Note:** To verify that each step is performed correctly, it is recommended to build the solution at the end of each task.

**Task 1 – Using Hybrid Connection Mode**

In this task, you change the connection mode to **Hybrid**.

1. Open Microsoft Visual Studio 2010 from **Start | All Programs | Microsoft Visual Studio 2010 | Microsoft Visual Studio 2010**.

2. Open the solution file located by default at **Ex01-BindingsConnectionModesSample\begin\{CS|VB\}\BindingsConnectionModesSample.sln** in the **Source** folder of the lab.

   **Note:** This solution contains a locally hosted WCF service that exposes itself to the Service Bus and a client that consumes the service via the Service Bus. The service namespace domain and credentials necessary to authenticate the service and the client with the Service bus are retrieved from their respective configuration files.

3. Open the **App.config** file in the **Client** project. Add a section for the WCF **bindings** inside the **system.serviceModel** section and include an override for the **netTcpRelayBinding** binding. This is the plumbing for adding a custom binding configuration, overriding the default behavior of the **netTcpRelayBinding**. To do this, add the following highlighted code.

   (Code Snippet - Introduction to Service Bus Lab Part 2 - Ex01 Client NetTcpRelayBinding - XML)

   **XML**
   
   ```xml
   <configuration>
   <system.serviceModel>
   <bindings>
      <!-- Application Binding -->
      <netTcpRelayBinding>
         <binding name="lab">
            <security mode="None" />
         </binding>
      </netTcpRelayBinding>
   </bindings>
   </system.serviceModel>
   </configuration>
   ```
4. Now, change the **Client** connection mode to **Hybrid**. To do this, add a **connectionMode** attribute to the binding configuration and set its value to **Hybrid**.

   **XML**

   ```xml
   <system.serviceModel>
   ...
   <client>
   <bindings>
   <!-- Application Binding -->
   <netTcpRelayBinding>
   <binding name="lab" connectionMode="Hybrid">
   <security mode="None"/>
   </binding>
   </netTcpRelayBinding>
   </bindings>
   ...
   </client>
   </system.serviceModel>
   ```

   **Note:** One important property on the **NetTcpRelayBinding** object is **ConnectionMode**. The property type is an enumeration named **Microsoft.ServiceModel.TcpRelayConnectionMode** and describes the type of connection to be established between clients and services.

   Here, the connection mode is set to **Hybrid**, which first establishes a relayed connection, and if possible, switches automatically to a direct connection between a client and service.

5. Next, add a **bindingConfiguration** attribute to associate this binding to the endpoint as shown in the following code:

   **XML**

   ```xml
   <system.serviceModel>
   ...
   <client>
   <!-- Application Endpoint -->
   <endpoint name="RelayEndpoint" contract="Client.IEchoContract">
   binding="netTcpRelayBinding"
   </endpoint>
   ...
   </client>
   ```
Now, configure the service. Open the App.config file in the Service project and add a bindings section to define a netTcpRelayBinding, as shown in the following code:

(XML - Introduction to Service Bus Lab Part 2 - Ex01 Service NetTcpRelayBinding - XML)

```xml
<configuration>
  <system.serviceModel>
    <!-- Application Bindings -->
    <bindings>
      <netTcpRelayBinding>
        <binding name="lab">
          <security mode="None" />
        </binding>
      </netTcpRelayBinding>
    </bindings>
    ...
  </system.serviceModel>
</configuration>
```
7. Next, change the Service connection mode to Hybrid as you did for the client. To do this, add a connectionMode attribute to the binding configuration as shown below:

**XML**

```xml
<bindings>
  <netTcpRelayBinding>
    <binding name="lab" connectionMode="Hybrid">
      <security mode="None" />  
    </binding>
  </netTcpRelayBinding>
</bindings>
```

8. Associate the newly created binding to the service endpoint by setting its bindingConfiguration attribute as shown below:

**XML**

```xml
<system.serviceModel>
  ...
  <services>
    <!-- Application Service -->
    <service name="Service.EchoService">
      <endpoint contract="Service.IEchoContract"
        binding="netTcpRelayBinding"
        bindingConfiguration="lab"
        behaviorConfiguration="SharedSecretCredentials" />
    </service>
  </services>
</system.serviceModel>
```

9. Update the client application to display a notification when the connection switches to Hybrid mode after communication has been established. To do this, add the following highlighted code to the Program.cs file (for Visual C# projects) or Module1.vb file (for Visual Basic projects) of the Client project immediately after the line that opens the channel.

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex01 Connection Change Notification - CS)
C#

```csharp
...
(ICommunicationObject)channel).Open();

IHybridConnectionStatus hybridConnectionStatus =
((System.ServiceModel.Channels.IChannel)channel).GetProperty<IHybridConnectionStatus>();
if (hybridConnectionStatus != null)
{
    hybridConnectionStatus.ConnectionStateChanged += (o, e) =>
    {
        Console.WriteLine("Connection state changed to: {0}.", e.ConnectionState);
    };
}

Console.WriteLine("Enter text to echo (or [Enter] to exit):";)
```

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex01 Connection Change Notification - VB)

Visual Basic

```vbnet
...
' create and open the client channel
Dim channel As IEchoContract = channelFactory.CreateChannel()
CType(channel, ICommunicationObject).Open()

Dim hybridConnectionStatus As IHybridConnectionStatus = (CType(channel, System.ServiceModel.Channels.IChannel)).GetProperty(Of IHybridConnectionStatus)()
If hybridConnectionStatus IsNot Nothing Then
    AddHandler hybridConnectionStatus.ConnectionStateChanged, Sub(o, e)
    Console.WriteLine("Connection state changed to: {}.", e.ConnectionState)
End If

Console.WriteLine("Enter text to echo (or [Enter] to exit):")
Dim input As String = Console.ReadLine()
```

**Verification**

1. Open the App.config file in the Client project. In the appSettings section, locate the serviceNamespaceDomain setting and replace [YOUR-SERVICE-NAMESPACE-DOMAIN] with the domain service namespace that you registered at the portal. Next, in the behaviors section
inside `system.serviceModel`, locate the `sharedSecret` element inside `endpointBehaviors | behavior | transportClientEndpointBehavior | clientCredentials` 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 in the Getting Started section.

```xml
<configuration>
  <add key="serviceNamespaceDomain" value="[YOUR-SERVICE-NAMESPACE-DOMAIN]" />
  ...
  <behaviors>
    <endpointBehaviors>
      <behavior name="sharedSecretCredentials">
        <transportClientEndpointBehavior credentialType="SharedSecret">
          <clientCredentials>
            <sharedSecret issuerName="[YOUR-ISSUER-NAME]" issuerSecret="[YOUR-ISSUER-KEY]" />
          </clientCredentials>
        </transportClientEndpointBehavior>
      </behavior>
    </endpointBehaviors>
  </behaviors>
</system.serviceModel>
</configuration>
```

Figure 9
Configuring the service namespace domain and shared secret credentials

2. Similarly, open the `App.config` file in the Service project and update `[YOUR-SERVICE-NAMESPACE-DOMAIN]`, `[YOUR-ISSUER-NAME]` and `[YOUR-ISSUER-KEY]` with the same values used in the previous step.

3. Launch the service host. To do this, right-click the Service project in Solution Explorer, point to Debug and select Start new instance.
4. Now, execute the client application. In Solution Explorer, right-click the Client project, point to Debug and select Start new instance. Notice that after a few seconds the console displays a message announcing that the connection has changed to Direct mode.

![Client](image)

**Figure 11**

*Client running and connection mode changed to Direct mode.*

**Note:** When running the Client, the firewall may pop up a message at this point. If this happens, click Allow Access.

5. Send several messages to ensure the messages are correctly echoed by the service.
Figure 12
Service receives and outputs messages

Note: When a direct connection between the sender and receiver is established, the messages do not need to be relayed through the Service Bus.

Figure 13
Hybrid Connection Mode
6. Close both the **Client** and the **Service**.

**Task 2 – Changing the AppFabric Service Bus Binding**

In this task, you learn how to change the transport protocol by using different bindings provided by the Service Bus.

1. Change the binding used in the Service from **netTcpRelayBinding** to **ws2007HttpRelayBinding**. To do this, open the **App.config** file in the **Service** project and replace the **bindings** section with the following code.

   (Code Snippet - Introduction to Service Bus Lab Part 2 - *Ex01 Service ws2007HttpRelayBinding - XML* )

   ```xml
   <system.serviceModel>
   <behaviors>
   ...
   </behaviors>
   <!-- Application Binding -->
   <bindings>
   <ws2007HttpRelayBinding>
   <binding name="lab">
   <security mode="Transport"/>
   </binding>
   </ws2007HttpRelayBinding>
   </bindings>
   ...
   </system.serviceModel>
   ```

   **Note:** The binding uses transport security to provide the secure channel necessary to protect the access control tokens.

2. Change the binding specified for the service endpoint from **netTcpRelayBinding** to **ws2007HttpRelayBinding** as shown below (highlighted in **bold**).

   ```xml
   <system.serviceModel>
   <bindings>
   <!-- Application Binding -->
   <ws2007HttpRelayBinding>
   <binding name="lab">
   <security mode="None" />
   </binding>
   </ws2007HttpRelayBinding>
   </bindings>
   ```
3. Change the endpoint address where the service will be hosted to match the current binding. To do this, open Program.cs (for Visual C# projects) or Module1.vb (for Visual Basic projects) in the Service project, locate the call to CreateServiceUri in method Main and update the scheme used to create the URI from sb to https as shown below (in bold text).

C#:

```csharp
static void Main(string[] args)
{
    Console.Title = "Service";
    // retrieve service namespace domain from the configuration file
    string serviceNamespaceDomain = ConfigurationManager.AppSettings["serviceNamespaceDomain"];

    // create the service URI based on the service namespace
    Uri address = ServiceBusEnvironment.CreateServiceUri("https", serviceNamespaceDomain, "EchoService");
...
```

Visual Basic:

```vbnet
Sub Main()
    Console.Title = "Service"
    ' retrieve service namespace domain from the configuration file
    Dim serviceNamespaceDomain As String = ConfigurationManager.AppSettings("serviceNamespaceDomain")

    ' create the service URI based on the service namespace
    Dim address As Uri = ServiceBusEnvironment.CreateServiceUri("https", serviceNamespaceDomain, "EchoService")
...
End Sub
```
4. Now, change the binding for the client. Open App.config in the Client project and replace the bindings section with the following code.

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex01 Client ws2007HttpRelayBinding - XML)

```
<system.serviceModel>
    <bindings>
        <!-- Application Binding -->
        <ws2007HttpRelayBinding>
            <binding name="lab">
                <security mode="Transport"/>
            </binding>
        </ws2007HttpRelayBinding>
    </bindings>

    <!-- Additional bindings...

    <bindings>
        ...
    </bindings>

    <client>
        <endpoint name="RelayEndpoint"
            contract="Client.IEchoContract"
            binding="ws2007HttpRelayBinding"
            bindingConfiguration="lab" behaviorConfiguration="SharedSecretCredentials"/>
    </client>
</system.serviceModel>
```

**Note:** The binding uses transport security to provide the secure channel necessary to protect the access control tokens.

5. Change the binding specified for the client endpoint from netTcpRelayBinding to ws2007HttpRelayBinding, as shown below (in **bold** text).

```
<system.serviceModel>
    <bindings>
        ...
        <bindings>

        <client>
            <endpoint name="RelayEndpoint"
                contract="Client.IEchoContract"
                binding="ws2007HttpRelayBinding"
                bindingConfiguration="lab" behaviorConfiguration="SharedSecretCredentials"/>
        </client>
    </bindings>
</system.serviceModel>
```
6. Change the endpoint address where the service is now listening to match the current binding. To do this, open **Program.cs** (for Visual C# projects) or **Module1.vb** (for Visual Basic projects) in the **Client** project, locate the call to **CreateServiceUri** in method **Main** and update the scheme used to create the URI from **sb** to **https** as show below (in **bold**).

**C#**

```csharp
static void Main(string[] args)
{
    Console.Title = "Client";
    // retrieve service namespace domain from the configuration file
    string serviceNamespaceDomain = ConfigurationManager.AppSettings["serviceNamespaceDomain"];

    // create the service URI based on the service namespace
    Uri serviceUri = ServiceBusEnvironment.CreateServiceUri("
https", serviceNamespaceDomain, "EchoService");
    ... 
}
```

**Visual Basic**

```visualbasic
Sub Main()
    Console.Title = "Client"
    ' retrieve service namespace domain from the configuration file
    Dim serviceNamespaceDomain As String = ConfigurationManager.AppSettings("serviceNamespaceDomain")

    ' create the service URI based on the service namespace
    Dim address As Uri = ServiceBusEnvironment.CreateServiceUri("https", serviceNamespaceDomain, "EchoService")
    ...
End Sub
```

**Verification**

1. Launch the service. To do this, right-click the **Service** project in **Solution Explorer**, point to **Debug** and select **Start new instance**. Notice that the service is now listening at an HTTP endpoint.
Figure 14
Service listening on the Service Bus using ws2007HttpRelayBinding

Note: HTTP services require administrative privileges in order to register a URL. For instance, if you see the following error message:

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:

```
netsh http add urlacl url=http://+:80/EchoService/ user=DOMAIN\user
```

Where `DOMAIN\user` is the user that will run the Service console application.

Notice that you could also workaround this by launching Visual Studio as an administrator.
2. Now, execute the Client application. In Solution Explorer, right-click the Client project, point to Debug and select Start new instance.

3. Send several messages to ensure they are correctly echoed by the service.

![Figure 15](image)

Service receives and outputs the messages

4. Stop debugging both services.

Task 3 – Exposing a Metadata Endpoint

In this task, you learn how to expose a metadata endpoint through the Service Bus.

1. Add the metadata endpoint definition to the service configuration. Open App.config in the Service project and add the following endpoint element to the service section in system.serviceModel.services.

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex01 Service MexEndpoint - XML)

```
<services>
    <service name="Service.EchoService">
        <endpoint contract="Service.IEchoContract" binding="ws2007HttpRelayBinding">
            <!-- Add additional endpoint attributes here -->
        </endpoint>
        <!-- Add more endpoints if needed -->
    </service>
</services>
```

Page 25
2. To configure metadata publishing for the Service, update the behaviors section in system.serviceModel to add a serviceBehaviors section as shown below:

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex01 Service Behaviors - XML)

XML

```xml
<system.serviceModel>
 ...
 <behaviors>
   <serviceBehaviors>
     <behavior name="serviceMetadata" />
     <serviceMetadata /> 
   </behavior>
   <serviceBehaviors>
     ... 
   </serviceBehaviors>
 </behaviors>
</system.serviceModel>
```

3. Set the metadata behavior for the service. To do this, add the following behaviorConfiguration attribute to the service definition:

XML

```xml
<services>
 <service name="Service.EchoService" behaviorConfiguration="serviceMetadata">
 ... 
 </service>
</services>
```
4. Restore the service binding to **netTcpRelayBinding** by replacing the existing binding in the bindings section of **system.serviceModel** with the following code (in **bold** text).

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex01 Service Restoring NetTcpRelayBinding - XML)

XML

```xml
<system.serviceModel>
  <behaviors>...
  </behaviors>
  <bindings>
    <!-- Application Binding -->
    <netTcpRelayBinding>
      <binding name="lab">
        <security mode="None" />
      </binding>
    </netTcpRelayBinding>
  </bindings>

  ...</system.serviceModel>
```

5. Next, restore the binding of the service endpoint to **netTcpRelayBinding** as shown below (in **bold** text).

XML

```xml
<system.serviceModel>
  ...<services>
    <service name="Service.EchoService"
        behaviorConfiguration="serviceMetadata">
      <endpoint contract="Service.IEchoContract"
          binding="netTcpRelayBinding"
          bindingConfiguration="lab"
          behaviorConfiguration="SharedSecretCredentials" />

      ...</endpoint>
    </service>
  </services>

  ...</system.serviceModel>
```

6. Change the endpoint address where the service will be hosted to match the current binding. To do this, open **Program.cs** (for Visual C# projects) or **Module1.vb** (for Visual Basic projects) in the **Service** project, locate the call to **CreateServiceUri** in method **Main** and update the scheme used to create the URI from **https** to **sb** as shown in the following code (in **bold** text).
In the **Program.cs** file (for Visual C# projects) or the **Module1.vb** file (for Visual Basic projects), add the following namespace declaration below the existing namespace directives.

### C#

```csharp
using System.ServiceModel.Dispatcher;
```

### Visual Basic

```vbnet
Imports System.ServiceModel.Dispatcher
```

In method **Main**, insert the following (highlighted) code to list the addresses where the service listens.

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex01 Display Addresses - CS)
C#  

```csharp
static void Main(string[] args)
{
    ...
    host.Open();

    foreach (ChannelDispatcherBase channelDispatcherBase in host.ChannelDispatchers)
    {
        ChannelDispatcher channelDispatcher = channelDispatcherBase as ChannelDispatcher;
        foreach (EndpointDispatcher endpointDispatcher in channelDispatcher.Endpoints)
        {
            Console.WriteLine("Listening at: {0}", endpointDispatcher.EndpointAddress);
        }
    }

    Console.WriteLine("Service address: " + address);
    ...
}
```

Visual Basic  

```vbnet
...  
host.Open()

For Each channelDispatcherBase As ChannelDispatcherBase In host.ChannelDispatchers
    Dim channelDispatcher As ChannelDispatcher = TryCast(channelDispatcherBase, ChannelDispatcher)
    For Each endpointDispatcher As EndpointDispatcher In channelDispatcher.Endpoints
        Console.WriteLine("Listening at: {0}", endpointDispatcher.EndpointAddress)
        Next endpointDispatcher
    Next channelDispatcherBase

Console.WriteLine("Service address: " & address.ToString())
Console.WriteLine("Press [Enter] to exit")
Console.ReadLine()
...
```

9. Open the App.config file in the Client project.
10. Restore the client binding to **netTcpRelayBinding** by replacing the existing binding in the bindings section of **system.serviceModel** with the following code (in **bold** text).

**XML**

```xml
<system.serviceModel>
  ...
  <bindings>
    <!-- Application Binding -->
    <netTcpRelayBinding>
      <binding name="lab"/>
    </netTcpRelayBinding>
  </bindings>
  ...
</system.serviceModel>
```

11. Next, restore the binding of the client endpoint to **netTcpRelayBinding** as shown below (in **bold** text).

**XML**

```xml
<system.serviceModel>
  ...
  <client>
    <!-- Application Endpoint -->
    <endpoint name="RelayEndpoint"
      contract="Client.IEchoContract"
      binding="netTcpRelayBinding"
      bindingConfiguration="lab"
      behaviorConfiguration="SharedSecretCredentials"/>
  </client>
  ...
</system.serviceModel>
```

12. Finally, restore the endpoint address where the service is now listening to match the current binding. To do this, open **Program.cs** (for Visual C# projects) or **Module1.vb** (for Visual Basic projects) in the **Client** project, locate the call to **CreateServiceUri** in method **Main** and update the scheme used to create the URI from **https** to **sb** as show below (in **bold**).

**C#**

```csharp
static void Main(string[] args)
{
  Console.Title = "Client";
  // retrieve service namespace domain from the configuration file
```
string serviceNamespaceDomain =
ConfigurationManager.AppSettings["serviceNamespaceDomain"];  

// create the service URI based on the service namespace
Uri serviceUri = ServiceBusEnvironment.CreateServiceUri("sb",
serviceNamespaceDomain, "EchoService");
...

Visual Basic
Sub Main()
    Console.Title = "Client"
    ' retrieve service namespace domain from the configuration file
    Dim serviceNamespaceDomain As String =
ConfigurationManager.AppSettings("serviceNamespaceDomain")

    ' create the service URI based on the service namespace
    Dim address As Uri = ServiceBusEnvironment.CreateServiceUri("sb",
serviceNamespaceDomain, "EchoService")

    ...
End Sub

Verification
In order to verify that the service is exposing metadata, proceed as follows. You will use the ServiceModel Metadata Utility (svcutil) to download metadata documents from the service.

Note: Svcutil.exe is installed as part of the Windows SDK. A typical location might be %ProgramFiles%\Microsoft SDKs\Windows\V7.0a\Bin\NETFX 4.0 Tools\SvcUtil.exe.

1. Launch the service. To do this, right-click the Service project in Solution Explorer, point to Debug and select Start new instance.
In order to download metadata from the running service, you need to change the settings in `SvcUtil.exe.config` file to authenticate the client. To do this, navigate to `%ProgramFiles%\Microsoft SDKs\Windows\V7.0a\Bin\NETFX 4.0 Tools`. Open the `SvcUtil.exe.config` file, or create it if does not exist, and then add the following settings inside the configuration section. Replace the `ISSUER_NAME` and `ISSUER_SECRET` placeholder strings with the Default Issuer Name and Default Issuer Key of your service namespace.

```
<system.serviceModel>
  <bindings>
    <netTcpRelayBinding>
      <binding name="default" />
    </netTcpRelayBinding>
  </bindings>
</system.serviceModel>
```

**Note:** You may wish to create a backup of the configuration file before performing this change so that you can restore its original state after you complete the exercise.

If you need to create the configuration file, make sure that you also include the `configuration` root element.

**Note:** The `SvcUtil.exe.config` may reside in another place in your computer. For example, on 64 bits environments, it would be located inside `%ProgramFiles%\Microsoft SDKs\Windows\V7.0a\Bin\NETFX 4.0 Tools\64\`.

Figure 16
Running service showing an additional mex endpoint for metadata
3. Execute the `Svcutil` tool to download the metadata from the running service. To do this, open a Visual Studio Command Prompt from Start | All Programs | Microsoft Visual Studio 2010 | Visual Studio Tools | Visual Studio Command Prompt (2010), change the current directory to the folder where you wish to store the generated files (using the `CD` command) - the folder containing the source code for the lab is recommended - and execute the following command.

**Note:** Please replace `[serviceNamespace]` with your current service namespace, which you created in the Getting Started section.

**Command (Visual C#)**

```
svcutil.exe /language:CS
sb://[serviceNamespace].servicebus.windows.net/EchoService/mex
```

**Command (Visual Basic)**

```
svcutil.exe /language:VB
sb://[serviceNamespace].servicebus.windows.net/EchoService/mex
```
4. The command creates an **EchoService.cs** (for Visual C# projects) or **EchoService.vb** file (for Visual Basic projects) that contains a client proxy for the service and an **Output.config** file with configuration settings.

![Image of Visual Studio Command Prompt]

**Figure 17**

*Running the Svcutil tool to download metadata*

5. The generated **output.config** file should be similar to the following:

```xml
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.serviceModel>
    <bindings>
      <netTcpRelayBinding>
        <binding name="RelayEndpoint" closeTimeout="00:01:00"
          openTimeout="00:01:00" receiveTimeout="00:10:00"
          sendTimeout="00:01:00" transferMode="Buffered"
          connectionMode="Relayed" listenBacklog="10"
          maxBufferPoolSize="524288" maxBufferSize="65536"
          maxConnections="10" maxReceivedMessageSize="65536">
          <readerQuotas maxDepth="32" maxStringLength="8192"
            maxArrayLength="16384" maxBytesPerRead="4096"
            maxNameTableCharCount="16384" />
          <reliableSession ordered="true" inactivityTimeout="00:10:00"
            enabled="false" />
        </binding>
        <security mode="None"
          relayClientAuthenticationType="RelayAccessToken">
          <transport protectionLevel="EncryptAndSign" />
      </security>
    </netTcpRelayBinding>
    <basicHttpBinding>
      <binding name="BasicHttpEndpoint" closeTimeout="00:01:00"
        openTimeout="00:01:00" receiveTimeout="00:10:00"
        sendTimeout="00:01:00" transferMode="Buffered"
        connectionMode="Relayed" listenBacklog="10"
        maxBufferPoolSize="524288" maxBufferSize="65536"
        maxConnections="10" maxReceivedMessageSize="65536">
        <readerQuotas maxDepth="32" maxStringLength="8192"
          maxArrayLength="16384" maxBytesPerRead="4096"
          maxNameTableCharCount="16384" />
        <reliableSession ordered="true" inactivityTimeout="00:10:00"
          enabled="false" />
        <security mode="None"
          relayClientAuthenticationType="RelayAccessToken">
          <transport protectionLevel="EncryptAndSign" />
      </security>
    </basicHttpBinding>
  </bindings>
</system.serviceModel>
</configuration>
```
Exercise 2: REST-Based Access and Binary Data

In this exercise, you learn how to expose a REST Service through the Service Bus. This service, when accessed through a web browser, will expose binary data through a URL with the following pattern: `http://[ServiceNamespace].servicebus.windows.net/Image/`, where `[ServiceNamespace]` is the service namespace domain that you choose for your project.

Notice that the protocol being used differs from the one you looked at in the previous exercise. You are still accessing the service through the Service Bus but now you are taking advantage of the `webHttpRelayBinding` instead of the default `netTcpRelayBinding` used before.

This exercise will expose several Service Bus features such as:

- REST-based access to services
- Exposing binary data through the Service Bus

For more information on REST, see [WCF REST Programming Model](#).

**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 Service Contracts**

2. Open the **RESTSample.sln** solution file from `Ex02-RESTSample\begin\{CS\,|\,VB\}` in the **Source** folder of the lab.

3. Define the contract for the service. To do this, in **Solution Explorer**, double-click **ImageContract.cs** (for Visual C# projects) or **ImageContract.vb** (for Visual Basic projects) and paste the following code (shown in **bold**) inside the **ImageContract** interface.

   (Code Snippet - Introduction to Service Bus Lab Part 2 - Ex02 GetImage Operation Contract - CS)

   **C#**
   
   ```csharp
   [ServiceContract(Name = "ImageContract", Namespace = "http://samples.microsoft.com/ServiceModel/Relay")]
   public interface IImageContract
   {
       [OperationContract, WebGet(UriTemplate="" )]
       Stream GetImage();
   }
   ```

   (Code Snippet - Introduction to Service Bus Lab Part 2 - Ex02 GetImage Operation Contract - VB)

   **Visual Basic**
   
   ```vb
   <ServiceContract(Name:="ImageContract",
   Namespace:="http://samples.microsoft.com/ServiceModel/Relay/")> 
   Public Interface IImageContract
   <OperationContract(), WebGet(UriTemplate:="" )> 
   Function GetImage() As Stream
   End Interface
   ```

   **Note:** The **GetImage** method is adorned with an **OperationContract** attribute and a **WebGet** attribute. The latter associates the service operation with the HTTP GET verb, so that incoming requests that specify this verb are routed to this operation. The **UriTemplate** associates a specific URI with the operation. In this case, an empty value is specified, which means that the service will use the base URI.

4. Implement the **GetImage** operation of the **ImageService**. Open **ImageService.cs** (for Visual C# projects) or **ImageService.vb** (for Visual Basic projects) and add the following code to the **ImageService** class.

   (Code Snippet - Introduction to Service Bus Lab Part 2 - Ex02 GetImage Implementation - CS)

   **C#**
   
   ```csharp
   ```

   ```vb
   ```
public class ImageService : IImageContract
{
    ...

    public Stream GetImage()
    {
        MemoryStream stream = new MemoryStream();
        this.bitmapImage.Save(stream, ImageFormat.Jpeg);

        stream.Position = 0;

        return stream;
    }
}

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex02 GetImage Implementation - VB)

Visual Basic
Public Class ImageService
    Implements IImageContract
    ...

    Public Function GetImage() As Stream Implements IImageContract.GetImage
        Dim stream As New MemoryStream()
        Me.bitmapImage.Save(stream, ImageFormat.Jpeg)

        stream.Position = 0
        WebOperationContext.Current.OutgoingResponse.ContentType = "image/jpeg"

        Return stream
    End Function
End Class

Note: The code creates a memory stream from the bitmap read from disk in the constructor, sets the content type of the response, and returns the stream as the operation response to be sent back to the client.

Task 2 – Configuring the Service Application

In this task, you create the configuration file (App.config) for the service.
1. Configure the service binding to use a **WebHttpRelayBinding**. This binding allows you to expose a service that can be accessed through HTTP. Open **App.config** and paste the following code (shown in **bold**) in the **bindings** section.

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex02 *webHttpRelayBinding* -XML)

**XML**

```xml
<bindings>
  <!-- Application Binding -->
  <webHttpRelayBinding>
    <binding name="lab">
      <!-- Turn off client authentication so that client does not need to present credential through browser or fiddler -->
      <security relayClientAuthenticationType="None" />
    </binding>
  </webHttpRelayBinding>
</bindings>
```

2. Configure the service and its endpoints. To do this, paste the following code inside the **services** section.

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex02 *REST Service* -XML)

**XML**

```xml
<services>
  <!-- Application Service -->
  <service name="RestService.ImageService" behaviorConfiguration="default">
    <endpoint contract="RestService.IImageContract"
      binding="webHttpRelayBinding"
      bindingConfiguration="lab"
      behaviorConfiguration="SharedSecretCredentials" />
  </service>
</services>
```

3. Set the **issuerName** and **issuerSecret** attribute in the **sharedSecret** section. These are the **Default Issuer Name** and **Default Issuer Key** of your **Service Namespace Domain**, which you obtained when provisioning your account at the portal Web Site.
4. Create a new **WebServiceHost** to expose the REST service on the Service Bus. To do this, open **Program.cs** (for Visual C# projects) or **Module1.vb** (for Visual Basic projects) and paste the code shown below (in **bold** text), replacing the existing code in the file that creates a regular **ServiceHost**.

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex02 Service URI – CS)

```csharp
private static void Main()
{
    Console.Title = "REST Image Service";
    Console.WriteLine("Your Service Namespace Domain (e.g. https://[YOUR-SERVICE-NAMESPACE-DOMAIN].servicebus.windows.net/): ");
    string serviceNamespaceDomain = Console.ReadLine();

    Uri address = ServiceBusEnvironment.CreateServiceUri("https", serviceNamespaceDomain, "Image");

    WebServiceHost host = new WebServiceHost(typeof(ImageService), address);
    host.Open();

    Console.WriteLine("Copy the following address into a browser to see the image: ");
    Console.WriteLine(address);
    Console.WriteLine();
    Console.WriteLine("Press [Enter] to exit");
    Console.ReadLine();

    host.Close();
}
```
### Visual Basic

```visual-basic
Sub Main()
    Console.Write("Your Service Namespace Domain (e.g. https://<YOUR-SERVICE-NAMESPACE-DOMAIN>.servicebus.windows.net/): ")
    Dim serviceNamespaceDomain As String = Console.ReadLine()

    Dim address As Uri = ServiceBusEnvironment.CreateServiceUri("https", serviceNamespaceDomain, "Image")
    Dim host As New WebServiceHost(GetType(ImageService), address)

    host.Open()

    Console.WriteLine("Copy the following address into a browser to see the image: ")
    Console.WriteLine(address)
    Console.WriteLine()
    Console.WriteLine("Press [Enter] to exit")
    Console.ReadLine()

    host.Close()
End Sub
```

**Note:** The service will be listening at:

`https://[YOUR-SERVICE-NAMESPACE-DOMAIN].servicebus.windows.net/Image/`

Notice that the service is hosted by an instance of the `WebServiceHost`. The `WebServiceHost` is similar to the `ServiceHost` class, but it makes it easier to expose services that are built on HTTP and POX. For a comparison of these two classes, see [WebServiceHost vs ServiceHost](#).

---

### Verification

In order to verify that you have performed every step of the exercise correctly, proceed as follows.

1. Launch the service. To do this, right-click the `Service` project, point to `Debug` and select `Start new instance`. When prompted, provide the service namespace you set up during the provisioning process at the developer portal. At this point, the service should indicate that it is listening at the configured address.
2. Access the service URL from your browser. To do this, open a browser window and navigate to the URL indicated in the service console (e.g. https://YOUR-SERVICE-NAMESPACE-DOMAIN/servicebus.windows.net/Image/). The service should return an image in its response.
3. Close the browser window and press ENTER at the service console to shut down the service.

Exercise 3: Message Buffers

In this exercise, you learn how message buffers work in AppFabric and how to use them.

The exercise is based on an order-processing scenario. It consists of three small applications: the order submitter, the order processor, and the order peeker. The submitter sends orders as typed messages to an AppFabric message buffer. These orders are read from the AppFabric message buffer and processed by one or more order processors. Finally, the order peeker allows you to see the first pending order that needs processing at any given time.
The following figure shows how the scenario works.

![Figure 21: Message buffer scenario](image)

This exercise shows how to create, connect and delete a message buffer. In addition, you learn how to peek/lock, release/lock and retrieve messages from a buffer.

This exercise will expose the usage of the following classes:

- MessageBufferPolicy
- MessageBufferClient
- TypedMessageConverter
- TransportClientEndpointBehavior

**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 Order Peek**
1. Open Microsoft Visual Studio 2010 from **Start | All Programs | Microsoft Visual Studio 2010 | Microsoft Visual Studio 2010**.

2. Open the **MessageBuffer.sln** solution file from **Ex03-MessageBuffer\begin\{CS|VB\}** in the **Source** folder of the lab.

3. Add a new method called **EnsureMessageBufferExists** that verifies if the message buffer is already created, and if not, creates a new one. To do this, in **Solution Explorer**, double-click **Program.cs** (for Visual C# projects) or **Module1.vb** (for Visual Basic projects) in the **OrderPeek** project and paste the following code at the end of the **Program** or **Module1** class.

   (Code Snippet - Introduction to Service Bus Lab Part 2 - Ex03 **EnsureMessageBufferExists method** - CS)

   ```csharp
   private static MessageBufferClient EnsureMessageBufferExists(TransportClientEndpointBehavior endpointBehavior, Uri messageBufferUri, ref MessageBufferPolicy messageBufferPolicy)
   {
      MessageBufferClient client;

      try
      {
         client = MessageBufferClient.GetMessageBuffer(endpointBehavior, messageBufferUri);
         messageBufferPolicy = client.GetPolicy();
         Console.WriteLine("Using existing message buffer.");
         return client;
      }
      catch (EndpointNotFoundException)
      {
         // Not found; absorb and make a new message buffer below.
         // Other exceptions get bubbled up.
      }
      catch (FaultException)
      {
      }
      
      client = MessageBufferClient.CreateMessageBuffer(endpointBehavior, messageBufferUri, messageBufferPolicy);
      Console.WriteLine("New message buffer created.");
      return client;
   }
   ```

   (Code Snippet - Introduction to Service Bus Lab Part 2 - Ex03 **EnsureMessageBufferExists method** - VB)
Visual Basic

Private Function EnsureMessageBufferExists(ByVal endpointBehavior As TransportClientEndpointBehavior, ByVal messageBufferUri As Uri, ByRef messageBufferPolicy As MessageBufferPolicy) As MessageBufferClient
    Dim client As MessageBufferClient

    Try
        client = MessageBufferClient.GetMessageBuffer(endpointBehavior, messageBufferUri)
        messageBufferPolicy = client.GetPolicy()
        Console.WriteLine("Using existing message buffer.")
        Return client
    Catch e1 As EndpointNotFoundException
        ' Not found; absorb and make a new message buffer below.
        ' Other exceptions get bubbled up.
        Catch e2 As FaultException
            End Try
    
    client = MessageBufferClient.CreateMessageBuffer(endpointBehavior, messageBufferUri, messageBufferPolicy)
    Console.WriteLine("New message buffer created.")
    Return client
End Function

If the preceding method successfully creates the message buffer, it uses the provided message buffer policy.

Note: The MessageBufferClient static class contains methods to get and create a message buffer.

4. Modify method Main to create a MessageBufferPolicy and invoke the EnsureMessageBufferExists method added in the previous step. You have to pass the endpoint behavior, the message buffer URI, and the message buffer policy as parameters. To do this, paste the following code below the corresponding comment placeholder.

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex03 Invoke EnsureMessageBufferExists method - CS)

C#

private static void Main()
{
    ... 

    // Get Message Buffer URI
Console.WriteLine("Message Buffer Address: " + messageBufferUri.AbsoluteUri);
Console.WriteLine();

// Message Buffer Policy
MessageBufferPolicy messageBufferPolicy = new MessageBufferPolicy();
messageBufferPolicy.MaxMessageCount = 10;
messageBufferPolicy.ExpiresAfter = TimeSpan.FromHours(1); // messages in the message buffer expire after 1 hour
messageBufferPolicy.TransportProtection = TransportProtectionPolicy.None;

MessageBufferClient messageBufferClient =
EnsureMessageBufferExists(endpointBehavior, messageBufferUri, ref messageBufferPolicy);

Sub Main()
...

' Get Message Buffer URI
Dim messageBufferUri As Uri = ServiceBusEnvironment.CreateServiceUri("https", serviceNamespaceDomain, "OrderMessageBuffer")
Console.WriteLine("Message Buffer Address: " & messageBufferUri.AbsoluteUri)
Console.WriteLine()

' Message Buffer Policy
Dim messageBufferPolicy As New MessageBufferPolicy()
messageBufferPolicy.MaxMessageCount = 10
messageBufferPolicy.ExpiresAfter = TimeSpan.FromHours(1) ' messages in the message buffer expire after 1 hour
messageBufferPolicy.TransportProtection = TransportProtectionPolicy.None

Dim messageBufferClient As MessageBufferClient =
EnsureMessageBufferExists(endpointBehavior, messageBufferUri, messageBufferPolicy)

...
End Sub
**Note:** When creating a message buffer, the message buffer policy is one of the required parameters. This policy is modeled by the `MessageBufferPolicy` class, which provides the functionality to set the security type used on the message buffer as well as the messages lifespan and what to do when the buffer fills up with messages (`OverflowPolicy`). Currently, the only available action is to reject the incoming message by faulting the message back to the sender.

The following table shows the different settings that you can configure with the `MessageBufferPolicy` class.

<table>
<thead>
<tr>
<th>MessageBufferPolicy Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExpireAfter</td>
<td>This is the lifespan of the message buffer.</td>
</tr>
<tr>
<td>MaxMessageCount</td>
<td>The maximum number of messages that can be in the message buffer before the overflow policy activates.</td>
</tr>
<tr>
<td>OverflowPolicy</td>
<td>What to do when the buffer fills up with messages. Currently, the only available action is to reject the incoming message by faulting the message back to the sender.</td>
</tr>
<tr>
<td>TransportProtection</td>
<td>The transport protection refers to the level of end-to-end security that must be used.</td>
</tr>
<tr>
<td>Authorization</td>
<td>Specifies the type of authorization required for sending and receiving messages on a Buffer. You can choose between “NotRequired”, “Required”, “RequiredToReceive” and “RequiredToSend”.</td>
</tr>
</tbody>
</table>

5. Next, add the logic to peek the pending submitted order from the Service Bus using the `PeekLock` method of the `MessageBufferClient` class, which retrieves the first available message from the message buffer and locks that message in the buffer. In addition, it uses the `ReleaseLock` method to release the message. To do this, paste the following code below the code inserted in the previous step.

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex03 Order Peek Logic - CS)

```csharp
private static void Main()
{
    ...
    // Message Buffer Policy
    ...
```
// Message converter for Order messages
TypedMessageConverter orderConverter =
TypedMessageConverter.Create(typeof(Order), "SubmitOrder");

while (true)
{
    try
    {
        Message message = messageBufferClient.PeekLock();

        if (message != null)
        {
            Console.WriteLine("Pending Order:");
            Order order = orderConverter.FromMessage(message) as Order;
            if (order != null)
            {
                Console.WriteLine(string.Format(CultureInfo.InvariantCulture, 
                    "- {0} (1)", order.Product, order.Quantity));

            }
        }
    }
    catch (TimeoutException)
    {
        Console.WriteLine("No pending orders...");
    }
    catch (FaultException)
    {
    }

    Console.WriteLine("Sleeping...");
    Thread.Sleep(10000);
}
...
}
Dim orderConverter As TypedMessageConverter =
TypedMessageConverter.Create(GetType(Order), "SubmitOrder")

Do
    Try
        Dim message As Message = messageBufferClient.PeekLock()
        If message IsNot Nothing Then
            Console.WriteLine("Pending Order:")
            Dim order As Order = TryCast(orderConverter.FromMessage(message), Order)
            If order IsNot Nothing Then
                Console.WriteLine(String.Format(CultureInfo.InvariantCulture, "- {0} ({1})", order.Product, order.Quantity))
            End If
            messageBufferClient.ReleaseLock(message)
        End If
    Catch e1 As TimeoutException
        Console.WriteLine("No pending orders...")
    Catch e2 As FaultException
        End Try
    End If

    Console.WriteLine("Sleeping...")
    Thread.Sleep(10000)
Loop

End Sub

Note: Notice that the message buffer is stored in a MessageBufferClient instance. This class has functionality to manipulate the message buffer. It contains methods to send and receive messages. In addition, it contains methods to perform a peek/lock read of a message, which is held temporarily by the message buffer, but "locked" so that no one else can retrieve it. Nevertheless, you or other clients can still pull other messages from the message buffer. The message then goes back into the message buffer, where it can be retrieved again. The message buffer releases the lock message if the client does not explicitly instruct it to delete the locked message before the timeout expires. In the OrderPeek project of the solution, the MessageBufferClient class is used to peek/lock and release a message.

The default timeout of a locked message is one minute. If the call does not return in that time, the message is released.

Task 2 – Creating the Order Submitter
1. Add code in the project’s **Main** method to retrieve the existing message buffer. To do this, in **Solution Explorer**, double-click **Program.cs** (for Visual C# projects) or **Module1.vb** (for Visual Basic projects) in the **OrderSubmitter** project and paste the following code (in **bold** text) in method **Main**, as shown below.

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex03 Get Message Buffer - CS)

<table>
<thead>
<tr>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>private static void Main()</strong></td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>// create the URI for the Message Buffer</td>
</tr>
<tr>
<td><strong>Console.WriteLine(“Message Buffer address: “ + messageBufferUri.AbsoluteUri);</strong></td>
</tr>
<tr>
<td>// Get existing Message Buffer</td>
</tr>
<tr>
<td><strong>MessageBufferClient messageBufferClient =</strong></td>
</tr>
<tr>
<td><strong>MessageBufferClient.GetMessageBuffer(endpointBehavior, messageBufferUri);</strong></td>
</tr>
<tr>
<td><strong>Console.WriteLine(“Using existing message buffer”);</strong></td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex03 Get Message Buffer - VB)

<table>
<thead>
<tr>
<th>Visual Basic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sub Main()</strong></td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>' create the URI for the Message Buffer</td>
</tr>
<tr>
<td><strong>Console.WriteLine(“Message Buffer address: “ &amp; messageBufferUri.AbsoluteUri)</strong>*</td>
</tr>
<tr>
<td>' Get existing Message Buffer</td>
</tr>
<tr>
<td><strong>Dim messageBufferClient As MessageBufferClient =</strong></td>
</tr>
<tr>
<td><strong>messageBufferClient.GetMessageBuffer(endpointBehavior, messageBufferUri)</strong></td>
</tr>
<tr>
<td><strong>Console.WriteLine(“Using existing message buffer”)</strong></td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

*End Sub*
**Note:** Notice that the message buffer is stored in a `MessageBufferClient` instance. This class has functionality to manipulate the message buffer. It contains methods to send and receive messages among others.

2. Add the logic to enter orders and submit them to the Service Bus, where they are captured and processed by the **Order Processors** (the processors application will be created later in this exercise). To do this, paste the following code below the code inserted in the previous step.

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex03 Orders Submission - CS)

```csharp
private static void Main()
{
    ...

    Console.WriteLine();
    Console.WriteLine("Enter your order in the form of '[Quantity] [Product]' and press [ENTER]");
    Console.WriteLine("Enter 'exit' and press [ENTER] to exit");

    string input = Console.ReadLine();
    while (input.ToUpperInvariant() != "EXIT")
    {
        var order = ParseOrder(input);
        if (order != null)
        {
            try
            {
                // Message converter for Order messages
                TypedMessageConverter orderConverter = TypedMessageConverter.Create(typeof(Order), "SubmitOrder");
                var message = orderConverter.ToMessage(order);

                messageBufferClient.Send(message);

                Console.WriteLine("Order Message Sent!");
            }
            catch (Exception e)
            {
                Console.WriteLine("Error: " + e.Message);
            }
        }
        else
        {
            Console.WriteLine("Invalid format");
            Console.WriteLine("Please use [Quantity] [Product]. Ex: '200 NAILS'" );
        }
    }
}
```
input = Console.ReadLine();
}
}  

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex03 Orders Submission - VB)

Visual Basic

Sub Main()
...
    Console.WriteLine()
    Console.WriteLine("Enter your order in the form of '[Quantity] [Product]' and press [ENTER]")
    Console.WriteLine("Enter 'exit' and press [ENTER] to exit")

    Dim input As String = Console.ReadLine()
    Do While input.ToUpperInvariant() <> "EXIT"
        Dim order = ParseOrder(input)
        If order IsNot Nothing Then
            Try
                ' Message converter for Order messages
                Dim orderConverter As TypedMessageConverter = TypedMessageConverter.Create(GetType(Order), "SubmitOrder")
                Dim message = orderConverter.ToMessage(order)

                messageBufferClient.Send(message)

                Console.WriteLine("Order Message Sent!")
            Catch e As Exception
                Console.WriteLine("Error: " & e.Message)
            End Try
        Else
            Console.WriteLine("Invalid format")
            Console.WriteLine("Please use [Quantity] [Product]. Ex: '200 NAILS'")
        End If
        input = Console.ReadLine()
    Loop
End Sub

Note: Notice that the preceding code also validates that each order is entered in the correct format: [Quantity] [Product].
Task 3 – Creating the Order Processor

1. Add code to the order processor to get the existing message buffer, if it exists, or create a new one if necessary. To do this, in Solution Explorer, double-click Program.cs (for Visual C# projects) or Module1.vb (for Visual Basic projects) in the OrderProcessor project and paste the following code (in bold text) into method Main, as shown below. This code defines a message buffer Policy and calls the EnsureMessageBufferExists method.

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex03 Message Buffer Policy - CS)

```csharp
private static void Main()
{
    ...

    // create the URI for the Message Buffer
    Console.WriteLine("Message Buffer address: " + messageBufferUri.AbsoluteUri);

    // Message Buffer Policy
    MessageBufferPolicy messageBufferPolicy = new MessageBufferPolicy();
    messageBufferPolicy.MaxMessageCount = 10;
    messageBufferPolicy.ExpiresAfter = TimeSpan.FromHours(1); // messages in the message buffer expire after 1 hour
    messageBufferPolicy.TransportProtection = TransportProtectionPolicy.None;

    MessageBufferClient messageBufferClient = EnsureMessageBufferExists(endpointBehavior, messageBufferUri, ref messageBufferPolicy);
}
```

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex03 Message Buffer Policy - VB)

```visualbasic
Sub Main()
    ...
    ' create the URI for the message buffer
    Dim messageBufferUri As Uri = ServiceBusEnvironment.CreateServiceUri("https", serviceNamespaceDomain, "OrderMessageBuffer")
    Console.WriteLine("Message Buffer address: " & messageBufferUri.AbsoluteUri)"
    Console.WriteLine()

    ' Message Buffer Policy
```
2. Create a message converter to convert from the untyped message to an Order-type message. To do this, paste the following code (in **bold** text) below the corresponding comment.

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex03 Message Converter - CS)

```csharp
private static void Main()
{
    ...
    // Message converter for order messages
    TypedMessageConverter orderConverter = TypedMessageConverter.Create(typeof(Order), "SubmitOrder");
}
```

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex03 Message Converter - VB)

```visualbasic
Sub Main()
    ...
    ' Message converter for Order messages
    Dim orderConverter As TypedMessageConverter = TypedMessageConverter.Create(GetType(Order), "SubmitOrder")
End Sub
```

**Note:** The **TypedMessageConverter** class provides the functionality to convert a typed message to an untyped message and vice versa. A typed message is an instance of a type that
is annotated with the `MessageContractAttribute` attribute. An untyped message is one that is an instance of the `Message` class.

3. Add code to retrieve a message from the message buffer and convert it to an `Order` instance using the `TypedMessageConverter` converter created in the previous step. In addition, add code to delete the message buffer using the `DeleteMessageBuffer` method of the `MessageBufferClient` when `delete buffer` is entered in the client console application, and to close the console when `exit` is input. To do this, add the following code (in **bold**) to method `Main`, as shown below.

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex03 Retrieve Message from the Message Buffer code - CS)

```csharp
private static void Main()
{
...
    TypedMessageConverter orderConverter =
    TypedMessageConverter.Create(typeof(Order), "SubmitOrder");

    Console.WriteLine("Type at any time 'delete buffer' to destroy the Message Buffer or 'exit' to close the console.");
    string command;
    bool running = true;
    do
    {
        ThreadPool.QueueUserWorkItem(state =>
        {
            while (running)
            {
                try
                {
                    // Until a new message is retrieved, the Retrieve method won't return (unless the default timeout expires)
                    Message message = messageBufferClient.Retrieve();

                    // Convert buffer message to Order type
                    Order order = orderConverter.FromMessage(message) as Order;

                    if (order != null)
                    {
                        ProcessOrder(order);
                    }
                }
                catch (TimeoutException)
```
{  
    Console.WriteLine("No pending orders to be processed.");
    Thread.Sleep(100000);
}

try
{

}  
catch (FaultException)
{
}
}
}

command = Console.ReadLine();

} while ((command.ToLowerInvariant() != "delete buffer") &&
    (command.ToLowerInvariant() != "exit");

running = false;

if (command.ToLowerInvariant() == "delete buffer")
{
    messageBufferClient.DeleteMessageBuffer();
    Console.WriteLine("Message Buffer Deleted");
    Console.WriteLine("Press any key to exit");
    Console.ReadLine();
}

(Code Snippet - Introduction to Service Bus Lab Part 2 - Ex03 Retrieve Message from the Message Buffer code - VB)

Visual Basic

Sub Main()
...
    ' Message converter for Order messages
    Dim orderConverter As TypedMessageConverter = 
        TypedMessageConverter.Create(GetType(Order), "SubmitOrder")

    Console.WriteLine("Type at any time 'delete buffer' to destroy the Message Buffer or 'exit' to close the console.")
    Dim command As String
    Dim running As Boolean = True
    Do
        ' Until a new message is retrieved, the Retrieve method won't return
        ' (unless the default timeout expires)
        ' Convert buffer message to Order type
        ThreadPool.QueueUserWorkItem(Sub(state)
            Do While running
                Try
Dim message As Message = messageBufferClient.Retrieve()

Dim order As Order = TryCast(orderConverter.FromMessage(message), Order)

If order IsNot Nothing Then
    ProcessOrder(order)
End If

Catch e1 As TimeoutException
    Console.WriteLine("No pending orders to be processed.")
End Try

Thread.Sleep(100000)

Catch e2 As FaultException
End Try

End Sub

command = Console.ReadLine()

Loop While (command.ToLowerInvariant() <> "delete buffer") AndAlso (command.ToLowerInvariant() <> "exit")

running = False

If command.ToLowerInvariant() = "delete buffer" Then
    messageBufferClient.DeleteMessageBuffer()
    Console.WriteLine("Message Buffer Deleted")
    Console.WriteLine("Press any key to exit")
    Console.ReadLine()
End If

End Sub

**Note:** In this sample, the actual order processing is not carried out; instead, the **ProcessOrder** method invokes **Thread.Sleep** to simulate the processing of the order.

**Verification**

In order to verify that you have correctly performed every step of this exercise, proceed as follows.

1. Launch the order peeker application. In **Solution Explorer**, right-click the **OrderPeek** project, point to **Debug** and select **Start new instance**. When prompted, provide the **Service Namespace Domain** you set up during the provisioning process at the developer portal. After that, enter the **Default Issuer Name** and **Default Issuer Key** that was generated for the service namespace at the developer portal. At this point, the console should indicate that a new message buffer has been created.
The Order Peek console creates a message buffer and starts listening for messages.

**Note:** The Order Peek console will look for the message buffer and, if not found, it will create a new one.

2. Run the order submitter application. To do this, right-click the **OrderSubmitter** project, point to **Debug** and select **Start new instance**. Provide the **Service Namespace Domain** you set up during the provisioning process, and then the **Default Issuer Name** and **Default Issuer Key**. At this point, the **ORDER SUBMITTER** console should indicate that it will use the existing message buffer — the one created by the **Order Peek** console — and will prompt you to enter orders.

3. Generate a few random orders in the console; for example:
   - 200 nails [enter]
   - 1 hammer [enter]
   - 1 wrench [enter]
Figure 23

Submitting orders to the application

**Note:** The OrderPeek console should show the first submitted order item. It will continuously show the orders that have not been processed.

4. Notice how the Peek console is showing the first message sent by the OrderSubmitter console (the next message to be processed), which in this case is the “200 nails” item.
5. Open an instance of the **Order Processor** application and provide the **Service Namespace Domain**, **Default Issuer Name** and **Default Issuer Key**, as you did in the previous steps. The console should show the processing of the three pending orders as shown in the following figure.

---

**Figure 24**
The Order Peek application showing the next message to be processed.

**Figure 25**
The Order Processor processing pending orders
6. Open one or more additional instances of the **OrderProcessing** console and start sending new orders from the **OrderSubmitter**.

   **Note:** The processing of the new orders should balance between the **OrderProcessor** consoles. The more **OrderProcessor** instances, the less time that orders remain in the message buffer.

7. Close the **OrderPeeker** console. Type “exit” and press ENTER to close each of the running **OrderProcessor** consoles, except for the last one, where you should use “delete buffer” instead to delete the message buffer and exit the console. In the **OrderSubmitter** console type “exit” and press ENTER to close the console and end this verification.

---

**Summary**

By completing this hands-on lab, you’ve learnt how to use different types of bindings and connections modes to establish communications between clients and services: **netTcpRelayBinding**, **Hybrid and Relayed modes**, **ws2007HttpRelayBinding**. In addition, you’ve how to expose a Metadata Exchange (MEX) endpoint through the Service Bus, and to make a REST Service expose binary data through the Service Bus. Finally, you’ve learnt to manipulate a message buffer in AppFabric based on an order-processing scenario.