Windows Azure Service Bus Reference

Seth Manheim and Ralph Squillace

Reference

Microsoft

Windows Azure Service Bus Reference

Seth Manheim and Ralph Squillace

Summary: The Windows Azure Service Bus provides a hosted, secure, and widely available infrastructure for widespread communication, large-scale event distribution, naming, and service publishing. The Service Bus provides connectivity options for Windows Communication Foundation (WCF) and other service endpoints – including REST endpoints -- that would otherwise be difficult or impossible to reach. Endpoints can be located behind network address translation (NAT) boundaries, or bound to frequently-changing, dynamically-assigned IP addresses, or both.

Category: Reference
Applies to: Windows Azure Service Bus
Source: MSDN Library (link to source content)
E-book publication date: May 2012
260 pages

Microsoft

Contents

Service Bus

The Windows Azure Service Bus provides a hosted, secure, and widely available infrastructure for widespread communication, large-scale event distribution, naming, and service publishing. The Service Bus provides connectivity options for Windows Communication Foundation (WCF) and other service endpoints – including REST endpoints -- that would otherwise be difficult or impossible to reach. Endpoints can be located behind network address translation (NAT) boundaries, or bound to frequently-changing, dynamically-assigned IP addresses, or both.

The Service Bus provides both “relayed” and “brokered” messaging capabilities. In the relayed messaging pattern, the relay service supports direct one-way messaging, request/response messaging, and peer-to-peer messaging. Brokered messaging provides durable, asynchronous messaging components such as *Queues*, *Topics*, and *Subscriptions*, with features that support publish-subscribe and temporal decoupling: senders and receivers do not have to be online at the same time; the messaging infrastructure reliably stores messages until the receiving party is ready to receive them.

Note

To use the Service Bus features, install the Windows Azure SDK from the SDK download page.

Use the following links to learn more about the Service Bus. These links come from various content providers across Microsoft. This page will be updated periodically when new content is available, so check back often to see what’s new.

What’s New

- Now Available: The Service Bus September 2011 Release (blog post)
- Windows Azure Queues and Windows Azure Service Bus Queues - Compared and Contrasted in Windows Azure Developer Guidance
- Windows Azure Libraries for Java Available, including support for Service Bus (Jan 9 2012 blog post)
- Creating Applications that Use Service Bus Queues in Windows Azure Developer Guidance
- Creating Applications that Use Service Bus Topics and Subscriptions in Windows

Blogs

- Windows Azure Team Blog
- Windows Azure Customer Advisory Team

Relayed Messaging

- How to Use the Service Bus Relay Service
- An Introduction to Service Bus Relay (video)
- How to use Service Bus Relay (video and code sample)
- Service Bus Relay Load Balancing–The Missing Feature (blog post)

Queues

- How to Use Service Bus Queues
- Creating Applications that Use Service Bus

Azure Developer Guidance

- Managing and Testing Topics, Queues and Relay Services with the Service Bus Explorer Tool (Nov 11 2011 blog post)
- Relay Load Balancing (October 31 2011 blog post)
- Deadlettering in Service Bus Brokered Messaging (October 21 2011 blog post)
- New Service Bus Samples in CodePlex (Oct 16 2011)
- Best Practices for Performance Improvements Using Service Bus Brokered Messaging
- How to integrate a BizTalk Server application with Service Bus Queues and Topics (Oct 12 2011 blog post)

Basics

- Service Bus Overview
- An Introduction to the Service Bus (video)
- Getting Started with the Service Bus
- System and Developer Requirements
- Building loosely-coupled apps with Windows Azure Service Bus Topics and Queues (video)
- Windows Azure Service Bus Brokered Messaging (blog post)

Tools

- Service Bus Explorer (article)
- Service Bus Explorer (code)
- RelayConfigurationInstaller.exe Tool

Forums

- Connectivity and Messaging - Windows Azure Platform

Code Samples

- Service Bus Samples (CodePlex)
- Windows Azure Inter-Role Communication Using Service Bus Topics & Subscriptions (MSDN)
- Hybrid Reference Implementation Using BizTalk Server, Windows Azure, Service Bus &Windows Azure SQL Database

Queues in Windows Azure Developer Guidance.

- An Introduction to Service Bus Queues (video)
- How to use Service Bus Queues (video and code sample)
- Sending large messages to Service Bus session-enabled queues (blog post)
- Using Service Bus Queues with WCF (blog post)

Publish-Subscribe with Topics

- How to Use Service Bus Topics/Subscriptions
- Creating Applications that Use Service Bus Topics and Subscriptions in Windows Azure Developer Guidance.
- An Introduction to Service Bus Topics (video)
- How to use Service Bus Topics (video and code sample)
- Using Service Bus Topics and Subscriptions with WCF (blog post)

Patterns and Best Practices

- Service Bus Queues and Topics Advanced (video)
- Securing Service Bus with ACS (video)
- Best Practices for Leveraging Windows Azure Service Bus Brokered Messaging API (Windows Azure CAT team blog post)
- Achieving Transactional Behavior with Messaging (blog post)
- Windows Azure Service Bus & Windows Azure Connect: Compared & Contrasted (Windows Azure CAT team blog post)
- How to Simplify & Scale Inter-Role Communication Using Windows Azure Service Bus (Windows Azure CAT team blog post)

In this section

About the Windows Azure Service Bus

Provides a conceptual introduction to the Service Bus.

Release Notes for the Service Bus November 2011 Release

Contains important late-breaking information about the Service Bus.

Service Bus Feedback and Community Information

Contains links to resources for community information and ways to provide feedback.

Service Bus and Pricing FAQ

Contains a list of frequently-asked questions about the Service Bus and pricing model.

System and Developer Requirements

Describes the requirements that you must have in order to build and run a connected application that communicates using the Service Bus.

Managing Service Bus Service Namespaces

Describes how to create and manage Service Bus service namespaces.

Windows Azure Service Bus Quotas

Describes quotas allowed in the Service Bus.

Getting Started with the Service Bus

Contains a set of tutorials that demonstrate using a SOAP-based WCF service as well as a REST WCF service to build a service that is registered by using the Service Bus and a client application that invokes the service that uses the Service Bus. A corresponding set of tutorials demonstrate the new Service Bus “brokered” messaging features using both the .NET managed and the REST API.

Developing Applications that Use the Service Bus

Describes the complete Service Bus development cycle. This includes design, implementation, hosting, and configuration of the service; the management of the Service Bus service namespaces, endpoints, and security claims; and the development

of SOAP and REST-based clients. It also contains an overview of the new Service Bus "brokered" messaging features, including Queues, Topics, and Subscriptions. This overview discusses the differences between the original "relayed" and the new brokered messaging patterns.

Troubleshooting

Describes common problems building applications that use the Service Bus and solutions that may address those situations.

RelayConfigurationInstaller.exe Tool

Describes the usage of this tool, which adds the necessary configuration information to the machine or application configuration file.

Best Practices for Performance Improvements Using Service Bus Brokered Messaging

Describes how to use the Service Bus to optimize performance when exchanging brokered messages.

Windows Azure Service Bus REST API Reference

A listing of the Service Bus API available over the REST protocol.

Appendix: Messaging Exceptions

A list of messaging exception types and their causes, and suggested actions you can take.

About the Windows Azure Service Bus

Note

The Service Bus components are now included with the "Windows Azure Libraries for .NET." To install, visit the Windows Azure SDK download page.

The Windows Azure Libraries for .NET consist of three services that provide important capabilities for your Windows Azure applications:

- The Service Bus securely relays messages to and from any Web service regardless of the device or computer on which they are hosted, or whether that device is behind a firewall or

NAT router. It provides direct one-way or request/response (relayed) messaging as well as brokered, or asynchronous, messaging patterns.

- The **Access Control Service** is an interoperable, claims-based service that provides federated authentication and authorization solutions for any resource, whether in the cloud, behind a firewall, or on a smart device.
- The **Windows Azure Caching Service** provides a distributed, in-memory cache that helps Windows Azure applications to achieve increased performance and scalability.

The Service Bus and Access Control together make hybrid, connected applications—applications that communicate from behind firewalls, across the Internet, from hosted cloud servers, between rich desktops and smart devices—easier to build, secure, and manage. Although you can build hybrid, connected applications today, doing this often means you have to build important infrastructure components before you can build the applications themselves. The Service Bus and Access Control provide several important infrastructure elements so that you can more easily begin making your connected applications work now.

Software, Services, Clouds, and Devices

Today's business infrastructure is more feature-rich, connected, and interoperable than ever. People access applications and data through stunning graphical programs running on traditional operating systems; powerful Web applications that are running in browsers; and very small, intelligent computers such as cell phones, netbooks, and other smart devices. Applications run locally, often on powerful servers and server farms, and critical data is stored in performant databases, on desktops, and in the cloud.

The Internet connects people, applications, devices, and data across the world. Clouds of computing power—such as Windows Azure—can help us reduce costs and increase scalability and manageability. Web services can expose functionality to any caller safely and securely.

With these technologies, platforms, and devices, you can build significantly distributed, interactive applications that can reach almost anyone, use almost any useful data, and do both securely and robustly regardless of where the user is at the moment. Such hybrid, connected programs – including those often referred to as "Software plus Services" -- could use proprietary or private data behind a firewall and return only the appropriate results to a calling device, or notify that device when a particular event occurs.

Fulfilling the Potential

However, building these distributed applications currently is very, very hard—and it should not be. There are many reasons why, without a platform that solves these problems for you, it remains difficult to take advantage of these wonderful technologies that could make a business more efficient, more productive, and your customers happier.

- Operating systems are still located—trapped is often a better word—on a local computer, typically behind a firewall and perhaps network address translation (NAT) of some sort. This problem is true of smart devices and phones, too.

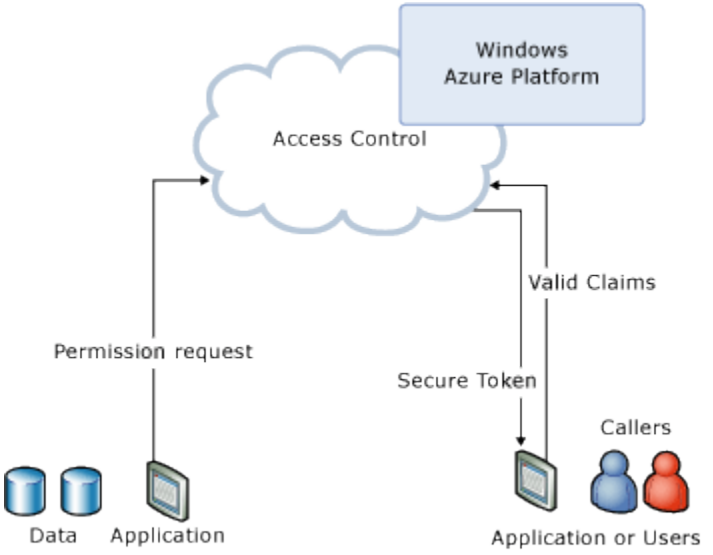
- As ubiquitous as Web browsers are, their reach into data is limited to an interactive exchange in a format they understand.
- Heterogeneous platforms, such as server applications, desktop or portable computers, smart devices, and advanced cell phones, often interoperate at a rudimentary level, if at all. They can rarely share the same code base or benefit from feature or component reuse.
- Much of the most valuable data is stored in servers and embedded in applications that will not be replaced immediately—sometimes referred to as "legacy" systems. The data in these systems are trapped by technical limitations, security concerns, or privacy restrictions.
- The Internet is not always the network being used. Private networks are an important part of the application environment, and their insulation from the Internet is a simple fact of information technology (IT) life.

Service Bus and Access Control are built to overcome these kinds of obstacles; they provide the "fabric" that you can use to build, deploy, and manage the distributed applications that can help make the promise of "Software + Services" become real. The Service Bus and Access Control services together are highly-scalable services that are running in Microsoft data centers that can be used by applications anywhere to securely bridge the gap between local applications behind a firewall, applications that are running in the cloud, and client applications of any kind around the world. Another way of saying this is that the Service Bus and Access Control are the glue that makes "Software" and "Services" work together.

Feature Overview

The Service Bus connects local, firewalled applications and data with applications in the cloud, rich desktop applications, and smart, Web-enabled devices anywhere in the world.

Access Control Service is a claims-based access control service that can be used on most Web-enabled devices to build interoperable, federated authentication and authorization into any connected application. The following diagram illustrates this architecture.

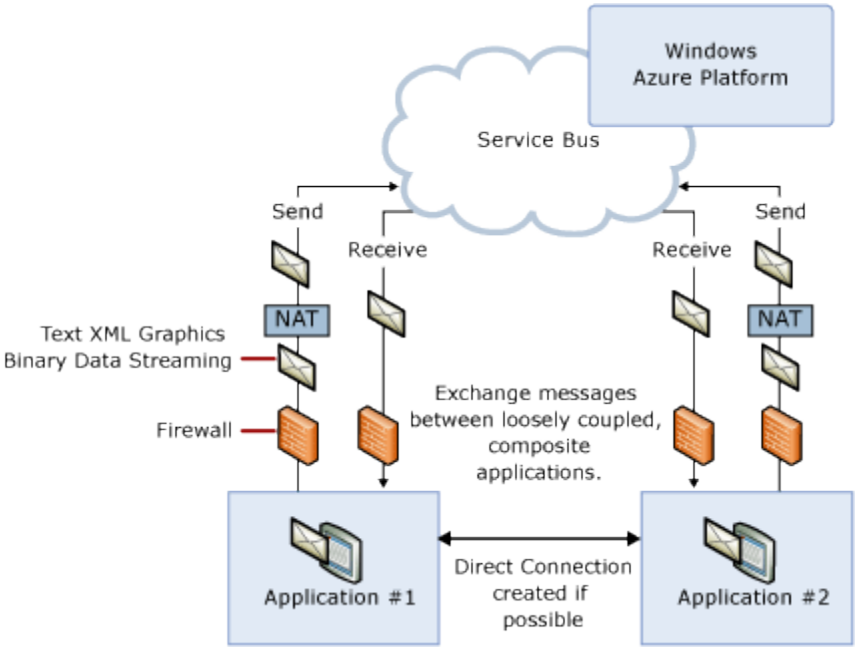


The **Windows Azure Caching Service** provides a distributed, in-memory cache that helps Windows Azure applications to achieve increased performance and scalability.

Service Bus Features

- Securely exposes to external callers Windows Communication Foundation (WCF)-based Web services that are running behind firewalls and NAT routers -- without requiring you to open any inbound ports or otherwise change firewall and router configurations.
- Enables secure inbound communication from devices outside the firewall.
- Provides a global namespace system that is location-independent: the name of a service in the Service Bus provides no information about the final destination of the communication.
- Provides a service registry for publishing and discovering service endpoint references in a service namespace.
- Provides relayed messaging capabilities: the relay service supports direct one-way messaging, request/response messaging, and peer-to-peer messaging.
- Provides brokered (or asynchronous) messaging capabilities: Senders and receivers do not have to be online at the same time. The messaging infrastructure reliably stores messages until the receiving party is ready to receive them. The core components of the brokered messaging infrastructure are Queues, Topics, and Subscriptions.
- Builds and hosts service endpoints that support:
    - Exposing a Web service to remote users. Expose and secure a local Web service in the cloud without managing any firewall or NAT settings.
    - Eventing behavior. Listen for notifications on any device, anywhere in the world.
    - Tunneling between any two endpoints to enable bidirectional streams.

The following diagram illustrates the capabilities of the Service Bus.



Access Control Features

The Access Control service provides claims-based authentication and authorization for REST Web services.

- Usable from any platform.
- Low friction way to onboard new clients.
- Integrates with AD FS v2.
- Implements OAuth Web Resource Authorization Protocol (WRAP)and Simple Web Tokens (SWT).
- Enables simple delegation.
- Trusted as an identity provider by the Service Bus.
- Extensible to provide integration with any identity provider.

Caching Features

Windows Azure Caching enables applications to cache activity and reference data for .NET applications running in Windows Azure. This includes the following features.

- Usable from any .NET application hosted in Windows Azure.
- Provides automatic management of the caching infrastructure in the cloud.
- Provides ASP.NET session state and output caching providers.
- Configurable through application configuration files or programmatically.
- Provides a API centered around cache access with key-value pairs.
- Usable with optimistic or pessimistic concurrency.
- Supports the local cache feature for additional performance and scalability benefits.

Release Notes for the Service Bus November 2011 Release

Note
The Service Bus components are now included with the "Windows Azure Libraries for .NET." To install, visit the Windows Azure SDK download page.

The release notes for the Windows Azure Service Bus November 2011 release contain the following topics:

1. Prerequisites
2. What's New
3. Changes
4. Known Issues

These release notes will be updated periodically. For the latest update, please click here.

See the Service Bus and Pricing FAQ topic for a list of Service Bus FAQs.

Prerequisites

Account Requirements

Before running Windows Azure Service Bus applications, you must create one or more service namespaces. To create and manage your service namespaces, log on to the Windows Azure Management portal, click **Service Bus, Access Control & Caching**, then click **Service Bus**. In the left-hand tree, click the service namespace you want to manage. For more information, see Managing Service Bus Service Namespaces.

SDK Samples

Note
By default, the Service Bus samples are no longer installed with the SDK. To obtain the samples, visit the Windows Azure SDK samples page.

Most SDK samples and applications contain three authentication requirements:

1. Service Namespace: You can use the service namespace you created in your project on the portal. The service namespace is used to construct Service Bus endpoints (for example, `sb://<domain>.servicebus.windows.net/Echo/`)
2. Issuer Name: You can use **owner**, which is an issuer that is created by default for you.
3. Issuer Key/Secret: You can use the **Default Issuer Key** option listed on the **Service Namespace** management page on the portal.

Runtime Requirements

The November 2011 release of the Service Bus includes reliable, or “brokered’ messaging enhancements and is commercially available today. This release is fully backward compatible and you can continue to use your existing applications. Windows Azure Access Control Service (ACS) integration has been updated from ACS v1 to ACS v2. You can obtain updated client assemblies for these features by installing the Windows Azure SDK version 1.6.

The .NET Framework managed APIs for accessing all Service Bus functionality including the existing relayed messaging and new brokered messaging features are included in a single assembly: Microsoft.ServiceBus.dll (version 1.6.0.0). This assembly targets the .NET Framework version 4 and is backward compatible with both Microsoft.ServiceBus.dll v1.0.0.0 and v1.5.0.0.

Note

The Service Bus assemblies are no longer installed in the .NET Global Assembly Cache (GAC) by default. The updated default locations for these assemblies are:

- Service Bus assemblies: C:\Program Files\Windows Azure SDK\v1.6\ServiceBus\ref.
- Service Bus tools (RelayConfigurationInstaller.exe): C:\Program Files\Windows Azure SDK\v1.6\ ServiceBus\bin.

Any existing applications that target the .NET Framework 3.5 can continue to use Microsoft.ServiceBus.dll (version 1.0.0.0), which includes the previously shipped relayed messaging APIs and QFE fixes.

There are three versions of the SDK now available:

- Windows Azure SDK version 1.6: Includes Microsoft.ServiceBus.dll version 1.6.0.0, which is an updated version of Microsoft.ServiceBus.dll version 1.5.0.0 that includes all the same brokered and relayed messaging features. Previously these assemblies shipped as part of the Windows Azure AppFabric SDK.
- Windows Azure AppFabric SDK version 1.5: Includes Microsoft.ServiceBus.dll version 1.5.0.0, which first introduced the new Service Bus *Topics* and *Queues* brokered messaging features (and includes the existing relayed messaging features), and updated Caching client assemblies. This SDK is targeted at the .NET Framework 4.
- Windows Azure AppFabric SDK version 1.0: Includes Microsoft.ServiceBus.dll version 1.0.0.0, which contains only the existing relayed messaging features and targets the .NET Framework 3.5.

Note

Installing the current Windows Azure 1.6 SDK does not uninstall the 1.0 Windows Azure SDK. You must uninstall it manually.

The Windows Azure SDK samples are available as Visual Studio 2010 solutions (C# and Visual Basic), and require either Microsoft .NET Framework 3.5 SP1 or .NET Framework 4 to run. The SDK is supported on the Windows Server 2008, Windows°7, Windows Vista, Windows Server 2003, and Windows XP operating systems. Additionally, some of the samples require the Windows Azure SDK version 1.6 and Windows Azure tools for Visual Studio version 1.6.

Windows Identity Foundation (WIF) is required for Active Directory Federation Services V2 (ADFS) integration. For WIF system requirements, click here.

Note
If a sample has both a service and a client application, please use the credentials from the same service namespace in the both the service and the client.

What's New

- **Relay load balancing**: when using the **Microsoft.ServiceBus.NetTcpRelayBinding**, **Microsoft.ServiceBus.RelayedOnewayTransportBindingElement**, **Microsoft.ServiceBus.NetOnewayRelayBinding**, **Microsoft.ServiceBus.TcpRelayTransportBindingElement** bindings, or any of the HTTP bindings (**Microsoft.ServiceBus.WebHttpRelayBinding**, and so on), if you previously opened more than one listener on a particular relay endpoint, you would receive an **System.ServiceModel.AddressAlreadyInUseException** exception. For example:

```
Uri address = ServiceBusEnvironment.CreateServiceUri("sb",
serviceNamespace, "MyService");


ServiceHost host = new ServiceHost(typeof(MyService), address);
host.Open();
```

  This behavior has now changed, and you can open multiple listeners (up to 25) on the same endpoint. When the Service Bus receives a request for your endpoints, the system load balances which of the connected listeners receives the request or connection/session. Note that if you have multiple listeners on the same endpoint that are mismatched (for example, if one listener has enabled Access Control and another has not), you will still receive the **System.ServiceModel.AddressAlreadyInUseException** exception.

  Please note that this new behavior can potentially change the behavior of existing applications. If your application relies on the existence of one listener on an endpoint, you should make sure that your code limits the number of listeners per endpoint to one. To take advantage of the new relay load balancing feature, your code should be updated to manage the number of concurrent listeners per endpoint and provide limits to your desired number of listeners per endpoint.

  There are no explicit guarantees about order or fairness across the load balanced endpoints. The service makes a best-effort attempt at fairness by choosing listeners at random using a strategy that ensures reasonably good distribution.

  Note
  If you exceed the limit of 25 concurrent listeners, you will receive a **System.ServiceModel.QuotaExceededException** exception.
- Support for ports in messaging operations: you must now explicitly set the **Microsoft.ServiceBus.ConnectivityMode** enumeration to **Microsoft.ServiceBus.ConnectivityMode.Http**, in order to get messaging operations to use

port 80/443. Using the **Microsoft.ServiceBus.ConnectivityMode.AutoDetect** mode does not achieve this result.

Changes

This section lists important changes in the Service Bus November 2011 release.

- There is no longer a separate SDK installer for the Windows Azure SDK. The Service Bus and cache components are now included with the "Windows Azure Libraries for .NET." To install, visit the Windows Azure SDK download page.
- The Service Bus and cache assemblies are no longer installed in the .NET Global Assembly Cache (GAC) by default. The updated default locations for these assemblies are:
  - cache assemblies: C:\Program Files\Windows Azure SDK\v1.6\Cache\ref.
  - Service Bus assemblies: C:\Program Files\Windows Azure SDK\v1.6\ServiceBus\ref.
  - Service Bus tools (RelayConfigurationInstaller.exe): C:\Program Files\Windows Azure SDK\v1.6\ ServiceBus\bin.
- The current Message Buffers feature, including their management protocol, will remain supported for backwards compatibility. However, the general recommendation is that you explicitly change client code to use the new Service Bus *Queues* feature. For more information, see Queues, Topics, and Subscriptions.
- The Service Bus no longer adds entries to the Machine.config file. When running code developed with previous versions of the Windows Azure SDK, you may see errors such as the following:

```
Configuration binding extension
'system.serviceModel/bindings/netTcpRelayBinding' could not be found. Verify
that this binding extension is properly registered in
system.serviceModel/extensions/bindingExtensions and that it is spelled
correctly.
```

  It is recommended that you add these extensions to the App.config files for your projects or use the Relayconfiginstaller.exe tool in the SDK to add these bindings. For example:

```
<configuration>
<system.serviceModel>
<extensions>
<!-- Adding all known service bus extensions. You can remove the
ones you don't need. -->
<behaviorExtensions>
<add name="connectionStatusBehavior"
type="Microsoft.ServiceBus.Configuration.ConnectionStatusElement
, Microsoft.ServiceBus, Version=1.6.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
```

```
<add name="transportClientEndpointBehavior"
type="Microsoft.ServiceBus.Configuration.TransportClientEndpoint
BehaviorElement, Microsoft.ServiceBus, Version=1.6.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
<add name="serviceRegistrySettings"
type="Microsoft.ServiceBus.Configuration.ServiceRegistrySettings
Element, Microsoft.ServiceBus, Version=1.6.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
</behaviorExtensions>
<bindingElementExtensions>
<add name="netMessagingTransport"
type="Microsoft.ServiceBus.Messaging.Configuration.NetMessagingT
ransportExtensionElement, Microsoft.ServiceBus, Version=1.6.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
<add name="tcpRelayTransport"
type="Microsoft.ServiceBus.Configuration.TcpRelayTransportElemen
t, Microsoft.ServiceBus, Version=1.6.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
<add name="httpRelayTransport"
type="Microsoft.ServiceBus.Configuration.HttpRelayTransportEleme
nt, Microsoft.ServiceBus, Version=1.6.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
<add name="httpsRelayTransport"
type="Microsoft.ServiceBus.Configuration.HttpsRelayTransportElem
ent, Microsoft.ServiceBus, Version=1.6.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
<add name="onewayRelayTransport"
type="Microsoft.ServiceBus.Configuration.RelayedOnewayTransportE
lement, Microsoft.ServiceBus, Version=1.6.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
</bindingElementExtensions>
<bindingExtensions>
<add name="basicHttpRelayBinding"
type="Microsoft.ServiceBus.Configuration.BasicHttpRelayBindingCo
llectionElement, Microsoft.ServiceBus, Version=1.6.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
<add name="webHttpRelayBinding"
type="Microsoft.ServiceBus.Configuration.WebHttpRelayBindingColl
```

```
ectionElement, Microsoft.ServiceBus, Version=1.6.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
<add name="ws2007HttpRelayBinding"
type="Microsoft.ServiceBus.Configuration.WS2007HttpRelayBindingC
ollectionElement, Microsoft.ServiceBus, Version=1.6.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
<add name="netTcpRelayBinding"
type="Microsoft.ServiceBus.Configuration.NetTcpRelayBindingColle
ctionElement, Microsoft.ServiceBus, Version=1.6.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
<add name="netOnewayRelayBinding"
type="Microsoft.ServiceBus.Configuration.NetOnewayRelayBindingCo
llectionElement, Microsoft.ServiceBus, Version=1.6.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
<add name="netEventRelayBinding"
type="Microsoft.ServiceBus.Configuration.NetEventRelayBindingCol
lectionElement, Microsoft.ServiceBus, Version=1.6.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
<add name="netMessagingBinding"
type="Microsoft.ServiceBus.Messaging.Configuration.NetMessagingB
indingCollectionElement, Microsoft.ServiceBus, Version=1.6.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
</bindingExtensions>
</extensions>
</system.serviceModel>
</configuration>
```

- If you perform management operations using the Access Control version 1.0 Acm.exe tool, this tool is no longer shipped with the version Windows Azure 1.6 SDK. If you require access to this tool, you will have to roll back to the Windows Azure version 1.0 SDK.

Known Issues

The following section lists known issues in this release of the Service Bus:

- After installing the Windows Azure 1.6 SDK, any application that uses the Machine.config file may encounter the following error:

```
Exception type:
System.Configuration.ConfigurationErrorsException
Message:          Configuration binding extension
'system.serviceModel/bindings/netTcpRelayBinding' could not be
```

```
found. Verify that this binding extension is properly registered
in system.serviceModel/extensions/bindingExtensions and that it
is spelled correctly.
InnerException:   <none>
StackTrace (generated):
    SP       IP       Function
    0F1DE230 5F22EA4C
System_Configuration_ni!System.Configuration.BaseConfigurationRe
cord.EvaluateOne(System.String[],
System.Configuration.SectionInput, Boolean,
System.Configuration.FactoryRecord,
System.Configuration.SectionRecord, System.Object)+0xc8


    0F1DE30C 5F22E86E
System_Configuration_ni!System.Configuration.BaseConfigurationRe
cord.Evaluate(System.Configuration.FactoryRecord,
System.Configuration.SectionRecord, System.Object, Boolean,
Boolean, System.Object ByRef, System.Object ByRef)+0x482
    0F1DE3CC 5F226F8D
System_Configuration_ni!System.Configuration.BaseConfigurationRe
cord.GetSectionRecursive(System.String, Boolean, Boolean,
Boolean, Boolean, System.Object ByRef, System.Object
ByRef)+0x5bd
    0F1DE454 5F226F8D
System_Configuration_ni!System.Configuration.BaseConfigurationRe
cord.GetSectionRecursive(System.String, Boolean, Boolean,
Boolean, Boolean, System.Object ByRef, System.Object
ByRef)+0x5bd
    0F1DE4DC 5F226F8D
System_Configuration_ni!System.Configuration.BaseConfigurationRe
cord.GetSectionRecursive(System.String, Boolean, Boolean,
Boolean, Boolean, System.Object ByRef, System.Object
ByRef)+0x5bd
    0F1DE564 5F226F8D
System_Configuration_ni!System.Configuration.BaseConfigurationRe
cord.GetSectionRecursive(System.String, Boolean, Boolean,
Boolean, Boolean, System.Object ByRef, System.Object
ByRef)+0x5bd
```

```
    0F1DE5EC 5F2269BA
System_Configuration_ni!System.Configuration.BaseConfigurationRe
cord.GetSection(System.String)+0x2a
    0F1DE5FC 5F22A7D5
System_Configuration_ni!System.Configuration.ClientConfiguration
System.System.Configuration.Internal.IInternalConfigSystem.GetSe
ction(System.String)+0x55
    0F1DE610 5F2210EF
System_Configuration_ni!System.Configuration.ConfigurationManage
r.GetSection(System.String)+0x4f
```

This is caused by a known issue in the upgrade from the Windows Azure 1.0 SDK to the Windows Azure 1.6 SDK. The following workaround resolves the issue:

a. Uninstall the Windows Azure SDK version 1.6.

b. Install the Windows Azure SDK version 1.0 from http://go.microsoft.com/fwlink/?LinkID=228910. On the **Add/Remove Programs** Control Panel applet, this appears as version 1.0.1471.

c. Reinstall the Windows Azure SDK version 1.6 from http://go.microsoft.com/fwlink/?LinkID=226941.

This will remove all Service Bus bindings from the Machine.config file. For more information, see the second Service Bus bullet in the “Changes” section of these release notes.

- It is currently not possible to use the Windows Communication Foundation (WCF) Service Model Metadata Tool (Svcutil.exe) to generate a proxy from a service that uses **Microsoft.ServiceBus**.
- **Microsoft.ServiceBus.BasicHttpRelayBinding** limitation: The size of each individual attribute value cannot exceed 16k. If it exceeds this threshold, an exception is generated indicating that the **MaxNameTableCharCount** has been exceeded. The **MaxNameTableCharCount** value of the **ReaderQuotas** property is set to the default value of 16K, and this is the value which indicates the maximum size of each of the attribute values.
- Message Buffer characters: If you use certain non-alphanumeric characters in the name of a message buffer, such as a question mark (encoded as %3F), or an equal sign (encoded as %3D), you may experience an authorization failure (**RelayedHttpServiceAuthorizationFailure**) and be unable to send messages to the buffer. Be sure to use alphanumeric characters in your message buffer names.
- ProtocolException when closing a service while it is receiving multicast messages: When you end a multicast service while it is still receiving messages, you may receive a **System.ServiceModel.ProtocolException** with the message “Unhandled Exception: System.ServiceModel.ProtocolException: The channel received an unexpected input message …” The error will not interfere with the closing of the channel and can be safely ignored.
- Listener recovery delay: Under rare hardware failure conditions, listeners constructed using the **Microsoft.ServiceBus.NetOnewayRelayBinding** binding can require up to five minutes

to recover. During this period, an attempt to re-create a failed listener may be unsuccessful and generate an AddressAlreadyInUseException message.

- When authenticating with SAML tokens, clients must send a new token to the Service Bus before the SAML token expires. Doing so avoids an interruption in connectivity with the Service Bus. Assign the new token to the same **Microsoft.ServiceBus.TransportClientEndpointBehavior** behavior you used to establish the original connection.
- In a Retrieve/Peeklock operation, a client may generate a **System.ServiceModel.FaultException** ("Message could not be retrieved") instead of a **System.TimeoutException** ("Message could not be retrieved: NoContent, No message available within the specified timeout."). This may occur when the server closes the keepalive connection if no request is received within 1 minute of the first request. To work around this issue, set the retrieve time-out to a value less than 1 minute (for example, 55 seconds).
- Message Buffer expiration: When you create a message buffer, you can specify a time after which the message buffer will expire if there are no requests to receive a message within that time interval. The default is five minutes. When you request a message from the message buffer, you can specify the number of seconds the request will wait for a message to arrive at an empty buffer before the request times out. The default is one minute. When a request to receive a message reaches a buffer, the expiration timer is reset and starts counting down again. Therefore, if you specify a longer receive request time-out than the expiration time on the buffer, and no messages are received, the buffer can actually expire while a receive request is still pending. In that case, the buffer will disappear and the receive request throws an exception. To avoid this behavior, accept the default message buffer settings, or make sure that you specify an expiration interval for the buffer that is longer than the receive time-out interval.
- Because of a bug in Windows Communication Foundation (WCF), when WCF activity tracing is on, a call to **Channel.Open** when you are using the **NetTcpRelayBinding** binding generates a **NullReferenceException**. This bug has been fixed in WCF 4.0.
- ATOM feed always comes back empty when a GET request is made against an HTTP URI unless the Cache-Control max-age=0 header is set. A GET against HTTPS URIs works fine without the need for this header.
- When using the **Ws2007HttpRelayBinding** binding protocol, a time-out can occur during periods of moderate to high system load. Because of this behavior, we recommend that for solutions requiring a high degree of reliability, you use the **WebHttpRelayBinding** binding instead.
- The following operations will not work with Flash clients that are trying to use message buffer in the Service Bus:
    a. PeekLock
    b. Unlock locked message
    c. Delete locked message
- The following are known issues in the Windows Azure portal while managing Service Bus entities:
    - The maximum number of queues or topics that can be displayed per service namespace is 25. The maximum number of displayed subscriptions per topic is 5. They are sorted

alphabetically. It is strongly recommended that you use the Service Bus managed APIs to create or delete queues, topics, or subscriptions if you intend to manage more entities than the current portal limitations.

- The topic length is always set to zero. However, the size of the topic reflects the number of bytes currently occupied by messages still in the queue.

- The default timeout on the **System.Net.ServicePointManager.MaxServicePointIdleTime** property (an Http library that maintains a pool of connections) is 100secs. The timeout for Windows Azure server keepalive timeouts is 60 seconds. Therefore, if you have a connection that is idle for more than 60 seconds, it is disconnected by Windows Azure, and the next time you try to use the connection it returns an error. The workaround is to set the **System.Net.ServicePointManager.MaxServicePointIdleTime** property to a value less than the Windows Azure server keepalive timeout. This way, the connection is removed before Windows Azure disconnects it
- If you are upgrading from the Windows Azure SDK version 1.0, you may see the following behavior:

  a. Warning messages due to obsolete classes:

```
'Microsoft.ServiceBus.TransportClientCredentialType' is obsolete


'Microsoft.ServiceBus.TransportClientEndpointBehavior.CredentialT
ype' is obsolete: '"This property is deprecated. Please use
TransportClientEndpointBehavior.TokenProvider instead."'


'Microsoft.ServiceBus.TransportClientEndpointBehavior.Credentials
' is obsolete: '"This property is deprecated. Please use
TransportClientEndpointBehavior.TokenProvider instead."'


'Microsoft.ServiceBus.TransportClientEndpointBehavior.Credentials
' is obsolete: '"This property is deprecated. Please use
TransportClientEndpointBehavior.TokenProvider instead."'
```

  b. Exceptions when running your application due to missing entries in Machine.config (please add entries to the App.config file, as described in the “Changes” section):

```
System.Configuration.ConfigurationErrorsException was unhandled


Message=Configuration binding extension
'system.serviceModel/bindings/netTcpRelayBinding' could not be
found. Verify that this binding extension is properly registered
in system.serviceModel/extensions/bindingExtensions and that it
is spelled correctly.
```

Quotas

For information about quotas for the Service Bus, see the Windows Azure Platform pricing FAQ.

Service Bus Feedback and Community Information

We appreciate your comments and concerns about the Windows Azure Service Bus. For more information, or to share techniques with others who are interested in the SDK, visit the Connectivity and Messaging - Windows Azure Platform or Windows Azure Platform Development forums.

For technical support, visit the Windows Azure Platform Support page.

Supplementary technical information is available at the TechNet Wiki site. This site is frequently updated, so be sure to check back regularly for new content.

The Windows Azure and Windows Azure SQL Database documentation teams maintain a presence on Twitter. For more information, follow us at http://twitter.com/MsftCloudUETeam.

Service Bus and Pricing FAQ

If you have questions about the Windows Azure Service Bus pricing structure, see the FAQ in the following section. You can also visit the Windows Azure Platform pricing FAQ for general Windows Azure pricing information.

Service Bus FAQ

- What is the Windows Azure Service Bus?
- What are typical usage scenarios for the Service Bus?
- What are the main capabilities and benefits of the Service Bus?
- How do you currently charge for the Service Bus?
- How will you charge for the Service Bus once the promotional period ends?
- What usage of the Service Bus is subject to data transfer? What is not?
- What exactly is a Service Bus “relay”?
- How is the Relay Hours meter calculated?
- What if I have more than one listener connected to a given relay?
- What happened to the “Service Bus Connections” billing meter?
- How is the Messages meter calculated for queues, topics/subscriptions, and message buffers?
- How is the Messages meter calculated for relays?
- Are management operations and control messages counted as billable Messages?

What is the Windows Azure Service Bus?

The Windows Azure Service Bus provides secure messaging and relay capabilities that enable building distributed and loosely-coupled applications in the cloud. The Windows Azure Service Bus also enables developing hybrid applications that span private clouds, public clouds and clients running on PCs, mobile devices, or in the browser. It supports multiple messaging protocols and patterns and handles delivery assurance, reliable messaging and scale for your applications. The Service Bus is a managed service that is operated by Microsoft and has a 99.9% monthly SLA.

What are typical usage scenarios for the Service Bus?

Some common applications of the Service Bus include:

- **Hybrid applications:** Enables you to securely connect and integrate enterprise systems running in your private cloud with applications running on Windows Azure. This makes it easier to extend solutions to the cloud without having to port or migrate all of your data or code from an existing enterprise datacenter to Windows Azure.
- **Mobile applications:** Enables you to easily build applications that can distribute event notifications and data to occasionally connected clients, such as smartphones or tablets. You can expose notifications or events from an application running either in Windows Azure or in your private cloud environment, and ensure that they are ultimately delivered to mobile devices.
- **Loosely coupled architectures:** Enables you to build loosely coupled systems that are more resilient to network failure and can more easily scale out based on demand. The Service Bus can act as the connecting broker between the different components of a system, eliminating direct dependencies between different components. Easily leverage the Service Bus to architect applications that support application load balancing.

What are the main capabilities and benefits of the Service Bus?

Service Bus Messaging

- Service Bus queues offer a reliable, highly scalable way to store messages as they travel between systems without losing messages in the event of connectivity failure.
- Service Bus topics and subscriptions implement a publish/subscribe pattern that delivers a highly scalable, flexible, and cost-effective way to publish messages from an application and deliver them to multiple subscribers.

Service Bus Connectivity

- The Service Bus relay enables applications hosted in Windows Azure to securely call back to private cloud-based applications hosted in your own datacenter behind a firewall, and vice versa. The relay service avoids the need to instantiate and set up a new connection on each call and makes connectivity faster and more reliable. It also supports the ability to integrate applications across existing NATs and firewalls. The relay service supports a variety of different transport protocols and Web services standards, including REST, SOAP, and WS-*.
- Companies can use the Service Bus relay to expose just the information they want from their private cloud environment, which creates an architecture more secure than opening up a VPN. Enterprises can use a SOA-based architecture and expose just the services they want to deliver from their on-premise data centers.

How do you currently charge for the Service Bus?

We are currently not charging for any of the Service Bus capabilities. This promotional period will end for all billing months beginning May 31, 2012.

How will you charge for the Service Bus once the promotional period ends?

At the end of the promotional period, these meters will be billed as follows:

1. **Messages** – Messages sent to or delivered by the Service Bus will be billed at $0.01 per 10,000 messages. Messages are charged based on the number of messages sent to, or delivered by, the Service Bus during the billing month. This includes delivery of “null” messages in response to receive requests against empty queues/subscriptions. Messages over 64KB in size will be charged an additional message for each additional 64KB of data (rounded up). This meter applies to relays as well as queues, topics, subscriptions, and message buffers.
2. **Relay Hours** – This meter applies only when using the Service Bus relay capability. There is no relay hour charge if you are only using Service Bus queues, topics/subscriptions, or message buffers. Relay hours will be billed at $0.10 per 100 relay hours, and charged from the time the relay is opened (the first listener connect on a given relay address) to the close of the relay (the last listener disconnect from that relay address), and rounded up to the next whole hour.

In addition to the prices noted above for the Service Bus, you will be charged for associated data transfers for egress outside of the data center in which your application is provisioned. You can find more details in the What usage of the Service Bus is subject to data transfer? What is not? section below.

What usage of the Service Bus is subject to data transfer? What is not?

Any data transfer within a given Windows Azure sub-region is provided at no charge. Any data transfer outside a sub-region is subject to egress charges at the rate of $0.15 per GB from the North America and Europe regions, and $0.20 per GB from the Asia-Pacific region. Any inbound data transfer is provided at no charge.

What exactly is a Service Bus "relay"?

A relay is a Service Bus entity that relays messages between clients and Web services. The relay provides the service with a persistent, discoverable Service Bus address, reliable connectivity with firewall/NAT traversal capabilities, and additional features such as automatic load balancing. A relay is implicitly instantiated and opened at a given Service Bus address (namespace URL) whenever a relay-enabled WCF service, or "relay listener," first connects to that address. Applications create relay listeners by using the Service Bus .NET managed API, which provides special relay-enabled versions of the standard WCF bindings.

How is the Relay Hours meter calculated?

Relay hours are billed for the cumulative amount of time during which each Service Bus relay is "open" during a given billing period. A relay is implicitly instantiated and opened at a given Service Bus address (service namespace URL) when a relay-enabled WCF service, or "Relay listener," first connects to that address. The relay is closed only when the last listener disconnects from its address. Therefore, for billing purposes a relay is considered "open" from the time the first relay listener connects, to the time the last relay listener disconnects from the Service Bus address of that relay. In other words, a relay is considered "open" whenever one or more relay listeners are connected to its Service Bus address.

What if I have more than one listener connected to a given relay?

In some cases, a single relay in the Service Bus may have multiple connected listeners. This can occur with load-balanced services that use the **netTCPRelay** or ***HttpRelay** WCF bindings, or with broadcast event listeners that use the **netEventRelay** WCF binding. A relay in the Service Bus is considered "open" when at least one relay listener is connected to it. Adding additional listeners to an open relay does not change the status of that relay for billing purposes. The number of relay *senders* (clients that invoke or send messages to relays) connected to a relay also has no effect on the calculation of relay hours.

What happened to the "Service Bus Connections" billing meter?

The Service Bus no longer charges for connections. However, there are quotas limiting the number of simultaneous connections that can be open against any single Service Bus entity. See the Does the Service Bus have any usage quotas? section below.

How is the Messages meter calculated for queues, topics/subscriptions, and message buffers?

Each message sent to or delivered by the Service Bus counts as a billable message. This applies to all Service Bus entity types, including queues, topics/subscriptions, message buffers, and relays.

A message is defined as a unit of data which is 64KB or less in size. In the case of brokered entities (queues, topics/subscriptions, message buffers), any message that is less than or equal to 64KB in size is considered as one billable message. If the message is greater than 64KB in size, the number of billable messages is calculated according to the message size in multiples of 64KB. For example, an 8 KB message sent to the Service Bus will be billed as one message, but a 96 KB message sent to the Service Bus will be billed as two messages. In most cases, the same method of determining billable messages is applicable to relays as well. See the How is the Messages meter calculated for relays? section for details about the exception cases for relays.

Multiple deliveries of the same message (for example, message fan out to multiple listeners or message retrieval after abandon, deferral, or dead lettering) will be counted as independent messages. For example, in the case of a topic with three subscriptions, a single 64 KB message sent and subsequently received will generate four billable messages (one “in” plus three “out”, assuming all messages are delivered to all subscriptions).

In general, management operations and “control messages,” such as completes and deferrals, are not counted as billable messages. There are two exceptions:

1. Null messages delivered by the Service Bus in response to requests against an empty queue, subscription, or message buffer, are also billable. Thus, applications that poll against Service Bus entities will effectively be charged one message per poll.
2. Setting and getting state on a **MessageSession** will also result in billable messages, using the same message size-based calculation described above.

How is the Messages meter calculated for relays?

In general, billable messages are calculated for relays using the same method as described above for brokered entities (queues, topics/subscriptions and message buffers). However, there are several notable differences:

1. Sending a message to a Service Bus relay is treated as a “full through” send to the relay listener that receives the message, rather than a send to the Service Bus relay followed by a delivery to the relay listener. Therefore, a request-reply style service invocation (of up to 64 KB) against a relay listener will result in two billable messages: one billable message for the request and one billable message for the response (assuming the response is also <= 64 KB). This differs from using a queue to mediate between a client and a service. In the latter case, the same request-reply pattern would require a request send to the queue, followed by a dequeue/delivery from the queue to the service, followed by a response send to another queue, and a dequeue/delivery from that queue to the client. Using the same (<= 64 KB) size assumptions throughout, the mediated queue pattern would thus result in four billable messages, twice the number billed to implement the same pattern using relay. Of course, there are benefits to using queues to achieve this pattern, such as durability and load leveling. These benefits may justify the additional expense.

2. Relays that are opened using the **NetTCPRelay** WCF binding treat messages not as individual messages but as a stream of data flowing through the system. In other words, only the sender and listener have visibility into the framing of the individual messages sent/received using this binding. Thus, for relays using the **netTCPRelay** bindng, all data is treated as a stream for the purpose of calculating billable messages. In this case, the Service Bus will calculate the total amount of data sent or received via each individual relay on an hourly basis and divide that total by 64 KB in order to determine the number of billable messages for the relay in question during that hour.

Are management operations and control messages counted as billable Messages?

Management operations, such as enumerating subscriptions on a topic or determining queue depth, and "control messages," such as completes and deferrals, are not generally counted as billable messages. There are two exceptions:

1. "Null" messages delivered by the Service Bus in response to requests against an empty queue, subscription or message buffer are billable. Therefore, applications that poll against Service Bus entities will effectively be charged one message per poll.
2. Setting and getting state on a **MessageSession** will also result in billable messages, using the same message size based calculation described above.

Does the Service Bus charge for storage?

No, the Service Bus does not charge for storage. However, there is a quota limiting the maximum amount of data that can be persisted per queue/topic. See the Does the Service Bus have any usage quotas? section below.

How much billable usage will I see if I operate 100 queues for 24 hours, each processing one 128 KB message per minute?

- Assume all messages in each queue are delivered exactly once.
- 1 message of 128 KB = 2 billable messages (128 KB/64 KB).
- 2 billable messages sent per minute + 2 billable messages delivered per minute = 4 billable messages per queue per minute.
- 4 messages per queue per minute * 1,440 minutes per day = 5,760 messages per queue per day.
- Total billable messages per day sent to/delivered by the Service Bus = 5,760 messages per queue per day * 100 queues = 576,000 messages per day.
- 576,000 Service Bus messages cost 576,000/10,000 * $0.01 = 58 * $0.01 = $0.58 per day.

How much billable usage will I see if I operate 1 topic with 4 subscriptions for 24 hours, processing one 48 KB message per second?

- Assume all subscriptions receive all messages, and all messages in each subscription are delivered exactly once.

- 1 message of 48 KB = 1 billable message.
- 1 billable message per second * 86,400 seconds per day = 86,400 billable messages per day sent to the topic.
- 86,400 messages per day * 4 subscriptions = 345,600 messages per day delivered to subscription clients.
- Total billable messages per day sent to/delivered by the Service Bus = 86,400 + 345,600 = 432,000 messages per day.
- 432,000 Service Bus messages cost 432,000/10,000 * $0.01 = 44 * $0.01 = $0.44 per day.

How much billable usage will I see if I operate 10 non-netTCP relays for 24 hours, each processing one 8KB message per second?

- Assume a request/reply pattern and all requests receive replies <= 64 KB in size.
- 1 message of 8 KB = 1 billable message.
- 1 billable message per second * 86,400 seconds per day = 86,400 billable messages per day for request messages sent via each relay.
- Since all requests receive replies, also have 86,400 billable messages per day for reply messages sent via each relay.
- Total billable messages per day per relay = 86,400 * 2 = 172,800.
- Total billable messages per day sent to/received by the Service Bus = 172,800 * 10 Relays = 1,728,000.
- 1,728,000 Service Bus messages cost 1,728,000/10,000 * $0.01 = 173 * $0.01 = $1.73.
- 10 relays open 24 hours = 240 relay hours.
- 240 relay hours cost 240/100 * $0.10 = 3 * $0.10 = $0.30.
- Total cost = $1.73 + $0.30 = $2.03 per day.

How much billable usage will I see if I operate 10 netTCP relays for 24 hours, each processing one 8KB message per second?

- Assume a request/reply pattern and all requests receive replies <= 64 KB in size.
- 8 KB per second * 3,600 seconds per hour = 28,800 KB per hour for request messages sent via each relay.
- Since all requests receive replies, also have 28,800 KB per hour for reply messages sent via each relay.
- Total message data per hour per relay = 57,600 KB.
- 57,600 KB = 57,600/64 or 900 billable messages per hour per relay = 900 * 24 or 21,600 billable messages per day per relay.
- Total billable messages per day sent to/received by the Service Bus = 21,600 messages * 10 relays = 216,000.
- 216,000 Service Bus messages cost 216,000/10,000 * $0.01 = 22 * $0.01 = $0.22.
- 10 relays open 24 hours = 240 relay hours.
- 240 relay hours cost 240/100 * $0.10 = 3 * $0.10 = $0.30.

- Total cost = $0.22 + $0.30 = $0.52 per day.

Does the Service Bus have any usage quotas?

By default, for any cloud service Microsoft sets an aggregate monthly usage quota that is calculated across all of a customer's subscriptions. Because we understand that you may need more than these limits, please contact customer service at any time so that we can understand your needs and adjust these limits appropriately. For the Service Bus, the aggregate usage quotas are as follows:

- 5 billion messages
- 2 million relay hours

While we do reserve the right to disable a customer's account that has exceeded its usage quotas in a given month, we will provide e-mail notification and make multiple attempts to contact a customer before taking any action. Customers exceeding these quotas will still be responsible for charges that exceed the quotas.

As with other services on Windows Azure, the Service Bus enforces a set of specific quotas to ensure that there is fair usage of resources. The following are the usage quotas that the service enforces:

- **Queue/topic size** – You specify the maximum queue or topic size upon creation of the queue or topic. This quota can have a value of 1, 2, 3, 4, or 5 GB. If the maximum size is reached, additional incoming messages will be rejected and an exception will be received by the calling code.
- **Number of concurrent connections**
  - **Queue/Topic/Subscription** - The number of concurrent TCP connections on a queue/topic/subscription is limited to 100. If this quota is reached, subsequent requests for additional connections will be rejected and an exception will be received by the calling code. For every messaging factory, the Service Bus maintains one TCP connection if any of the clients created by that messaging factory have an active operation pending, or have completed an operation less than 60 seconds ago. **REST operations do not count towards concurrent TCP connections.**
  - **Message Buffer** - The Service Bus only supports REST operations for message buffers. Therefore, a connection quota does not apply.
- **Number of concurrent listeners on a relay** – The number of concurrent **NetTcpRelay** and **NetHttpRelay** listeners on a relay is limited to 25 (1 for a **NetOneway** relay).
- **Number of concurrent relay listeners per service namespace** – The Service Bus enforces a limit of 2000 concurrent relay listeners per service namespace. If this quota is reached, subsequent requests to open additional relay listeners will be rejected and an exception will be received by the calling code.
- **Number of topics/queues per service namespace** – The maximum number of topics/queues (durable storage-backed entities) on a service namespace is limited to 10,000. If this quota is reached, subsequent requests for creation of a new topic/queue on the service namespace will be rejected. In this case, the management portal will display an error message or the calling client code will receive an exception, depending on whether the create attempt was done via the portal or in client code.

- **Message size quotas**
  - **Queue/Topic/Subscription**
    - **Message size** – Each message is limited to a total size of 256KB, including message headers.
    - **Message header size** – Each message header is limited to 64KB.
  - **Message Buffer** - Each message is limited to a total size of 64KB, including message headers.
  - **NetOneway and NetEvent relays** - Each message is limited to a total size of 64KB, including message headers.
  - **Http and NetTcp relays** – The Service Bus does not enforce an upper bound on the size of these messages.

  Messages that exceed these size quotas will be rejected and an exception will be received by the calling code.
- **Number of subscriptions per topic** – The maximum number of subscriptions per topic is limited to 2,000. If this quota is reached, subsequent requests for creating additional subscriptions to the topic will be rejected. In this case, the management portal will display an error message or the calling client code will receive an exception, depending on whether the create attempt was done via the portal or in client code.
- **Number of SQL filters per topic** – the maximum number of SQL filters per topic is limited to 2,000. If this quota is reached, any subsequent requests for creation of additional filters on the topic will be rejected and an exception will be received by the calling code.
- **Number of correlation filters per topic** – the maximum number of correlation filters per topic is limited to 100,000. If this quota is reached, any subsequent requests for creation of additional filters on the topic will be rejected and an exception will be received by the calling code.

For more information about quotas, see Windows Azure Service Bus Quotas.

System and Developer Requirements

Requirements

The following topic describes the developer and system requirements for running a Windows Azure application that uses the Service Bus.

Developer Requirements

In order to develop and run an Windows Azure application, you should have a basic level of familiarity with either C# or Visual Basic .NET. You should also be familiar with the following technologies:

- **Service Bus**

If you are developing an Service Bus basic service or client application, you should be generally familiar with the Windows Communication Foundation (WCF) programming framework. It is possible to use this documentation without knowing WCF programming. However, many of the topics here reference WCF as a source of additional information, or use the WCF documentation as a base from which to expand.

- **Windows Azure and Service Bus**

  Creating a Windows Azure application that uses the Service Bus is very similar to a basic Service Bus application: the main additions are mainly around configuration and setup. Therefore, you should be generally familiar with the Windows Azure programming environment, so that you can reasonably create a basic Windows Azure application.

- **REST and the Service Bus**

  As with Windows Azure, creating a REST-based service or client application is very similar to the basic Service Bus programming model: other than the choice of bindings and some tagging, the actual differences are relatively minor. Therefore, you should be generally familiar with the REST protocol and Web programming. If you want to use the message buffer feature to create a purely HTTP-based application (for example, one that does not use the Service Bus and the Service Bus service), you should be much more experienced with the Web programming and messaging models.

System Requirements

In order to run an Service Bus client or service application, you must have the following:

- Windows XP Service Pack 3 or higher, Windows Vista, Windows Server 2008, Windows Server 2008 R2, or Windows°7.
- .NET Framework 3.5, Service Pack 1.
- Windows Azure SDK.
- HTTP/S connectivity to the Internet.
- To use TCP connectivity, your computer must be able to open outbound connections to the Service Bus using ports 808 and 828.
- To establish direct connections to clients (hybrid mode), your computer must be able to connect to the Service Bus using port 819.

For more information about port settings, see Service Bus Port Settings.

Managing Service Bus Service Namespaces

The following topics describe how to create and manage Windows Azure Service Bus service namespaces using the Windows Azure management portal.

In This Section

How to: Create or Modify a Service Bus Service Namespace

How to: Delete a Service Bus Service Namespace

How to: Create or Modify a Service Bus Service Namespace

The following topic describes how to create a new Service Bus service namespace using the Windows Azure management portal, and how to modify and existing service namespace. You can create one or more service namespaces in your Service Bus subscription.

To create a Service Bus service namespace

1. Open an Internet browser and visit the Windows Azure Management Portal.
2. Log on to the Web site using a Windows Live ID. If you do not have a Windows Live ID, click **Sign up** to create one for yourself.

   After you are signed in with your Live ID, you are redirected to the Management Portal page. On the lower-left-hand side of this page, click **Service Bus, Access Control & Caching**.
3. To create a new service namespace at the global (**Windows Azure**) level, click **Services** in the tree on the left-hand side. Then click **New** from the **Service Namespace** area of the toolbar at the top of the page.

   OR
4. Click the name of the service in the tree. Then click **New** from the **Service Namespace** area of the toolbar. In the resulting dialog, you can select the service or services for which you want to create the service namespace.
5. In the **Create a new Service Namespace** box, type a name for your new service namespace. Note that each service namespace must be globally unique so that it can be identified by the service.

   Note

   You do not have to use the same service namespace for both client and service applications.
6. Click **Check Availability**. This will check that the name that you selected for your service namespace is valid and available. If it is not, you can enter a different name.
7. Choose a region and in the available dropdown. Then click **OK**.
8. The system now creates your service namespace and enables it.

   You might have to wait several minutes as the system provisions resources for your account.

To modify a service namespace

1. To modify a service namespace, click the service namespace you want to change. Then click **Modify** in the toolbar at the top of the page.
2. In the **Available Services** pane of the **Modify a Service Namespace** dialog, you can

add or remove the services for which this namespace is valid. You can only customize the service properties for services being added. At this time you cannot modify the properties of service namespaces for existing services.

3. Note that if you clear the checkboxes for all services in the **Available Services** pane, the service namespace is deleted.
4. When you are finished, click **Modify Namespace** to commit the changes and close the **Modify a Service Namespace** dialog.

How to: Delete a Service Bus Service Namespace

The following topic describes how to delete a service namespace from a Windows Azure Service Bus subscription.

To delete a service namespace

1. In the tree on the left-hand side of the Windows Azure Management Portal, click the service that contains the service namespace you want to delete.
2. Click the service namespace. Then click **Delete** in the **Service Namespace** area of the toolbar at the top of the page.

   OR
3. Click the service namespace you want to delete. Then click **Modify** in the toolbar at the top of the page.
4. Clear the checkboxes for all services in the **Available Services** pane, then click **Modify Namespace**. The service namespace is deleted.

Windows Azure Service Bus Quotas

This section enumerates basic quotas and throttling thresholds in Windows Azure Service Bus messaging.

General Quotas

The maximum number of service namespaces allowed per Windows Azure account is 50.

For information about other quotas for the Windows Azure Service Bus, see the Windows Azure Platform pricing FAQ.

Messaging Quotas

The following table lists quota information specific to the Service Bus:

Quota Name	Scope	Type	Behavior when exceeded	Value
Queue/Topic size	Entity	Defined upon creation of the queue/topic.	Incoming messages will be rejected and an exception will be received by the calling code.	1,2,3,4 or 5 Gigabytes.
Number of concurrent connections on a queue/topic/subscription entity	Entity	Static	Subsequent requests for additional connections will be rejected and an exception will be received by the calling code. REST operations do not count towards concurrent TCP connections.	100
Number of concurrent listeners on a relay	Entity	Static	Subsequent requests for additional connections will be rejected and an exception will be received by the calling code.	25
Number of concurrent relay listeners	System-wide	Static	Subsequent requests for additional connections will be rejected and an exception will be received by the calling code.	2000
Number of topics/queues per service namespace	System-wide	Static	Subsequent requests for creation of a new topic or queue on the service namespace will be rejected. As a result, if configured through the management portal, an error message will be generated. If called from the management API, an exception will be received by the calling code.	10,000
Message size for a queue/topic/subscription entity	System-wide	Static	Incoming messages that exceed these quotas will be rejected and an exception will be received by the calling code.	Maximum message size: 256KB Maximum header size: 64KB Maximum number of header properties in property bag:

Quota Name	Scope	Type	Behavior when exceeded	Value
				MaxValue Maximum size of property in property bag: No explicit limit. Limited by maximum header size.
Message size for Message Buffer	System-wide	Static	Incoming messages that exceed these quotas will be rejected and an exception will be received by the calling code.	64KB
Message size for **Microsoft.ServiceBus.NetOnewayRelayBinding** and **Microsoft.ServiceBus.NetEventRelayBinding** relays	System-wide	Static	Incoming messages that exceed these quotas will be rejected and an exception will be received by the calling code.	64KB
Message size for **Microsoft.ServiceBus.HttpRelayTransportBindingElement** and **Microsoft.ServiceBus.NetTcpRelayBinding** relays	System-wide	Static		Unlimited
Message property size for a queue/topic/subscription entity	System-wide	Static	A **SerializationException** exception is generated.	Maximum message property size for each property is 32K. Cumulative size of all properties

Quota Name	Scope	Type	Behavior when exceeded	Value
				cannot exceed 64K. This applies to the entire header of the **Microsoft.ServiceBus.Messaging.BrokeredMessage**, which has both user properties as well as system properties (such as **Microsoft.ServiceBus.Messaging.BrokeredMessage.SequenceNumber**, **Microsoft.ServiceBus.Messaging.BrokeredMessage.Label**, **Microsoft.ServiceBus.Messaging.BrokeredMessage.MessageId**, and so on).
Number of subscriptions per topic	System-wide	Static	Subsequent requests for creating additional subscriptions for the topic will be rejected. As a result, if configured through the management portal, an error message will be shown. If called from the management API an exception will be received by the calling code.	2,000

Quota Name	Scope	Type	Behavior when exceeded	Value
Number of SQL filters per topic	System-wide	Static	Subsequent requests for creation of additional filters on the topic will be rejected and an exception will be received by the calling code.	2,000
Number of correlation filters per topic	System-wide	Static	Subsequent requests for creation of additional filters on the topic will be rejected and an exception will be received by the calling code.	100,000
Size of SQL filters/actions	System-wide	Static	Subsequent requests for creation of additional filters will be rejected and an exception will be received by the calling code.	Maximum length of filter condition string: 4K Maximum length of rule action string: 4K Maximum number of expressions per rule action: 64

Getting Started with the Service Bus

The following topics contain tutorials for the Windows Azure Service Bus.

The Service Bus Relayed Messaging Tutorial topics create a basic Windows Communication Foundation (WCF) service application that is configured to register an endpoint for publication with the Service Bus and a WCF client application that invokes it through the Service Bus endpoint. In this tutorial, both the host and client applications are executed on a Windows server or desktop computer (that is, they are not hosted in Windows Azure) and use a common standard protocol and security measures to access the Service Bus. Read through this tutorial to see how to use the Service Bus to securely expose a WCF service to a WCF client application by using standard SOAP-based WCF bindings.

The Service Bus Brokered Messaging Tutorials contain both REST-based and managed API-based tutorials that demonstrate how to use the new Service Bus brokered messaging features, such as Queues, Topics, and Subscriptions.

The Service Bus Message Buffer Tutorial topics show how to securely use REST message exchanges to register a REST endpoint with the Service Bus and build a REST client that communicates with the REST service through the Service Bus endpoint. This tutorial demonstrates one programming approach, by using the **WCF Web HTTP Programming Model** to create a service application that registers an HTTP/GET endpoint securely through the Service Bus, by using the standard WCF bindings that are available with Windows Azure. The client can then use any Web browser to start the service by using the URL of the Service Bus endpoint.

You can of course also use REST-style communication to register a REST endpoint with the Service Bus, although the REST tutorial does not demonstrate this.

In This Section

Service Bus Relayed Messaging Tutorial

Includes topics that describe how to build a simple Service Bus client application and service.

Service Bus Brokered Messaging Tutorials

Includes topics that describe how to build applications that use the new Service Bus brokered messaging features.

Service Bus Message Buffer Tutorial

Includes topics that describe how to build a simple Service Bus host application that exposes a REST interface.

Service Bus Relayed Messaging Tutorial

The following topics describe how to build a simple Windows Azure Service Bus client application and service using the Service Bus “relay” messaging capabilities. For a corresponding tutorial that describes how to build an application that uses the Service Bus “brokered,” or asynchronous messaging capabilities, see the Service Bus Brokered Messaging .NET Tutorial.

The topics that are contained in this section are intended to give you quick exposure to the Service Bus programming experience. They are designed to be completed in the order listed at the bottom of this topic. Working through this tutorial gives you an introductory understanding of the steps that are required to create an Service Bus client and service application. Like their WCF counterparts, a service is a construct that exposes one or more endpoints, each of which exposes one or more service operations. The endpoint of a service specifies an address where the service can be found, a binding that contains the information that a client must communicate with the service, and a contract that defines the functionality provided by the service to its clients. The

main difference between a WCF and an Service Bus service is that the endpoint is exposed in the cloud instead of locally on your computer.

After you work through the sequence of topics in this tutorial, you will have a running service, and a client that can invoke the operations of the service. The first topic describes how to set up an account. The next three topics describe how to define a service that uses a contract, how to implement the service, and how to configure the service in code. They also describe how to host and run the service. The service that is created is self-hosted and the client and service run on the same computer. You can configure the service by using either code or a configuration file. For more information, see Configuring a WCF Service to Register with the Service Bus and Building a Service for the Service Bus.

The next three topics describe how to create a client application, configure the client application, and create and use a client that can access the functionality of the host. For more information, see Building a Service Bus Client Application and Discovering and Exposing a Service Bus Service.

All of the topics in this section assume that you are using Visual Studio 2010 as the development environment. If you are using another development environment, ignore the Visual Studio-specific instructions.

For more in-depth information about how to create Service Bus client applications and hosts, see the Developing Applications that Use the Service Bus section.

In This Section

Step 1: Sign up for an Account

Step 2: Define a WCF Service Contract to use with Service Bus

Step 3: Implement the WCF Contract to use Service Bus

Step 4: Host and Run a Basic Web Service to Register with Service Bus

Step 5: Create a WCF Client for the Service Contract

Step 6: Configure the WCF Client

Step 7: Implement WCF Client to Call the Service Bus

See Also

Service Bus Brokered Messaging .NET Tutorial

Step 1: Sign up for an Account

This is the first of seven tasks required to create a basic Windows Communication Foundation (WCF) service and a client that can call the service that uses the Windows Azure Service Bus. For an overview of all seven of the tasks, see the Service Bus Relayed Messaging Tutorial overview. The next task is Step 2: Define a WCF Service Contract to use with Service Bus.

The first step is to create a Windows Azure service namespace, and to obtain a *shared secret* key. A service namespace provides an application boundary for each application exposed

through the Service Bus. A shared secret key is automatically generated by the system when a service namespace is created. The combination of service namespace and shared secret key provides a credential for the Service Bus to authenticate access to an application.

Expected time to complete: 5 minutes

To create a service namespace

1. To create a service namespace, follow the steps outlined in How to: Create or Modify a Service Bus Service Namespace.

   Note
   You do not have to use the same service namespace for both client and service applications.

Step 2: Define a WCF Service Contract to use with Service Bus

This is the second of seven tasks required to create a basic Windows Communication Foundation (WCF) service and a client that can call the service that uses the Windows Azure Service Bus. For an overview of all seven of the tasks, see the Service Bus Relayed Messaging Tutorial. The previous step is Step 1: Sign up for an Account; the subsequent step is Step 3: Implement the WCF Contract to use Service Bus.

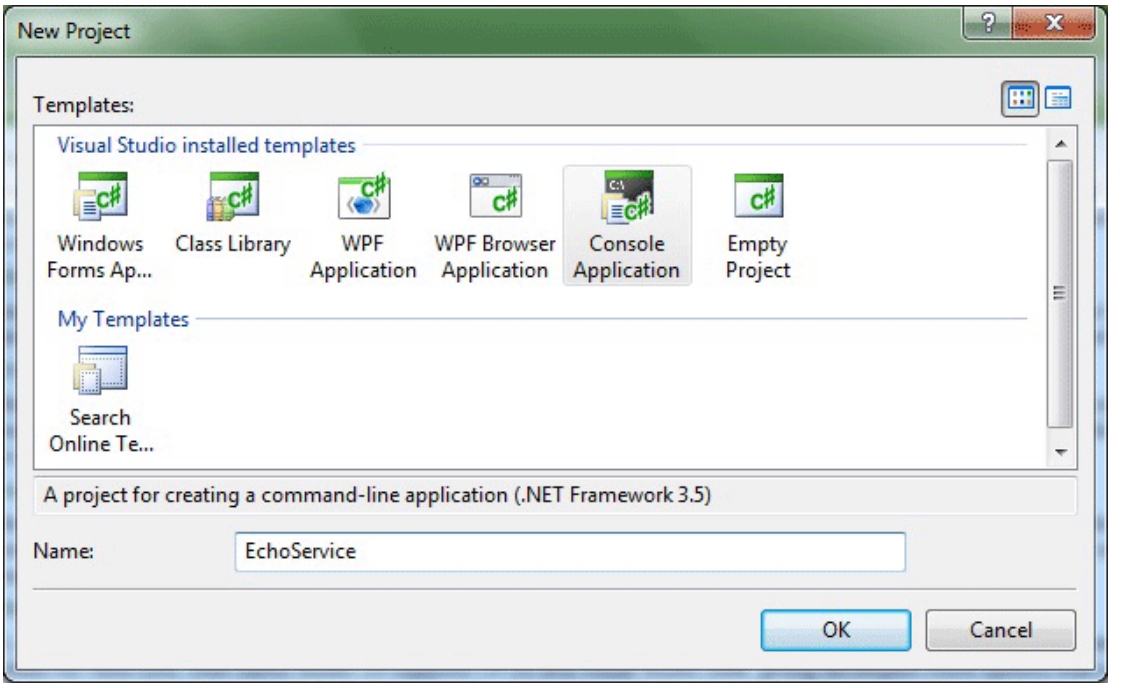
When creating a basic WCF service (whether for the Service Bus or not), you must define the service contract, which specifies what operations (the Web service terminology for *methods* or *functions*) the service supports. Contracts are created by defining a C++, C#, or Visual Basic interface. This tutorial demonstrates how to create such a contract using C#. Each method in the interface corresponds to a specific service operation. Each interface must have the **System.ServiceModel.ServiceContractAttribute** attribute applied to it, and each operation must have the **System.ServiceModel.OperationContractAttribute** applied to it. If a method in an interface that has the **System.ServiceModel.ServiceContractAttribute** does not have the **System.ServiceModel.OperationContractAttribute**, that method is not exposed. The code for these tasks is provided in the example following the procedure. For more information about how to define a contract, see Designing a WCF Contract for the Service Bus. For a larger discussion of contracts and services, see **Designing and Implementing Services** in the WCF documentation.

Expected time to completion: 10 minutes.

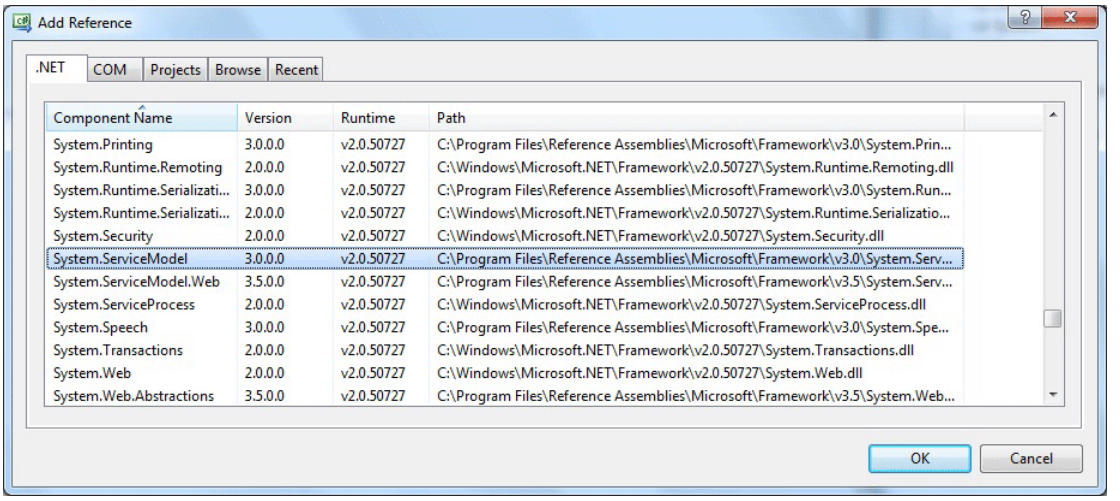
To create a Service Bus contract with an interface

1. Open Visual Studio 2010 as an administrator by right-clicking the program in the **Start** menu and selecting **Run as administrator**.
2. Create a new console application project. Click the **File** menu and select **New**, then click **Project**. In the **New Project** dialog, click **Visual C#** (if **Visual C#** does not appear, look under **Other Languages**), click the **Console Application** template, and name it

EchoService. Use the default **Location**. Click **OK** to create the project.



3. Note that the following two steps (4 and 5) are not necessary if you are running Visual Studio 2008.
4. In the **Solution Explorer**, right-click the name of your project (in this example, **EchoService**), and click **Properties**.
5. Click the **Application** tab on the left, then select **.NET Framework 4** from the **Target framework:** dropdown. Click **Yes** when prompted to reload the project.
6. For a C# project, Visual Studio creates a file that is named Program.cs. This class will contain an empty method called `Main()`. This method is required for a console application project to build correctly. Therefore, you can safely leave it in the project.
7. Add a reference to System.ServiceModel.dll to the project:
   a. In the **Solution Explorer**, right-click the **References** folder under the project folder and then click **Add Reference…**.
   b. Select the **.NET** tab in the **Add Reference** dialog and scroll down until you see **System.ServiceModel**, select it, and then click **OK**.



Note

When using a command-line compiler (for example, Csc.exe), you must also provide the path of the assemblies. By default, for example on a computer that is running Windows°7, the path is: Windows\Microsoft.NET\Framework\v3.0\Windows Communication Foundation.

8. In the **Solution Explorer**, double-click the Program.cs file to open it in the editor.
9. Add a `using` statement for the `System.ServiceModel` namespace.

```
using System.ServiceModel;
```

System.ServiceModel is the namespace that lets you programmatically access the basic features of WCF. Service Bus uses many of the objects and attributes of WCF to define service contracts. You will most likely use this namespace in most of your Service Bus applications.

10. Change the namespace name from its default name of `EchoService` to `Microsoft.ServiceBus.Samples`.

Important

This tutorial uses the C# namespace **Microsoft.ServiceBus.Samples**, which is the namespace of the contract managed type that is used in the configuration file in Step 6: Configure the WCF Client. You can specify any namespace you want when you build this sample; however, the tutorial will not work unless you then modify the namespaces of the contract and service accordingly, in the application configuration file. The namespace specified in the App.config file must be the same as the namespace specified in your C# files.

11. Directly after the `Microsoft.ServiceBus.Samples` namespace declaration, but within the namespace, define a new interface named `IEchoContract` and apply the `ServiceContractAttribute` attribute to the interface with a namespace value of **http://samples.microsoft.com/ServiceModel/Relay/**. The namespace value differs from the namespace that you use throughout the scope of your code. Instead, the namespace

value is used as a unique identifier for this contract. Specifying the namespace explicitly prevents the default namespace value from being added to the contract name.

```
[ServiceContract(Name = "IEchoContract", Namespace =
"http://samples.microsoft.com/ServiceModel/Relay/")]
public interface IEchoContract
{
}
```

Note

Typically, the service contract namespace contains a naming scheme that includes version information. Including version information in the service contract namespace enables services to isolate major changes by defining a new service contract with a new namespace and exposing it on a new endpoint. In in this manner, clients can continue to use the old service contract without having to be updated. Version information can consist of a date or a build number. For more information, seeService Versioning. For the purposes of this tutorial, the naming scheme of the service contract namespace does not contain version information.

12. Within the `IEchoContract` interface, declare a method for the single operation the `IEchoContract` contract exposes in the interface and apply the `OperationContractAttribute` attribute to the method that you want to expose as part of the public Service Bus contract.

```
[OperationContract]
string Echo(string text);
```

13. Outside the contract, declare a channel that inherits from both `IEchoChannel` and also to the `IClientChannel` interface, as shown here:

```
    [ServiceContract(Name = "IEchoContract", Namespace =
"http://samples.microsoft.com/ServiceModel/Relay/")]
    public interface IEchoContract
    {
        [OperationContract]
        String Echo(string text);
    }

    public interface IEchoChannel : IEchoContract,
IClientChannel { }
```

A channel is the WCF object through which the host and client pass information to each other. Later, you will write code against the channel to echo information between the two applications.

14. From the **Build** menu, select **Build Solution** or press **F6** to confirm the accuracy of your work.

Example

Description

The following code example shows a basic interface that defines an Service Bus contract.

Code

```
using System;
using System.ServiceModel;


namespace Microsoft.ServiceBus.Samples
{
    [ServiceContract(Name = "IEchoContract", Namespace =
"http://samples.microsoft.com/ServiceModel/Relay/")]
    public interface IEchoContract
    {
        [OperationContract]
        String Echo(string text);
    }

    public interface IEchoChannel : IEchoContract, IClientChannel { }



    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Comments

Now that the interface is created, you can implement the interface, as described in Step 3: Implement the WCF Contract to use Service Bus.

Step 3: Implement the WCF Contract to use Service Bus

This is the third of seven tasks required to create a basic Windows Communication Foundation (WCF) service and a client that can call the service that uses the Windows Azure Service Bus. For an overview of all seven tasks, see the Service Bus Relayed Messaging Tutorial topic. The previous task is Step 2: Define a WCF Service Contract to use with Service Bus; the following task is Step 4: Host and Run a Basic Web Service to Register with Service Bus. Creating an Service Bus service requires that you first create the contract, which is defined by using an interface. For more information about creating the interface, see Step 2: Define a WCF Service Contract to use with Service Bus. The next step, shown in this topic, is to implement the interface. This involves creating a class named `EchoService` that implements the user-defined `IEchoContract` interface. After you implement the interface, you then configure the interface using an App.config configuration file. The configuration file contains necessary information for the application, such as the name of the service, the name of the contract, and the type of protocol that is used to communicate with the Service Bus. The code used for these tasks is provided in the example following the procedure. For a more general discussion about how to implement a service contract, see **Implementing Service Contracts** in the Windows Communication Foundation (WCF) documentation.

Expected time to completion: 10 minutes

To implement a Service Bus contract

1. Create a new class named `EchoService` directly underneath the definition of the `IEchoContract` interface. The `EchoService` class implements the `IEchoContract` interface.

   ```
   class EchoService : IEchoContract
   {
   }
   ```

   Similar to other interface implementations, you can implement the definition in a different file. However, for this tutorial, the implementation is located in the same file as the interface definition and the `Main` method.
2. Apply the **System.ServiceModel.ServiceBehaviorAttribute** attribute that indicates the service name and namespace.

   ```
   [ServiceBehavior(Name = "EchoService", Namespace =
   "http://samples.microsoft.com/ServiceModel/Relay/")]
   class EchoService : IEchoContract
   {
   }
   ```

3. Implement the `Echo` method defined in the `IEchoContract` interface in the `EchoService` class.

```
public string Echo(string text)
{
    Console.WriteLine("Echoing: {0}", text);
    return text;
}
```

4. Click **Build**. Then click **Build Solution** to confirm the accuracy of your work.

To define the configuration for the service host

1.

   Note

   Steps 1 and 2 are not necessary if you are using Visual Studio 2010 , because by default, the App.config file is already present in the project.

   In **Solution Explorer**, right-click the **EchoService** project, select **Add**. Then click **New Item**.

2. In the **Add New Item** dialog, in the **Visual Studio installed templates** pane, select **Application Configuration** file, and then click **Add**.

   The configuration file is very similar to a WCF configuration file, and includes the service name, endpoint (that is, the location Service Bus exposes for clients and hosts to communicate with each other), and the binding (the type of protocol that is used to communicate). The main difference is that this configured service endpoint refers to a `netTcpRelayBinding`, which is not part of the .NET Framework 3.5. **Microsoft.ServiceBus.NetTcpRelayBinding** is one of the new bindings introduced with the Service Bus.

3. In **Solution Explorer**, click **App.config**, which currently contains the following XML elements:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
</configuration>
```

4. Add a `<system.serviceModel>` XML element to the App.config file. This is a WCF element that defines one or more services. This example uses it to define the service name and endpoint.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.serviceModel>
```

```
</system.serviceModel>


</configuration>
```

5. Within the `<system.serviceModel>` tags, add a `<services>` element. You can define multiple Service Bus applications in a single configuration file. However, this tutorial defines only one.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.serviceModel>
<services>


</services>
</system.serviceModel>
</configuration>
```

6. Within the `<services>` element, add a `<service>` element to define the name of the service.

```
<service name="Microsoft.ServiceBus.Samples.EchoService">
</service>
```

7. Within the `<service>` element, define the location of the endpoint contract, and also the type of binding for the endpoint.

```
<endpoint
contract="Microsoft.ServiceBus.Samples.IEchoContract"
binding="netTcpRelayBinding" />
```

   The endpoint defines where the client will look for the host application. Later, the tutorial uses this step to create a URI that fully exposes the host through the Service Bus. The binding declares that we are using TCP as the protocol to communicate with the Service Bus.

8. Directly after the `<services>` element, add the following binding extension:

```
<extensions>
<bindingExtensions>
<add name="netTcpRelayBinding"
type="Microsoft.ServiceBus.Configuration.NetTcpRelayBindingCo
llectionElement, Microsoft.ServiceBus, Version=1.5.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
</bindingExtensions>
</extensions>
```

9. Click **Build**, and then click **Build Solution** to confirm the accuracy of your work.

Example

Description

The following code example shows the implementation of the service contract.

Code

```
[ServiceBehavior(Name = "EchoService", Namespace =
"http://samples.microsoft.com/ServiceModel/Relay/")]


    class EchoService : IEchoContract
    {
        public string Echo(string text)
        {
            Console.WriteLine("Echoing: {0}", text);
            return text;
        }
    }
```

Example

Description

The following example shows the basic format of the App.config file associated with the service host.

Code

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.serviceModel>
<services>
<service name="Microsoft.ServiceBus.Samples.EchoService">
<endpoint contract="Microsoft.ServiceBus.Samples.IEchoContract"
binding="netTcpRelayBinding" />
</service>
</services>
<extensions>
```

```
<bindingExtensions>
<add name="netTcpRelayBinding"
type="Microsoft.ServiceBus.Configuration.NetTcpRelayBindingCollectionElement,
Microsoft.ServiceBus, Version=1.5.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"
/>
</bindingExtensions>
</extensions>
</system.serviceModel>
</configuration>
```

Comments

Now that you have implemented the Service Bus contract and configured your endpoints, proceed to Step 4: Host and Run a Basic Web Service to Register with Service Bus.

Step 4: Host and Run a Basic Web Service to Register with Service Bus

This is the fourth of seven tasks required to create a basic Service Bus service application and a client that can call the service. For an overview of all six of the tasks, see the Service Bus Relayed Messaging Tutorial topic. This topic describes how to run a basic Service Bus service.

A complete listing of the code written in this task is provided in the example following the procedure.

Estimated time to completion: 10 minutes

To create the Service Bus credentials

1. Add a reference to Microsoft.ServiceBus.dll to the project:
   a. In **Solution Explorer**, right-click **References** under the project folder and then click **Add Reference**.
   b. Select the **.NET** tab in the **Add Reference** dialog and scroll down until you see **Microsoft.ServiceBus**. Or, click Browse and navigate to the assembly on your hard drive. Typically, this is located in Program Files\<sdkInstallDirectory>\v1.0\Assemblies\NET4.0. Select the **Microsoft.ServiceBus** assembly, then click **OK**.

   Note
   When using a command-line compiler (for example, Csc.exe), you must also provide the path to the assemblies.
2. In Program.cs, add a `using` statement for the `Microsoft.ServiceBus` namespace.

   ```
   using Microsoft.ServiceBus;
   ```

   Microsoft.ServiceBus is the namespace that lets you programmatically access many of the core features of the Service Bus. You will most likely use this namespace in all your

Service Bus applications.

3. In the `Main()` method, create three variables in which to store the service namespace, issuer name, and issuer secret (a name/password credential) that are read from the console window.

```
Console.Write("Your Service Namespace: ");
string serviceNamespace = Console.ReadLine();
Console.Write("Your Issuer Name: ");
string issuerName = Console.ReadLine();
Console.Write("Your Issuer Secret: ");
string issuerSecret = Console.ReadLine();
```

   The issuer name and issuer secret will be used later to access your Service Bus project. The service namespace is passed as a parameter to `CreateServiceUri` to create a service URI.

4. Using a **Microsoft.ServiceBus.TransportClientEndpointBehavior** object, declare that you will be using a shared secret as the credential type. Add the following code directly underneath the code added in the last step.

```
TransportClientEndpointBehavior
sharedSecretServiceBusCredential = new
TransportClientEndpointBehavior();
sharedSecretServiceBusCredential.TokenProvider =
TokenProvider.CreateSharedSecretTokenProvider(issuerName,
issuerSecret);
```

To create a base address for the service

1. Following the code you added in the last step, create a `Uri` instance for the base address of the service. This URI specifies the Service Bus scheme, the service namespace, and the path of the service interface.

```
Uri address = ServiceBusEnvironment.CreateServiceUri("sb",
serviceNamespace, "EchoService");
```

   "`sb`" is an abbreviation for the Service Bus scheme, and indicates that we are using TCP as the protocol. This was also previously indicated in the configuration file, when **Microsoft.ServiceBus.NetTcpRelayBinding** was specified as the binding.

   For this tutorial, the URI is `sb://putServiceNamespaceHere.windows.net/EchoService`.

To create and configure the service host

1. Set the connectivity mode to `AutoDetect`.

```
ServiceBusEnvironment.SystemConnectivity.Mode =
ConnectivityMode.AutoDetect;
```

The connectivity mode describes the protocol the service uses to communicate with the Service Bus; either HTTP or TCP. Using the default setting `AutoDetect`, the service will attempt to connect to the Service Bus over TCP, if it is available, and HTTP if TCP is not available. Note that this differs from the protocol the service specifies for client communication. That protocol is determined by the binding used. For example, a service can use the **Microsoft.ServiceBus.BasicHttpRelayBinding** binding, which specifies that its endpoint (exposed on the Service Bus) communicates with clients over HTTP. That same service could specify **ConnectivityMode.AutoDetect** so that the service communicates with the Service Bus over TCP.

2. Create the service host, using the URI created earlier in this section.

```
ServiceHost host = new ServiceHost(typeof(EchoService),
address);
```

The service host is the WCF object that instantiates the service. Here, you pass it the type of service you want to create (an `EchoService` type), and also to the address at which you want to expose the service.

3. At the top of the Program.cs file, add references to **System.ServiceModel.Description** and **Microsoft.ServiceBus.Description**.

```
using System.ServiceModel.Description;
using Microsoft.ServiceBus.Description;
```

4. Back in `Main()`, configure the endpoint to enable public access.

```
IEndpointBehavior serviceRegistrySettings = new
ServiceRegistrySettings(DiscoveryType.Public);
```

This step informs the Service Bus that your application can be found publicly by examining the Service Bus ATOM feed for your project. If you set **DiscoveryType** to **private**, a client would still be able to access the service. However, the service would not appear when it searches the Service Bus namespace. Instead, the client would have to know the endpoint path beforehand. For more information, see Discovering and Exposing a Service Bus Service.

5. Apply the service credentials to the service endpoints defined in the App.config file:

```
foreach (ServiceEndpoint endpoint in
host.Description.Endpoints)
{
    endpoint.Behaviors.Add(serviceRegistrySettings);
    endpoint.Behaviors.Add(sharedSecretServiceBusCredential);
}
```

As stated in the previous step, you could have declared multiple services and endpoints in the configuration file. If you had, this code would traverse the configuration file and search for every endpoint to which it should apply your credentials. However, for this tutorial, the configuration file has only one endpoint.

To open the service host

1. Open the service.

```
host.Open();
```

2. Inform the user that the service is running, and explain how to shut down the service.

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

3. When finished, close the service host.

```
host.Close();
```

4. Press F6 to build the project.

Example

Description

The following example includes the service contract and implementation from previous steps in the tutorial, and hosts the service in a console application. Compile the following into an executable named EchoService.exe.

Code

```
using System;
using System.ServiceModel;
using System.ServiceModel.Description;
using Microsoft.ServiceBus;
using Microsoft.ServiceBus.Description;


namespace Microsoft.ServiceBus.Samples
{
    [ServiceContract(Name = "IEchoContract", Namespace =
"http://samples.microsoft.com/ServiceModel/Relay/")]


    public interface IEchoContract
    {
        [OperationContract]
        String Echo(string text);
    }
```

```
    public interface IEchoChannel : IEchoContract, IClientChannel { };

    [ServiceBehavior(Name = "EchoService", Namespace =
"http://samples.microsoft.com/ServiceModel/Relay/")]

    class EchoService : IEchoContract
    {
        public string Echo(string text)
        {
            Console.WriteLine("Echoing: {0}", text);
            return text;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {

            ServiceBusEnvironment.SystemConnectivity.Mode = ConnectivityMode.AutoDetect;



            Console.Write("Your Service Namespace: ");
            string serviceNamespace = Console.ReadLine();
            Console.Write("Your Issuer Name: ");
            string issuerName = Console.ReadLine();
            Console.Write("Your Issuer Secret: ");
            string issuerSecret = Console.ReadLine();

           // Create the credentials object for the endpoint.
            TransportClientEndpointBehavior sharedSecretServiceBusCredential = new
TransportClientEndpointBehavior();
```

```
            sharedSecretServiceBusCredential.TokenProvider =
TokenProvider.CreateSharedSecretTokenProvider(issuerName, issuerSecret);


            // Create the service URI based on the service namespace.
            Uri address = ServiceBusEnvironment.CreateServiceUri("sb", serviceNamespace,
"EchoService");

            // Create the service host reading the configuration.
            ServiceHost host = new ServiceHost(typeof(EchoService), address);

            // Create the ServiceRegistrySettings behavior for the endpoint.
            IEndpointBehavior serviceRegistrySettings = new
ServiceRegistrySettings(DiscoveryType.Public);

            // Add the Service Bus credentials to all endpoints specified in
configuration.
            foreach (ServiceEndpoint endpoint in host.Description.Endpoints)
            {
                endpoint.Behaviors.Add(serviceRegistrySettings);
                endpoint.Behaviors.Add(sharedSecretServiceBusCredential);
            }

            // Open the service.
            host.Open();

            Console.WriteLine("Service address: " + address);
            Console.WriteLine("Press [Enter] to exit");
            Console.ReadLine();

            // Close the service.
            host.Close();
        }
    }
```

```
}
```

Comments

Now that the interface is created, proceed to Step 5: Create a WCF Client for the Service Contract to implement the interface.

Step 5: Create a WCF Client for the Service Contract

This is the fifth of seven tasks required to create a basic Service Bus service and a client application that can call the service. For an overview of all seven of the tasks, see the Service Bus Relayed Messaging Tutorial topic. This topic describes how to create a basic Service Bus client application and define the service contract you will be implementing in later steps. Note that many of these steps resemble the steps used to create a service: defining a contract, editing an App.config file, using credentials to connect to the Service Bus, and so on. The code used for these tasks is provided in the example following the procedure.

Note

For the purposes of this tutorial, both the client and the service run on the same network and computer. However, this is not required. The advantage of using the Service Bus is to enable applications across network boundaries to seamlessly communicate.

Time to completion: 10 minutes

To create the Service Bus client application

1. Create a new project in the current Visual Studio solution for the client by doing the following:
   a. In **Solution Explorer**, in the same solution that contains the service, right-click the current solution (not the project), and select **Add**. Then click **New Project**.
   b. In the **Add New Project** dialog, click **Visual C#** (if **Visual C#** does not appear, look under **Other Languages**), select the **Console Application** template, and name it **EchoClient**.
   c. Click **OK**.
2. Note that the following two steps (3 and 4) are not necessary if you are running Visual Studio 2008.
3. In the **Solution Explorer**, right-click the name of your project (in this example, **EchoClient**), and click **Properties**.
4. Click the **Application** tab on the left, then select **.NET Framework 4** from the **Target framework:** dropdown. Click **Yes** when prompted to reload the project.
5. In the **Solution Explorer**, double-click the Program.cs file in the **EchoClient** project to open it in the editor.
6. Change the namespace name from its default name of `EchoClient` to `Microsoft.ServiceBus.Samples`.
7. Add a reference to System.ServiceModel.dll for the project:
   a. Right-click **References** under the **EchoClient** project in **Solution Explorer**. Then

click **Add Reference**.

b. Because you already added a reference to this assembly in the first step of this tutorial, it is now listed in the **Recent** tab. Click the **Recent** tab, select **System.ServiceModel.dll** from the list. Then click **OK**. If you do not see **System.ServiceModel.dll** in the **Recent** tab, click the **Browse** tab and move to **C:\Windows\Microsoft.NET\Framework\v3.0\Windows Communication Foundation**. Then select the assembly from there.

8. Add a `using` statement for the **System.ServiceModel** namespace in the Program.cs file.

```
using System.ServiceModel;
```

9. Repeat steps two and three to add a reference to the Microsoft.ServiceBus.dll and **Microsoft.ServiceBus** namespace to your project.
10. Add the service contract definition to the namespace, as shown in the following example. Note that this definition is identical to the definition used in the **Service** project. You should add this code at the top of the `Microsoft.ServiceBus.Samples` namespace.

```
[ServiceContract(Name = "IEchoContract", Namespace =
"http://samples.microsoft.com/ServiceModel/Relay/")]
public interface IEchoContract
{
    [OperationContract]
    string Echo(string text);
}

public interface IEchoChannel : IEchoContract, IClientChannel
{ }
```

11. Press F6 to build the client.

Example

Description

The following code shows the current status of the Program.cs file in the **EchoClient** project.

Code

```
using System;

using Microsoft.ServiceBus;

using System.ServiceModel;


namespace Microsoft.ServiceBus.Samples

{
```

```
[ServiceContract(Name = "IEchoContract", Namespace =
"http://samples.microsoft.com/ServiceModel/Relay/")]
    public interface IEchoContract
    {
        [OperationContract]
        string Echo(string text);
    }

    public interface IEchoChannel : IEchoContract, IClientChannel { }


    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Comments

Now that the interface is created, proceed to Step 6: Configure the WCF Client to implement the interface.

Step 6: Configure the WCF Client

This is the sixth of seven tasks required to create a basic Service Bus service and a client that calls the service. For an overview of all seven tasks, see Service Bus Relayed Messaging Tutorial. This topic describes how to create an App.config file for a basic client application that accesses the service created previously in this tutorial. This App.config file defines the contract, binding, and name of the endpoint. The code used for these tasks is provided in the example following the procedure.

Time to completion: 10 minutes

To configure the client

1.

Note

Steps 1 and 2 are not necessary if you are using Visual Studio 2010 , because by default, the App.config file is already present in the project.

Right-click the client project, select **Add**, **New Item**.

2. In the **Add New Item** dialog, in the **Templates** pane, select **Application Configuration File**. Then click **Add**.
3. In **Solution Explorer**, in the client project, double-click **App.config** to open the file, which currently contains the following XML elements:

```
<?xml version="1.0"?>
<configuration>
<startup><supportedRuntime version="v4.0"
sku=".NETFramework,Version=v4.0"/></startup></configuration>
```

4. Add an XML element to the App.config file for `system.serviceModel`.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.serviceModel>


</system.serviceModel>


</configuration>
```

This element declares that your application uses WCF-style endpoints. As stated previously, much of the configuration of an Service Bus application is identical to a WCF application; the main difference is the location to which the configuration file points.

5. Within the `system.serviceModel` element, add a `<client>` element.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.serviceModel>
<client>


</client>
</system.serviceModel>
</configuration>
```

This step declares that you are defining a WCF-style client application.

6. Within the `client` element, define the name, contract, and binding type for the endpoint.

```
<endpoint name="RelayEndpoint"


contract="Microsoft.ServiceBus.Samples.IEchoContract"
```

```
                    binding="netTcpRelayBinding"/>
```

This step defines the name of the endpoint, the contract defined in the service, and the fact that the client application uses TCP to communicate with the Service Bus. The endpoint name will be used in the next step, to link this endpoint configuration with the service URI.

7. Directly after the `<client>` element, add the following binding extension:

```
<extensions>
<bindingExtensions>
<add name="netTcpRelayBinding"
type="Microsoft.ServiceBus.Configuration.NetTcpRelayBindingCo
llectionElement, Microsoft.ServiceBus, Version=1.5.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
</bindingExtensions>
</extensions>
```

8. Click **File**. Then click **Save All**.

Example

Description

The following code sample shows the App.config file for the Echo client.

Code

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.serviceModel>
<client>
<endpoint name="RelayEndpoint"
                      contract="Microsoft.ServiceBus.Samples.IEchoContract"
                      binding="netTcpRelayBinding"/>
</client>
<extensions>
<bindingExtensions>
<add name="netTcpRelayBinding"
type="Microsoft.ServiceBus.Configuration.NetTcpRelayBindingCollectionElement,
Microsoft.ServiceBus, Version=1.5.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"
/>
</bindingExtensions>
```

```
</extensions>
</system.serviceModel>
</configuration>
```

Comments

Now that you have configured the client application, proceed to Step 7: Implement WCF Client to Call the Service Bus to implement the rest of the application.

Step 7: Implement WCF Client to Call the Service Bus

This is the last of seven tasks required to create a basic Service Bus service and a client that can call the service. For an overview of all seven of the tasks, see Service Bus Relayed Messaging Tutorial.

This topic describes how to implement a basic client application that accesses the service you created previously in this tutorial. Similar to the service, the client performs many of the same operations to access the Service Bus:

1. Sets the connectivity mode.
2. Creates the URI that locates the host service.
3. Defines the security credentials.
4. Applies the credentials to the connection.
5. Opens the connection.
6. Performs the application-specific tasks.
7. Closes the connection.

However, one of the main differences is that the client application uses a channel to connect to the Service Bus, whereas the service uses a call to **ServiceHost**. The code used for these tasks is provided in the example following the procedure.

Time to completion: 15 minutes

To implement a client application

1. Set the connectivity mode to `AutoDetect`.

   Add the following code inside the `Main()` method of the client application.

   ```
   ServiceBusEnvironment.SystemConnectivity.Mode =
   ConnectivityMode.AutoDetect;
   ```

2. Define variables to hold the values for the service namespace, issuer name, and issuer secret that are read from the console.

   ```
   Console.Write("Your Service Namespace: ");
   string serviceNamespace = Console.ReadLine();
   ```

```
Console.Write("Your Issuer Name: ");
string issuerName = Console.ReadLine();
Console.Write("Your Issuer Secret: ");
string issuerSecret = Console.ReadLine();
```

3. Create the URI that defines the location of the host in your Service Bus project.

```
Uri serviceUri = ServiceBusEnvironment.CreateServiceUri("sb",
serviceNamespace, "EchoService");
```

4. Create the credential object for your service namespace endpoint.

```
TransportClientEndpointBehavior
sharedSecretServiceBusCredential = new
TransportClientEndpointBehavior();
sharedSecretServiceBusCredential.TokenProvider =
TokenProvider.CreateSharedSecretTokenProvider(issuerName,
issuerSecret);
```

5. Create the channel factory that loads the configuration described in the App.config file.

```
ChannelFactory<IEchoChannel> channelFactory = new
ChannelFactory<IEchoChannel>("RelayEndpoint", new
EndpointAddress(serviceUri));
```

   A channel factory is a WCF object that creates a channel through which the service and client applications communicate.

6. Apply the Service Bus credentials.

```
channelFactory.Endpoint.Behaviors.Add(sharedSecretServiceBusC
redential);
```

7. Create and open the channel to the service.

```
IEchoChannel channel = channelFactory.CreateChannel();
channel.Open();
```

8. Write the basic user interface and functionality for the echo.

```
Console.WriteLine("Enter text to echo (or [Enter] to
exit):");
string input = Console.ReadLine();
while (input != String.Empty)
{
    try
```

```
        {
            Console.WriteLine("Server echoed: {0}",
    channel.Echo(input));
        }
        catch (Exception e)
        {
            Console.WriteLine("Error: " + e.Message);
        }
        input = Console.ReadLine();
    }
```

Note that the code uses the instance of the channel object as a proxy for the service.

9. Close the channel, and close the factory.

```
channel.Close();
channelFactory.Close();
```

To run the client application

1. Press F6 to build the solution.

   This builds both the client project and the service project that you created in a previous step of this tutorial and creating an executable file for each.

2. Before running the client application, make sure that the service application is running.

   You should now have an executable file for the Echo service application named EchoService.exe, located under your service project folder at \bin\Debug\EchoService.exe (for the debug configuration) or \bin\Release\EchoService.exe (for the release configuration). Double-click this file to start the service application.

3. A console window opens and prompts you for the service namespace. In this console window, enter the service namespace and press ENTER.
4. Next, you are prompted for your issuer name. Enter the issuer name and press ENTER.
5. After entering your issuer name, enter the issuer secret and press ENTER.

   Here is an example output from the console window. Note that the values provided here are for example purposes only.

```
Your Service Namespace: myNamespace


Your Issuer Name: owner


Your Issuer Secret: 1deCBMEhx/RV3bgwIhCohqdtzj/ZG2WnyC1cLhHTpk4=
```

The service application starts and prints the address it is listening on to the console window as seen in the following example.

```
Service address: sb://mynamespace.servicebus.windows.net/EchoService/


Press [Enter] to exit
```

6. Run the client application.

   You should now have an executable for the Echo client application named EchoClient.exe that is located under the client project directory at .\bin\Debug\EchoClient.exe (for the debug configuration) or .\bin\Release\EchoClient.exe (for the release configuration). Double-click this file to start the client application.
7. A console window opens and prompts you for the same information that you entered previously for the service application. Follow the previous steps to enter the same values for the client application for the service namespace, issuer name, and issuer secret.
8. After entering these values, the client opens a channel to the service and prompts you to enter some text as seen in the following console output example.

```
Enter text to echo (or [Enter] to exit):
```

   Enter some text to send to the service application and press ENTER.

   This text is sent to the service through the `Echo` service operation and appears in the service console window as in the following example output.

```
Echoing: My sample text
```

   The client application receives the return value of the `Echo` operation, which is the original text, and prints it to its console window. The following is an example output from the client console window.

```
Server echoed: My sample text
```

9. You can continue sending text messages from the client to the service in this manner. When you are finished, press ENTER in the client and service console windows to end both applications.

Example

Description

The following example shows how to create a client application, how to call the operations of the service, and how to close the client after the operation call is finished.

Code

```
using System;

using Microsoft.ServiceBus;

using System.ServiceModel;
```

```
namespace Microsoft.ServiceBus.Samples
{
    [ServiceContract(Name = "IEchoContract", Namespace =
"http://samples.microsoft.com/ServiceModel/Relay/")]
    public interface IEchoContract
    {
        [OperationContract]
        String Echo(string text);
    }

    public interface IEchoChannel : IEchoContract, IClientChannel { }

    class Program
    {
        static void Main(string[] args)
        {
            ServiceBusEnvironment.SystemConnectivity.Mode = ConnectivityMode.AutoDetect;


            Console.Write("Your Service Namespace: ");
            string serviceNamespace = Console.ReadLine();
            Console.Write("Your Issuer Name: ");
            string issuerName = Console.ReadLine();
            Console.Write("Your Issuer Secret: ");
            string issuerSecret = Console.ReadLine();

            Uri serviceUri = ServiceBusEnvironment.CreateServiceUri("sb",
serviceNamespace, "EchoService");

            TransportClientEndpointBehavior sharedSecretServiceBusCredential = new
TransportClientEndpointBehavior();
            sharedSecretServiceBusCredential.TokenProvider =
TokenProvider.CreateSharedSecretTokenProvider(issuerName, issuerSecret);
```

```
            ChannelFactory<IEchoChannel> channelFactory = new
ChannelFactory<IEchoChannel>("RelayEndpoint", new EndpointAddress(serviceUri));

            channelFactory.Endpoint.Behaviors.Add(sharedSecretServiceBusCredential);

            IEchoChannel channel = channelFactory.CreateChannel();
            channel.Open();

            Console.WriteLine("Enter text to echo (or [Enter] to exit):");
            string input = Console.ReadLine();
            while (input != String.Empty)
            {
                try
                {
                    Console.WriteLine("Server echoed: {0}", channel.Echo(input));
                }
                catch (Exception e)
                {
                    Console.WriteLine("Error: " + e.Message);
                }
                input = Console.ReadLine();
            }

            channel.Close();
            channelFactory.Close();

        }
    }
}
```

Comments

Ensure that the service is running before you start the client. For more information, see Step 4: Host and Run a Basic Web Service to Register with Service Bus.

Service Bus Brokered Messaging Tutorials

This section contains two tutorials that use the Service Bus brokered messaging pattern. The tutorials cover both the managed API (.NET) and REST programming models.

In This Section

Service Bus Brokered Messaging .NET Tutorial

Service Bus Brokered Messaging REST Tutorial

Service Bus Brokered Messaging .NET Tutorial

The Windows Azure Service Bus provides two comprehensive messaging solutions – one, through a centralized “relay” service running in the cloud that supports a variety of different transport protocols and Web services standards, including SOAP, WS-*, and REST. The client does not need a direct connection to the on-premises service nor does it need to know where the service resides, and the on-premises service does not need any inbound ports open on the firewall.

The second messaging solution, new in the latest release of the Service Bus, enables “brokered” messaging capabilities. These can be thought of as asynchronous, or decoupled messaging features that support publish-subscribe, temporal decoupling, and load balancing scenarios using the Service Bus messaging infrastructure. Decoupled communication has many advantages; for example, clients and servers can connect as needed and perform their operations in an asynchronous fashion.

The topics in this section are intended to give you an overview and hands-on experience with one of the core components of the brokered messaging capabilities of the Service Bus, a feature called *Queues*. After you work through the sequence of topics in this tutorial, you will have an application that populates a list of messages, creates a queue, and sends messages to that queue. Finally, the application receives and displays the messages from the queue, then cleans up its resources and exits. For a corresponding tutorial that describes how to build an application that uses the Service Bus “relayed” messaging capabilities, see the Service Bus Relayed Messaging Tutorial.

In This Section

See Also

Service Bus Relayed Messaging Tutorial

Step 1: Introduction and Prerequsites

Queues offer First In, First Out (FIFO) message delivery to one or more competing consumers. That is, messages are typically expected to be received and processed by the receivers in the temporal order in which they were enqueued, and each message will be received and processed by only one message consumer. A key benefit of using queues is to achieve “temporal decoupling” of application components: in other words, the producers and consumers do not need to be sending and receiving messages at the same time, since messages are stored durably in the queue. A related benefit is “load leveling”, which enables producers and consumers to send and receive messages at different rates.

The following are some administrative and prerequisite steps you should follow before beginning the tutorial. The first is to create a Windows Azure service namespace, and to obtain a *shared secret* key. A service namespace provides an application boundary for each application exposed through the Service Bus. A shared secret key is automatically generated by the system when a service namespace is created. The combination of service namespace and shared secret key provides a credential for the Service Bus to authenticate access to an application.

To create a service namespace and obtain a shared secret key

1. For information about how to do this, follow the steps outlined in How to: Create or Modify a Service Bus Service Namespace. You will use the shared secret key later in this tutorial.

The next step is to create a Visual Studio 2010 project and write two helper functions that load a comma-delimited list of messages into a strongly-typed (**Microsoft.ServiceBus.Messaging.BrokeredMessage**) .NET Framework **System.Collections.Generic.List** object.

To create a Visual Studio 2010 project

1. Open Visual Studio 2010 as an administrator by right-clicking the program in the **Start** menu and selecting **Run as administrator**.
2. Create a new console application project. Click the **File** menu and select **New**, then click **Project**. In the **New Project** dialog, click **Visual C#** (if **Visual C#** does not appear, look

under **Other Languages**), click the **Console Application** template, and name it **QueueSample**. Use the default **Location**. Click **OK** to create the project.

3. In the **Solution Explorer**, right-click the name of your project (in this example, **QueueSample**), and click **Properties**.
4. Click the **Application** tab on the left, then select **.NET Framework 4** from the **Target framework:** dropdown. Click **Yes** when prompted to reload the project.
5. Add references to the Microsoft.ServiceBus, System.Runtime.Serialization, and System.ServiceModel assemblies:
   a. In the **Solution Explorer**, right-click the **References** folder under the project folder and then click **Add Reference…**.
   b. Select the **.NET** tab in the **Add Reference** dialog and scroll down until you see **Microsoft.ServiceBus**, select it, and then click **OK**.
   c. Repeat the above step for **System.Runtime.Serialization** and **System.ServiceModel**.
6. In the **Solution Explorer**, double-click the Program.cs file to open it in the Visual Studio editor. Change the namespace name from its default name of `QueueSample` to `Microsoft.ServiceBus.Samples`.

```
namespace Microsoft.ServiceBus.Samples
{
    …
```

7. Add `using` statements for the `Microsoft.ServiceBus`, `Microsoft.ServiceBus.Messaging` , `Microsoft.ServiceBus.Description`, `System.IO`, and `System.Data` namespaces.

```
using Microsoft.ServiceBus;
using Microsoft.ServiceBus.Messaging;
using Microsoft.ServiceBus.Description;
using System.Data;
using System.IO;
```

8. Create a text file named Data.csv, and copy in the following comma-delimited text.

```
IssueID,IssueTitle,CustomerID,CategoryID,SupportPackage,Prior
ity,Severity,Resolved
1,Package lost,1,1,Basic,5,1,FALSE
2,Package damaged,1,1,Basic,5,1,FALSE
3,Product defective,1,2,Premium,5,2,FALSE
4,Product damaged,2,2,Premium,5,2,FALSE
5,Package lost,2,2,Basic,5,2,TRUE
6,Package lost,3,2,Basic,5,2,FALSE
7,Package damaged,3,7,Premium,5,3,FALSE
8,Product defective,3,2,Premium,5,3,FALSE
```

```
9,Product damaged,4,6,Premium,5,3,TRUE
10,Package lost,4,8,Basic,5,3,FALSE
11,Package damaged,5,4,Basic,5,4,FALSE
12,Product defective,5,4,Basic,5,4,FALSE
13,Package lost,6,8,Basic,5,4,FALSE
14,Package damaged,6,7,Premium,5,5,FALSE
15,Product defective,6,2,Premium,5,5,FALSE
```

Save and close the Data.csv file, and remember the location to which you saved it.

9. In the **Solution Explorer**, right-click the name of your project (in this example, **QueueSample**), click **Add**, then click **Existing Item**.
10. Browse to the Data.csv file that you created in step 6. Click the file, then click **Add**. You may need to ensure that **All Files (*.*)** is selected in the file type dropdown.

To create a function that parses a list of messages

1. Before the `Main()` method, declare two variables: one of type **DataTable**, to contain the list of messages in Data.csv. The other should be of type **List** object, strongly typed to **Microsoft.ServiceBus.Messaging.BrokeredMessage**. The latter is the list of brokered messages that subsequent steps in the tutorial will use.

```
namespace Microsoft.ServiceBus.Samples
{
    public class Program
    {

        private static DataTable issues;
        private static List<BrokeredMessage> MessageList;
```

2. Outside the `Main()` method, define a ParseCSV() method that parses the list of messages in Data.csv and loads the messages into a **System.Data.DataTable** table, as shown here. The method returns a **DataTable** object.

```
static DataTable ParseCSVFile()
{
    DataTable tableIssues = new DataTable("Issues");
    string path = @"..\..\data.csv";
    try
```

```
        {
            using (StreamReader readFile = new
    StreamReader(path))
            {
                string line;
                string[] row;

                // create the columns
                line = readFile.ReadLine();
                foreach (string columnTitle in line.Split(','))
                {
                    tableIssues.Columns.Add(columnTitle);
                }

                while ((line = readFile.ReadLine()) != null)
                {
                    row = line.Split(',');
                    tableIssues.Rows.Add(row);
                }
            }
        }
        catch (Exception e)
        {
            Console.WriteLine("Error:" + e.ToString());
        }

        return tableIssues;
    }
```

3. In the `Main()` method, add a statement that calls the `ParseCSVFile()` method:

```
    public static void Main(string[] args)
    {

        // Populate test data
        issues = ParseCSVFile();
```

```
}
```

To create a function that loads the list of messages

1. Outside the `Main()` method, define a GenerateMessages() method that takes the DataTable object returned by ParseCSVFile() and loads the table into a strongly-typed list of brokered messages. The method then returns the **List** object. For example:

```
static List<BrokeredMessage> GenerateMessages(DataTable
issues)
{
    // Instantiate the brokered list object
    List<BrokeredMessage> result = new
List<BrokeredMessage>();

    // Iterate through the table and create a brokered
message for each row
    foreach (DataRow item in issues.Rows)
    {
        BrokeredMessage message = new BrokeredMessage();
        foreach (DataColumn property in issues.Columns)
        {
            message.Properties.Add(property.ColumnName,
item[property]);
        }
        result.Add(message);
    }
    return result;
}
```

2. In the `Main()` method, directly below the call to `ParseCSVFile()`, add a statement that calls the `GenerateMessages()` method with the return value from `ParseCSVFile()` as an argument:

```
public static void Main(string[] args)
{

    // Populate test data
    issues = ParseCSVFile();
```

```
        MessageList = GenerateMessages(issues);
    }
```

To obtain user credentials

1. First, create three global string variables to hold these values. Declare these variables directly after the previous variable declarations, for example:

```
namespace Microsoft.ServiceBus.Samples
{
    public class Program
    {

        private static DataTable issues;
        private static List<BrokeredMessage> MessageList;
        // add these variables
        private static string ServiceNamespace;
        private static string IssuerName;
        private static string IssuerKey;
        …
```

2. Next, create a function that accepts and stores the service namespace, issuer name, and issuer key. Add this method outside `Main()`. For example:.

```
static void CollectUserInput()
{
    // User service namespace
    Console.Write("Please enter the service namespace to use:
");
    ServiceNamespace = Console.ReadLine();

    // Issuer name
    Console.Write("Please enter the issuer name to use: ");
    IssuerName = Console.ReadLine();

    // Issuer key
    Console.Write("Please enter the issuer key to use: ");
    IssuerKey = Console.ReadLine();
}
```

3. In the `Main()` method, directly below the call to `GenerateMessages()`, add a statement that calls the `CollectUserInput()` method:

```
public static void Main(string[] args)
{

    // Populate test data
    issues = ParseCSVFile();
    MessageList = GenerateMessages(issues);

    // Collect user input
    CollectUserInput();
}
```

Compiling the Code

- From the **Build** menu in Visual Studio, select **Build Solution** or press **F6** to confirm the accuracy of your work so far.

Step 2: Create Management Credentials

This is the second step in the Service Bus messaging features tutorial. In this step, you define the management operations you will use to create secure (shared secret) credentials with which your application will be authorized.

To create management credentials

1. For clarity, this tutorial places all the queue operations in a separate method. Create a `Queue()` method in the `Program` class, below the `Main()` method. For example:

```
public static void Main(string[] args)
{
…}
static void Queue()
{
}
```

2. The next step is to create a shared secret credential using a **Microsoft.ServiceBus.TokenProvider** object. The creation method takes the issuer name and key that was obtained in the `CollectUserInput()` method. Add the following code to the `Queue()` method:

```
static void Queue()
```

```
        {
            // Create management credentials
            TokenProvider credentials =
        TokenProvider.CreateSharedSecretTokenProvider(IssuerName,
        IssuerKey);
        }
```

To create the service namespace manager

1. Create a new service namespace management object, with a URI containing the service namespace name and the management credentials obtained in the last step, as arguments. Add this code directly beneath the code added in the previous step:

```
NamespaceManager namespaceClient = new
NamespaceManager(ServiceBusEnvironment.CreateServiceUri("sb",
ServiceNamespace, string.Empty), credentials);
```

Example

Description

At this point, your code should look similar to the following:

Code

```
namespace Microsoft.ServiceBus.Samples
{
    public class Program
    {

        private static DataTable issues;
        private static List<BrokeredMessage> MessageList;
        private static string ServiceNamespace;
        private static string IssuerName;
        private static string IssuerKey;

        public static void Main(string[] args)
        {
            // Collect user input
            CollectUserInput();
```

```
            // Populate test data
            issues = ParseCSVFile();
            MessageList = GenerateMessages(issues);

        }

        static void Queue()
        {

            // Create management credentials
                TokenProvider credentials =
TokenProvider.CreateSharedSecretTokenProvider(IssuerName, IssuerKey);
            // Create namespace client
            NamespaceManager namespaceClient = new
NamespaceManager(ServiceBusEnvironment.CreateServiceUri("sb", ServiceNamespace,
string.Empty), credentials);
        }

        static void CollectUserInput()
        {
            // User service namespace
            Console.Write("Please provide the service namespace to use: ");
            ServiceNamespace = Console.ReadLine();

            // Issuer name
            Console.Write("Please provide the issuer name to use: ");
            IssuerName = Console.ReadLine();

            // Issuer key
            Console.Write("Please provide the issuer key to use: ");
            IssuerKey = Console.ReadLine();
        }
```

```
static List<BrokeredMessage> GenerateMessages(DataTable issues)
 {
     // Instantiate the brokered list object
     List<BrokeredMessage> result = new List<BrokeredMessage>();

     // Iterate through the table and create a brokered message for each row
     foreach (DataRow item in issues.Rows)
     {
         BrokeredMessage message = new BrokeredMessage();
         foreach (DataColumn property in issues.Columns)
         {
             message.Properties.Add(property.ColumnName, item[property]);
         }
         result.Add(message);
     }
     return result;
 }

 static DataTable ParseCSVFile()
 {
     DataTable tableIssues = new DataTable("Issues");
     string path = @"..\..\data.csv";
     using (StreamReader readFile = new StreamReader(path))
     {
         string line;
         string[] row;

         // create the columns
         line = readFile.ReadLine();
         foreach (string columnTitle in line.Split(','))
         {
             tableIssues.Columns.Add(columnTitle);
         }
```

```
                while ((line = readFile.ReadLine()) != null)
                {
                    row = line.Split(',');
                    tableIssues.Rows.Add(row);
                }
            }

        return tableIssues;
    }
}
}
```

Comments

In the next step, you create the queue to which you will send messages.

Compiling the Code

- From the **Build** menu in Visual Studio, select **Build Solution** or press **F6** to confirm the accuracy of your work so far.

Step 3: Send Messages to the Queue

This is the third step in the Service Bus messaging features tutorial. In this step, you create the queue, then send the messages contained in the list of brokered messages to the queue.

To create and send messages to the queue

1. First, create the queue. For example, call it `myQueue`, and declare it directly after the management operations you added in the last step:

   ```
   QueueDescription myQueue;
   myQueue = namespaceClient.CreateQueue("IssueTrackingQueue");
   ```

2. In the `Queue()` method, create a messaging factory object with a newly-created Service Bus URI as an argument. Add the following code directly after the management operations you added in the last step:

   ```
   MessagingFactory factory =
   MessagingFactory.Create(ServiceBusEnvironment.CreateServiceUr
   i("sb", ServiceNamespace, string.Empty), credentials);
   ```

3. Next, create the queue object using the **Microsoft.ServiceBus.Messaging.QueueClient** class. Add the following code directly after the code you added in the last step:

   ```
   QueueClient myQueueClient =
   factory.CreateQueueClient("IssueTrackingQueue");
   ```

4. Next, add code that loops through the list of brokered messages you created and

populated in Step 1 of the tutorial, sending each to the queue. Add the following code directly after the `CreateQueueClient()` statement in the previous step:

```
// Send messages
Console.WriteLine("Now sending messages to the Queue.");
for (int count = 0; count < 6; count++)
{
    var issue = MessageList[count];
    issue.Label = issue.Properties["IssueTitle"].ToString();
    myQueueClient.Send(issue);
    Console.WriteLine(string.Format("Message sent: {0}, {1}",
issue.Label, issue.MessageId));
}
```

Step 4: Receive Messages from the Queue

This is the fourth step in the Service Bus messaging features tutorial. In this step, you obtain the list of messages from the queue you created in the previous step.

To create a receiver and receive messages from the queue

1. In the `Queue()` method, iterate through the queue and receive the messages using the **Microsoft.ServiceBus.Messaging.MessageReceiver.Receive** method, printing out each message to the console. Add the following code directly beneath the code you added in the previous step:

```
Console.WriteLine("Now receiving messages from Queue.");
BrokeredMessage message;
while ((message = myQueueClient.Receive(new TimeSpan(hours:
0, minutes: 0, seconds: 5))) != null)
    {
        Console.WriteLine(string.Format("Message received:
{0}, {1}, {2}", message.SequenceNumber, message.Label,
message.MessageId));
        message.Complete();

        Console.WriteLine("Processing message
(sleeping...)");
        Thread.Sleep(1000);
```

```
}
```

To end the Queue() method and clean up resources

1. Directly beneath the previously added code, add the following code to clean up the message factory and queue resources:

```
factory.Close();
myQueueClient.Close();
namespaceClient.DeleteQueue("IssueTrackingQueue");
```

To call the Queue() method

1. The last step is to add a statement that calls the `Queue()` method from the `Main()` method. Add the following highlighted line of code at the end of `Main()`:

```
public static void Main(string[] args)
{
    // Collect user input
    CollectUserInput();

    // Populate test data
    issues = ParseCSVFile();
    MessageList = GenerateMessages(issues);

    // Add this call
    Queue();
}
```

Example

Description

The following code contains the complete QueueSample application.

Code

```
using System;
using System.Threading;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.ServiceBus;
using Microsoft.ServiceBus.Messaging;
```

```
using Microsoft.ServiceBus.Description;
using System.Data;
using System.IO;

namespace Microsoft.ServiceBus.Samples
{
    class Program
    {

        private static DataTable issues;
        private static List<BrokeredMessage> MessageList;

        private static string ServiceNamespace;
        private static string IssuerName;
        private static string IssuerKey;



public static void Main(string[] args)
{

    // Populate test data
    issues = ParseCSVFile();

       MessageList = GenerateMessages(issues);

       CollectUserInput();

       Queue();



}
        static DataTable ParseCSVFile()
        {
            DataTable tableIssues = new DataTable("Issues");
```

```
    string path = @"..\..\data.csv";
    try
    {
        using (StreamReader readFile = new StreamReader(path))
        {
            string line;
            string[] row;

            // create the columns
            line = readFile.ReadLine();
            foreach (string columnTitle in line.Split(','))
            {
                tableIssues.Columns.Add(columnTitle);
            }

            while ((line = readFile.ReadLine()) != null)
            {
                row = line.Split(',');
                tableIssues.Rows.Add(row);
            }
        }
    }
    catch (Exception e)
    {
        Console.WriteLine("Error:" + e.ToString());
    }

    return tableIssues;
}

static List<BrokeredMessage> GenerateMessages(DataTable issues)
{
    // Instantiate the brokered list object
    List<BrokeredMessage> result = new List<BrokeredMessage>();
```

```
            // Iterate through the table and create a brokered message for each row
            foreach (DataRow item in issues.Rows)
            {
                BrokeredMessage message = new BrokeredMessage();
                foreach (DataColumn property in issues.Columns)
                {
                    message.Properties.Add(property.ColumnName, item[property]);
                }
                result.Add(message);
            }
            return result;
        }

        static void CollectUserInput()
        {
            // User service namespace
            Console.Write("Please enter the service namespace to use: ");
            ServiceNamespace = Console.ReadLine();

            // Issuer name
            Console.Write("Please enter the issuer name to use: ");
            IssuerName = Console.ReadLine();

            // Issuer key
            Console.Write("Please enter the issuer key to use: ");
            IssuerKey = Console.ReadLine();
        }


            static void Queue()
{
            // Create management credentials
    TokenProvider credentials =
```

```
                TokenProvider.CreateSharedSecretTokenProvider(IssuerName, IssuerKey);

                NamespaceManager namespaceClient = new
NamespaceManager(ServiceBusEnvironment.CreateServiceUri("sb", ServiceNamespace,
string.Empty), credentials);

                //QueueDescription myQueue;
                //myQueue = namespaceClient.CreateQueue("IssueTrackingQueue");
                namespaceClient.CreateQueue("IssueTrackingQueue");

                MessagingFactory factory =
MessagingFactory.Create(ServiceBusEnvironment.CreateServiceUri("sb", ServiceNamespace,
string.Empty), credentials);

                QueueClient myQueueClient =
factory.CreateQueueClient("IssueTrackingQueue");

                // Create a sender
                //MessageSender myMessageSender = myQueueClient.CreateSender();

                // Send messages
                Console.WriteLine("Now sending messages to the Queue.");
                for (int count = 0; count < 6; count++)
                {
                    var issue = MessageList[count];
                    issue.Label = issue.Properties["IssueTitle"].ToString();
                    myQueueClient.Send(issue);
                    Console.WriteLine(string.Format("Message sent: {0}, {1}",
issue.Label, issue.MessageId));
                }

                Console.WriteLine("Now receiving messages from Queue.");
                BrokeredMessage message;
                while ((message = myQueueClient.Receive(new TimeSpan(hours: 0, minutes:
0, seconds: 5))) != null)
```

```
                {
                    Console.WriteLine(string.Format("Message received: {0}, {1}, {2}",
message.SequenceNumber, message.Label, message.MessageId));
                    message.Complete();

                    Console.WriteLine("Processing message (sleeping...)");
                    Thread.Sleep(1000);
                }

                factory.Close();
                myQueueClient.Close();
                namespaceClient.DeleteQueue("IssueTrackingQueue");
}

    }

}
```

Step 5: Build and Run the QueueSample Application

This is the fifth step in the Service Bus messaging features tutorial. Now that you have completed the preceding steps, you can build and run the QueueSample application.

To build the QueueSample application

1. In Visual Studio, from the **Build** menu, click **Build Solution**, or press **F6**. If you encounter errors, please verify that your code is correct based on the complete example presented at the end of Step 4 of this tutorial.

To run the QueueSample application

1. Before you run the application, you must ensure that you have created a service namespace and obtained a shared secret key, as described in Step 1: Introduction and Prerequsites.
2. Open an Internet browser and visit the Windows Azure Management Portal.
3. Click **Service Bus, Access Control & Caching** in the left-hand tree, then click **Service Bus**.
4. Click the name of the service namespace that you want to use. Then, in the right-hand pane, click **Default Key**. Note the **Default Issuer** and **Default Key**.
5. In Visual Studio, from the **Debug** menu, click **Start Debugging**, or press **F5**. When

prompted, enter the name of the service namespace, the issuer name, and the key that you obtained in step 4.

Service Bus Brokered Messaging REST Tutorial

The topics in this section show how to create a basic REST-based Windows Azure Service Bus queueing and topic/subscription service.

In This Section

Step 1: Create a Service Namespace for the REST Queue and Topic/Subscription Tutorial

This is the first of nine tasks required to create a basic REST-based Windows Azure Service Bus queue and topic/subscription service.

The first step is to create a service namespace, and to obtain a shared secret key. A service namespace provides an application boundary for each application exposed through the Service Bus. A shared secret key is automatically generated by the system when a service namespace is created. The combination of service namespace and shared secret key provides a credential for the Service Bus to authenticate access to an application.

Expected time to complete: 5 minutes

How to: Create a Service Namespace for Queues, Topics, and Subscriptions

To create a service namespace

1. For complete information about how to create a service namespace, see the topic How to: Create or Modify a Service Bus Service Namespace in the Managing Service Bus Service Namespaces section.

Step 2: Create a Console Client

This is the second of nine tasks required to create a basic REST-style queue and publication/subscription application that uses the Windows Azure Service Bus.

Service Bus queues enable you to store messages in a first-in, first-out queue. Topics and subscriptions implement a publish/subscribe pattern; you create a topic and then create one or more subscriptions associated with that topic. When messages are sent to the topic, they are immediately sent to the subscribers of that topic.

The code in this tutorial:

- Uses your service namespace, issuer name, and issuer key to contact the Windows Azure Access Control Service (ACS) to obtain a Simple Web Token (SWT) to gain access to your Service Bus service namespace resources.
- Creates a queue, sends a message to the queue, and reads the message from the queue.
- Creates a topic, a subscription to that topic, and sends and reads the message from the subscription.
- Retreives all the queue, topic, and subscription information – including subscription rules -- from the Service Bus for your service namespace.
- It then deletes the queue, topic, and subscription resources.

Because the service is a REST-style Web service, there are no special types involved, as the entire exchange involves strings. This means that the Visual Studio project must make no references other than the defaults, although if your configuration has modified the defaults, you may have to add some basic .NET Framework references to the code.

After obtaining the service namespace and credentials in step 1, the next step is to create a basic Visual Studio console application.

To create a console application

1. Open Visual Studio 2010 as an administrator by right-clicking the program in the **Start** menu and selecting **Run as administrator**.
2. Create a new console application project. Click the **File** menu and select **New**, **Project**. In the **New Project** dialog, select **Visual C#** (if **Visual C#** does not appear, look under **Other Languages**), select the **Console Application** template, and name it **Microsoft.ServiceBus.Samples**. Use the default **Location**. Click **OK** to create the project.
3. For a C# project, Visual Studio creates a file that is named `Program.cs`. This class will contain an empty method called `Main()`.This method is required for a console application project to build correctly. Therefore, you can safely leave it in the project.
4. Make sure your using statements appear as follows:

```
using System;
using System.Collections.Specialized;
using System.IO;
using System.Net;
```

```
using System.Text;
using System.Xml;
```

5. If necessary, rename the service namespace for the program from the Visual Studio default to `Microsoft.ServiceBus.Samples`.
6. Inside the `Program` class, add the following global variables:

```
static string serviceNamespace;
static string baseAddress;
static string token;
const string sbHostName = "servicebus.windows.net";
const string acsHostName = "accesscontrol.windows.net";
```

7. Inside the `Main()` method, copy the following code:

```
Console.Write("Enter your service namespace: ");
serviceNamespace = Console.ReadLine();

Console.Write("Enter your issuer name: ");
string issuerName = Console.ReadLine();

Console.Write("Enter your issuer secret: ");
string issuerSecret = Console.ReadLine();

baseAddress = "https://" + serviceNamespace + "." +
sbHostName + "/";
try
{
    // Get a SWT token from the Access Control Service, given
the issuerName and issuerSecret values.
    token = GetToken(issuerName, issuerSecret);

    string queueName = "Queue" + Guid.NewGuid().ToString();

    // Create and put a message in the queue using the SWT
token.
    CreateQueue(queueName, token);
    SendMessage(queueName, "msg1");
    string msg = ReceiveAndDeleteMessage(queueName);
```

```
    string topicName = "Topic" + Guid.NewGuid().ToString();
    string subscriptionName = "Subscription" +
Guid.NewGuid().ToString();
    CreateTopic(topicName);
    CreateSubscription(topicName, subscriptionName);
    SendMessage(topicName, "msg2");

    // Wait for messages to post:
    //System.Threading.Thread.Sleep(500);
    Console.WriteLine(ReceiveAndDeleteMessage(topicName +
"/Subscriptions/" + subscriptionName));

    // Get an Atom feed with all the queues in the namespace
    Console.WriteLine(GetResources("$Resources/Queues"));

    // Get an Atom feed with all the topics in the namespace
    Console.WriteLine(GetResources("$Resources/Topics"));

    // Get an Atom feed with all the subscriptions for the
topic we just created
    Console.WriteLine(GetResources(topicName +
"/Subscriptions"));

    // Get an Atom feed with all the rules for the topic and
subscritpion we just created
    Console.WriteLine(GetResources(topicName +
"/Subscriptions/" + subscriptionName + "/Rules"));

    // Delete the queue we created
    DeleteResource(queueName);

    // Delete the topic we created
    DeleteResource(topicName);
```

```
    // Get an Atom feed with all the topics in the namespace,
it shouldn't have the one we created now
    Console.WriteLine(GetResources("$Resources/Topics"));

    // Get an Atom feed with all the queues in the namespace,
it shouldn't have the one we created now
    Console.WriteLine(GetResources("$Resources/Queues"));
}
catch (WebException we)
{
    using (HttpWebResponse response = we.Response as
HttpWebResponse)
    {
        if (response != null)
        {
            Console.WriteLine(new
StreamReader(response.GetResponseStream()).ReadToEnd());
        }
        else
        {
            Console.WriteLine(we.ToString());
        }
    }
}

Console.WriteLine("\nPress ENTER to exit.");
Console.ReadLine();
```

Step 3: Create Management Credentials

This is the third of nine tasks required to create a basic REST-style queue and publication/subscription application that uses the Service Bus.

The next step is to write a method that processes the service namespace, issuer name, and issuer secret that you entered in the previous step, and returns a Simple Web Token (SWT).

To create a GetToken() method

1. Paste the following code after the `Main()` method in the `Program` class:

```
private static string GetToken(string issuerName, string
issuerSecret)
{
    var acsEndpoint = "https://" + serviceNamespace + "-sb."
+ acsHostName + "/WRAPv0.9/";

    // Note that the realm used when requesting a token uses
the HTTP scheme, even though
    // calls to the service are always issued over HTTPS
    var realm = "http://" + serviceNamespace + "." +
sbHostName + "/";

    NameValueCollection values = new NameValueCollection();
    values.Add("wrap_name", issuerName);
    values.Add("wrap_password", issuerSecret);
    values.Add("wrap_scope", realm);

    WebClient webClient = new WebClient();
    byte[] response = webClient.UploadValues(acsEndpoint,
values);

    string responseString =
Encoding.UTF8.GetString(response);

    var responseProperties = responseString.Split('&');
    var tokenProperty = responseProperties[0].Split('=');
    var token = Uri.UnescapeDataString(tokenProperty[1]);

    return "WRAP access_token=\"" + token + "\"";
}
```

Step 4: Create the Queue

This is the fourth of nine tasks required to create a basic REST-style queue and publication/subscription application that uses the Service Bus.

The next step is to write a method that uses the REST-style HTTP PUT command to create a queue.

To create a queue

1. Paste the following code directly beneath the `GetToken()` code you added in step 3:

```
// Uses HTTP PUT to create the queue
private static string CreateQueue(string queueName, string
token)
{
    // Create the URI of the new queue, note that this uses
the HTTPS scheme
    string queueAddress = baseAddress + queueName;
    WebClient webClient = new WebClient();
    webClient.Headers[HttpRequestHeader.Authorization] =
token;

    Console.WriteLine("\nCreating queue {0}", queueAddress);
    // Prepare the body of the create queue request
    var putData = @"<entry
xmlns=""http://www.w3.org/2005/Atom"">
<title type=""text"">" + queueName + @"</title>
<content type=""application/xml"">
<QueueDescription xmlns:i=""http://www.w3.org/2001/XMLSchema-
instance""
xmlns=""http://schemas.microsoft.com/netservices/2010/10/serv
icebus/connect"" />
</content>
</entry>";

    byte[] response = webClient.UploadData(queueAddress,
"PUT", Encoding.UTF8.GetBytes(putData));
    return Encoding.UTF8.GetString(response);
}
```

Step 5: Send a Message to the Queue

This is the fifth of nine tasks required to create a basic REST-style queue and publication/subscription application that uses the Service Bus.

In this step, you add a method that uses the REST-style HTTP POST command to send a message to the queue you created in the previous step.

To send a message to the queue

1. Paste the following code directly beneath the `CreateQueue()` code you added in step 4:

```
// Sends a message to the "queueName" queue, given the name,
the value to enqueue, and the SWT token
// Uses an HTTP POST request.
private static void SendMessage(string queueName, string
body)
{
    string fullAddress = baseAddress + queueName +
"/messages" + "?timeout=60";
    Console.WriteLine("\nSending message {0} - to address
{1}", body, fullAddress);
    WebClient webClient = new WebClient();
    webClient.Headers[HttpRequestHeader.Authorization] =
token;

    webClient.UploadData(fullAddress, "POST",
Encoding.UTF8.GetBytes(body));
}
```

Step 6: Receive a Message from the Queue

This is the sixth of nine tasks required to create a basic REST-style queue and publication/subscription application that uses the Service Bus.

The next step is to add a method that uses the REST-style HTTP DELETE command to receive and delete a message from the queue.

To receive and delete a message from the queue

1. Paste the following code directly beneath the `SendMessage()` code you added in step 5:

```
// Receives and deletes the next message from the given
resource (Queue, Topic, or Subscription)
// using the resourceName, the SWT token, and an HTTP DELETE
request.
private static string ReceiveAndDeleteMessage(string
resourceName)
{
    string fullAddress = baseAddress + resourceName +
"/messages/head" + "?timeout=60";
    Console.WriteLine("\nRetrieving message from {0}",
fullAddress);
    WebClient webClient = new WebClient();
    webClient.Headers[HttpRequestHeader.Authorization] =
token;

    byte[] response = webClient.UploadData(fullAddress,
"DELETE", new byte[0]);
    string responseStr = Encoding.UTF8.GetString(response);

    Console.WriteLine(responseStr);
    return responseStr;
}
```

Step 7: Create a Topic and Subscription

This is the seventh of nine tasks required to create a basic REST-style queue and publication/subscription application that uses the Service Bus.

The next step is to write a method that uses the REST-style HTTP PUT command to create a topic. Then, you write a method that creates a subscription to that topic.

To create a topic

1. Paste the following code directly beneath the `ReceiveAndDeleteMessage()` code you added in step 6:

```
// Creates a Topic with the given topic name and the SWT
token
```

```
// Using an HTTP PUT request.
private static string CreateTopic(string topicName)
{
    var topicAddress = baseAddress + topicName;
    WebClient webClient = new WebClient();
    webClient.Headers[HttpRequestHeader.Authorization] =
token;

    Console.WriteLine("\nCreating topic {0}", topicAddress);
    // Prepare the body of the create queue request
    var putData = @"<entry
xmlns=""http://www.w3.org/2005/Atom"">
<title type=""text"">" + topicName + @"</title>
<content type=""application/xml"">
<TopicDescription xmlns:i=""http://www.w3.org/2001/XMLSchema-
instance""
xmlns=""http://schemas.microsoft.com/netservices/2010/10/serv
icebus/connect"" />
</content>
</entry>";

    byte[] response = webClient.UploadData(topicAddress,
"PUT", Encoding.UTF8.GetBytes(putData));
    return Encoding.UTF8.GetString(response);
}
```

To create a subscription

1. The following code creates a subscription to the topic you created in the previous section. Add the following code directly beneath the `CreateTopic()` definition:

```
private static string CreateSubscription(string topicName,
string subscriptionName)
{
    var subscriptionAddress = baseAddress + topicName +
"/Subscriptions/" + subscriptionName;
    WebClient webClient = new WebClient();
    webClient.Headers[HttpRequestHeader.Authorization] =
```

```
token;

    Console.WriteLine("\nCreating subscription {0}",
subscriptionAddress);
    // Prepare the body of the create queue request
    var putData = @"<entry
xmlns=""http://www.w3.org/2005/Atom"">
<title type=""text"">" + subscriptionName + @"</title>
<content type=""application/xml"">
<SubscriptionDescription
xmlns:i=""http://www.w3.org/2001/XMLSchema-instance""
xmlns=""http://schemas.microsoft.com/netservices/2010/10/serv
icebus/connect"" />
</content>
</entry>";

    byte[] response =
webClient.UploadData(subscriptionAddress, "PUT",
Encoding.UTF8.GetBytes(putData));
    return Encoding.UTF8.GetString(response);
}
```

Step 8: Retreive Message Resources

This is the eighth of nine tasks required to create a basic REST-style queue and publication/subscription application that uses the Service Bus.

In this step, you add code that retrieves the message properties, then deletes the messaging resources you created in the previous steps.

To retrieve an Atom feed with the specified resources

1. Add the following code directly beneath the `CreateSubscription()` method you added in the previous step:

```
private static string GetResources(string resourceAddress)
{
    string fullAddress = baseAddress + resourceAddress;
    WebClient webClient = new WebClient();
```

```
    webClient.Headers[HttpRequestHeader.Authorization] =
token;
    Console.WriteLine("\nGetting resources from {0}",
fullAddress);
    return FormatXml(webClient.DownloadString(fullAddress));
}
```

To delete messaging entities

1. Add the following code directly beneath the code you added in the previous section:

```
private static string DeleteResource(string resourceName)
{
    string fullAddress = baseAddress + resourceName;
    WebClient webClient = new WebClient();
    webClient.Headers[HttpRequestHeader.Authorization] =
token;

    Console.WriteLine("\nDeleting resource at {0}",
fullAddress);
    byte[] response = webClient.UploadData(fullAddress,
"DELETE", new byte[0]);
    return Encoding.UTF8.GetString(response);
}
```

To format the Atom feed

1. The `GetResources()` method contains a call to a `FormatXml()` method that reformats the retrieved Atom feed to be more readable. The following is the definition of `FormatXml()`; add this code directly beneath the `DeleteResource()` code you added in the previous section:

```
// Formats the XML string to be more human-readable; intended
for display purposes
private static string FormatXml(string inputXml)
{
    XmlDocument document = new XmlDocument();
    document.Load(new StringReader(inputXml));

    StringBuilder builder = new StringBuilder();
```

```
        using (XmlTextWriter writer = new XmlTextWriter(new
    StringWriter(builder)))
        {
            writer.Formatting = Formatting.Indented;
            document.Save(writer);
        }

        return builder.ToString();
    }
```

Step 9: Build and Run the Application

This is the last of nine tasks required to create a basic REST-style queue and publication/subscription application that uses the Service Bus.

You can now build and run the application.

To build the application

1. From the **Build** menu in Visual Studio, click **Build Solution**, or press **F6** to confirm the accuracy of your work.

To run the application

1. If there are no errors, press **F5** to run the application. When prompted, enter your service namespace, issuer name, and default key that you obtained in Step 1.

Example

Description

The following example is the complete code, as it should appear after following steps 1 through 8:

```
using System;

using System.Collections.Specialized;

using System.IO;

using System.Net;

using System.Text;

using System.Xml;


namespace Microsoft.ServiceBus.Samples

{
```

```
    class Program
    {

        static string serviceNamespace;
        static string baseAddress;
        static string token;
        const string sbHostName = "servicebus.windows.net";
        const string acsHostName = "accesscontrol.windows.net";


        static void Main(string[] args)
        {

            Console.Write("Enter your service namespace: ");
            serviceNamespace = Console.ReadLine();

            Console.Write("Enter your issuer name: ");
            string issuerName = Console.ReadLine();

            Console.Write("Enter your issuer secret: ");
            string issuerSecret = Console.ReadLine();

            baseAddress = "https://" + serviceNamespace + "." + sbHostName + "/";
            try
            {
                // Get a SWT token from the Access Control Service, given the issuerName
and issuerSecret values.
                token = GetToken(issuerName, issuerSecret);

                string queueName = "Queue" + Guid.NewGuid().ToString();

                // Create and put a message in the queue using the SWT token.
                CreateQueue(queueName, token);
                SendMessage(queueName, "msg1");
```

```
            string msg = ReceiveAndDeleteMessage(queueName);

            string topicName = "Topic" + Guid.NewGuid().ToString();
            string subscriptionName = "Subscription" + Guid.NewGuid().ToString();
            CreateTopic(topicName);
            CreateSubscription(topicName, subscriptionName);
            SendMessage(topicName, "msg2");

            // Wait for messages to post:
            //System.Threading.Thread.Sleep(500);
            Console.WriteLine(ReceiveAndDeleteMessage(topicName + "/Subscriptions/" +
subscriptionName));

            // Get an Atom feed with all the queues in the namespace
            Console.WriteLine(GetResources("$Resources/Queues"));

            // Get an Atom feed with all the topics in the namespace
            Console.WriteLine(GetResources("$Resources/Topics"));

            // Get an Atom feed with all the subscriptions for the topic we just
created
            Console.WriteLine(GetResources(topicName + "/Subscriptions"));

            // Get an Atom feed with all the rules for the topic and subscritpion we
just created
            Console.WriteLine(GetResources(topicName + "/Subscriptions/" +
subscriptionName + "/Rules"));

            // Delete the queue we created
            DeleteResource(queueName);

            // Delete the topic we created
            DeleteResource(topicName);
```

```
                // Get an Atom feed with all the topics in the namespace, it shouldn't
have the one we created now
                Console.WriteLine(GetResources("$Resources/Topics"));

                // Get an Atom feed with all the queues in the namespace, it shouldn't
have the one we created now
                Console.WriteLine(GetResources("$Resources/Queues"));
            }
            catch (WebException we)
            {
                using (HttpWebResponse response = we.Response as HttpWebResponse)
                {
                    if (response != null)
                    {
                        Console.WriteLine(new
StreamReader(response.GetResponseStream()).ReadToEnd());
                    }
                    else
                    {
                        Console.WriteLine(we.ToString());
                    }
                }
            }

            Console.WriteLine("\nPress ENTER to exit.");
            Console.ReadLine();
        }

        private static string GetToken(string issuerName, string issuerSecret)
        {
            var acsEndpoint = "https://" + serviceNamespace + "-sb." + acsHostName +
"/WRAPv0.9/";

            // Note that the realm used when requesting a token uses the HTTP scheme,
even though
```

```
            // calls to the service are always issued over HTTPS
            var realm = "http://" + serviceNamespace + "." + sbHostName + "/";

            NameValueCollection values = new NameValueCollection();
            values.Add("wrap_name", issuerName);
            values.Add("wrap_password", issuerSecret);
            values.Add("wrap_scope", realm);

            WebClient webClient = new WebClient();
            byte[] response = webClient.UploadValues(acsEndpoint, values);

            string responseString = Encoding.UTF8.GetString(response);

            var responseProperties = responseString.Split('&');
            var tokenProperty = responseProperties[0].Split('=');
            var token = Uri.UnescapeDataString(tokenProperty[1]);

            return "WRAP access_token=\"" + token + "\"";
        }

        // Uses HTTP PUT to create the queue
        private static string CreateQueue(string queueName, string token)
        {
            // Create the URI of the new Queue, note that this uses the HTTPS scheme
            string queueAddress = baseAddress + queueName;
            WebClient webClient = new WebClient();
            webClient.Headers[HttpRequestHeader.Authorization] = token;

            Console.WriteLine("\nCreating queue {0}", queueAddress);
            // Prepare the body of the create queue request
            var putData = @"<entry xmlns=""http://www.w3.org/2005/Atom"">
<title type=""text"">" + queueName + @"</title>
<content type=""application/xml"">
```

```
<QueueDescription xmlns:i=""http://www.w3.org/2001/XMLSchema-instance""
xmlns=""http://schemas.microsoft.com/netservices/2010/10/servicebus/connect"" />
</content>
</entry>";

            byte[] response = webClient.UploadData(queueAddress, "PUT",
Encoding.UTF8.GetBytes(putData));
            return Encoding.UTF8.GetString(response);
        }

        // Sends a message to the "queueName" queue, given the name, the value to
enqueue, and the SWT token
        // Uses an HTTP POST request.
        private static void SendMessage(string queueName, string body)
        {
            string fullAddress = baseAddress + queueName + "/messages" + "?timeout=60";
            Console.WriteLine("\nSending message {0} - to address {1}", body,
fullAddress);
            WebClient webClient = new WebClient();
            webClient.Headers[HttpRequestHeader.Authorization] = token;

            webClient.UploadData(fullAddress, "POST", Encoding.UTF8.GetBytes(body));
        }

        // Receives and deletes the next message from the given resource (Queue, Topic,
or Subscription)
        // using the resourceName, the SWT token, and an HTTP DELETE request.
        private static string ReceiveAndDeleteMessage(string resourceName)
        {
            string fullAddress = baseAddress + resourceName + "/messages/head" +
"?timeout=60";
            Console.WriteLine("\nRetrieving message from {0}", fullAddress);
            WebClient webClient = new WebClient();
            webClient.Headers[HttpRequestHeader.Authorization] = token;
```

```
            byte[] response = webClient.UploadData(fullAddress, "DELETE", new byte[0]);
            string responseStr = Encoding.UTF8.GetString(response);

            Console.WriteLine(responseStr);
            return responseStr;
        }

        // Creates a Topic with the given topic name and the SWT token
        // Using an HTTP PUT request.
        private static string CreateTopic(string topicName)
        {
            var topicAddress = baseAddress + topicName;
            WebClient webClient = new WebClient();
            webClient.Headers[HttpRequestHeader.Authorization] = token;

            Console.WriteLine("\nCreating topic {0}", topicAddress);
            // Prepare the body of the create queue request
            var putData = @"<entry xmlns=""http://www.w3.org/2005/Atom"">
<title type=""text"">" + topicName + @"</title>
<content type=""application/xml"">
<TopicDescription xmlns:i=""http://www.w3.org/2001/XMLSchema-instance""
xmlns=""http://schemas.microsoft.com/netservices/2010/10/servicebus/connect"" />
</content>
</entry>";

            byte[] response = webClient.UploadData(topicAddress, "PUT",
Encoding.UTF8.GetBytes(putData));
            return Encoding.UTF8.GetString(response);
        }

        private static string CreateSubscription(string topicName, string
subscriptionName)
        {
```

```
            var subscriptionAddress = baseAddress + topicName + "/Subscriptions/" +
subscriptionName;
            WebClient webClient = new WebClient();
            webClient.Headers[HttpRequestHeader.Authorization] = token;

            Console.WriteLine("\nCreating subscription {0}", subscriptionAddress);
            // Prepare the body of the create queue request
            var putData = @"<entry xmlns=""http://www.w3.org/2005/Atom"">
<title type=""text"">" + subscriptionName + @"</title>
<content type=""application/xml"">
<SubscriptionDescription xmlns:i=""http://www.w3.org/2001/XMLSchema-instance""
xmlns=""http://schemas.microsoft.com/netservices/2010/10/servicebus/connect"" />
</content>
</entry>";

            byte[] response = webClient.UploadData(subscriptionAddress, "PUT",
Encoding.UTF8.GetBytes(putData));
            return Encoding.UTF8.GetString(response);
        }

        private static string GetResources(string resourceAddress)
        {
            string fullAddress = baseAddress + resourceAddress;
            WebClient webClient = new WebClient();
            webClient.Headers[HttpRequestHeader.Authorization] = token;
            Console.WriteLine("\nGetting resources from {0}", fullAddress);
            return FormatXml(webClient.DownloadString(fullAddress));
        }

        private static string DeleteResource(string resourceName)
        {
            string fullAddress = baseAddress + resourceName;
            WebClient webClient = new WebClient();
            webClient.Headers[HttpRequestHeader.Authorization] = token;
```

```
            Console.WriteLine("\nDeleting resource at {0}", fullAddress);
            byte[] response = webClient.UploadData(fullAddress, "DELETE", new byte[0]);
            return Encoding.UTF8.GetString(response);
        }

        // Formats the XML string to be more human-readable; intended for display
purposes
        private static string FormatXml(string inputXml)
        {
            XmlDocument document = new XmlDocument();
            document.Load(new StringReader(inputXml));

            StringBuilder builder = new StringBuilder();
            using (XmlTextWriter writer = new XmlTextWriter(new StringWriter(builder)))
            {
                writer.Formatting = Formatting.Indented;
                document.Save(writer);
            }

            return builder.ToString();
        }

    }
}
```

Service Bus Message Buffer Tutorial

The following topics describe how to build a simple Service Bus host application that exposes a REST-based interface. A Web client, such as a Web browser, can access the Service Bus service API through HTTP requests.

This tutorial uses the Windows Communication Foundation (WCF) REST programming model to construct a REST service on the Service Bus. For more information, see **WCF REST Programming Model** and **Designing and Implementing Services** in the WCF documentation.

In This Section

Step 1: Sign up for an Account for the REST Tutorial

This is the first of four tasks required to create a basic REST-based Windows Azure Service Bus service. For an overview of all four of the tasks, see the Service Bus Message Buffer Tutorial.

The first step is to create a Windows Azure service namespace, and to obtain a *shared secret* key. A service namespace provides an application boundary for each application exposed through the Service Bus. A shared secret key is automatically generated by the system when a service namespace is created. The combination of service namespace and shared secret key provides a credential for the Service Bus to authenticate access to an application.

Expected time to complete: 5 minutes

To create a service namespace

1. To create a service namespace by using the Windows Azure portal, follow the steps in How to: Create or Modify a Service Bus Service Namespace.

Step 2: Define a REST-based WCF Service Contract to use with Service Bus

This is the second of four tasks required to create a basic REST-style service for the Service Bus. For an overview of all four of the tasks, see the Service Bus Message Buffer Tutorial.

As with other Service Bus services, when you create a REST-style service, you must define the contract. The contract specifies what operations the host supports. A service operation can be thought of as a Web service method. Contracts are created by defining a C++, C#, or Visual Basic interface. Each method in the interface corresponds to a specific service operation. The **System.ServiceModel.ServiceContractAttribute** attribute must be applied to each interface, and the **System.ServiceModel.OperationContractAttribute** attribute must be applied to each operation. If a method in an interface that has the **System.ServiceModel.ServiceContractAttribute** does not have the **System.ServiceModel.OperationContractAttribute**, that method is not exposed. The code used for these tasks is shown in the example following the procedure.

The primary difference between a basic Service Bus contract and a REST-style contract is the addition of a property to the **System.ServiceModel.OperationContractAttribute**: **System.ServiceModel.Web.WebGetAttribute**. This property lets you map a method in your interface to a method on the other side of the interface. In this case, we will use

System.ServiceModel.Web.WebGetAttribute to link a method to HTTP GET. This allows the Service Bus to accurately retrieve and interpret commands sent to the interface.

Expected time to completion: 10 minutes.

To create a Service Bus contract with an interface

1. Open **Visual Studio 2008** as an administrator by right-clicking the program in the **Start** menu and selecting **Run as administrator**.
2. Create a new console application project. Click the **File** menu and select **New**, **Project**. In the **New Project** dialog, select **Visual C#** (if **Visual C#** does not appear, look under **Other Languages**), select the **Console Application** template, and name it **ImageListener**. Use the default **Location**. Click **OK** to create the project.
3. For a C# project, Visual Studio creates a file that is named `Program.cs`. This class will contain an empty method called `Main()`.This method is required for a console application project to build correctly. Therefore, you can safely leave it in the project.
4. Add a reference to `System.ServiceModel.dll` to the project:
   a. In the **Solution Explorer**, right-click the **References** folder under the project folder and then click **Add Reference**.
   b. Select the **.NET** tab in the **Add Reference** dialog and scroll down until you see **System.ServiceModel**, select it. Then click **OK**.

   Note
   When using a command-line compiler (for example, Csc.exe), you must also provide the path of the assemblies. By default, on a computer that is running Windows°7 for example, the path is: Windows\Microsoft.NET\Framework\v3.0\Windows Communication Foundation.
5. Repeat the previous step to add a reference to the `System.ServiceModel.Web.dll` assembly.
6. Add a `using` statement for the `System.ServiceModel`, `System.ServiceModel.Channels`, `System.ServiceModel.Web`, and `System.IO` namespaces.

```
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Web;
using System.IO;
```

System.ServiceModel is the namespace that lets you programmatically access the basic features of the Windows Communication Foundation (WCF). The Service Bus uses many of the objects and attributes of WCF to define service contracts. You will use this namespace in most of your Service Bus applications. Similarly, **System.ServiceModel.Channels** helps define the channel, which is the object through which you communicate with the Service Bus and the client Web browser. Finally, **System.ServiceModel.Web** contains the types that let you create Web-based

applications.

7. Rename the namespace for the program from the Visual Studio default to `Microsoft.ServiceBus.Samples`.

```
namespace Microsoft.ServiceBus.Samples
{
...
```

8. Directly after the namespace declaration, define a new interface named `IImageContract` and apply the `ServiceContractAttribute` attribute to the interface with a value of **http://samples.microsoft.com/ServiceModel/Relay/**. The namespace value differs from the namespace that you use throughout the scope of your code. The namespace value is used as a unique identifier for this contract, and should have versioning information. For more information, see, see Service Versioning. Specifying the namespace explicitly prevents the default namespace value from being added to the contract name.

```
[ServiceContract(Name = "ImageContract", Namespace =
"http://samples.microsoft.com/ServiceModel/Relay/RESTTutorial
1")]
public interface IImageContract
{
}
```

9. Within the `IImageContract` interface, declare a method for the single operation the `IImageContract` contract exposes in the interface and apply the `OperationContractAttribute` attribute to the method that you want to expose as part of the public Service Bus contract.

```
public interface IImageContract
{
    [OperationContract]
    Stream GetImage();
}
```

10. Next to the `OperationContract` attribute, apply the `WebGet` attribute.

```
public interface IImageContract
{
    [OperationContract, WebGet]
    Stream GetImage();
}
```

Doing so allows the Service Bus to route HTTP GET requests to **GetImage**, and to translate the return values of **GetImage** into an HTTP GETRESPONSE reply. Later in the tutorial, you will use a Web browser to access this method, and to display the image in the browser.

11. Directly underneath the `IImageContract` definition, declare a channel that inherits from both the `IImageContract` and `IClientChannel` interfaces.

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

public interface IImageChannel : IImageContract,
IClientChannel { }
```

    A channel is the WCF object through which the service and client pass each other information. Later on, you will create the channel in your host application. The Service Bus then uses this channel to pass the HTTP GET requests from the browser to your `GetImage` implementation. The Service Bus also uses the channel to take the `GetImage` return value and translate it into an HTTP GETRESPONSE for the client browser.

12. From the **Build** menu, click **Build Solution** to confirm the accuracy of your work.

Example

Description

The following code example shows a basic interface that defines an Service Bus contract.

Code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Web;
using System.IO;
```

```
namespace Microsoft.ServiceBus.Samples
{

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

    public interface IImageChannel : IImageContract, IClientChannel { }

    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Comments

Now that the interface is created, proceed to Step 3: Implement a REST-based WCF Service Contract to use Service Bus to implement the interface.

See Also

Step 3: Implement a REST-based WCF Service Contract to use Service Bus

Step 3: Implement a REST-based WCF Service Contract to use Service Bus

This is the third of four tasks required to create a basic REST-style Service Bus service. For an overview of all tasks, see the Service Bus Message Buffer Tutorial topic. Creating a REST-style Service Bus service requires that you first create the contract, which is defined by using an interface. For more information about creating the interface, see Step 2: Define a REST-based WCF Service Contract to use with Service Bus. The next step, shown in this example, is to implement the interface. This involves creating a class named `ImageService` that implements the

user-defined `IImageContract` interface. After you implement the contract, you then configure the interface using an App.config file. The configuration file contains necessary information for the application, such as the name of the service, the name of the contract, and the type of protocol that is used to communicate with the Service Bus. The code used for these tasks is provided in the example following the procedure.

As with the previous steps, there is very little difference between implementing a REST-style contract and a basic Service Bus contract.

Expected time to completion: 10 minutes

To implement a REST-style Service Bus contract

1. Create a new class named `ImageService` directly underneath the definition of the `IImageContract` interface. The `ImageService` class implements the `IImageContract` interface.

   ```
   class ImageService : IImageContract
   {
   }
   ```

   Similar to other interface implementations, you can implement the definition in a different file. However, for this tutorial, the implementation appears in the same file as the interface definition and `Main()` method.
2. Apply the **System.ServiceModel.ServiceBehaviorAttribute** attribute to the `IImageService` class to indicate that the class is an implementation of a WCF contract:

   ```
   [ServiceBehavior(Name = "ImageService", Namespace =
   "http://samples.microsoft.com/ServiceModel/Relay/")]
   class ImageService : IImageContract
   {
   }
   ```

   As mentioned previously, this namespace is not a traditional namespace. Instead, it is part of the WCF architecture that identifies the contract. For more information, see the **Data Contract Names** topic in the WCF documentation.
3. Add a .jpg image to your project.

   This is a picture the service displays in the receiving browser. Right-click your project, click **Add**. Then click **Existing Item**. Use the **Add Existing Item** dialog to browse to an appropriate .jpg, and then click **Add**. An example .jpg file is available at `<SDKInstallDir>\Samples\ServiceBus\ExploringFeatures\Bindings\WebHttp\CS35\Service\image.jpg`.

   When adding the file, make sure that **All Files (*.*)** is selected in the drop-down list next to the **File name:** field. The rest of this tutorial assumes that the name of the image is

"image.jpg". If you have a different .jpg, you will have to rename the image, or change your code to compensate.

4. To make sure that the running service can find the image file, in **Solution Explorer** right-click the image file. In the **Properties** pane, set **Copy to Output Directory** to **Copy if newer**.
5. Add references to the `System.Drawing.dll`, `System.Runtime.Serialization.dll`, and `Microsoft.ServiceBus.dll` assemblies to the project, and also to the following associated `using` statements.

```
using System.Drawing;
using System.Drawing.Imaging;
using Microsoft.ServiceBus;
using Microsoft.ServiceBus.Web;
```

6. Define a constructor that loads the bitmap and prepares to send it to the client browser:

```
class ImageService : IImageContract
{
    const string imageFileName = "image.jpg";

    Image bitmap;

    public ImageService()
    {
        this.bitmap = Image.FromFile(imageFileName);
    }
}
```

7. Directly underneath the previous code, add the following `GetImage` method in the `ImageService` class to return an HTTP message that contains the image:

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

   stream.Position = 0;
   WebOperationContext.Current.OutgoingResponse.ContentType =
```

```
"image/jpeg";

    return stream;
}
```

This implementation uses **MemoryStream** to retrieve the image and prepare it for streaming to the browser. It starts the stream position at zero, declares the stream content as a jpeg, and streams the information.

8. From the **Build** menu, click **Build Solution** to build the whole solution.

To define the configuration to run the Web service on the Service Bus

1. Right-click the **ImageListener** project. Then click **Add**, **New Item**.
2. In the **Add New Item** dialog, in the **Templates** pane, select **Application Configuration**. Then click **Add**.

   The configuration file resembles a WCF configuration file, and includes the service name, endpoint (that is, the location Service Bus exposes for clients and hosts to communicate with each other), and binding (the type of protocol that is used to communicate). The main difference here is that the configured service endpoint refers to a **Microsoft.ServiceBus.WebHttpRelayBinding** binding, which is not part of the .NET Framework. **Microsoft.ServiceBus.WebHttpRelayBinding** is one of the new bindings introduced with the Service Bus. For more information about how to configure an Service Bus application, see Configuring a WCF Service to Register with the Service Bus.
3. In **Solution Explorer**, click **App.config**, which currently contains the following XML elements:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
</configuration>
```

4. Add an XML element to the App.config file for `system.serviceModel`. This is a WCF element that defines one or more services. Here, it is used to define the service name and endpoint.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.serviceModel>


</system.serviceModel>


</configuration>
```

5. Within the `system.serviceModel` element, add a `<bindings>` element that has the following content. This defines the bindings used in the application. You can define multiple

bindings, but for this tutorial you are defining only one.

```
<bindings>
<!-- Application Binding -->
<webHttpRelayBinding>
<binding name="default">
<security relayClientAuthenticationType="None" />
</binding>
</webHttpRelayBinding>
</bindings>
```

This step defines an Service Bus **Microsoft.ServiceBus.WebHttpRelayBinding** binding with the `relayClientAuthenticationType` as `None`. This indicates that an endpoint using this binding will not require a client credential.

6. Below the `<bindings>` element, add a `<services>` element. As with the bindings, you can define multiple services in a single configuration file. However, for this tutorial, you define only one.

```
<services>
<!-- Application Service -->
<service name="Microsoft.ServiceBus.Samples.ImageService"
           behaviorConfiguration="default">
<endpoint name="RelayEndpoint"

contract="Microsoft.ServiceBus.Samples.IImageContract"
              binding="webHttpRelayBinding"
              bindingConfiguration="default"

behaviorConfiguration="sharedSecretClientCredentials"
              address="" />
</service>
</services>
```

This step configures a service that uses the previously defined default `webHttpRelayBinding`. It also uses the default `sharedSecretClientCredentials`, which is defined in the next step.

7. Below the `<services>` element, create a `<behaviors>` element, with the following content, replacing `"ISSUER_NAME"` and `"ISSUER_SECRET"` with your issuer name and secret, respectively.

```
<behaviors>
```

```
<endpointBehaviors>
<behavior name="sharedSecretClientCredentials">
<transportClientEndpointBehavior
credentialType="SharedSecret">
<clientCredentials>
<sharedSecret issuerName="ISSUER_NAME"
issuerSecret="ISSUER_SECRET" />
</clientCredentials>
</transportClientEndpointBehavior>
</behavior>
</endpointBehaviors>
<serviceBehaviors>
<behavior name="default">
<serviceDebug httpHelpPageEnabled="false"
httpsHelpPageEnabled="false" />
</behavior>
</serviceBehaviors>
</behaviors>
```

The `sharedSecretClientCredentials` behavior defines the type of credentials the service uses to access the Service Bus: `SharedSecret`. In addition, the actual issuer names and issuer secrets are stored in the App.config file. Note that storing secrets in clear text is not considered good programming practice for production code. Be sure to implement more rigorous security in your own code.

This code also defines the default debugging behavior, which consists of turning off the HTTP and HTTPS help pages.

8. From the **Build** menu, select **Build Solution** to build the whole solution.

Example

Description

The following code shows the contract and service implementation for a REST-based service that is running on the Service Bus using the **WebHttpRelayBinding** binding.

Code

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Web;
using System.IO;
using System.Drawing;
using System.Drawing.Imaging;
using Microsoft.ServiceBus;
using Microsoft.ServiceBus.Web;

namespace Microsoft.ServiceBus.Samples
{



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

    public interface IImageChannel : IImageContract, IClientChannel { }

    [ServiceBehavior(Name = "ImageService", Namespace =
"http://samples.microsoft.com/ServiceModel/Relay/")]
    class ImageService : IImageContract
    {
        const string imageFileName = "image.jpg";

        Image bitmap;
```

```
        public ImageService()
        {
            this.bitmap = Image.FromFile(imageFileName);
        }

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

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

            return stream;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Example

Description

The following example shows the App.config file associated with the service.

Code

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.serviceModel>
<bindings>
```

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

<services>
<!-- Application Service -->
<service name="Microsoft.ServiceBus.Samples.ImageService"
               behaviorConfiguration="default">
<endpoint name="RelayEndpoint"
                  contract="Microsoft.ServiceBus.Samples.IImageContract"
                  binding="webHttpRelayBinding"
                  bindingConfiguration="default"
                  behaviorConfiguration="sharedSecretClientCredentials"
                  address="" />
</service>
</services>

<behaviors>
<endpointBehaviors>
<behavior name="sharedSecretClientCredentials">
<transportClientEndpointBehavior credentialType="SharedSecret">
<clientCredentials>
<sharedSecret issuerName="ISSUER_NAME" issuerSecret="ISSUER_SECRET" />
</clientCredentials>
</transportClientEndpointBehavior>
</behavior>
</endpointBehaviors>
<serviceBehaviors>
```

```
<behavior name="default">
<serviceDebug httpHelpPageEnabled="false" httpsHelpPageEnabled="false" />
</behavior>
</serviceBehaviors>
</behaviors>


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

Comments

Now that you have configured and implemented the Web service contract, proceed to Step 4: Host the REST-based WCF Service to use the Service Bus.

Step 4: Host the REST-based WCF Service to use the Service Bus

This is the fourth of four tasks required to create a basic REST-based Service Bus service. For an overview of all four of the tasks, see the Service Bus Message Buffer Tutorial topic. This topic describes how to run a Web service on Service Bus using a console application. A complete listing of the code written in this task is provided in the example following the procedure.

Estimated time to completion: 10 minutes

To create a base address for the service

1. In the `Main()` function declaration, create a variable to store the service namespace of your Service Bus project.

   ```
   string serviceNamespace = "InsertServiceNamespaceHere";
   ```

   The Service Bus uses the name of your service namespace to create a unique URI.
2. Create a `Uri` instance for the base address of the service that is based on the service namespace.

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

To create and configure the Web service host

1. Create the Web service host, using the URI address created earlier in this section.

   ```
   WebServiceHost host = new
   WebServiceHost(typeof(ImageService), address);
   ```

   The service host is the WCF object that instantiates the host application. This example passes it the type of host you want to create (an `ImageService`), and also the address at which you want to expose the host application.

To run the Web service host

1. Open the service.

   ```
   host.Open();
   ```

   The service is now running.
2. Display a message indicating that the service is running, and how to stop the service.

   ```
   Console.WriteLine("Copy the following address into a browser
   to see the image: ");
   Console.WriteLine(address + "GetImage");
   Console.WriteLine();
   Console.WriteLine("Press [Enter] to exit");
   Console.ReadLine();
   ```
3. When finished, close the service host.

   ```
   host.Close();
   ```

Example

Description

The following example includes the service contract and implementation from previous steps in the tutorial and hosts the service in a console application. Compile the following into an executable named ImageListener.exe..

Code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.ServiceModel.Web;
using System.IO;
using System.Drawing;
using System.Drawing.Imaging;
using Microsoft.ServiceBus;
using Microsoft.ServiceBus.Web;
```

```
namespace Microsoft.ServiceBus.Samples
{

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

    public interface IImageChannel : IImageContract, IClientChannel { }

    [ServiceBehavior(Name = "ImageService", Namespace =
"http://samples.microsoft.com/ServiceModel/Relay/")]
    class ImageService : IImageContract
    {
        const string imageFileName = "image.jpg";

        Image bitmap;

        public ImageService()
        {
            this.bitmap = Image.FromFile(imageFileName);
        }

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

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

```
            return stream;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            string serviceNamespace = "InsertServiceNamespaceHere";
            Uri address = ServiceBusEnvironment.CreateServiceUri("https",
serviceNamespace, "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 + "GetImage");
            Console.WriteLine();
            Console.WriteLine("Press [Enter] to exit");
            Console.ReadLine();

            host.Close();
        }
    }
}
```

Compiling the Code

After building the solution, do the following to run the application:

1. From a command prompt, run the service (ImageListener\bin\Debug\ImageListener.exe).
2. Copy and paste the address from the command prompt into a browser to see the image.

Developing Applications that Use the Service Bus

The Windows Azure Service Bus can be thought of as a “relay in the sky” that enables two applications to communicate securely regardless of where they may be located. At the highest conceptual level, using the Service Bus requires:

1. One Web service that is trusted by the Access Control service to create endpoints and receive and send messages (typically responses but also event notifications) from the Service Bus endpoint that it creates.
2. One client application that is trusted by the Access Control service to send and receive messages (typically requests and responses but also event notifications) at a Service Bus endpoint.

There is only one difference between the two Web service applications: the first Web service is trusted by Access Control to create an endpoint—that is, an address, or Uniform Resource Indicator (URI)—with Service Bus. For more information about endpoints and addresses and how they are used in Windows Communication Foundation (WCF) and the Service Bus, see **Specifying an Endpoint Address**. The second Web application (the client), however, cannot create, affect, or manage the registered Service Bus endpoint; instead, it is trusted by Access Control to interact with endpoints that are already registered.

Typically, the former type of application is referred to as a *service* (instead of *client* or calling application), because without a controlling or managing Web service, the Service Bus would have no endpoints with which other Web-enabled applications can communicate. Web applications that interact with a pre-existing Service Bus endpoint are conventionally referred to as Web service client applications, because they consume the features available at a registered Service Bus endpoint.

Overview

This topic provides an overview of the technical features that the Service Bus provides at a high level. The following additional topics in this section describe application development with the Service Bus from the point of view of a development lifecycle. Not every development goal requires starting at any one point in the cycle; if you have to develop a client that invokes a service published by the Service Bus, you do not have to read about publishing a service endpoint. Instead, start with Building a Service Bus Client Application.

This section contains the following topics, starting with a development lifecycle outline that describes the steps in the context of your own development processes.

- Overview of Service Bus Messaging Patterns
- Service Bus Programming Lifecycle
- Service Bus Authentication and Authorization with the Access Control Service
- Service Bus Bindings
- Designing a WCF Contract for the Service Bus

- Configuring a WCF Service to Register with the Service Bus
- Securing and Authenticating a Service Bus Connection
- Building a Service for the Service Bus
- Building a Service Bus Client Application
- Discovering and Exposing a Service Bus Service
- Working with a Service Bus Message Buffer

What Do I Need to Build to Use the Service Bus?

There must be a host application for the Web service that registers an endpoint with the Service Bus, and a client application must make requests on the Service Bus endpoint. Typically the following host application types are used:

- .NET Framework applications
- Windows Azure applications
- Scripts in both server-side and client-side Web pages
- Smart devices that have either SOAP or REST networking programming models

What Types of Applications Can Use the Service Bus?

Fundamentally, you can use any HTTP programming model to use REST-style messages to register an endpoint with the Service Bus or to use a Service Bus endpoint that has already been published. However, it is often easier to understand how to use the Access Control service and Service Bus in the context of a technology with which you are familiar.

- **If you are familiar with WCF applications**, you can use this service to obtain permission from Access Control to create (or register) an endpoint address that uses the Service Bus. Subsequently, any SOAP or REST client—whether WCF or built on a non-Microsoft platform—can use the WCF Web service by invoking operations on the Service Bus address. Depending on the security requirements established by the original Web service when it registered the endpoint with the Service Bus, the client application may also have to obtain permission from Access Control to send or listen for messages by using the Service Bus.
- **If you are familiar with Windows Azure applications** with WCF-based services and clients, you will develop the host application locally and then run it in Windows Azure. This is true even if you use .NET Framework HTTP programming to do REST-style Web service and client communication.
- **If you are familiar with REST-based Web services** and you can use either a WCF SOAP-based service to authorize your application and use the Service Bus, or you can use a REST-based application to do this. The WCF Web Programming Model makes it easy to build a REST service that does this, but you can do your REST-based communication with the Service Bus any way you want. In fact, if you use REST on the Web service side and the client side, you can use the Message Buffer as a temporary storage of messages that can be retrieved by callers.

How Much Can I Do with the Service Bus?

The Service Bus is made to enable bidirectional communication between service-oriented applications anywhere in the world. However, the real world is full of limits, and the Service Bus does not support every protocol you might ever want or need. For example, in this release, the system-supplied bindings support only a subset of the protocols supported by WCF. What happens when you want to use a protocol that is not supported by the system-supplied bindings? The answer is that you can either implement a WCF custom binding that does support your required protocol – or you can create a bridge between any two endpoints that enables a bidirectional stream. As the Service Bus securely exposes service contracts if the contract specifies a two-way stream, you can then host that service on the Service Bus and exchange binary data – that is, any custom protocol – using the Service Bus. You can even associate ports on either side with the binary stream, which enables pre-existing applications to communicate through the Service Bus using their own proprietary protocol. Finally, you are doing this secured by tokens from Access Control and through firewalls and NAT routers.

Understanding the Windows Azure Management Portal

Building an application that uses the Service Bus is straightforward; your applications must obtain a security token from the Access Control service, pass that to the Service Bus in order to obtain permission to register endpoints and send or receive messages, and start the service, the client, or both. Of course, when you are building your applications, you must decide upon a design for your service (or client) and you will also consider which bindings to use depending on the network environment and application needs. However, the basic development process is that of building a WCF SOAP or REST application, and configuring it to use the Service Bus.

If you are familiar with building applications that use SOAP- or REST-based communication, the only new information is how to interact with the Windows Azure platform management Web site to obtain or perform the following.

- A Live ID to access the Windows Azure Web portal.
- Acquire and configure the appropriate permissions from the Access Control service.
- Create projects and namespaces using the Service Bus service by using the tokens obtained in the first item earlier in this section.

Accessing the Windows Azure Web Portal

To create your account and obtain or manage the necessary tokens, projects, and namespaces, you must have a Windows Live ID. (To start that process, see **http://go.microsoft.com/fwlink/?LinkID=129428**.)

Choosing Namespaces

Once that has been accomplished, you must create a project and create some namespaces that scope the work that you will be doing. Namespaces are scoping mechanisms, just as they are in .NET programming or in their use in XML.

The namespaces you create must be unique across all Service Bus and Access Control accounts; when you create a namespace in a Windows Azure project, you are declaring a base or root namespace that is owned and secured by you. That root namespace is the namespace that is used to manage security tokens in the Access Control service, and it is a namespace under which you can register any number of services related to your work. (Note that you can declare multiple namespaces in a project; each one is completely isolated from the others for all work that you may do.) For example, if you use the Windows Azure portal to declare a namespace of `contoso-samples`, the Service Bus creates the resources that you must have in order to host, secure, and bill for services underneath the complete namespace URI <protocol scheme>`://contoso-samples.servicebus.windows.net/` (where protocol scheme is in the end either **sb**, **http**, or **https**, depending on which protocol schemes your endpoints require when you publish them).

The important things to note about the namespace created are as follows:

- It is location and transport independent. Knowing a namespace provides no information about the transport used or about the location or type of service endpoint that is registered with the Service Bus. (You can publish service metadata to all clients so that they can discover your functionality; but you must take that step yourself. By default, this information is kept private.)
- Creating service names underneath the namespace is a way of partitioning data and functionality that makes sense to your work environment.

An example can illustrate the second point. Given a registered namespace of `contoso-samples` (again, the fully qualified namespace is `contoso-samples.servicebus.windows.net`), the following endpoint URIs have names that indicate logical geographical divisions in your company in order to expose the same structural functionality but secured differently depending on local considerations:

sb://contoso-samples.servicebus.windows.net/redmond
sb://contoso-samples.servicebus.windows.net/paris
sb://contoso-samples.servicebus.windows.net/tokyo

The root namespace of **contoso-samples.servicebus.windows.net** can then be used with the Access Control service to provide secure tokens that enable interaction with one or more of the local services as is appropriate for the business requirements. The previous example demonstrates the use of namespaces and endpoints to partition functionality by geographical location. However, you can use namespaces and names to partition functionality and data by any logical category you want: by company division, by geographical location, by role, or anything else. (If you take the extra step to integrate the Access Control service together with Active Directory Federated Services or any other custom authentication and authorization system, you can integrate your connected application together with your pre-existing authorization system.)

Discovering Services

By default, services registered with the Service Bus are private. However, you can configure the Service Bus to make your endpoints public when you register them. The Service Bus exposes public endpoints in a registry published in an ATOM 1.0 feed that callers can discover by browsing the root namespace URI in a Web browser. For more information about how to declare

that a registered endpoint is to be published as part of the namespace ATOM feed, see How to: Publish a Service to the Service Bus Registry.

High Level Lifecycle Roadmap

At the highest level, the standard Service Bus development lifecycle looks as follows. (For a different view of the development lifecycle based more on technological approach than workflow, see Service Bus Programming Lifecycle.)

1. Create an account that uses the Service Bus and establish a namespace and endpoint. For more information, see How to: Create or Modify a Service Bus Service Namespace.
2. There must be a Web service that is running to register an endpoint with the Service Bus. If you have one built, go to the next step. If not, build one. For more information, seeDesigning a WCF Contract for the Service Bus. (For more information about building services that use WCF, see Designing and Implementing Services in WCF)
3. There are two main mechanisms to register and host endpoints with the Service Bus: Use WCF and SOAP with the Windows Azure Service Bus SDK, or use REST-style HTTP Web requests. In the latter case, you can use the WCF REST Programming Model, .NET HTTP programming, or any other REST programming platform on any device.
    a. Using WCF and SOAP.
        i. Design and implement your WCF service.
        ii. Choose and implement a Host. You can be a local .NET application, a Windows Azure application, or an HTTP application.
        iii. Configure the local WCF service to register itself with the Service Bus. For more information, seeConfiguring a WCF Service to Register with the Service Bus
        iv. Start the host application.
        v. Build any clients. See step 4.
    b. Using any REST-enabled HTTP programming platform. This includes the WCF REST Programming Model.
        i. Obtain the authorization token from the Access Service and create a configuration that represents the message buffer parameters. For more information, seeHow to: Configure a Service Bus Message Buffer
        ii. Connect to the Service Bus and use your configuration values to create a message buffer. For more information, seeHow to: Create and Connect to a Service Bus Message Buffer.
        iii. Send messages to the message buffer. For more information, seeHow to: Send Messages to a Service Bus Message Buffer.
        iv. Retrieve messages from the message buffer. For more information, seeHow to: Retrieve a Message from a Service Bus Message Buffer.
4. Create or modify a client application. For more information, seeBuilding a Service Bus Client Application.

Overview of Service Bus Messaging Patterns

This section contains information about the different types of messaging patterns supported by the Windows Azure Service Bus.

In This Section

Relayed and Brokered Messaging

Describes the new “brokered” messaging features and how they differ from the relayed messaging pattern of earlier Service Bus releases.

Queues, Topics, and Subscriptions

Describes the new queues, topics/subscriptions, rules, and filtering features of the Service Bus.

Naming and Registry

An overview of the Service Bus service namespace naming system and service registry.

Relayed and Brokered Messaging

The messaging pattern associated with the initial releases of the Windows Azure Service Bus is referred to as *relayed* messaging. The latest version of the Service Bus adds another type of messaging option known as *brokered* messaging. The brokered messaging scheme can also be thought of as asynchronous messaging.

Relayed Messaging

The central component of the Service Bus is a centralized (but highly load-balanced) relay service that supports a variety of different transport protocols and Web services standards. This includes SOAP, WS-*, and even REST. The relay service provides a variety of different relay connectivity options and can even help negotiate direct peer-to-peer connections when it is possible. The Service Bus is optimized for .NET developers who use the Windows Communication Foundation (WCF), both with regard to performance and usability, and provides full access to its relay service through SOAP and REST interfaces. This makes it possible for any SOAP or REST programming environment to integrate with it.

The relay service supports traditional one-way messaging, request/response messaging, and peer-to-peer messaging. It also supports event distribution at Internet-scope to enable publish/subscribe scenarios and bi-directional socket communication for increased point-to-point efficiency. In the relayed messaging pattern, an on-premise service connects to the relay service through an outbound port and creates a bi-directional socket for communication tied to a

particular rendezvous address. The client can then communicate with the on-premises service by sending messages to the relay service targeting the rendezvous address. The relay service will then “relay” messages to the on-premises service through the bi-directional socket already in place. The client does not need a direct connection to the on-premises service nor is it required to know where the service resides, and the on-premises service does not need any inbound ports open on the firewall.

You must initiate the connection between your on-premise service and the relay service, using a suite of WCF “relay” bindings. Behind the scenes, the relay bindings map to new transport binding elements designed to create WCF channel components that integrate with the Service Bus in the cloud.

Relayed messaging provides many benefits, but requires the server and client to both be online at the same time in order to send and receive messages. This is not optimal for HTTP-style communication, in which the requests may not be typically long lived, nor for clients that connect only occasionally, such as browsers, mobile applications, and so on. Brokered messaging supports decoupled communication, and has its own advantages; clients and servers can connect when needed and perform their operations in an asynchronous manner.

Brokered Messaging

In contrast to the relayed messaging scheme, brokered messaging can be thought of as asynchronous, or “temporally decoupled.” Producers (senders) and consumers (receivers) do not have to be online at the same time. The messaging infrastructure reliably stores messages until the consuming party is ready to receive them. This allows the components of the distributed application to be disconnected, either voluntarily; for example, for maintenance, or due to a component crash, without affecting the whole system. Furthermore, the receiving application may only have to come online during certain times of the day, such as an inventory management system that only is required to run at the end of the business day.

The core components of theService Bus brokered messaging infrastructure are Queues, Topics, and Subscriptions. These components enable new asynchronous messaging scenarios, such as temporal decoupling, publish/subscribe, and load balancing. For more information about these structures, see the next section.

As with the relayed messaging infrastructure, the brokered messaging capability is provided for WCF and .NET Framework programmers and also via REST.

Queues, Topics, and Subscriptions

The new release of the Windows Azure Service Bus adds a set of cloud-based, message-oriented-middleware technologies including reliable message queuing and durable publish/subscribe messaging. These “brokered” messaging capabilities can be thought of as asynchronous, or decoupled messaging features that support publish-subscribe, temporal decoupling, and load balancing scenarios using the Service Bus messaging fabric. Decoupled communication has many advantages; for example, clients and servers can connect as needed and perform their operations in an asynchronous fashion.

There are three messaging patterns that form the core of the new brokered messaging capabilities in the Service Bus: *Queues*, *Topics/Subscriptions*, and *Rules/Actions*.

Queues

Queues offer First In, First Out (FIFO) message delivery to one or more competing consumers. That is, messages are typically expected to be received and processed by the receivers in the temporal order in which they were added to the queue, and each message is received and processed by only one message consumer. A key benefit of using queues is to achieve “temporal decoupling” of application components. In other words, the producers (senders) and consumers (receivers) do not have to be sending and receiving messages at the same time, because messages are stored durably in the queue. Furthermore, the producer does not have to wait for a reply from the consumer in order to continue to process and send messages.

A related benefit is “load leveling,” which enables producers and consumers to send and receive messages at different rates. In many applications, the system load varies over time; however, the processing time required for each unit of work is typically constant. Intermediating message producers and consumers with a queue means that the consuming application only has to be provisioned to be able to handle average load instead of peak load. The depth of the queue will grow and contract as the incoming load varies. This directly saves money with regard to the amount of infrastructure required to service the application load. As the load increases, more worker processes can be added to read from the queue. Each message is processed by only one of the worker processes. Furthermore, this pull-based load balancing allows for optimum use of the worker computers even if the worker computers differ with regard to processing power, as they will pull messages at their own maximum rate. This pattern is often termed the “competing consumer” pattern.

Using queues to intermediate between message producers and consumers provides an inherent loose coupling between the components. Because producers and consumers are not aware of each other, a consumer can be upgraded without having any effect on the producer.

Creating a queue is a multi-step process. Management operations for Service Bus messaging entities (both queues and topics) are performed via the **Microsoft.ServiceBus.NamespaceManager** class, which is constructed by supplying the base address of the Service Bus namespace and the user credentials. **Microsoft.ServiceBus.NamespaceManager** provides methods to create, enumerate and delete messaging entities. After creating a **Microsoft.ServiceBus.Description.SharedSecretCredential** object from the issuer name and shared key, and a service namespace management object, you can use the **Microsoft.ServiceBus.NamespaceManager.CreateQueue(Microsoft.ServiceBus.Messaging.QueueDescription)** method to create the queue. For example:

```
// Create management credentials

TokenProvider credentials = TokenProvider.CreateSharedSecretTokenProvider(IssuerName,

IssuerKey);

// Create namespace client
```

```
namespaceManager namespaceClient = new
namespaceManager(ServiceBusEnvironment.CreateServiceUri("sb", ServiceNamespace,
string.Empty), credentials);
```

You can then create a queue object and a messaging factory with the Service Bus URI as an argument. For example:

```
QueueDescription myQueue;
myQueue = namespaceClient.CreateQueue("TestQueue");
MessagingFactory factory =
MessagingFactory.Create(ServiceBusEnvironment.CreateServiceUri("sb", ServiceNamespace,
string.Empty), credentials);
QueueClient myQueueClient = factory.CreateQueueClient("TestQueue");
```

You can then send messages to the queue. For example, if you have a list of brokered messages called `MessageList`, the code would appear similar to the following:

```
for (int count = 0; count < 6; count++)
{
    var issue = MessageList[count];
    issue.Label = issue.Properties["IssueTitle"].ToString();
    myQueueClient.Send(issue);
}
```

You can receive messages from the queue, as follows:

```
while ((message = myQueueClient.Receive(new TimeSpan(hours: 0, minutes: 0, seconds: 5)))
!= null)
    {
        Console.WriteLine(string.Format("Message received: {0}, {1}, {2}",
message.SequenceNumber, message.Label, message.MessageId));
        message.Complete();

        Console.WriteLine("Processing message (sleeping...)");
        Thread.Sleep(1000);
    }
```

In the **Microsoft.ServiceBus.Messaging.ReceiveMode.ReceiveAndDelete** mode, the receive operation is single-shot, that is, when the Service Bus receives the request, it marks the message as being consumed and returns it to the application.
Microsoft.ServiceBus.Messaging.ReceiveMode.ReceiveAndDelete mode is the simplest model and works best for scenarios in which the application can tolerate not processing a message in the event of a failure. To understand this, consider a scenario in which the consumer issues the receive request and then crashes before processing it. Since the Service Bus marks

the message as being consumed, when the application restarts and begins consuming messages again, it will have missed the message that was consumed prior to the crash.

In **Microsoft.ServiceBus.Messaging.ReceiveMode.PeekLock** mode, the receive operation becomes two-stage, which makes it possible to support applications that cannot tolerate missing messages. When the Service Bus receives the request, it finds the next message to be consumed, locks it to prevent other consumers from receiving it, and then returns it to the application. After the application finishes processing the message (or stores it reliably for future processing), it completes the second stage of the receive process by calling **Microsoft.ServiceBus.Messaging.BrokeredMessage.Complete** on the received message. When the Service Bus sees the **Microsoft.ServiceBus.Messaging.BrokeredMessage.Complete**, it will mark the message as being consumed.

If the application is unable to process the message for some reason, it can call the **Microsoft.ServiceBus.Messaging.BrokeredMessage.Abandon** method on the received message (instead of **Microsoft.ServiceBus.Messaging.BrokeredMessage.Complete**). This will cause the Service Bus to unlock the message and make it available to be received again, either by the same consumer or by another completing consumer. Secondly, there is a timeout associated with the lock and if the application fails to process the message before the lock timeout expires (for example, if the application crashes), then Service Bus will unlock the message and make it available to be received again.

Note that in the event that the application crashes after processing the message, but before the **Microsoft.ServiceBus.Messaging.BrokeredMessage.Complete** request was issued, the message will be redelivered to the application when it restarts. This is often called *At Least Once* processing; that is, each message will be processed at least once but in certain situations the same message may be redelivered. If the scenario cannot tolerate duplicate processing, then additional logic is required in the application to detect duplicates which can be achieved based upon the **MessageId** property of the message which will remain constant across delivery attempts. This is known as *Exactly Once* processing.

For more information and a working example of how to create and send messages to and from queues, see the Service Bus Brokered Messaging .NET Tutorial.

Topics and Subscriptions

In contrast to queues, in which each message is consumed by a single consumer, topics and subscriptions provide a one-to-many form of communication, in a “publish/subscribe” pattern. Useful for scaling to very large numbers of recipients, each published message is made available to each subscription registered with the topic. Messages are sent to a topic and delivered to one or more associated subscriptions, depending on filter rules that can be set on a per-subscription basis. The subscriptions can use additional filters to restrict the messages that they want to receive. Messages are sent to a topic in the same way they are sent to a queue, but messages are not received from the topic directly. Instead, they are received from subscriptions. A topic subscription resembles a virtual queue that receives copies of the messages that are sent to the

topic. Messages are received from a subscription in the identical way as they are received from a queue.

By way of comparison, the message sending functionality of a queue maps directly to a topic and its message receiving functionality to a subscription. Among other things, this means that subscriptions support the same patterns described earlier in this section with regard to queues: competing consumer, temporal decoupling, load leveling and load balancing.

Creating a topic is a process similar to creating a queue, as shown in the example in the previous section. Create the service URI, and then use the **Microsoft.ServiceBus.NamespaceManager** class to create the namespace client. You can then create a topic using the **Microsoft.ServiceBus.NamespaceManager.CreateTopic(System.String)** method. For example:

```
TopicDescription dataCollectionTopic =
namespaceClient.CreateTopic("DataCollectionTopic");
```

Next, add subscriptions as you want:

```
SubscriptionClient myAuditSubscription = factory.CreateSubscriptionClient(myTopic.Path,
"Inventory", ReceiveMode.ReceiveAndDelete);

SubscriptionClient myAgentSubscription = factory.CreateSubscriptionClient(myTopic.Path,
"Dashboard", ReceiveMode.ReceiveAndDelete);
```

You then create a topic client. For example:

```
MessagingFactory factory = MessagingFactory.Create(serviceUri, tokenProvider);

TopicClient myTopicClient = factory.CreateTopicClient(myTopic.Path)
```

Using the message sender, you can send and receive messages to and from the topic, as shown in the previous section. For example:

```
foreach (BrokeredMessage message in messageList)

{

    myTopicClient.Send(message);

    Console.WriteLine(

    string.Format("Message sent: Id = {0}, Body = {1}", message.MessageId,
message.GetBody<string>()));

}
```

Similar to queues, messages are received from a subscription using a **Microsoft.ServiceBus.Messaging.SubscriptionClient** object instead of a **Microsoft.ServiceBus.Messaging.QueueClient** object. Create the subscription client, passing the name of the topic, the name of the subscription, and (optionally) the receive mode as parameters. For example, with the **Inventory** subscription:

```
// Create the subscription client

MessagingFactory factory = MessagingFactory.Create(serviceUri, tokenProvider);
```

```
SubscriptionClient agentSubscriptionClient =
factory.CreateSubscriptionClient("IssueTrackingTopic", "Inventory",
ReceiveMode.PeekLock);
SubscriptionClient auditSubscriptionClient =
factory.CreateSubscriptionClient("IssueTrackingTopic", "Dashboard",
ReceiveMode.ReceiveAndDelete);


while ((message = agentSubscriptionClient.Receive(TimeSpan.FromSeconds(5))) != null)
{
    Console.WriteLine("\nReceiving message from Inventory...");
    Console.WriteLine(string.Format("Message received: Id = {0}, Body = {1}",
message.MessageId, message.GetBody<string>()));
    message.Complete();
}


// Create a receiver using ReceiveAndDelete mode
while ((message = auditSubscriptionClient.Receive(TimeSpan.FromSeconds(5))) != null)
{
    Console.WriteLine("\nReceiving message from Dashboard...");
    Console.WriteLine(string.Format("Message received: Id = {0}, Body = {1}",
message.MessageId, message.GetBody<string>()));
}
```

Important

As noted in the topic How to: Publish a Service to the Service Bus Registry, you can use **Microsoft.ServiceBus.ServiceRegistrySettings** to indicate whether you want your service to be discoverable on the Service Bus. If your service is private, then only individuals that know the specific URI can connect. If it is public, then anyone can navigate the Service Bus hierarchy and find your listener. However, queues, topics, and subscriptions cannot be exposed via the service registry.

Rules and Actions

In many scenarios, messages that have specific characteristics must be processed in specific ways. To enable this, you can configure subscriptions to find messages that have desired properties and then perform certain modifications to those properties. While Service Bus subscriptions see all messages sent to the topic, you can only copy a subset of those messages

to the virtual subscription queue. This is accomplished using subscription filters. Such modifications are called Filter Actions. When a subscription is created, you can supply a filter expression that can operate over the properties of the message, both the system properties (for example, **Label**) and the application properties, such as **StoreName** in the previous example. The SQL filter expression is optional in this case; without a SQL filter expression, any filter action defined on a subscription will be performed on all the messages for that subscription.

Using the previous example, to filter messages coming only from Store1, you would create the Dashboard subscription as follows:

```
namespaceManager.CreateSubscription(("Dashboard", new SqlFilter("StoreName = 'Store1'");
```

With this subscription filter in place, only messages that have the **StoreName** property set to **Store1** will be copied to the virtual queue for the **Dashboard** subscription.

For more information about possible filter values, see the documentation for the **Microsoft.ServiceBus.Messaging.SqlFilter** and **Microsoft.ServiceBus.Messaging.SqlRuleAction** classes. Also, see the **AdvancedFiltersSample** in the Windows Azure SDK.

See Also

Service Bus Brokered Messaging .NET Tutorial

Naming and Registry

DNS is designed to map domain names to IP addresses. When you browse to a Web site, the first thing that occurs is a DNS lookup that determines to what IP address the friendly domain name resolves. Since DNS relies on public IP addresses, it does not work for identifying hosts located behind NAT devices without the help of a layered service such as Dynamic DNS. It is common for a single IP address to identify a complete network of hosts located behind a single NAT device. Ultimately, the DNS model is less than ideal for naming and identifying endpoints in a service oriented world.

Unlike DNS, the Windows Azure Service Bus naming system is optimized for naming service endpoints in a host-independent manner. You can think of the naming system as a global forest of federated naming trees projected onto host-independent URIs. Each service namespace maps to a naming tree; therefore, each service namespace must have a globally unique name. The naming trees are “federated” because each service namespace owner controls the names within a service namespace. They are “trees” because of the hierarchical nature of the namespace (names within names within names). There can be a natural projection for these names onto URIs, but the resulting URIs are completely host-independent – you can have multiple services running on different hosts that share the same solution name. These characteristics of the Service Bus naming system provide a more granular, endpoint-level approach that complements DNS.

Naming System

The root of the Service Bus naming system is resolvable through traditional DNS techniques. The naming system relies on host-independent criteria – specifically the service namespace – to distinguish between different domains of control in the naming system. Service namespace owners control the names within their respective service namespaces.

You project Service Bus names onto URIs as follows:

```
[scheme]://[service-namespace].servicebus.windows.net/[name1]/[name2]/...
```

The Service Bus supports three URI schemes: “sb”, “http”, and “https”. You use “http” and “https” for all HTTP-based endpoints, and the “sb” scheme for all other TCP-based endpoints. The `[service-namespace]` part of the host name identifies a unique naming tree in the complete Service Bus namespace, which is controlled by the service namespace owner.

Note

In the current release, nesting in the URI naming scheme is not supported. For example, topics and queues cannot be nested under each other. You cannot create a topic at `https://contoso.servicebus.windows.net/HumanResources/Topic1`, then a queue at its child location: `https://contoso.servicebus.windows.net/HumanResources/Topic1/Queue1`. Conversely, you cannot create a queue at a location such as `https://contoso.servicebus.windows.net/HumanResources/Queue1`, then a topic at its child location: `https://contoso.servicebus.windows.net/HumanResources/Queue1/Topic1`.

Similarly, a relay endpoint (for example, an endpoint for **Microsoft.ServiceBus.NetTcpRelayBinding**, any Http relay binding, or a message buffer) and a messaging endpoint (for example, a queue or topic) cannot be nested under each other. You cannot have a messaging endpoint at `https://contoso.servicebus.windows.net/HumanResources/Queue1` and a relay endpoint at `https://contoso.servicebus.windows.net/HumanResources/Queue1/Relay1`.

Registry

The Service Bus provides a service registry for publishing and discovering service endpoint references in a service namespace. Others can then discover the endpoints in a service namespace by browsing to the service namespace base address and retrieving an Atom feed. The service registry exposes the service namespace endpoints through a linked tree of Atom 1.0 feeds. You navigate the service registry by navigating the naming system via HTTP, browsing to each level in the naming structure of the solution you want to inspect. When you browse to the service namespace base HTTP address, you obtain the root Atom 1.0 feed describing the first level of nested names. If you then browse to one of the nested names, you obtain another Atom 1.0 feed that describes the second level of nested names. This continues until you reach a leaf name in the tree.

The Service Bus can publish endpoint information into the registry whenever you register new endpoints. If you want a particular endpoint to be discoverable, you associate the **ServiceRegistrySettings** behavior with the Windows Communication Foundation (WCF)

endpoint, setting its **Microsoft.ServiceBus.ServiceRegistrySettings.DiscoveryMode** property to **DiscoveryType.Public**. The following code shows how to do this in the WCF host application:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("**** Service ****");
        ServiceHost host = new ServiceHost(typeof(myExample));
        host.Open();

        ServiceRegistrySettings settings = new ServiceRegistrySettings();
        settings.DiscoveryMode = DiscoveryType.Public;
        foreach(ServiceEndpoint se in host.Description.Endpoints)
            se.Behaviors.Add(settings);

        Console.WriteLine("Press [Enter] to exit");
        Console.ReadLine();

        host.Close();
    }
}
```

With this behavior, the relay service automatically populates the service registry with information about the endpoint in question.

Note
Queues, Topics, and Subscriptions are not discoverable in the service registry.

Service Bus Programming Lifecycle

The following topics describe general programming lifecycles for creating an application that uses the Windows Azure Service Bus and the Windows Azure Access Control service. These topics also include optimized lifecycles for creating an Service Bus application that is hosted on Windows Azure, and also to a lifecycle for creating Web-based Service Bus applications that comply with the REST standard.

In This Section

Basic Service Bus Programming Lifecycle

This topic describes building a Windows Communication Foundation (WCF) Web service, configuring it to register itself with the Service Bus, and creating a client that calls the service by using the Service Bus.

Windows Azure and Service Bus Programming Lifecycle

This topic describes building a Windows Communication Foundation (WCF) Web service hosted in Windows Azure, configuring it to register itself with the Service Bus, and creating a client that calls the service by using the Service Bus.

REST and Service Bus Relay Programming Lifecycle

This topic describes building two REST-style Web service applications for the Service Bus.

- The first is a Windows Communication Foundation (WCF) REST Web service that registers itself with the Service Bus just as the previous topic does.
- The second application is an HTTP REST-style application that is not a WCF application that creates an Service Bus message buffer to store messages and a REST client that retrieves them from the buffer.

See Also

Building Web Services that Trust ACS

Basic Service Bus Programming Lifecycle

The Windows Azure Service Bus enables Web service applications expose their functionality to clients through firewalls and on different application platforms. This topic outlines the tasks that are required to build an application that uses the Service Bus to expose its functionality to clients. For a working sample application, see the Service Bus Relayed Messaging Tutorial.

Basic Service Bus Tasks

The tasks required to create a Windows Communication Foundation (WCF) application that accesses the Service Bus are as follows. If you are hosting in Windows Azure, see Windows Azure and Service Bus Programming Lifecycle. If you want to use a REST-style application together with the Service Bus, see REST and Service Bus Relay Programming Lifecycle.

1. Create a service namespace. This service namespace creates a named scope within which the Service Bus creates resources to support Web services regardless of where they are originally hosted or how. For more information, see Managing Service Bus Service Namespaces.
2. Define the service contract, whether using WCF or using HTTP programming directly in the .NET Framework (if you are using a message buffer). A contract specifies the signature of the service, the data it exchanges, and other required inputs, behavior specifications, and object invariants. For more information, see Designing a WCF Contract for the Service Bus.
3. Implement the contract. To implement a service contract, create a class that implements the interface and specify custom runtime behavior.
4. Configure the service by specifying endpoint and other behavior information. For more information, see Configuring a WCF Service to Register with the Service Bus.
5. Build and run the service. For more information, see Building a Service for the Service Bus.
6. Build and run the client application. For more information, see Building a Service Bus Client Application.

As with any iterative, service-oriented software development, it may not always be appropriate to follow the previous steps sequentially, or even start from step 1. For example, if you want to build a client for a pre-existing service, you start at step 5. Or, if you are building a host service that others will use, you can skip step 6.

Windows Azure and Service Bus Programming Lifecycle

The Windows Azure Service Bus enables Web service applications expose their functionality to clients through firewalls and on different application platforms. This topic outlines the tasks that you can use to host a Web service on Windows Azure and expose its functionality using the Service Bus to connect your application with those of your customers.

The tasks required for creating a Windows Azure application for the Service Bus

The basic tasks required to create a Windows Azure application that accesses the Service Bus are the same as those in the topics described in Basic Service Bus Programming Lifecycle with the sole exception that you must perform some tasks required to host your application in Windows Azure. This is true whether your .NET application uses WCF and SOAP, the WCF REST Programming Model, or .NET HTTP request programming directly.

To restate the complete lifecycle considering this:

1. Create the Windows Azure project and service namespace. The project and service namespace contain the resources to support your application. For more information, see Managing Service Bus Service Namespaces. You must also create a Windows Azure Services account and project. For more information, see the Windows Azure documentation.
2. Define the service contract that is to be registered with the Service Bus. A contract specifies the signature of the service and the data it exchanges. For more information, see Designing a WCF Contract for the Service Bus. There are no additional steps at this point: Windows

Azure can act as the platform for an Windows Communication Foundation (WCF) interface for both a service and client application that has no modifications to the interface.

3. Implement the service contract. To implement a contract, create the class whose methods define the operations. As with the previous step, there is nothing specific for Windows Azure to do at this point: Windows Azure is the host for a Web service that has no additional modifications to the interface implementation.
4. Configure the service by specifying endpoint information and other behavior information. For more information, see Configuring a WCF Service to Register with the Service Bus. In addition, you must configure the Windows Azure service or client to use full trust authentication. You must also make sure that the Service Bus assembly is uploaded to Windows Azure. For more information, see How to: Configure a Windows Azure-Hosted Service Bus Service or Client Application.
5. Create and run the service. For more information, see How to: Host a Service on Windows Azure that Accesses the Service Bus. If your service is running locally (but connecting to a Windows Azure-hosted client), there are no additional steps to perform.
6. Create and build the client application. If the client application is running on the local computer, it requires no additional steps.

Depending upon the state of your application, you may not have to perform all these steps, or in this sequence. For example, if you want to build a client for a pre-existing service, you can start at step 5. Or, if you are building a service that others will use, you can skip step 6.

REST and Service Bus Relay Programming Lifecycle

This walkthrough illustrates how to use REST-style Web service applications that use the Windows Azure Service Bus in two different ways. The first uses the Windows Communication Foundation (WCF) Web programming model to create and register a REST-style Web service for use with the Service Bus, and then creates a REST client that invokes the REST service through the Service Bus.

The second example shows how to create a REST-style service application that uses the Service Bus message buffer to store messages until they are retrieved by a REST client. The message buffer is exposed as a REST interface to and from which applications can send and receive messages. This enables any form of REST-capable application to access the Service Bus, even if the service or client is hosted behind a firewall. In addition, a managed application can access the message buffer through a class wrapper. This topic outlines the tasks that are required to build a managed Service Bus application that complies with the REST protocol. For a working sample application, see the Service Bus Message Buffer Tutorial.

Note

Using message buffers is still supported, but future REST-style applications are advised to use queues, topics, and subscriptions – depending on the specific need. For more information on using queues, topics, and subscriptions in a REST-style application, see the Service Bus Brokered Messaging REST Tutorial.

Creating a REST-based Application that uses the Service Bus

The basic tasks required to create an application that accesses the Service Bus using the REST architecture model are as follows:

1. Create the Service Bus and Service Bus Access Control projects and the service namespace. The project and service namespace contain the resources to support your application. For more information, see**Creating a .NET Services Account**. Both styles of REST applications (REST applications that register Web service endpoints and those that only use the message buffer) must have a Windows Azure Service Bus project and namespace created by using the Service Bus Web portal: The former because it is a full Service Bus application; the latter because the message buffer is one of the resources created when you create a service namespace. The project also helps provide security and authentication, through the Access Control service. For more information, see **Building Applications that Use Access Control Services**.
2. Define the Service Bus contract. For more information, see Designing a WCF Contract for the Service Bus. The main difference between using the message buffer and registering a service endpoint is that if you are creating and registering a service endpoint that supports the REST protocol, you must apply WCF attributes to your interface that map to the HTTP verbs GET, PUT, DELETE, and UPDATE. For For more information, seeHow to: Expose a REST-based Web Service Through the Service Bus. If you are only using the message buffer, you do not have to define a REST interface: the message buffer itself is exposed through a REST interface, which means there is no additional interface necessary.
3. If you are registering a REST-based service endpoint, you must implement the contract in the previous step. The important point here is that the information passed through the interface must be in a format that is transmittable by a REST-style service, for example, a stream. As with the previous step, if you are only using the message buffer without additional support from the .NET Framework, you do not have to implement any form of service contract: the REST-based contract is already implemented and exposed by the message buffer.
4. Configure the service by specifying endpoint information and other behavior information. For more information, seeConfiguring a WCF Service to Register with the Service Bus. For a full Service Bus application that supports the REST protocol, the main difference is that the application must use a binding that supports the REST protocol, such as **Microsoft.ServiceBus.WebHttpRelayBinding**. However, other than that restriction, the actual configuration is identical to any other WCF application that uses the Service Bus. In contrast, an application that uses only the message buffer is much simpler. For more information, seeHow to: Configure a Service Bus Message Buffer.
5. Build and run the service. For more information, seeBuilding a Service for the Service Bus. For more information about creating an Service Bus service that supports the REST protocol, see How to: Create a REST-based Service that Accesses the Service Bus. In contrast, when non-WCF applications use the message buffer, the message buffer itself is the closest thing to a “host”, although one of the REST applications must have requested creating the message buffer. For more information, seeHow to: Create and Connect to a Service Bus Message Buffer.
6. Build a client application. For more information, seeBuilding a Service Bus Client Application. For a message buffer application, when the message buffer is created, any application that connects to the message buffer (including the application that created the buffer) can send

and receive information: there are no host or client applications. For more information, seeHow to: Send Messages to a Service Bus Message Buffer and How to: Retrieve a Message from a Service Bus Message Buffer.

As with WCF applications, some scenarios do not start at the first step. For example, if you want to build a client for a pre-existing service, you can start at step 5 (this would be the case for an application that uses only the message buffer). Or, if you are building a host service that others will use, you can skip step 6.

REST and Brokered Messaging Programming Lifecycle

Service Bus queues are first-in, first-out, durable lists of messages that you can use to build messaging applications that are not tightly coupled to a particular time or component, enabling all kinds of applications to participate in loosely coupled and robust distributed applications.

The development lifecycle mirrors that of the REST and Service Bus Relay Programming Lifecycle, as you do not have to create a REST-style Windows Communication Foundation (WCF) service contract in order to send a message. Creating a queue, a topic, or a subscription to a topic, or sending a message to or from one of these resources requires only the appropriate credentials for the specific action and the appropriate URI for the resource you want to create, delete, or use.

Basic Lifecycle

The basic lifecycle is as follows, and is completely implemented in a simple example in the Service Bus Brokered Messaging REST Tutorial.

1. Use your service namespace, issuer name, and issuer key to contact the Windows Azure Access Control service to obtain a Simple Web Token (SWT). You use the SWT to gain access to your Service Bus service namespace resources.
2. Create the resources you want to use. For example, you might create a queue or a topic. If a topic in which you are interested already exists, you can create a subscription to that topic, add a filter, and so on.
3. Send messages to a queue or a topic.
4. Retrieve messages from a queue or a subscription.
5. If needed, delete the queue, topic, or subscription whose resources you want to enable the Service Bus to reclaim.

See Also

Relayed and Brokered Messaging

Queues, Topics, and Subscriptions

Service Bus Brokered Messaging .NET Tutorial

Service Bus Authentication and Authorization with the Access Control Service

The authorization of Windows Azure Service Bus operations, meaning the act of deciding whether an operation may or may not be performed within the current security context, is a cooperative effort between the Windows Azure Access Control Service (ACS) and the Service Bus.

Windows Azure Access Control

Access Control facilitates authentication, meaning it establishes the identity of a caller. Access Control has two means of establishing the caller identity. It either establishes the identity based on a namespace-scoped list of service identities (or accounts) using a classic user name and password scheme, or it delegates establishing the identity to an external identity provider, such as Active Directory Federation Services (ADFS), Windows Live ID, Facebook, Google ID, Yahoo ID, or OpenID.

Once the identity has been established, Access Control has (or receives) a number of 'claims' about the identity. Those claims make statements about the person (or the non-person account), and they are digitally signed by the identity provider that issued the claims, which provides an assurance to Access Control that the claims are correct or at least in compliance with the governance rules of the issuer. In other words, a set of claims stating that the represented identity is "Bill Gates, Chairman, Microsoft Corporation" are likely most trustworthy when issued by the Microsoft ADFS gateway, and less so when issued by a third party. The claims that are consistently available across all identity providers and also for the Access Control built-in service identities are the provider claim (identifying the provider itself) and the 'nameidentifier' claim, which is a provider-specific and provider-unique identifier for the given identity.

Second, Access Control facilitates authorization by allowing the claims issued by identity providers to be mapped to claims that are understood by a 'relying party'. The Service Bus is such a relying party, meaning that it relies on Access Control to handle authentication and authorization. The mapping of claims serves two purposes: first, it normalizes the claims from a multitude of different claim 'lingos' into a single set of claims understood by the service, and, second, the mapping acts as an authorization rule set. If there's no mapping for a given identity to a set of claims understood by the service, the identity doesn't have access to the service.

Authentication and authorization always flows through the client, and the client is the only component requiring direct network visibility to all parties. It is, for example, possible to use an ADFS service that is not exposed outside the corporate firewall in conjunction with Access Control since Access Control and ADFS never talk to each other directly. Whenever the client wants to perform an operation on a protected resource, such as sending a message to a Service Bus queue, it needs to obtain proof that it is authorized to do so. That proof is acquired, from Access Control, in form of a 'token'. The token is simply a container for a set of claims, and is digitally signed by the issuer. If Access Control is configured to establish the identity using an external identity provider such as ADFS, there are at least two tokens in play. The first token is acquired from an identity provider such as ADFS, providing one of many kinds of proof of the

user's identity as input. That token is then handed to Access Control, which evaluates it, runs the rules, and emits the token for the relying party.

The Service Bus and Access Control

The Service Bus and Access Control have a special relationship in that each Service Bus service namespace is paired with a matching Access Control service namespace of the same name, suffixed with "–sb". The reason for this special relationship is in the way that Service Bus and Access Control manage their mutual trust relationship and the associated cryptographic secrets.

The Service Bus can federate with Access Control V1 as well as with Access Control V2. All service namespaces that were created before the September 2011 release of the Service Bus are federated with Access Control V1, and all service namespaces created after the service upgrade are federated with Access Control V2. This topic only covers Access Control V2, which is the current release of the Access Control service.

Inside the "-sb" Access Control service namespace, which you can explore from the Windows Azure Portal by selecting the Service Bus service namespace and then clicking the Access Control icon on the ribbon, is a "ServiceBus" relying party definition following the 'Relying Party Applications' navigation. The relying party definition has a 'Realm' value mapping to the root of the matching Service Bus service namespace (using the 'http' scheme), and sets the token type to 'SWT' and the expiration time of tokens to 1200 seconds. Furthermore, the signing keys are not manageable or accessible through the portal or the API.

Associated with the "ServiceBus" relying party definition is a "Default Rule Group for Service Bus" containing the basic mapping that enables the 'owner' of a service namespace to act as super-user on the service namespace. The rule group contains, by default, three simple rules that map the input 'nameidentifier' claim for the "owner" service identity to the three permission claims understood by Service Bus: 'Send' for all send operations, 'Listen' to open up listeners or receive messages, and 'Manage' to observe or manage the state of the Service Bus tenant. The Service Bus ignores all other claims contained in tokens issued to it. The "owner" service identity is a regular service identity in the Access Control service namespace. It is possible, and advised, to create more. In fact, using the "owner" identity should be restricted to performing administrative tasks.

Relying Party Definitions and Scoping

When a client requests an authorization token for sending a message to a queue residing at, for example, https://tenant.servicebus.windows.net/my/test, the token request will include a normalized form of the target address as the intended target realm. This 'normalization' simply uses a common URI scheme across all protocols. Therefore, requesting a token for interacting with a Service Bus entity residing at https://tenant.servicebus.windows.net/my/test or sb://tenant.servicebus.windows.net/my/test will always be done using a Realm URI using the 'http' scheme http://tenant.servicebus.windows.net/foo/bar. Consequently, all relying party definitions must also use the 'normalized' URI scheme 'http' for the Realm URI.

When the request arrives at Access Control, Access Controlwill match the realm URI to relying party definitions by means of a 'longest prefix match', which means that the relying party whose 'Realm URI' address is the longest available prefix of the address that the token is requested for, the relying party definition, and its associated rule definitions are selected and run. The default 'ServiceBus' relying party definition is scoped to the entirety of the corresponding Service Bus service namespace, meaning that its Realm URI, corresponding to the Service Bus service namespace root address, is a prefix to all possible addresses on a Service Bus service namespace. As such, the rule definitions enabled on this relying party definition grant full access across the entire Service Bus service namespace.

The way to create a scoped set of authorization rules for a queue residing at, for example, https://tenant.servicebus.windows.net/my/test, is to create a new relying party definition, providing the address of the queue or a prefix of that address as the Realm URI of the new definition, either through the Access Control portal or the Access Control management API. On the portal, the steps are:

- Under **Relying Party Applications** click **Add**.
- Enter some display name, for example **MyTest**.
- Enter **http://tenant.servicebus.windows.net/my/test** as the Realm URI for the scope.
- Choose **SWT** as the token format.
- Set **Encryption Policy** to None.
- Set **Token lifetime** to 1200 seconds.
- Click **Save**.

The result is a relying party definition that is exclusive to this address. Because its Realm URI is a suffix of the built-in 'ServiceBus' relying party definition, the definition automatically inherits the correct signing keys so that the Service Bus trusts tokens issued based on the new definition. However, since there are no associated rules for new the relying party definition, so far, nobody will be able to access the queue, not even the "owner", because there is no automatic implicit inheritance of rules between relying party definitions even if they form a hierarchy.

After creating the new definition, there will be a "Default Rule Group for <displayname>" in the Rule Groups section of the Access Control portal. This new group is empty by default. In order to permit access to the queue, rules need to be added to the group, which is explained in the following section. Alternatively, an already existing rule group with rules can be enabled for the relying party definition. Each rule group can be seen as a separate access control list that can be enabled anywhere in the relaying party hierarchy. To enable file-system like inheritance, for example to inherit the default rules of the Service Bus service namespace root, the corresponding "Default Rule Group for ServiceBus" and any other rule group can simply be enabled on the relying party definition – which requires checking the right box in the 'Rule groups' section of the Relying Party definition on the portal. For cases in which a common set of access rules should be applied across a number of resources, for example common rules for a set of parallel resources such as sibling queues at http://tenant.servicebus.windows.net/my/test and http://tenant.servicebus.windows.net/my/zoo, the relying party definition can also be scoped to the shared service namespace branch, such as http://tenant.servicebus.windows.net/my.

In other cases, scenarios may call for managing access control differently for aspects of the same Service Bus entity, such as different permissions on different subscriptions of a topic. In these cases it is possible to create a relying party definition scoped to a particular subscription name, such as http://tenant.servicebus.windows.net/my/test/subscriptions/sub1/, and have that definition hold the rules applying only to the particular named subscription.

Defining Rules

Rules are defined in rule groups and generally map an input claim to an output claim. All rules in a group yield a single combined result, so if there are three matching rules for a given input claim set that yield three distinct output claims, the issued authorization token will contain all three claims. For the Service Bus, the three permission claims are 'Send' for all send operations, 'Listen' to open up listeners or receive messages, and 'Manage' to observe or manage the state of the Service Bus tenant. To be precise, 'Send', 'Listen', and 'Manage' are the permitted values of the claim-type 'net.windows.servicebus.action'. Creating a rule for the Service Bus requires mapping an input claim, such as the nameidentifier of a service identity, to the desired permission claim. To grant the service identity "contoso" the permission to 'Send' on a queue, the rule definition would therefore map the issuer's nameidentifier claim with the value "contoso" to a custom output claim of type 'net.windows.servicebus.action' with a value of 'Send'. Granting the service identity all three permission claims requires three distinct rules. The goal of having just three permission claims is to limit the complexity of defining rules. The table below shows how the permission claims map to concrete operations on Service Bus entities:

Operation	Claim Required	Claim Scope
Service Registry		
Enumerate Private Policies	Manage	Any service namespace address
Relay		
Begin listening on a service namespace	Listen	Any service namespace address
Send messages to a listener at a service namespace	Send	Any service namespace address
Queue		
Create a queue	Manage	Any service namespace address
Delete a queue	Manage	Any valid queue address
Enumerate queues	Manage	/$Resources/Queues
Get the queue	Manage or Listen or	Any valid queue address

Operation	Claim Required	Claim Scope
description	Send	
Send into to the queue	Send	Any valid queue address
Receive messages from a queue	Listen	Any valid queue address
Abandon or complete messages after receiving the message in peek-lock mode	Listen	Any valid queue address
Defer a message for later retrieval	Listen	Any valid queue address
Deadletter a message	Listen	Any valid queue address
Get the state associated with a message queue session	Listen	Any valid queue address
Set the state associated with a message queue session	Listen	Any valid queue address
Topic		
Create a topic	Manage	Any service namespace address
Delete a topic	Manage	Any valid topic address
Enumerate topics	Manage	/$Resources/Topics
Get the topic description	Manage or Send	Any valid topic address
Send to the topic	Send	Any valid topic address
Subscription		
Create a subscription	Manage	Any service namespace address
Delete subscription	Manage	../myTopic/Subscriptions/mySubscription
Enumerate subscriptions	Manage	../myTopic/Subscriptions
Get subscription description	Manage or Listen	../myTopic/Subscriptions/mySubscription
Abandon or complete messages after	Listen	../myTopic/Subscriptions/mySubscription

Operation	Claim Required	Claim Scope
receiving the message in peek-lock mode		
Defer a message for later retrieval	Listen	../myTopic/Subscriptions/mySubscription
Deadletter a message	Listen	../myTopic/Subscriptions/mySubscription
Get the state associated with a topic session	Listen	../myTopic/Subscriptions/mySubscription
Set the state associated with a topic session	Listen	../myTopic/Subscriptions/mySubscription
Rule		
Create a rule	Manage	../myTopic/Subscriptions/mySubscription
Delete a rule	Manage	../myTopic/Subscriptions/mySubscription
Enumerate rules	Manage or Listen	../myTopic/Subscriptions/mySubscription/Rules

Using Token Providers

A token provider is a generic construct in the .NET managed API for the Service Bus that allows turning some form of credential into an authorization token issued by the Access Control service, that can then be passed on to the Service Bus to perform the desired operation. **Microsoft.ServiceBus.TokenProvider** is an abstract base class with three concrete implementations accessible via factory methods for the most basic scenarios:

- Shared Secret – allows obtaining a token based on a service identity (and the shared key associated with that identity) that has been defined in the Service Bus service namespace "-sb" buddy namespace in Access Control. The pre-provisioned service identity created when the service namespace is created is called "owner" and its shared secret is available through the management portal.
- Simple Web Token (SWT) – allows obtaining a token based on a previously acquired SWT token passed to Access Control via the token provider. The token is passed to Access Control as a binary token using a WS-Trust/WS-Federation RST/RSTR request. Please refer to the Access Controldocumentation for information about how to configure WS-Federation providers.
- SAML – allows obtaining a token based on a previously acquired SAML token passed to Access Control via the token provider. The token is passed to Access Control using a WS-Trust/WS-Federation RST/RSTR request. Please refer to the Access Controldocumentation for information about how to configure WS-Federation providers.

The Service Bus **Microsoft.ServiceBus.Messaging.MessagingFactory**, **Microsoft.ServiceBus.NamespaceManager**, and

Microsoft.ServiceBus.TransportClientEndpointBehavior APIs accept **Microsoft.ServiceBus.TokenProvider** instances. The token provider is called as tokens are required, which includes scenarios where a long-lived connection needs to acquire a new token once the existing token has passed its expiration (which defaults to 1200 seconds). Federation scenarios that require user interaction do require implementation of a custom token provider.

As an example, a custom token provider to enable a particular Facebook user to send messages to a particular queue will have to present the user, via Access Control, with the appropriate Facebook UI to establish the Facebook identity, redirect via Access Control to trade the Facebook token for an Access Control token for the Service Bus, and then extract the Access Control token as the request gets redirected to the local application. The Service Bus Codeplex site will contain a growing collection of token provider examples for customization.

Service Bus Bindings

The primary programming model for working with the Windows Azure Service Bus on the .NET platform is Windows Communication Foundation (WCF). The SDK includes a set of new WCF bindings that automate the integration between your WCF services and clients with the relay service offered by the Service Bus. In most cases, you just have to replace the current WCF binding that you are using with one of the Service Bus “relay” bindings.

The following table lists all of the Service Bus WCF bindings and the standard WCF bindings to which they correspond. The most frequently used WCF bindings, such as BasicHttpBinding, WebHttpBinding, WS2007HttpBinding, and NetTcpBinding, all have a corresponding Service Bus binding with a very similar name (just insert “Relay” before “Binding”). There are only a few new relay bindings – NetOnewayRelayBinding and NetEventRelayBinding – that do not have a corresponding WCF binding.

Standard WCF Binding	Equivalent Relay Binding
BasicHttpBinding	BasicHttpRelayBinding
WebHttpBinding	WebHttpRelayBinding
WS2007HttpBinding	WS2007HttpRelayBinding
NetTcpBinding	NetTcpRelayBinding
N/A	NetOnewayRelayBinding
N/A	NetEventRelayBinding

The relay bindings work in a similar manner to the standard WCF bindings. For example, they support the different WCF message versions (SOAP 1.1, SOAP 1.2, and None), the various WS-* security scenarios, reliable messaging, streaming, metadata exchange, the Web programming model (e.g., [WebGet] and [WebInvoke]), and many more standard WCF features. There are only a few WCF features not supported by design including atomic transaction flow and transport level authentication.

If you are familiar with how WCF works, you might be interested to know how the new bindings (shown earlier in this topic) map to the underlying WCF transport binding elements. The following table specifies the transport binding element for each relay binding. As you can see, the SDK includes several new WCF transport binding elements including RelayedHttpBindingElement, RelayedHttpsBindingElement, TcpRelayTransportBindingElement, and RelayedOnewayTransportBindingElement.

Relay Binding	Transport Binding Element
BasicHttpRelayBinding	RelayedHttp(s)BindingElement
WebHttpRelayBinding	RelayedHttp(s)BindingElement
WS2007HttpRelayBinding	RelayedHttp(s)BindingElement
NetTcpRelayBinding	TcpRelayTransportBindingElement
NetOnewayRelayBinding	RelayedOnewayTransportBindingElement
NetEventRelayBinding	RelayedOnewayTransportBindingElement

These new WCF primitives are ultimately what provide the low-level channel integration with the relay service behind the scenes, but those details are hidden from view behind the binding. The following sections discuss the details of the main WCF relay bindings and show how to use them.

NetMessagingBinding

The **Microsoft.ServiceBus.Messaging.NetMessagingBinding** binding can be used by WCF-enabled applications to send and receive messages through queues, topics and subscriptions. For more information, see NetMessagingBinding.

NetOnewayRelayBinding

Microsoft.ServiceBus.NetOnewayRelayBinding is the most constrained of the all the relay bindings, because it only supports one-way messages. However, it is also specifically optimized for that scenario. By default, the **Microsoft.ServiceBus.NetOnewayRelayBinding** binding uses SOAP 1.2 over TCP together with a binary encoding of the messages, although these communication settings are configurable through standard binding configuration techniques. Services that use this binding must always use the “sb” protocol scheme.

When using this binding in the default configuration, the on-premise WCF service attempts to establish an outbound connection with the relay service in order to create a bidirectional socket. In this case, it always creates a secure TCP/SSL connection through outbound port 828. During the connection process the WCF service authenticates (by supplying a token acquired from Access Control), specifies a name on which to listen in the relay service, and tells the relay service what type of listener to create. When a WCF client uses this binding in the default configuration, it creates a TCP connection with the relay via port 808 (TCP) or 828 (TCP/SSL),

depending on the binding configuration. During the connection process it must authenticate with the relay by supplying a token acquired from Access Control. Once the client has successfully connected, it can start to send one-way messages to the Service Bus to be “relayed” to the on-premises service through its TCP connection

If you set the **Microsoft.ServiceBus.NetOnewayRelayBinding** binding security mode property to **Transport**, the channel will require SSL protection. In this case, all traffic sent to and from the relay service will be protected via SSL; however, it is important to realize that the message will pass through the relay service in the clear. If you want to ensure full privacy, you should use the **Message** security mode, in which case you can encrypt everything except the addressing information in the message passing through the relay service.

The **Microsoft.ServiceBus.NetOnewayRelayBinding** binding requires all operations on the service contract to be marked as one-way operations (`IsOneWay=true`). Assuming that’s the case, to use this WCF binding, specify it on your endpoint definitions and supply the necessary credentials.

System Connectivity Mode

When using the **Microsoft.ServiceBus.NetOnewayRelayBinding**, the on-premise WCF service connects to the relay service over TCP by default. If you are operating in a network environment that does not enable any outbound TCP connections beyond HTTP(s), you can configure the various relay bindings to use a more aggressive connection mode to work around those constraints. This is made possible by configuring the on-premises WCF service to establish an HTTP connection with the relay service (instead of a TCP connection). The Service Bus provides a system-wide **ConnectivityMode** setting that you can configure with one of three values: `Tcp`, `Http`, and `AutoDetect` (see the following table). If you want to make sure that your services connect over HTTP, set this property to `Http`.

ConnectivityMode	Description
`Tcp`	Services create TCP connections with the relay service through port 828 (SSL).
`Http`	Services create an HTTP connection with the relay service making it easier to work around TCP port constraints.
`AutoDetect` (Default)	This mode automatically selects between the `Tcp` and `Http` modes based on an auto-detection mechanism that probes whether either connectivity option is available for the current network environment and prefers `Tcp`.

`AutoDetect` is the default mode, which means the relay bindings will automatically determine whether to use TCP or HTTP for connecting the on-premises service to the relay service. If TCP is possible on the given network configuration, it will use that mode by default (that is, it attempts to use TCP by sending a ping message to a connection-detecting URL). If the TCP connection

fails, it automatically switches to the HTTP mode. Hence, most of the time, you do not have to set this property explicitly because the default “auto detect” behavior determines the behavior for you. The only time that you have to set this property explicitly is when you want to force either TCP or HTTP.

You can set the connectivity mode at the AppDomain-level through the static **Microsoft.ServiceBus.ServiceBusEnvironment** class. It provides a **Microsoft.ServiceBus.ServiceBusEnvironment.SystemConnectivity** property in which you can specify one of the three **ConnectivityMode** values shown earlier in this section. The following code illustrates how to modify an application to use the HTTP connectivity mode:

```
...

ServiceBusEnvironment.SystemConnectivity.Mode = ConnectivityMode.Http;

ServiceHost host = new ServiceHost(typeof(OnewayService), address);

host.Open();

...
```

The system connectivity mode setting takes effect on all of the relay bindings.

NetEventRelayBinding

Microsoft.ServiceBus.NetEventRelayBinding is very similar to the **Microsoft.ServiceBus.NetOnewayRelayBinding** binding, in the way it is implemented. The binding defaults and security options are identical to those for **Microsoft.ServiceBus.NetOnewayRelayBinding**. In addition, the mechanics around how clients/services interact with the relay service are basically the same. In fact, the **Microsoft.ServiceBus.NetEventRelayBinding** class actually derives from the **Microsoft.ServiceBus.NetOnewayRelayBinding** class.

The main difference in the **Microsoft.ServiceBus.NetEventRelayBinding** binding is that it lets you register multiple WCF services with the same Service Bus address. When a client sends a message to such an address, the relay service multicasts the message to all on-premise WCF services currently subscribed to that address.

The **Microsoft.ServiceBus.NetEventRelayBinding** binding supports the same **Microsoft.ServiceBus.ServiceBusEnvironment.SystemConnectivity** options as **Microsoft.ServiceBus.NetEventRelayBinding**. When you configure the **Microsoft.ServiceBus.ServiceBusEnvironment.SystemConnectivity** property on the **Microsoft.ServiceBus.ServiceBusEnvironment** class, it takes effect for all endpoints. Hence, you can use the aggressive HTTP connectivity mode for all your on-premise **Microsoft.ServiceBus.NetEventRelayBinding** endpoints if they are hosted in a locked-down network environment that blocks outbound TCP connections.

NetTcpRelayBinding

The **Microsoft.ServiceBus.NetTcpRelayBinding** binding supports two-way messaging semantics and is very closely aligned with the standard WCF **NetTcpBinding** – the key

difference is that **Microsoft.ServiceBus.NetTcpRelayBinding** creates a publicly-reachable TCP endpoint in the relay service.

By default, the **Microsoft.ServiceBus.NetTcpRelayBinding** binding supports SOAP 1.2 over TCP and it uses binary serialization for efficiency. Although its configuration is very similar to that of the **NetTcpBinding**, their underlying TCP socket layers are different and are therefore not directly compatible with each other. This means that client applications will also have to be configured to use **Microsoft.ServiceBus.NetTcpRelayBinding** in order to integrate.

First, the on-premises WCF service establishes a secure outbound TCP connection with the relay service. During the process, it must authenticate, specify an address to listen on, and specify what type of listener to create in the relay. Up to this point, it is very similar to the **Microsoft.ServiceBus.NetOnewayRelayBinding** binding. When an incoming message arrives on one of the front nodes, a control message is then routed down to the on-premises WCF service indicating how to create a rendezvous connection back with the client front-end node. This establishes a direct socket-to-socket forwarder for relaying TCP messages.

The **Microsoft.ServiceBus.NetTcpRelayBinding** binding supports two connection modes (see **Microsoft.ServiceBus.TcpRelayConnectionMode**) that control how the client and service communicate with each other through the relay service (see the following table).

TcpConnectionMode	Description
Microsoft.ServiceBus.TcpRelayConnectionMode.Relayed (default)	All communication is relayed through the relay service. The SSL-protected control connection is used to negotiate a relayed end-to-end socket connection that all communication flows through. Once the connection is established the relay service behaves as a socket forwarder proxy relaying a bi-directional byte stream.
Microsoft.ServiceBus.TcpRelayConnectionMode.Hybrid	The initial communication is relayed through the relay service infrastructure while the client/service negotiate a direct socket connection to each other. The coordination of this direct connection is governed by the relay service. The direct socket connection algorithm can establish direct connections between two parties that sit behind opposing firewalls and NAT devices. The algorithm uses only outbound connections for firewall traversal and relies on a mutual port prediction algorithm for NAT traversal. Once a direct connection can be established the relayed connection is automatically upgraded to a direct connection without message or data loss. If the direct connection cannot be established, data will continue to flow through the relay service as usual.

Relayed mode is the default, while **Hybrid** mode instructs the relay service to establish a direct connection between the client and service applications. Therefore, no data has to pass through the relay. It is considered a “hybrid” mode because it starts by relaying information through the relay while it attempts to upgrade to a direct connection. If successful, it will switch over to a direct

connection without any data loss. If it cannot establish a direct connection, it will continue to use the relay service. The **Microsoft.ServiceBus.NetTcpRelayBinding** binding also supports the **Microsoft.ServiceBus.ServiceBusEnvironment.SystemConnectivity** feature when you must configure the on-premises service to connect to the relay service over HTTP. When you configure the **Microsoft.ServiceBus.ServiceBusEnvironment.SystemConnectivity** property, it takes effect for all endpoints. Hence, you can use the aggressive HTTP connectivity mode for all your on-premise **Microsoft.ServiceBus.NetTcpRelayBinding** endpoints if they are being hosted in a locked-down network environment that blocks outbound TCP connections.

HTTP Relay Bindings

All of the bindings discussed to this point require clients to use WCF on the client side of the interaction. When you need non-WCF clients to integrate with your Service Bus endpoints, you can support relaying HTTP-based messages by selecting one of the various HTTP relay bindings.

The Service Bus includes several HTTP bindings – **Microsoft.ServiceBus.WebHttpRelayBinding**, **Microsoft.ServiceBus.BasicHttpRelayBinding**, and **Microsoft.ServiceBus.WS2007HttpRelayBinding**. These HTTP bindings offer wider reach and more interoperability because they can support any client that knows how to use the standard protocols supported by each of these bindings. **Microsoft.ServiceBus.WebHttpRelayBinding** and **Microsoft.ServiceBus.BasicHttpRelayBinding** provide the greatest reach because they are based on HTTP/REST and basic SOAP, respectively. The **Microsoft.ServiceBus.WS2007HttpRelayBinding** binding can provide additional layers of functionality through the WS-* protocols. When using the **Microsoft.ServiceBus.WS2007HttpRelayBinding** binding, clients will have to support the same suite of WS-* protocols enabled on the endpoint.

Regardless of which HTTP relay binding you use, the mechanics of what occurs in the relay service is largely the same. The on-premises WCF service first establishes either a TCP or HTTP connection with the relay service depending on the **Microsoft.ServiceBus.ConnectivityMode** setting that is being used. The **Microsoft.ServiceBus.ConnectivityMode** functionality works the same on all HTTP relay bindings. Clients then start to send messages to the HTTP endpoint exposed by the relay service. This means WCF is no longer necessary on the client – any HTTP/SOAP compatible library will do. When an incoming message arrives on one of the front nodes, a control message is then routed to the service indicating how to create a rendezvous connection back with the front-end node of the client. This establishes a direct HTTP-to-socket forwarder for relaying the HTTP messages.

The relay service knows how to route SOAP 1.1, SOAP 1.2, and plain HTTP (REST) messages transparently. You control the messaging style and the various WS-* protocols you want to use by configuring one of the HTTP relay bindings as you would any other WCF binding.

Designing a WCF Contract for the Service Bus

The following topics describe how to design an Windows Communication Foundation (WCF) service contract that can be registered to be available on at a Windows Azure Service Bus endpoint.

In This Section

How to: Design a WCF Service Contract for use with the Service Bus

This topic describes how to create a SOAP-based WCF service contract that can be configured to use the Service Bus.

How to: Expose a REST-based Web Service Through the Service Bus

This topic describes how to create a REST-based WCF service contract that can be configured to use the Service Bus.

Designing an Windows Azure-compliant Windows Azure Service Contract

Because WCF service contracts – whether SOAP- or REST-based – can be hosted in Windows Azure, you can follow the instructions in either of the previous two topics.

How to: Design a WCF Service Contract for use with the Service Bus

After you have created your Windows Azure Service Bus project, you can start writing code. The first step in writing the code is to define the interface that your service application uses to communicate with the client application. This interface, known as a service contract, is almost identical to a Windows Communication Foundation (WCF) contract: it defines the name of the interface, and also to the methods and properties exposed by the interface. You can use WCF-style attributes to add information to the contract, and you use the same syntax to do this. The main difference is that the Service Bus is an extension of WCF. Therefore, you must also define a channel to connect to the Service Bus. However, other extensions of WCF use a similar channel. Therefore, the channel itself is not unique to the Service Bus. The following discussion is a brief overview of creating an Service Bus contract.

As with WCF, both the service and client applications are required to have a copy of the contract in their code. There are four ways this can occur:

1. **Manually define the contract** – this is the default, and is used most often when you are developing the interface. A simplified process for doing this is shown later in this section. For a complete discussion, see **Designing Service Contracts** in the WCF documentation.

2. **Copy the contract from the service code** – this is copying and pasting the contract from the service code, or sharing in the project. This is accomplished when you have quick access to the code, for example, when you are also the developer writing the client. Many of the sample applications in the Windows Azure SDK share the same interface definition, because both the client and service are in the same project.
3. **Use the ServiceModel Metadata Utility Tool (scvutil.exe)** – this is an application that you point to an exposed metadata endpoint on a running service application. It returns a file that contains the associated service contract. A simplified procedure for doing this is described later in this section. For a complete discussion, see **Accessing Services Using a WCF Client** and **ServiceModel Metadata Utility Tool (Svcutil.exe)** in the WCF documentation. The main difference in using Svcutil.exe on an Service Bus application is that the URI passed to the tool is on the Service Bus, instead of on the local host. Note that Svcutil.exe requires the target service to have the appropriate metadata exposed. For more information, seeHow to: Expose a Metadata Endpoint.
4. **Add a service reference through Visual Studio** – this is the UI version of Svcutil.exe, and can be accessed through the Visual Studio environment. A simplified procedure for accessing the **Add Service Reference** dialog box is shown later in this section. For more information, see**How to: Add, Update, or Remove a Service Reference** in the Visual Studio documentation. As stated previously, adding a service reference requires that the target service expose the necessary information through a metadata endpoint.

To manually create a Service Bus contract

1. Create the service contract by applying the **System.ServiceModel.ServiceContractAttribute** attribute to the interface that defines the methods the service is to implement.

```
[ServiceContract]
public interface IMyContract
{
    void Send(int count);
}
```

2. Indicate which methods in the interface a client can invoke by applying the **System.ServiceModel.OperationContractAttribute** attribute to them.

```
[ServiceContract]
public interface IMyContract
{
    [OperationContract]
    void Send(int count);
}
```

3. It best to explicitly define the name of your contract, and also to the namespace of your application, when declaring the contract. Doing so prevents the infrastructure from using the default name and namespace values. Note that this is not the service namespace: in this case, it represents a unique identifier for your contract, and should contain some kind of versioning information.

```
[ServiceContract(Name = "IMyContact", Namespace =
"http://samples.microsoft.com/ServiceModel/Relay/MyContractV1
")]
public interface IMyContract
{
    [OperationContract]
    void Send(int count);
}
```

4. Declare a channel that inherits from your interface and also to the `IClientChannel` interface.

```
[ServiceContract(Name = "IMyContact", Namespace =
"http://samples.microsoft.com/ServiceModel/Relay/MyContractV1
")]
public interface IMyContract
{
    [OperationContract]
    void Send(int count);
}

public interface IOnewayChannel : IOnewayContract,
IClientChannel { }
```

5. If you are creating a service application, implement the interface elsewhere in your code.

To use Svcutil.exe to create a service contract from a service application

1. Ensure that the service is running before you try to retrieve the metadata.
2. Use the command line to move to the location of the Svcutil.exe took in the Windows SDK.

   The default installation path is C:\Program Files\Microsoft SDKs\Windows\v6.0\Bin.

   From the command line, run the following command:

```
Svcutil.exe <service's Metadata Exchange (MEX) address or
HTTP GET address>
```

If the target address has the appropriate metadata exposed, you will retrieve a file that contains WCF client code that the client application can use to start the service application.

To add a Service Reference through Visual Studio

1. In **Solution Explorer**, right-click the name of the project to which you want to add the service. Then click **Add Service Reference**.

   The **Add Service Reference** dialog box appears.
2. In the **Address** box, enter the URL for the service. Then click **Go** to search for the service. If the service implements username/password security, you may be prompted for a username and password.
3. In the **Service** list, expand the node for the service that you want to use, and then select a service contract.
4. In the **Namespace** box, enter the namespace that you want to use for the reference.
5. Click **OK** to add the reference to the project.
6. A service client (proxy) is generated, and metadata describing the service is added to the App.config file.

How to: Expose a REST-based Web Service Through the Service Bus

Exposing a REST-based service via the Windows Azure Service Bus requires no special steps beyond what is required to make any Windows Communication Foundation (WCF) service REST-based. The main change is the addition of a series of attributes to the contract definition that map the contract operations to commands in the REST protocol. Note that these attributes are WCF attributes; the ability to define an interface as REST-compliant is an aspect of WCF, instead of anything specific to the Service Bus. Thus, the following is a simplified procedure for tagging a contract to be REST-compliant. For a complete discussion, see **WCF REST Programming Model** in the WCF documentation.

Note

If you plan to develop an application that uses the message buffer, you do not have to define an Service Bus contract: the message buffer is already exposed by using a REST interface. For more information, seeWorking with a Service Bus Message Buffer.

To define a REST-compliant interface for a Service Bus application

1. Define a standard Service Bus contract, as shown in the topic How to: Design a WCF Service Contract for use with the Service Bus.
2. When defining the service contract with the **System.ServiceModel.OperationContractAttribute** attribute, use one of the following values to indicate how the member maps to the REST protocol:

GET	[OperationContract, WebGet] or [OperationContract] [WebGet]
PUT	[OperationContract] [WebInvoke(Method = “PUT”)]
DELETE	[OperationContract] [WebInvoke(Method = “DELETE”)]
POST	[OperationContract] [WebInvoke]

The following example shows how to tag an interface member as a REST-style GET member.

```
public interface ImageContract
{
    [OperationContract, WebGet]
    Message GetImage();
}
```

3. If you are designing a service, implement the contract as a class elsewhere in your project.

Designing a Windows Azure-Compliant Service Bus Contract

A contract used by a Windows Azure Service Bus application that is running on Windows Azure is no different from the contract used by an Service Bus application that is running on the local computer. Any differences are in the code, or in the configuration file specific to the application itself, not in how the interfaces are created. For more information about that procedure, see, How to: Design a WCF Service Contract for use with the Service Bus.

Configuring a WCF Service to Register with the Service Bus

Configuring an application that uses the Windows Azure Service Bus requires that you set the following properties:

- The name of the service you are exposing.
- The interface (representing the service contract) your application either exposes or connects through.

- The type of binding your application uses, which includes transport, security, and encoding settings.
- The address at which the contract is available.

When using WCF you can set these properties in an WCF service or client application either programmatically or in an App.config file. However, it is usually best to specify the binding and address information declaratively in a configuration file, instead of imperatively in code unless your specific scenario requires it. Defining endpoints in code is usually not practical because the bindings and addresses for a deployed service are typically different from those used as the service is being developed. More generally, keeping the binding and addressing information out of the code enables them to change without having to recompile or redeploy the application.

Note that if you do not have the Windows Azure SDK installed on a particular computer the Machine.config file does not have the necessary extensions that enable the .NET run time to interpret Service Bus-specific information. While there are workarounds for this situation, it is usually easier to define your configuration in the code. (For more information about working to enable application configuration files, see the RelayConfigurationInstaller.exe Tool).

The following list contains the general scenarios for configuring an Service Bus application. For additional information about how to configure a WCF application, see **Configuring Services** in the WCF documentation.

- **Basic Application**

  This type of application is a WCF SOAP-based application that is configured to use the Service Bus as a secured relay to connect to other applications. The only differences required to use the Service Bus are the type of binding and the endpoint address. WCF applications that use the Service Bus use one of the relay bindings available in the Windows Azure SDK, bindings which have authentication and transport elements not present in the standard WCF bindings. Similarly, the endpoint address that is used for the service endpoint is an Service Bus URI based on the registered Service Bus namespace name (and may also have a different protocol scheme), whereas a regular WCF application uses an address based on the local host. Because these are the only two configuration differences, you can often – but not always – just reconfigure a currently existing WCF application to expose its services through the Service Bus

- **REST-based Application**

  The topics in this documentation describe are two types of REST applications: those that use the WCF Web programming model and the Windows Azure SDK, and one that does not. However, the topics in this section only describe REST applications that use the Windows Azure SDK as a programming foundation. (REST-based applications that do not use the Windows Azure SDK can use the message buffer as a location to and from which to send and receive messages. For more information about about REST applications that use the message buffer, see Working with a Service Bus Message Buffer.)

  If your application does use the Windows Azure SDK, you can configure your application just as any other REST-based WCF service: you must use a relay binding that supports HTTP, such as **Microsoft.ServiceBus.WebHttpRelayBinding** or **Microsoft.ServiceBus.WS2007HttpRelayBinding**, and you must apply the appropriate

WCF attributes to your interface, and confirm that your implementation can send and receive HTTP messages and events. For more information, seeDesigning a WCF Contract for the Service Bus.

- **Windows Azure-hosted Application**

  Because Windows Azure does not have the Windows Azure SDK installed it does not have the information necessary in its Machine.config file to identify configuration elements specific to Service Bus. If you must use an App.config file, you can include all of the necessary information into an App.config file by using the RelayConfigurationInstaller.exe Tool. However, whereas this enables you to use an App.config file together with Windows Azure, you may encounter duplication issues on your development computer. Therefore, we recommend that when you use Windows Azure, you configure your applications programmatically.

  In addition, you must set your Windows Azure worker or Web role to Full Trust – this is required for all applications that use the Service Bus. This is not specified in the configuration file, but in the ServiceDefinition.csdef file. For more information, seeHow to: Configure a Windows Azure-Hosted Service Bus Service or Client Application.

In This Section

How to: Configure a Service Bus Service Programmatically

How to: Configure a Service Bus Service Using a Configuration File

How to: Configure a Service Bus Client Using Code

How to: Configure a Service Bus Client Using a Configuration File

How to: Configure a Windows Azure-Hosted Service Bus Service or Client Application

How to: Change the Connection Mode

Creating a Custom Service Bus Binding

Windows Azure Service Bus Quotas

Service Bus Port Settings

How to: Configure a Service Bus Service Programmatically

Once you have defined and implemented the interface for the Service Bus service in your code, you can start configuring your application. Note that configuring an Service Bus application is very similar to configuring a Windows Communication Foundation (WCF) application, as described in **Configuring Services** in the WCF documentation. Therefore, this topic contains a simplified procedure for configuring an Service Bus application, and also a discussion of the setting specific to the Service Bus.

In addition to the issues discussed in Configuring a WCF Service to Register with the Service Bus, an Service Bus service must determine what kind of authentication and transport security, if any, is required. Authentication security is the type of security necessary for the service to connect to the Service Bus. A service is always required to present authentication credentials to

the Service Bus, usually in the form of a shared secret (that is, issuer name and secret) token. However, the service also determines what type of authentication credentials the client applications must use in order to connect to the service. By default, client authentication is set to **RelayClientAuthenticationType.RelayAccessToken**, which means that the client must present some form of authentication to the Service Bus. In the current version of Windows Azure, this is always another shared secret token. In contrast, transport security determines whether it must connect with some form of secure line. This is referred to as “end-to-end” security because it covers the whole connection between the service, the Service Bus, and the client. In contrast, client authentication covers only the required relationship to connect from the service to the Service Bus. By default, the transport security is set to **EndToEndSecurityMode.Transport**. This means that security is provided using some form of secure transport, such as HTTPS. It is recommended that you keep the end-to-end security mode set to **Transport** unless you have a compelling reason to change it, as doing this might reduce the security of your application. For more information about security settings, see Securing and Authenticating a Service Bus Connection

The following procedure describes how to configure an Service Bus service programmatically.

To programmatically configure a Service Bus Service

1. Create the URI of the endpoint that includes your service namespace name and schema type.

```
string serviceNamespace = "myServiceNamespace";
Uri uri = ServiceBusEnvironemnt.CreateServiceUri("sb",
serviceNamespace, "sample/log/";
```

   The prefix “sb” indicates that this URI uses the Service Bus schema. Other schemas include HTTP or HTTPS.

2. Instantiate the host with the contract and URI.

```
host = new ServiceHost(typeof(EchoService), uri);
```

3. Declare and implement the type of authentication credentials to use.

```
string issuerName = "MY ISSUER NAME"
string issuerSecret = "MY SECRET";
TransportClientEndpointBehavior
sharedSecretServiceBusCredential = new
TransportClientEndpointBehavior();
sharedSecretServiceBusCredential.CredentialType =
TransportClientCredentialType.SharedSecret;
sharedSecretServiceBusCredential.Credentials.SharedSecret.Iss
uerName = issuerName;
sharedSecretServiceBusCredential.Credentials.SharedSecret.Iss
```

```
uerSecret = issuerSecret;
```

All services are required to use authentication credentials to connect to the Service Bus. Note that hard-coding the issuer name and secret into your code is not a secure practice. For example, many of the samples in the Windows Azure SDK prompt the user for this information.

4. Declare the type and instance of the contract.

```
ContractDescription contractDescription =
ContractDescription.GetContract(typeof(IEchoContract),
typeof(EchoService));
```

5. Add the contract description to the service endpoint.

```
ServiceEndpoint serviceEndPoint = new
ServiceEndpoint(contractDescription);
```

6. Add the URI to the service endpoint.

```
serviceEndPoint.Address = new EndpointAddress(uri);
```

7. Declare the type of binding to use for the endpoint.

```
serviceEndPoint.Binding = new NetTcpRelayBinding();
```

At this point you can declare the **Authentication** and **EndToEndSecurity** mode. This particular example uses the default constructor, which sets the **Microsoft.ServiceBus.EndToEndSecurityMode** to **Transport** and the **Microsoft.ServiceBus.RelayClientAuthenticationType** to **RelayAccessToken**. Therefore,, the following snippet is identical to the default constructor, except that it sets those two parameters explicitly:

```
serviceEndPoint.Binding = new
NetTcpRelayBinding(EndToEndSecurityMode.Transport,
RelayClientAuthenticationType.RelayAccessToken);
```

8. Add the security credentials to the endpoint.

```
serviceEndpoint.Behaviors.Add(sharedSecretServiceBusCredentia
l);
```

These security credentials are required for all services to authenticate with the Service Bus. Because we set the **Microsoft.ServiceBus.RelayClientAuthenticationType** to **RelayAccessToken** (either by default or explicitly), any client applications are also required to use the same type of authentication credentials.

9. Add the endpoint to the host.

```
host.Description.Endpoints.Add(serviceEndPoint);
```

You have now created the minimum configuration necessary for an Service Bus service application. At this point, you can add more service-level or endpoint-level configurations, as you would with any other WCF application. For more information about configuring a WCF application, see **Configuring Services** in the WCF documentation. When you are finished configuring your application, you can host and run your application. For more

information, seeBuilding a Service for the Service Bus.

Example

Description

The following example shows how to define configuration information programmatically. The main difference is that all information is set programmatically; in the tutorial, some information not specific to Windows Azure is stored in an App.config file.

Code

```
namespace AzureSample_WorkerRole
{
    public class WorkerRole : RoleEntryPoint
    {
        private ServiceHost host;

        public override void Start()
        {
            string serviceNamespace = "myDomainName";
            string issuerName = "MY ISSUER NAME"
            string issuerSecret = "MY SECRET";

            Uri uri = ServiceBusEnvironemnt.CreateServiceUri("sb", serviceNamespace,
"sample/log/";

            host = new ServiceHost(typeof(EchoService), uri);

            TransportClientEndpointBehavior sharedSecretServiceBusCredential = new
TransportClientEndpointBehavior();
            sharedSecretServiceBusCredential.CredentialType =
TransportClientCredentialType.SharedSecret;
            sharedSecretServiceBusCredential.Credentials.SharedSecret.IssuerName =
issuerName;
            sharedSecretServiceBusCredential.Credentials.SharedSecret.IssuerSecret =
issuerSecret;
```

```
            ContractDescription contractDescription =
ContractDescription.GetContract(typeof(IEchoContract), typeof(EchoService));

            ServiceEndpoint serviceEndPoint = new ServiceEndpoint(contractDescription);
            serviceEndPoint.Address = new EndpointAddress(uri);

            serviceEndPoint.Binding = new NetTcpRelayBinding();

           serviceEndpoint.Behaviors.Add(sharedSecretServiceBusCredential);
            host.Description.Endpoints.Add(serviceEndPoint);

            host.Open();

            while (true)
            {
                //Loop
            }
        }

        public override void Stop()
        {
            host.Close();
            base.Stop();
        }

        public override RoleStatus GetHealthStatus()
        {
            // This is a sample worker implementation. Replace with your logic.
            return RoleStatus.Healthy;
        }
    }
}
```

Comments

How to: Configure a Service Bus Service Using a Configuration File

Once you have defined and implemented your Service Bus interface, you can configure the service. You can configure an Service Bus service programmatically or in an App.config file. Configuring your application in an App.config file lets you easily see what the configuration settings are, and let users easily modify the settings after deployment. Note that configuring an Service Bus application by using a configuration file is like configuring a Windows Communication Foundation (WCF) application, which is discussed in the topic **Configuring Services Using Configuration Files**, in the WCF documentation. Therefore,, the following discussion is a simplified overview of configuration techniques, with an emphasis on the unique features relevant to the Service Bus.

In addition to the issues discussed in Configuring a WCF Service to Register with the Service Bus, an Service Bus service must determine what type of authentication and transport security, if any, is required. Authentication security is the type of security necessary for a service to connect to the Service Bus. A service is always required to present authentication credentials to the Service Bus, usually in the form of a shared secret (that is, issuer name and secret) token. However, the service also determines what type of authentication credentials the client applications must use in order to connect to the service. By default, client authentication is set to **RelayClientAuthenticationType.RelayAccessToken**, which means that the client must present some form of authentication to the Service Bus. In the current version of Windows Azure, this is always another shared secret token. In contrast, transport security determines whether it must connect with some form of secure line. This is referred to as “end-to-end” security because it covers the whole connection between the service, the Service Bus, and the client. In contrast, client authentication covers only the required relationship to connect from the service to the Service Bus. By default, transport security is set to **EndToEndSecurityMode.Transport**, which means that security is provided using some form of secure transport, such as HTTPS. It is recommended that you keep the end-to-end security mode set to **Transport** unless you have a compelling reason to change it, as doing this might reduce the security level in your application. For more information about setting security, see Securing and Authenticating a Service Bus Connection

Note

The Service Bus uses the default configuration that you have specified in the App.config file. Therefore, you do not have to directly reference the configuration in your code. However, if you have multiple endpoints and bindings, you may want to explicitly state which configuration to use, in order to avoid confusion.

The following procedure describes how to configure an Service Bus service by using an App.config file.

To configure a Service Bus service application by using an App.config file

1. To configure the endpoint for the service that uses the contract with a specified binding, create the App.config file. For example:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.serviceModel>
<services>
<!-- Application Service -->
<service name="Microsoft.ServiceBus.Samples.EchoService">
<endpoint
contract="Microsoft.ServiceBus.Samples.IEchoContract"
                  binding="netTcpRelayBinding"

address="sb://MyCodeSample.ServiceBus.Microsoft.com/EchoServi
ce" />
</service>
</services>
</system.serviceModel>
</configuration>
```

   Note the minimum parameters that are required to configure an endpoint: the service name, the contract that implements the service, the type of binding used, and the address. The service uses the default **Transport** and **Authentication** security parameters. Therefore, they are not explicitly declared. The address is explicitly declared here, although very often it is built programmatically using the URI type.

   Once you have completed the steps that are required to configure an Service Bus service, you can add more endpoint and service-level configurations. For more information, see**Configuring Services** in the WCF documentation.

2. After you have finished configuring the service, you can host and run the application. For more information, seeBuilding a Service for the Service Bus.

How to: Configure a Service Bus Client Using Code

Once you have defined the Windows Azure Service Bus interface (as described in Designing a WCF Contract for the Service Bus) in your code, you can continue configuring the client application. Configuring an Service Bus client application programmatically is very similar to configuring any other Windows Communication Foundation (WCF) application, which is described in detail in the **Configuring Services** topic in the WCF documentation. The following topic

describes programmatically creating and configuring a simple client that uses the **Microsoft.ServiceBus.NetTcpRelayBinding** binding, and also a discussion of issues specific to Service Bus.

The main difference between configuring a service and a client is that clients must know some of the configuration settings that are used by the service, and comply with them. These settings include the type of binding to use, and also what security options you must have to access the service (that is, authentication and transport-level security.) For more information about setting security, see Securing and Authenticating a Service Bus Connection.

To configure a Service Bus client using code

1. Define the endpoint using the service namespace name and schema type.

```
string serviceNamespace = "ServiceBusTutorial";
Uri uri = ServiceBusEnvironemnt.CreateServiceUri("sb",
serviceNamespace, "sample/EchoService/";
```

   In this context, the schema type is “sb”, indicating the Service Bus, and the service namespace is “ServiceBusTutorial”.

2. Create a **Microsoft.ServiceBus.TransportClientEndpointBehavior** with your credentials.

```
TransportClientEndpointBehavior
sharedSecretServiceBusCredential = new
TransportClientEndpointBehavior();
sharedSecretServiceBusCredential.CredentialType =
TransportClientCredentialType.SharedSecret;
sharedSecretServiceBusCredential.Credentials.SharedSecret.Iss
uerName = "YOUR ISSUER NAME";
sharedSecretServiceBusCredential.Credentials.SharedSecret.Iss
uerSecret = "YOUR ISSUER SECRET";
```

   In your code you will have to substitute “YOUR ISSUER NAME” and “YOUR ISSUER SECRET” with the issuer name and secret you want to use when you connect to the service endpoint.

3. Create and initialize the channel factory with the endpoint, binding type, and contract type:

```
ChannelFactory<IEchoChannel> channelFactory = new
ChannelFactory<IEchoChannel>();
channelFactory.Endpoint.Address = new EndpointAddress(uri);
channelFactory.Endpoint.Binding = new NetTcpRelayBinding();
channelFactory.Endpoint.Contract.ContractType =
typeof(IEchoChannel);
```

4. Apply the Service Bus credentials:

```
ChannelFactory.Endpoint.Behaviors.Add(sharedSecretServiceBusC
redential);
```

You are now finished configuring the client application. You can move on to implementing the rest of the client application, in Building a Service Bus Client Application. Note that this tutorial does use an App.config file; however, it is only to store name and password information in a string, and is easily replaced by the code used in this procedure.

How to: Configure a Service Bus Client Using a Configuration File

Once you have defined and implemented your Windows Azure Service Bus interface, you can configure the client application. You can configure your application programmatically or in an App.config file. Configuring your application in an App.config file lets you easily see what the configuration settings are, and lets users modify the settings after deployment. Note that configuring an Service Bus application by using a configuration file is very similar to configuring a Windows Communication Foundation (WCF) application, which is discussed in **Configuring Services Using Configuration Files** in the Windows Communication Foundation (WCF) documentation. Therefore,, the following discussion is a simplified overview of configuration, with an emphasis on the unique features relevant to the Service Bus.

The main difference between configuring a service and a client is that the client must know what configuration settings the service is using, and match them. Such settings typically include what type of binding to use, and also what security protocols you must have to access the service (that is, authentication and transport-level security). For more information about setting security, see Securing and Authenticating a Service Bus Connection.

To configure a Service Bus Client using an App.config file

1. Create the App.config file to define the client endpoint.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.serviceModel>
<client>
<endpoint name="RelayEndpoint"

contract="Microsoft.ServiceBus.Samples.IEchoContract"
                binding="netTcpRelayBinding"
                bindingConfiguration="default"
```

```
behaviorConfiguration="sharedSecretEndpointBehavior"
                address="" />
</client>
</system.serviceModel>
</configuration>
```

Similar to configuring a service application, the minimum parameters that you must have to configure an endpoint are the contract that implements the service, and the type of binding used. The address attribute is the Service Bus address. It can be specified explicitly in the configuration file but is usually constructed programmatically using the **Microsoft.ServiceBus.ServiceBusEnvironment.CreateServiceUri(System.String,System.String,System.String)** method. This example also associates an endpoint behavior with the client.

2. Define the endpoint behavior that contains the security settings.

```
<behaviors>
<endpointBehaviors>
<behavior name="sharedSecretEndpointBehavior">
<transportClientEndpointBehavior
credentialType="SharedSecret">
<clientCredentials>
<sharedSecret issuerName="ISSUER_NAME"
issuerSecret="ISSUER_SECRET" />
</clientCredentials>
</transportClientEndpointBehavior>
</behavior>
</endpointBehaviors>
</behaviors>
```

In this example, for simplicity the security credentials are defined by using the issuer name and secret in clear text. Note that this is an nonsecure programming practice: a more secure process (and the process used by many samples in the Windows Azure SDK) is to query the user for this information. Alternately, you could decide to encrypt the App.config file to avoid exposing this information.

3. Define the binding that the client application is to use when it connects to the Service Bus.

```
<bindings>
<!-- Application Binding -->
<netTcpRelayBinding>
```

```
<!-- Default Binding Configuration-->
<binding name="default" />
</binding>
</bindings>
```

4. You have finished configuring the client application through the App.config file. For more information about creating a Service Bus client application, see Building a Service Bus Client Application.

How to: Configure a Windows Azure-Hosted Service Bus Service or Client Application

Configuring a service or client application that runs on Windows Azure follows the same general programming patterns for both a Windows Azure and a basic Windows Azure Service Bus application. However, note the following issues:

- **Windows Azure does not include the Service Bus assembly**

  The default Windows Azure installation does not include the Service Bus assembly and, because of Windows Azure security restrictions, you cannot install the Windows Azure SDK on the Windows Azure platform.

  Therefore, to run any Service Bus application on Windows Azure, you must redistribute the Service Bus assembly with your Service Bus application. For more information about how to package an assembly with your application, see the following procedure.

- **Service Bus and Access Control must have Full Trust authorization to run on Azure**

  As with all other applications that use the Service Bus, you must make sure that the operating system is running with Full Trust authorization. This can be set in the Servicedefinition.csdef file, using the following procedure.

- The Windows Azure SDK version 1.5 no longer adds entries to the Machine.config file. You may see errors such as the following:

  ```
  Configuration binding extension
  'system.serviceModel/bindings/netTcpRelayBinding' could not be found. Verify
  that this binding extension is properly registered in
  system.serviceModel/extensions/bindingExtensions and that it is spelled
  correctly.
  ```

  It is recommended that you add these extensions to the App.config files for your projects or use the Relayconfiginstaller.exe tool in the SDK to add these bindings. For example:

  ```
  <configuration>
  <system.serviceModel>
  <extensions>
  ```

```
<!-- Adding all known service bus extensions. You can remove the
ones you don't need. -->
<behaviorExtensions>
<add name="connectionStatusBehavior"
type="Microsoft.ServiceBus.Configuration.ConnectionStatusElement
, Microsoft.ServiceBus, Version=1.5.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
<add name="transportClientEndpointBehavior"
type="Microsoft.ServiceBus.Configuration.TransportClientEndpoint
BehaviorElement, Microsoft.ServiceBus, Version=1.5.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
<add name="serviceRegistrySettings"
type="Microsoft.ServiceBus.Configuration.ServiceRegistrySettings
Element, Microsoft.ServiceBus, Version=1.5.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
</behaviorExtensions>
<bindingElementExtensions>
<add name="netMessagingTransport"
type="Microsoft.ServiceBus.Messaging.Configuration.NetMessagingT
ransportExtensionElement, Microsoft.ServiceBus, Version=1.5.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
<add name="tcpRelayTransport"
type="Microsoft.ServiceBus.Configuration.TcpRelayTransportElemen
t, Microsoft.ServiceBus, Version=1.5.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
<add name="httpRelayTransport"
type="Microsoft.ServiceBus.Configuration.HttpRelayTransportEleme
nt, Microsoft.ServiceBus, Version=1.5.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
<add name="httpsRelayTransport"
type="Microsoft.ServiceBus.Configuration.HttpsRelayTransportElem
ent, Microsoft.ServiceBus, Version=1.5.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
<add name="onewayRelayTransport"
type="Microsoft.ServiceBus.Configuration.RelayedOnewayTransportE
lement, Microsoft.ServiceBus, Version=1.5.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" />
</bindingElementExtensions>
```

```
<bindingExtensions>
<add name="basicHttpRelayBinding"
type="Microsoft.ServiceBus.Configuration.BasicHttpRelayBindingCo
llectionElement, Microsoft.ServiceBus, Version=1.5.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
<add name="webHttpRelayBinding"
type="Microsoft.ServiceBus.Configuration.WebHttpRelayBindingColl
ectionElement, Microsoft.ServiceBus, Version=1.5.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
<add name="ws2007HttpRelayBinding"
type="Microsoft.ServiceBus.Configuration.WS2007HttpRelayBindingC
ollectionElement, Microsoft.ServiceBus, Version=1.5.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
<add name="netTcpRelayBinding"
type="Microsoft.ServiceBus.Configuration.NetTcpRelayBindingColle
ctionElement, Microsoft.ServiceBus, Version=1.5.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
<add name="netOnewayRelayBinding"
type="Microsoft.ServiceBus.Configuration.NetOnewayRelayBindingCo
llectionElement, Microsoft.ServiceBus, Version=1.5.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
<add name="netEventRelayBinding"
type="Microsoft.ServiceBus.Configuration.NetEventRelayBindingCol
lectionElement, Microsoft.ServiceBus, Version=1.5.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
<add name="netMessagingBinding"
type="Microsoft.ServiceBus.Messaging.Configuration.NetMessagingB
indingCollectionElement, Microsoft.ServiceBus, Version=1.5.0.0,
Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
</bindingExtensions>
</extensions>
</system.serviceModel>
</configuration>
```

To package the Service Bus assembly with your application

1. In **Solution Explorer**, under the **WorkerRole** or **WebRole** node (depending on where you have your code), add the **Microsoft.ServiceBus** assembly to your Windows Azure project as a reference.

This step is the standard process for adding a reference to an assembly.

2. In the **Reference** folder, right-click **Microsoft.ServiceBus**. Then click **Properties**.
3. In the **Properties** dialog, set **Copy Local** to **True**.

   Doing so makes sure that the **Microsoft.ServiceBus** assembly will be available to your application when it runs on Windows Azure.

To set a Windows Azure application to Full Trust

1. In your ServiceDefinition.csdef file, set the `enableNativeCodeExecution` field to `true` as shown in the following code, replacing `ApplicationNameHere` with the name of your application:

```
<?xml version="1.0" encoding="utf-8"?>
<ServiceDefinition name="ApplicationNameHere"
xmlns="http://schemas.microsoft.com/ServiceHosting/2008/10/Se
rviceDefinition">
<WebRole name="WebRole" enableNativeCodeExecution="true">
<InputEndpoints>
<!-- Must use port 80 for http and port 443 for https when
running in the cloud -->
<InputEndpoint name="HttpIn" protocol="http" port="80" />
</InputEndpoints>
</WebRole>
<WorkerRole name="WorkerRole" enableNativeCodeExecution="true">
</WorkerRole>
</ServiceDefinition>
```

How to: Change the Connection Mode

The connection mode defines how your Windows Azure Service Bus application connects to other applications when it uses a TCP connection: either Relayed or Hybrid:

- **Relayed** – all communications between the service and client applications use the Service Bus.
- **Hybrid** – the initial connection between the applications uses the Service Bus. However, if the two applications can connect to each other, they will attempt to do this. If at any time this connection is lost, the applications will revert to communicating through the Service Bus.

The connection mode is defined by the **Microsoft.ServiceBus.TcpRelayConnectionMode** enumeration, which is in turn used by

Microsoft.ServiceBus.TcpRelayTransportBindingElement to define the **Microsoft.ServiceBus.NetTcpRelayBinding** binding. Therefore,, you can set the connection mode for a **Microsoft.ServiceBus.NetTcpRelayBinding** binding by setting the **Microsoft.ServiceBus.NetTcpRelayBindingBase.ConnectionMode** property. You can do this at any time in your application: the Service Bus will take the change and attempt to modify the connection. As with most properties, you can decide to set the connection status in the App.config file. Finally, the Service Bus exposes the current status of the connection through the **Microsoft.ServiceBus.IHybridConnectionStatus** interface, which you can decide to implement.

The following topic describes how to determine or modify the mode of an Service Bus application.

To modify the connection mode programmatically

1. Create the binding, and then set the property.

```
NetTcpRelayBinding myTcpBinding = new NetTcpRelayBinding();
myTcpBinding.ConnectionMode = TcpRelayConnectionMode.Hybrid;
```

To modify the connection mode with an App.config file

1. Open the application configuration file for the client application, as in the following example.

```
<bindings>
<netTcpRelayBinding>
<binding name="default"/>
</netTcpRelayBinding>
</bindings>
```

   Note that the **netTcpRelayBinding** tag does not specify the connection mode attribute. By default, the connection mode used is **Relayed**.

2. Explicitly change the connection mode to **Hybrid** by adding the **connectionMode** field, which is set to **Hybrid**.

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

To determine when the connection mode changes

1. Optionally, you might want to receive a notification when the connection between the connected endpoints changes from a **Relayed** connection to a **Hybrid** connection.

```
channel.Open();

IHybridConnectionStatus hybridConnectionStatus =
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):");
```

Creating a Custom Service Bus Binding

This section discusses the transport binding primitives. If you want to create a Windows Communication Foundation (WCF) custom binding, use one of these binding elements to send to or listen on the Windows Azure Service Bus instead of using the corresponding WCF transport binding elements.

TcpRelayTransportBindingElement

The **Microsoft.ServiceBus.TcpRelayTransportBindingElement** element is closely aligned with the WCF version, **System.ServiceModel.Channels.TcpTransportBindingElement** and is the foundation for **NetTcpRelayBinding**.

Additional members	Description
RelayClientAuthenticationType	The **TcpRelayTransportBindingElement** uses a federated security token authorization scheme to guard access to the

Additional members	Description
	Service Bus and to the services listening through it. This property controls whether clients of a service are required to present a security token issued by the Access Control service to the Service Bus service when it sends messages. Services are always required to authenticate with Access Control and present an authorization token to the Service Bus. If the service assumes the responsibility of authenticating/authorizing clients, it can opt out of the integration between Access Control and Service Bus by setting this property to **RelayClientAuthenticationType.None**. The default value is **RelayClientAuthenticationType.RelayAccessToken**.
ConnectionMode	See the discussion of Connection Modes in **NetTcpRelayBinding**. The following are the **Connection Mode** values: • **TcpConnectionMode.Relayed**: In this mode, all communication is relayed through the Service Bus cloud. • **TcpConnectionMode.Hybrid**: In this mode, communication is relayed through the Service Bus cloud whereas the client and service endpoints negotiate a direct socket connection to each other. If a direct connection can be established, the relayed connection is automatically upgraded to the direct connection. • **TcpConnectionMode.Direct**: This mode is identical to **TcpConnectionMode.Hybrid**.
TransportProtectionEnabled	This property gets or sets a Boolean value that indicates whether transport protection (SSL) is enabled for the connection. With this property set to **true**, outbound communication uses SSL through port 828. With the property set to **false**, outbound communication uses port 808.

HttpRelayTransportBindingElement

The **Microsoft.ServiceBus.HttpRelayTransportBindingElement** element is closely aligned with the WCF version, **System.ServiceModel.Channels.HttpTransportBindingElement** and is the foundation for all HTTP relay bindings that are configured to use unsecured HTTP communication.

Additional members	Description
RelayClientAuthenticationType	The **HttpRelayTransportBindingElement** uses a federated security token authorization scheme to guard access to the Service Bus and to the services listening through it. This property controls whether clients of a service are required to present a security token issued by the Access Control service to the Service Bus service when it sends messages. Services (listeners) are always required to authenticate with Access Control and present an authorization token to the Service Bus. If the service assumes the responsibility of authenticating and authorizing clients, it can opt out of the integration between Access Control and Service Bus by setting this property to **RelayClientAuthenticationType.None**. The default value is **RelayClientAuthenticationType.RelayAccessToken**.

HttpsRelayTransportBindingElement

The **Microsoft.ServiceBus.HttpsRelayTransportBindingElement** element derives from **HttpRelayTransportBindingElement** and is the foundation for all HTTP Relay bindings that are configured to use secured HTTPS communication.

OnewayRelayTransportBindingElement

The **Microsoft.ServiceBus.OnewayRelayTransportBindingElement** element is the foundation for the **Microsoft.ServiceBus.NetEventRelayBinding** and **Microsoft.ServiceBus.NetOnewayRelayBinding** bindings. As a TCP-based transport binding element, it is similar to the WCF **System.ServiceModel.Channels.TcpTransportBindingElement** element, but only supports one-way communication.

Additional members	Description
RelayClientAuthenticationType	The **OnewayRelayTransportBindingElement** uses a federated security token authorization scheme to guard access to the Service Bus and to the services listening through it. This property controls whether clients of a service are required to present a security token issued by the Access Control service to the Service Bus service when it sends messages. Services are always required to authenticate with Access Control and present an authorization token to the

Additional members	Description
	Service Bus. If the service assumes the responsibility of authenticating and authorizing clients, it can opt out of the integration between Access Control and Service Bus by setting this property to **RelayClientAuthenticationType.None**. The default value is **RelayClientAuthenticationType.RelayAccessToken**.
ConnectionMode	The transport binding element has two connection modes: • **OnewayConnectionMode.Unicast**: Unicast messaging. Used by **NetOnewayRelayBinding**. • **OnewayConnectionMode.Multicast**: Multicast messages. Used by **NetEventRelayBinding**.
TransportProtectionEnabled	This property gets or sets a Boolean value that indicates whether transport protection (SSL) is enabled for the connection. With this property set to **true**, outbound communication uses SSL through port 828; with the property set to **false**, outbound communication uses port 808.

Service Bus Port Settings

The following table describes the required configuration for port values for a Windows Azure Service Bus binding:

Binding	Transport Security	Port
Microsoft.ServiceBus.BasicHttpRelayBinding (client)	yes	HTTP
	no	HTTPS
Microsoft.ServiceBus.BasicHttpRelayBinding (service)	either	9351/HTTP
Microsoft.ServiceBus.NetEventRelayBinding (client)	yes	9351/HTTPS
	no	9350/HTTP
Microsoft.ServiceBus.NetEventRelayBinding (service)	either	9351/HTTP
Microsoft.ServiceBus.NetTcpRelayBinding (client/service)	either	9352/HTTP (9352/9353 if using Hybrid)

Microsoft.ServiceBus.NetOnewayRelayBinding (client)	yes	9351/HTTPS
	no	9350/HTTP
Microsoft.ServiceBus.NetOnewayRelayBinding (service)	either	9351/HTTP
Microsoft.ServiceBus.WebHttpRelayBinding (client)	yes	HTTPS
	no	HTTP
Microsoft.ServiceBus.WebHttpRelayBinding (service)	either	9351/HTTP
Microsoft.ServiceBus.WS2007HttpRelayBinding (client)	yes	HTTPS
	no	HTTP
Microsoft.ServiceBus.WS2007HttpRelayBinding (service)	either	9351/HTTP

Securing and Authenticating a Service Bus Connection

Applications that use the Windows Azure Service Bus are required to perform security tasks at two points. First, services exposed for use by the Service Bus are *resources*. Therefore, access to them -- whether for configuration and registration purposes or for invoking service functionality -- requires authentication and authorization using tokens from the Windows Azure Access Control service. Second, when permission to interact with the service has been granted by the Service Bus, the service has its own security considerations that are associated with the authentication, authorization, encryption, and signatures required by the message exchange itself. (This second set of security issues has nothing to do with the functionality of the Service Bus; it is purely a consideration of the service and its clients.)

In the first case, authentication and authorization to use a service exposed by the Service Bus are controlled by the Access Control service, and can be programmatically accessed through the Service Bus API. There are four kinds of authentication currently available:

- **Microsoft.ServiceBus.TransportClientCredentialType.SharedSecret**, a slightly more complex but easy-to-use form of username/password authentication.
- **Microsoft.ServiceBus.TransportClientCredentialType.Saml**, which can be used to interact with SAML 2.0 authentication systems.
- **Microsoft.ServiceBus.TransportClientCredentialType.SimpleWebToken**, which uses the OAuth Web Resource Authorization Protocol (WRAP)and Simple Web Tokens (SWT).

- **Microsoft.ServiceBus.TransportClientCredentialType.Unauthenticated**, which enables interaction with the service endpoint without any authentication behavior.

In the second case, the originating service itself typically applies some end-to-end security that specifies message-level security (such as message encryption) and transport-level security (such as Windows or NTLM). End-to-end conversation security follows the Windows Communication Foundation (WCF) programming model, and is discussed more fully in the **Securing Services** topic in the WCF documentation. Therefore, although the following topics contain a general discussion of end-to-end security, they focus mainly on the features unique to a service configured to use the Service Bus. For additional information about Service Bus authentication and Access Control, see **Building Applications that Use Access Control Services**.

Every Service Bus relay binding has a security binding element – for example, the **Microsoft.ServiceBus.Configuration.NetTcpRelaySecurityElement** performs the security functions for the **Microsoft.ServiceBus.NetTcpRelayBinding** – that contains the following security values that you can specify either programmatically or in a configuration file.

Mode

Short for end-to-end security mode, this value defines the security across the message exchange through the Service Bus. The programmatic value depends on the specific relay binding; for example, the **Microsoft.ServiceBus.EndToEndSecurityMode** type supports the **Microsoft.ServiceBus.NetTcpRelayBinding** binding, and the **Microsoft.ServiceBus.EndToEndWebHttpSecurityMode** value performs this service together with the **Microsoft.ServiceBus.WebHttpRelayBinding** binding. When used with the **Microsoft.ServiceBus.NetTcpRelayBinding** binding, this property can be set to **Microsoft.ServiceBus.EndToEndSecurityMode.None**, **Microsoft.ServiceBus.EndToEndSecurityMode.Message**, **Microsoft.ServiceBus.EndToEndSecurityMode.Transport**, or **Microsoft.ServiceBus.EndToEndSecurityMode.TransportWithMessageCredential**. The default is **Microsoft.ServiceBus.EndToEndSecurityMode.Transport**, which means that the transport-specific security settings are enabled. If you use any setting that includes **Microsoft.ServiceBus.EndToEndSecurityMode.Message** or **Microsoft.ServiceBus.EndToEndSecurityMode.Transport**, you will have to set additional properties. In general, **Mode** value follows the standard WCF security programming model.

Message

Defines security on a per-message basis if you set end-to-end message security to **Microsoft.ServiceBus.EndToEndSecurityMode.Message** or **Microsoft.ServiceBus.EndToEndSecurityMode.TransportWithMessageCredential**. Setting one of those values for the **Mode** property requires that this property also be set to specify the type of credentials that are used, and also to the algorithm that is used to help secure the credentials. As with **Mode**, the message security setting follows the WCF programming model.

Transport

This property is a wrapper for security properties unique to a given binding's transport binding element. For example, the **Microsoft.ServiceBus.RelayedOnewayTransportSecurity** class exposes and implements the **Microsoft.ServiceBus.RelayedOnewayTransportSecurity.ProtectionLevel** setting on the **Microsoft.ServiceBus.NetEventRelayBinding** and **Microsoft.ServiceBus.NetOnewayRelayBinding** bindings. In contrast, the **Microsoft.ServiceBus.HttpRelayTransportSecurity** type sets proxy credentials for **Microsoft.ServiceBus.BasicHttpRelayBinding** and **Microsoft.ServiceBus.WS2007HttpRelayBinding** bindings. As with the previous properties, **Transport** security generally follows the WCF security model.

RelayClientAuthenticationType

Controls whether clients of a service are required to present a security token issued by Access Control to the Service Bus when it sends messages. Therefore, this security property is unique to the Service Bus, and is the focus of topics in this section of the documentation. Services are always required to authenticate with Access Control and present an authorization token to the Service Bus; otherwise they cannot register endpoints or create message buffers, each of which engages Service Bus resources. However, clients are required to authenticate with the Service Bus only if the **Microsoft.ServiceBus.RelayClientAuthenticationType** is set to **Microsoft.ServiceBus.RelayClientAuthenticationType.RelayAccessToken**. Setting **Microsoft.ServiceBus.RelayClientAuthenticationType** to **Microsoft.ServiceBus.RelayClientAuthenticationType.None** waives the requirement of a token. If you are providing your own authentication or if you do not need authentication, you may want to opt out of authentication on the client (sender) in the Service Bus leg of the communication. The default value is **Microsoft.ServiceBus.RelayClientAuthenticationType.RelayAccessToken**.

In addition, every binding contains the **Scheme** property, which defines the scheme used to encode information. For HTTP-based bindings (such as **Microsoft.ServiceBus.BasicHttpRelayBinding**, the default scheme is HTTPS, which includes its own security protocols.

This section describes specific processes for using authentication on the Service Bus.

In This Section

How to: Set Security and Authentication on a Service Bus Application

Setting Security on a REST-based Service Bus Application

Choosing Authentication for a Service Bus Application

Choosing a Type of Relay Authentication

How to: Modify the Service Bus Connectivity Settings

Creating a Service Bus URI

How to: Set Security and Authentication on a Service Bus Application

This topic discusses how to authenticate a service and client application by using the Windows Azure Service Bus. For more information about setting transport and message-level security, see Securing and Authenticating a Service Bus Connection, and also the **Securing Services** topic in the Windows Communication Foundation (WCF) documentation.

If you are developing a service, you must first determine what type of credentials you will use to authenticate with the Service Bus, and whether a client that connects to your service must authenticate. All services are required to authenticate with the Service Bus, using SAML, shared secret, or a simple Web token. You may decide to have a different form of authentication for your service as you do for the client. For more information, seeChoosing Authentication for a Service Bus Application.

If you are developing a client, determine what type of authentication credentials are required by the service to which you are connecting. This can be done in a variety of ways. This includes retrieving the information from the contract metadata. For more information, seeHow to: Design a WCF Service Contract for use with the Service Bus.

To set Service Bus authentication with an App.config file

1. Define a behavior that contains the specified `<transportClientEndpointBehavior>` element, and also the relevant credentials.

   The following code, from the **WebHttpSample** in the Windows Azure SDK, shows how to declare and configure a shared secret credential.

```
<behaviors>
<endpointBehaviors>
<behavior name="sharedSecretClientCredentials">
<transportClientEndpointBehavior
credentialType="SharedSecret">
<clientCredentials>
<sharedSecret issuerName="ISSUER_NAME"
issuerSecret="ISSUER_SECRET" />
</clientCredentials>
</transportClientEndpointBehavior>
```

```
</behavior>
</endpointBehaviors>
</behaviors>
```

In this procedure, the issuer name and secret are held directly in the App.config file. It is recommended that you implement some form of security on any configuration file that contains such security information.

Once you have defined the credentials in the App.config file, the application will use the security configuration automatically. There are no additional steps necessary.

To set Service Bus authentication programmatically

1. Retrieve the security credentials:

```
Console.Write("Your Issuer Name: ");
string issuerName = Console.ReadLine();
Console.Write("Your Issuer Secret: ");
string issuerSecret = Console.ReadLine();
```

As is common in the Windows Azure SDK samples, this procedure has the issuer name and secret known by the user, and they are typed in directly. For more information about retrieving such information, see **Building Applications that Use Access Control Services**.

2. Create the credential endpoint behavior object that contains the security credentials:

```
TransportClientEndpointBehavior
sharedSecretServiceBusCredential = new
TransportClientEndpointBehavior();
sharedSecretServiceBusCredential.CredentialType =
TransportClientCredentialType.SharedSecret;
sharedSecretServiceBusCredential.Credentials.SharedSecret.Iss
uerName = issuerName;
sharedSecretServiceBusCredential.Credentials.SharedSecret.Iss
uerSecret = issuerSecret;
```

3. Create the channel factory to connect to the endpoint:

```
ChannelFactory<IEchoChannel> channelFactory = new
ChannelFactory<IEchoChannel>("RelayEndpoint", new
EndpointAddress(serviceUri));
```

4. Apply the credentials to the channel factory:

```
channelFactory.Endpoint.Behaviors.Add(sharedSecretServiceBusC
redential);
```

Once you have applied the credentials to the channel factory, you can open a connection

to the endpoint and access the Service Bus.

Setting Security on a REST-based Service Bus Application

There are two types of REST-based applications that interact with the Windows Azure Service Bus: those that use a traditional Windows Communication Foundation (WCF)-style contract and binding, and those that use a message buffer. For more information about message buffers, see Working with a Service Bus Message Buffer. Instead, this topic assumes that your application interacts with the Service Bus via the Microsoft.ServiceBus.dll assembly and the authentication features available through the Windows Azure. In particular, this topic covers applications that use the **Microsoft.ServiceBus.WebHttpRelayBinding** binding, which is the default binding for Web applications.

In the current Windows Azure release, the relay authentication options for Web clients that are accessing services built on the **Microsoft.ServiceBus.WebHttpRelayBinding** binding have been created to fit the most common scenarios. Most frequently, Web-style clients communicate with services that decide to accept all incoming traffic. These clients perform only lightweight authentication using a variety of custom techniques to enable and enrich AJAX-style user experiences. You can achieve the same result and provide similar fidelity by setting the **Security.Transport.RelayAuthenticationType** property on the **WebHttpRelayBinding** binding to **Microsoft.ServiceBus.RelayClientAuthenticationType.None**. You can see this option in the Service Bus **WebNoAuth** relay authentication sample in the Windows Azure SDK. A simplified procedure for setting this option is described later in this section.

To set authentication in a Service Bus Web application to None

1. In the service, configure the authentication as required:

```
Console.Write("Your Issuer Name: ");
string issuerName = Console.ReadLine();
Console.Write("Your Issuer Secret: ");
string issuerSecret = Console.ReadLine();
…
TransportClientEndpointBehavior clientBehavior = new
TransportClientEndpointBehavior();
clientBehavior.CredentialType =
TransportClientCredentialType.SharedSecret;
clientBehavior.Credentials.SharedSecret.IssuerName =
issuerName;
clientBehavior.Credentials.SharedSecret.IssuerSecret =
issuerSecret;
```

As with other applications, you can configure the authentication in an App.config file or programmatically.

2. Set the **RelayClientAuthenticationType** field to **None**.

```
<bindings>
<!-- Application Binding -->
<webHttpRelayBinding>
<binding name="default">
<security relayClientAuthenticationType="None" />
</binding>
</webHttpRelayBinding>
</bindings>
```

This allows the service to authenticate with the Service Bus (as required), but also enables any client to connect, without authentication required. In this scenario, the App.config file defines the type of security to use for the whole scenario, but the programmatic configuration (in step 1) overrides the App.config file – which is necessary, because it is impossible to have “None” for service authentication.

If you use the **RelayAccessToken** option for the **Microsoft.ServiceBus.Configuration.TcpRelayTransportElement.RelayClientAuthenticationType** property, the Service Bus provides a security layer over plain HTTP services that require authentication and authorization to be performed before any HTTP traffic is forwarded to the listening service. If **Relay** authentication is enabled on the Service Bus, the required security token can be provided through programmatic credentials.

If you decide to implement programmatic credentials, you can use any of the authentication options available to Service Bus through the Access Control service, such as shared secret or simple Web tokens. For more information, seeHow to: Set Security and Authentication on a Service Bus Application. The following procedure shows a simplified procedure for creating a Web token.

To programmatically create a simple Web token

1. Retrieve the issuer name and secret from the user:

```
Console.Write("Your Issuer Name: ");
string issuerName = Console.ReadLine();
Console.Write("Your Issuer Secret: ");
string issuerSecret = Console.ReadLine();
```

2. Define the transport client credential type as **Microsoft.ServiceBus.TransportClientCredentialType.SimpleWebToken**:

```
TransportClientEndpointBehavior behavior = new
TransportClientEndpointBehavior();
```

```
behavior.CredentialType =
TransportClientCredentialType.SimpleWebToken;
```

3. Compute and initialize the Web token with a call to **Microsoft.ServiceBus.Description.SharedSecretCredential.ComputeSimpleWebTokenString(System.String,System.String)**:

```
behavior.Credentials.SimpleWebToken.SimpleWebToken =
SharedSecretCredential.ComputeSimpleWebTokenString(issuerName
, issuerSecret);
```

When you have created the Web token, you can add the behavior to the endpoint, create the channel factory, and open a channel to the Service Bus.

Choosing Authentication for a Service Bus Application

The **Microsoft.ServiceBus.TransportClientEndpointBehavior** behavior is a Windows Communication Foundation (WCF) class that is used to specify the Windows Azure Service Bus authentication credentials for a particular endpoint. Instances of this behavior are shareable across endpoints so that the descriptions of multiple endpoints (listener and channels) using the same Service Bus credentials can be populated with the same configured instance of this class.

The behavior can be defined and applied to endpoints in code and in configuration files.

T:Microsoft.ServiceBus.TransportClientEndpointBehavior **members**	**Description**
Microsoft.ServiceBus.Configuration.TransportClientEndpointBehaviorElement.CredentialType	The **Microsoft.ServiceBus.Configuration.TransportClientEndpointBehaviorElement.CredentialType** property specifies which authentication method will be used on the endpoint. The possible values for this property are as follows: • **Microsoft.ServiceBus.TransportClientCredentialType.Saml**: this option specifies that the client credential is provided in the Security Assertion Markup Language (SAML) format, over the Secure Sockets Layer protocol. This option requires that you write your own SSL credential server. • **Microsoft.ServiceBus.TransportClientCredentialType.SharedSecret**: This option specifies that the client credential is provided as a self-issued shared secret that is registered with Access Control through

T:Microsoft.ServiceBus.TransportClientEndpointBehavior **members**	**Description**
	the Windows Azure portal. This option requires no additional settings on the **Microsoft.ServiceBus.TransportClientEndpointBehavior.Credentials** property. • **Microsoft.ServiceBus.TransportClientCredentialType.SimpleWebToken**: This option specifies that the client credential is provided as a self-issued shared secret that is registered with Access Control through the Windows Azure portal, and presented in the emerging industry-standard format called simple Web token (SWT). Similar to the shared secret option, this option requires no additional settings on the **Microsoft.ServiceBus.TransportClientEndpointBehavior.Credentials** property. • **Microsoft.ServiceBus.TransportClientCredentialType.Unauthenticated**: This option specifies that there is no client credential provided. This option avoids acquiring and sending a token. It is used by clients that are not required to authenticate, based on the policy of their service binding. Note that this setting might leave data nonsecure if not used together with another security measure.
Microsoft.ServiceBus.TransportClientEndpointBehavior.Credentials	This property refers to the composite credentials container **Microsoft.ServiceBus.Description.TransportClientCredentials**, which holds the specific credentials for the credential types described earlier in this section.

Choosing a Type of Relay Authentication

The **Microsoft.ServiceBus.RelayClientAuthenticationType** enumeration is referenced by the security settings in all of the relay bindings. The use of this property is identical throughout all bindings. The following table lists the possible values for this enumeration.

T:Microsoft.ServiceBus.RelayClientAuthenticationType **value**	**Description**
Microsoft.ServiceBus.RelayClientAut	The client is required to provide a relay

T:Microsoft.ServiceBus.RelayClientAuthenticationType value	Description
henticationType.RelayAccessToken	access token to access the service endpoint, and access control is performed by the Windows Azure Access Control service. If this option is set on the service binding, all clients must acquire and present tokens to the Service Bus when establishing the channel. Furthermore, all subsequent access control is delegated to Access Control. The relay access token may be a shared secret, simple Web access token, or a SAML token. This is the default value.
Microsoft.ServiceBus.RelayClientAuthenticationType.None	The client is not required to provide a relay access token. Services are not required to present access tokens at any time. Therefore,, this represents an opt-out mechanism with which services can waive the Access Control protection on the endpoint and perform their own access control.

When the **Microsoft.ServiceBus.RelayClientAuthenticationType.RelayAccessToken** option is chosen, all access control management is delegated to the Access Control service. Access Control yields an access control token for the relay that indicates whether the requestor can listen on or send to the relay (or both). At the same time, it protects the actual identity of the caller. Therefore, the listening service will not be able to gather any user-specific information. Effectively, the service operates in an anonymous authentication mode, trusting the Access Control service.

The **Microsoft.ServiceBus.RelayClientAuthenticationType.None** option causes the relay to pass all incoming messages to the service. The service assumes the responsibility for performing all access control locally, and also more precise access control. The type of access control depends on business data, local credentials stores, or other criteria—but it potentially exposes the service to unwanted traffic.

The hybrid solution is to combine the **Microsoft.ServiceBus.RelayClientAuthenticationType.RelayAccessToken** option with end-to-end message security, which is an option supported by most bindings. In this combination, the client is required to provide two separate credentials: one via the **Microsoft.ServiceBus.TransportClientEndpointBehavior** behavior, for gaining access to the service through the Service Bus. The latter is specified on the regular **Microsoft.ServiceBus.Configuration.TransportClientEndpointBehaviorElement.ClientCrede**

ntials property of the channel factory, which is used to help secure the end-to-end communication path. The latter credential can be a token issued by Access Control in the scope of the solution.

You are required to use message-security for end-to-end authorization. This is because the Service Bus bindings do not support the use of standard WCF transport-credentials. For most uses, transport-credentials can only be passed point-to-point on a connection, and cannot traverse intermediaries such as the Service Bus.

How to: Modify the Service Bus Connectivity Settings

The **Microsoft.ServiceBus.ConnectivitySettings** class contains settings effective for all endpoints, based on the **Microsoft.ServiceBus.NetOnewayRelayBinding** or **Microsoft.ServiceBus.NetEventRelayBinding** bindings, that are active in the current application domain. The reason for the shared nature of these settings is that the connectivity path to the Service Bus is identical across all endpoints in the same process. Most corporate network environments prefer to limit ports opened to outbound traffic, and typically restrict outbound HTTP and TCP traffic to the same narrow range of ports.

By default, all service endpoints listening for messages using one of these two bindings connect to the Service Bus using outbound TCP port 828 (for SSL-protected connections) or outbound TCP port 808.

If neither of these ports is available for outbound communication, the **Microsoft.ServiceBus.ServiceBusEnvironment.SystemConnectivity** property of the connectivity settings can be set to **Microsoft.ServiceBus.ConnectivityMode.Http**, which enables the HTTP polling through outbound ports 80 and 443 using RFC 2616-compliant HTTP requests. RFC 2616 strongly recommends constraining the concurrent requests to a particular domain be limited to two, and the operating system and networking devices or upstream proxies can enforce that limit. Considering that, the HTTP polling mode is using a single HTTP connection to implement polling. All messages destined for all one-way and event endpoints in the current application domain are multiplexed through the HTTP polling connection and distributed locally.

To set or modify the Service Bus connectivity settings

1. Set or modify the connectivity settings with a call to **Microsoft.ServiceBus.ServiceBusEnvironment.SystemConnectivity**.

```
ServiceBusEnvironment.SystemConnectivity.Mode =
ConnectivityMode.Http;
```

Example

Description

The following example, taken from the **Echo** sample in the Windows Azure SDK, describes how to set the connectivity mode in a command-line application.

Code

```
ServiceBusEnvironment.SystemConnectivity.Mode = ConnectivityMode.Http;


Console.Write("Your Service Namespace (ex.
sb://<ServiceNamespace>.servicebus.windows.net/): ");
string serviceNamespace = Console.ReadLine();
Console.Write("Your Issuer Name: ");
string issuerName = Console.ReadLine();
Console.Write("Your Issuer Secret: ");
string issuerSecret = Console.ReadLine();


// Create the service URI based on the service namespace name.
Uri address = ServiceBusEnvironment.CreateServiceUri("sb", serviceNamespace,
"EchoService");
```

Creating a Service Bus URI

A Windows Azure Service Bus URI is a universal resource identifier (URI) that describes the location at which an endpoint is exposed on the Service Bus. You use an Service Bus URI when you connect with both a service and client application. You also use an Service Bus URI when you connect to retrieve tokens for authentication and credential verification with the Access Control service. While you can manually create the URI, or store it in the App.config file, we recommend that you use the **Microsoft.ServiceBus.ServiceBusEnvironment.CreateServiceUri(System.String,System.String,System.String)** or **Microsoft.ServiceBus.ServiceBusEnvironment.CreateAccessControlUri(System.String)** methods to create your URIs, as those members contain all the relevant default settings.

Note

The Service Bus performs case-insensitive comparisons of service namespaces to align with the behavior of iis60. Because Access Control is designed as a general purpose access control service, it performs case-sensitive comparisons of service namespaces and scopes. Thus, applications that rely on Access Control can decide to be case sensitive or case insensitive, depending on the needs of that application. When designing applications that support multitenancy, you should realize that Access Control performs case-sensitive comparisons and Service Bus performs case-insensitive comparisons, if this difference in behavior produces unexpected results in your application. Because Access Control prevents creating scopes that differ only by case, this difference in behavior should not present a security issue.

To create an Access Control or Service Bus URI

1. Create the URI with a call to **Microsoft.ServiceBus.ServiceBusEnvironment.CreateServiceUri(System.String,System.String,System.String)** or **Microsoft.ServiceBus.ServiceBusEnvironment.CreateAccessControlUri(System.String)**, respectively.

```
Uri address = ServiceBusEnvironment.CreateServiceUri("sb",
solutionName, "EchoService");
```

Example

Description

The following example, taken from the **Echo** sample in the Windows Azure SDK, shows how to programmatically create an Service Bus URI.

Code

```
Console.Write("Your Service Namespace (ex.
sb://<ServiceNamespace>.servicebus.windows.net/): ");
string serviceNamespace = Console.ReadLine();
Console.Write("Your Issuer Name: ");
string issuerName = Console.ReadLine();
Console.Write("Your Issuer Secret: ");
string issuerSecret = Console.ReadLine();


// create the service URI based on the solution name
Uri address = ServiceBusEnvironment.CreateServiceUri("sb", serviceNamespace,
"EchoService");
```

Building a Service for the Service Bus

Unless you are building a non-Windows Communication Foundation (WCF) REST application that creates a message buffer, using the Windows Azure Service Bus requires creating and then *hosting* a Web service that registers itself with the Service Bus. (For more information about using a message buffer, see Service Bus Message Buffer Overview.)

Hosting an application means instantiating and running a service that is configured to use a relay binding that connects to the Service Bus. Therefore,, the hosted service is the originating service, which the application registers with the Service Bus. Clients then use the service endpoint exposed by the Service Bus, which relays authorized messages to and from the originating service regardless of where it is physically located.

Note that, before you actually write the code that instantiates and runs a service, you must perform several steps. The following procedure describes the setup work necessary before hosting an Service Bus service.

To set up a service for hosting

1. Design the WCF contract for your service, as described in Designing a WCF Contract for the Service Bus. The contract, as a WCF interface, is the same for both service and client applications.
2. Implement the WCF contract for your service. The implementation of the contract is used later, as part of the hosting process.
3. Configure your service, as defined in Configuring a WCF Service to Register with the Service Bus. Configuration can be done programmatically or through the App.config file. Common scenarios include defining the service endpoint and security in the App.config file. These values are used implicitly or explicitly later when the endpoints are created.
4. Create the authorization and authentication credentials, as defined in Securing and Authenticating a Service Bus Connection. As in the previous step, the credentials can be defined either programmatically or in the App.config file.

After completing these steps, you can host your service.

In This Section

The following topics describe the most common hosting scenarios and contain information to help with common issues related to hosting.

How to: Host a WCF Service that Uses the Service Bus Service

This topic describes how to create a WCF service host using the Windows Azure SDK.

How to: Host a Service on Windows Azure that Accesses the Service Bus

This topic describes how to create a WCF service host using the Windows Azure SDK, that you can run in Windows Azure.

How to: Create a REST-based Service that Accesses the Service Bus

This topic describes how to use the WCF Web Programming Model to create and host a WCF REST-style Web service that registers a REST service endpoint with the Service Bus.

How to: Use a Third Party Hosting Service with the Service Bus

This topic describes how to create and host with a third-party hosting system a WCF Web service that registers a service endpoint with the Service Bus.

Hosting Behind a Firewall with the Service Bus

This topic describes some of the important items to remember when hosting behind a firewall.

See Also

How to: Expose a Metadata Endpoint

How to: Host a WCF Service that Uses the Service Bus Service

Hosting the service is the final step in creating a Windows Azure Service Bus application. Before reaching this point, you will have defined and implemented the service contract, defined and configured the service endpoint, and created the security credentials. For more information about what you must do before hosting the application, see Building a Service for the Service Bus. The next step is to put all these separate parts together and get them running. This process is accomplished through the service host, which takes the URL of your project, together with the contract, and creates a connection to the Service Bus.

The first procedure describes how to create a service that uses the Service Bus with the configuration settings defined programmatically. The second procedure shows how to create a service when most of the configuration is specified in the App.config file. This procedure follows the **NetOneWay** sample in the Windows Azure SDK. For a hybrid approach that uses both programmatic configuration and also an App.config file, see steps 1 through 4 of the Service Bus Relayed Messaging Tutorial.

For a complete discussion of hosting an application, see **Hosting Services** in the Windows Communication Foundation (WCF) documentation.

To host a Service Bus service programmatically

1. Create an address for your service.

   The address for your Service Bus project is used in both service and client applications. For a service, the URI is used to determine where the Service Bus exposes the service. For a client, the URI determines where the client looks for the service:

   ```
   string servicePath = "ServicePath";
   string serviceNamespace = "ServiceNamespace";
   Uri uri = ServiceBusEnvironment.CreateServiceUri("sb",
   serviceNamespace, servicePath);
   ```

2. Create a new instance of **System.ServiceModel.ServiceHost**.

   ```
   host = new ServiceHost(typeof(EchoService), uri);
   ```

   In this example, the service host takes the supplied address, in addition to the type that the service contract implements. In this example, the class that implements the service

contract is named `EchoService`.

3. Create a description of the contract with a call to **System.ServiceModel.Description.ContractDescription**.

```
ContractDescription contractDescription =
ContractDescription.GetContract(typeof(IEchoContract),
typeof(EchoService));
```

System.ServiceModel.Description.ContractDescription links the contract with the specific implementation you want to use. In this example, the contract is defined in `IEchoContract`, and the implementation of the contract is `EchoService`.

4. Define the address and binding for the endpoint:

```
ServiceEndpoint serviceEndPoint = new
ServiceEndpoint(contractDescription);
serviceEndPoint.Address = new EndpointAddress(uri);
serviceEndPoint.Binding = new NetTcpRelayBinding();
```

5. Add any additional behaviors to the endpoint, such as security or publishing behaviors:

```
serviceEndPoint.Behaviors.Add(sharedSecretServiceBusCredentia
l);
```

In this code sample, `sharedSecretServiceBusCredential` had previously been created to store the security credentials.

6. Add the service endpoint to the service host instance. This step indicates which endpoint you want to instantiate.

```
host.Description.Endpoints.Add(serviceEndPoint);
```

7. Open the service by using a call to `ServiceHost.Open`.

If successful, the service will be available for a client application to contact and communicate with through the Service Bus without additional action required. However, you may want to perform additional tasks, such as notifying the user that the host has succeeded.

```
host.Open();

Console.WriteLine(String.Format("Listening at: {0}",
endPoint));
Console.WriteLine("Press [Enter] to exit");
Console.ReadLine();
```

8. When you are finished, close the host with `ServiceHost.Close`.

```
host.Close();
```

To host a Service Bus service that uses an App.config file

1. Create the name of the project to expose on the Service Bus:

```
string serviceBusProjectName = "myProjectNameHere";
```

2. Create the URI for your service:

```
Uri address = ServiceBusEnvironment.CreateServiceUri("sb",
serviceBusProjectName, "OnewayService");
```

3. Create a new instance of the **ServiceHost**.

```
ServiceHost host = new ServiceHost(typeof(LogService), uri);
```

   Here, the service host takes the supplied address, and also the type that the service contract implements. In this example, the class that implements the service contract is named `LogService`.

4. If successful, the service will be available for a client application to contact and communicate with through the Service Bus without additional action required.

```
host.Open();


Console.WriteLine("Press [Enter] to exit");
Console.ReadLine();
```

5. When you are finished, close the host with `ServiceHost.Close`.

```
host.Close();
```

6. In your App.config file, add the credential and binding information for your project.

   An example App.config file that contains this information is located in the code sample at the end of this topic.

Example

Description

The following example shows how to programmatically define and create a service application.

Code

```
class Program
{
    static void Main(string[] args)
    {
        string servicePath = "ServicePath";
        string serviceNamespace = "ServiceNamespace";
        string issuerName = "IssuerName";
```

```
            string issuerSecret = "IssuerSecret";

            // Construct a Service Bus URI
            Uri uri = ServiceBusEnvironment.CreateServiceUri("sb", serviceNamespace,
servicePath);

            // Create a Behavior for the Credentials
            TransportClientEndpointBehavior sharedSecretServiceBusCredential = new
TransportClientEndpointBehavior();
            sharedSecretServiceBusCredential.CredentialType =
TransportClientCredentialType.SharedSecret;
            sharedSecretServiceBusCredential.Credentials.SharedSecret.IssuerName =
issuerName;
            sharedSecretServiceBusCredential.Credentials.SharedSecret.IssuerSecret =
issuerSecret;

            // Create the Service Host
            host = new ServiceHost(typeof(EchoService), uri);
            ContractDescription contractDescription =
ContractDescription.GetContract(typeof(IEchoContract), typeof(EchoService));
            ServiceEndpoint serviceEndPoint = new ServiceEndpoint(contractDescription);
            serviceEndPoint.Address = new EndpointAddress(uri);
            serviceEndPoint.Binding = new NetTcpRelayBinding();
            serviceEndPoint.Behaviors.Add(sharedSecretServiceBusCredential);
            host.Description.Endpoints.Add(serviceEndPoint);

            host.Open();

            Console.WriteLine(String.Format("Listening at: {0}", endPoint));
            Console.WriteLine("Press [Enter] to exit");
            Console.ReadLine();

            host.Close();
        }
    }
```

```
//Service that is configured mainly with an App.config file
   class Program
    {
        static void Main(string[] args)
        {
            string serviceNamespace = GetServiceNamespace();
            Uri address = ServiceBusEnvironment.CreateServiceUri("sb", serviceNamespace,
"OnewayService");

            ServiceHost host = new ServiceHost(typeof(OnewayService), address);
            host.Open();

            Console.WriteLine("Press [Enter] to exit");
            Console.ReadLine();

            host.Close();
        }
    }


//App.config file associated with the previous code sample

<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.serviceModel>
<behaviors>
<endpointBehaviors>
<behavior name="sharedSecretClientCredentials">
<transportClientEndpointBehavior credentialType="SharedSecret">
<clientCredentials>
```

```
<sharedSecret issuerName="ISSUER_NAME" issuerSecret="ISSUER_SECRET" />
</clientCredentials>
</transportClientEndpointBehavior>
</behavior>
</endpointBehaviors>
</behaviors>
<bindings>
<!-- Application Binding -->
<netOnewayRelayBinding>
<binding name="default" />
</netOnewayRelayBinding>
</bindings>
<services>
<service name="Microsoft.ServiceBus.Samples.OnewayService">
<endpoint address="" behaviorConfiguration="sharedSecretClientCredentials"
          binding="netOnewayRelayBinding" bindingConfiguration="default"
          name="RelayEndpoint" contract="Microsoft.ServiceBus.Samples.IOnewayContract" />
</service>
</services>

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

See Also

Hosting a WCF Service in IIS

How to: Host a Service on Windows Azure that Accesses the Service Bus

Creating a service application that runs on Windows Azure follows the programming patterns for both a Windows Azure and a traditional Windows Azure Service Bus application: you define and implement the service contract, configure the endpoint, create the credentials, and then start the host. Once you are finished, you stop the host. However, note that there are three unique issues when you create a service in Windows Azure:

- **Windows Azure does not have the Service Bus assembly pre-installed**

Windows Azure is not integrated with Windows Azure. Therefore,, Windows Azure does not install the Service Bus Bus assembly. Due to Windows Azure security restrictions, you cannot install the Windows Azure SDK on the Windows Azure platform. Therefore, to run a Service Bus application on Windows Azure, you must redistribute the Service Bus assembly with your Service Bus application. For more information about packaging an assembly with your application, see the following procedure.

- **Windows Azure does not store Service Bus and Access Control configuration information in the Machine.config file**

  Because Windows Azure does not install the Windows Azure SDK, the Machine.config file on a Windows Azure computer has no information about Service Bus bindings or endpoints. As stated previously, Windows Azure security restrictions prevent you from modifying the Windows Azure Machine.config file. Therefore, there are two options to make Service Bus and Access Control configuration information available to your Service Bus applications.

  a. The recommended solution is to use the Service Bus APIs to programmatically configure your application. For example, although you could store name and password information in the App.config file, you would programmatically set any relay binding configurations. For more information about setting configuration programmatically, see Configuring a WCF Service to Register with the Service Bus.

  b. The second solution is to manually modify the App.config file for your application by adding all of the relevant information. Once you do this, you can use the App.config file to configure bindings and endpoints. To do so, you can see the Machine.config file on a computer that has the Windows Azure SDK installed, find all Windows Azure-related configuration information, and copy them to your application App.config file. While this lets you use the App.config file on the host service, it will be difficult to test your code: you may encounter duplication issues with the Machine.config file of the local test computer, which will already have the Windows Azure SDK installed. Therefore, we recommend that you use the previous option, and set everything programmatically.

- The **Service Bus Service Bus must have Full Trust authorization to run on Windows Azure**

  As with all other Service Bus applications, you must make sure that the operating system is running with Full Trust authorization. This can be set in the ServiceDefinition.csdef file of your Windows Azure project, using the following procedure.

To package the Service Bus assembly with your application

1. In **Solution Explorer**, under the **WorkerRole** or **WebRole** node (depending on where your code is located), add the **Microsoft.ServiceBus.dll** assembly to your Windows Azure project as a reference.

   This step is the standard process for adding a reference to an assembly.
2. In the **Reference** folder, right-click **Microsoft.ServiceBus**. Then click **Properties**.
3. In the **Properties** dialog, set **Copy Local** to **True**.

   Doing so makes sure that the Microsoft.ServiceBus.dll assembly is copied to the local \bin path and available to your application when it is running on Windows Azure.

To set the Windows Azure application to Full Trust

1. In the ServiceDefinition.csdef file in your project, set the `enableNativeCodeExecution` field to `"true"`, as shown in the following code. Replace `"ApplicationNameHere"` with the name of your application:

```
<?xml version="1.0" encoding="utf-8"?>
<ServiceDefinition name="ApplicationNameHere"
xmlns="http://schemas.microsoft.com/ServiceHosting/2008/10/Se
rviceDefinition">
<WebRole name="WebRole" enableNativeCodeExecution="true">
<InputEndpoints>
<!-- Must use port 80 for http and port 443 for https when
running in the cloud -->
<InputEndpoint name="HttpIn" protocol="http" port="80" />
</InputEndpoints>
</WebRole>
<WorkerRole name="WorkerRole" enableNativeCodeExecution="true">
</WorkerRole>
</ServiceDefinition>
```

How to: Create a REST-based Service that Accesses the Service Bus

Windows Azure currently supports two different styles of applications that qualify as REST-based services: a traditional Service Bus application that complies with the Web programming model, and an HTTP-compliant application that uses a message buffer.

- **Traditional application**

  This style of application uses the basic Windows Communication Foundation (WCF) programming model: defining and creating a service contract, using a binding and security credentials to connect to the Service Bus, and so on. The main addition is that a REST-based Service Bus application uses a service contract to whose members the `[OperationContract, WebGet]` or `[OperationContract, WebInvoke]` attributes are applied. These behaviors define the interface as a REST interface, and allow the Service Bus to interact with other REST-style applications. And therefore, the applications contain additional code that enables them to build HTTP-style messages. Finally, all these applications use the **Microsoft.ServiceBus.WebHttpRelayBinding** binding. For more information, seeHow to: Expose a REST-based Web Service Through the Service Bus. For an extended example of a

REST-based service application, see the Service Bus Message Buffer Tutorial, which is in turn based on the **WebHttp** sample in the Windows Azure SDK.

- **Message Buffer applications**

  The *Message Buffer* is an Service Bus feature that exposes a REST interface to a buffer location. Sender applications can use this message buffer to temporarily store messages and events, similar to any other buffer. Similarly, client applications can subscribe to the message buffer to receive stored messages or events. Because the interface is exposed through the Service Bus, it is available to any application that can connect to the Internet. Because the message buffer is a REST-style interface, you can connect to it using non-WCF style applications, such as JavaScript applications, Web browsers, and other non-Microsoft products. This includes writing services in JavaScript. However, because this technology is so different from a typical Service Bus application, it has its own section of the documentation. For more information about creating applications that interact with the message buffer, see Service Bus Message Buffer Overview.

Because message buffers have a dedicated section in the documentation, this topic focuses mainly on the details of creating a basic Service Bus application that uses the REST standard. The process of hosting a REST-based Service Bus application is very similar to hosting a standard Service Bus application. The main differences are in the contract and configuration: the actual hosting process is basically the same.

To host a Service Bus service application that complies with the REST standard

1. Create the service by using the standard pattern as defined in Building a Service for the Service Bus; that is, define and implement a service contract, configure and implement the service host, and so on.
   a. When applying the **System.ServiceModel.OperationContractAttribute** attribute to the service contract, make sure that you apply the relevant attributes to identify the REST-based members. For more information, see How to: How to: Expose a REST-based Web Service Through the Service Bus.

      The following example code shows how to tag an interface member as a REST-style GET member.

      ```
      public interface IImageContract
      {
          [OperationContract, WebGet]
          Stream GetImage();
      }
      ```

   b. When implementing the contract, set the appropriate content type header for the outgoing Web responses, as defined by the needs of your application.

      ```
      public ImageService()
      {
          this.bitmap = Image.FromFile(imageFileName);
      ```

```
    }

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

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

        return stream;
    }
```

2. Create the address for the service by using the **Microsoft.ServiceBus.ServiceBusEnvironment.CreateServiceUri(System.String,System.String,System.String)** method:

```
string serviceNamespace = "myServiceNamespace"
Uri address = ServiceBusEnvironment.CreateServiceUri("https",
serviceNamespace, "Image");
```

3. Create a new instance of **System.ServiceModel.Web.WebServiceHost**.

```
WebServiceHost host = new
WebServiceHost(typeof(ImageService), address);
```

   The **System.ServiceModel.Web.WebServiceHost** class is derived from the **System.ServiceModel.ServiceHost** class, and complements the WCF Web programming model. It also makes it easier to host REST-based services. It is recommended that you use **System.ServiceModel.Web.WebServiceHost** instead of **System.ServiceModel.ServiceHost** in your REST-based Service Bus application implementation. For more information, see**WCF REST Programming Model** in the WCF documentation.

4. Specify the address, binding, and contracts (also known as the "ABCs") used by the service endpoint.

```
<services>
<!-- Application Service -->
<service name="Microsoft.ServiceBus.Samples.ImageService"
```

```
                behaviorConfiguration="default">
<endpoint name="RelayEndpoint"

contract="Microsoft.ServiceBus.Samples.IImageContract"
                  binding="webHttpRelayBinding"
                  bindingConfiguration="default"

behaviorConfiguration="sharedSecretClientCredentials"
                  address="" />
</service>
```

Here, the ABC is linked to the endpoint in the App.config file. For more information about configuring an application, see Configuring a WCF Service to Register with the Service Bus.

The only binding to use for a service endpoint in a REST-based Service Bus application is **Microsoft.ServiceBus.WebHttpRelayBinding**.

5. If necessary, disable client authentication.

```
<bindings>
<!-- Application Binding -->
<webHttpRelayBinding>
<binding name="default">
<security relayClientAuthenticationType="None" />
</binding>
</webHttpRelayBinding>
</bindings>
```

By default, the **Microsoft.ServiceBus.WebHttpRelayBinding** binding requires client authentication. This step describes how to disable it in the `<binding>` element in the App.config file, so that the client does not have to present credentials (for example, when you use a browser). For more information about authenticating with the Service Bus, see Securing and Authenticating a Service Bus Connection.

6. Define the security for your application:

```
<behaviors>
<endpointBehaviors>
<behavior name="sharedSecretClientCredentials">
```

```
<transportClientEndpointBehavior
credentialType="SharedSecret">
<clientCredentials>
<sharedSecret issuerName="ISSUER_NAME"
issuerSecret="ISSUER_SECRET" />
</clientCredentials>
</transportClientEndpointBehavior>


</behaviors>
```

In this example, the security is defined in the App.config file. For more information about security, see Securing and Authenticating a Service Bus Connection.

7. Open the Service with a call to `WebServiceHost.Open`:

```
host.Open()
```

8. When you are finished, close the host with `WebServiceHost.Close`.

```
host.Close();
```

How to: Use a Third Party Hosting Service with the Service Bus

It is possible to run an application that uses the Windows Azure Service Bus on a third-party hosting platform. There are no special settings that you must use in order to deploy your application on the hosting service, other than what the hosting service requires. There are also no special security requirements for accessing the Service Bus from a third-party system. However, in order to get your application to run correctly, there are two things to note, which are very similar to running an application on Windows Azure:

- **The hosting service may not install the Windows Azure SDK**

  If the hosting service does not have the Windows Azure SDK installed, you cannot know for sure that the Microsoft.ServiceBus.dll assembly is available for your application to use. Therefore, you must make sure that the appropriate assembly is packaged and redistributed with your application. To do so, see the following procedure.

- **The hosting service may not have the appropriate listings in the Machine.config file**

  Because the third-party hosting service may not have the Windows Azure SDK installed, the Machine.config file on the host has no information about Service Bus bindings or endpoints. Due to security restrictions on many hosting services, you will likely not be able to install the SDK on the host computer in order to add those configuration elements to the Machine.config file. Therefore, the App.config file for your Service Bus application will likely not have any information specific to Windows Azure in it.

  There are two solutions to this issue.

a. The recommended solution is to use the Windows Azure APIs to programmatically configure your application. For example, although you could store name and password information in the App.config file, you would programmatically set any relay binding configurations. For more information about setting the configuration programmatically, see Configuring a WCF Service to Register with the Service Bus.

b. The second solution is to manually modify the App.config file for your application by adding all of the relevant Service Bus information. Once you do this, you can use the App.config file to configure bindings and endpoints. To do so, you can see the Machine.config file on a computer that has the Windows Azure SDK installed, find all Windows Azure related configuration information, and copy it to the App.config file for your application. Although this will let you use the App.config file on the host service, it will be difficult to test your code: you may experience duplication issues with the Machine.config file on the local test computer, which will already have the Windows Azure SDK installed. Therefore, we recommend that you use the previous option, and set everything programmatically.

To package the Service Bus assembly with your application

1. In **Solution Explorer**, add the **Microsoft.ServiceBus.dll** assembly to your project as a reference.

   This step is the standard process for adding a reference to an assembly.
2. In the **Reference** folder, right-click **Microsoft.ServiceBus**. Then click **Properties**.
3. In the **Properties** dialog, set **Copy Local** to **True**.
4. Doing so makes sure that the Microsoft.ServiceBus.dll assembly is copied to the local \bin path and available to your application when it runs on the host service.

Hosting Behind a Firewall with the Service Bus

This topic describes several ways to connect to the Windows Azure Service Bus from behind a firewall or through a proxy server.

Troubleshoot your Firewall Connection

The following troubleshooting topics discuss common solutions to problems encountered when you connect through a firewall to the Service Bus.

Configure the Ports on your Firewall

To use the Service Bus relay, ensure that your firewall allows outgoing TCP communication on TCP ports 9350 to 9354. For Service Bus brokered messaging, use port 9354.

Configure the WinHTTP Proxy Settings

If you are running behind a firewall/proxy that requires authentication, or if you are running in an IPsec-protected network, there are additional obstacles for any client to reach the network proxy. For example, Windows accounts might not have permissions to communicate through the

firewall. Therefore, you might have to explicitly configure the WinHTTP proxy settings with the appropriate credentials.

Set OpenTimeout

Setting the connectivity mode to HTTP (that is, `ConnectivityMode = http`) may cause connections in the presence of some proxies to be very slow. For example, some connections can require up to 20 seconds to connect. Extending the **OpenTimeout** option for the service to up to two minutes can help, because you might run out of time between the acquisition of the Access Control token and getting the Web stream working. After the Web stream is established, the throughput often improves.

Building a Service Bus Client Application

The following topic describes how to build a client application that connects to and uses a Windows Azure Service Bus service endpoint to communicate with the originating service.

In This Section

How to: Create a WCF SOAP Client Application for the Service Bus

This topic describes how to create a SOAP-based client application that uses Windows Communication Foundation (WCF).

Creating a REST-based Client Application for the Service Bus

This topic describes how to create one REST-based client application that uses the Windows Communication Foundation (WCF) Web programming model and the Windows Azure SDK assemblies and one application without them.

Creating a Windows Azure Client for the Service Bus

This topic describes the mechanisms you can use to access a SOAP- or REST-based service from a Windows Azure client application.

See Also

Building a Service Bus Client Application

How to: Create a WCF SOAP Client Application for the Service Bus

The following topic describes how to create a traditional client application that accesses the Windows Azure Service Bus. For a complete discussion of building a client application, see **Building Clients** in the Windows Communication Foundation (WCF) documentation. The

following procedure is a simplified process for creating a client application that highlights the features unique to the Service Bus. For a complete sample, see the **Echo** sample in the Windows Azure SDK, or Step 5: Create a WCF Client for the Service Contract in the Service Bus Relayed Messaging Tutorial.

To create a Service Bus client application

1. Retrieve a copy of the contract of the service and include it in your code:

```
using System;
using System.ServiceModel;

[ServiceContract(Name = "IEchoContract", Namespace =
"http://samples.microsoft.com/ServiceModel/Relay/")]
public interface IEchoContract
{
    [OperationContract]
    string Echo(string text);
}

public interface IEchoChannel : IEchoContract, IClientChannel
{ }
```

   You can retrieve the contract from the service in a variety of ways, such as through metadata exposed by the service. For more information, seeHow to: Design a WCF Service Contract for use with the Service Bus.

2. Add references to the **System.ServiceModel** and **Microsoft.ServiceBus** namespaces to your project:

```
using System.ServiceModel;
using Microsoft.ServiceBus;
```

3. Retrieve your service namespace and relevant credential information:

```
static void Main(string[] args)
{
    Console.Write("Your Service Namespace (ex.
sb://<ServiceNamespace>.servicebus.windows.net/): ");
    string serviceNamespace = Console.ReadLine();
    Console.Write("Your Issuer Name: ");
```

```
        string issuerName = Console.ReadLine();
        Console.Write("Your Issuer Secret: ");
        string issuerSecret = Console.ReadLine();
    }
```

Note

The previous code example assumes that the service endpoint requires issuer name and secret credentials. Service endpoints may not require authentication; if one did not require authentication, setting the issuer name and secret would also not be necessary.

The type of security and authentication that is required to connect is defined by the service. You can retrieve this information in a variety of ways, such as through metadata that the service exposes. For more information, seeHow to: Design a WCF Service Contract for use with the Service Bus. This topic assumes that the Service Bus endpoint requires client applications to authenticate and uses issuer name and secret credentials. (An endpoint may not require any authentication, although this example does.) You can also use other credential types, such as a simple Web token (SWT), or SAML. At this point, you can also set the client transport or message-level security. However, for many scenarios, the default settings are sufficient. For more information, seeSecuring and Authenticating a Service Bus Connection.

4. Define the security credentials to use with the endpoint:

```
TransportClientEndpointBehavior
sharedSecretServiceBusCredential = new
TransportClientEndpointBehavior();
sharedSecretServiceBusCredential.CredentialType =
TransportClientCredentialType.SharedSecret;
sharedSecretServiceBusCredential.Credentials.SharedSecret.Iss
uerName = issuerName;
sharedSecretServiceBusCredential.Credentials.SharedSecret.Iss
uerSecret = issuerSecret;}
```

Note

The previous code example assumes that the service endpoint requires issuer name and secret credentials. Service endpoints may not require authentication; if authentication is not required, skip this step and step 8, later in this section.

5. Create a URI object pointing to the Service Bus service, as shown in the following code:

```
Uri serviceUri = ServiceBusEnvironment.CreateServiceUri("sb",
serviceNamespace, "EchoService");
```

In this code sample, the

Microsoft.ServiceBus.ServiceBusEnvironment.CreateServiceUri(System.String,System.String,System.String) method takes the schema (`"sb"` for the Service Bus, used for TCP relay connections), the service namespace, and the name of the endpoint to which to connect. For more information, seeCreating a Service Bus URI.

6. Configure the client endpoint used to connect to the service.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
<system.serviceModel>
<client>
<!-- Application Endpoint -->
<endpoint name="RelayEndpoint"

contract="Microsoft.ServiceBus.Samples.IEchoContract"
                binding="netTcpRelayBinding"/>
</client>


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

In this example, the client endpoint is defined using the name `"RelayEndpoint"`, which is used later to help create the channel factory. The endpoint configuration also declares the contract defined in the first step of this procedure, as well as the fact that the binding used to connect is a **Microsoft.ServiceBus.NetTcpRelayBinding**. For this procedure, the information is declared in an App.config file. You can also define this information programmatically. For more information, seeConfiguring a WCF Service to Register with the Service Bus.

7. Instantiate a channel factory:

```
ChannelFactory<IEchoChannel> channelFactory = new
ChannelFactory<IEchoChannel>("RelayEndpoint", new
EndpointAddress(serviceUri));
```

This channel factory uses the type of channel defined in the client contract at the beginning of this procedure.

8. Apply the credentials and any other behaviors to the endpoint:

```
channelFactory.Endpoint.Behaviors.Add(sharedSecretServiceBusC
redential);
```

Note

The preceding code example assumes that the service endpoint requires issuer name and secret credentials. Service endpoints may or may not require

authentication; if an endpoint does not require authentication, adding the credential behavior to the client endpoint would not be required.

9. Create a new channel to the service and open it:

```
IEchoChannel channel = channelFactory.CreateChannel();
channel.Open();
```

10. Perform whatever tasks are necessary to your scenario:

```
Console.WriteLine("Enter text to echo (or [Enter] to
exit):");
string input = Console.ReadLine();
while (input != String.Empty)
{
     try
    {
        Console.WriteLine("Server echoed: {0}",
channel.Echo(input));
    }
    catch (Exception e)
    {
        Console.WriteLine("Error: " + e.Message);
    }
    input = Console.ReadLine();
}
```

This step consists of accessing the service application through the exposed endpoint. For more information, seeBuilding a Service for the Service Bus.

11. When you are finished, close the channel, as shown in the following code:

```
channel.Close();
channelFactory.Close();
```

Creating a REST-based Client Application for the Service Bus

A client application can access a REST-style service by using the **Microsoft.ServiceBus.WebHttpRelayBinding** or by directly sending HTTP requests to the service endpoint.

Some applications, such as browsers or JavaScript applications, do not use the Microsoft.ServiceBus.dll assembly. Therefore, they do not have access to the Windows Azure SDK. These applications can access a REST service endpoint directly using whatever HTTP

capabilities available to them. In this case, the only issue is what type of security and authentication the service requires. However, in this kind of scenario, it is common for the service to require little or no authentication. For more information, seeHow to: Create a REST-based Service that Accesses the Service Bus. For a full example of this scenario, see the **WebHttp** sample in the Windows Azure SDK, or the Service Bus Message Buffer Tutorial.

If the client has access to the Windows Azure SDK, it follows the same procedure as a standard Service Bus client application. For more information, seeHow to: Create a WCF SOAP Client Application for the Service Bus. The only unique feature in this scenario is that attributes such as `[WebGet]` are applied to the contract you retrieve from the service. These attributes map the contract to the REST standard. Because it is an Internet-based protocol, you will be required to use an HTTP-based binding, such as **Microsoft.ServiceBus.WebHttpRelayBinding**.

Creating a Windows Azure Client for the Service Bus

Although the Windows Azure Service Bus runs on Windows Azure, an application can use either service independently of the other. To emphasize the independence of the technologies, the following is possible:

- An application executing on a computer on a desktop (that is, not hosted in Windows Azure) can use the Service Bus without knowing anything about Windows Azure.
- An application can be hosted in Windows Azure without knowing anything about or using the Service Bus.
- An application can be hosted in Windows Azure and also use the Service Bus.

Therefore, applications that are hosted in Windows Azure and that use the Service Bus (whether as a service or a client) have no different development or configuration requirements other than those required to host any application in Windows Azure with one exception: Because the Microsoft.ServiceBus.dll assembly is not available to Windows Azure, applications hosted in Windows Azure must package and deploy that assembly and make any configuration changes that are required to use it from Windows Azure.

For more information about Service Bus clients that are hosted in Windows Azure, see How to: Configure a Windows Azure-Hosted Service Bus Service or Client Application.

Discovering and Exposing a Service Bus Service

The following topics describe how to use the Windows Azure Service Bus service registry to register service endpoints, expose them in the registry, publish WSDL metadata exchange endpoints, and locate service endpoints that have been registered to be publicly visible.

In This Section

How to: Publish a Service to the Service Bus Registry

This topic describes how to use the **Microsoft.ServiceBus.ServiceRegistrySettings** behavior to publish a service endpoint in the ATOM 1.0 feed and to control the name that appears there.

How to: Discover and Expose a Service Bus Application

This topic describes how to navigate the endpoints published in the registry to obtain the one to which you want to connect.

How to: Expose a Metadata Endpoint

This topic describes how to publish an **System.ServiceModel.Description.IMetadataExchange** service endpoint on the Service Bus to enable SOAP-based clients to create their client channels using tools such as those available in Visual Studio.

How to: Publish a Service to the Service Bus Registry

The **Microsoft.ServiceBus.ServiceRegistrySettings** endpoint behavior gives you control over how a given service is published in the Service Registry. By default, all services are "cloaked" and are not visible in the Service Registry ATOM feed.

The following table lists the properties that you can set on the **Microsoft.ServiceBus.ServiceRegistrySettings** endpoint:

T:Microsoft.ServiceBus.ServiceRegistrySettings **properties**	**Description**
Microsoft.ServiceBus.ServiceRegistrySettings.DisplayName	This is the display name for the endpoint and is used as the `<title>` field for the endpoint in the discovery ATOM feed. By default, the **Microsoft.ServiceBus.ServiceRegistrySettings.DisplayName** property is set to the last segment of the service URI.
Microsoft.ServiceBus.ServiceRegistrySettings.DiscoveryMode	This property is set to the **Microsoft.ServiceBus.DiscoveryType.Public** or **Microsoft.ServiceBus.DiscoveryType.Private** values, with the latter being the default. If you set this property to **Microsoft.ServiceBus.DiscoveryType.Public**, the endpoint is published into the Service Registry ATOM feed.

To add an Application to the Service Bus registry

1. Create an instance of the **Microsoft.ServiceBus.ServiceRegistrySettings** behavior, using the **Microsoft.ServiceBus.DiscoveryType.Public** parameter.

```
ServiceRegistrySettings serviceRegistrySettings = new
ServiceRegistrySettings(DiscoveryType.Public);

serviceRegistrySettings.DisplayName = "MyService";
```

2. Add the description to the associated endpoint.

```
foreach (ServiceEndpoint subscriberEndpoint in
subscriberHost.Description.Endpoints)
{

subscriberEndpoint.Behaviors.Add(serviceRegistrySettings);
}
```

How to: Discover and Expose a Service Bus Application

Once a service has been deployed to a Windows Azure Service Bus endpoint, you can create a client to connect to that service. However, in order to do this, you must first know the URI of the service, which you can discover in one of two ways:

1. The creator of the service can explicitly provide the URI.
2. You can discover the address by navigating the naming hierarchy of the service namespace under which the service has been published.

This second step occurs through the service registry, which is a database of services and their associated URIs. For more information about exposing an endpoint in the service registry, see How to: Publish a Service to the Service Bus Registry.

To discover a service that has been published in the Service Registry

1. You can discover a published service by navigating the naming hierarchy, which can be accessed through a nested tree of Atom 1.0 feeds. The root feed for a given project is located at `http://<service-namespace>.servicebus.windows.net/` .

   Note that, by default, services are "cloaked," and not visible in the Atom feed. A developer must explicitly decide to make the service visible. However, cloaking only makes a service invisible in the Atom feed; any client that has the necessary credentials can still connect to the service if it knows the address.

How to: Expose a Metadata Endpoint

A Windows Azure Service Bus metadata endpoint is a URI that exposes additional information about a service or client application. For example, the Svcutil.exe tool uses the exposed metadata from a service to build a contract so that a developer can access that service. Without the metadata, the developer would have to gain access to the contract in some other way, such as asking the creator for a copy of it directly via e-mail. Note that you can still implement an interface without metadata: metadata just lets you easily obtain the contract if you do not already have it. Also note that exposing a metadata endpoint differs from publishing your interface to the ATOM feed: the metadata endpoint contains additional information about the contract, whereas publishing on the ATOM feed just lists the service URI in a publicly-accessed database.

The following is a simplified procedure for exposing metadata on an application that uses the Service Bus. For a complete discussion of metadata, see **Metadata Architecture Overview** in the Windows Communication Foundation (WCF) documentation.

To expose a metadata endpoint

1. In the App.config file for the host application, add the metadata endpoint definition to the service configuration information.

```
<services>
<service name="Service.EchoService">
<endpoint name="RelayEndpoint"
              ... />


<endpoint name="MexEndpoint"
 contract="IMetadataExchange"
 binding="netTcpRelayBinding"
 bindingConfiguration="default"
 address="mex" />


</service>
</services>
```

2. To add metadata publishing to the service, modify the application configuration information to include an additional behavior section.

```
<system.serviceModel>
  ...
<behaviors>
<endpointBehaviors>
```

```
        ...
    <endpointBehaviors>

    <serviceBehaviors>
    <behavior name="serviceMetadata">
    <serviceMetadata />
    </behavior>
    </serviceBehaviors>

    </behaviors>

    </system.serviceModel>
```

3. Add the metadata behavior to the service by specifying the `behaviorConfiguration` property in the service definition.

```
    <services>
    <service name="Service.EchoService" behaviorConfiguration="serviceMetadata">
        ...
    </service>
    </services>
```

Warning

If the metadata endpoint is specified with a different end-to-end security mode than the service endpoint, and uses a relative address while sharing the same base address with the service endpoint, an exception of type **System.ArgumentException** is thrown when you open the service host. The following error message accompanies the exception: **Incompatible channel listener settings**. To resolve this issue, perform one of the following workarounds:

- Specify the address of the metadata endpoint as a fully-qualified address.
- If you want to use a relative address for the metadata endpoint that shares a base address with the service endpoint, specify the same end-to-end security mode for both the metadata and service endpoints.
- Use a relative address for the metadata endpoint with a base address that differs from the base address of the service endpoint.

Working with a Service Bus Message Buffer

The following topics provide an overview of the Windows Azure Service Bus message buffer feature. They also describe how to create, configure, and use an Service Bus message buffer.

In This Section

Service Bus Message Buffer Overview

Service Bus Message Buffer Overview

Important
The current Message Buffers feature, including their management protocol, will remain supported for backwards compatibility. However, the general recommendation is that you explicitly change client code to use the new Queue feature. For more information, see Queues, Topics, and Subscriptions.

Message buffers are small, temporary cache locations in which messages can be held for a short time until they are retrieved. Message buffers are especially useful in Web programming model scenarios when Windows Azure Service Bus bindings are not available; for example, when the message consumer is running on a computer that is not running Windows, or is implemented in Java. Message buffers can be accessed by applications that use HTTP and do not require the Windows Azure SDK. Hence, message buffers enable Web developers, mobile device programmers, and others to integrate their applications together with the Service Bus by creating message consumers that use HTTP requests to poll for messages.

Message buffers use the HTTP REST protocol to expose various operations on the message buffer such as creating a message buffer, sending a message to the message buffer, and retrieving a message from the message buffer. These operations are described in the following Message Buffer Protocol section.

The following code example shows how to use the REST protocol to create a message buffer, send and retrieve a message from the message buffer, and finally delete the message buffer. This example uses **System.Net.WebClient** to send and receive HTTP requests. For a complete working sample, see the **PlainHttp** sample in the Windows Azure SDK samples folder under ServiceBus\ExploringFeatures\MessageBuffer.

```
// Prompt the user for the service namespace and issuer key.

Console.Write("Please enter your Service Namespace: ");
```

```
string serviceNamespace = Console.ReadLine();
Console.Write("Please enter the key for the 'owner' issuer: ");
string ownerKey = Console.ReadLine();
// Create a GUID for the buffer name.
string bufferName = Guid.NewGuid().ToString("N");

// Construct the message buffer URI.
string messageBufferLocation = string.Format("http://{0}.servicebus.windows.net/{1}",
serviceNamespace, bufferName);

// Get the AC token
WebClient client = new WebClient();
client.BaseAddress = string.Format("https://{0}-sb.accesscontrol.windows.net/",
serviceNamespace);
NameValueCollection values = new NameValueCollection();
values.Add("wrap_name", "owner");
values.Add("wrap_password", ownerKey);
values.Add("wrap_scope", messageBufferLocation);
byte[] responseBytes = client.UploadValues("WRAPv0.9", "POST", values);
string response = Encoding.UTF8.GetString(responseBytes);

string token = Uri.UnescapeDataString(response.Split('&').Single(value =>
value.StartsWith("wrap_access_token=",
StringComparison.OrdinalIgnoreCase)).Split('=')[1]);

// Create the auth header from the token
string authHeaderValue = string.Format("WRAP access_token=\"{0}\"", token);

// Create the message buffer policy.
string policy =
    "<entry xmlns=\"http://www.w3.org/2005/Atom\">" +
    "<content type=\"text/xml\">" +
    "<MessageBufferPolicy
xmlns=\"http://schemas.microsoft.com/netservices/2009/05/servicebus/connect\"/>" +
    "</content>" +
```

```
    "</entry>";

// Create a message buffer.
client.BaseAddress = string.Format("https://{0}.servicebus.windows.net/{1}/",
serviceNamespace, bufferName);
client.Headers[HttpRequestHeader.ContentType] =
"application/atom+xml;type=entry;charset=utf-8";
client.Headers[HttpRequestHeader.Authorization] = authHeaderValue;
client.UploadData(String.Empty, "PUT", Encoding.UTF8.GetBytes(policy));
Console.WriteLine("Message buffer was created at '{0}'.", messageBufferLocation);

// Send a message to the message buffer.
client.Headers[HttpRequestHeader.ContentType] = "text/xml";
client.Headers[HttpRequestHeader.Authorization] = authHeaderValue;
client.UploadData("messages?timeout=20", "POST", Encoding.UTF8.GetBytes("<msg1>This is
message #1</msg1>"));
Console.WriteLine("Message sent.");

// Retrieve message.
client.Headers[HttpRequestHeader.Authorization] = authHeaderValue;
string payload = Encoding.UTF8.GetString(client.UploadData("messages/head?timeout=20",
"DELETE", new byte[0]));
Console.WriteLine("Retrieved the message '{0}'.", payload);

// Delete the message buffer.
client.Headers[HttpRequestHeader.Authorization] = authHeaderValue;
client.UploadData(String.Empty, "DELETE", new byte[0]);
Console.WriteLine("Message buffer at '{0}' was deleted.", messageBufferLocation);
```

You can also use message buffers through the APIs provided by the Windows Azure SDK. This requires the Windows Azure SDK to be installed. For more information about using message buffers with the Windows Azure SDK, see the Using the Message Buffer with the Windows Azure SDK section later in this topic.

Message Buffer Protocol

The message buffer protocol is an HTTP REST protocol that is designed to follow REST principles and to be simple and easy to understand. The goal is to make sure that developers can easily use the protocol from any client without requiring a library or SDK.

The protocol relies on the Access Control service HTTP authorization model to help it enforce access control on the message buffer. This means that it uses the Simple Web Token (SWT) mechanism that you can use to retrieve a token using HTTP, and then embed it in an HTTP request as a header. This token includes claims that are used to determine whether an operation should be allowed.

The protocol expects each message buffer to be located at a unique URI in the Service Bus namespace. This URI then becomes the root for a set of resources that represents the message buffer *instance*. Each resource type has a unique URI and an associated set of HTTP verbs for interacting with it. The URIs are organized in a way that helps communicate the logical relationships between the different types of resources.

The verbs used to interact with message buffers are modeled on standard HTTP commands. The following is a list of these verbs:

- POST/PUT: Use to create new resources. POST is used to create a new instance of the message resource. PUT is used to either create or update a message buffer resource. When using POST, the address of the new resource is returned in the response.
- PUT: Use to update an existing resource.
- DELETE: Use to delete an existing resource.
- GET: Use to retrieve a representation of a resource. In this case the GET is expected to represent a snapshot of the resource and it can be cached when appropriate.

The following is a summary of the different message buffer resources and the associated verbs for each resource.

URI	Resource	Operations	
http://{serviceNamespace}.servicebus.windows.net/{path}/{buffer}	Message buffer	PUT	Creates or updates message buffer.
		GET	Gets message buffer policy.
		DELETE	Deletes message buffer along with its policy and all associated state.
http://{serviceNamespace}.servicebus.windows.net/{path}/{buffer}/messages	Message buffer store	POST	Creates message. (Returns message URI.)
http://{serviceNamespace}.servicebus.windows.net/{path}/{buffer}/m	First unlocked message	POST	Gets the first unlocked message and locks it.

URI	Resource	Operations	
essages/head			(Returns message content, message URI, lock duration, lock URI, and lock ID.)
		DELETE	Retrieves the first unlocked message and deletes it from the buffer. (Returns message content.)
http://{serviceNamespace}.servicebus.windows.net/{path}/{buffer}/messages/{messageid}	Message	DELETE	Deletes message. Supports delete with lock ID.
http://{serviceNamespace}.servicebus.windows.net/{path}/{buffer}/messages/{messageid}/{lockid}	Message lock	DELETE	Unlocks message.

Request/Response Details of the Message Buffer Protocol

The following tables list the contents of the requests and their corresponding responses for each message buffer operation.

Create or update a message buffer

Resource URI	https://{serviceNamespace}.servicebus.windows.net/{path}/{buffer}
HTTP Verb	PUT
Request Headers	Authorization: WRAPv0.8 {token} Content-Type: application/atom+xml;type=entry;charset=utf-8
Request Body	{policy-xml} For example: <entry xmlns="http://www.w3.org/2005/Atom"> <content type="text/xml"> <MessageBufferPolicy xmlns="http://schemas.microsoft.com/netservices/2009/05/servicebus/connect"> <Authorization>[AuthorizationPolicy enum value]</Authorization> <Discoverability>[DiscoverabilityPolicy enum value]</Discoverability> <TransportProtection>[TransportProtectionPolicy enum value]</TransportProtection> <ExpiresAfter>PTnHnMnS</ExpiresAfter>

	<MaxMessageCount>nnn</MaxMessageCount> <OverflowPolicy>[OverflowPolicy enum value]</OverflowPolicy> </MessageBufferPolicy> </content> </entry> Notes: The ExpiresAfter time interval is broken up into hours (H), minutes (M), and seconds (S). Replace n by the corresponding amount of time for each unit of time. The enumeration values for the policy properties are part of the Microsoft.ServiceBus namespace.
Response Body	{policy-xml} The message buffer policy is returned because some policy property might be defaulted if their value supplied in the request is invalid.
Expected HTTP Status Code	201 Created: The message buffer was successfully created. 403 Forbidden: Quota exceeded on the number of message buffers. Please increase the per solution connection quota.

Get a message buffer policy

Resource URI	https://{serviceNamespace}.servicebus.windows.net/{path}/{buffer}
HTTP Verb	GET
Request Headers	Authorization: WRAPv0.8 {token}
Request Body	Empty.
Response Body	{policy-xml}
Expected HTTP Status Code	200 OK: The message buffer policy was successfully retrieved.

Delete a message buffer

Resource URI	https://{serviceNamespace}.servicebus.windows.net/{path}/{buffer}
HTTP Verb	DELETE
Request Headers	Authorization: WRAPv0.8 {token}

Request Body	Empty.
Response Body	Empty.
Expected HTTP Status Code	200 OK: The message buffer was successfully deleted.

Send a message to a message buffer

Resource URI	https://{serviceNamespace}.servicebus.windows.net/{path}/{buffer}/messages
HTTP Verb	POST
Request Headers	Authorization: WRAPv0.8 {token} Content-Type: application/atom+xml;type=entry;charset=utf-8
Request Body	{message-payload}
Response Body	Empty.
Expected HTTP Status Code	201 Created: The message was sent successfully. 408 Request Timeout: The request timed out. Please retry. 400 Bad Request: The request exceeded the quota of pending senders to the buffer. Please retry after some time. 410 Gone: The message buffer does not exist anymore. Please recreate the message buffer and retry.

Retrieve a message from the message buffer (destructive read)

Resource URI	https://{serviceNamespace}.servicebus.windows.net/{path}/{buffer}/messages/head?timeout={timeout-in-seconds} The timeout parameter specifies the maximum amount of time that the retrieve request will wait if a message is not readily available in the message buffer. The maximum time that can be specified is 2 minutes.
HTTP Verb	DELETE
Request Headers	Authorization: WRAPv0.8 {token}
Request Body	Empty.
Respon	{message-payload}

se Body	
Expected HTTP Status Code	200 OK: The message was successfully retrieved. 204 No Content: No messages are present in the message buffer. 400 Bad Request: This request exceeds the quota of pending receivers to the buffer. Please retry after some time. 410 Gone: The message buffer is no longer available.

Lock a message in the message buffer (non-destructive read)

Resource URI	https://{serviceNamespace}.servicebus.windows.net/{path}/{buffer}/messages/head?timeout={timeout-in-seconds}&lockduration={lockduration-in-secs} The timeout parameter specifies the maximum amount of time that this request will wait if a message is not readily available in the message buffer. The maximum time that can be specified is 2 minutes. The lockduration parameter specifies the time in seconds that the returned message is locked so that no other consumer can see the message. The maximum lock duration is 5 minutes and the minimum lock duration is 10 seconds.
HTTP Verb	POST
Request Headers	Authorization: WRAPv0.8 {token}
Request Body	Empty.
Response Headers	X-MS-MESSAGE-LOCATION: https://{serviceNamespace}.servicebus.windows.net/{path}/{buffer}/messages/{message-id} This header indicates the location of the message. This URI is needed to unlock or delete the message. For more information, see the “Unlock a locked message” and “Delete a locked message from the message buffer” tables. X-MS-LOCK-ID: {lock-id} This header provides the lock ID of the message. This lock ID is needed to delete the locked message. For more information, see the “Delete a locked message from the message buffer” table. X-MS-LOCK-LOCATION: https://{serviceNamespace}.servicebus.windows.net/{path}/{buffer}/messages/{message-id}/{lock-id} This header indicates the lock location for the message. This URI is needed to unlock the message. For more information, see the “Unlock a locked message” table.

Response Body	{message-payload}
Expected HTTP Status Code	200 OK: The message was successfully read and locked. 204 No Content: No messages are present in the message buffer. 400 Bad Request: This request exceeds the quota of pending receivers to the buffer. Please retry after some time. 410 Gone: The message buffer is no longer available.

Unlock a locked message (that is, delete a lock on a message) in the message buffer

Resource URI	https://{serviceNamespace}.servicebus.windows.net/{path}/{buffer}/messages/{message-id}/{lock-id}
HTTP Verb	DELETE
Request Headers	Authorization: WRAPv0.8 {token}
Request Body	Empty.
Response Body	Empty.
Expected HTTP Status Code	200 OK: The message was successfully unlocked. 404 Not Found: No message with the specified lock ID was found. Please retry with a valid lock ID. 410 Gone: The message buffer is no longer available.

Delete a locked message from the message buffer

Resource URI	https://{serviceNamespace}.servicebus.windows.net/{path}/{buffer}/messages/{message-id}?lockid={lock-id} The lockid parameter should be a valid lock ID from the response header value “X-MS-LOCK-ID” that is returned in the response of a non-destructive read operation. For more information, see the “Lock a message in the message buffer” table.
HTTP Verb	DELETE
Request Headers	Authorization: WRAPv0.8 {token}

Request fBody	Empty.
Response Body	Empty.
Expected HTTP Status Code	200 OK: The message was successfully deleted. 404 Not Found: No message with the specified lock ID was found. Please retry with a valid lock ID. 410 Gone: The message buffer is no longer available.

Message Buffer Policy

The message buffer policy is an XML document that defines the desired semantics for a message buffer. You must include a message buffer policy document in the request when creating a new message buffer instance. You can later retrieve the message buffer policy document to determine the semantics for an existing message buffer. You can also update an existing message buffer policy document to renew the expiration time-out. When doing this, you must ensure that the other properties are unchanged.

For more information about the message buffer policy properties that can be configured, see the documentation for the **Microsoft.ServiceBus.MessageBufferPolicy** class.

Message Buffer Quota

By default, the message buffer can contain up to 10 messages. You can modify this limit through the **Microsoft.ServiceBus.MessageBufferPolicy.MaxMessageCount** property in the **Microsoft.ServiceBus.MessageBufferPolicy** class. The maximum number of messages that the message buffer can hold is 50 messages.

Using the Message Buffer with the Windows Azure SDK

The Windows Azure SDK provides a set of managed client APIs that make it easy for developers to use a message buffer. They are designed to closely follow the semantics of the protocol and enable the use and configuration of message buffers. These APIs are exposed through the **Microsoft.ServiceBus.MessageBufferClient** and the **Microsoft.ServiceBus.MessageBufferPolicy** classes.

You can use the **Microsoft.ServiceBus.MessageBufferClient** class to create a new message buffer or to retrieve an object that you can then use to interact with an existing message buffer. It provides methods for operating directly on the message buffer such as queuing messages in the message buffer and retrieving messages from the message buffer. For more information, see the documentation for the **Microsoft.ServiceBus.MessageBufferClient** class.

You can use the **Microsoft.ServiceBus.MessageBufferPolicy** to configure the message buffer including the security used on the message buffer, the message buffer lifespan, and the message buffer capacity. For more information, see the documentation for the **Microsoft.ServiceBus.MessageBufferPolicy** class.

The following code example shows how to configure and create a message buffer, and send and retrieve messages from the message buffer.

```
string serviceNamespace = "...";

MessageVersion messageVersion = MessageVersion.Soap12WSAddressing10;

string messageAction = "urn:Message";


// Configure credentials.

TransportClientEndpointBehavior behavior = new TransportClientEndpointBehavior();

behavior.CredentialType = TransportClientCredentialType.SharedSecret;

behavior.Credentials.SharedSecret.IssuerName = "...";

behavior.Credentials.SharedSecret.IssuerSecret = "...";


// Configure buffer policy.

MessageBufferPolicy policy = new MessageBufferPolicy

{

    ExpiresAfter = TimeSpan.FromMinutes(2.0d),

    MaxMessageCount = 100

};


// Create message buffer.

string bufferName = "MyBuffer";

Uri bufferLocation = new Uri("https://" + serviceNamespace +

".servicebus.windows.net/services/" + bufferName);

MessageBufferClient client = MessageBufferClient.CreateMessageBuffer(behavior,

bufferLocation, policy, messageVersion);


// Send 10 messages.

for (int i = 0; i < 10; ++i)

{

    client.Send(Message.CreateMessage(messageVersion, messageAction, "Message #" + i));

}


Message message;

string content;
```

```
// Retrieve a message (destructive read).
message = client.Retrieve();
content = message.GetBody<string>();
message.Close();

Console.WriteLine("Retrieve message content: {0}", content);

// Retrieve a message (peek/lock).
message = client.PeekLock();
content = message.GetBody<string>();

Console.WriteLine("PeekLock message content: {0}", content);

// Delete previously locked message.
client.DeleteLockedMessage(message);
message.Close();

// If no more messages are retrieved within the ExpiresAfter time span,
// the buffer will automatically be deleted...
```

Important

Because message buffer contents are stored in active memory, there are no strong fault tolerance or reliability guarantees. If the server hosting a message buffer crashes, you may lose messages. In the event of a server crash, you do not necessarily lose the buffer itself: knowledge of the buffer, including policy settings, is distributed across multiple servers and can be recovered. However, any messages in your buffer at the time of the crash are lost. Therefore, if you are designing an application that requires a high degree of message reliability, we recommend that you provide for message redundancy and recovery through another way.

How to: Configure a Service Bus Message Buffer

Before you create a message buffer, you must determine the configuration that you want to apply to the buffer. A message buffer is configured by using the **Microsoft.ServiceBus.MessageBufferPolicy** class, which exposes several properties that control the message buffer behavior.

MessageBufferPolicy Properties	Description
Microsoft.ServiceBus.MessageBufferPolicy.Authorization	Specifies the authorization policy for managing the message buffer and sending or receiving messages. The default is **Microsoft.ServiceBus.AuthorizationPolicy.Required**, which means that authorization is required.
Microsoft.ServiceBus.MessageBufferPolicy.Discoverability	Controls whether the message buffer is visible on the ATOM feed. If the message buffer is not visible, only clients that know the exact URI can access it. The default is **Microsoft.ServiceBus.DiscoverabilityPolicy.Managers**, which means that the applications that created the buffer with manager permissions can see the URI.
Microsoft.ServiceBus.MessageBufferPolicy.ExpiresAfter	Specifies the time interval during which the message buffer idles before automatically deleting itself. The lifespan is implicitly renewed every time that an application sends a message to the buffer or requests a message from the buffer. The default is 5 minutes and the maximum is 10 minutes.
Microsoft.ServiceBus.MessageBufferPolicy.MaxMessageCount	Specifies the maximum number of messages that can be in the message buffer before the overflow policy becomes active. The default is 10 messages and the maximum is 50 messages.
Microsoft.ServiceBus.MessageBufferPolicy.OverflowPolicy	Determines the action to perform on incoming messages when the message buffer capacity is reached, as defined by the **Microsoft.ServiceBus.MessageBufferPolicy.MaxMessageCount** property. Currently, the only available action is to reject the incoming message by returning the message to the sender.
Microsoft.ServiceBus.MessageBufferPolicy.TransportProtection	Specifies the level of end-to-end security that will be used. End-to-end security refers to the security used between the sender, the message buffer, and the receiver. The default is

	Microsoft.ServiceBus.TransportProtectionPolicy.AllPaths, which means that messages must be sent and received from the message buffer using a secure communication channel.

You apply a policy to the message buffer when you create the message buffer.

A message buffer can be thought of as a service hosted on a Windows Azure Service Bus endpoint. Similar to other services hosted on Service Bus endpoints, you must create a message buffer with credentials that enable manage-level operations upon the target service namespace. Therefore, to create and configure an Service Bus message buffer, you must first obtain a valid token from the Access Control service.

To configure a Service Bus message buffer using the REST protocol

1. Before creating the message buffer policy, you must obtain the Access Control token that is used to authenticate your application with the Service Bus. The following example obtains a token from the Access Control service for the specified service namespace, then creates the authorization header that is sent later in the HTTP request.

```
string serviceNamespace = "...";
string ownerKey = "...";
string bufferName = "...";

// Construct the message buffer URI.
string messageBufferLocation =
string.Format("http://{0}.servicebus.windows.net/{1}",
serviceNamespace, bufferName);

// Get the AC token.
WebClient client = new WebClient();
client.BaseAddress = string.Format("https://{0}-
sb.accesscontrol.windows.net/", serviceNamespace);
NameValueCollection values = new NameValueCollection();
values.Add("wrap_name", "owner");
values.Add("wrap_password", ownerKey);
values.Add("wrap_scope", messageBufferLocation);
byte[] responseBytes = client.UploadValues("WRAPv0.9",
"POST", values);
string response = Encoding.UTF8.GetString(responseBytes);
```

```
string token =
Uri.UnescapeDataString(response.Split('&').Single(value =>
value.StartsWith("wrap_access_token=",
StringComparison.OrdinalIgnoreCase)).Split('=')[1]);


// Create the auth header from the token.
string authHeaderValue = string.Format("WRAP
access_token=\"{0}\"", token);
```

To build and run the previous code, add the following `using` statements at the top of the class file:

```
using System.Net;
using System.Collections.Specialized;
```

2. Create the message buffer policy XML string that will be used when you create the message buffer. The following message buffer policy configures the message buffer with an expiration time of 2 minutes and a maximum capacity of 20 messages.

```
// Create the message buffer policy.
string policy =
    "<entry xmlns=\"http://www.w3.org/2005/Atom\">" +
    "<content type=\"text/xml\">" +
    "<MessageBufferPolicy
xmlns=\"http://schemas.microsoft.com/netservices/2009/05/serv
icebus/connect\">" +
    "<ExpiresAfter>PT0H2M0S</ExpiresAfter>" +
    "<MaxMessageCount>20</MaxMessageCount>" +
    "</MessageBufferPolicy>" +
    "</content>" +
    "</entry>";
```

3. After you have finished creating your message buffer policy, you can apply the configuration information at the time that you create the message buffer and start to send and receive messages. For more information, seeHow to: Create and Connect to a Service Bus Message Buffer.

How to configure a Service Bus message buffer using the Windows Azure SDK

1. Configure your credentials as you would any other application that uses the Service Bus.

```
TransportClientEndpointBehavior behavior = new
TransportClientEndpointBehavior();
behavior.CredentialType =
TransportClientCredentialType.SharedSecret;
behavior.Credentials.SharedSecret.IssuerName = "...";
behavior.Credentials.SharedSecret.IssuerSecret = "...";
```

   To build and run the previous code, you must add a reference to the Microsoft.ServiceBus.dll assembly. Also, add the following `using` statement at the top of the class file to reference this namespace, as follows:

```
using Microsoft.ServiceBus;
```

2. Configure the message buffer policy using the **Microsoft.ServiceBus.MessageBufferPolicy** class. The following message buffer policy configures the message buffer with a maximum capacity of 20 messages and an expiration time of 2 minutes.

```
MessageBufferPolicy policy = new MessageBufferPolicy();
policy.MaxMessageCount = 20;
policy.ExpiresAfter = TimeSpan.FromMinutes(2);
```

3. After you have finished configuring your message buffer policy, you can apply the configuration at the time that you instantiate the message buffer and start to send and receive messages. For more information, see How to: Create and Connect to a Service Bus Message Buffer. You can also use the **Microsoft.ServiceBus.MessageBufferClient.GetPolicy** method to retrieve the message buffer policy at any time in order to discover the current configuration settings.

How to: Create and Connect to a Service Bus Message Buffer

After you have specified the desired configuration of your message buffer by creating a message buffer policy (see How to: Configure a Service Bus Message Buffer), you must define the Windows Azure Service Bus endpoint URI for the message buffer, and then instruct the Service Bus to instantiate and expose a message buffer at that location.

The following table lists the operations available to create, access, and delete a message buffer using the REST protocol.

Resource URI	HTTP Verb	Description
https://{serviceNamespace}.servicebus.	PUT	Creates a message buffer on the

windows.net/{path}/{buffer}		Service Bus with the specified policy.
	GET	Retrieves the message buffer policy.
	DELETE	Deletes the message buffer along with its policy and all associated state.

The following table lists the methods available to create, access, and delete a message buffer using the Windows Azure SDK.

Method	Description
Microsoft.ServiceBus.MessageBufferClient.CreateMessageBuffer(Microsoft.ServiceBus.TransportClientEndpointBehavior,System.Uri,Microsoft.ServiceBus.MessageBufferPolicy)	Creates a message buffer on the Service Bus with the specified policy and location.
Microsoft.ServiceBus.MessageBufferClient.GetMessageBuffer(Microsoft.ServiceBus.TransportClientEndpointBehavior,System.Uri,System.ServiceModel.Channels.MessageVersion)	Connects to a pre-existing message buffer.
Microsoft.ServiceBus.MessageBufferClient.DeleteMessageBuffer	Deletes the message buffer.

Note

You can create only one message buffer at a given Service Bus endpoint. Furthermore, when a message buffer is open on a given endpoint, no other services can be open on that same endpoint.

To create and delete a Service Bus message buffer using the REST protocol

1. Send a PUT HTTP request to create a message buffer using the authorization header and message buffer policy previously created in How to: Configure a Service Bus Message Buffer.

```
WebClient client = new WebClient();
client.BaseAddress =
string.Format("https://{0}.servicebus.windows.net/{1}/",
```

```
serviceNamespace, bufferName);
client.Headers[HttpRequestHeader.ContentType] =
"application/atom+xml;type=entry;charset=utf-8";
client.Headers[HttpRequestHeader.Authorization] =
authHeaderValue;
client.UploadData(String.Empty, "PUT",
Encoding.UTF8.GetBytes(policy));
```

2. At this point, you can send and receive messages from the message buffer. For more information, see How to: Send Messages to a Service Bus Message Buffer and How to: Retrieve a Message from a Service Bus Message Buffer.
3. To maintain the existence of the message buffer, periodically request a message from the buffer.

   A message buffer renews its own lifespan implicitly whenever a request for a message is received. This is the only method currently supported for renewing the lifespan of a message buffer.
4. When finished, you can delete the message buffer by sending the following DELETE HTTP request.

```
client.Headers[HttpRequestHeader.Authorization] =
authHeaderValue;
client.UploadData(String.Empty, "DELETE", new byte[0]);
```

   You do not have to delete the message buffer when you are finished: all message buffers have an expiration time that causes the Service Bus to automatically delete the buffer. However, you may want to delete the buffer before this expiration time. For more information, see the **Microsoft.ServiceBus.MessageBufferPolicy.ExpiresAfter** property.

To create and delete a Service Bus message buffer using the Windows Azure SDK

1. Create a URI that contains the address of the buffer.

```
Uri bufferLocation = new Uri("https://" + serviceNamespace +
".servicebus.windows.net/services/MyBuffer");
```

2. Create the buffer by calling the **Microsoft.ServiceBus.MessageBufferClient.CreateMessageBuffer(Microsoft.ServiceBus.TransportClientEndpointBehavior,System.Uri,Microsoft.ServiceBus.MessageBufferPolicy)** method.

```
MessageBufferClient client =
MessageBufferClient.CreateMessageBuffer(behavior,
bufferLocation, policy);
```

You can also obtain access to a pre-existing message buffer by calling the **Microsoft.ServiceBus.MessageBufferClient.GetMessageBuffer(Microsoft.ServiceBus.TransportClientEndpointBehavior,System.Uri)** method.

3. At this point, you can send and receive messages from the message buffer. For more information, see How to: Send Messages to a Service Bus Message Buffer and How to: Retrieve a Message from a Service Bus Message Buffer.
4. To maintain the existence of the message buffer, periodically request a message from the buffer.

   A message buffer renews its own lifespan implicitly whenever a request for a message is received. This is the only method currently supported for renewing the lifespan of a message buffer.
5. When finished, you can close the message buffer by calling the **Microsoft.ServiceBus.MessageBufferClient.DeleteMessageBuffer** method.

   You do not have to delete the message buffer when you are finished: all buffers have an expiration time that causes the Service Bus to automatically delete the buffer. However, you may want to delete the buffer before this expiration time. For more information, see the **Microsoft.ServiceBus.MessageBufferPolicy.ExpiresAfter** property.

How to: Send Messages to a Service Bus Message Buffer

After you have created a Windows Azure Service Bus message buffer, you can send messages to it.

The following describes the operations you can perform to send messages to a message buffer using the REST protocol.

Resource URI	HTTP Verb	Description
https://{serviceNamespace}.servicebus.windows.net/{path}/{buffer}/messages	POST	Attempts to send the message specified in the request body.
https://{serviceNamespace}.servicebus.windows.net/{path}/{buffer}/messages?timeout=n	POST	Attempts to send the message specified in the request body for a time interval equal to the specified timeout value.

The following is a description of the methods that you can use to send a message to a message buffer using the API in the Windows Azure SDK.

Method	Description
Microsoft.ServiceBus.MessageBufferClient.Send(System.ServiceModel.Channels.Message)	Attempts to send the specified message.

Microsoft.ServiceBus.MessageBufferClient.Send(System.ServiceModel.Channels.Message,System.TimeSpan)	Attempts to send the specified message for a time interval equal to the specified timeout value.

To send a message to a message buffer using the REST protocol

1. Send a message to the message buffer using an HTTP POST request with the message content in its body. The address specified in the `UploadData` call is relative to the base address URI of the Web client. To create the Web client and set its base address, see How to: Create and Connect to a Service Bus Message Buffer. To construct the authorization header, see How to: Configure a Service Bus Message Buffer.

```
// Add request headers.
client.Headers[HttpRequestHeader.ContentType] = "text/xml";
client.Headers[HttpRequestHeader.Authorization] =
authHeaderValue;
// Send the POST HTTP request with the message as the request
body.
client.UploadData("messages", "POST",
Encoding.UTF8.GetBytes("<msg1>This is message #1</msg1>"));
```

2. Alternatively, by appending a time-out parameter to the URI, you can specify the time interval during which the attempt to send the message is made.

```
client.UploadData("messages?timeout=20", "POST",
Encoding.UTF8.GetBytes("<msg1>This is message #1</msg1>"));
```

To send a message to a message buffer using the Windows Azure SDK

1. Send a message to the message buffer with a call to the **Microsoft.ServiceBus.MessageBufferClient.Send(System.ServiceModel.Channels.Message)** or **Microsoft.ServiceBus.MessageBufferClient.Send(System.ServiceModel.Channels.Message,System.TimeSpan)** methods.

```
client.Send(Message.CreateMessage(messageVersion,
messageAction, "Message #1"));
client.Send(Message.CreateMessage(messageVersion,
messageAction, "Message #2"), TimeSpan.FromSeconds(30));
```

The first method sends the message to the message buffer. The second method lets you specify a time-out period. If the message was not successfully sent during that time, the call will generate a **TimeoutException**.

The following example, taken from the code example in Service Bus Message Buffer Overview, shows how to send several messages to a message buffer.

```
MessageBufferClient client =
MessageBufferClient.CreateMessageBuffer(behavior,
bufferLocation, policy, messageVersion);

// Send 10 messages.
for (int i = 0; i < 10; ++i)
{
    client.Send(Message.CreateMessage(messageVersion,
messageAction, "Message #" + i));
}
```

To build and run the previous code, you must add references to the Microsoft.ServiceBus.dll and System.ServiceModel.dll assemblies. Also, add the following `using` statements at the top of the class file to reference these namespaces:

```
using Microsoft.ServiceBus;
using System.ServiceModel.Channels;
```

How to: Retrieve a Message from a Service Bus Message Buffer

There are two ways to retrieve a single message from a Windows Azure Service Bus message buffer: a peek/lock read or a destructive read.

The message buffer orders messages using the principle of first-in-first-out; therefore, a peek/lock read retrieves the next available message from the start of the buffer. However, the message is not deleted from the buffer until a destructive read is performed by the client. Locked messages are not available for additional retrieval unless they are deleted, unlocked, or their lock duration expires. The next non-locked message in the message buffer can be retrieved.

Therefore, a peek/lock read allows the client to see the next message in the buffer, while at the same time locking that message in place until you delete it, unlock it, or until its lock duration expires. In contrast, a destructive read retrieves the message, and then deletes the message from the buffer.

The following describes the operations you can perform for destructive and non-destructive retrieval of a message from a message buffer using the REST protocol.

Resource URI	HTTP Verb	Description

https://{serviceNamespace}.servicebus.windows.net/{path}/{buffer}/messages/head?timeout={timeout-in-seconds}	DELETE	Retrieves the first message and deletes it from the message buffer. The optional time-out parameter specifies the length of time for the operation to finish.
https://{serviceNamespace}.servicebus.windows.net/{path}/{buffer}/messages/head?timeout={timeout-in-seconds}&lockduration={lockduration-in-secs}	POST	Retrieves the first message from the message buffer and locks the message until the specified lock duration expires or until instructed by the caller to either delete it or release the lock. If there are no messages in the buffer, this method waits up to the specified time-out interval for a message to arrive. The `timeout` and `lockduration` parameters are optional. If not specified, the default time-out and lock duration values are used.
https://{serviceNamespace}.servicebus.windows.net/{path}/{buffer}/messages/{message-id}/{lock-id}	DELETE	Releases the lock on the specified message in the message buffer.
https://{serviceNamespace}.servicebus.windows.net/{path}/{buffer}/messages/{message-id}?lockid={lock-id}	DELETE	Deletes the specified locked message.

The following is a description of the methods you can use for destructive and non-destructive retrieval of a message from a message buffer using the API in the Windows Azure SDK.

Method	Description
Microsoft.ServiceBus.MessageBufferClient.Retrieve	Retrieves the first message and deletes it from the message buffer.
Microsoft.ServiceBus.MessageBufferClient.Retrieve(System.TimeSpan)	Retrieves the first message and deletes it from the message buffer, using the specified time-out.
Microsoft.ServiceBus.MessageBufferClient.PeekLock	Retrieves the first message from the message buffer and locks the message until instructed by the caller to either delete the message or release the lock, or until the default lock duration expires.
Microsoft.ServiceBus.MessageBufferClient.PeekLock(System.TimeSpan)	Retrieves the first message from the message buffer and locks the message until instructed by the caller to either delete it or release the lock, or until the default lock duration expires. If there are no messages in the buffer, this method will wait up to

	the specified time-out interval for a message to arrive.
Microsoft.ServiceBus.MessageBufferClient.PeekLock(System.TimeSpan,System.TimeSpan)	Retrieves the first message from the message buffer and locks the message until the specified lock duration expires or until instructed by the caller to either delete it or release the lock. If there are no messages in the buffer, this method will wait up to the specified time-out interval for a message to arrive.
Microsoft.ServiceBus.MessageBufferClient.ReleaseLock(System.ServiceModel.Channels.Message)	Releases the lock on the specified message in the message buffer.
Microsoft.ServiceBus.MessageBufferClient.ReleaseLock(System.Uri)	Releases the lock on the message that is contained at the specified URI.
Microsoft.ServiceBus.MessageBufferClient.DeleteLockedMessage(System.ServiceModel.Channels.Message)	Deletes the specified locked message.

To perform a peek/lock read using the REST protocol

1. Retrieve the first message while locking it in the message buffer. The following example uses a lock duration of 120 seconds and a time-out value of 20 seconds in the peek/lock request. The `PeekLockMessage` method expects the Web client and authorization header to be passed in. To create the Web client and set its base address, see How to: Create and Connect to a Service Bus Message Buffer. To construct the authorization header, see How to: Configure a Service Bus Message Buffer.

```
static string PeekLockMessage(WebClient webClient, string
authHeaderValue, out string messageLocation, out string
lockId, out string lockLocation)
{
    webClient.Headers[HttpRequestHeader.Authorization] =
authHeaderValue;
    byte[] response =
webClient.UploadData("messages/head?timeout=20&lockduration=1
20", "POST", new byte[0]);
    messageLocation = webClient.ResponseHeaders["X-MS-
MESSAGE-LOCATION"];
    lockId = webClient.ResponseHeaders["X-MS-LOCK-ID"];
    lockLocation = webClient.ResponseHeaders["X-MS-LOCK-
```

```
LOCATION"];

    return Encoding.UTF8.GetString(response);
}
```

2. After you have performed a peek/lock operation on the message, you can delete the locked message from the message buffer using the message location and lock ID returned by the peek/lock operation.

```
static void DeleteLockedMessage(string authHeaderValue,
string messageLocation, string lockId)
{
    WebClient webClient = new WebClient();
    webClient.BaseAddress = string.Format("{0}?lockid={1}",
messageLocation, lockId);
    webClient.Headers[HttpRequestHeader.ContentType] =
"application/atom+xml;type=entry;charset=utf-8";
    webClient.Headers[HttpRequestHeader.Authorization] =
authHeaderValue;
    webClient.UploadData(string.Empty, "DELETE", new
byte[0]);
}
```

3. Alternatively, if you do not want to delete the message, you can release the lock on the locked message using the lock location returned by the peek/lock operation.

```
static void UnlockMessage(string authHeaderValue, string
lockLocation)
{
    WebClient webClient = new WebClient();
    webClient.BaseAddress = lockLocation;
    webClient.Headers[HttpRequestHeader.ContentType] =
"application/atom+xml;type=entry;charset=utf-8";
    webClient.Headers[HttpRequestHeader.Authorization] =
authHeaderValue;
    webClient.UploadData(string.Empty, "DELETE", new
byte[0]);
}
```

To perform a destructive read using the REST protocol

1. Retrieve the first message and delete it from the message buffer. The following example

performs a retrieval, with a time-out value of 20 seconds.

```
webClient.Headers[HttpRequestHeader.Authorization] =
authHeaderValue;
string payload =
Encoding.UTF8.GetString(webClient.UploadData("messages/head?t
imeout=20", "DELETE", new byte[0]));
```

To perform a Peek/Lock read using the Windows Azure SDK

1. Retrieve a message while locking it in the message buffer with a call to the **Microsoft.ServiceBus.MessageBufferClient.PeekLock** method.

   ```
   Message message = client.PeekLock();
   ```

2. After you have read and locked the message and want to delete it from the message buffer, call the **Microsoft.ServiceBus.MessageBufferClient.DeleteLockedMessage(System.ServiceModel.Channels.Message)** method to delete the locked message.

   ```
   client.DeleteLockedMessage(message);
   ```

3. You can also unlock the previously locked message by using a call to **Microsoft.ServiceBus.MessageBufferClient.ReleaseLock(System.ServiceModel.Channels.Message)**, as follows:

   ```
   client.ReleaseLock(message);
   ```

To perform a destructive read using the Windows Azure SDK

1. Retrieve the message from the message buffer with a call to the **Microsoft.ServiceBus.MessageBufferClient.Retrieve** method.

   ```
   Message message = client.Retrieve();
   ```

Example

Description

The following code example, taken from the code sample in Service Bus Message Buffer Overview, describes how to perform a destructive read in addition to a peek/lock read.

Code

```
// Retrieve a message (destructive read)

Message message = client.Retrieve();

string content = message.GetBody<string>();

message.Close();


Console.WriteLine("Retrieve message content: {0}", content);
```

```
// Retrieve a message (peek/lock)
message = client.PeekLock();
content = message.GetBody<string>();

Console.WriteLine("PeekLock message content: {0}", content);

// Delete previously locked message
client.DeleteLockedMessage(message);
message.Close();
```

Silverlight and Flash Support

The Windows Azure SDK now includes support for cross-domain scenarios for Silverlight and Flash clients.

Same Origin Policy and Cross Domain Access

Browsers and client-side programming languages, such as Javascript, enforce a security policy called the *same origin* policy. This policy restricts cross domain calls; that is, it only allows code running in the browser to call code that resides on the host site from which the browser code was downloaded. This policy does not allow code running in a browser to call code residing on a different site (the target site).

You can set up cross-domain access explicitly if the target site places a specific policy file at its root. For Silverlight, this policy file is called **ClientAccessPolicy.xml**. For Flash, the cross-domain policy file is called **CrossDomain.xml**. The following is an example of a simple ClientAccessPolicy.xml file:

```
<?xml version=""1.0"" encoding=""utf-8""?>
<access-policy>
<cross-domain-access>
<policy>
<allow-from http-request-headers="*" http-methods="*">
<domain uri="https://*"/>
<domain uri="http://*"/>
</allow-from>
<grant-to>
<resource path="/" include-subpaths="true" />
```

```
</grant-to>
</policy>
</cross-domain-access>
</access-policy>
```

For the Access Control service, by default all service namespaces have cross-domain access set up. For the Service Bus, you must explicitly configure cross-domain access. To enable uploads of the policy file to the root of the service namespace, the Service Bus provides REST commands (PUT, DELETE) that enable this. The following example is taken from the **MessageBufferForSilverlight** sample application in the Windows Azure SDK:

```
static HttpStatusCode PublishClientAccessPolicy(string serviceNamespace, string
authHeaderValue, byte[] fileContentArray)
{
    HttpWebRequest request =
(HttpWebRequest)WebRequest.Create(string.Format("https://{0}.{1}/clientaccesspolicy.xml",
serviceNamespace, ServiceBusUriPostFix));

    request.Method = "PUT";
    request.ContentType = "text/xml";

    request.ContentLength = fileContentArray.Length;
    request.Headers[AuthorizationHeader] = authHeaderValue;

    Stream dataStream = request.GetRequestStream();
    dataStream.Write(fileContentArray, 0, fileContentArray.Length);
    dataStream.Close();

    return SendHttpRequestAndGetResponse(request);
}
```

Similarly, the following code deletes the policy file:

```
static HttpStatusCode DeleteClientAccessPolicy(string serviceNamespace, string
authHeaderValue)
{
    HttpWebRequest request =
(HttpWebRequest)WebRequest.Create(string.Format("https://{0}.{1}/clientaccesspolicy.xml",
serviceNamespace, ServiceBusUriPostFix));
```

```
    request.Method = "DELETE";
    request.Headers[AuthorizationHeader] = authHeaderValue;
    return SendHttpRequestAndGetResponse(request);
}
```

Custom HTTP Header

To programmatically address secured endpoints, Windows Azure requires an Access Control token with a “Manage” claim to be sent through an **Authorization** HTTP header. Because Silverlight does not allow applications to write directly to the **Authorization** HTTP header, the Windows Azure Services API has introduced the **X-MS-Authorization** HTTP custom header, which behaves similarly to the **Authorization** HTTP header. The following example, also taken from the **MessageBufferForSilverlight** sample application, demonstrates the use of this custom header.

```
const string XMSAuthorizationHeader = "X-MS-Authorization";


public void CreateMessageBuffer(UploadStringCompletedEventHandler
createMessageBufferCompleted)
        {
            string bufferLocation = AddTrailingSlashIfNeeded(this.messageBufferLocation);


            WebClient webClient = new WebClient();
            webClient.BaseAddress = bufferLocation;
            webClient.Headers[XMSAuthorizationHeader] = this.authHeaderValue;
            webClient.Headers[HttpRequestHeader.ContentType] =
"application/atom+xml;type=entry;charset=utf-8";
            webClient.UploadStringCompleted += createMessageBufferCompleted;


            this.AppendOutputText("MessageBuffer.Create: Sending request to create
message buffer");
            webClient.UploadStringAsync(new Uri(bufferLocation), "PUT", DefaultPolicy,
"MessageBuffer.Create");
        }
```

For more information, see the **MessageBufferForSilverlight** sample application in the Windows Azure SDK.

Service Bus Troubleshooting

The following topics contain information and recommendations for troubleshooting applications that use the Windows Azure Service Bus.

In This Section

Troubleshooting the Service Bus

Hosting Behind a Firewall with the Service Bus

Troubleshooting the Service Bus

The following sections describe issues that may occur in writing applications for the Windows Azure Service Bus, and how to resolve them.

Troubleshooting Windows Azure Applications

The following sections describe troubleshooting Windows Azure applications.

Application runs in the development fabric, but crashes when it is deployed

Symptom

An Service Bus application runs without error in the deployment fabric, but crashes after it is deployed to the Windows Azure servers.

Cause

The Service Bus assemblies are currently not integrated into the Windows Azure platform. Therefore, the application is trying to access an assembly that does not exist on the server.

Resolution

Add the Service Bus assembly to your deployment package (for example, set the assembly to **Local Copy**). Note that therefore, you will have to update the assemblies manually.

Endpoints can be set programmatically, but fail when set in the configuration file

Symptom

An application works when the endpoints are set programmatically, but fail when those same configuration settings are stored in the associated App.config file. The application generates the following error:

System.Configuration.ConfigurationErrorsException: Configuration binding extension 'system.serviceModel/bindings/netTcpRelayBinding' could not be found. Verify that this binding extension is correctly registered in system.serviceModel/extensions/bindingExtensions and that it is spelled correctly.

Cause

The Service Bus assemblies are currently not integrated into the Windows Azure platform. On the local computer, if you have the Windows Azure SDK installed, your Machine.config file will be modified to add several Windows Communication Foundation (WCF) extensions, such as **Microsoft.ServiceBus.NetTcpRelayBinding**. The application can then find the relevant information in the Machine.config file, such as in the `netTcpRelayBinding` section. This does not occur on the Windows Azure platform, therefore the application cannot find the relevant information.

Resolution

For your configuration to work with Windows Azure, copy the extensions from the local Machine.config file to your App.config file. Otherwise, tags such as `netTcpRelayBinding` will not be recognized. However, if you do this, your application will not run locally because of duplicate extensions. Therefore, you will have to keep two versions of the App.config file; one locally, and one for Windows Azure. It is recommended that you to perform this particular task programmatically.

Connectivity Issues

The following troubleshooting topics contain information about how to connect to the Service Bus. If you cannot find your solution later in this section, you may want to consider one of the following possibilities:

- Run a network trace – the network may be down. Using Network Monitor to determine the status of the network may assist you in debugging your problem.

Client application cannot find the targeted endpoint

Symptom

When you attempt to connect to the Service Bus with a client application, you receive the following error:

Unhandled Exception: System.ServiceModel.EndpointNotFoundException: The endpoint was not found. Please make sure that you can connect to the Internet using HTTP port 80 and TCP port 808.

Cause

There are a variety of possible causes for this error.

Resolution

- Check to see whether the host is running. If not, there is no endpoint to which to connect. You can run the service by using one of the two procedures that are shown here:
    a. Debug Mode - Right click your service project in the Visual Studio **Solution Explorer**, click **Debug**. Then click **Start new instance**. After the service starts, repeat this procedure for the client. Note that you can debug the client even though the service is already running.
    b. Outside Debug Mode - Set your service project as the start-up project. From the **Debug** menu, click **Start Without Debugging**. By doing this, although the service application will

run, Visual Studio is not affected. After the service starts, set the client project as the startup project, then run it.

- Ensure that you have set the **Copy Local** property for the Microsoft.ServiceBus.dll assembly (in Visual Studio) to **true**. The cloud servers do not have the Windows Azure SDK installed. Therefore, you must include the assembly with your package. Without it, your worker role will not run correctly.
- Confirm that you can, in fact, connect to the Internet using HTTP port 80 and TCP port 808.

Client is unable to finish the security negotiation in the configured time-out

Symptom

The application is not able to connect to the Service Bus, but instead returns the following error message:

"Client is unable to finish the security negotiation in the configured time-out (00:01:00). The current negotiation leg is 1 (00:00:59.9429968)."

Cause

The authentication credentials may be set incorrectly.

Resolution

Check to see whether you are authenticating the client. Specifically, check to see whether you have you set **Microsoft.ServiceBus.RelayClientAuthenticationType** to **Microsoft.ServiceBus.RelayClientAuthenticationType.None**. By default, the value is **Microsoft.ServiceBus.RelayClientAuthenticationType.RelayAccessToken**, which requires that you provide an authentication claim, such as a shared secret. However, if you manually set the value to **None**, the client should not provide any authentication.

If you have not set the authentication type, you likely have **Microsoft.ServiceBus.RelayClientAuthenticationType** with the default **Microsoft.ServiceBus.RelayClientAuthenticationType.RelayAccessToken** value. If this is the case, check to see whether your authentication claim is specified correctly. For more information, seeSecuring and Authenticating a Service Bus Connection.

Application cannot verify security when it connects to the Service Bus

Symptom

When trying to connect to the Service Bus, you receive the following error:

"An error occurred when verifying security for the message"

Cause

One possible cause of this error is that the UTC time for the local computer is ahead of the UTC time on the Service Bus server. For example, the local UTC time stamp may be 8:06, whereas on the server the time stamp is 8:05. The server considers this to be an invalid time stamp, and generates the error message that is mentioned in the "Symptoms" section.

Resolution

Confirm that the UTC time for the local computer is correct. If necessary, manually set the clock back several minutes. The Service Bus does not consider slightly older messages to be a security violation; only ones that appear to come from further in the future.

The ATOM feed does not display your service

Symptom

You cannot locate a service on the ATOM feed, even though you know the service is successfully running.

Cause

The default behavior for the service registry is not to expose a service through the ATOM feed.

Resolution

Set the endpoint to be discoverable, as described in How to: Publish a Service to the Service Bus Registry.

Hosting Behind a Firewall with the Service Bus

This topic describes several ways to connect to the Windows Azure Service Bus from behind a firewall or through a proxy server.

Troubleshoot your Firewall Connection

The following troubleshooting topics discuss common solutions to problems encountered when you connect through a firewall to the Service Bus.

Configure the Ports on your Firewall

To use the Service Bus relay, ensure that your firewall allows outgoing TCP communication on TCP ports 9350 to 9354. For Service Bus brokered messaging, use port 9354.

Configure the WinHTTP Proxy Settings

If you are running behind a firewall/proxy that requires authentication, or if you are running in an IPsec-protected network, there are additional obstacles for any client to reach the network proxy. For example, Windows accounts might not have permissions to communicate through the firewall. Therefore, you might have to explicitly configure the WinHTTP proxy settings with the appropriate credentials.

Set OpenTimeout

Setting the connectivity mode to HTTP (that is, `ConnectivityMode = http`) may cause connections in the presence of some proxies to be very slow. For example, some connections can require up to 20 seconds to connect. Extending the **OpenTimeout** option for the service to up to two minutes can help, because you might run out of time between the acquisition of the Access

Control token and getting the Web stream working. After the Web stream is established, the throughput often improves.

RelayConfigurationInstaller.exe Tool

The RelayConfigurationInstaller.exe tool is located in the <installdir>/Assemblies directory of the Windows Azure SDK, and enables you to easily add the Machine.config settings necessary for the Service Bus bindings to be supported in the configuration file. You can also use the **Microsoft.ServiceBus.Configuration.RelayConfigurationInstaller** class to accomplish this programmatically.

The primary scenario for the RelayConfigurationInstaller.exe tool is to help in installing the necessary Machine.config or App.config information that is required to run an Service Bus application on a computer that does not have the Service Bus installed. However, in most scenarios, the application installer for the Service Bus application automatically installs the necessary configuration information. Therefore, the tool or class would likely be used by developers who want additional control over the installation process.

Command-line Options

/i

Adds the entries to the configuration file to allow an application that uses that configuration to use Service Bus elements.

/u

Removes the entries from the Machine.config file.

Best Practices for Performance Improvements Using Service Bus Brokered Messaging

This topic describes how to use the Windows Azure Service Bus to optimize performance when exchanging brokered messages. The first half of this topic describes the different mechanisms that are offered to help increase performance. The second half provides guidance on how to use the Service Bus in a way that can offer the best performance for a given scenario.

Throughout this topic, the term “client” refers to any entity that accesses the Service Bus. A client can take the role of a sender or a receiver. The term “sender” is used for a Service Bus queue or topic client that sends messages to a Service Bus queue or topic. The term “receiver” refers to a Service Bus queue or subscription client that receives messages from a Service Bus queue or subscription.

Mechanisms

This section introduces different concepts employed by the Service Bus to help boost performance.

Protocols

The Service Bus enables clients to send and receive messages via two protocols: the Service Bus client protocol, and HTTP. The Service Bus client protocol is more efficient, because it maintains the connection to the Service Bus service as long as the message factory exists. It also implements batching and prefetching. The Service Bus client protocol is available for .NET applications using the .NET managed API.

Unless explicitly mentioned, all content in this topic assumes the use of the Service Bus client protocol.

Reusing factories and clients

Service Bus client objects, such as **Microsoft.ServiceBus.Messaging.QueueClient** or **Microsoft.ServiceBus.Messaging.MessageSender**, are created through a **Microsoft.ServiceBus.Messaging.MessagingFactory** object, which also provides internal management of connections. You should not close messaging factories or queue, topic, and subscription clients after you send a message, and then re-create them when you send the next message. Closing a messaging factory deletes the connection to the Service Bus service, and a new connection is established when recreating the factory. Establishing a connection is an expensive operation that can be avoided by re-using the same factory and client objects for multiple operations.

Concurrent operations

Performing an operation (send, receive, delete, etc.) takes some time. This time includes the processing of the operation by the Service Bus service in addition to the latency of the request and the reply. To increase the number of operations per time, operations must execute concurrently. You can do this in several different ways:

Asynchronous operations: the client pipelines operations by performing asynchronous operations. The next request is started before the previous request is completed. The following is an example of an asynchronous **send** operation:

```
BrokeredMessage m1 = new BrokeredMessage(body);
BrokeredMessage m2 = new BrokeredMessage(body);
```

```
queueClient.BeginSend(m1, processEndSend, queueClient); // Send
message 1.
queueClient.BeginSend(m2, processEndSend, queueClient); // Send
message 2.

void processEndSend(IAsyncResult result)
{
    QueueClient qc = result.AsyncState as QueueClient;
    qc.EndSend(result);
    Console.WriteLine("Message sent");
}
```

The following is an example of an asynchronous **receive** operation:

```
queueClient.BeginReceive(processEndReceive, queueClient); //
Receive message 1.
queueClient.BeginReceive(processEndReceive, queueClient); //
Receive message 2.

void processEndReceive(IAsyncResult result)
{
    QueueClient qc = result.AsyncState as QueueClient;
    BrokeredMessage m = qc.EndReceive(result);
    m.BeginComplete(processEndComplete, m);
    Console.WriteLine("Received message " + m.Label);
}

void processEndComplete(IAsyncResult result)
{
    QueueClient qc = result.AsyncState as QueueClient;
    BrokeredMessage m = result.AsyncState as BrokeredMessage;
    m.EndComplete(result);
    Console.WriteLine("Completed message " + m.Label);
}
```

Multiple factories: all clients (senders in addition to receivers) that are created by the same factory share one TCP connection. The maximum message throughput is limited by the number of operations that can go through this TCP connection. The throughput that can be

obtained with a single factory varies greatly with TCP round-trip times and message size. In benchmarks, a maximum throughput per factory of around 800msg/s (message size: 1KB) has been observed. To obtain rates beyond this, you should use multiple messaging factories.

Receive mode

When creating a queue or subscription client, you can specify a receive mode: *Peek-lock* or *Receive and delete*. The default receive mode is **Microsoft.ServiceBus.Messaging.ReceiveMode.PeekLock**. When operating in this mode, the client sends a request to receive a message from the Service Bus. After the client has received the message, it sends a request to complete the message.

When setting the receive mode to **Microsoft.ServiceBus.Messaging.ReceiveMode.ReceiveAndDelete**, both steps are combined in a single request. This reduces the overall number of operations, and can improve the overall message throughput. This performance gain comes at the risk of losing messages.

The September 2011 release of the Service Bus does not support transactions for receive-and-delete operations. In addition, peek-lock semantics are required for any scenarios in which the client wants to defer or deadletter a message.

Client-side batching

Client-side batching enables a queue or topic client to delay the sending of a message for a certain period of time. If the client sends additional messages during this time period, it transmits the messages in a single batch. Client-side batching also causes a queue/subscription client to batch multiple **Complete** requests into a single request. Batching is only available for asynchronous **Send** and **Complete** operations. Synchronous operations are immediately sent to the Service Bus service. Batching does not occur for peek or receive operations, nor does batching occur across clients.

If the batch exceeds the maximum message size, the last message is removed from the batch, and the client immediately sends the batch. The last message becomes the first message of the next batch. By default, a client uses a batch interval of 20ms. You can change the batch interval by setting the **Microsoft.ServiceBus.Messaging.NetMessagingTransportSettings.BatchFlushInterval** property before creating the messaging factory. This setting affects all clients that are created by this factory.To disable batching, set the **Microsoft.ServiceBus.Messaging.NetMessagingTransportSettings.BatchFlushInterval** property to **TimeSpan.Zero**. For example:

```
MessagingFactorySettings mfs = new MessagingFactorySettings();

mfs.TokenProvider = tokenProvider;

mfs.NetMessagingTransportSettings.BatchFlushInterval = TimeSpan.FromSeconds(0.05);

MessagingFactory messagingFactory = MessagingFactory.Create(namespaceUri, mfs);
```

Batching does not affect the number of billable messaging operations, and is available only for the Service Bus client protocol. The HTTP protocol does not support batching.

Batching store access

To increase the throughput of a queue/topic/subscription, the Service Bus service batches multiple messages when it writes to its internal store. If enabled on a queue or topic, writing messages into the store will be batched. If enabled on a queue or subscription, deleting messages from the store will be batched. If batched store access is enabled for an entity, the Service Bus delays a store write operation regarding that entity by up to 20ms. Additional store operations that occur during this interval are added to the batch. Batched store access only affects **Send** and **Complete** operations; receive operations are not affected. Batched store access is a property on an entity. Batching occurs across all entities that enable batched store access.

When creating a new queue, topic or subscription, batched store access is enabled by default. To disable batched store access, set the **Microsoft.ServiceBus.Messaging.QueueDescription.EnableBatchedOperations** property to **false** before creating the entity. For example:

```
QueueDescription qd = new QueueDescription();

qd.EnableBatchedOperations = false;

Queue q = namespaceManager.CreateQueue(qd);
```

Batched store access does not affect the number of billable messaging operations, and is a property of a queue, topic, or subscription. It is independent of the receive mode and the protocol that is used between a client and the Service Bus service.

Prefetching

Prefetching enables the queue or subscription client to load additional messages from the service when it performs a receive operation. The client stores these messages in a local cache. The size of the cache is determined by the **Microsoft.ServiceBus.Messaging.QueueClient.PrefetchCount** and **Microsoft.ServiceBus.Messaging.SubscriptionClient.PrefetchCount** properties. Each client that enables prefetching maintains its own cache. A cache is not shared across clients. If the client initiates a receive operation and its cache is empty, the service transmits a batch of messages. The size of the batch equals the size of the cache or 256KB, whichever is smaller. If the client initiates a receive operation and the cache contains a message, the message is taken from the cache.

When a message is prefetched, the service locks the prefetched message. By doing this, the prefetched message cannot be received by a different receiver. If the receiver cannot complete the message before the lock expires, the message becomes available to other receivers. The prefetched copy of the message remains in the cache. The receiver that consumes the expired cached copy will receive an exception when it tries to complete that message. By default, the message lock expires after 60 seconds. This value can be extended to 5 minutes. To prevent the

consumption of expired messages, the cache size should always be smaller than the number of messages that can be consumed by a client within the lock time-out interval.

When using the default lock expiration of 60 seconds, a good value for SubscriptionClient.PrefetchCount is 20 times the maximum processing rates of all receivers of the factory. For example, a factory creates 3 receivers. Each receiver can process up to 10 messages per second. The prefetch count should not exceed 20*3*10 = 600.By default, QueueClient.PrefetchCount is set to 0, which means that no additional messages are fetched from the service.

Prefetching messages increases the overall throughput for a queue or subscription because it reduces the overall number of message operations, or round trips. Fetching the first message, however, will take longer (due to the increased message size). Receiving prefetched messages will be faster because these messages have already been downloaded by the client.

The time-to-live (TTL) property of a message is checked by the server at the time the server sends the message to the client. The client does not check the message's TTL property when the message is received. Instead, the message can be received even if the message's TTL has passed while the message was cached by the client.

Prefetching does not affect the number of billable messaging operations, and is available only for the Service Bus client protocol. The HTTP protocol does not support prefetching. Prefetching is available for synchronous and asynchronous receive operations.

Use of multiple queues

Internally, the Service Bus uses the same node to process all messages for an entity. To achieve throughput beyond several thousand messages per second, the messages must be distributed through multiple entities. Note that all subscription of a topic are handled by the same node that handles the topic. When using multiple entities, you should use a dedicated client for each of these entities instead of employing the same client for all entities.

Scenarios

The following sections describe typical messaging scenarios and outline the preferred Service Bus settings. Throughput rates are classified as small (<1msg/s), moderate (≥1msg/s, <100msg/s) and high (≥100msg/s). The number of clients are classified as small (≤5), moderate (>5, ≤20), and large (>20).

High-throughput queue

Goal: Maximize throughput of a single queue. The number of senders and receivers is small.

- To increase the overall send rate into the queue, use multiple message factories to create senders. For each sender, use asynchronous operations or multiple threads.
- To increase the overall receive rate from the queue, use multiple message factories to create receivers.
- Use asynchronous operations to take advantage of client-side batching.

- Set the batching interval to 50ms to reduce the number of Service Bus client protocol transmissions. If multiple senders are used, increase the batching interval to 100ms.
- Leave batched store access enabled. This increases the overall rate at which messages can be written into the queue.
- Set the prefetch count to 20 times the maximum processing rates of all receivers of a factory. This reduces the number of Service Bus client protocol transmissions.

Benchmarks suggest that a single queue can achieve a message throughput of up to 2000msg/s (message size: 1KB). To obtain higher throughput, use multiple queues.

Multiple high-throughput queues

Goal: Maximize overall throughput of multiple queues. The throughput of an individual queue is moderate or high.

To obtain maximum throughput across multiple queues, use the settings outlined to maximize the throughput of a single queue. In addition, use different factories to create clients that send or receive from different queues.

Low latency queue

Goal: Minimize end-to-end latency of a queue or topic. The number of senders and receivers is small. The throughput of the queue is small or moderate.

- Disable client-side batching. The client immediately sends a message.
- Disable batched store access. The service immediately writes the message to the store.
- If using a single client, set the prefetch count to 20 times the processing rate of the receiver. If multiple messages arrive at the queue at the same time, the Service Bus client protocol transmits them all at the same time. When the client receives the next message, that message is already in the local cache. The cache should be small.
- If using multiple clients, set the prefetch count to 0. By doing this, the second client can receive the second message while the first client is still processing the first message.

Queue with a large number of senders

Goal: Maximize throughput of a queue or topic with a large number of senders. Each sender sends messages with a moderate rate. The number of receivers is small.

The Service Bus enables up to 100 concurrent connections to an entity. For queues, this number is shared between senders and receivers. If all 100 connections are required for senders, you should replace the queue with a topic and a single subscription. A topic accepts up to 100 concurrent connections from senders, whereas the subscription accepts an additional 100 concurrent connections from receivers. If more than 100 concurrent senders are required, the senders should send messages to the Service Bus protocol via HTTP.

To maximize throughput, do the following:

- If each sender resides in a different process, use only a single factory per process.
- Use asynchronous operations to take advantage of client-side batching.

- Use the default batching interval of 20ms to reduce the number of Service Bus client protocol transmissions.
- Leave batched store access enabled. This increases the overall rate at which messages can be written into the queue or topic.
- Set the prefetch count to 20 times the maximum processing rates of all receivers of a factory. This reduces the number of Service Bus client protocol transmissions.

Queue with a large number of receivers

Goal: Maximize the receive rate of a queue or subscription with a large number of receivers. Each receiver receives messages at a moderate rate. The number of senders is small.

The Service Bus enables up to 100 concurrent connections to an entity. If a queue requires more than 100 receivers, you should replace the queue with a topic and multiple subscriptions. Each subscription can support up to 100 concurrent connections. Alternatively, receivers can access the queue via the HTTP protocol.

To maximize throughput, do the following:

- If each receiver resides in a different process, use only a single factory per process.
- Receivers can use synchronous or asynchronous operations. Given the moderate receive rate of an individual receiver, client-side batching of a Complete request does not affect receiver throughput.
- Leave batched store access enabled. This reduces the overall load of the entity. It also reduces the overall rate at which messages can be written into the queue or topic.
- Set the prefetch count to a small value (for example, PrefetchCount = 10). This prevents receivers from being idle while other receivers have large numbers of messages cached.

Benchmarks suggest that a single topic with 5 subscriptions can achieve a message throughput of up to 600msg/s (message size: 1KB) if all messages are routed to all subscriptions. To obtain higher throughput, multiple topics must be used.

Topic with a small number of subscriptions

Goal: Maximize the throughput of a topic with a small number of subscriptions. A message is received by many subscriptions, which means the combined receive rate over all subscriptions is larger than the send rate. The number of senders is small. The number of receivers per subscription is small.

To maximize throughput, do the following:

- To increase the overall send rate into the topic, use multiple message factories to create senders. For each sender, use asynchronous operations or multiple threads.
- To increase the overall receive rate from a subscription, use multiple message factories to create receivers. For each receiver, use asynchronous operations or multiple threads.
- Use asynchronous operations to take advantage of client-side batching.
- Use the default batching interval of 20ms to reduce the number of Service Bus client protocol transmissions.

- Leave batched store access enabled. This increases the overall rate at which messages can be written into the topic.
- Set the prefetch count to 20 times the maximum processing rates of all receivers of a factory. This reduces the number of Service Bus client protocol transmissions.

Benchmarks suggest that a single topic with 5 subscriptions can achieve a message throughput of up to 600msg/s (message size: 1KB) if all messages are routed to all subscriptions. To obtain higher throughput, use multiple topics.

Topic with a large number of subscriptions

Goal: Maximize the throughput of a topic with a large number of subscriptions. A message is received by many subscriptions, which means the combined receive rate over all subscriptions is much larger than the send rate. The number of senders is small. The number of receivers per subscription is small.

Topics with a large number of subscriptions typically expose a low overall throughput if all messages are routed to all subscriptions. This is caused by the fact that each message is received many times, and all messages that are contained in a topic and all its subscriptions are stored in the same store. It is assumed that the number of senders and number of receivers per subscription is small. The September 2011 release of Service Bus supports up to 2,000 subscriptions per topic.

To maximize throughput, do the following:

- Use asynchronous operations to take advantage of client-side batching.
- Use the default batching interval of 20ms to reduce the number of Service Bus client protocol transmissions.
- Leave batched store access enabled. This increases the overall rate at which messages can be written into the topic.
- Set the prefetch count to 20 times the expected receive rate in seconds. This reduces the number of Service Bus client protocol transmissions.

Benchmarks suggest that a single topic with 250 subscriptions can achieve a message throughput of up to 5msg/s (message size: 1KB) if all messages are routed to all subscriptions. To obtain higher throughput, use multiple topics.

Appendix: Messaging Exceptions

This section lists the various exceptions generated by the Service Bus messaging API. This reference is subject to change, so check back for updates.

Exception Categories

The messaging API generate exceptions that can fall into the following categories, with the associated action you can take to try to fix them:

1. User code error (**System.ArgumentException**, **System.InvalidOperationException**, **System.ObjectDisposedException**, **System.Runtime.Serialization.SerializationException**). General action: try to fix the code before proceeding.
2. Setup/configuration error (**Microsoft.ServiceBus.Messaging.MessagingEntityNotFoundException**, **System.UnauthorizedAccessException**. General action: review your configuration and change if necessary.
3. Transient exceptions (**Microsoft.ServiceBus.Messaging.ServerBusyException**, **Microsoft.ServiceBus.Messaging.MessagingCommunicationException**). General action: retry the operation or notify users.
4. Other exceptions (**Microsoft.ServiceBus.Messaging.MessagingException**, **System.Transactions.TransactionException**, **System.TimeoutException**, **Microsoft.ServiceBus.Messaging.MessageLockLostException**/**Microsoft.ServiceBus.Messaging.SessionLockLostException**). General action: you generally do not handle these exceptions to perform cleanup or aborts. They might be used for tracing.

Exception Types

The following table lists messaging exception types, and their causes, and notes suggested action you can take.

Exception Type	Description/Cause/Examples	Suggested Action	Note on automatic/immediate retry
System.TimeoutException	The server did not respond to the requested operation within the specified time which is controlled by **Microsoft.ServiceBus.Messaging.MessagingFactorySettings.OperationTimeout**. The server may have completed the requested operation.This can happen due to network or other infrastructure delays.	Check the system state for consistency and retry if necessary.	Retry might help in some cases; add retry logic to code.
System.InvalidOperationException	The requested operation is not supported on the current state of the object. **Microsoft.ServiceBus.Messaging.BrokeredMessage.Complete** will generate this exception if the message was received in **ReceiveAndDelete** mode.	Check the code and the documentation. Make sure the requested operation is valid.	Retry will not help.

Exception Type	Description/Cause/Examples	Suggested Action	Note on automatic/immediate retry
System.ObjectDisposedException	An attempt is made to invoke an operation on an object that has already been closed, aborted or disposed.In rare cases, the ambient transaction is already disposed.	Check the code and make sure it does not invoke operations on a disposed object.	Retry will not help.
System.UnauthorizedAccessException	The **Microsoft.ServiceBus.TokenProvider** object could not acquire a token, the token is invalid, or the token does not contain the claims required to perform the operation.	Make sure the token provider is created with the correct values. Check the configuration of the Access Control service.	Retry might help in some cases; add retry logic to code.
System.ArgumentException **System.ArgumenNullException** **System.ArgumenOutOfRangeException**	• One or more arguments supplied to the method are invalid. • The URI supplied to **Microsoft.ServiceBus.NamespaceManager** or **Microsoft.ServiceBus.Messaging.MessagingFactory.Create(System.Uri,Microsoft.ServiceBus.Messaging.MessagingFactorySettings)** contains path segment(s). • The URI scheme supplied to **Microsoft.ServiceBus.NamespaceManager** or **Microsoft.ServiceBus.Messaging.MessagingFactory.Create(System.Uri,Microsoft.ServiceBus.Messaging.MessagingFactorySettings)** is invalid. • The property value is larger than 32KB.	Check the calling code and make sure the arguments are correct.	Retry will not help.
Microsoft.ServiceBus.Messaging.MessagingEntityNotFoundException	Entity associated with the operation does not exist or it has been deleted.	Make sure the entity exists.	Retry will not help.

Exception Type	Description/Cause/Examples	Suggested Action	Note on automatic/immediate retry
Microsoft.ServiceBus.Messaging.MessagingCommunicationException	Client is not able to establish a connection to the Service Bus.	Make sure the supplied host name is correct and the host is reachable.	Retry might help if there are intermittent connectivity issues.
Microsoft.ServiceBus.Messaging.ServerBusyException	Service is not able to process the request at this time.	Client may retry the operation.	Client may retry after certain interval. If a retry results in a different exception, check retry behavior of that exception.
Microsoft.ServiceBus.Messaging.MessageLockLostException	Lock token associated with the message has expired or the lock token is not found.	Dispose the message.	Retry will not help.
Microsoft.ServiceBus.Messaging.SessionLockLostException	Lock associated with this session is lost.	Abort the **Microsoft.ServiceBus.Messaging.MessageSession** object.	Retry will not help.
Microsoft.ServiceBus.Messaging.MessagingException	Generic messaging exception that may be thrown in the following cases: • An attempt is made to create a **Microsoft.ServiceBus.Messaging.QueueClient** using a name or path that belongs to a different entity type (for example, a topic). • An attempt is made to send a message larger than 256KB. • An internal server error might also be one of the reasons this error is thrown. In that case, the exception message will indicate the same, and this is usually a transient exception.	• Check the code and ensure that only serializable objects are used for the message body (or use a custom serializer). • Check the documentation for the supported	Retry behavior is undefined and might not help.

Exception Type	Description/Cause/Examples	Suggested Action	Note on automatic/immediate retry
		value types of the properties and only use supported types.	
System.Transactions.TransactionException	The ambient transaction (**System.Transactions.Transaction.Current**) is invalid. It may have been completed or aborted. Inner exception may provide additional information.		Retry will not help.
System.Transactions.TransactionInDoubtException	An operation is attempted on a transaction that is in doubt, or an attempt is made to commit the transaction and the transaction becomes in doubt.	You application must handle this exception (as a special case), as the transaction may have already been committed.	