Table of Contents

Service Bus Explorer

Service updates

Videos

Service Bus messaging: flexible data delivery in the cloud

2/16/2017 • 4 min to read • Edit on GitHub

Microsoft Azure Service Bus is a reliable information delivery service. The purpose of this service is to make communication easier. When two or more parties want to exchange information, they need a communication facilitator. Service Bus is a brokered, or third-party communication mechanism. This is similar to a postal service in the physical world. Postal services make it very easy to send different kinds of letters and packages with a variety of delivery guarantees, anywhere in the world.

Similar to the postal service delivering letters, Service Bus is flexible information delivery from both the sender and the recipient. The messaging service ensures that the information is delivered even if the two parties are never both online at the same time, or if they aren't available at the exact same time. In this way, messaging is similar to sending a letter, while non-brokered communication is similar to placing a phone call (or how a phone call used to be - before call waiting and caller ID, which are much more like brokered messaging).

The message sender can also require a variety of delivery characteristics including transactions, duplicate detection, time-based expiration, and batching. These patterns have postal analogies as well: repeat delivery, required signature, address change, or recall.

Service Bus supports two distinct messaging patterns: *Azure Relay* and *Service Bus Messaging*.

Azure Relay

The WCF Relay component of Service Bus is a centralized (but highly load-balanced) 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 help negotiate direct peer-to-peer connections when it is possible. 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 Service Bus.

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-premises 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 initiate the connection between your on-premises service and the relay service, using a suite of WCF "relay" bindings. Behind the scenes, the relay bindings map to transport binding elements designed to create WCF channel components that integrate with Service Bus in the cloud.

WCF Relay 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 relay scheme, Service Bus Messaging, or 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 in a "broker" (such as a queue) 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 entire 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 the Service Bus brokered messaging infrastructure are queues, topics, and subscriptions. The primary difference is that topics support publish/subscribe capabilities that can be used for sophisticated content-based routing and delivery logic, including sending to multiple recipients. These components enable new asynchronous messaging scenarios, such as temporal decoupling, publish/subscribe, and load balancing. For more information about these messaging entities, see Service Bus queues, topics, and subscriptions.

As with the WCF Relay infrastructure, the brokered messaging capability is provided for WCF and .NET Framework programmers, and also via REST.

Next steps

To learn more about Service Bus Messaging, see the following topics.

- Service Bus fundamentals
- Service Bus queues, topics, and subscriptions
- How to use Service Bus queues
- How to use Service Bus topics and subscriptions

Azure Service Bus

3/8/2017 • 9 min to read • Edit on GitHub

Whether an application or service runs in the cloud or on premises, it often needs to interact with other applications or services. To provide a broadly useful way to do this, Microsoft Azure offers Service Bus. This article takes a look at this technology, describing what it is and why you might want to use it.

Service Bus fundamentals

Different situations call for different styles of communication. Sometimes, letting applications send and receive messages through a simple queue is the best solution. In other situations, an ordinary queue isn't enough; a queue with a publish-and-subscribe mechanism is better. In some cases, all that's really needed is a connection between applications; queues aren't required. Service Bus provides all three options, enabling your applications to interact in several different ways.

Service Bus is a multi-tenant cloud service, which means that the service is shared by multiple users. Each user, such as an application developer, creates a *namespace*, then defines the communication mechanisms she needs within that namespace. Figure 1 shows how this looks.

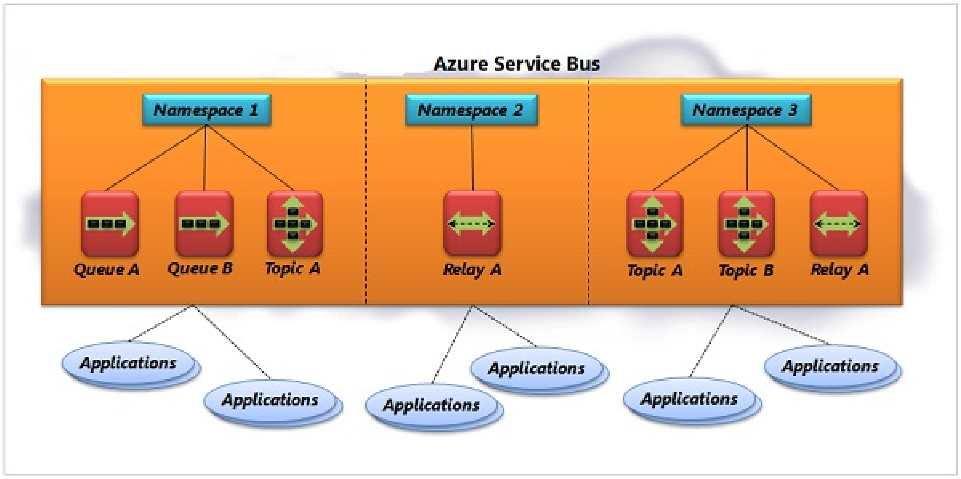


Figure 1: Service Bus provides a multi-tenant service for connecting applications through the cloud.

Within a namespace, you can use one or more instances of three different communication mechanisms, each of which connects applications in a different way. The choices are:

- *Queues*, which allow one-directional communication. Each queue acts as an intermediary (sometimes called a *broker*) that stores sent messages until they are received. Each message is received by a single recipient.
- *Topics*, which provide one-directional communication using *subscriptions*-a single topic can have multiple subscriptions. Like a queue, a topic acts as a broker, but each subscription can optionally use a filter to receive only messages that match specific criteria.
- *Relays*, which provide bi-directional communication. Unlike queues and topics, a relay doesn't store in-flight messages; it's not a broker. Instead, it just passes them on to the destination application.

When you create a queue, topic, or relay, you give it a name. Combined with whatever you called your namespace, this name creates a unique identifier for the object. Applications can provide this name to Service Bus, then use that queue, topic, or relay to communicate with one another.

To use any of these objects in the relay scenario, Windows applications can use Windows Communication Foundation (WCF). For queues and topics, Windows applications can use Service Bus-defined messaging APIs. To make these objects easier to use from non-Windows applications, Microsoft provides SDKs for Java, Node.js, and other languages. You can also access queues and topics using REST APIs over HTTP(s).

It's important to understand that even though Service Bus itself runs in the cloud (that is, in Microsoft's Azure datacenters), applications that use it can run anywhere. You can use Service Bus to connect applications running on Azure, for example, or applications running inside your own datacenter. You can also use it to connect an application running on Azure or another cloud platform with an on-premises application or with tablets and phones. It's even possible to connect household appliances, sensors, and other devices to a central application or to one other. Service Bus is a communication mechanism in the cloud that's accessible from pretty much anywhere. How you use it depends on what your applications need to do.

Queues

Suppose you decide to connect two applications using a Service Bus queue. Figure 2 illustrates this situation.

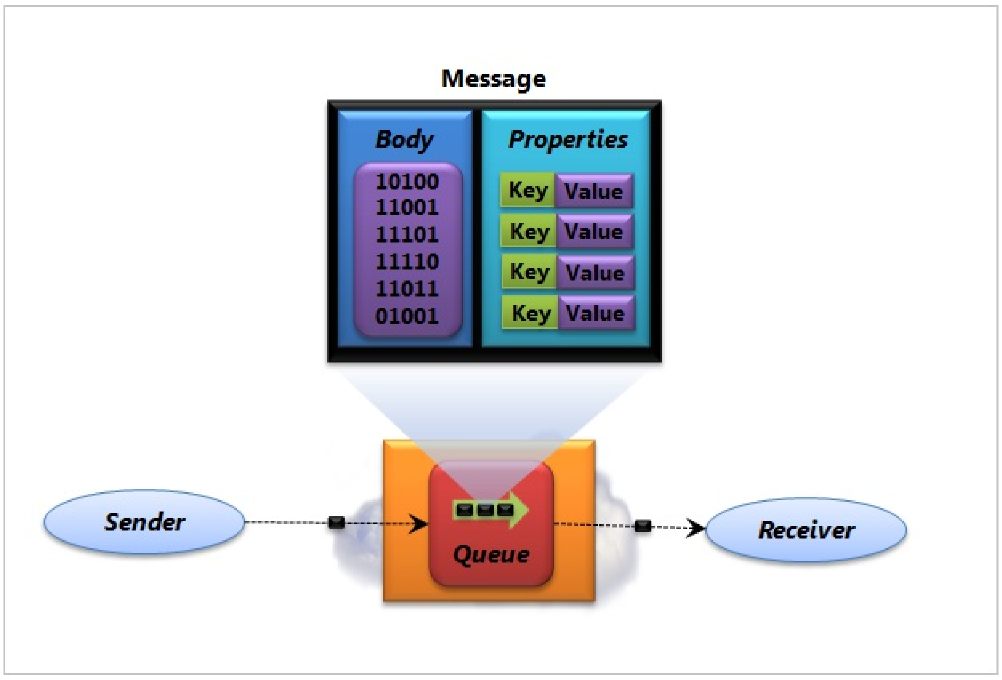


Figure 2: Service Bus queues provide one-way asynchronous queuing.

The process is simple: A sender sends a message to a Service Bus queue, and a receiver picks up that message at some later time. A queue can have just a single receiver, as Figure 2 shows. Or, multiple applications can read from the same queue. In the latter situation, each message is read by just one receiver. For a multi-cast service, you should use a topic instead.

Each message has two parts: a set of properties, each a key/value pair, and a message payload. The payload can be binary, text, or even XML. How they're used depends on what an application is trying to do. For example, an application sending a message about a recent sale might include the properties *Seller="Ava"* and *Amount=10000*. The message body might contain a scanned image of the sale's signed contract or, if there isn't one, just remain empty.

A receiver can read a message from a Service Bus queue in two different ways. The first option, called *ReceiveAndDelete*, removes a message from the queue and immediately deletes it. This is simple, but if the receiver crashes before it finishes processing the message, the message will be lost. Because it's been removed from the queue, no other receiver can access it.

The second option, *PeekLock*, is meant to help with this problem. Like **ReceiveAndDelete**, a **PeekLock** read removes a message from the queue. It doesn't delete the message, however. Instead, it locks the message, making it invisible to other receivers, then waits for one of three events:

- If the receiver processes the message successfully, it calls Complete(), and the queue deletes the message.

- If the receiver decides that it can't process the message successfully, it calls Abandon(). The queue then removes the lock from the message and makes it available to other receivers.
- If the receiver calls neither of these within a configurable period of time (by default, 60 seconds), the queue assumes the receiver has failed. In this case, it behaves as if the receiver had called **Abandon**, making the message available to other receivers.

Notice what can happen here: the same message might be delivered twice, perhaps to two different receivers. Applications using Service Bus queues must be prepared for this. To make duplicate detection easier, each message has a unique MessageID property that by default stays the same no matter how many times the message is read from a queue.

Queues are useful in quite a few situations. They enable applications to communicate even when both aren't running at the same time, something that's especially handy with batch and mobile applications. A queue with multiple receivers also provides automatic load balancing, since sent messages are spread across these receivers.

Topics

Useful as they are, queues aren't always the right solution. Sometimes, Service Bus topics are better. Figure 3 illustrates this idea.

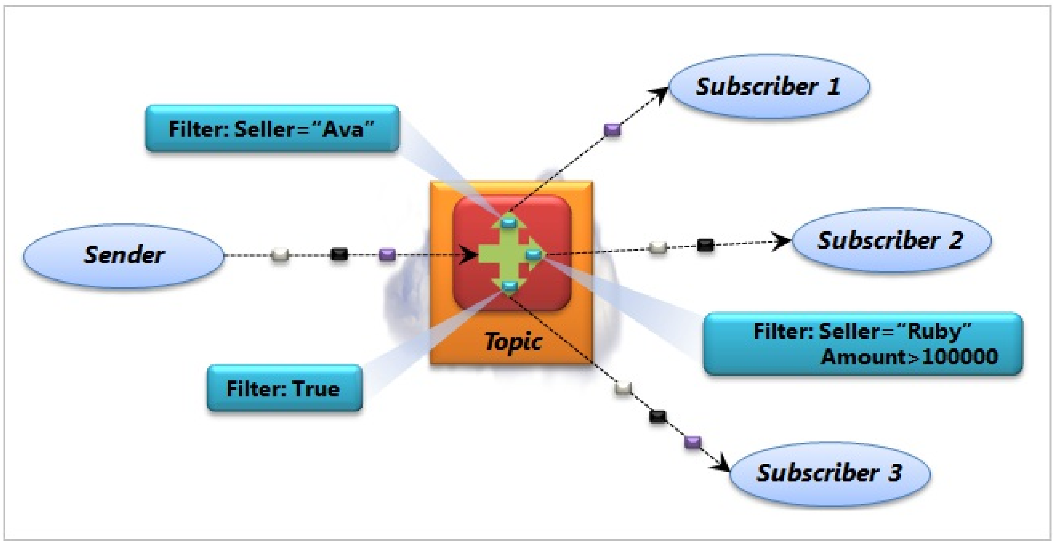


Figure 3: Based on the filter a subscribing application specifies, it can receive some or all of the messages sent to a Service Bus topic.

A *topic* is similar in many ways to a queue. Senders submit messages to a topic in the same way that they submit messages to a queue, and those messages look the same as with queues. The big difference is that topics enable each receiving application to create its own *subscription* by defining a *filter*. A subscriber will then see only the messages that match that filter. For example, Figure 3 shows a sender and a topic with three subscribers, each with its own filter:

- Subscriber 1 receives only messages that contain the property *Seller*="*Ava*".
- Subscriber 2 receives messages that contain the property *Seller*="*Ruby*" and/or contain an *Amount* property whose value is greater than 100,000. Perhaps Ruby is the sales manager, so she wants to see both her own sales and all big sales regardless of who makes them.
- Subscriber 3 has set its filter to *True*, which means that it receives all messages. For example, this application might be responsible for maintaining an audit trail and therefore it needs to see all the messages.

As with queues, subscribers to a topic can read messages using either ReceiveAndDelete or PeekLock. Unlike queues, however, a single message sent to a topic can be received by multiple subscriptions. This approach, commonly called *publish and subscribe* (or *pub/sub*), is useful whenever multiple applications are interested in the same messages. By defining the right filter, each subscriber can tap into just the part of the message stream that it needs to see.

Relays

Both queues and topics provide one-way asynchronous communication through a broker. Traffic flows in just one direction, and there's no direct connection between senders and receivers. But what if you don't want this? Suppose your applications need to both send and receive messages, or perhaps you want a direct link between them and you don't need a broker to store messages. To address scenarios such as this, Service Bus provides *relays*, as Figure 4 shows.

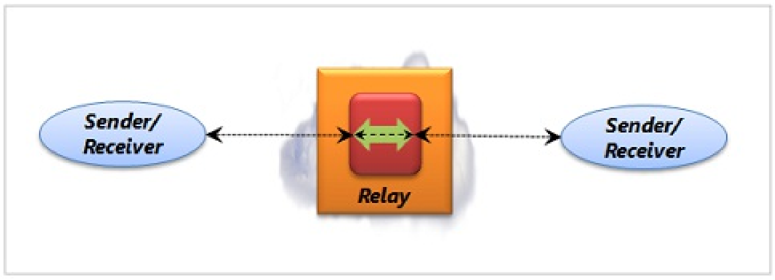


Figure 4: Service Bus relay provides synchronous, two-way communication between applications.

The obvious question to ask about relays is this: why would I use one? Even if I don't need queues, why make applications communicate via a cloud service rather than just interact directly? The answer is that talking directly can be harder than you might think.

Suppose you want to connect two on-premises applications, both running inside corporate datacenters. Each of these applications sits behind a firewall, and each datacenter probably uses network address translation (NAT). The firewall blocks incoming data on all but a few ports, and NAT implies that the machine each application is running on doesn't have a fixed IP address that you can reach directly from outside the datacenter. Without some extra help, connecting these applications over the public internet is problematic.

A Service Bus relay can help. To communicate bi-directionally through a relay, each application establishes an outbound TCP connection with Service Bus, then keeps it open. All communication between the two applications travels over these connections. Because each connection was established from inside the datacenter, the firewall allows incoming traffic to each application without opening new ports. This approach also gets around the NAT problem, because each application has a consistent endpoint in the cloud throughout the communication. By exchanging data through the relay, the applications can avoid the problems that would otherwise make communication difficult.

To use Service Bus relays, applications rely on the Windows Communication Foundation (WCF). Service Bus provides WCF bindings that make it straightforward for Windows applications to interact via relays. Applications that already use WCF can typically just specify one of these bindings, then talk to each other through a relay. Unlike queues and topics, however, using relays from non-Windows applications, while possible, requires some programming effort; no standard libraries are provided.

Unlike queues and topics, applications don't explicitly create relays. Instead, when an application that wishes to receive messages establishes a TCP connection with Service Bus, a relay is created automatically. When the connection is dropped, the relay is deleted. To enable an application to find the relay created by a specific listener, Service Bus provides a registry that enables applications to locate a specific relay by name.

Relays are the right solution when you need direct communication between applications. For example, consider an airline reservation system running in an on-premises datacenter that must be accessed from check-in kiosks, mobile devices, and other computers. Applications running on all of these systems could rely on Service Bus relays in the cloud to communicate, wherever they might be running.

Summary

Connecting applications has always been part of building complete solutions, and the range of scenarios that require applications and services to communicate with each other is set to increase as more applications and devices are connected to the Internet. By providing cloud-based technologies for achieving this through queues, topics, and relays, Service Bus aims to make this essential function easier to implement and more broadly available.

Next steps

Now that you've learned the fundamentals of Azure Service Bus, follow these links to learn more.

- How to use Service Bus queues
- How to use Service Bus topics
- How to use Service Bus relay
- Service Bus samples

Service Bus FAQ

2/9/2017 • 7 min to read • Edit on GitHub

This article answers some frequently-asked questions about Microsoft Azure Service Bus. You can also visit the Azure Support FAQ for general Azure pricing and support information.

General questions about Azure Service Bus

What is Azure Service Bus?

Azure Service Bus is an asynchronous messaging cloud platform that enables you to send data between decoupled systems. Microsoft offers this feature as a service, which means that you do not need to host any of your own hardware in order to use it.

What is a Service Bus namespace?

A namespace provides a scoping container for addressing Service Bus resources within your application. Creating one is necessary to use Service Bus and will be one of the first steps in getting started.

What is an Azure Service Bus queue?

A Service Bus queue is an entity in which messages are stored. Queues are particularly useful when you have multiple applications, or multiple parts of a distributed application that need to communicate with each other. The queue is similar to a distribution center in that multiple products (messages) are received and then sent from that location.

What are Azure Service Bus topics and subscriptions?

A topic can be visualized as a queue and when using multiple subscriptions, it becomes a richer messaging model; essentially a one-to-many communication tool. This publish/subscribe model (or *pub/sub*) enables an application that sends a message to a topic with multiple subscriptions to have that message received by multiple applications.

What is a partitioned entity?

A conventional queue or topic is handled by a single message broker and stored in one messaging store. A partitioned queue or topic is handled by multiple message brokers and stored in multiple messaging stores. This means that the overall throughput of a partitioned queue or topic is no longer limited by the performance of a single message broker or messaging store. In addition, a temporary outage of a messaging store does not render a partitioned queue or topic unavailable.

Note that ordering is not ensured when using partitioning entities. In the event that a partition is unavailable, you can still send and receive messages from the other partitions.

Best practices

What are some Azure Service Bus best practices?

- Best practices for performance improvements using Service Bus – this article describes how to optimize performance when exchanging messages.

What should I know before creating entities?

The following properties of a queue and topic are immutable. Please take this into account when you provision your entities as this cannot be modified, without creating a new replacement entity.

- Size
- Partitioning

- Sessions
- Duplicate detection
- Express entity

Pricing

This section answers some frequently-asked questions about the Service Bus pricing structure. You can also visit the Azure Support FAQ for general Microsoft Azure pricing information. For complete information about Service Bus pricing, see Service Bus pricing details.

How do you charge for Service Bus?

For complete information about Service Bus pricing, please see Service Bus pricing details. In addition to the prices noted, you are charged for associated data transfers for egress outside of the data center in which your application is provisioned.

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

Any data transfer within a given Azure region is provided at no charge, as well as any inbound data transfer. Data transfer outside a region is subject to egress charges which can be found here.

Does Service Bus charge for storage?

No, 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 next FAQ.

Quotas

For a list of Service Bus limits and quotas, see Quotas overview.

Does 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 Service Bus, the aggregate usage quotas is 5 billion messages per month.

While we do reserve the right to disable a customer 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 Azure, 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.

Naming restrictions

A Service Bus namespace name can only be between 6-50 characters in length. The character count limit for each queue, topic, or subscription is between 1-50 characters.

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, 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.

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 Azure classic 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.

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 Azure classic 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.

Troubleshooting

What are some of the exceptions generated by Azure Service Bus APIs and their suggested actions?

For a list of possible Service Bus exceptions, see Exceptions overview.

What is a Shared Access Signature and which languages support generating a signature?

Shared Access Signatures are an authentication mechanism based on SHA – 256 secure hashes or URIs. For information about how to generate your own signatures in Node, PHP, Java and C#, see the Shared Access Signatures article.

Subscription and namespace management

How do I migrate a namespace to another Azure subscription?

Using the Azure portal, you can migrate Service Bus namespaces to another subscription by following the directions here. If you prefer to use PowerShell, follow the instructions below:

The following sequence of commands moves a namespace from one Azure subscription to another. To execute this operation, the namespace must already be active, and the user running the PowerShell commands must be an administrator on both the source and target subscriptions.

```
# Create a new resource group in target subscription
Select-AzureRmSubscription -SubscriptionId 'ffffffff-ffff-ffff-ffff-ffffffffffff'
New-AzureRmResourceGroup -Name 'targetRG' -Location 'East US'

# Move namespace from source subscription to target subscription
Select-AzureRmSubscription -SubscriptionId 'aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa'
$res = Find-AzureRmResource -ResourceNameContains mynamespace -ResourceType 'Microsoft.ServiceBus/namespaces'
Move-AzureRmResource -DestinationResourceGroupName 'targetRG' -DestinationSubscriptionId 'ffffffff-ffff-ffff-
ffff-ffffffffffff' -ResourceId $res.ResourceId
```

Next steps

To learn more about Service Bus, see the following topics.

- Introducing Azure Service Bus Premium (blog post)
- Introducing Azure Service Bus Premium (Channel9)
- Service Bus overview
- Azure Service Bus architecture overview
- Get started with Service Bus queues

Create a Service Bus namespace using the Azure portal

2/27/2017 • 1 min to read • Edit on GitHub

A namespace is a common container for all messaging components. Multiple queues and topics can reside in a single namespace, and namespaces often serve as application containers. There are currently two different ways to create a Service Bus namespace.

1. Azure portal (this article)
2. Resource Manager templates

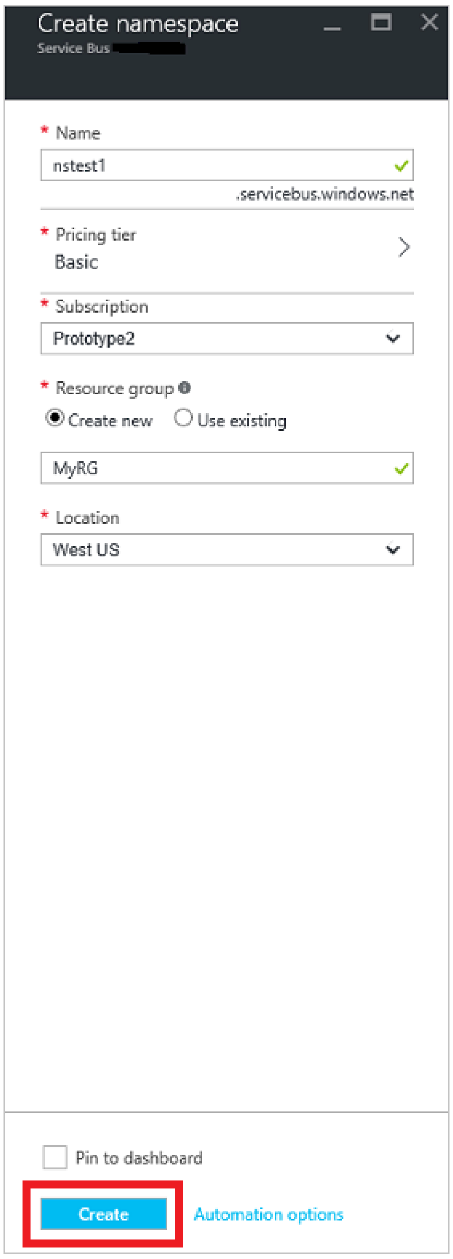
Create a namespace in the Azure portal

Create a service namespace

To begin using Service Bus queues in Azure, you must first create a namespace. A namespace provides a scoping container for addressing Service Bus resources within your application.

To create a namespace:

1. Log on to the Azure portal.
2. In the left navigation pane of the portal, click **New**, then click **Enterprise Integration**, and then click **Service Bus**.
3. In the **Create namespace** dialog, enter a namespace name. The system immediately checks to see if the name is available.
4. After making sure the namespace name is available, choose the pricing tier (Basic, Standard, or Premium).
5. In the **Subscription** field, choose an Azure subscription in which to create the namespace.
6. In the **Resource group** field, choose an existing resource group in which the namespace will live, or create a new one.
7. In **Location**, choose the country or region in which your namespace should be hosted.


Create namespace

Service Bus [redacted]

* Name

nstest1

.servicebus.windows.net

* Pricing tier

Basic

* Subscription

Prototype2

* Resource group

[✓ Create new] Use existing

MyRG

* Location

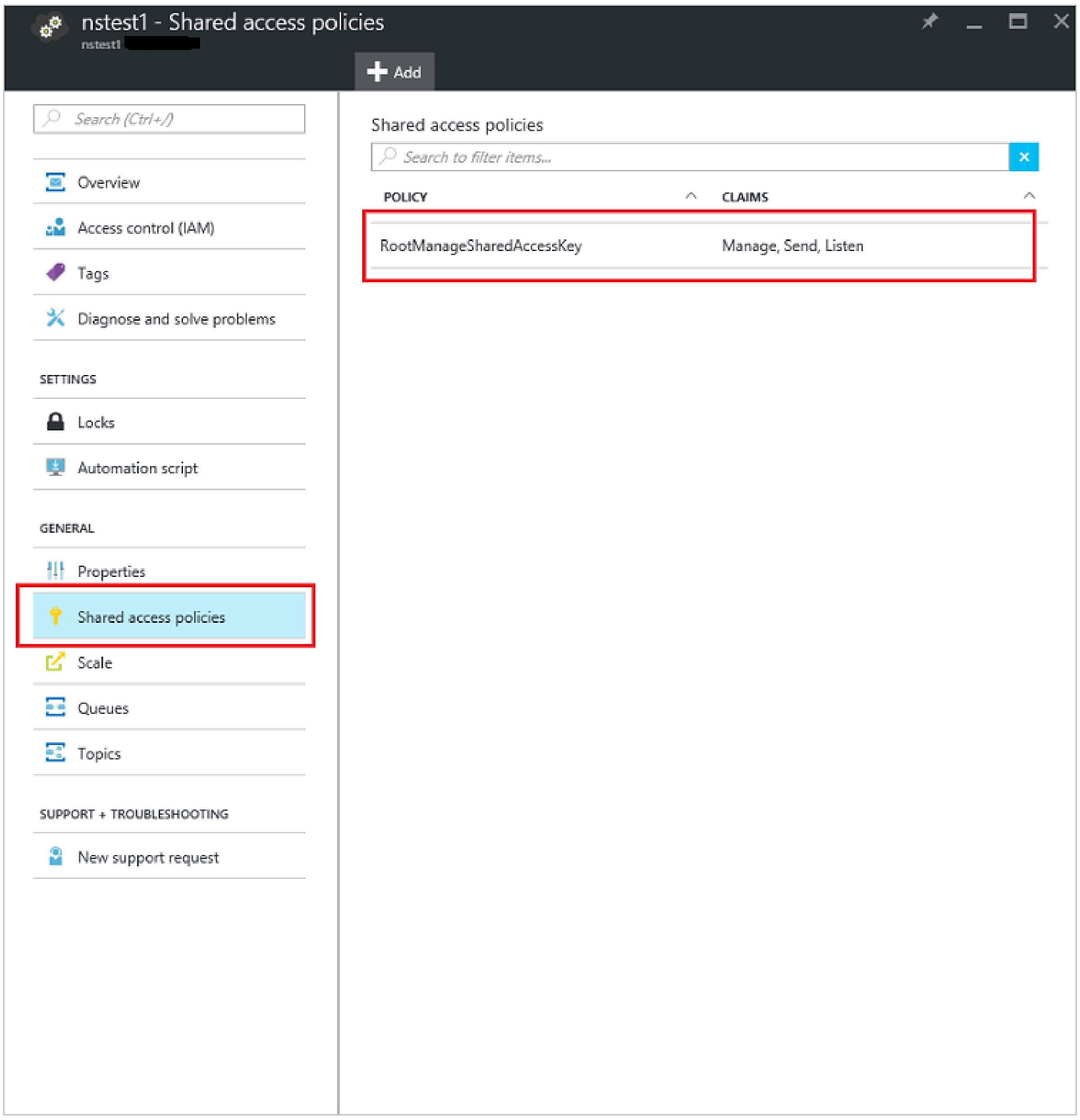
West US

Pin to dashboard

8. Click **Create**. The system now creates your namespace and enables it. You might have to wait several minutes as the system provisions resources for your account.

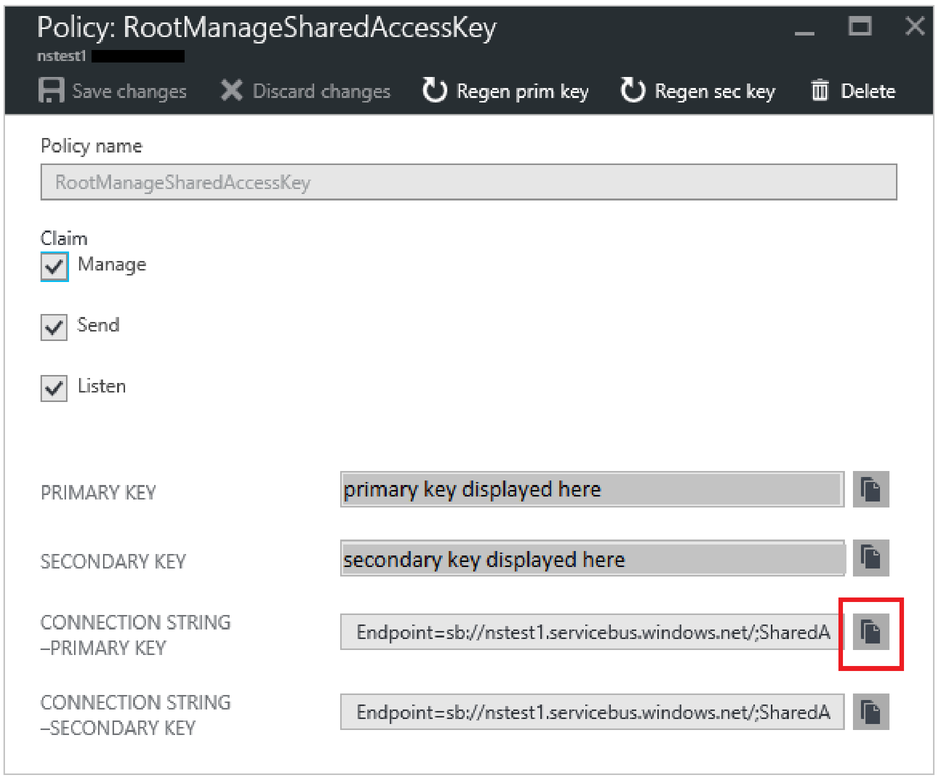
Obtain the management credentials

1. In the list of namespaces, click the newly created namespace name.
2. In the namespace blade, click **Shared access policies**.
3. In the **Shared access policies** blade, click **RootManageSharedAccessKey**.



POLICY	CLAIMS
RootManageSharedAccessKey	Manage, Send, Listen

4. In the **Policy: RootManageSharedAccessKey** blade, click the copy button next to **Connection string–primary key**, to copy the connection string to your clipboard for later use. Paste this value into Notepad or some other temporary location.


Policy: RootManageSharedAccessKey

nstest1 [redacted]

Policy name: RootManageSharedAccessKey

Claim
- ✓ Manage
- ✓ Send
- ✓ Listen

PRIMARY KEY	primary key displayed here
SECONDARY KEY	secondary key displayed here
CONNECTION STRING –PRIMARY KEY	Endpoint=sb://nstest1.servicebus.windows.net/;SharedA
CONNECTION STRING –SECONDARY KEY	Endpoint=sb://nstest1.servicebus.windows.net/;SharedA

5. Repeat the previous step, copying and pasting the value of **Primary key** to a temporary location for later use.

Congratulations! You have now created a Service Bus Messaging namespace.

Next steps

Check out our GitHub samples which show some of the more advanced features of Azure Service Bus Messaging.

Get started with Service Bus queues

2/28/2017 • 4 min to read • Edit on GitHub

What will be accomplished

In this tutorial, we will complete the following:

1. Create a Service Bus namespace, using the Azure portal.
2. Create a Service Bus Messaging queue, using the Azure portal.
3. Write a console application to send a message.
4. Write a console application to receive messages.

Prerequisites

1. Visual Studio 2015 or higher. The examples in this tutorial use Visual Studio 2015.
2. An Azure subscription.

NOTE

To complete this tutorial, you need an Azure account. You can activate your MSDN subscriber benefits or sign up for a free account.

1. Create a namespace using the Azure portal

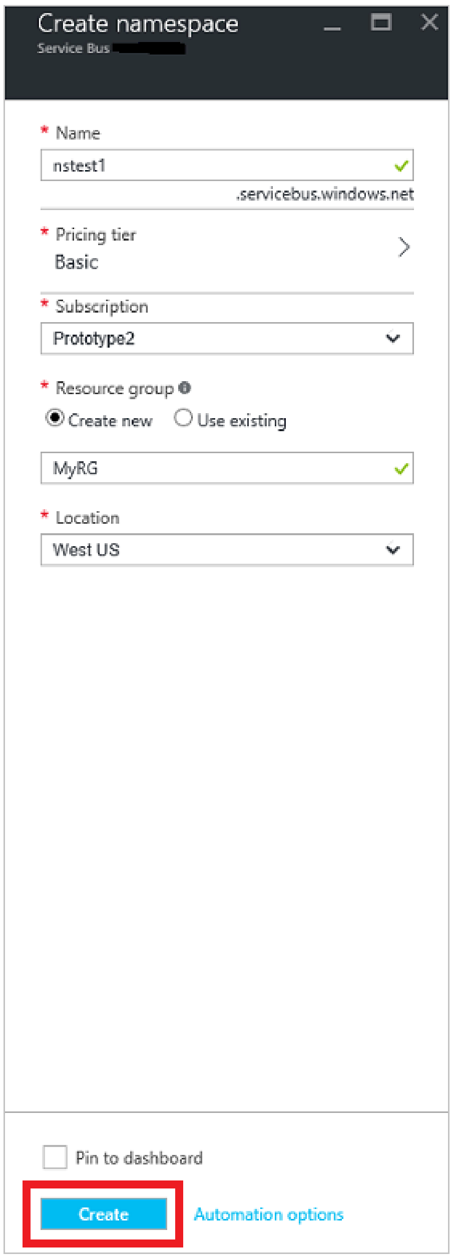
If you already have a Service Bus namespace created, jump to the Create a queue using the Azure portal section.

Create a service namespace

To begin using Service Bus queues in Azure, you must first create a namespace. A namespace provides a scoping container for addressing Service Bus resources within your application.

To create a namespace:

1. Log on to the Azure portal.
2. In the left navigation pane of the portal, click **New**, then click **Enterprise Integration**, and then click **Service Bus**.
3. In the **Create namespace** dialog, enter a namespace name. The system immediately checks to see if the name is available.
4. After making sure the namespace name is available, choose the pricing tier (Basic, Standard, or Premium).
5. In the **Subscription** field, choose an Azure subscription in which to create the namespace.
6. In the **Resource group** field, choose an existing resource group in which the namespace will live, or create a new one.
7. In **Location**, choose the country or region in which your namespace should be hosted.


Create namespace
Service Bus [redacted]

* Name
nstest1 ✓
.servicebus.windows.net

* Pricing tier
Basic

* Subscription
Prototype2

* Resource group
(●) Create new ( ) Use existing
MyRG ✓

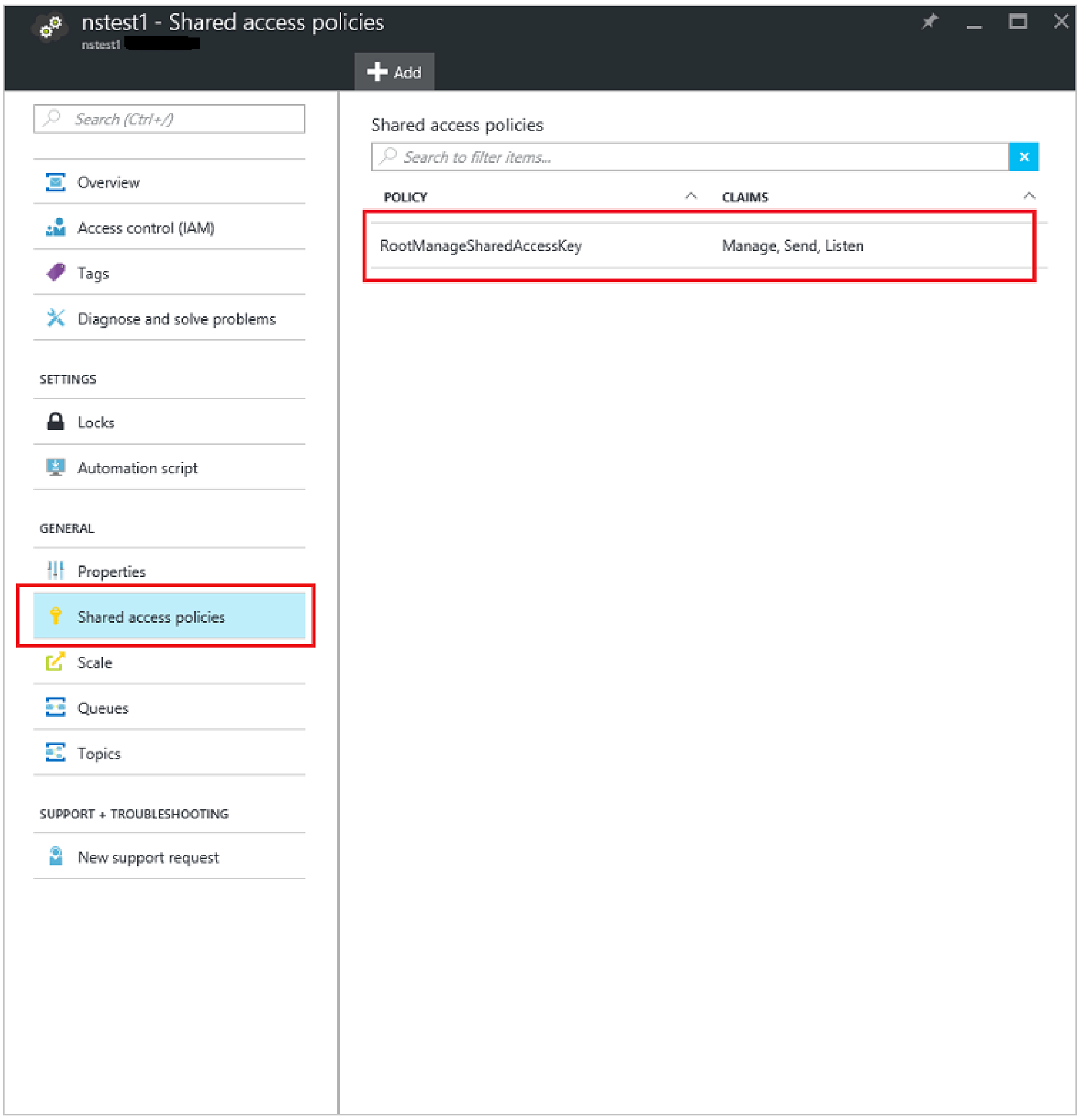
* Location
West US

☐ Pin to dashboard

8. Click **Create**. The system now creates your namespace and enables it. You might have to wait several minutes as the system provisions resources for your account.

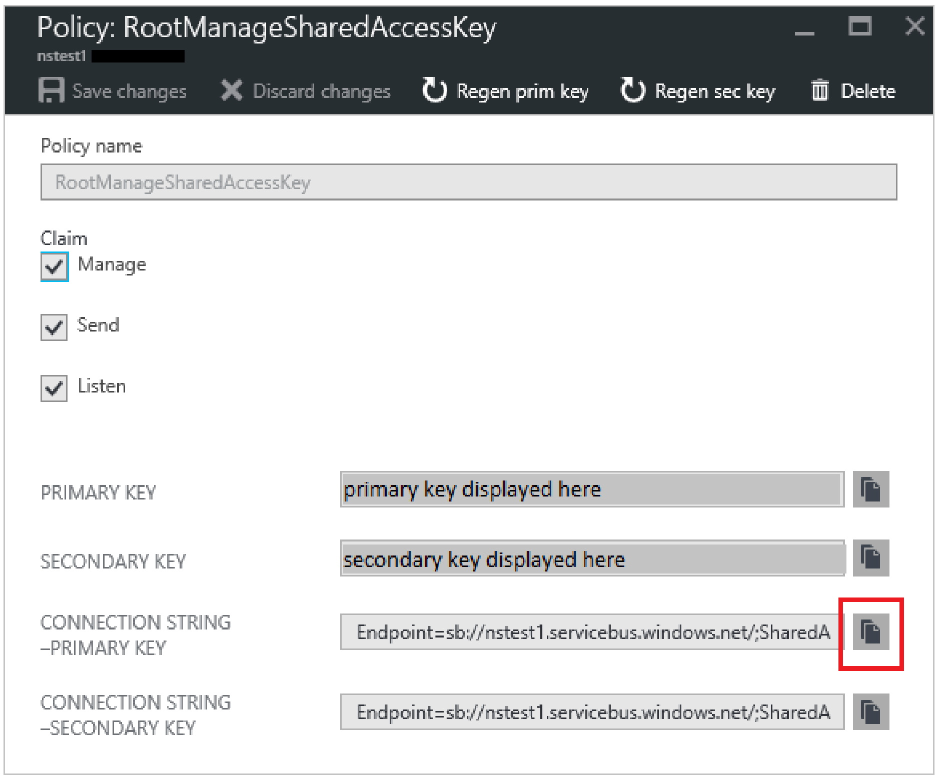
Obtain the management credentials

1. In the list of namespaces, click the newly created namespace name.
2. In the namespace blade, click **Shared access policies**.
3. In the **Shared access policies** blade, click **RootManageSharedAccessKey**.



POLICY	CLAIMS
RootManageSharedAccessKey	Manage, Send, Listen

4. In the **Policy: RootManageSharedAccessKey** blade, click the copy button next to **Connection string–primary key**, to copy the connection string to your clipboard for later use. Paste this value into Notepad or some other temporary location.



Policy: RootManageSharedAccessKey

nstest1 [redacted]

Save changes | Discard changes | Regen prim key | Regen sec key | Delete

Policy name

RootManageSharedAccessKey

Claim

✓ Manage

✓ Send

✓ Listen

PRIMARY KEY	primary key displayed here
SECONDARY KEY	secondary key displayed here
CONNECTION STRING –PRIMARY KEY	Endpoint=sb://nstest1.servicebus.windows.net/;SharedA
CONNECTION STRING –SECONDARY KEY	Endpoint=sb://nstest1.servicebus.windows.net/;SharedA

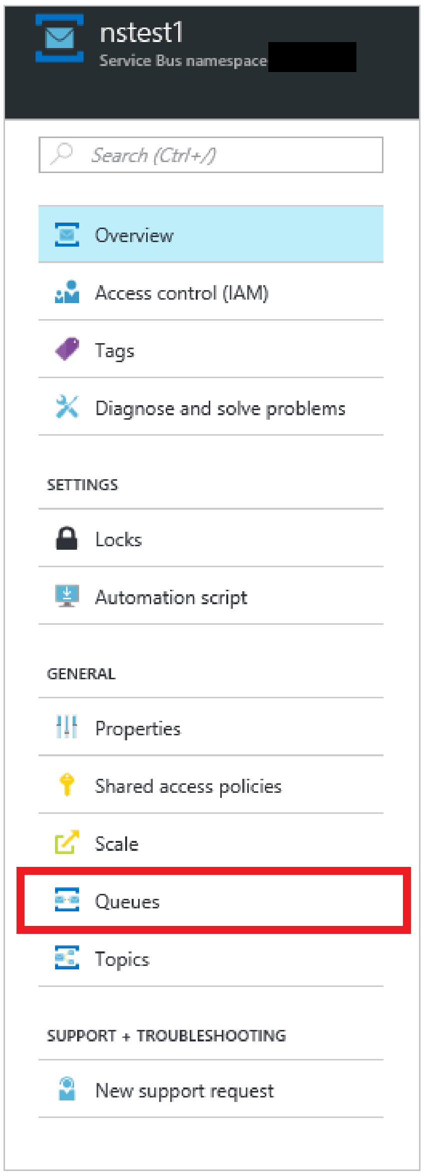
5. Repeat the previous step, copying and pasting the value of **Primary key** to a temporary location for later use.

2. Create a queue using the Azure portal

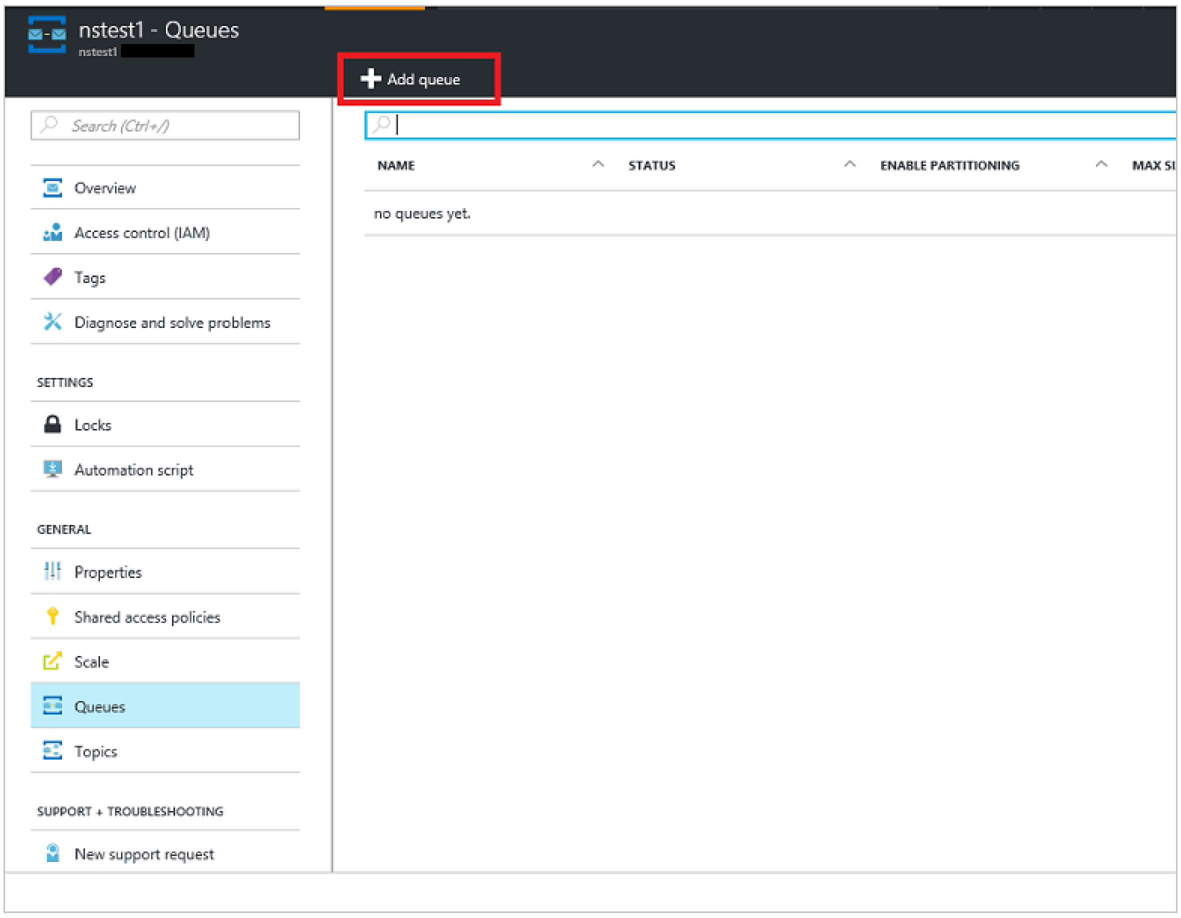
If you already have a Service Bus queue created, jump to the Send messages to the queue section.

Please ensure that you have already created a Service Bus namespace, as shown here.

1. Log on to the Azure portal.
2. In the left navigation pane of the portal, click **Service Bus** (if you don't see **Service Bus**, click **More services**).
3. Select the namespace that you would like to create the queue in. In this case, it is **nstest1**.

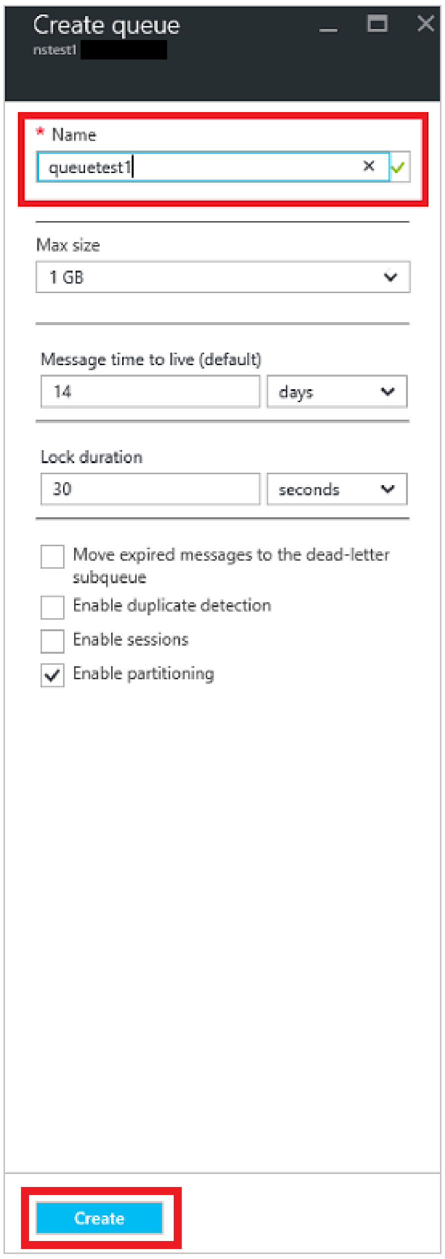


4. In the **Service Bus namespace** blade, select **Queues**, then click **Add queue**.



NAME	STATUS	ENABLE PARTITIONING	MAX SI
no queues yet.			

5. Enter the **Queue Name** and leave the other values with their defaults.



Name: queuetest1

Max size: 1 GB

Message time to live (default): 14 days

Lock duration: 30 seconds

- [ ] Move expired messages to the dead-letter subqueue
- [ ] Enable duplicate detection
- [ ] Enable sessions
- [✓] Enable partitioning

6. At the bottom of the blade, click **Create**.

3. Send messages to the queue

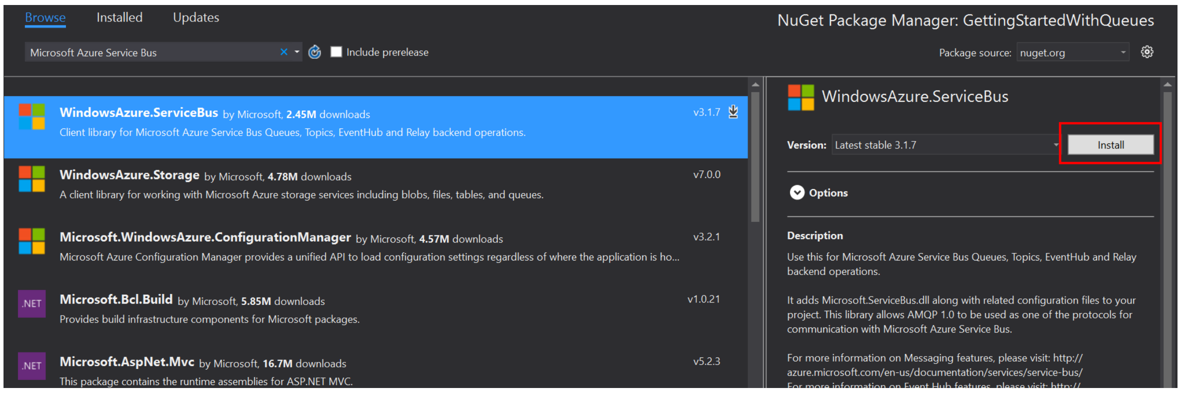
To send messages to the queue, we will write a C# console application using Visual Studio.

Create a console application

- Launch Visual Studio and create a new Console application.

Add the Service Bus NuGet package

1. Right-click the newly created project and select **Manage NuGet Packages**.
2. Click the **Browse** tab, then search for "Microsoft Azure Service Bus" and select the **Microsoft Azure Service Bus** item. Click **Install** to complete the installation, then close this dialog box.



NuGet Package Manager: GettingStartedWithQueues

Microsoft Azure Service Bus

Package source: nuget.org

WindowsAzure.ServiceBus by Microsoft, **2.45M** downloads v3.1.7
Client library for Microsoft Azure Service Bus Queues, Topics, EventHub and Relay backend operations.

WindowsAzure.Storage by Microsoft, **4.78M** downloads v7.0.0
A client library for working with Microsoft Azure storage services including blobs, files, tables, and queues.

Microsoft.WindowsAzure.ConfigurationManager by Microsoft, **4.57M** downloads v3.2.1
Microsoft Azure Configuration Manager provides a unified API to load configuration settings regardless of where the application is ho...

Microsoft.Bcl.Build by Microsoft, **5.85M** downloads v1.0.21
Provides build infrastructure components for Microsoft packages.

Microsoft.AspNet.Mvc by Microsoft, **16.7M** downloads v5.2.3
This package contains the runtime assemblies for ASP.NET MVC.

WindowsAzure.ServiceBus

Version: Latest stable 3.1.7

Description

Use this for Microsoft Azure Service Bus Queues, Topics, EventHub and Relay backend operations.

It adds Microsoft.ServiceBus.dll along with related configuration files to your project. This library allows AMQP 1.0 to be used as one of the protocols for communication with Microsoft Azure Service Bus.

For more information on Messaging features, please visit: http://azure.microsoft.com/en-us/documentation/services/service-bus/

Write some code to send a message to the queue

1. Add the following using statement to the top of the Program.cs file.

```
using Microsoft.ServiceBus.Messaging;
```

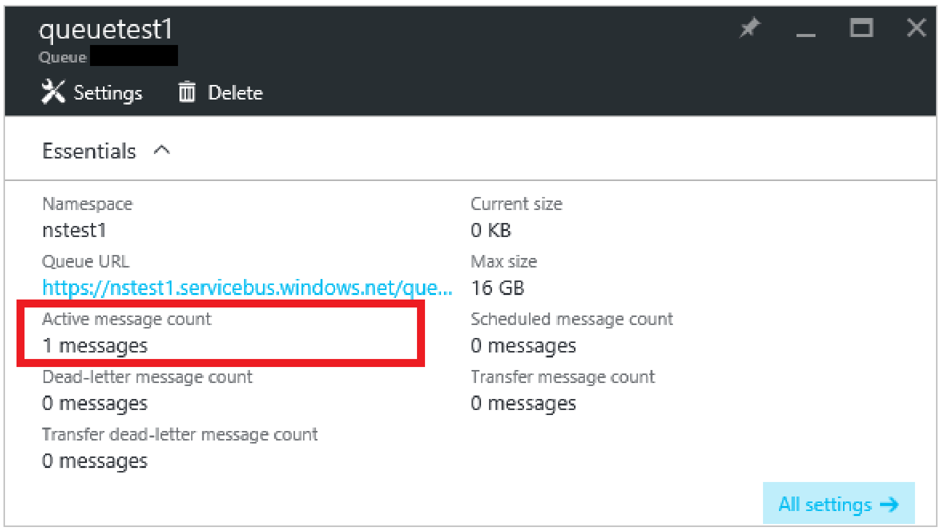
2. Add the following code to the `Main` method, set the **connectionString** variable as the connection string that was obtained when creating the namespace, and set **queueName** as the queue name that used when creating the queue.

```
var connectionString = "<Your connection string>";
var queueName = "<Your queue name>";

var client = QueueClient.CreateFromConnectionString(connectionString, queueName);
var message = new BrokeredMessage("This is a test message!");
client.Send(message);
```

Here is what your Program.cs should look like.

```
using System;
using Microsoft.ServiceBus.Messaging;

namespace GettingStartedWithQueues
{
    class Program
    {
        static void Main(string[] args)
        {
            var connectionString = "<Your connection string>";
            var queueName = "<Your queue name>";

            var client = QueueClient.CreateFromConnectionString(connectionString, queueName);
            var message = new BrokeredMessage("This is a test message!");

            client.Send(message);
        }
    }
}
```

3. Run the program, and check the Azure portal. Click the name of your queue in the namespace **Overview** blade. Notice that the **Active message count** value should now be 1.


queuetest1

Queue [redacted]

Settings Delete

Essentials

Namespace
nstest1

Queue URL
https://nstest1.servicebus.windows.net/que...

Active message count
1 messages

Dead-letter message count
0 messages

Transfer dead-letter message count
0 messages

Current size
0 KB

Max size
16 GB

Scheduled message count
0 messages

Transfer message count
0 messages

All settings →

4. Receive messages from the queue

1. Create a new console application and add a reference to the Service Bus NuGet package, similar to the previous sending application.
2. Add the following `using` statement to the top of the Program.cs file.

```
using Microsoft.ServiceBus.Messaging;
```

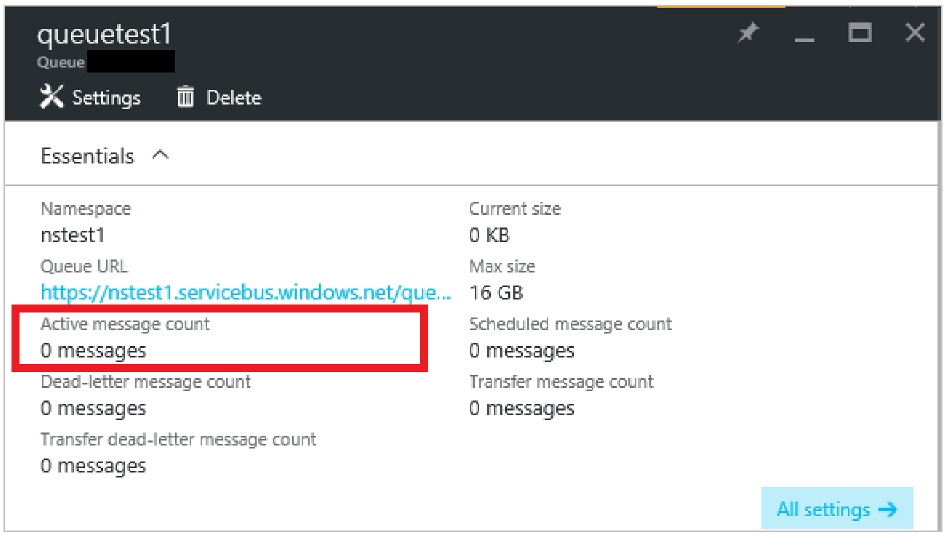
3. Add the following code to the `Main` method, set the **connectionString** variable as the connection string that was obtained when creating the namespace, and set **queueName** as the queue name that you used when creating the queue.

```
var connectionString = "";
var queueName = "samplequeue";

var client = QueueClient.CreateFromConnectionString(connectionString, queueName);

client.OnMessage(message =>
{
  Console.WriteLine(String.Format("Message body: {0}", message.GetBody<String>()));
  Console.WriteLine(String.Format("Message id: {0}", message.MessageId));
});

Console.ReadLine();
```

Here is what your Program.cs file should look like:

```
using System;
using Microsoft.ServiceBus.Messaging;

namespace GettingStartedWithQueues
{
  class Program
  {
    static void Main(string[] args)
    {
      var connectionString = "";
      var queueName = "samplequeue";

      var client = QueueClient.CreateFromConnectionString(connectionString, queueName);

      client.OnMessage(message =>
      {
        Console.WriteLine(String.Format("Message body: {0}", message.GetBody<String>()));
        Console.WriteLine(String.Format("Message id: {0}", message.MessageId));
      });

      Console.ReadLine();
    }
  }
}
```

4. Run the program, and check the portal. Notice that the **Queue Length** value should now be 0.



Namespace
nstest1
Queue URL
https://nstest1.servicebus.windows.net/que...
Active message count
0 messages
Dead-letter message count
0 messages
Transfer dead-letter message count
0 messages
Current size
0 KB
Max size
16 GB
Scheduled message count
0 messages
Transfer message count
0 messages

Congratulations! You have now created a queue, sent a message, and received a message.

Next steps

Check out our GitHub repository with samples that demonstrate some of the more advanced features of Azure Service Bus Messaging.

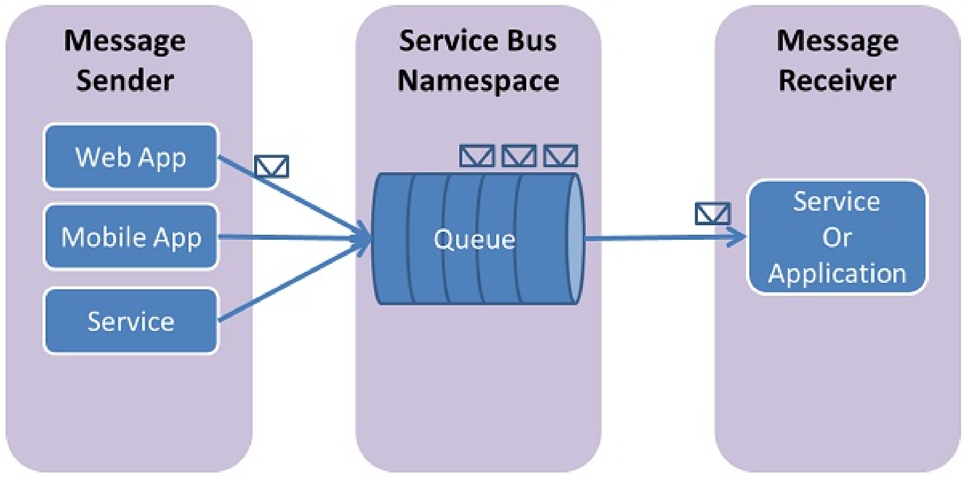
How to use Service Bus queues

3/6/2017 • 8 min to read • Edit on GitHub

This article describes how to use Service Bus queues. The samples are written in Java and use the Azure SDK for Java. The scenarios covered include **creating queues**, **sending and receiving messages**, and **deleting queues**.

What are Service Bus queues?

Service Bus queues support a **brokered messaging** communication model. When using queues, components of a distributed application do not communicate directly with each other; instead they exchange messages via a queue, which acts as an intermediary (broker). A message producer (sender) hands off a message to the queue and then continues its processing. Asynchronously, a message consumer (receiver) pulls the message from the queue and processes it. The producer does not have to wait for a reply from the consumer in order to continue to process and send further messages. Queues offer **First In, First Out (FIFO)** message delivery to one or more competing consumers. That is, messages are typically received and processed by the receivers in the order in which they were added to the queue, and each message is received and processed by only one message consumer.



Service Bus queues are a general-purpose technology that can be used for a wide variety of scenarios:

- Communication between web and worker roles in a multi-tier Azure application.
- Communication between on-premises apps and Azure-hosted apps in a hybrid solution.
- Communication between components of a distributed application running on-premises in different organizations or departments of an organization.

Using queues enables you to scale your applications more easily, and enable more resiliency to your architecture.

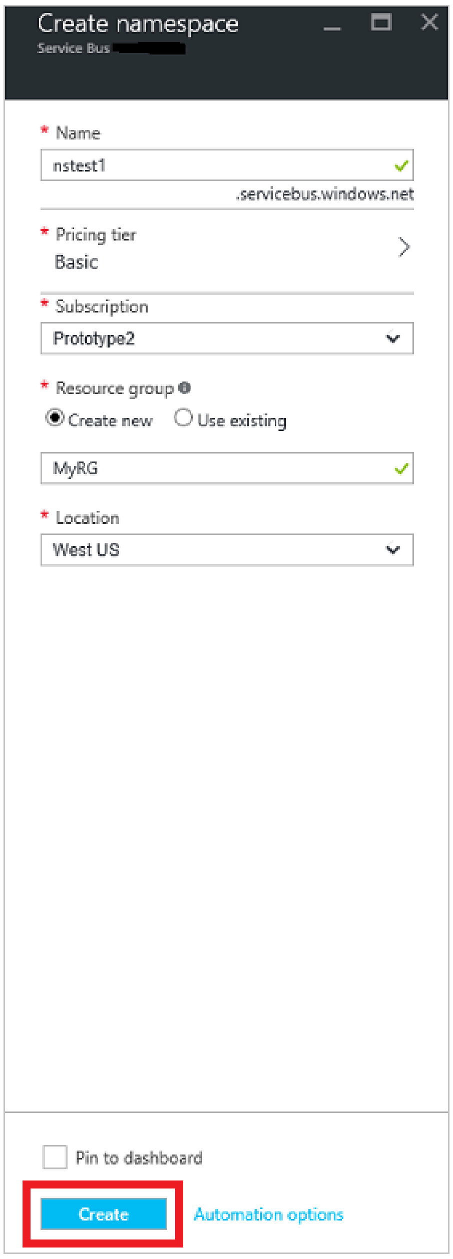
Create a service namespace

To begin using Service Bus queues in Azure, you must first create a namespace. A namespace provides a scoping container for addressing Service Bus resources within your application.

To create a namespace:

1. Log on to the Azure portal.
2. In the left navigation pane of the portal, click **New**, then click **Enterprise Integration**, and then click **Service Bus**.
3. In the **Create namespace** dialog, enter a namespace name. The system immediately checks to see if the name is available.

4. After making sure the namespace name is available, choose the pricing tier (Basic, Standard, or Premium).
5. In the **Subscription** field, choose an Azure subscription in which to create the namespace.
6. In the **Resource group** field, choose an existing resource group in which the namespace will live, or create a new one.
7. In **Location**, choose the country or region in which your namespace should be hosted.


Create namespace
Service Bus [redacted]

* Name
nstest1 ✓
.servicebus.windows.net

* Pricing tier
Basic

* Subscription
Prototype2

* Resource group
● Create new ○ Use existing
MyRG ✓

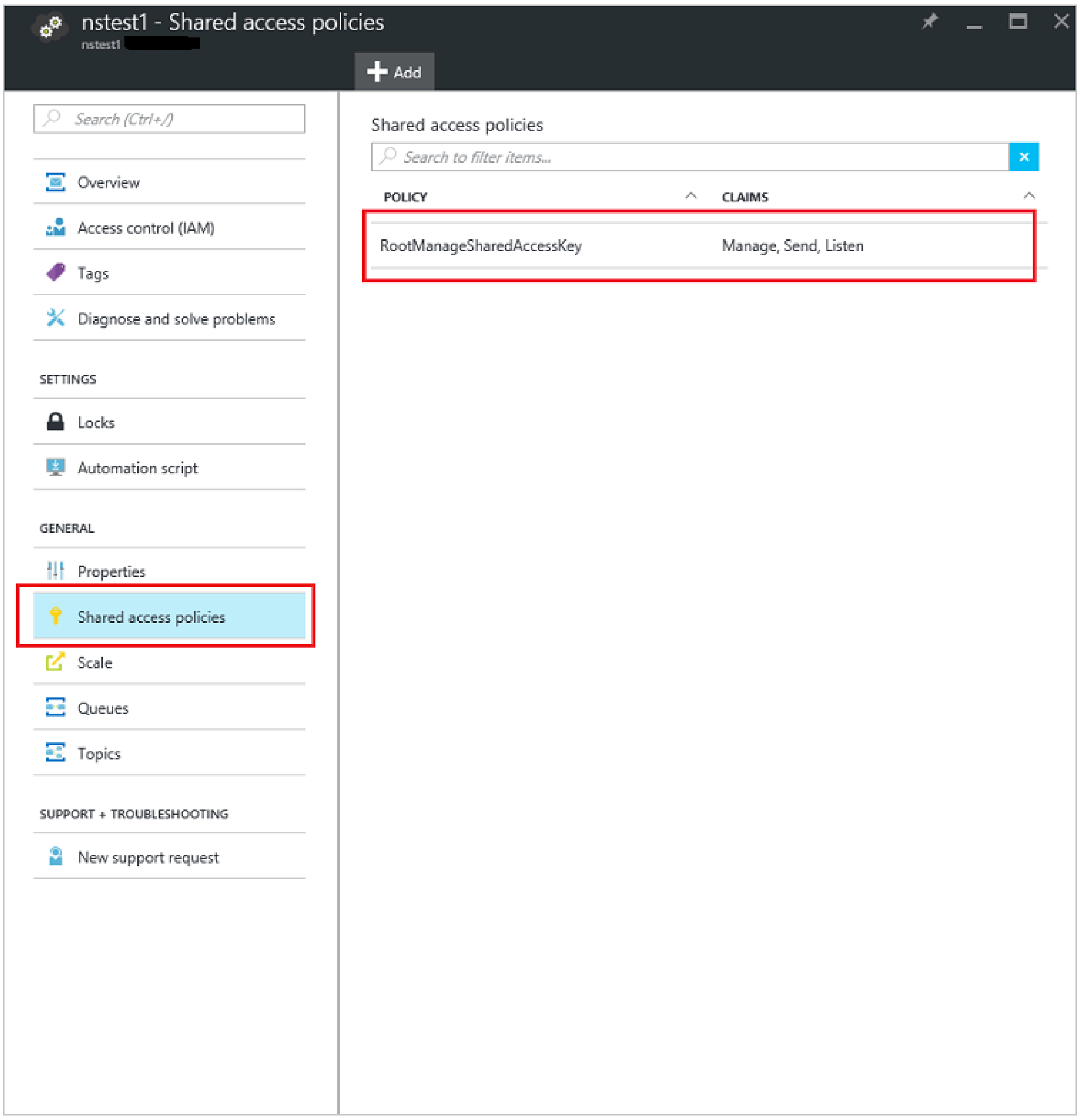
* Location
West US

☐ Pin to dashboard

8. Click **Create**. The system now creates your namespace and enables it. You might have to wait several minutes as the system provisions resources for your account.

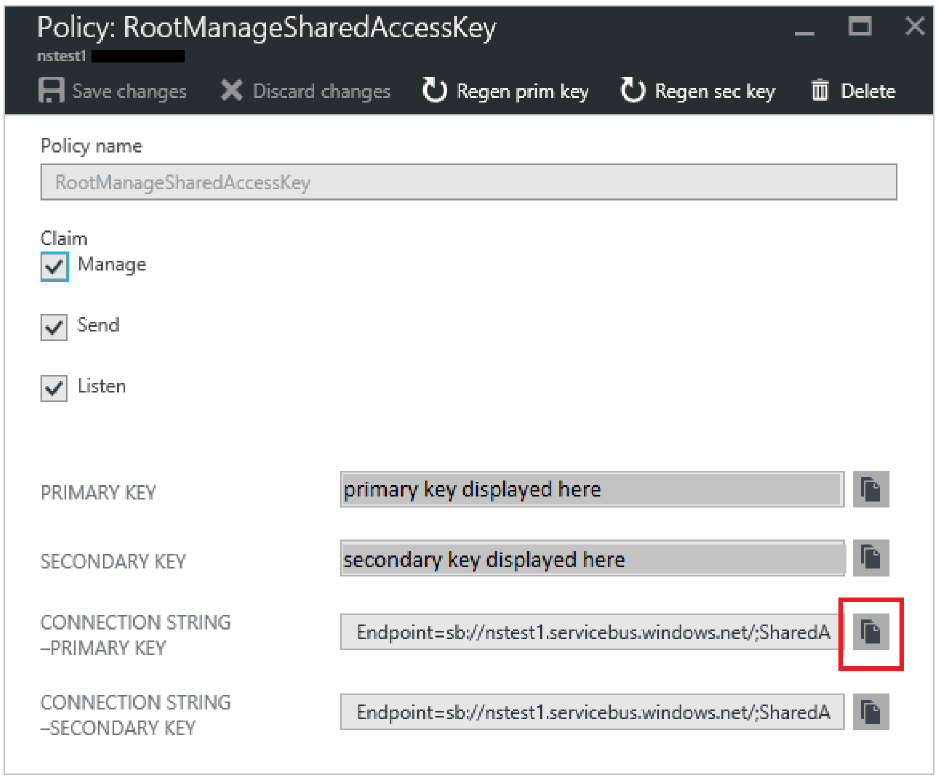
Obtain the management credentials

1. In the list of namespaces, click the newly created namespace name.
2. In the namespace blade, click **Shared access policies**.
3. In the **Shared access policies** blade, click **RootManageSharedAccessKey**.



POLICY	CLAIMS
RootManageSharedAccessKey	Manage, Send, Listen

4. In the **Policy: RootManageSharedAccessKey** blade, click the copy button next to **Connection string–primary key**, to copy the connection string to your clipboard for later use. Paste this value into Notepad or some other temporary location.


Policy: RootManageSharedAccessKey

nstest1 [redacted]

Save changes | Discard changes | Regen prim key | Regen sec key | Delete

Policy name: RootManageSharedAccessKey

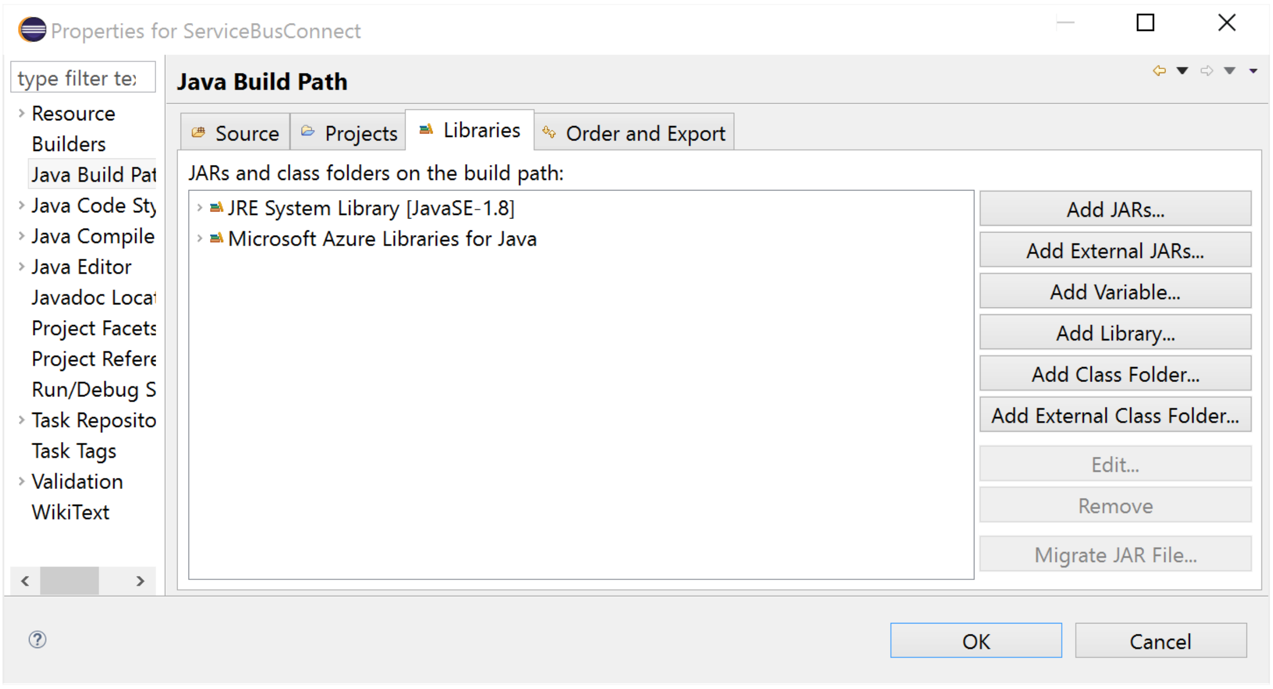
Claim
- [✓] Manage
- [✓] Send
- [✓] Listen

PRIMARY KEY	primary key displayed here
SECONDARY KEY	secondary key displayed here
CONNECTION STRING –PRIMARY KEY	Endpoint=sb://nstest1.servicebus.windows.net/;SharedA
CONNECTION STRING –SECONDARY KEY	Endpoint=sb://nstest1.servicebus.windows.net/;SharedA

5. Repeat the previous step, copying and pasting the value of **Primary key** to a temporary location for later use.

Configure your application to use Service Bus

Make sure you have installed the Azure SDK for Java before building this sample. If you are using Eclipse, you can install the Azure Toolkit for Eclipse that includes the Azure SDK for Java. You can then add the **Microsoft Azure Libraries for Java** to your project:


Properties for ServiceBusConnect

Java Build Path

Tabs: Source | Projects | Libraries | Order and Export

JARs and class folders on the build path:
- JRE System Library [JavaSE-1.8]
- Microsoft Azure Libraries for Java

Add the following `import` statements to the top of the Java file:

```
// Include the following imports to use Service Bus APIs
import com.microsoft.windowsazure.services.servicebus.*;
import com.microsoft.windowsazure.services.servicebus.models.*;
import com.microsoft.windowsazure.core.*;
import javax.xml.datatype.*;
```

Create a queue

Management operations for Service Bus queues can be performed via the **ServiceBusContract** class. A **ServiceBusContract** object is constructed with an appropriate configuration that encapsulates the SAS token with permissions to manage it, and the **ServiceBusContract** class is the sole point of communication with Azure.

The **ServiceBusService** class provides methods to create, enumerate, and delete queues. The example below shows how a **ServiceBusService** object can be used to create a queue named "TestQueue", with a namespace named "HowToSample":

```
Configuration config =
    ServiceBusConfiguration.configureWithSASAuthentication(
            "HowToSample",
            "RootManageSharedAccessKey",
            "SAS_key_value",
            ".servicebus.windows.net"
            );

ServiceBusContract service = ServiceBusService.create(config);
QueueInfo queueInfo = new QueueInfo("TestQueue");
try
{
    CreateQueueResult result = service.createQueue(queueInfo);
}
catch (ServiceException e)
{
    System.out.print("ServiceException encountered: ");
    System.out.println(e.getMessage());
    System.exit(-1);
}
```

There are methods on **QueueInfo** that allow properties of the queue to be tuned (for example: to set the default time-to-live (TTL) value to be applied to messages sent to the queue). The following example shows how to create a queue named `TestQueue` with a maximum size of 5GB:

```
long maxSizeInMegabytes = 5120;
QueueInfo queueInfo = new QueueInfo("TestQueue");
queueInfo.setMaxSizeInMegabytes(maxSizeInMegabytes);
CreateQueueResult result = service.createQueue(queueInfo);
```

Note that you can use the **listQueues** method on **ServiceBusContract** objects to check if a queue with a specified name already exists within a service namespace.

Send messages to a queue

To send a message to a Service Bus queue, your application obtains a **ServiceBusContract** object. The following code shows how to send a message for the `TestQueue` queue previously created in the `HowToSample` namespace:

```
try
{
    BrokeredMessage message = new BrokeredMessage("MyMessage");
    service.sendQueueMessage("TestQueue", message);
}
catch (ServiceException e)
{
    System.out.print("ServiceException encountered: ");
    System.out.println(e.getMessage());
    System.exit(-1);
}
```

Messages sent to, and received from Service Bus queues are instances of the BrokeredMessage class. BrokeredMessage objects have a set of standard properties (such as Label and TimeToLive), a dictionary that is used to hold custom application-specific properties, and a body of arbitrary application data. An application can set the body of the message by passing any serializable object into the constructor of the BrokeredMessage, and the appropriate serializer will then be used to serialize the object. Alternatively, you can provide a **java.IO.InputStream** object.

The following example demonstrates how to send five test messages to the `TestQueue` **MessageSender** we obtained in the previous code snippet:

```
for (int i=0; i<5; i++)
{
     // Create message, passing a string message for the body.
     BrokeredMessage message = new BrokeredMessage("Test message " + i);
     // Set an additional app-specific property.
     message.setProperty("MyProperty", i);
     // Send message to the queue
     service.sendQueueMessage("TestQueue", message);
}
```

Service Bus queues support a maximum message size of 256 KB in the Standard tier and 1 MB in the Premium tier. The header, which includes the standard and custom application properties, can have a maximum size of 64 KB. There is no limit on the number of messages held in a queue but there is a cap on the total size of the messages held by a queue. This queue size is defined at creation time, with an upper limit of 5 GB.

Receive messages from a queue

The primary way to receive messages from a queue is to use a **ServiceBusContract** object. Received messages can work in two different modes: **ReceiveAndDelete** and **PeekLock**.

When using the **ReceiveAndDelete** mode, receive is a single-shot operation - that is, when Service Bus receives a read request for a message in a queue, it marks the message as being consumed and returns it to the application. **ReceiveAndDelete** mode (which is the default mode) is the simplest model and works best for scenarios in which an 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. Because Service Bus will have marked the message as being consumed, then when the application restarts and begins consuming messages again, it will have missed the message that was consumed prior to the crash.

In **PeekLock** mode, receive becomes a two stage operation, which makes it possible to support applications that cannot tolerate missing messages. When Service Bus receives a request, it finds the next message to be consumed, locks it to prevent other consumers 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 **Delete** on the received message. When Service Bus sees the **Delete** call, it will mark the message as being consumed and remove it from the queue.

The following example demonstrates how messages can be received and processed using **PeekLock** mode (not the default mode). The example below does an infinite loop and processes messages as they arrive into our "TestQueue":

```
try
{
    ReceiveMessageOptions opts = ReceiveMessageOptions.DEFAULT;
    opts.setReceiveMode(ReceiveMode.PEEK_LOCK);

    while(true)  {
         ReceiveQueueMessageResult resultQM =
                 service.receiveQueueMessage("TestQueue", opts);
        BrokeredMessage message = resultQM.getValue();
        if (message != null && message.getMessageId() != null)
        {
            System.out.println("MessageID: " + message.getMessageId());
            // Display the queue message.
            System.out.print("From queue: ");
            byte[] b = new byte[200];
            String s = null;
            int numRead = message.getBody().read(b);
            while (-1 != numRead)
            {
                s = new String(b);
                s = s.trim();
                System.out.print(s);
                numRead = message.getBody().read(b);
            }
            System.out.println();
            System.out.println("Custom Property: " +
                message.getProperty("MyProperty"));
            // Remove message from queue.
            System.out.println("Deleting this message.");
            //service.deleteMessage(message);
        }
        else
        {
            System.out.println("Finishing up - no more messages.");
            break;
            // Added to handle no more messages.
            // Could instead wait for more messages to be added.
        }
    }
}
catch (ServiceException e) {
    System.out.print("ServiceException encountered: ");
    System.out.println(e.getMessage());
    System.exit(-1);
}
catch (Exception e) {
    System.out.print("Generic exception encountered: ");
    System.out.println(e.getMessage());
    System.exit(-1);
}
```

How to handle application crashes and unreadable messages

Service Bus provides functionality to help you gracefully recover from errors in your application or difficulties processing a message. If a receiver application is unable to process the message for some reason, then it can call the **unlockMessage** method on the received message (instead of the **deleteMessage** method). This will cause Service Bus to unlock the message within the queue and make it available to be received again, either by the same consuming application or by another consuming application.

There is also a timeout associated with a message locked within the queue, and if the application fails to process the message before the lock timeout expires (e.g., if the application crashes), then Service Bus will unlock the message automatically and make it available to be received again.

In the event that the application crashes after processing the message but before the **deleteMessage** request is issued, then 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 application developers should add additional logic to their application to handle duplicate message delivery. This is often achieved using the **getMessageId** method of the message, which will remain constant across delivery attempts.

Next Steps

Now that you've learned the basics of Service Bus queues, see Queues, topics, and subscriptions for more information.

For more information, see the Java Developer Center.

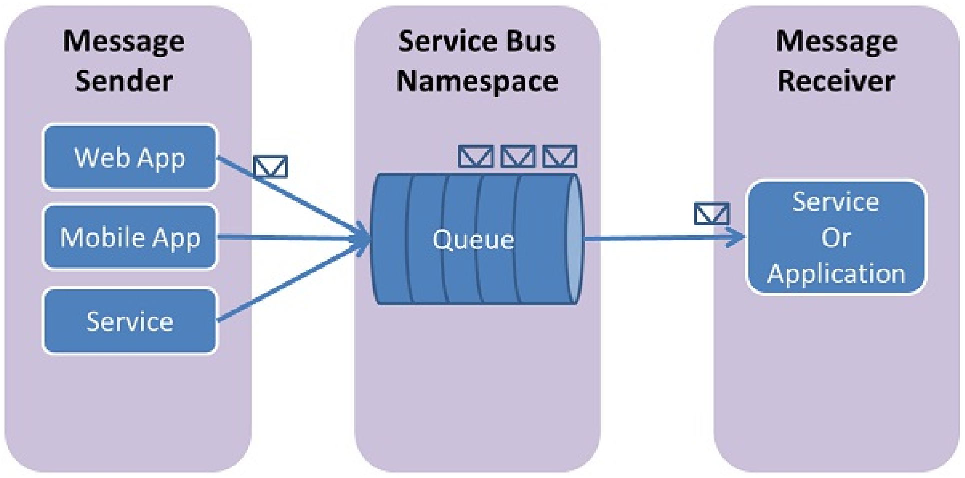
How to use Service Bus queues

1/17/2017 • 9 min to read • Edit on GitHub

This article describes how to use Service Bus queues in Node.js. The samples are written in JavaScript and use the Node.js Azure module. The scenarios covered include **creating queues**, **sending and receiving messages**, and **deleting queues**. For more information on queues, see the Next steps section.

What are Service Bus queues?

Service Bus queues support a **brokered messaging** communication model. When using queues, components of a distributed application do not communicate directly with each other; instead they exchange messages via a queue, which acts as an intermediary (broker). A message producer (sender) hands off a message to the queue and then continues its processing. Asynchronously, a message consumer (receiver) pulls the message from the queue and processes it. The producer does not have to wait for a reply from the consumer in order to continue to process and send further messages. Queues offer **First In, First Out (FIFO)** message delivery to one or more competing consumers. That is, messages are typically received and processed by the receivers in the order in which they were added to the queue, and each message is received and processed by only one message consumer.



Service Bus queues are a general-purpose technology that can be used for a wide variety of scenarios:

- Communication between web and worker roles in a multi-tier Azure application.
- Communication between on-premises apps and Azure-hosted apps in a hybrid solution.
- Communication between components of a distributed application running on-premises in different organizations or departments of an organization.

Using queues enables you to scale your applications more easily, and enable more resiliency to your architecture.

Create a service namespace

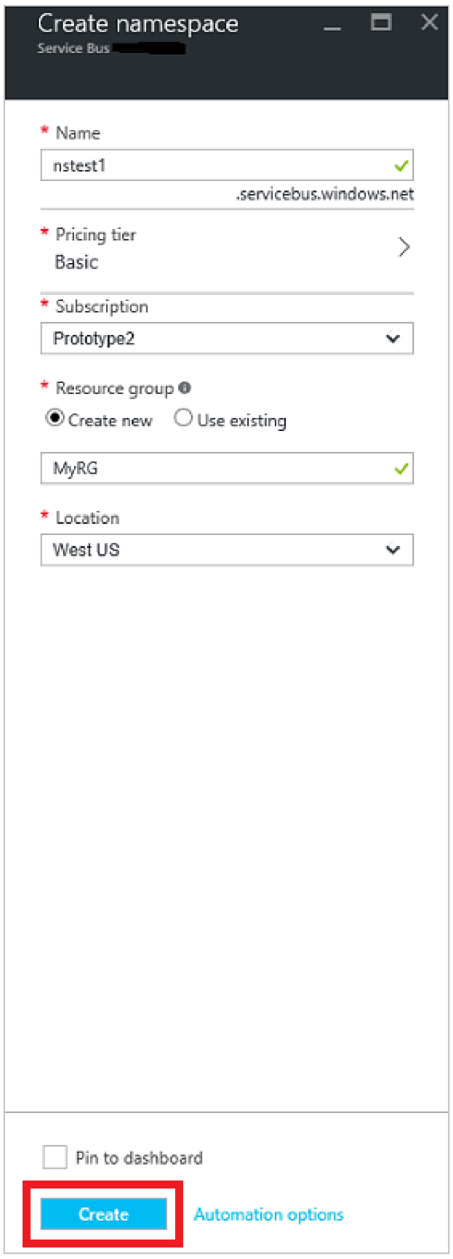
To begin using Service Bus queues in Azure, you must first create a namespace. A namespace provides a scoping container for addressing Service Bus resources within your application.

To create a namespace:

1. Log on to the Azure portal.
2. In the left navigation pane of the portal, click **New**, then click **Enterprise Integration**, and then click **Service Bus**.
3. In the **Create namespace** dialog, enter a namespace name. The system immediately checks to see if the name

is available.

4. After making sure the namespace name is available, choose the pricing tier (Basic, Standard, or Premium).
5. In the **Subscription** field, choose an Azure subscription in which to create the namespace.
6. In the **Resource group** field, choose an existing resource group in which the namespace will live, or create a new one.
7. In **Location**, choose the country or region in which your namespace should be hosted.


Create namespace
Service Bus [redacted]

* Name: nstest1 .servicebus.windows.net

* Pricing tier: Basic

* Subscription: Prototype2

* Resource group: [✓ Create new] Use existing

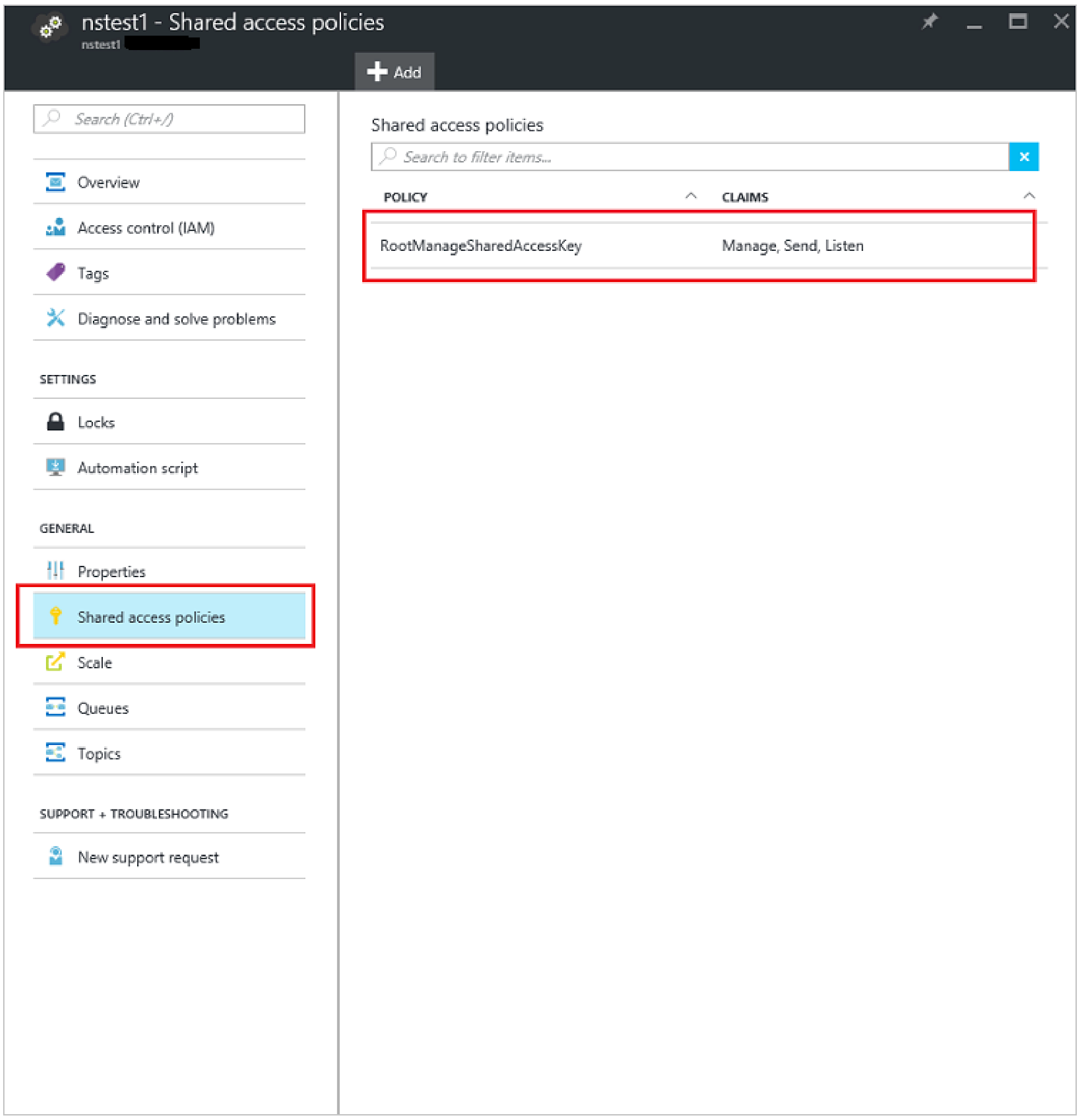
MyRG

* Location: West US

8. Click **Create**. The system now creates your namespace and enables it. You might have to wait several minutes as the system provisions resources for your account.

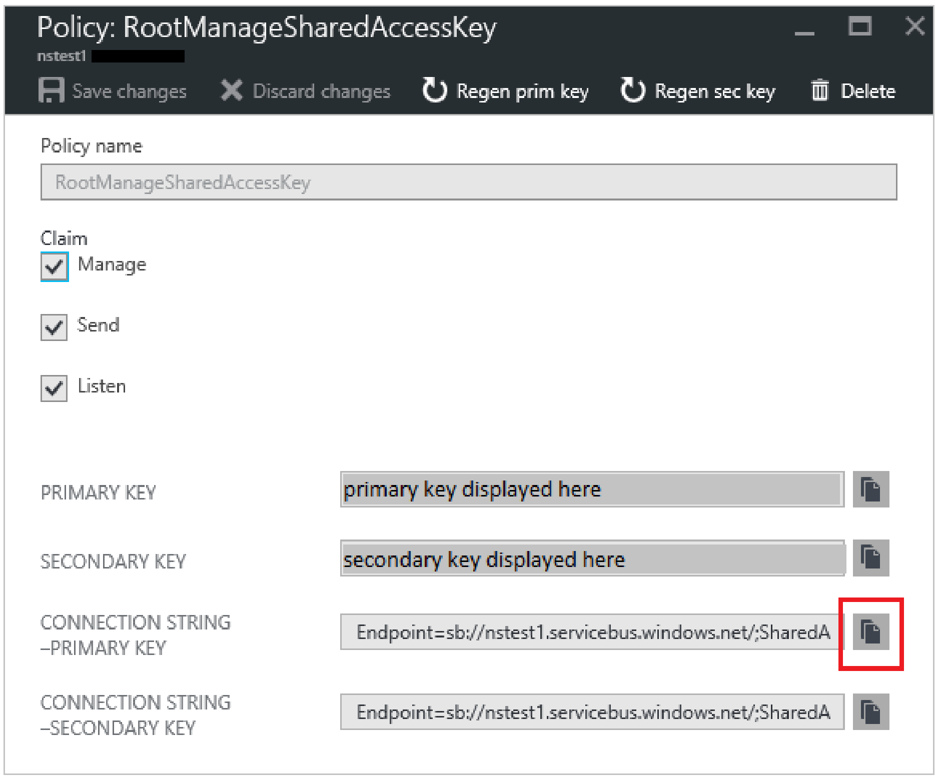
Obtain the management credentials

1. In the list of namespaces, click the newly created namespace name.
2. In the namespace blade, click **Shared access policies**.
3. In the **Shared access policies** blade, click **RootManageSharedAccessKey**.



POLICY	CLAIMS
RootManageSharedAccessKey	Manage, Send, Listen

4. In the **Policy: RootManageSharedAccessKey** blade, click the copy button next to **Connection string–primary key**, to copy the connection string to your clipboard for later use. Paste this value into Notepad or some other temporary location.



nstest1 [redacted]

Policy name: RootManageSharedAccessKey

Claim
- [✓] Manage
- [✓] Send
- [✓] Listen

PRIMARY KEY: primary key displayed here

SECONDARY KEY: secondary key displayed here

CONNECTION STRING –PRIMARY KEY: Endpoint=sb://nstest1.servicebus.windows.net/;SharedA

CONNECTION STRING –SECONDARY KEY: Endpoint=sb://nstest1.servicebus.windows.net/;SharedA

5. Repeat the previous step, copying and pasting the value of **Primary key** to a temporary location for later use.

Create a Node.js application

Create a blank Node.js application. For instructions on how to create a Node.js application, see Create and deploy a Node.js application to an Azure Website, or Node.js Cloud Service using Windows PowerShell.

Configure your application to use Service Bus

To use Azure Service Bus, download and use the Node.js Azure package. This package includes a set of libraries that communicate with the Service Bus REST services.

Use Node Package Manager (NPM) to obtain the package

1. Use the **Windows PowerShell for Node.js** command window to navigate to the **c:\node\sbqueues\WebRole1** folder in which you created your sample application.
2. Type **npm install azure** in the command window, which should result in output similar to the following:

```
 azure@0.7.5 node_modules\azure
     ├── dateformat@1.0.2-1.2.3
     ├── xmlbuilder@0.4.2
     ├── node-uuid@1.2.0
     ├── mime@1.2.9
     ├── underscore@1.4.4
     ├── validator@1.1.1
     ├── tunnel@0.0.2
     ├── wns@0.5.3
     ├── xml2js@0.2.7 (sax@0.5.2)
     └── request@2.21.0 (json-stringify-safe@4.0.0, forever-agent@0.5.0, aws-sign@0.3.0, tunnel-
agent@0.3.0, oauth-sign@0.3.0, qs@0.6.5, cookie-jar@0.3.0, node-uuid@1.4.0, http-signature@0.9.11,
form-data@0.0.8, hawk@0.13.1)
```

3. You can manually run the **ls** command to verify that a **node_modules** folder was created. Inside that folder find the **azure** package, which contains the libraries you need to access Service Bus queues.

Import the module

Using Notepad or another text editor, add the following to the top of the **server.js** file of the application:

```
var azure = require('azure');
```

Set up an Azure Service Bus connection

The Azure module reads the environment variables AZURE_SERVICEBUS_NAMESPACE and AZURE_SERVICEBUS_ACCESS_KEY to obtain information required to connect to Service Bus. If these environment variables are not set, you must specify the account information when calling **createServiceBusService**.

For an example of setting the environment variables in a configuration file for an Azure Cloud Service, see Node.js Cloud Service with Storage.

For an example of setting the environment variables in the Azure classic portal for an Azure Website, see Node.js Web Application with Storage.

Create a queue

The **ServiceBusService** object enables you to work with Service Bus queues. The following code creates a **ServiceBusService** object. Add it near the top of the **server.js** file, after the statement to import the Azure module:

```
var serviceBusService = azure.createServiceBusService();
```

By calling **createQueueIfNotExists** on the **ServiceBusService** object, the specified queue is returned (if it exists), or a new queue with the specified name is created. The following code uses **createQueueIfNotExists** to create or connect to the queue named `myqueue`:

```
serviceBusService.createQueueIfNotExists('myqueue', function(error){
    if(!error){
        // Queue exists
    }
});
```

createServiceBusService also supports additional options, which enable you to override default queue settings such as message time to live or maximum queue size. The following example sets the maximum queue size to 5 GB, and a time to live (TTL) value of 1 minute:

```
var queueOptions = {
      MaxSizeInMegabytes: '5120',
      DefaultMessageTimeToLive: 'PT1M'
    };

serviceBusService.createQueueIfNotExists('myqueue', queueOptions, function(error){
    if(!error){
        // Queue exists
    }
});
```

Filters

Optional filtering operations can be applied to operations performed using **ServiceBusService**. Filtering operations can include logging, automatically retrying, etc. Filters are objects that implement a method with the signature:

```
function handle (requestOptions, next)
```

After doing its pre-processing on the request options, the method must call `next`, passing a callback with the following signature:

```
function (returnObject, finalCallback, next)
```

In this callback, and after processing the **returnObject** (the response from the request to the server), the callback must either invoke `next` if it exists to continue processing other filters, or simply invoke `finalCallback`, which ends the service invocation.

Two filters that implement retry logic are included with the Azure SDK for Node.js, **ExponentialRetryPolicyFilter** and **LinearRetryPolicyFilter**. The following creates a **ServiceBusService** object that uses the **ExponentialRetryPolicyFilter**:

```
var retryOperations = new azure.ExponentialRetryPolicyFilter();
var serviceBusService = azure.createServiceBusService().withFilter(retryOperations);
```

Send messages to a queue

To send a message to a Service Bus queue, your application calls the **sendQueueMessage** method on the **ServiceBusService** object. Messages sent to (and received from) Service Bus queues are **BrokeredMessage** objects, and have a set of standard properties (such as **Label** and **TimeToLive**), a dictionary that is used to hold custom application-specific properties, and a body of arbitrary application data. An application can set the body of the message by passing a string as the message. Any required standard properties are populated with default values.

The following example demonstrates how to send a test message to the queue named `myqueue` using **sendQueueMessage**:

```
var message = {
    body: 'Test message',
    customProperties: {
        testproperty: 'TestValue'
    }};
serviceBusService.sendQueueMessage('myqueue', message, function(error){
    if(!error){
        // message sent
    }
});
```

Service Bus queues support a maximum message size of 256 KB in the Standard tier and 1 MB in the Premium tier. The header, which includes the standard and custom application properties, can have a maximum size of 64 KB. There is no limit on the number of messages held in a queue but there is a cap on the total size of the messages held by a queue. This queue size is defined at creation time, with an upper limit of 5 GB. For more information about quotas, see Service Bus quotas.

Receive messages from a queue

Messages are received from a queue using the **receiveQueueMessage** method on the **ServiceBusService** object. By default, messages are deleted from the queue as they are read; however, you can read (peek) and lock the message without deleting it from the queue by setting the optional parameter **isPeekLock** to **true**.

The default behavior of reading and deleting the message as part of the receive operation is the simplest model, and works best for scenarios in which an 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. Because Service Bus will have marked the message as being consumed, then when the application restarts and begins consuming messages again, it will have missed the message that was consumed prior to the crash.

If the **isPeekLock** parameter is set to **true**, the receive becomes a two stage operation, which makes it possible to support applications that cannot tolerate missing messages. When Service Bus receives a request, it finds the next message to be consumed, locks it to prevent other consumers 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 **deleteMessage** method and providing the message to be deleted as a parameter. The **deleteMessage** method will mark the message as being consumed and remove it from the queue.

The following example demonstrates how to receive and process messages using **receiveQueueMessage**. The example first receives and deletes a message, and then receives a message using **isPeekLock** set to **true**, then deletes the message using **deleteMessage**:

```
serviceBusService.receiveQueueMessage('myqueue', function(error, receivedMessage){
    if(!error){
        // Message received and deleted
    }
});
serviceBusService.receiveQueueMessage('myqueue', { isPeekLock: true }, function(error, lockedMessage){
    if(!error){
        // Message received and locked
        serviceBusService.deleteMessage(lockedMessage, function (deleteError){
            if(!deleteError){
                // Message deleted
            }
        });
    }
});
```

How to handle application crashes and unreadable messages

Service Bus provides functionality to help you gracefully recover from errors in your application or difficulties processing a message. If a receiver application is unable to process the message for some reason, then it can call the **unlockMessage** method on the **ServiceBusService** object. This will cause Service Bus to unlock the message within the queue and make it available to be received again, either by the same consuming application or by another consuming application.

There is also a timeout associated with a message locked within the queue, and if the application fails to process the message before the lock timeout expires (e.g., if the application crashes), then Service Bus will unlock the message automatically and make it available to be received again.

In the event that the application crashes after processing the message but before the **deleteMessage** method is called, then 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 application developers should add additional logic to their application to handle duplicate message delivery. This is often achieved using the **MessageId** property of the message, which will remain constant across delivery attempts.

Next steps

To learn more about queues, see the following resources.

- Queues, topics, and subscriptions
- Azure SDK for Node repository on GitHub
- Node.js Developer Center

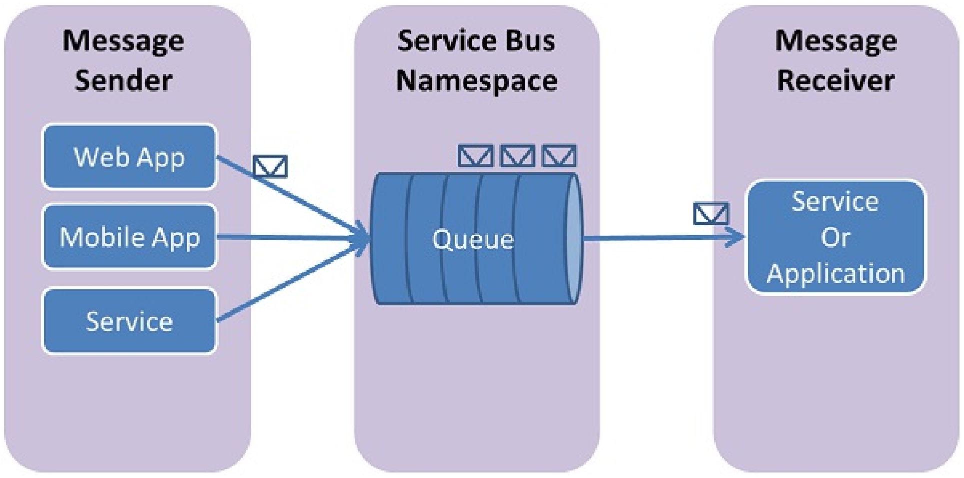
How to use Service Bus queues

1/31/2017 • 8 min to read • Edit on GitHub

This guide shows you how to use Service Bus queues. The samples are written in PHP and use the Azure SDK for PHP. The scenarios covered include **creating queues**, **sending and receiving messages**, and **deleting queues**.

What are Service Bus queues?

Service Bus queues support a **brokered messaging** communication model. When using queues, components of a distributed application do not communicate directly with each other; instead they exchange messages via a queue, which acts as an intermediary (broker). A message producer (sender) hands off a message to the queue and then continues its processing. Asynchronously, a message consumer (receiver) pulls the message from the queue and processes it. The producer does not have to wait for a reply from the consumer in order to continue to process and send further messages. Queues offer **First In, First Out (FIFO)** message delivery to one or more competing consumers. That is, messages are typically received and processed by the receivers in the order in which they were added to the queue, and each message is received and processed by only one message consumer.



Service Bus queues are a general-purpose technology that can be used for a wide variety of scenarios:

- Communication between web and worker roles in a multi-tier Azure application.
- Communication between on-premises apps and Azure-hosted apps in a hybrid solution.
- Communication between components of a distributed application running on-premises in different organizations or departments of an organization.

Using queues enables you to scale your applications more easily, and enable more resiliency to your architecture.

Create a PHP application

The only requirement for creating a PHP application that accesses the Azure Blob service is the referencing of classes in the Azure SDK for PHP from within your code. You can use any development tools to create your application, or Notepad.

NOTE

Your PHP installation must also have the OpenSSL extension installed and enabled.

In this guide, you will use service features which can be called from within a PHP application locally, or in code running within an Azure web role, worker role, or website.

Get the Azure client libraries

Install via Composer

1. Install Git. Note that on Windows, you must also add the Git executable to your PATH environment variable.
2. Create a file named **composer.json** in the root of your project and add the following code to it:

```
{
  "require": {
    "microsoft/windowsazure": "^0.4"
  }
}
```

3. Download **composer.phar** in your project root.
4. Open a command prompt and execute the following command in your project root

```
php composer.phar install
```

Configure your application to use Service Bus

To use the Service Bus queue APIs, do the following:

1. Reference the autoloader file using the require_once statement.
2. Reference any classes you might use.

The following example shows how to include the autoloader file and reference the `ServicesBuilder` class.

NOTE
This example (and other examples in this article) assumes you have installed the PHP Client Libraries for Azure via Composer. If you installed the libraries manually or as a PEAR package, you must reference the **WindowsAzure.php** autoloader file.

```
require_once 'vendor/autoload.php';
use WindowsAzure\Common\ServicesBuilder;
```

In the examples below, the `require_once` statement will always be shown, but only the classes necessary for the example to execute are referenced.

Set up a Service Bus connection

To instantiate a Service Bus client, you must first have a valid connection string in this format:

```
Endpoint=[yourEndpoint];SharedSecretIssuer=[Default Issuer];SharedSecretValue=[Default Key]
```

Where `Endpoint` is typically of the format `[yourNamespace].servicebus.windows.net`.

To create any Azure service client you must use the `ServicesBuilder` class. You can:

- Pass the connection string directly to it.
- Use the **CloudConfigurationManager (CCM)** to check multiple external sources for the connection string:

- By default it comes with support for one external source - environmental variables
- You can add new sources by extending the `ConnectionStringSource` class

For the examples outlined here, the connection string will be passed directly.

```
require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;

$connectionString = "Endpoint=[yourEndpoint];SharedSecretIssuer=[Default Issuer];SharedSecretValue=[Default
Key]";

$serviceBusRestProxy = ServicesBuilder::getInstance()->createServiceBusService($connectionString);
```

How to: create a queue

You can perform management operations for Service Bus queues via the `ServiceBusRestProxy` class. A `ServiceBusRestProxy` object is constructed via the `ServicesBuilder::createServiceBusService` factory method with an appropriate connection string that encapsulates the token permissions to manage it.

The following example shows how to instantiate a `ServiceBusRestProxy` and call `ServiceBusRestProxy->createQueue` to create a queue named `myqueue` within a `MySBNamespace` service namespace:

```
require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use WindowsAzure\Common\ServiceException;
use WindowsAzure\ServiceBus\Models\QueueInfo;

// Create Service Bus REST proxy.
$serviceBusRestProxy = ServicesBuilder::getInstance()->createServiceBusService($connectionString);

try    {
    $queueInfo = new QueueInfo("myqueue");

    // Create queue.
    $serviceBusRestProxy->createQueue($queueInfo);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/windowsazure/dd179357
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}
```

NOTE

You can use the `listQueues` method on `ServiceBusRestProxy` objects to check if a queue with a specified name already exists within a namespace.

How to: send messages to a queue

To send a message to a Service Bus queue, your application calls the `ServiceBusRestProxy->sendQueueMessage` method. The following code shows how to send a message to the `myqueue` queue previously created within the `MySBNamespace` service namespace.

```
require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use WindowsAzure\Common\ServiceException;
use WindowsAzure\ServiceBus\Models\BrokeredMessage;

// Create Service Bus REST proxy.
$serviceBusRestProxy = ServicesBuilder::getInstance()->createServiceBusService($connectionString);

try    {
    // Create message.
    $message = new BrokeredMessage();
    $message->setBody("my message");

    // Send message.
    $serviceBusRestProxy->sendQueueMessage("myqueue", $message);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/windowsazure/hh780775
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}
```

Messages sent to (and received from) Service Bus queues are instances of the BrokeredMessage class. BrokeredMessage objects have a set of standard methods and properties that are used to hold custom application-specific properties, and a body of arbitrary application data.

Service Bus queues support a maximum message size of 256 KB in the Standard tier and 1 MB in the Premium tier. The header, which includes the standard and custom application properties, can have a maximum size of 64 KB. There is no limit on the number of messages held in a queue but there is a cap on the total size of the messages held by a queue. This upper limit on queue size is 5 GB.

How to receive messages from a queue

The best way to receive messages from a queue is to use a `ServiceBusRestProxy->receiveQueueMessage` method. Messages can be received in two different modes: *ReceiveAndDelete* and *PeekLock*. **PeekLock** is the default.

When using ReceiveAndDelete mode, receive is a single-shot operation; that is, when Service Bus receives a read request for a message in a queue, it marks the message as being consumed and returns it to the application. ReceiveAndDelete mode is the simplest model and works best for scenarios in which an 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. Because Service Bus will have marked the message as being consumed, then when the application restarts and begins consuming messages again, it will have missed the message that was consumed prior to the crash.

In the default PeekLock mode, receiving a message becomes a two stage operation, which makes it possible to support applications that cannot tolerate missing messages. When Service Bus receives a 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 passing the received message to `ServiceBusRestProxy->deleteMessage`. When Service Bus sees the `deleteMessage` call, it will mark the message as being consumed and remove it from the queue.

The following example shows how to receive and process a message using PeekLock mode (the default mode).

```
require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use WindowsAzure\Common\ServiceException;
use WindowsAzure\ServiceBus\Models\ReceiveMessageOptions;

// Create Service Bus REST proxy.
$serviceBusRestProxy = ServicesBuilder::getInstance()->createServiceBusService($connectionString);

try    {
    // Set the receive mode to PeekLock (default is ReceiveAndDelete).
    $options = new ReceiveMessageOptions();
    $options->setPeekLock();

    // Receive message.
    $message = $serviceBusRestProxy->receiveQueueMessage("myqueue", $options);
    echo "Body: ".$message->getBody()."<br />";
    echo "MessageID: ".$message->getMessageId()."<br />";

    /*---------------------------
        Process message here.
    ----------------------------*/

    // Delete message. Not necessary if peek lock is not set.
    echo "Message deleted.<br />";
    $serviceBusRestProxy->deleteMessage($message);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/windowsazure/hh780735
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}
```

How to: handle application crashes and unreadable messages

Service Bus provides functionality to help you gracefully recover from errors in your application or difficulties processing a message. If a receiver application is unable to process the message for some reason, then it can call the `unlockMessage` method on the received message (instead of the `deleteMessage` method). This will cause Service Bus to unlock the message within the queue and make it available to be received again, either by the same consuming application or by another consuming application.

There is also a timeout associated with a message locked within the queue, 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 automatically and make it available to be received again.

In the event that the application crashes after processing the message but before the `deleteMessage` request is issued, then the message will be redelivered to the application when it restarts. This is often called *At Least Once* processing; that is, each message is processed at least once but in certain situations the same message may be redelivered. If the scenario cannot tolerate duplicate processing, then adding additional logic to applications to handle duplicate message delivery is recommended. This is often achieved using the `getMessageId` method of the message, which remains constant across delivery attempts.

Next steps

Now that you've learned the basics of Service Bus queues, see Queues, topics, and subscriptions for more information.

For more information, also visit the PHP Developer Center.

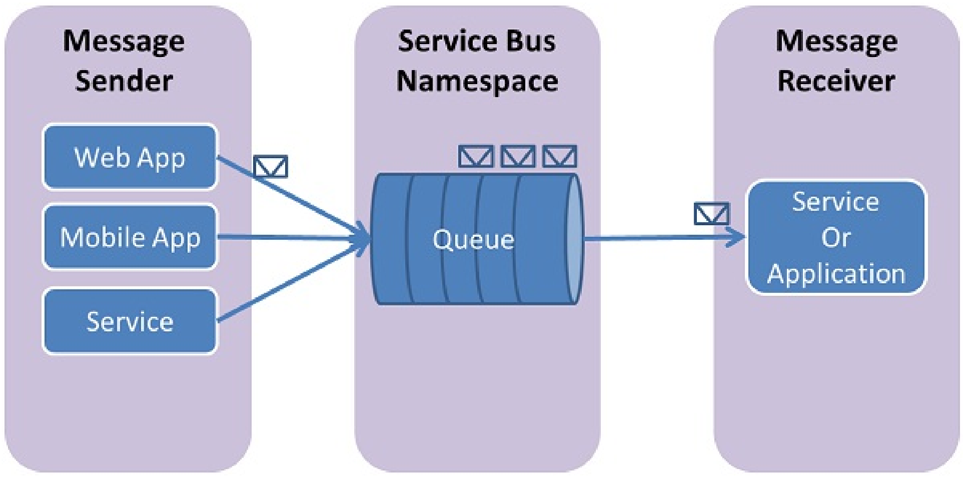
How to use Service Bus queues

1/17/2017 • 6 min to read • Edit on GitHub

This article describes how to use Service Bus queues. The samples are written in Python and use the Python Azure Service Bus package. The scenarios covered include **creating queues, sending and receiving messages**, and **deleting queues**.

What are Service Bus queues?

Service Bus queues support a **brokered messaging** communication model. When using queues, components of a distributed application do not communicate directly with each other; instead they exchange messages via a queue, which acts as an intermediary (broker). A message producer (sender) hands off a message to the queue and then continues its processing. Asynchronously, a message consumer (receiver) pulls the message from the queue and processes it. The producer does not have to wait for a reply from the consumer in order to continue to process and send further messages. Queues offer **First In, First Out (FIFO)** message delivery to one or more competing consumers. That is, messages are typically received and processed by the receivers in the order in which they were added to the queue, and each message is received and processed by only one message consumer.



Service Bus queues are a general-purpose technology that can be used for a wide variety of scenarios:

- Communication between web and worker roles in a multi-tier Azure application.
- Communication between on-premises apps and Azure-hosted apps in a hybrid solution.
- Communication between components of a distributed application running on-premises in different organizations or departments of an organization.

Using queues enables you to scale your applications more easily, and enable more resiliency to your architecture.

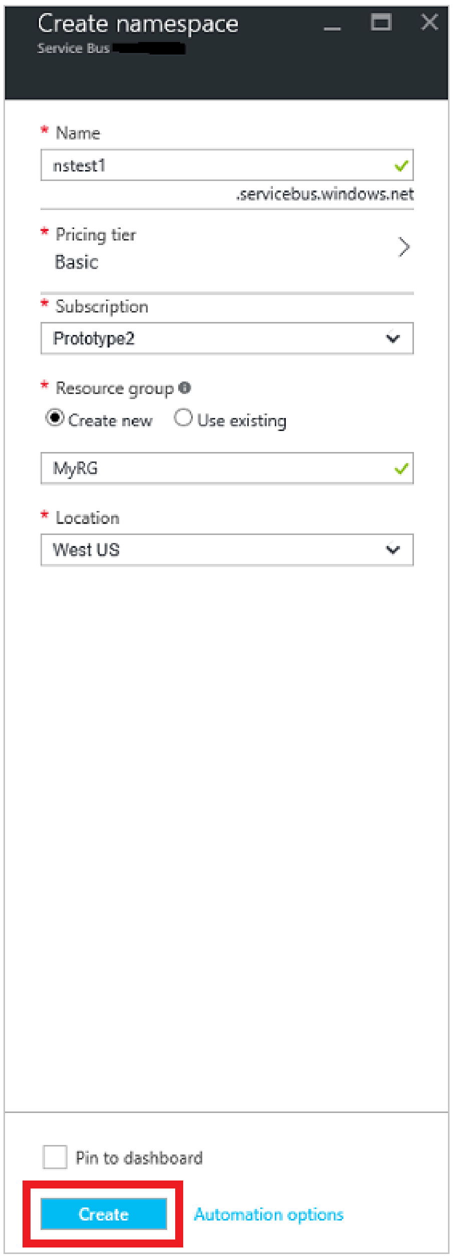
Create a service namespace

To begin using Service Bus queues in Azure, you must first create a namespace. A namespace provides a scoping container for addressing Service Bus resources within your application.

To create a namespace:

1. Log on to the Azure portal.
2. In the left navigation pane of the portal, click **New**, then click **Enterprise Integration**, and then click **Service Bus**.

3. In the **Create namespace** dialog, enter a namespace name. The system immediately checks to see if the name is available.
4. After making sure the namespace name is available, choose the pricing tier (Basic, Standard, or Premium).
5. In the **Subscription** field, choose an Azure subscription in which to create the namespace.
6. In the **Resource group** field, choose an existing resource group in which the namespace will live, or create a new one.
7. In **Location**, choose the country or region in which your namespace should be hosted.


Create namespace
Service Bus [redacted]

* Name
nstest1 ✓
.servicebus.windows.net

* Pricing tier
Basic

* Subscription
Prototype2

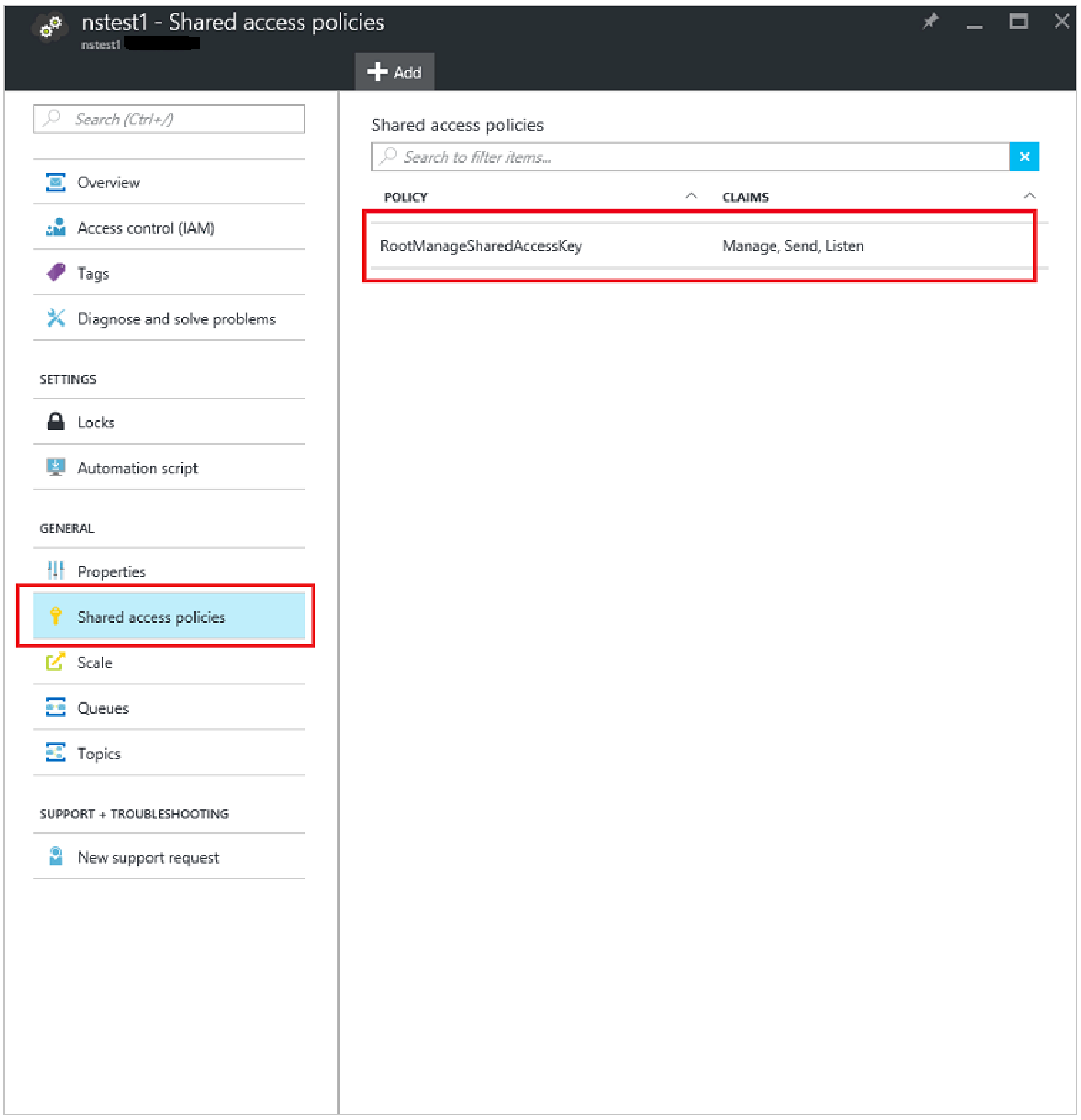
* Resource group
[✓ Create new] Use existing
MyRG ✓

* Location
West US

8. Click **Create**. The system now creates your namespace and enables it. You might have to wait several minutes as the system provisions resources for your account.

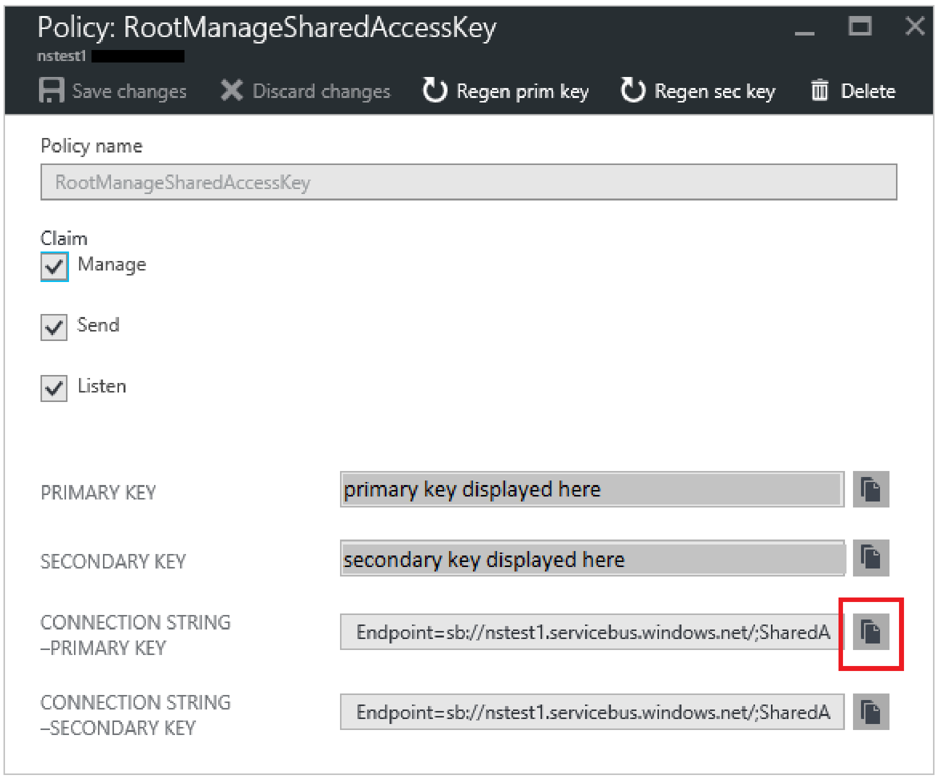
Obtain the management credentials

1. In the list of namespaces, click the newly created namespace name.
2. In the namespace blade, click **Shared access policies**.
3. In the **Shared access policies** blade, click **RootManageSharedAccessKey**.



POLICY	CLAIMS
RootManageSharedAccessKey	Manage, Send, Listen

4. In the **Policy: RootManageSharedAccessKey** blade, click the copy button next to **Connection string–primary key**, to copy the connection string to your clipboard for later use. Paste this value into Notepad or some other temporary location.



Policy: RootManageSharedAccessKey

nstest1 [redacted]

Policy name: RootManageSharedAccessKey

Claim

- [✓] Manage
- [✓] Send
- [✓] Listen

PRIMARY KEY	primary key displayed here
SECONDARY KEY	secondary key displayed here
CONNECTION STRING –PRIMARY KEY	Endpoint=sb://nstest1.servicebus.windows.net/;SharedA
CONNECTION STRING –SECONDARY KEY	Endpoint=sb://nstest1.servicebus.windows.net/;SharedA

5. Repeat the previous step, copying and pasting the value of **Primary key** to a temporary location for later use.

NOTE

To install Python or the Python Azure Service Bus package, see the Python Installation Guide.

Create a queue

The **ServiceBusService** object enables you to work with queues. Add the following code near the top of any Python file in which you wish to programmatically access Service Bus:

```
from azure.servicebus import ServiceBusService, Message, Queue
```

The following code creates a **ServiceBusService** object. Replace `mynamespace`, `sharedaccesskeyname`, and `sharedaccesskey` with your namespace, shared access signature (SAS) key name, and value.

```
bus_service = ServiceBusService(
    service_namespace='mynamespace',
    shared_access_key_name='sharedaccesskeyname',
    shared_access_key_value='sharedaccesskey')
```

The values for the SAS key name and value can be found in the Azure classic portal connection information, or in the Visual Studio **Properties** pane when selecting the Service Bus namespace in Server Explorer (as shown in the previous section).

```
bus_service.create_queue('taskqueue')
```

create_queue also supports additional options, which enable you to override default queue settings such as message time to live (TTL) or maximum queue size. The following example sets the maximum queue size to 5GB, and the TTL value to 1 minute:

```
queue_options = Queue()
queue_options.max_size_in_megabytes = '5120'
queue_options.default_message_time_to_live = 'PT1M'

bus_service.create_queue('taskqueue', queue_options)
```

Send messages to a queue

To send a message to a Service Bus queue, your application calls the **send_queue_message** method on the **ServiceBusService** object.

The following example demonstrates how to send a test message to the queue named *taskqueue using* **send_queue_message**:

```
msg = Message(b'Test Message')
bus_service.send_queue_message('taskqueue', msg)
```

Service Bus queues support a maximum message size of 256 KB in the Standard tier and 1 MB in the Premium tier. The header, which includes the standard and custom application properties, can have a maximum size of 64 KB. There is no limit on the number of messages held in a queue but there is a cap on the total size of the messages held by a queue. This queue size is defined at creation time, with an upper limit of 5 GB. For more information about quotas, see Service Bus quotas.

Receive messages from a queue

Messages are received from a queue using the **receive_queue_message** method on the **ServiceBusService** object:

```
msg = bus_service.receive_queue_message('taskqueue', peek_lock=False)
print(msg.body)
```

Messages are deleted from the queue as they are read when the parameter **peek_lock** is set to **False**. You can read (peek) and lock the message without deleting it from the queue by setting the parameter **peek_lock** to **True**.

The behavior of reading and deleting the message as part of the receive operation is the simplest model, and works best for scenarios in which an 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. Because Service Bus will have marked the message as being consumed, then when the application restarts and begins consuming messages again, it will have missed the message that was consumed prior to the crash.

If the **peek_lock** parameter is set to **True**, the receive becomes a two stage operation, which makes it possible to support applications that cannot tolerate missing messages. When Service Bus receives a request, it finds the next message to be consumed, locks it to prevent other consumers 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 the **delete** method on the **Message** object. The **delete** method will mark the message as being consumed and remove it from the queue.

```
msg = bus_service.receive_queue_message('taskqueue', peek_lock=True)
print(msg.body)

msg.delete()
```

How to handle application crashes and unreadable messages

Service Bus provides functionality to help you gracefully recover from errors in your application or difficulties processing a message. If a receiver application is unable to process the message for some reason, then it can call the **unlock** method on the **Message** object. This will cause Service Bus to unlock the message within the queue and make it available to be received again, either by the same consuming application or by another consuming application.

There is also a timeout associated with a message locked within the queue, and if the application fails to process the message before the lock timeout expires (e.g., if the application crashes), then Service Bus will unlock the message automatically and make it available to be received again.

In the event that the application crashes after processing the message but before the **delete** method is called, then 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 application developers should add additional logic to their application to handle duplicate message delivery. This is often achieved using the **MessageId** property of the message, which will remain constant across delivery attempts.

Next steps

Now that you have learned the basics of Service Bus queues, see these articles to learn more.

- Queues, topics, and subscriptions

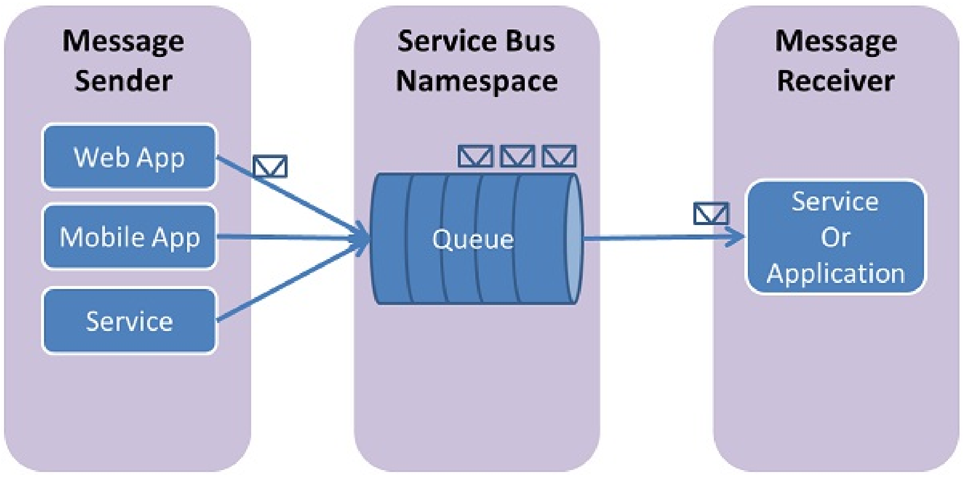
How to use Service Bus queues

1/17/2017 • 8 min to read • Edit on GitHub

This guide describes how to use Service Bus queues. The samples are written in Ruby and use the Azure gem. The scenarios covered include **creating queues, sending and receiving messages**, and **deleting queues**. For more information about Service Bus queues, see the Next Steps section.

What are Service Bus queues?

Service Bus queues support a **brokered messaging** communication model. When using queues, components of a distributed application do not communicate directly with each other; instead they exchange messages via a queue, which acts as an intermediary (broker). A message producer (sender) hands off a message to the queue and then continues its processing. Asynchronously, a message consumer (receiver) pulls the message from the queue and processes it. The producer does not have to wait for a reply from the consumer in order to continue to process and send further messages. Queues offer **First In, First Out (FIFO)** message delivery to one or more competing consumers. That is, messages are typically received and processed by the receivers in the order in which they were added to the queue, and each message is received and processed by only one message consumer.



Service Bus queues are a general-purpose technology that can be used for a wide variety of scenarios:

- Communication between web and worker roles in a multi-tier Azure application.
- Communication between on-premises apps and Azure-hosted apps in a hybrid solution.
- Communication between components of a distributed application running on-premises in different organizations or departments of an organization.

Using queues enables you to scale your applications more easily, and enable more resiliency to your architecture.

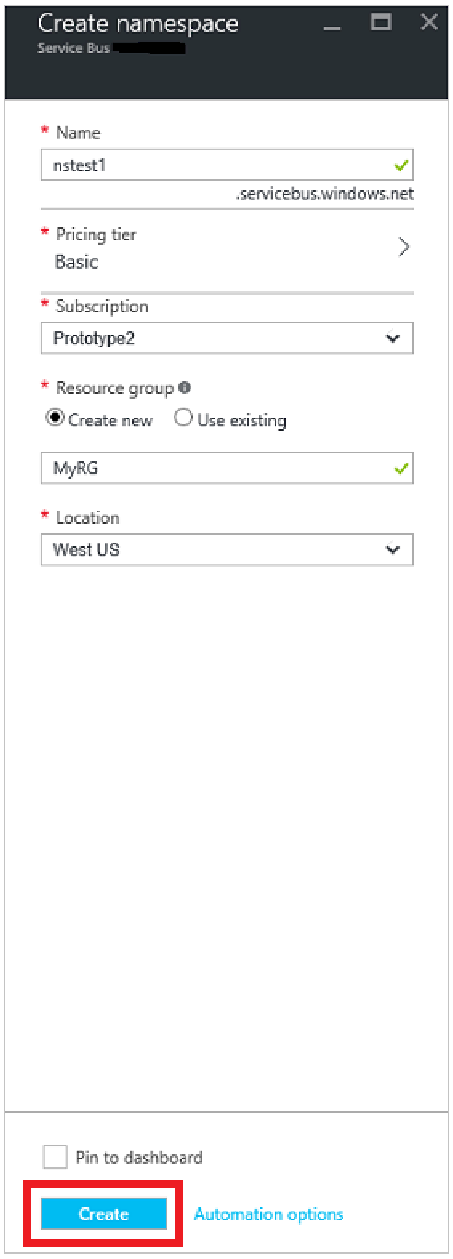
Create a service namespace

To begin using Service Bus queues in Azure, you must first create a namespace. A namespace provides a scoping container for addressing Service Bus resources within your application.

To create a namespace:

1. Log on to the Azure portal.
2. In the left navigation pane of the portal, click **New**, then click **Enterprise Integration**, and then click **Service Bus**.

3. In the **Create namespace** dialog, enter a namespace name. The system immediately checks to see if the name is available.
4. After making sure the namespace name is available, choose the pricing tier (Basic, Standard, or Premium).
5. In the **Subscription** field, choose an Azure subscription in which to create the namespace.
6. In the **Resource group** field, choose an existing resource group in which the namespace will live, or create a new one.
7. In **Location**, choose the country or region in which your namespace should be hosted.



Create namespace
Service Bus [redacted]

* Name
nstest1 ✓
.servicebus.windows.net

* Pricing tier
Basic

* Subscription
Prototype2

* Resource group
[✓ Create new] Use existing
MyRG ✓

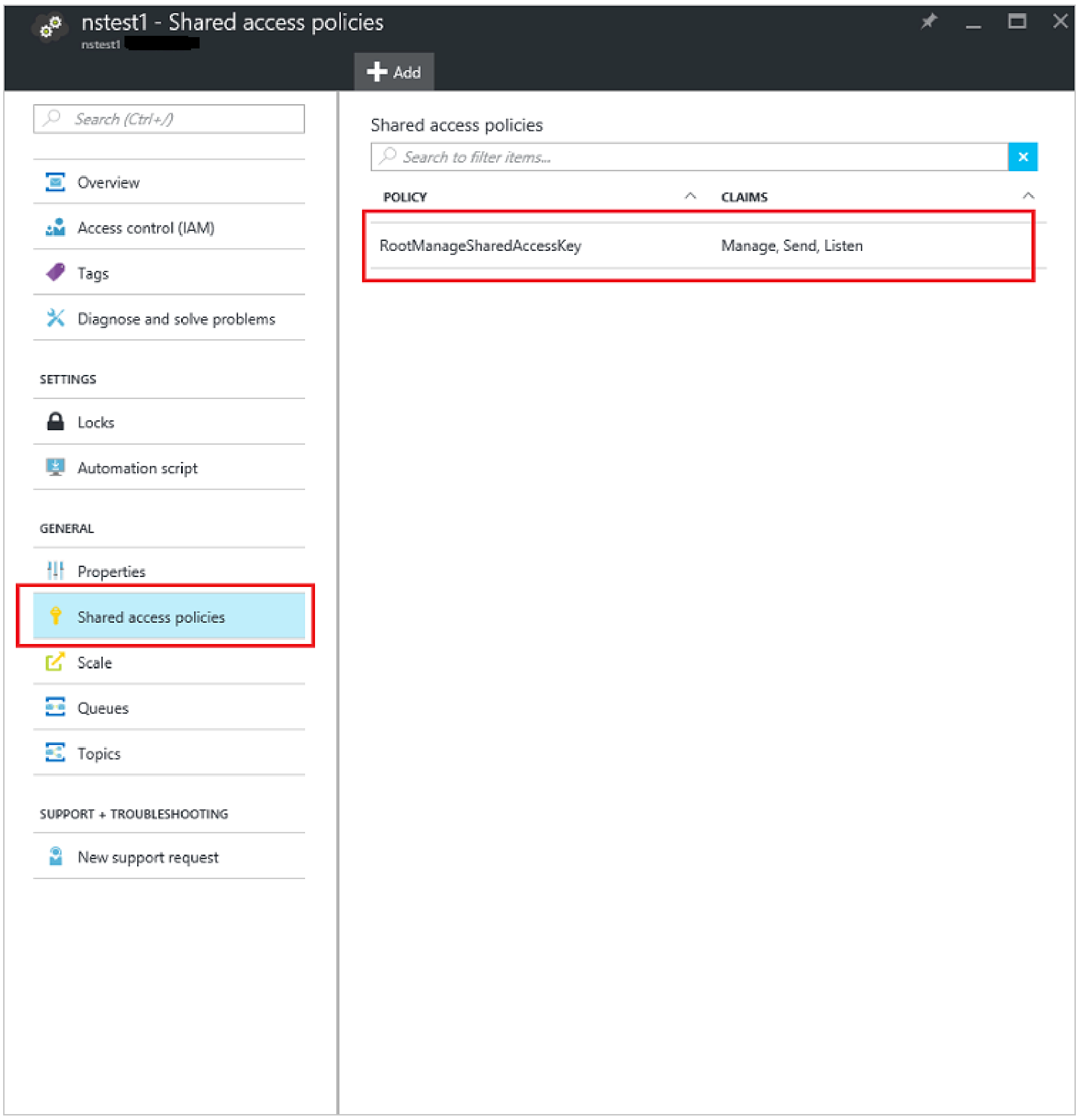
* Location
West US

Pin to dashboard

8. Click **Create**. The system now creates your namespace and enables it. You might have to wait several minutes as the system provisions resources for your account.

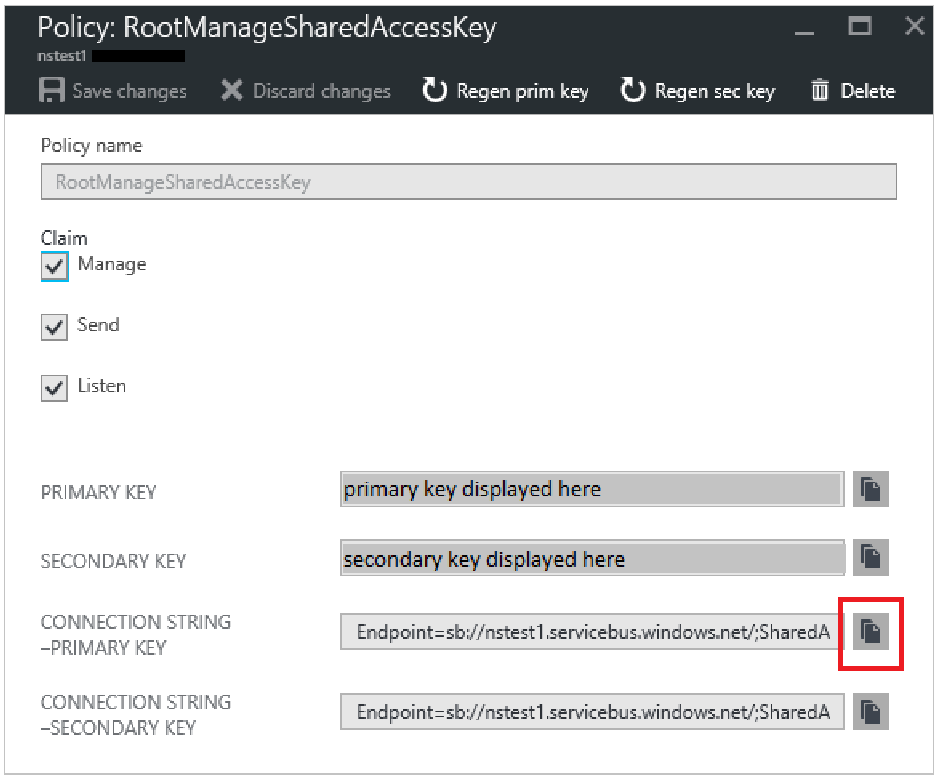
Obtain the management credentials

1. In the list of namespaces, click the newly created namespace name.
2. In the namespace blade, click **Shared access policies**.
3. In the **Shared access policies** blade, click **RootManageSharedAccessKey**.



POLICY	CLAIMS
RootManageSharedAccessKey	Manage, Send, Listen

4. In the **Policy: RootManageSharedAccessKey** blade, click the copy button next to **Connection string–primary key**, to copy the connection string to your clipboard for later use. Paste this value into Notepad or some other temporary location.


Policy: RootManageSharedAccessKey

nstest1 [redacted]

Policy name: RootManageSharedAccessKey

Claim
- ✓ Manage
- ✓ Send
- ✓ Listen

PRIMARY KEY: primary key displayed here

SECONDARY KEY: secondary key displayed here

CONNECTION STRING –PRIMARY KEY: Endpoint=sb://nstest1.servicebus.windows.net/;SharedA

CONNECTION STRING –SECONDARY KEY: Endpoint=sb://nstest1.servicebus.windows.net/;SharedA

5. Repeat the previous step, copying and pasting the value of **Primary key** to a temporary location for later use.

Create a Ruby application

Create a Ruby application. For instructions, see Create a Ruby Application on Azure.

Configure your application to use Service Bus

To use Azure Service Bus, download and use the Ruby Azure package, which includes a set of convenience libraries that communicate with the storage REST services.

Use RubyGems to obtain the package

1. Use a command-line interface such as **PowerShell** (Windows), **Terminal** (Mac), or **Bash** (Unix).
2. Type "gem install azure" in the command window to install the gem and dependencies.

Import the package

Using your favorite text editor, add the following to the top of the Ruby file where you intend to use storage:

```
require "azure"
```

Set up an Azure Service Bus connection

The Azure module reads the environment variables **AZURE_SERVICEBUS_NAMESPACE** and **AZURE_SERVICEBUS_ACCESS_KEY** for information required to connect to your Service Bus namespace. If these environment variables are not set, you must specify the namespace information before using **Azure::ServiceBusService** with the following code:

```
Azure.config.sb_namespace = "<your azure service bus namespace>"
Azure.config.sb_access_key = "<your azure service bus access key>"
```

Set the namespace value to the value you created, rather than the entire URL. For example, use

"yourexamplenamespace", not "yourexamplenamespace.servicebus.windows.net".

How to create a queue

The **Azure::ServiceBusService** object enables you to work with queues. To create a queue, use the **create_queue()** method. The following example creates a queue or prints out any errors.

```
azure_service_bus_service = Azure::ServiceBusService.new
begin
  queue = azure_service_bus_service.create_queue("test-queue")
rescue
  puts $!
end
```

You can also pass a **Azure::ServiceBus::Queue** object with additional options, which enables you to override the default queue settings, such as message time to live or maximum queue size. The following example shows how to set the maximum queue size to 5GB and time to live to 1 minute:

```
queue = Azure::ServiceBus::Queue.new("test-queue")
queue.max_size_in_megabytes = 5120
queue.default_message_time_to_live = "PT1M"

queue = azure_service_bus_service.create_queue(queue)
```

How to send messages to a queue

To send a message to a Service Bus queue, your application calls the **send_queue_message()** method on the **Azure::ServiceBusService** object. Messages sent to (and received from) Service Bus queues are **Azure::ServiceBus::BrokeredMessage** objects, and have a set of standard properties (such as **label** and **time_to_live**), a dictionary that is used to hold custom application-specific properties, and a body of arbitrary application data. An application can set the body of the message by passing a string value as the message and any required standard properties will be populated with default values.

The following example demonstrates how to send a test message to the queue named "test-queue" using **send_queue_message()**:

```
message = Azure::ServiceBus::BrokeredMessage.new("test queue message")
message.correlation_id = "test-correlation-id"
azure_service_bus_service.send_queue_message("test-queue", message)
```

Service Bus queues support a maximum message size of 256 KB in the Standard tier and 1 MB in the Premium tier. The header, which includes the standard and custom application properties, can have a maximum size of 64 KB. There is no limit on the number of messages held in a queue but there is a cap on the total size of the messages held by a queue. This queue size is defined at creation time, with an upper limit of 5 GB.

How to receive messages from a queue

Messages are received from a queue using the **receive_queue_message()** method on the **Azure::ServiceBusService** object. By default, messages are read and locked without being deleted from the queue. However, you can delete messages from the queue as they are read by setting the **:peek_lock** option to **false**.

The default behavior makes the reading and deleting a two-stage operation, which also makes it possible to support applications that cannot tolerate missing messages. When Service Bus receives a request, it finds the next message to be consumed, locks it to prevent other consumers 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 **delete_queue_message()** method and providing the message to be deleted as a parameter. The **delete_queue_message()** method will mark the message as being consumed and remove it from the queue.

If the **:peek_lock** parameter is set to **false**, reading and deleting the message becomes the simplest model, and works best for scenarios in which an 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. Because Service Bus will have marked 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.

The following example demonstrates how to receive and process messages using **receive_queue_message()**. The example first receives and deletes a message by using **:peek_lock** set to **false**, then it receives another message and then deletes the message using **delete_queue_message()**:

```
message = azure_service_bus_service.receive_queue_message("test-queue",
  { :peek_lock => false })
message = azure_service_bus_service.receive_queue_message("test-queue")
azure_service_bus_service.delete_queue_message(message)
```

How to handle application crashes and unreadable messages

Service Bus provides functionality to help you gracefully recover from errors in your application or difficulties processing a message. If a receiver application is unable to process the message for some reason, then it can call the **unlock_queue_message()** method on the **Azure::ServiceBusService** object. This causes Service Bus to unlock the message within the queue and make it available to be received again, either by the same consuming application or by another consuming application.

There is also a timeout associated with a message locked within the queue, and if the application fails to process the message before the lock timeout expires (for example, if the application crashes), then Service Bus unlocks the message automatically and makes it available to be received again.

In the event that the application crashes after processing the message but before the **delete_queue_message()** method is called, then the message will be redelivered to the application when it restarts. This is often called **At Least Once Processing**; that is, each message is processed at least once but in certain situations the same message may be redelivered. If the scenario cannot tolerate duplicate processing, then application developers should add additional logic to their application to handle duplicate message delivery. This is often achieved using the **message_id** property of the message, which remains constant across delivery attempts.

Next steps

Now that you've learned the basics of Service Bus queues, follow these links to learn more.

- Overview of queues, topics, and subscriptions.
- Visit the Azure SDK for Ruby repository on GitHub.

For a comparison between the Azure Service Bus queues discussed in this article and Azure Queues discussed in the How to use Queue storage from Ruby article, see Azure Queues and Azure Service Bus Queues - Compared and Contrasted

How to use Service Bus topics and subscriptions

1/19/2017 • 13 min to read • Edit on GitHub

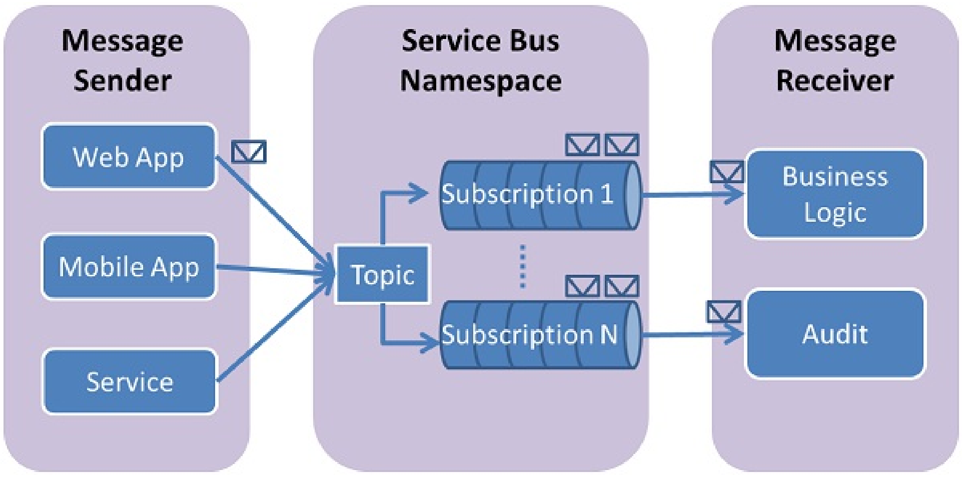
This article describes how to use Service Bus topics and subscriptions. The samples are written in C# and use the .NET APIs. The scenarios covered include creating topics and subscriptions, creating subscription filters, sending messages to a topic, receiving messages from a subscription, and deleting topics and subscriptions. For more information about topics and subscriptions, see the Next steps section.

NOTE

To complete this tutorial, you need an Azure account. You can activate your MSDN subscriber benefits or sign up for a free account.

What are Service Bus topics and subscriptions?

Service Bus topics and subscriptions support a *publish/subscribe* messaging communication model. When using topics and subscriptions, components of a distributed application do not communicate directly with each other; instead they exchange messages via a topic, which acts as an intermediary.



In contrast with Service Bus queues, in which each message is processed by a single consumer, topics and subscriptions provide a "one-to-many" form of communication, using a publish/subscribe pattern. It is possible to register multiple subscriptions to a topic. When a message is sent to a topic, it is then made available to each subscription to handle/process independently.

A subscription to a topic resembles a virtual queue that receives copies of the messages that were sent to the topic. You can optionally register filter rules for a topic on a per-subscription basis, which enables you to filter or restrict which messages to a topic are received by which topic subscriptions.

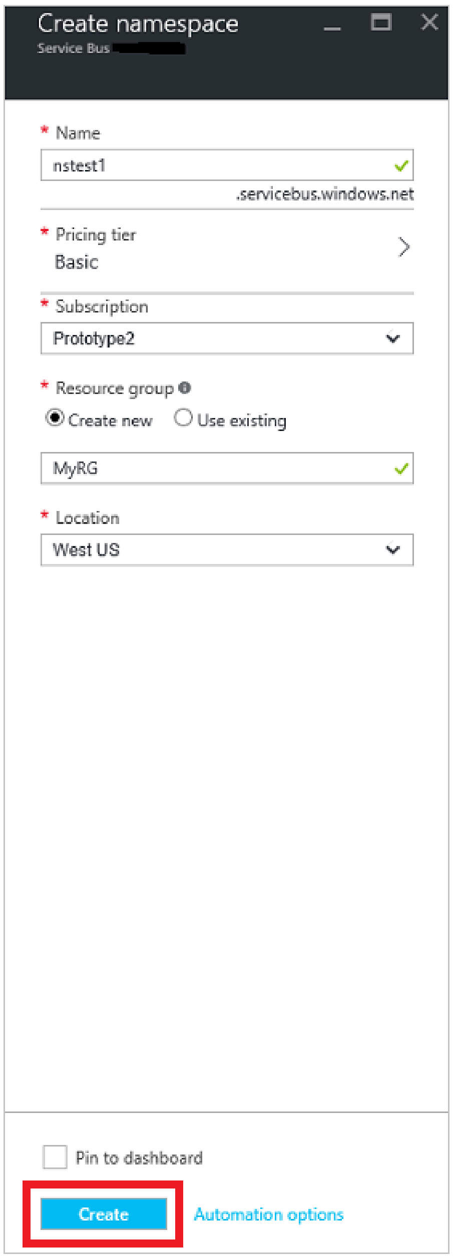
Service Bus topics and subscriptions enable you to scale and process a very large number of messages across many users and applications.

Create a namespace

To begin using Service Bus topics and subscriptions in Azure, you must first create a *service namespace*. A namespace provides a scoping container for addressing Service Bus resources within your application.

To create a namespace:

1. Log on to the Azure portal.
2. In the left navigation pane of the portal, click **New**, then click **Enterprise Integration**, and then click **Service Bus**.
3. In the **Create namespace** dialog, enter a namespace name. The system immediately checks to see if the name is available.
4. After making sure the namespace name is available, choose the pricing tier (Basic, Standard, or Premium).
5. In the **Subscription** field, choose an Azure subscription in which to create the namespace.
6. In the **Resource group** field, choose an existing resource group in which the namespace will live, or create a new one.
7. In **Location**, choose the country or region in which your namespace should be hosted.



Create namespace
Service Bus [redacted]

* Name
nstest1 ✓
.servicebus.windows.net

* Pricing tier
Basic

* Subscription
Prototype2

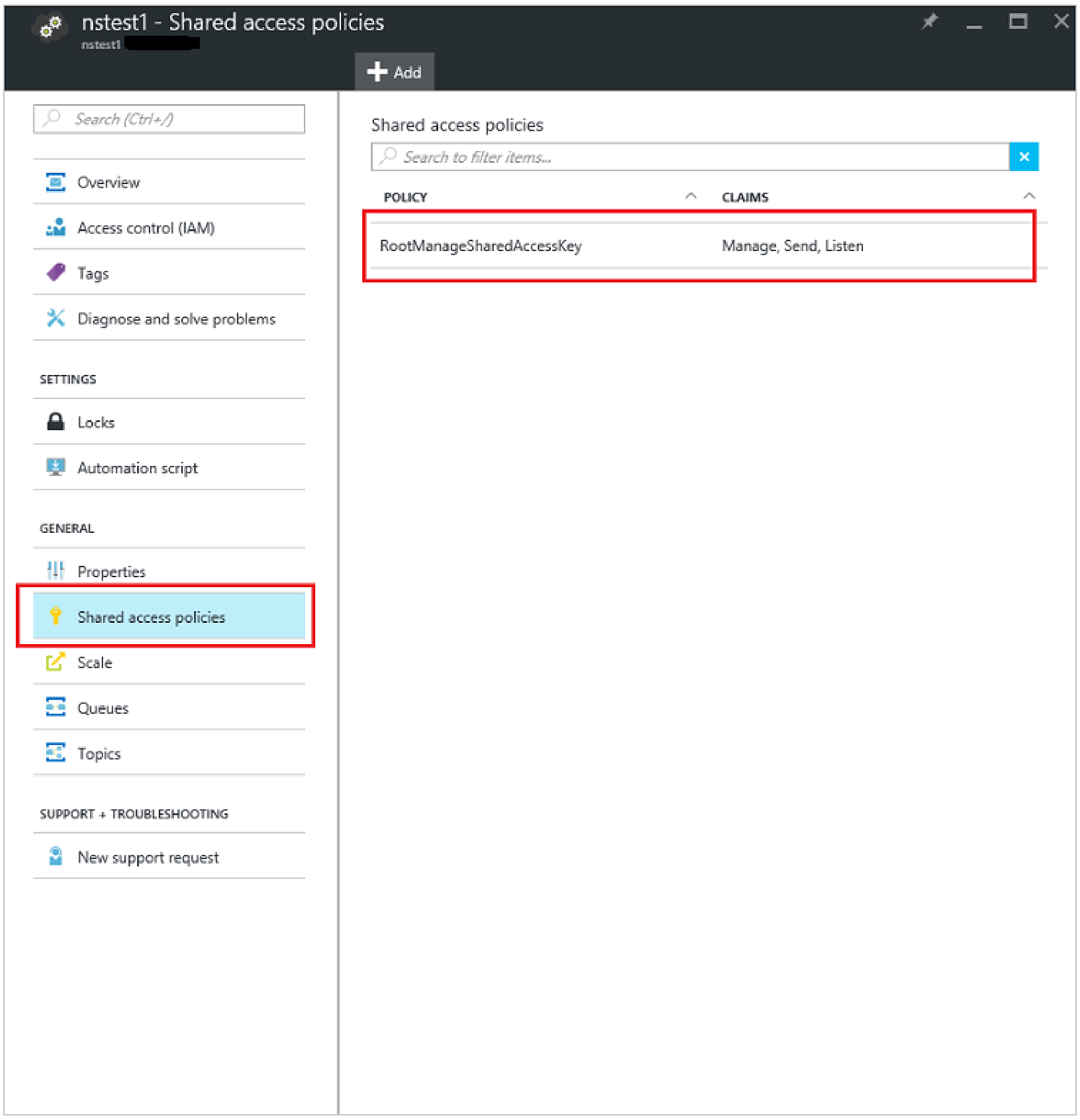
* Resource group
[✓ Create new] Use existing
MyRG ✓

* Location
West US

8. Click the **Create** button. The system now creates your namespace and enables it. You might have to wait several minutes as the system provisions resources for your account.

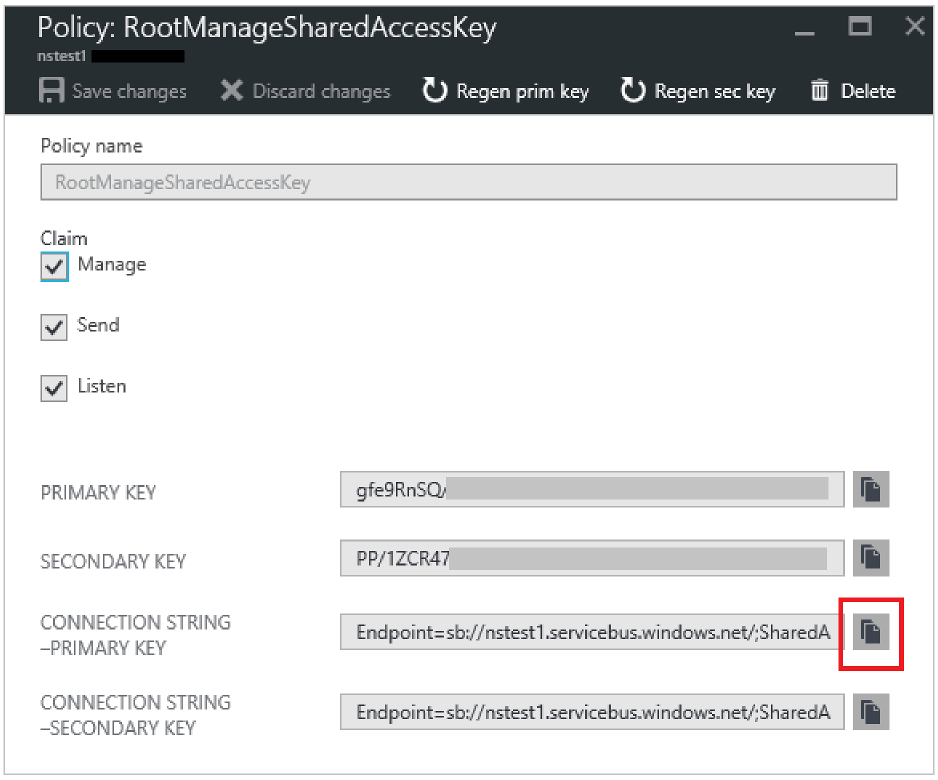
Obtain the credentials

1. In the list of namespaces, click the newly created namespace name.
2. In the **Service Bus namespace** blade, click **Shared access policies**.
3. In the **Shared access policies** blade, click **RootManageSharedAccessKey**.



POLICY	CLAIMS
RootManageSharedAccessKey	Manage, Send, Listen

4. In the **Policy: RootManageSharedAccessKey** blade, click the copy button next to **Connection string–primary key**, to copy the connection string to your clipboard for later use.


Policy: RootManageSharedAccessKey

nstest1 [redacted]

Save changes · Discard changes · Regen prim key · Regen sec key · Delete

Policy name: RootManageSharedAccessKey

Claim
- [✓] Manage
- [✓] Send
- [✓] Listen

PRIMARY KEY	gfe9RnSQ/
SECONDARY KEY	PP/1ZCR47
CONNECTION STRING –PRIMARY KEY	Endpoint=sb://nstest1.servicebus.windows.net/;SharedA
CONNECTION STRING –SECONDARY KEY	Endpoint=sb://nstest1.servicebus.windows.net/;SharedA

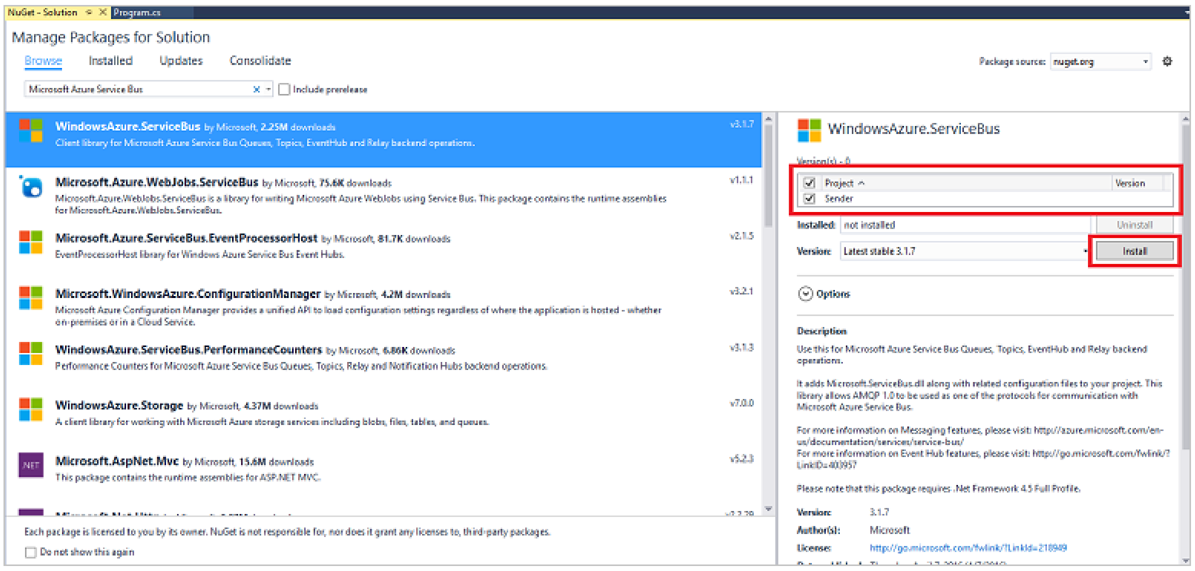
Configure the application to use Service Bus

When you create an application that uses Service Bus, you must add a reference to the Service Bus assembly and include the corresponding namespaces. The easiest way to do this is to download the appropriate NuGet package.

Get the Service Bus NuGet package

The Service Bus NuGet package is the easiest way to get the Service Bus API and to configure your application with all the necessary Service Bus dependencies. To install the Service Bus NuGet package in your project, do the following:

1. In Solution Explorer, right-click **References**, then click **Manage NuGet Packages**.
2. Search for "Service Bus" and select the **Microsoft Azure Service Bus** item. Click **Install** to complete the installation, then close the following dialog box:


NuGet - Solution | Program.cs

Manage Packages for Solution

Browse · Installed · Updates · Consolidate

Microsoft Azure Service Bus · [ ] Include prerelease · Package source: nugget.org

WindowsAzure.ServiceBus by Microsoft, 2.25M downloads — Client library for Microsoft Azure Service Bus Queues, Topics, EventHub and Relay backend operations.	v3.1.7
Microsoft.Azure.WebJobs.ServiceBus by Microsoft, 75.6K downloads — Microsoft.Azure.WebJobs.ServiceBus is a library for writing Microsoft Azure WebJobs using Service Bus. This package contains the runtime assemblies for Microsoft.Azure.WebJobs.ServiceBus.	v1.1.1
Microsoft.Azure.ServiceBus.EventProcessorHost by Microsoft, 81.7K downloads — EventProcessorHost library for Windows Azure Service Bus Event Hubs.	v2.1.5
Microsoft.WindowsAzure.ConfigurationManager by Microsoft, 4.2M downloads — Microsoft Azure Configuration Manager provides a unified API to load configuration settings regardless of where the application is hosted - whether on-premises or in a Cloud Service.	v3.2.1
WindowsAzure.ServiceBus.PerformanceCounters by Microsoft, 6.86K downloads — Performance Counters for Microsoft Azure Service Bus Queues, Topics, Relay and Notification Hubs backend operations.	v3.1.3
WindowsAzure.Storage by Microsoft, 4.37M downloads — A client library for working with Microsoft Azure storage services including blobs, files, tables, and queues.	v7.0.0
Microsoft.AspNet.Mvc by Microsoft, 15.6M downloads — This package contains the runtime assemblies for ASP.NET MVC.	v5.2.3

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.

[ ] Do not show this again

WindowsAzure.ServiceBus

Version(s) - 0

[✓] Project	Version
[✓] Sender	

Installed: not installed · Uninstall

Version: Latest stable 3.1.7 · Install

Options

Description

Use this for Microsoft Azure Service Bus Queues, Topics, EventHub and Relay backend operations.

It adds Microsoft.ServiceBus.dll along with related configuration files to your project. This library allows AMQP 1.0 to be used as one of the protocols for communication with Microsoft Azure Service Bus.

For more information on Messaging features, please visit: http://azure.microsoft.com/en-us/documentation/services/service-bus/
For more information on Event Hub features, please visit: http://go.microsoft.com/fwlink/?LinkID=403957

Please note that this package requires .Net Framework 4.5 Full Profile.

Version: 3.1.7
Author(s): Microsoft
License: http://go.microsoft.com/fwlink/?LinkId=218949

You are now ready to write code for Service Bus.

Create a Service Bus connection string

Service Bus uses a connection string to store endpoints and credentials. You can put your connection string in a configuration file, rather than hard-coding it:

- When using Azure services, it is recommended that you store your connection string using the Azure service configuration system (.csdef and .cscfg files).
- When using Azure websites or Azure Virtual Machines, it is recommended that you store your connection string using the .NET configuration system (for example, the Web.config file).

In both cases, you can retrieve your connection string using the `CloudConfigurationManager.GetSetting` method, as shown later in this article.

Configure your connection string

The service configuration mechanism enables you to dynamically change configuration settings from the Azure portal without redeploying your application. For example, add a `Setting` label to your service definition (**.csdef**) file, as shown in the next example.

```
<ServiceDefinition name="Azure1">
...
    <WebRole name="MyRole" vmsize="Small">
        <ConfigurationSettings>
            <Setting name="Microsoft.ServiceBus.ConnectionString" />
        </ConfigurationSettings>
    </WebRole>
...
</ServiceDefinition>
```

You then specify values in the service configuration (.cscfg) file.

```
<ServiceConfiguration serviceName="Azure1">
...
    <Role name="MyRole">
        <ConfigurationSettings>
            <Setting name="Microsoft.ServiceBus.ConnectionString"

value="Endpoint=sb://yourServiceNamespace.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessK
ey;SharedAccessKey=yourKey" />
        </ConfigurationSettings>
    </Role>
...
</ServiceConfiguration>
```

Use the Shared Access Signature (SAS) key name and key values retrieved from the portal as described previously.

Configure your connection string when using Azure websites or Azure Virtual Machines

When using websites or Virtual Machines, it is recommended that you use the .NET configuration system (for example, Web.config). You store the connection string using the `<appSettings>` element.

```
<configuration>
    <appSettings>
        <add key="Microsoft.ServiceBus.ConnectionString"

value="Endpoint=sb://yourServiceNamespace.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessK
ey;SharedAccessKey=yourKey" />
    </appSettings>
</configuration>
```

Use the SAS name and key values that you retrieved from the Azure portal, as described previously.

Create a topic

You can perform management operations for Service Bus topics and subscriptions using the NamespaceManager class. This class provides methods to create, enumerate, and delete topics.

The following example constructs a `NamespaceManager` object using the Azure `CloudConfigurationManager` class with a connection string consisting of the base address of a Service Bus namespace and the appropriate SAS credentials with permissions to manage it. This connection string is of the following form:

```
Endpoint=sb://<yourNamespace>.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAcc
essKey=<yourKey>
```

Use the following example, given the configuration settings in the previous section.

```
// Create the topic if it does not exist already.
string connectionString =
    CloudConfigurationManager.GetSetting("Microsoft.ServiceBus.ConnectionString");

var namespaceManager =
    NamespaceManager.CreateFromConnectionString(connectionString);

if (!namespaceManager.TopicExists("TestTopic"))
{
    namespaceManager.CreateTopic("TestTopic");
}
```

There are overloads of the CreateTopic method that enable you to set properties of the topic; for example, to set the default time-to-live (TTL) value to be applied to messages sent to the topic. These settings are applied by using the TopicDescription class. The following example shows how to create a topic named **TestTopic** with a maximum size of 5 GB and a default message TTL of 1 minute.

```
// Configure Topic Settings.
TopicDescription td = new TopicDescription("TestTopic");
td.MaxSizeInMegabytes = 5120;
td.DefaultMessageTimeToLive = new TimeSpan(0, 1, 0);

// Create a new Topic with custom settings.
string connectionString =
    CloudConfigurationManager.GetSetting("Microsoft.ServiceBus.ConnectionString");

var namespaceManager =
    NamespaceManager.CreateFromConnectionString(connectionString);

if (!namespaceManager.TopicExists("TestTopic"))
{
    namespaceManager.CreateTopic(td);
}
```

NOTE

You can use the TopicExists method on NamespaceManager objects to check whether a topic with a specified name already exists within a namespace.

Create a subscription

You can also create topic subscriptions using the NamespaceManager class. Subscriptions are named and can have an optional filter that restricts the set of messages passed to the subscription's virtual queue.

IMPORTANT

In order for messages to be received by a subscription, you must create that subscription before sending any messages to the topic. If there are no subscriptions to a topic, the topic discards those messages.

Create a subscription with the default (MatchAll) filter

If no filter is specified when a new subscription is created, the **MatchAll** filter is the default filter that is used. When you use the **MatchAll** filter, all messages published to the topic are placed in the subscription's virtual queue. The following example creates a subscription named "AllMessages" and uses the default **MatchAll** filter.

```
string connectionString =
    CloudConfigurationManager.GetSetting("Microsoft.ServiceBus.ConnectionString");

var namespaceManager =
    NamespaceManager.CreateFromConnectionString(connectionString);

if (!namespaceManager.SubscriptionExists("TestTopic", "AllMessages"))
{
    namespaceManager.CreateSubscription("TestTopic", "AllMessages");
}
```

Create subscriptions with filters

You can also set up filters that enable you to specify which messages sent to a topic should appear within a specific topic subscription.

The most flexible type of filter supported by subscriptions is the SqlFilter class, which implements a subset of SQL92. SQL filters operate on the properties of the messages that are published to the topic. For more information about the expressions that can be used with a SQL filter, see the SqlFilter.SqlExpression syntax.

The following example creates a subscription named **HighMessages** with a SqlFilter object that only selects messages that have a custom **MessageNumber** property greater than 3.

```
// Create a "HighMessages" filtered subscription.
SqlFilter highMessagesFilter =
   new SqlFilter("MessageNumber > 3");

namespaceManager.CreateSubscription("TestTopic",
   "HighMessages",
   highMessagesFilter);
```

Similarly, the following example creates a subscription named **LowMessages** with a SqlFilter that only selects messages that have a **MessageNumber** property less than or equal to 3.

```
// Create a "LowMessages" filtered subscription.
SqlFilter lowMessagesFilter =
   new SqlFilter("MessageNumber <= 3");

namespaceManager.CreateSubscription("TestTopic",
   "LowMessages",
   lowMessagesFilter);
```

Now when a message is sent to `TestTopic`, it is always delivered to receivers subscribed to the **AllMessages** topic subscription, and selectively delivered to receivers subscribed to the **HighMessages** and **LowMessages** topic subscriptions (depending on the message content).

Send messages to a topic

To send a message to a Service Bus topic, your application creates a TopicClient object using the connection string.

The following code demonstrates how to create a TopicClient object for the **TestTopic** topic created earlier using the CreateFromConnectionString API.

```
string connectionString =
    CloudConfigurationManager.GetSetting("Microsoft.ServiceBus.ConnectionString");

TopicClient Client =
    TopicClient.CreateFromConnectionString(connectionString, "TestTopic");

Client.Send(new BrokeredMessage());
```

Messages sent to Service Bus topics are instances of the BrokeredMessage class. **BrokeredMessage** objects have a set of standard properties (such as Label and TimeToLive), a dictionary that is used to hold custom application-specific properties, and a body of arbitrary application data. An application can set the body of the message by passing any serializable object to the constructor of the **BrokeredMessage** object, and the appropriate **DataContractSerializer** is then used to serialize the object. Alternatively, a **System.IO.Stream** object can be provided.

The following example demonstrates how to send five test messages to the **TestTopic** TopicClient object obtained in the previous code example. Note that the MessageId property value of each message varies depending on the iteration of the loop (this determines which subscriptions receive it).

```
for (int i=0; i<5; i++)
{
  // Create message, passing a string message for the body.
  BrokeredMessage message = new BrokeredMessage("Test message " + i);

  // Set additional custom app-specific property.
  message.Properties["MessageId"] = i;

  // Send message to the topic.
  Client.Send(message);
}
```

Service Bus topics support a maximum message size of 256 KB in the Standard tier and 1 MB in the Premium tier. The header, which includes the standard and custom application properties, can have a maximum size of 64 KB. There is no limit on the number of messages held in a topic but there is a cap on the total size of the messages held by a topic. This topic size is defined at creation time, with an upper limit of 5 GB. If partitioning is enabled, the upper limit is higher. For more information, see Partitioned messaging entities.

How to receive messages from a subscription

The recommended way to receive messages from a subscription is to use a SubscriptionClient object. **SubscriptionClient** objects can work in two different modes: *ReceiveAndDelete* and *PeekLock*. **PeekLock** is the default.

When using the **ReceiveAndDelete** mode, receive is a single-shot operation; that is, when Service Bus receives a read request for a message in a subscription, it marks the message as being consumed and returns it to the application. **ReceiveAndDelete** mode is the simplest model and works best for scenarios in which an 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. Because Service Bus has marked the message as consumed, when the application restarts and begins consuming messages again, it will have missed the message that was consumed prior to the crash.

In **PeekLock** mode (the default mode), the receive process becomes a two-stage operation, which makes it possible to support applications that cannot tolerate missing messages. When Service Bus receives a request, it finds the next message to be consumed, locks it to prevent other consumers 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 Complete on the received message. When Service Bus sees the **Complete** call, it marks the message as being consumed and removes it from the subscription.

The following example demonstrates how messages can be received and processed using the default **PeekLock** mode. To specify a different ReceiveMode value, you can use another overload for CreateFromConnectionString. This example uses the OnMessage callback to process messages as they arrive into the **HighMessages** subscription.

```
string connectionString =
    CloudConfigurationManager.GetSetting("Microsoft.ServiceBus.ConnectionString");

SubscriptionClient Client =
    SubscriptionClient.CreateFromConnectionString
            (connectionString, "TestTopic", "HighMessages");

// Configure the callback options.
OnMessageOptions options = new OnMessageOptions();
options.AutoComplete = false;
options.AutoRenewTimeout = TimeSpan.FromMinutes(1);

Client.OnMessage((message) =>
{
    try
    {
        // Process message from subscription.
        Console.WriteLine("\n**High Messages**");
        Console.WriteLine("Body: " + message.GetBody<string>());
        Console.WriteLine("MessageID: " + message.MessageId);
        Console.WriteLine("Message Number: " +
            message.Properties["MessageNumber"]);

        // Remove message from subscription.
        message.Complete();
    }
    catch (Exception)
    {
        // Indicates a problem, unlock message in subscription.
        message.Abandon();
    }
}, options);
```

This example configures the OnMessage callback using an OnMessageOptions object. AutoComplete is set to **false** to enable manual control of when to call Complete on the received message. AutoRenewTimeout is set to 1 minute, which causes the client to wait for up to one minute before terminating the auto-renewal feature and the client makes a new call to check for messages. This property value reduces the number of times the client makes chargeable calls that do not retrieve messages.

How to handle application crashes and unreadable messages

Service Bus provides functionality to help you gracefully recover from errors in your application or difficulties processing a message. If a receiving application is unable to process the message for some reason, then it can call the Abandon method on the received message (instead of the Complete method). This causes Service Bus to unlock the message within the subscription and make it available to be received again, either by the same consuming application or by another consuming application.

There is also a time-out associated with a message locked within the subscription, and if the application fails to process the message before the lock time-out expires (for example, if the application crashes), then Service Bus unlocks the message automatically and makes it available to be received again.

In the event that the application crashes after processing the message but before the Complete request is issued, the message will be redelivered to the application when it restarts. This is often called *At Least Once processing*; that is, each message is processed at least once but in certain situations the same message may be redelivered. If the scenario cannot tolerate duplicate processing, then application developers should add additional logic to their application to handle duplicate message delivery. This is often achieved using the MessageId property of the message, which remains constant across delivery attempts.

Delete topics and subscriptions

The following example demonstrates how to delete the topic **TestTopic** from the **HowToSample** service namespace.

```
// Delete Topic.
namespaceManager.DeleteTopic("TestTopic");
```

Deleting a topic also deletes any subscriptions that are registered with the topic. Subscriptions can also be deleted independently. The following code demonstrates how to delete a subscription named **HighMessages** from the **TestTopic** topic.

```
namespaceManager.DeleteSubscription("TestTopic", "HighMessages");
```

Next steps

Now that you've learned the basics of Service Bus topics and subscriptions, follow these links to learn more.

- Queues, topics, and subscriptions.
- Topic filters sample
- API reference for SqlFilter.
- Build a working application that sends and receives messages to and from a Service Bus queue: Service Bus brokered messaging .NET tutorial.
- Service Bus samples: Download from Azure samples or see the overview.

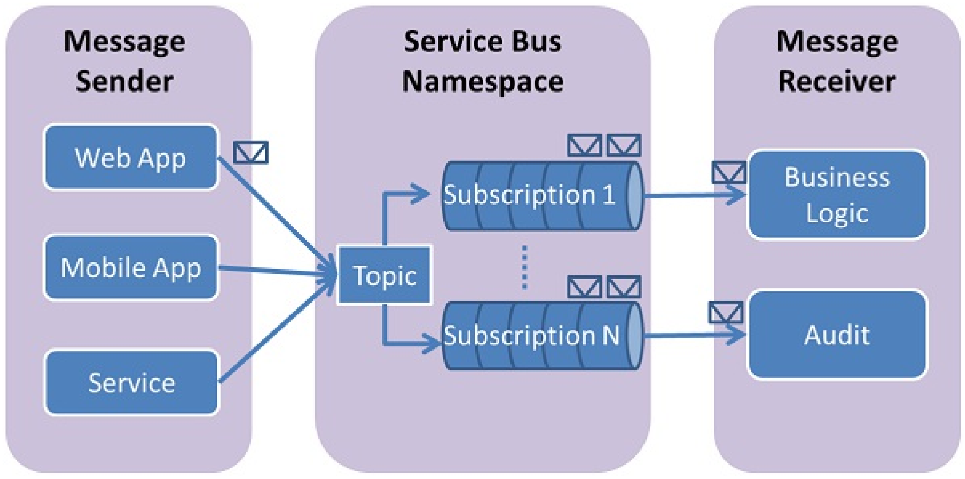
How to use Service Bus topics and subscriptions

1/17/2017 • 11 min to read • Edit on GitHub

This guide describes how to use Service Bus topics and subscriptions. The samples are written in Java and use the Azure SDK for Java. The scenarios covered include **creating topics and subscriptions**, **creating subscription filters**, **sending messages to a topic**, **receiving messages from a subscription**, and **deleting topics and subscriptions**.

What are Service Bus topics and subscriptions?

Service Bus topics and subscriptions support a *publish/subscribe* messaging communication model. When using topics and subscriptions, components of a distributed application do not communicate directly with each other; instead they exchange messages via a topic, which acts as an intermediary.



In contrast with Service Bus queues, in which each message is processed by a single consumer, topics and subscriptions provide a "one-to-many" form of communication, using a publish/subscribe pattern. It is possible to register multiple subscriptions to a topic. When a message is sent to a topic, it is then made available to each subscription to handle/process independently.

A subscription to a topic resembles a virtual queue that receives copies of the messages that were sent to the topic. You can optionally register filter rules for a topic on a per-subscription basis, which allows you to filter/restrict which messages to a topic are received by which topic subscriptions.

Service Bus topics and subscriptions enable you to scale to process a very large number of messages across a very large number of users and applications.

Create a service namespace

To begin using Service Bus topics and subscriptions in Azure, you must first create a service namespace. A namespace provides a scoping container for addressing Service Bus resources within your application.

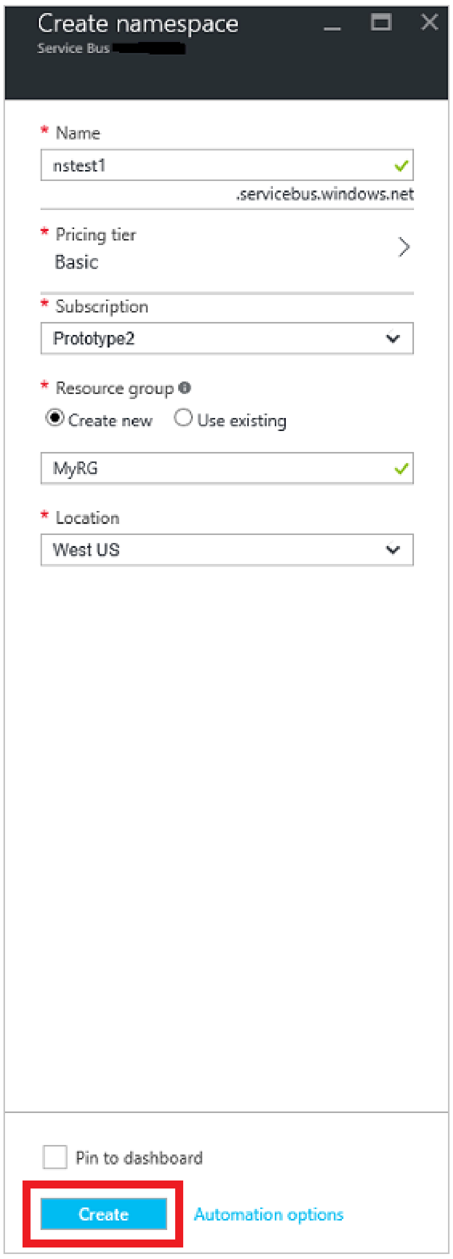
To create a namespace:

Create a service namespace

To begin using Service Bus queues in Azure, you must first create a namespace. A namespace provides a scoping container for addressing Service Bus resources within your application.

To create a namespace:

1. Log on to the Azure portal.
2. In the left navigation pane of the portal, click **New**, then click **Enterprise Integration**, and then click **Service Bus**.
3. In the **Create namespace** dialog, enter a namespace name. The system immediately checks to see if the name is available.
4. After making sure the namespace name is available, choose the pricing tier (Basic, Standard, or Premium).
5. In the **Subscription** field, choose an Azure subscription in which to create the namespace.
6. In the **Resource group** field, choose an existing resource group in which the namespace will live, or create a new one.
7. In **Location**, choose the country or region in which your namespace should be hosted.



Create namespace
Service Bus [redacted]

* Name
nstest1 ✓
.servicebus.windows.net

* Pricing tier
Basic

* Subscription
Prototype2

* Resource group
[✓ Create new] Use existing
MyRG ✓

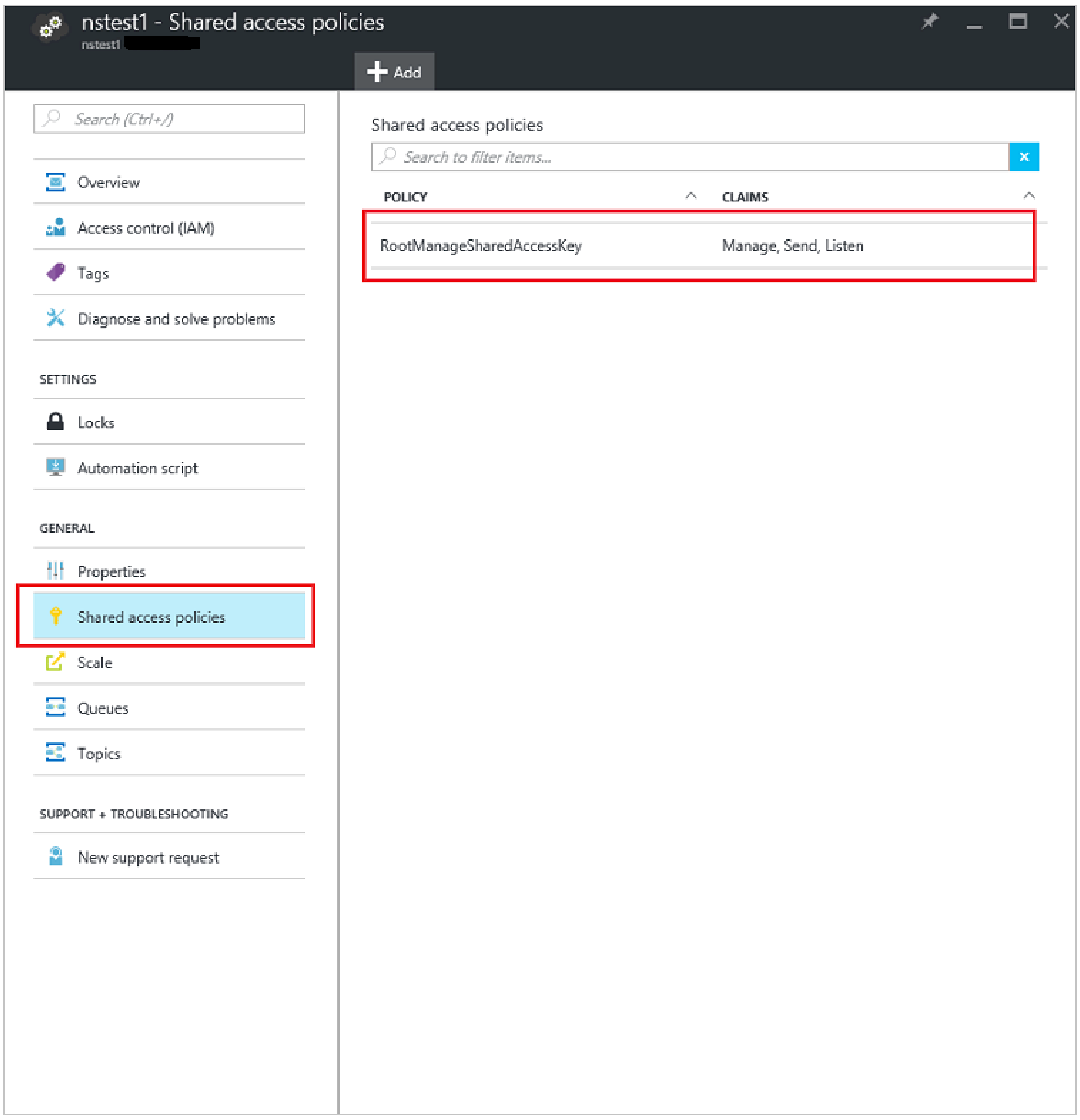
* Location
West US

☐ Pin to dashboard

8. Click **Create**. The system now creates your namespace and enables it. You might have to wait several minutes as the system provisions resources for your account.

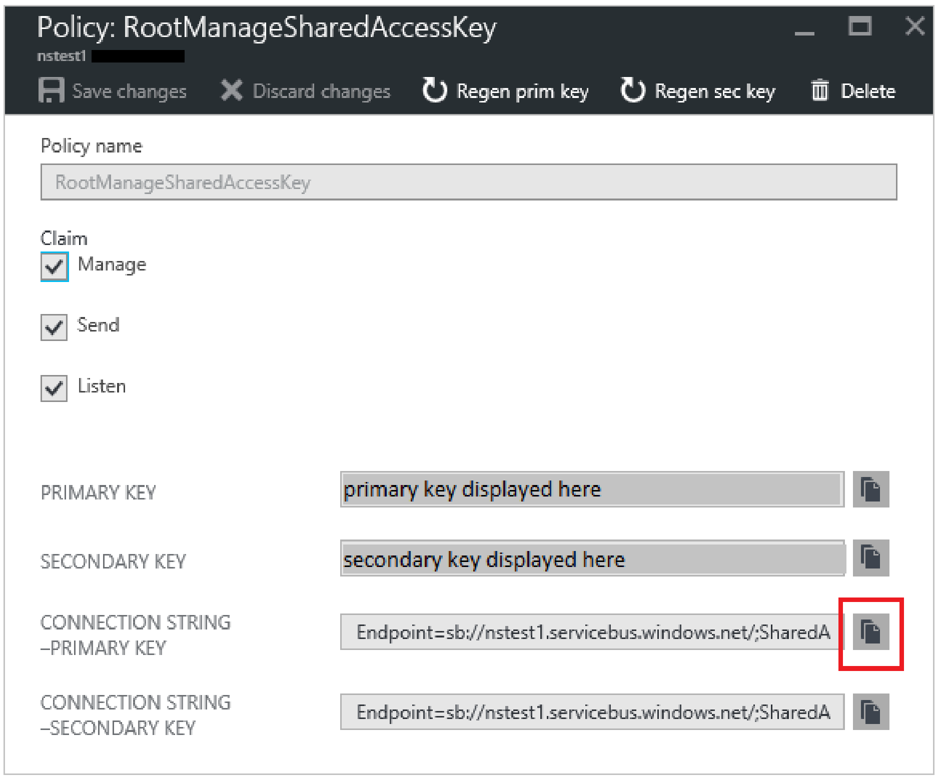
Obtain the management credentials

1. In the list of namespaces, click the newly created namespace name.
2. In the namespace blade, click **Shared access policies**.
3. In the **Shared access policies** blade, click **RootManageSharedAccessKey**.



POLICY	CLAIMS
RootManageSharedAccessKey	Manage, Send, Listen

4. In the **Policy: RootManageSharedAccessKey** blade, click the copy button next to **Connection string–primary key**, to copy the connection string to your clipboard for later use. Paste this value into Notepad or some other temporary location.


Policy: RootManageSharedAccessKey

nstest1 [redacted]

Policy name: RootManageSharedAccessKey

Claim
- [✓] Manage
- [✓] Send
- [✓] Listen

PRIMARY KEY: primary key displayed here

SECONDARY KEY: secondary key displayed here

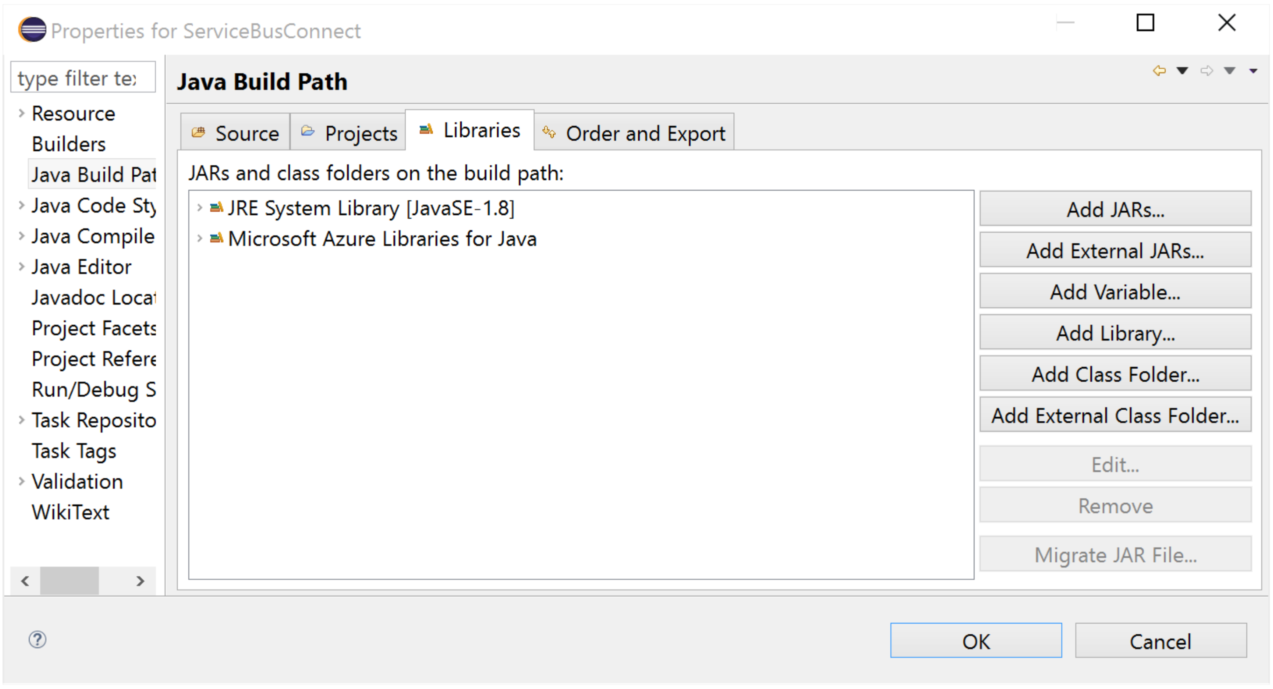
CONNECTION STRING –PRIMARY KEY: Endpoint=sb://nstest1.servicebus.windows.net/;SharedA

CONNECTION STRING –SECONDARY KEY: Endpoint=sb://nstest1.servicebus.windows.net/;SharedA

5. Repeat the previous step, copying and pasting the value of **Primary key** to a temporary location for later use.

Configure your application to use Service Bus

Make sure you have installed the Azure SDK for Java before building this sample. If you are using Eclipse, you can install the Azure Toolkit for Eclipse that includes the Azure SDK for Java. You can then add the **Microsoft Azure Libraries for Java** to your project:


Properties for ServiceBusConnect

Java Build Path

JARs and class folders on the build path:
- JRE System Library [JavaSE-1.8]
- Microsoft Azure Libraries for Java

Add the following import statements to the top of the Java file:

```
import com.microsoft.windowsazure.services.servicebus.*;
import com.microsoft.windowsazure.services.servicebus.models.*;
import com.microsoft.windowsazure.core.*;
import javax.xml.datatype.*;
```

Add the Azure Libraries for Java to your build path and include it in your project deployment assembly.

Create a topic

Management operations for Service Bus topics can be performed via the **ServiceBusContract** class. A **ServiceBusContract** object is constructed with an appropriate configuration that encapsulates the SAS token with permissions to manage it, and the **ServiceBusContract** class is the sole point of communication with Azure.

The **ServiceBusService** class provides methods to create, enumerate, and delete topics. The following example shows how a **ServiceBusService** object can be used to create a topic named `TestTopic`, with a namespace called `HowToSample`:

```
Configuration config =
    ServiceBusConfiguration.configureWithSASAuthentication(
      "HowToSample",
      "RootManageSharedAccessKey",
      "SAS_key_value",
      ".servicebus.windows.net"
      );

ServiceBusContract service = ServiceBusService.create(config);
TopicInfo topicInfo = new TopicInfo("TestTopic");
try
{
    CreateTopicResult result = service.createTopic(topicInfo);
}
catch (ServiceException e) {
    System.out.print("ServiceException encountered: ");
    System.out.println(e.getMessage());
    System.exit(-1);
}
```

There are methods on **TopicInfo** that enable properties of the topic to be set (for example: to set the default time-to-live (TTL) value to be applied to messages sent to the topic). The following example shows how to create a topic named `TestTopic` with a maximum size of 5 GB:

```
long maxSizeInMegabytes = 5120;
TopicInfo topicInfo = new TopicInfo("TestTopic");
topicInfo.setMaxSizeInMegabytes(maxSizeInMegabytes);
CreateTopicResult result = service.createTopic(topicInfo);
```

Note that you can use the **listTopics** method on **ServiceBusContract** objects to check if a topic with a specified name already exists within a service namespace.

Create subscriptions

Subscriptions to topics are also created with the **ServiceBusService** class. Subscriptions are named and can have an optional filter that restricts the set of messages passed to the subscription's virtual queue.

Create a subscription with the default (MatchAll) filter

The **MatchAll** filter is the default filter that is used if no filter is specified when a new subscription is created. When the **MatchAll** filter is used, all messages published to the topic are placed in the subscription's virtual queue. The following example creates a subscription named "AllMessages" and uses the default **MatchAll** filter.

```
SubscriptionInfo subInfo = new SubscriptionInfo("AllMessages");
CreateSubscriptionResult result =
    service.createSubscription("TestTopic", subInfo);
```

Create subscriptions with filters

You can also create filters that enable you to scope which messages sent to a topic should show up within a specific topic subscription.

The most flexible type of filter supported by subscriptions is the SqlFilter, which implements a subset of SQL92. SQL filters operate on the properties of the messages that are published to the topic. For more details about the expressions that can be used with a SQL filter, review the SqlFilter.SqlExpression syntax.

The following example creates a subscription named `HighMessages` with a SqlFilter object that only selects messages that have a custom **MessageNumber** property greater than 3:

```
// Create a "HighMessages" filtered subscription
SubscriptionInfo subInfo = new SubscriptionInfo("HighMessages");
CreateSubscriptionResult result = service.createSubscription("TestTopic", subInfo);
RuleInfo ruleInfo = new RuleInfo("myRuleGT3");
ruleInfo = ruleInfo.withSqlExpressionFilter("MessageNumber > 3");
CreateRuleResult ruleResult = service.createRule("TestTopic", "HighMessages", ruleInfo);
// Delete the default rule, otherwise the new rule won't be invoked.
service.deleteRule("TestTopic", "HighMessages", "$Default");
```

Similarly, the following example creates a subscription named `LowMessages` with a SqlFilter object that only selects messages that have a **MessageNumber** property less than or equal to 3:

```
// Create a "LowMessages" filtered subscription
SubscriptionInfo subInfo = new SubscriptionInfo("LowMessages");
CreateSubscriptionResult result = service.createSubscription("TestTopic", subInfo);
RuleInfo ruleInfo = new RuleInfo("myRuleLE3");
ruleInfo = ruleInfo.withSqlExpressionFilter("MessageNumber <= 3");
CreateRuleResult ruleResult = service.createRule("TestTopic", "LowMessages", ruleInfo);
// Delete the default rule, otherwise the new rule won't be invoked.
service.deleteRule("TestTopic", "LowMessages", "$Default");
```

When a message is now sent to `TestTopic`, it will always be delivered to receivers subscribed to the `AllMessages` subscription, and selectively delivered to receivers subscribed to the `HighMessages` and `LowMessages` subscriptions (depending upon the message content).

Send messages to a topic

To send a message to a Service Bus topic, your application obtains a **ServiceBusContract** object. The following code demonstrates how to send a message for the `TestTopic` topic created previously within the `HowToSample` namespace:

```
BrokeredMessage message = new BrokeredMessage("MyMessage");
service.sendTopicMessage("TestTopic", message);
```

Messages sent to Service Bus Topics are instances of the BrokeredMessage class. BrokeredMessage* objects have a set of standard methods (such as **setLabel** and **TimeToLive**), a dictionary that is used to hold custom application-specific properties, and a body of arbitrary application data. An application can set the body of the message by passing any serializable object into the constructor of the BrokeredMessage, and the appropriate

DataContractSerializer will then be used to serialize the object. Alternatively, a **java.io.InputStream** can be provided.

The following example demonstrates how to send five test messages to the `TestTopic` **MessageSender** we obtained in the code snippet above. Note how the **MessageNumber** property value of each message varies on the iteration of the loop (this will determine which subscriptions receive it):

```
for (int i=0; i<5; i++)  {
// Create message, passing a string message for the body
BrokeredMessage message = new BrokeredMessage("Test message " + i);
// Set some additional custom app-specific property
message.setProperty("MessageNumber", i);
// Send message to the topic
service.sendTopicMessage("TestTopic", message);
}
```

Service Bus topics support a maximum message size of 256 KB in the Standard tier and 1 MB in the Premium tier. The header, which includes the standard and custom application properties, can have a maximum size of 64 KB. There is no limit on the number of messages held in a topic but there is a cap on the total size of the messages held by a topic. This topic size is defined at creation time, with an upper limit of 5 GB.

How to receive messages from a subscription

To receive messages from a subscription, use a **ServiceBusContract** object. Received messages can work in two different modes: **ReceiveAndDelete** and **PeekLock**.

When using the **ReceiveAndDelete** mode, receive is a single-shot operation - that is, when Service Bus receives a read request for a message, it marks the message as being consumed and returns it to the application. **ReceiveAndDelete** mode is the simplest model and works best for scenarios in which an 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. Because Service Bus will have marked the message as being consumed, then when the application restarts and begins consuming messages again, it will have missed the message that was consumed prior to the crash.

In **PeekLock** mode, receive becomes a two stage operation, which makes it possible to support applications that cannot tolerate missing messages. When Service Bus receives a request, it finds the next message to be consumed, locks it to prevent other consumers 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 **Delete** on the received message. When Service Bus sees the **Delete** call, it will mark the message as being consumed and remove it from the topic.

The example below demonstrates how messages can be received and processed using **PeekLock** mode (not the default mode). The example below performs a loop and processes messages in the "HighMessages" subscription and then exits when there are no more messages (alternatively, it could be set to wait for new messages).

```
try
{
    ReceiveMessageOptions opts = ReceiveMessageOptions.DEFAULT;
    opts.setReceiveMode(ReceiveMode.PEEK_LOCK);

    while(true)  {
        ReceiveSubscriptionMessageResult  resultSubMsg =
            service.receiveSubscriptionMessage("TestTopic", "HighMessages", opts);
        BrokeredMessage message = resultSubMsg.getValue();
        if (message != null && message.getMessageId() != null)
        {
            System.out.println("MessageID: " + message.getMessageId());
            // Display the topic message.
            System.out.print("From topic: ");
            byte[] b = new byte[200];
            String s = null;
            int numRead = message.getBody().read(b);
            while (-1 != numRead)
            {
                s = new String(b);
                s = s.trim();
                System.out.print(s);
                numRead = message.getBody().read(b);
            }
            System.out.println();
            System.out.println("Custom Property: " +
                message.getProperty("MessageNumber"));
            // Delete message.
            System.out.println("Deleting this message.");
            service.deleteMessage(message);
        }
        else
        {
            System.out.println("Finishing up - no more messages.");
            break;
            // Added to handle no more messages.
            // Could instead wait for more messages to be added.
        }
    }
}
catch (ServiceException e) {
    System.out.print("ServiceException encountered: ");
    System.out.println(e.getMessage());
    System.exit(-1);
}
catch (Exception e) {
    System.out.print("Generic exception encountered: ");
    System.out.println(e.getMessage());
    System.exit(-1);
}
```

How to handle application crashes and unreadable messages

Service Bus provides functionality to help you gracefully recover from errors in your application or difficulties processing a message. If a receiver application is unable to process the message for some reason, then it can call the **unlockMessage** method on the received message (instead of the **deleteMessage** method). This will cause Service Bus to unlock the message within the topic and make it available to be received again, either by the same consuming application or by another consuming application.

There is also a timeout associated with a message locked within the topic, 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 automatically and make it available to be received again.

In the event that the application crashes after processing the message but before the **deleteMessage** request is issued, then 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 application developers should add additional logic to their application to handle duplicate message delivery. This is often achieved using the **getMessageId** method of the message, which will remain constant across delivery attempts.

Delete topics and subscriptions

The primary way to delete topics and subscriptions is to use a **ServiceBusContract** object. Deleting a topic will also delete any subscriptions that are registered with the topic. Subscriptions can also be deleted independently.

```
// Delete subscriptions
service.deleteSubscription("TestTopic", "AllMessages");
service.deleteSubscription("TestTopic", "HighMessages");
service.deleteSubscription("TestTopic", "LowMessages");

// Delete a topic
service.deleteTopic("TestTopic");
```

Next Steps

Now that you've learned the basics of Service Bus queues, see Service Bus queues, topics, and subscriptions for more information.

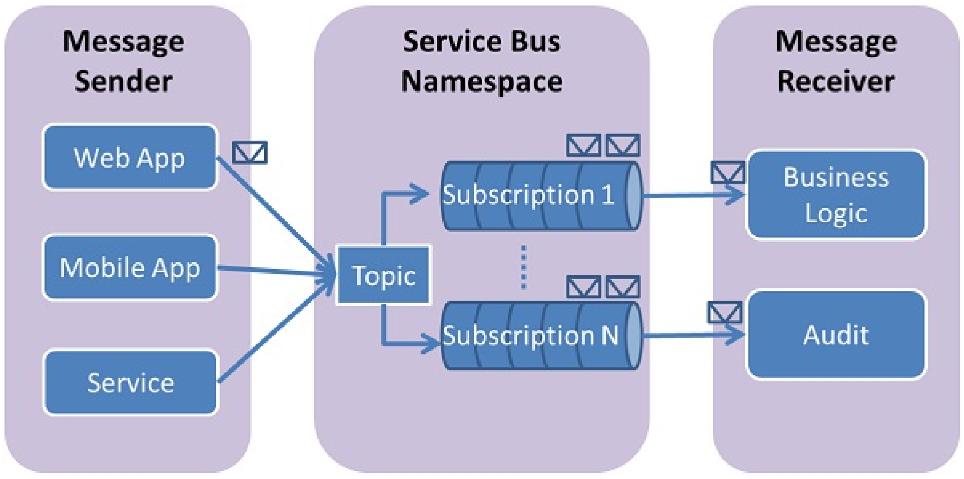
How to Use Service Bus topics and subscriptions

1/17/2017 • 12 min to read • Edit on GitHub

This guide describes how to use Service Bus topics and subscriptions from Node.js applications. The scenarios covered include **creating topics and subscriptions**, **creating subscription filters**, **sending messages** to a topic, **receiving messages from a subscription**, and **deleting topics and subscriptions**. For more information about topics and subscriptions, see the Next steps section.

What are Service Bus topics and subscriptions?

Service Bus topics and subscriptions support a *publish/subscribe* messaging communication model. When using topics and subscriptions, components of a distributed application do not communicate directly with each other; instead they exchange messages via a topic, which acts as an intermediary.



In contrast with Service Bus queues, in which each message is processed by a single consumer, topics and subscriptions provide a "one-to-many" form of communication, using a publish/subscribe pattern. It is possible to register multiple subscriptions to a topic. When a message is sent to a topic, it is then made available to each subscription to handle/process independently.

A subscription to a topic resembles a virtual queue that receives copies of the messages that were sent to the topic. You can optionally register filter rules for a topic on a per-subscription basis, which enables you to filter or restrict which messages to a topic are received by which topic subscriptions.

Service Bus topics and subscriptions enable you to scale and process a very large number of messages across many users and applications.

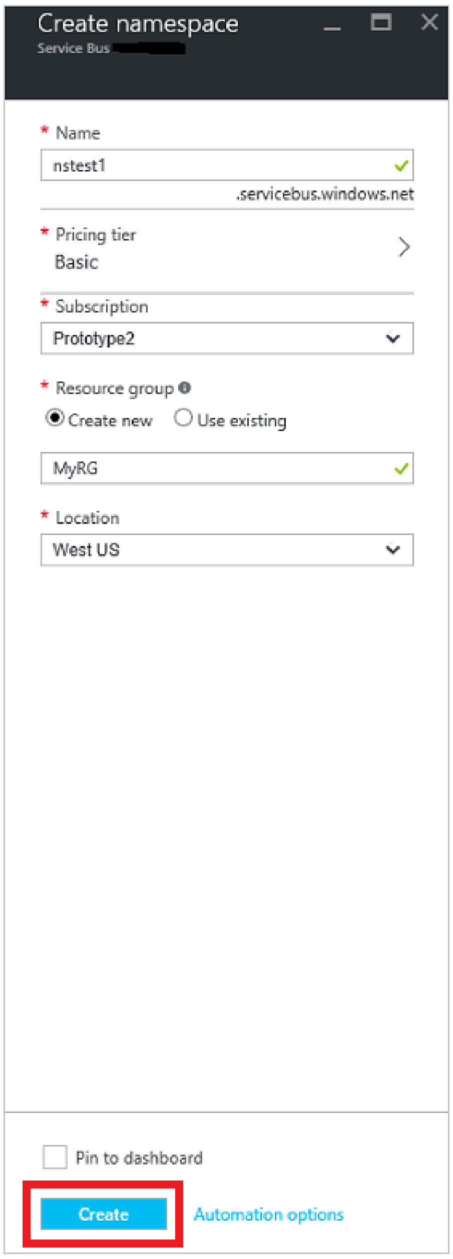
Create a namespace

To begin using Service Bus topics and subscriptions in Azure, you must first create a *service namespace*. A namespace provides a scoping container for addressing Service Bus resources within your application.

To create a namespace:

1. Log on to the Azure portal.
2. In the left navigation pane of the portal, click **New**, then click **Enterprise Integration**, and then click **Service Bus**.
3. In the **Create namespace** dialog, enter a namespace name. The system immediately checks to see if the name is available.

4. After making sure the namespace name is available, choose the pricing tier (Basic, Standard, or Premium).
5. In the **Subscription** field, choose an Azure subscription in which to create the namespace.
6. In the **Resource group** field, choose an existing resource group in which the namespace will live, or create a new one.
7. In **Location**, choose the country or region in which your namespace should be hosted.

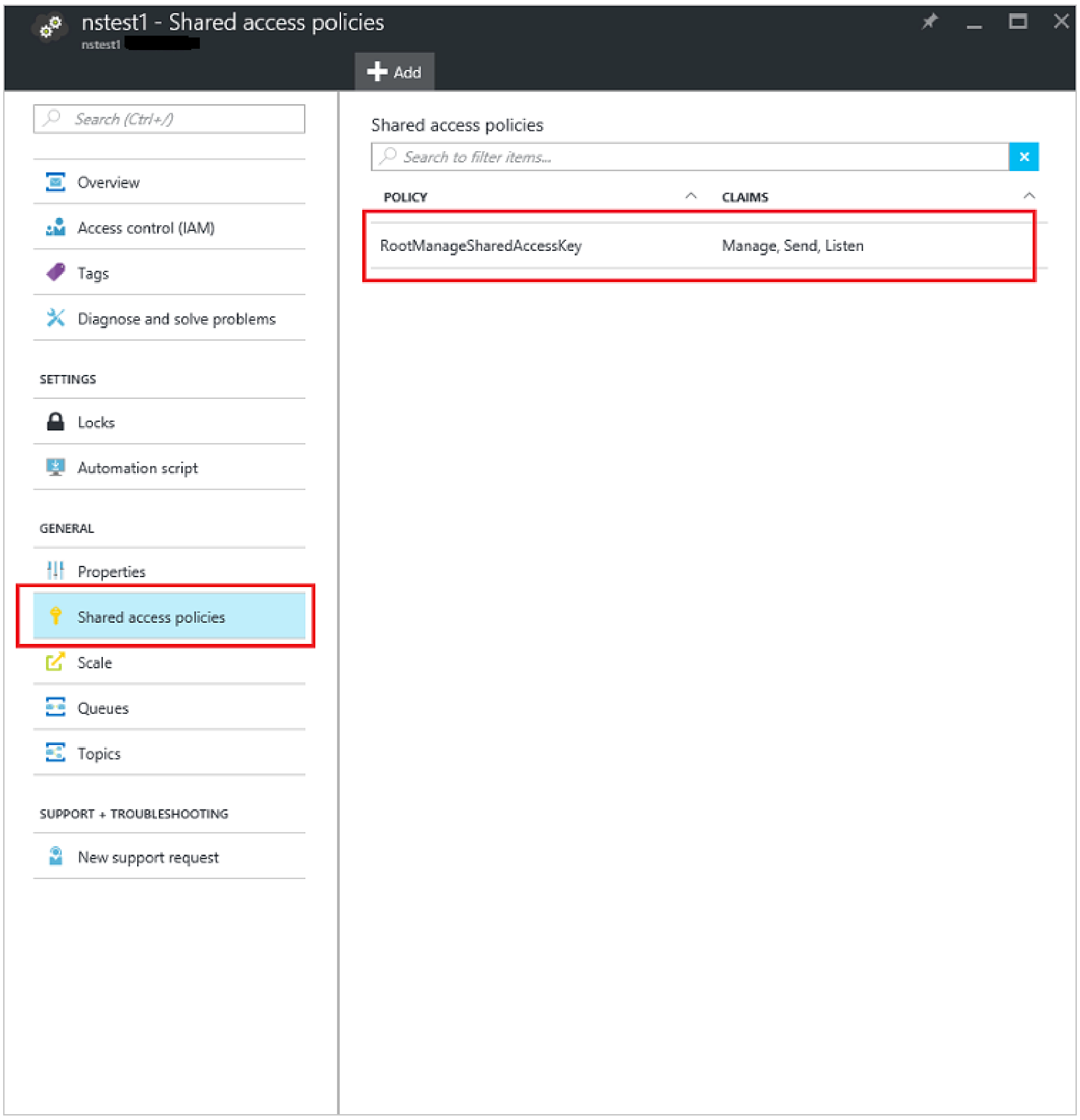


* Name: nstest1 .servicebus.windows.net
* Pricing tier: Basic
* Subscription: Prototype2
* Resource group: (●) Create new ( ) Use existing
MyRG
* Location: West US
[ ] Pin to dashboard

8. Click the **Create** button. The system now creates your namespace and enables it. You might have to wait several minutes as the system provisions resources for your account.

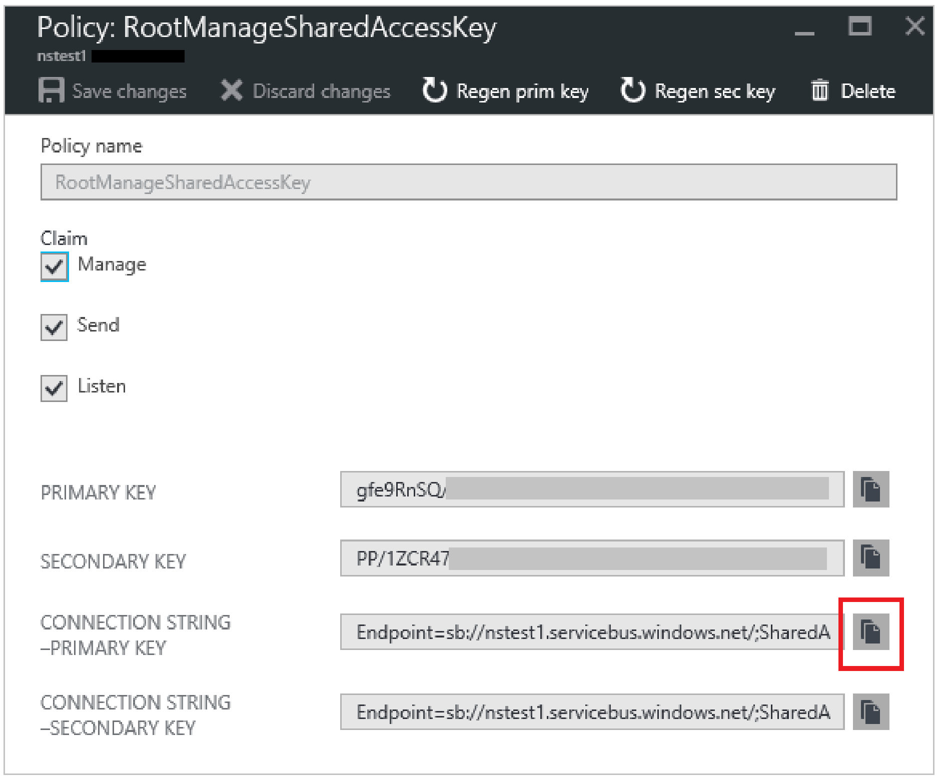
Obtain the credentials

1. In the list of namespaces, click the newly created namespace name.
2. In the **Service Bus namespace** blade, click **Shared access policies**.
3. In the **Shared access policies** blade, click **RootManageSharedAccessKey**.



POLICY	CLAIMS
RootManageSharedAccessKey	Manage, Send, Listen

4. In the **Policy: RootManageSharedAccessKey** blade, click the copy button next to **Connection string–primary key**, to copy the connection string to your clipboard for later use.



Policy: RootManageSharedAccessKey

nstest1 [redacted]

Save changes | Discard changes | Regen prim key | Regen sec key | Delete

Policy name

RootManageSharedAccessKey

Claim

✓ Manage

✓ Send

✓ Listen

PRIMARY KEY: gfe9RnSQ/[redacted]

SECONDARY KEY: PP/1ZCR47[redacted]

CONNECTION STRING –PRIMARY KEY: Endpoint=sb://nstest1.servicebus.windows.net/;SharedA

CONNECTION STRING –SECONDARY KEY: Endpoint=sb://nstest1.servicebus.windows.net/;SharedA

Create a Node.js application

Create a blank Node.js application. For instructions on creating a Node.js application, see Create and deploy a Node.js application to an Azure Web Site, Node.js Cloud Service using Windows PowerShell, or Web Site with WebMatrix.

Configure your application to use Service Bus

To use Service Bus, download the Node.js Azure package. This package includes a set of libraries that communicate with the Service Bus REST services.

Use Node Package Manager (NPM) to obtain the package

1. Use a command-line interface such as **PowerShell** (Windows,) **Terminal** (Mac,) or **Bash** (Unix), navigate to the folder where you created your sample application.
2. Type **npm install azure** in the command window, which should result in the following output:

```
    azure@0.7.5 node_modules\azure
├── dateformat@1.0.2-1.2.3
├── xmlbuilder@0.4.2
├── node-uuid@1.2.0
├── mime@1.2.9
├── underscore@1.4.4
├── validator@1.1.1
├── tunnel@0.0.2
├── wns@0.5.3
├── xml2js@0.2.7 (sax@0.5.2)
└── request@2.21.0 (json-stringify-safe@4.0.0, forever-agent@0.5.0, aws-sign@0.3.0, tunnel-agent@0.3.0,
oauth-sign@0.3.0, qs@0.6.5, cookie-jar@0.3.0, node-uuid@1.4.0, http-signature@0.9.11, form-data@0.0.8,
hawk@0.13.1)
```

3. You can manually run the **ls** command to verify that a **node_modules** folder was created. Inside that folder find the **azure** package, which contains the libraries you need to access Service Bus topics.

Import the module

Using Notepad or another text editor, add the following to the top of the **server.js** file of the application:

```
var azure = require('azure');
```

Set up a Service Bus connection

The Azure module reads the environment variables AZURE_SERVICEBUS_NAMESPACE and AZURE_SERVICEBUS_ACCESS_KEY for information required to connect to Service Bus. If these environment variables are not set, you must specify the account information when calling **createServiceBusService**.

For an example of setting the environment variables in a configuration file for an Azure Cloud Service, see Node.js Cloud Service with Storage.

For an example of setting the environment variables in the Azure classic portal for an Azure Website, see Node.js Web Application with Storage.

Create a topic

The **ServiceBusService** object enables you to work with topics. The following code creates a **ServiceBusService** object. Add it near the top of the **server.js** file, after the statement to import the azure module:

```
var serviceBusService = azure.createServiceBusService();
```

By calling **createTopicIfNotExists** on the **ServiceBusService** object, the specified topic will be returned (if it exists,) or a new topic with the specified name will be created. The following code uses **createTopicIfNotExists** to create or connect to the topic named 'MyTopic':

```
serviceBusService.createTopicIfNotExists('MyTopic',function(error){
    if(!error){
        // Topic was created or exists
        console.log('topic created or exists.');
    }
});
```

createServiceBusService also supports additional options, which enable you to override default topic settings such as message time to live or maximum topic size. The following example sets the maximum topic size to 5GB with a time to live of 1 minute:

```
var topicOptions = {
        MaxSizeInMegabytes: '5120',
        DefaultMessageTimeToLive: 'PT1M'
    };

serviceBusService.createTopicIfNotExists('MyTopic', topicOptions, function(error){
    if(!error){
        // topic was created or exists
    }
});
```

Filters

Optional filtering operations can be applied to operations performed using **ServiceBusService**. Filtering operations can include logging, automatically retrying, etc. Filters are objects that implement a method with the signature:

```
function handle (requestOptions, next)
```

After performing preprocessing on the request options, the method calls `next` passing a callback with the following signature:

```
function (returnObject, finalCallback, next)
```

In this callback, and after processing the **returnObject** (the response from the request to the server), the callback needs to either invoke next if it exists to continue processing other filters or simply invoke **finalCallback** otherwise to end up the service invocation.

Two filters that implement retry logic are included with the Azure SDK for Node.js, **ExponentialRetryPolicyFilter** and **LinearRetryPolicyFilter**. The following creates a **ServiceBusService** object that uses the **ExponentialRetryPolicyFilter**:

```
var retryOperations = new azure.ExponentialRetryPolicyFilter();
var serviceBusService = azure.createServiceBusService().withFilter(retryOperations);
```

Create subscriptions

Topic subscriptions are also created with the **ServiceBusService** object. Subscriptions are named and can have an optional filter that restricts the set of messages delivered to the subscription's virtual queue.

NOTE

Subscriptions are persistent and will continue to exist until either they, or the topic they are associated with, are deleted. If your application contains logic to create a subscription, it should first check if the subscription already exists by using the **getSubscription** method.

Create a subscription with the default (MatchAll) filter

The **MatchAll** filter is the default filter that is used if no filter is specified when a new subscription is created. When the **MatchAll** filter is used, all messages published to the topic are placed in the subscription's virtual queue. The following example creates a subscription named 'AllMessages' and uses the default **MatchAll** filter.

```
serviceBusService.createSubscription('MyTopic','AllMessages',function(error){
    if(!error){
        // subscription created
    }
});
```

Create subscriptions with filters

You can also create filters that allow you to scope which messages sent to a topic should show up within a specific topic subscription.

The most flexible type of filter supported by subscriptions is the **SqlFilter**, which implements a subset of SQL92. SQL filters operate on the properties of the messages that are published to the topic. For more details about the expressions that can be used with a SQL filter, review the SqlFilter.SqlExpression syntax.

Filters can be added to a subscription by using the **createRule** method of the **ServiceBusService** object. This method allows you to add new filters to an existing subscription.

NOTE

Because the default filter is applied automatically to all new subscriptions, you must first remove the default filter or the **MatchAll** will override any other filters you may specify. You can remove the default rule by using the **deleteRule** method of the **ServiceBusService** object.

The following example creates a subscription named `HighMessages` with a **SqlFilter** that only selects messages that have a custom **messagenumber** property greater than 3:

```
serviceBusService.createSubscription('MyTopic', 'HighMessages', function (error){
    if(!error){
        // subscription created
        rule.create();
    }
});
var rule={
    deleteDefault: function(){
        serviceBusService.deleteRule('MyTopic',
            'HighMessages',
            azure.Constants.ServiceBusConstants.DEFAULT_RULE_NAME,
            rule.handleError);
    },
    create: function(){
        var ruleOptions = {
            sqlExpressionFilter: 'messagenumber > 3'
        };
        rule.deleteDefault();
        serviceBusService.createRule('MyTopic',
            'HighMessages',
            'HighMessageFilter',
            ruleOptions,
            rule.handleError);
    },
    handleError: function(error){
        if(error){
            console.log(error)
        }
    }
}
```

Similarly, the following example creates a subscription named `LowMessages` with a **SqlFilter** that only selects messages that have a **messagenumber** property less than or equal to 3:

```
serviceBusService.createSubscription('MyTopic', 'LowMessages', function (error){
    if(!error){
        // subscription created
        rule.create();
    }
});
var rule={
    deleteDefault: function(){
        serviceBusService.deleteRule('MyTopic',
            'LowMessages',
            azure.Constants.ServiceBusConstants.DEFAULT_RULE_NAME,
            rule.handleError);
    },
    create: function(){
        var ruleOptions = {
            sqlExpressionFilter: 'messagenumber <= 3'
        };
        rule.deleteDefault();
        serviceBusService.createRule('MyTopic',
            'LowMessages',
            'LowMessageFilter',
            ruleOptions,
            rule.handleError);
    },
    handleError: function(error){
        if(error){
            console.log(error)
        }
    }
}
```

When a message is now sent to `MyTopic`, it will always be delivered to receivers subscribed to the `AllMessages` topic subscription, and selectively delivered to receivers subscribed to the `HighMessages` and `LowMessages` topic subscriptions (depending upon the message content).

How to send messages to a topic

To send a message to a Service Bus topic, your application must use the **sendTopicMessage** method of the **ServiceBusService** object. Messages sent to Service Bus topics are **BrokeredMessage** objects. **BrokeredMessage** objects have a set of standard properties (such as **Label** and **TimeToLive**), a dictionary that is used to hold custom application-specific properties, and a body of string data. An application can set the body of the message by passing a string value to the **sendTopicMessage** and any required standard properties will be populated by default values.

The following example demonstrates how to send five test messages to 'MyTopic'. Note that the **messagenumber** property value of each message varies on the iteration of the loop (this will determine which subscriptions receive it):

```
var message = {
    body: '',
    customProperties: {
        messagenumber: 0
    }
}

for (i = 0;i < 5;i++) {
    message.customProperties.messagenumber=i;
    message.body='This is Message #'+i;
    serviceBusService.sendTopicMessage(topic, message, function(error) {
      if (error) {
        console.log(error);
      }
    });
}
```

Service Bus topics support a maximum message size of 256 KB in the Standard tier and 1 MB in the Premium tier. The header, which includes the standard and custom application properties, can have a maximum size of 64 KB. There is no limit on the number of messages held in a topic but there is a cap on the total size of the messages held by a topic. This topic size is defined at creation time, with an upper limit of 5 GB.

Receive messages from a subscription

Messages are received from a subscription using the **receiveSubscriptionMessage** method on the **ServiceBusService** object. By default, messages are deleted from the subscription as they are read; however, you can read (peek) and lock the message without deleting it from the subscription by setting the optional parameter **isPeekLock** to **true**.

The default behavior of reading and deleting the message as part of the receive operation is the simplest model, and works best for scenarios in which an 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. Because Service Bus will have marked the message as being consumed, then when the application restarts and begins consuming messages again, it will have missed the message that was consumed prior to the crash.

If the **isPeekLock** parameter is set to **true**, the receive becomes a two stage operation, which makes it possible to support applications that cannot tolerate missing messages. When Service Bus receives a request, it finds the next message to be consumed, locks it to prevent other consumers 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 **deleteMessage** method and providing the message to be deleted as a parameter. The **deleteMessage** method will mark the message as being consumed and remove it from the subscription.

The following example demonstrates how messages can be received and processed using **receiveSubscriptionMessage**. The example first receives and deletes a message from the 'LowMessages' subscription, and then receives a message from the 'HighMessages' subscription using **isPeekLock** set to true. It then deletes the message using **deleteMessage**:

```
serviceBusService.receiveSubscriptionMessage('MyTopic', 'LowMessages', function(error, receivedMessage){
    if(!error){
        // Message received and deleted
        console.log(receivedMessage);
    }
});
serviceBusService.receiveSubscriptionMessage('MyTopic', 'HighMessages', { isPeekLock: true }, function(error,
lockedMessage){
    if(!error){
        // Message received and locked
        console.log(lockedMessage);
        serviceBusService.deleteMessage(lockedMessage, function (deleteError){
            if(!deleteError){
                // Message deleted
                console.log('message has been deleted.');
            }
        }
    }
});
```

How to handle application crashes and unreadable messages

Service Bus provides functionality to help you gracefully recover from errors in your application or difficulties processing a message. If a receiver application is unable to process the message for some reason, then it can call the **unlockMessage** method on the **ServiceBusService** object. This will cause Service Bus to unlock the message within the subscription and make it available to be received again, either by the same consuming application or by another consuming application.

There is also a timeout associated with a message locked within the subscription, and if the application fails to process the message before the lock timeout expires (for example, if the application crashes), then Service Bus unlocks the message automatically and makes it available to be received again.

In the event that the application crashes after processing the message but before the **deleteMessage** method is called, then 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 application developers should add additional logic to their application to handle duplicate message delivery. This is often achieved using the **MessageId** property of the message, which will remain constant across delivery attempts.

Delete topics and subscriptions

Topics and subscriptions are persistent, and must be explicitly deleted either through the Azure classic portal or programmatically. The following example demonstrates how to delete the topic named `MyTopic`:

```
serviceBusService.deleteTopic('MyTopic', function (error) {
    if (error) {
        console.log(error);
    }
});
```

Deleting a topic will also delete any subscriptions that are registered with the topic. Subscriptions can also be deleted independently. The following example shows how to delete a subscription named `HighMessages` from the `MyTopic` topic:

```
serviceBusService.deleteSubscription('MyTopic', 'HighMessages', function (error) {
    if(error) {
        console.log(error);
    }
});
```

Next Steps

Now that you've learned the basics of Service Bus topics, follow these links to learn more.

- See Queues, topics, and subscriptions.
- API reference for SqlFilter.
- Visit the Azure SDK for Node repository on GitHub.

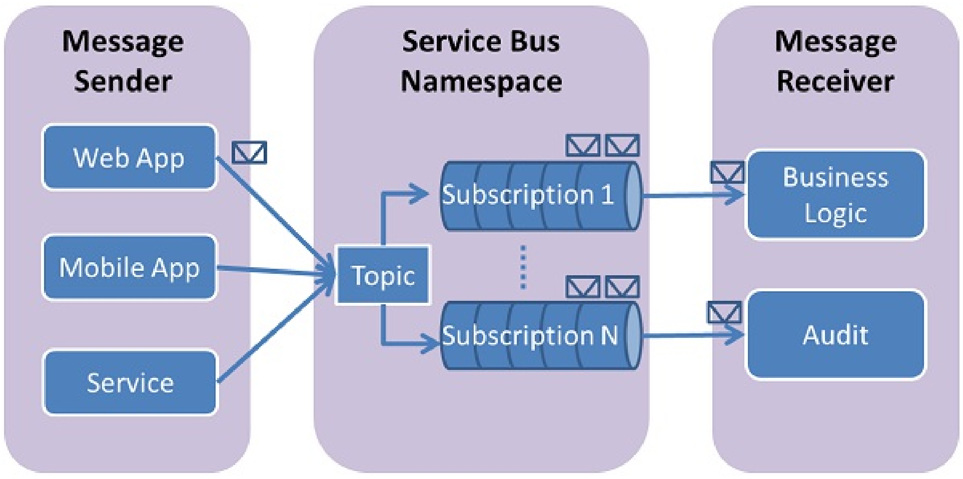
How to use Service Bus topics and subscriptions

1/19/2017 • 12 min to read • Edit on GitHub

This article shows you how to use Service Bus topics and subscriptions. The samples are written in PHP and use the Azure SDK for PHP. The scenarios covered include **creating topics and subscriptions**, **creating subscription filters**, **sending messages to a topic**, **receiving messages from a subscription**, and **deleting topics and subscriptions**.

What are Service Bus topics and subscriptions?

Service Bus topics and subscriptions support a *publish/subscribe* messaging communication model. When using topics and subscriptions, components of a distributed application do not communicate directly with each other; instead they exchange messages via a topic, which acts as an intermediary.



In contrast with Service Bus queues, in which each message is processed by a single consumer, topics and subscriptions provide a "one-to-many" form of communication, using a publish/subscribe pattern. It is possible to register multiple subscriptions to a topic. When a message is sent to a topic, it is then made available to each subscription to handle/process independently.

A subscription to a topic resembles a virtual queue that receives copies of the messages that were sent to the topic. You can optionally register filter rules for a topic on a per-subscription basis, which enables you to filter or restrict which messages to a topic are received by which topic subscriptions.

Service Bus topics and subscriptions enable you to scale and process a very large number of messages across many users and applications.

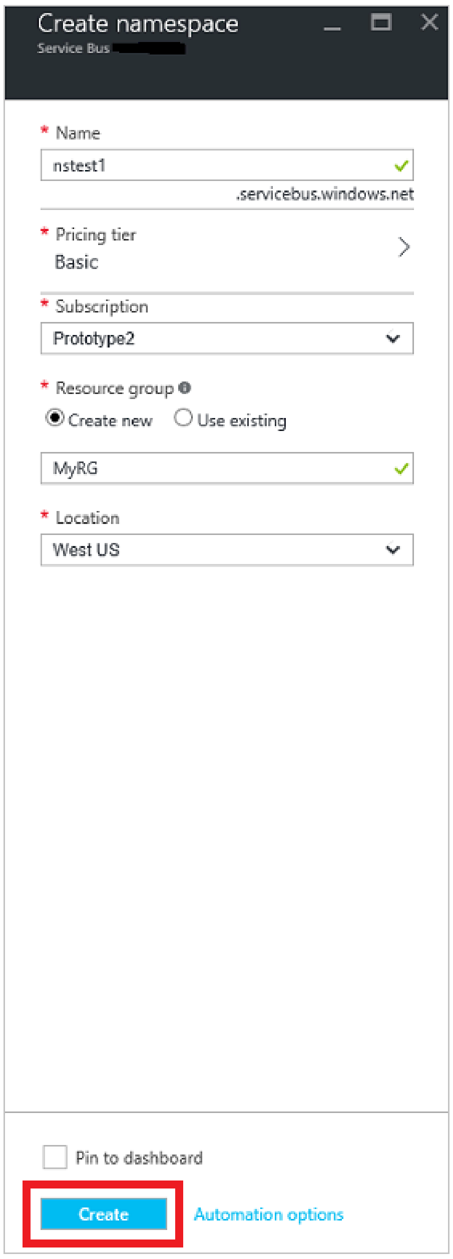
Create a namespace

To begin using Service Bus topics and subscriptions in Azure, you must first create a *service namespace*. A namespace provides a scoping container for addressing Service Bus resources within your application.

To create a namespace:

1. Log on to the Azure portal.
2. In the left navigation pane of the portal, click **New**, then click **Enterprise Integration**, and then click **Service Bus**.
3. In the **Create namespace** dialog, enter a namespace name. The system immediately checks to see if the name is available.

4. After making sure the namespace name is available, choose the pricing tier (Basic, Standard, or Premium).
5. In the **Subscription** field, choose an Azure subscription in which to create the namespace.
6. In the **Resource group** field, choose an existing resource group in which the namespace will live, or create a new one.
7. In **Location**, choose the country or region in which your namespace should be hosted.


Create namespace
Service Bus [redacted]

* Name
nstest1
.servicebus.windows.net

* Pricing tier
Basic

* Subscription
Prototype2

* Resource group
[✓ Create new] Use existing
MyRG

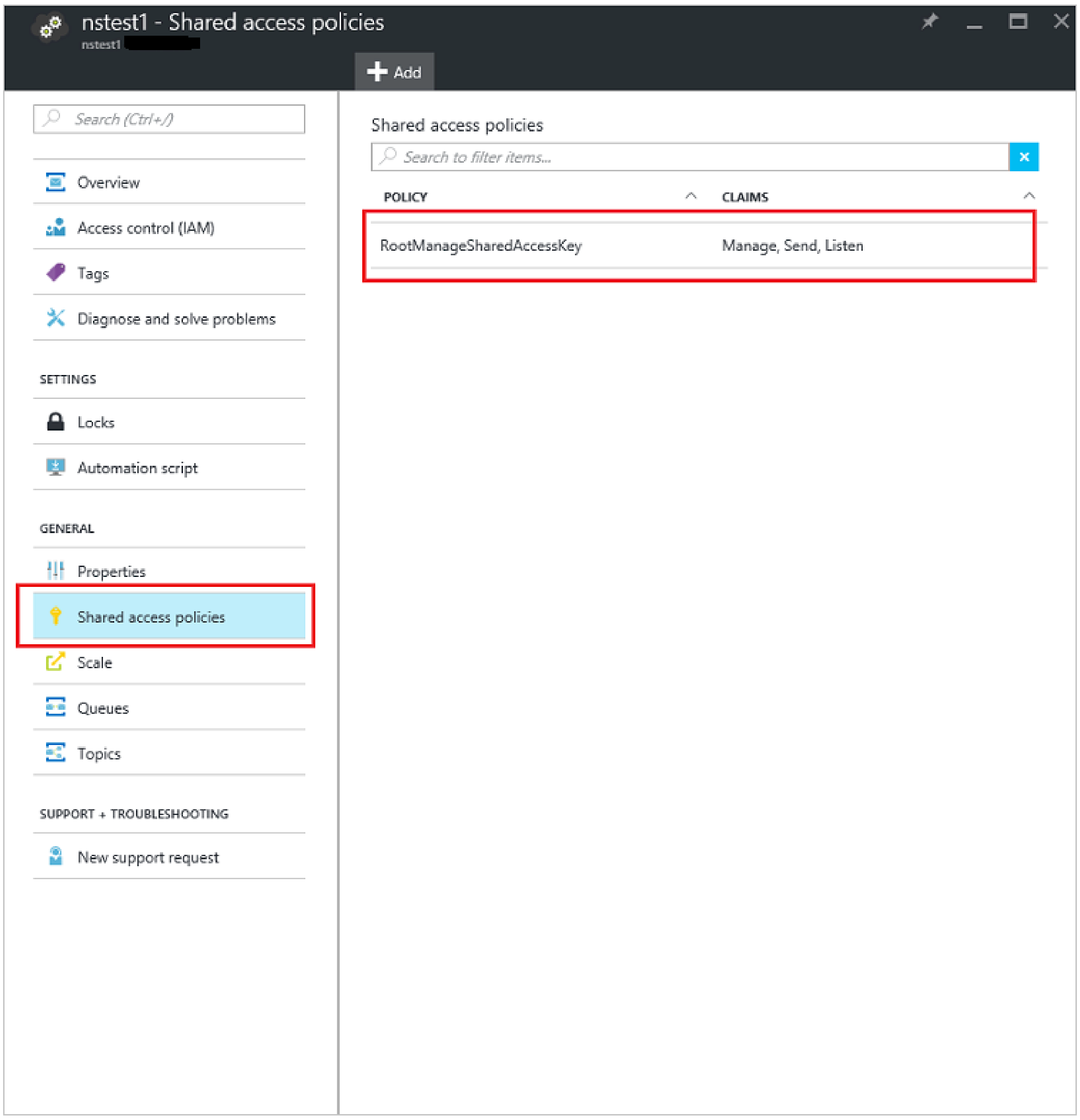
* Location
West US

Pin to dashboard

8. Click the **Create** button. The system now creates your namespace and enables it. You might have to wait several minutes as the system provisions resources for your account.

Obtain the credentials

1. In the list of namespaces, click the newly created namespace name.
2. In the **Service Bus namespace** blade, click **Shared access policies**.
3. In the **Shared access policies** blade, click **RootManageSharedAccessKey**.


nstest1 - Shared access policies
nstest1 [redacted]

Add

Search (Ctrl+/)

Overview
Access control (IAM)
Tags
Diagnose and solve problems

SETTINGS
Locks
Automation script

GENERAL
Properties
Shared access policies
Scale
Queues
Topics

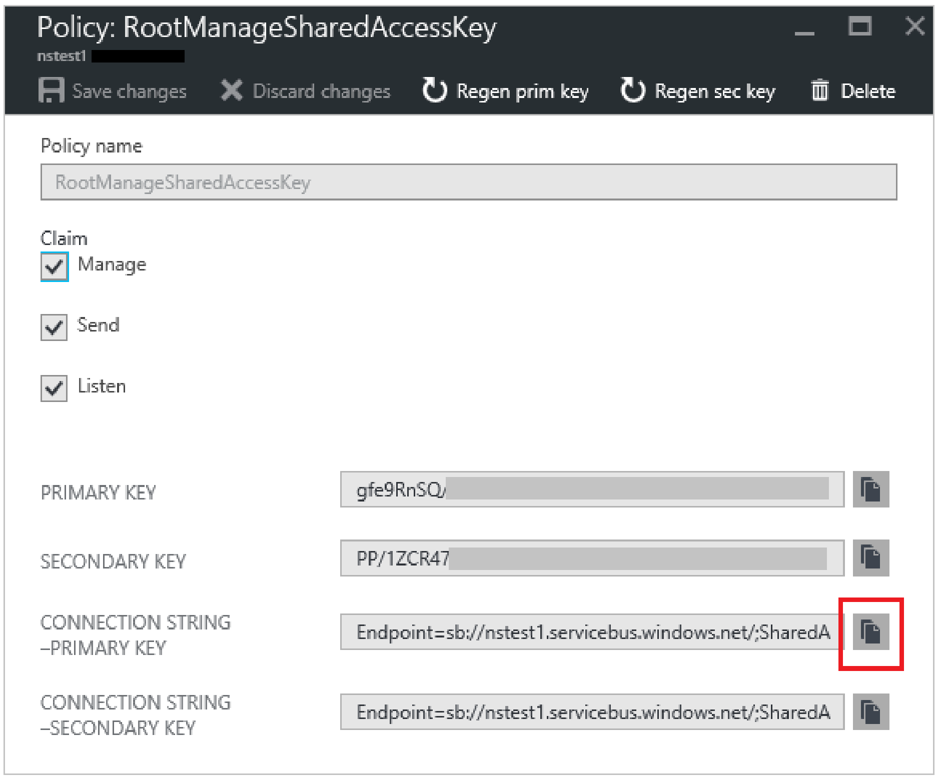
SUPPORT + TROUBLESHOOTING
New support request

Shared access policies

Search to filter items...

POLICY	CLAIMS
RootManageSharedAccessKey	Manage, Send, Listen

4. In the **Policy: RootManageSharedAccessKey** blade, click the copy button next to **Connection string–primary key**, to copy the connection string to your clipboard for later use.


Policy: RootManageSharedAccessKey

nstest1 [redacted]

Policy name: RootManageSharedAccessKey

Claim

- ✓ Manage
- ✓ Send
- ✓ Listen

PRIMARY KEY: gfe9RnSQ/

SECONDARY KEY: PP/1ZCR47

CONNECTION STRING –PRIMARY KEY: Endpoint=sb://nstest1.servicebus.windows.net/;SharedA

CONNECTION STRING –SECONDARY KEY: Endpoint=sb://nstest1.servicebus.windows.net/;SharedA

Create a PHP application

The only requirement for creating a PHP application that accesses the Azure Blob service is to reference classes in the Azure SDK for PHP from within your code. You can use any development tools to create your application, or Notepad.

NOTE

Your PHP installation must also have the OpenSSL extension installed and enabled.

This article describes how to use service features that can be called within a PHP application locally, or in code running within an Azure web role, worker role, or website.

Get the Azure client libraries

Install via Composer

1. Install Git. Note that on Windows, you must also add the Git executable to your PATH environment variable.
2. Create a file named **composer.json** in the root of your project and add the following code to it:

```
{
  "require": {
    "microsoft/windowsazure": "^0.4"
  }
}
```

3. Download **composer.phar** in your project root.
4. Open a command prompt and execute the following command in your project root

```
php composer.phar install
```

Configure your application to use Service Bus

To use the Service Bus APIs:

1. Reference the autoloader file using the require_once statement.
2. Reference any classes you might use.

The following example shows how to include the autoloader file and reference the **ServiceBusService** class.

NOTE

This example (and other examples in this article) assumes you have installed the PHP Client Libraries for Azure via Composer. If you installed the libraries manually or as a PEAR package, you must reference the **WindowsAzure.php** autoloader file.

```
require_once 'vendor\autoload.php';
use WindowsAzure\Common\ServicesBuilder;
```

In the following examples, the `require_once` statement will always be shown, but only the classes necessary for the example to execute are referenced.

Set up a Service Bus connection

To instantiate a Service Bus client you must first have a valid connection string in this format:

```
Endpoint=[yourEndpoint];SharedSecretIssuer=[Default Issuer];SharedSecretValue=[Default Key]
```

Where `Endpoint` is typically of the format `https://[yourNamespace].servicebus.windows.net`.

To create any Azure service client you must use the `ServicesBuilder` class. You can:

- Pass the connection string directly to it.
- Use the **CloudConfigurationManager (CCM)** to check multiple external sources for the connection string:
  - By default it comes with support for one external source - environmental variables.
  - You can add new sources by extending the `ConnectionStringSource` class.

For the examples outlined here, the connection string is passed directly.

```
require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;

$connectionString = "Endpoint=[yourEndpoint];SharedSecretIssuer=[Default Issuer];SharedSecretValue=[Default
Key]";

$serviceBusRestProxy = ServicesBuilder::getInstance()->createServiceBusService($connectionString);
```

Create a topic

You can perform management operations for Service Bus topics via the `ServiceBusRestProxy` class. A `ServiceBusRestProxy` object is constructed via the `ServicesBuilder::createServiceBusService` factory method with an appropriate connection string that encapsulates the token permissions to manage it.

The following example shows how to instantiate a `ServiceBusRestProxy` and call

`ServiceBusRestProxy->createTopic` to create a topic named `mytopic` within a `MySBNamespace` namespace:

```
require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use WindowsAzure\Common\ServiceException;
use WindowsAzure\ServiceBus\Models\TopicInfo;

// Create Service Bus REST proxy.
$serviceBusRestProxy = ServicesBuilder::getInstance()->createServiceBusService($connectionString);

try    {
    // Create topic.
    $topicInfo = new TopicInfo("mytopic");
    $serviceBusRestProxy->createTopic($topicInfo);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/windowsazure/dd179357
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}
```

NOTE

You can use the `listTopics` method on `ServiceBusRestProxy` objects to check if a topic with a specified name already exists within a service namespace.

Create a subscription

Topic subscriptions are also created with the `ServiceBusRestProxy->createSubscription` method. Subscriptions are named and can have an optional filter that restricts the set of messages passed to the subscription's virtual queue.

Create a subscription with the default (MatchAll) filter

The **MatchAll** filter is the default filter that is used if no filter is specified when a new subscription is created. When the **MatchAll** filter is used, all messages published to the topic are placed in the subscription's virtual queue. The following example creates a subscription named 'mysubscription' and uses the default **MatchAll** filter.

```
require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use WindowsAzure\Common\ServiceException;
use WindowsAzure\ServiceBus\Models\SubscriptionInfo;

// Create Service Bus REST proxy.
$serviceBusRestProxy = ServicesBuilder::getInstance()->createServiceBusService($connectionString);

try    {
    // Create subscription.
    $subscriptionInfo = new SubscriptionInfo("mysubscription");
    $serviceBusRestProxy->createSubscription("mytopic", $subscriptionInfo);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179357
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}
```

Create subscriptions with filters

You can also set up filters that enable you to specify which messages sent to a topic should appear within a specific topic subscription. The most flexible type of filter supported by subscriptions is the SqlFilter, which implements a subset of SQL92. SQL filters operate on the properties of the messages that are published to the topic. For more information about SqlFilters, see SqlFilter.SqlExpression Property.

NOTE

Each rule on a subscription processes incoming messages independently, adding their result messages to the subscription. In addition, each new subscription has a default **Rule** object with a filter that adds all messages from the topic to the subscription. To receive only messages matching your filter, you must remove the default rule. You can remove the default rule by using the `ServiceBusRestProxy->deleteRule` method.

The following example creates a subscription named `HighMessages` with a **SqlFilter** that only selects messages that have a custom `MessageNumber` property greater than 3. See Send messages to a topic for information about adding custom properties to messages.

```
$subscriptionInfo = new SubscriptionInfo("HighMessages");
$serviceBusRestProxy->createSubscription("mytopic", $subscriptionInfo);

$serviceBusRestProxy->deleteRule("mytopic", "HighMessages", '$Default');

$ruleInfo = new RuleInfo("HighMessagesRule");
$ruleInfo->withSqlFilter("MessageNumber > 3");
$ruleResult = $serviceBusRestProxy->createRule("mytopic", "HighMessages", $ruleInfo);
```

Note that this code requires the use of an additional namespace: `WindowsAzure\ServiceBus\Models\SubscriptionInfo`
.

Similarly, the following example creates a subscription named `LowMessages` with a `SqlFilter` that only selects messages that have a `MessageNumber` property less than or equal to 3.

```
$subscriptionInfo = new SubscriptionInfo("LowMessages");
$serviceBusRestProxy->createSubscription("mytopic", $subscriptionInfo);

$serviceBusRestProxy->deleteRule("mytopic", "LowMessages", '$Default');

$ruleInfo = new RuleInfo("LowMessagesRule");
$ruleInfo->withSqlFilter("MessageNumber <= 3");
$ruleResult = $serviceBusRestProxy->createRule("mytopic", "LowMessages", $ruleInfo);
```

Now, when a message is sent to the `mytopic` topic, it is always delivered to receivers subscribed to the `mysubscription` subscription, and selectively delivered to receivers subscribed to the `HighMessages` and `LowMessages` subscriptions (depending upon the message content).

Send messages to a topic

To send a message to a Service Bus topic, your application calls the `ServiceBusRestProxy->sendTopicMessage` method. The following code shows how to send a message to the `mytopic` topic previously created within the `MySBNamespace` service namespace.

```
require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use WindowsAzure\Common\ServiceException;
use WindowsAzure\ServiceBus\Models\BrokeredMessage;

// Create Service Bus REST proxy.
$serviceBusRestProxy = ServicesBuilder::getInstance()->createServiceBusService($connectionString);

try    {
    // Create message.
    $message = new BrokeredMessage();
    $message->setBody("my message");

    // Send message.
    $serviceBusRestProxy->sendTopicMessage("mytopic", $message);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/hh780775
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}
```

Messages sent to Service Bus topics are instances of the BrokeredMessage class. BrokeredMessage objects have a set of standard properties and methods, as well as properties that can be used to hold custom application-specific properties. The following example shows how to send 5 test messages to the `mytopic` topic previously created. The `setProperty` method is used to add a custom property (`MessageNumber`) to each message. Note that the `MessageNumber` property value varies on each message (you can use this value to determine which subscriptions receive it, as shown in the Create a subscription section):

```
for($i = 0; $i < 5; $i++){
    // Create message.
    $message = new BrokeredMessage();
    $message->setBody("my message ".$i);

    // Set custom property.
    $message->setProperty("MessageNumber", $i);

    // Send message.
    $serviceBusRestProxy->sendTopicMessage("mytopic", $message);
}
```

Service Bus topics support a maximum message size of 256 KB in the Standard tier and 1 MB in the Premium tier. The header, which includes the standard and custom application properties, can have a maximum size of 64 KB. There is no limit on the number of messages held in a topic but there is a cap on the total size of the messages held by a topic. This upper limit on topic size is 5 GB. For more information about quotas, see Service Bus quotas.

Receive messages from a subscription

The best way to receive messages from a subscription is to use a `ServiceBusRestProxy->receiveSubscriptionMessage` method. Messages can be received in two different modes: *ReceiveAndDelete and PeekLock*. **PeekLock** is the default.

When using the ReceiveAndDelete mode, receive is a single-shot operation; that is, when Service Bus receives a read request for a message in a subscription, it marks the message as being consumed and returns it to the application. ReceiveAndDelete * mode is the simplest model and works best for scenarios in which an 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. Because Service Bus will have marked the message as being consumed, then when the application restarts and begins consuming messages again, it will have missed the message that was consumed prior to the crash.

In the default PeekLock mode, receiving a message becomes a two stage operation, which makes it possible to support applications that cannot tolerate missing messages. When Service Bus receives a request, it finds the next message to be consumed, locks it to prevent other consumers 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 passing the received message to `ServiceBusRestProxy->deleteMessage`. When Service Bus sees the `deleteMessage` call, it will mark the message as being consumed and remove it from the queue.

The following example shows how to receive and process a message using PeekLock mode (the default mode).

```
require_once 'vendor/autoload.php';

use WindowsAzure\Common\ServicesBuilder;
use WindowsAzure\Common\ServiceException;
use WindowsAzure\ServiceBus\Models\ReceiveMessageOptions;

// Create Service Bus REST proxy.
$serviceBusRestProxy = ServicesBuilder::getInstance()->createServiceBusService($connectionString);

try    {
    // Set receive mode to PeekLock (default is ReceiveAndDelete)
    $options = new ReceiveMessageOptions();
    $options->setPeekLock();

    // Get message.
    $message = $serviceBusRestProxy->receiveSubscriptionMessage("mytopic", "mysubscription", $options);

    echo "Body: ".$message->getBody()."<br />";
    echo "MessageID: ".$message->getMessageId()."<br />";

    /*---------------------------
        Process message here.
    ----------------------------*/

    // Delete message. Not necessary if peek lock is not set.
    echo "Deleting message...<br />";
    $serviceBusRestProxy->deleteMessage($message);
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/hh780735
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}
```

How to: handle application crashes and unreadable messages

Service Bus provides functionality to help you gracefully recover from errors in your application or difficulties processing a message. If a receiver application is unable to process the message for some reason, then it can call the `unlockMessage` method on the received message (instead of the `deleteMessage` method). This will cause Service Bus to unlock the message within the queue and make it available to be received again, either by the same consuming application or by another consuming application.

There is also a timeout associated with a message locked within the queue, 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 automatically and make it available to be received again.

In the event that the application crashes after processing the message but before the `deleteMessage` request is issued, then the message will be redelivered to the application when it restarts. This is often called *At Least Once* processing; that is, each message is processed at least once but in certain situations the same message may be redelivered. If the scenario cannot tolerate duplicate processing, then application developers should add additional logic to applications to handle duplicate message delivery. This is often achieved using the `getMessageId` method of the message, which remains constant across delivery attempts.

Delete topics and subscriptions

To delete a topic or a subscription, use the `ServiceBusRestProxy->deleteTopic` or the `ServiceBusRestProxy->deleteSubscripton` methods, respectively. Note that deleting a topic also deletes any subscriptions that are registered with the topic.

The following example shows how to delete a topic named `mytopic` and its registered subscriptions.

```
require_once 'vendor/autoload.php';

use WindowsAzure\ServiceBus\ServiceBusService;
use WindowsAzure\ServiceBus\ServiceBusSettings;
use WindowsAzure\Common\ServiceException;

// Create Service Bus REST proxy.
$serviceBusRestProxy = ServicesBuilder::getInstance()->createServiceBusService($connectionString);

try    {
    // Delete topic.
    $serviceBusRestProxy->deleteTopic("mytopic");
}
catch(ServiceException $e){
    // Handle exception based on error codes and messages.
    // Error codes and messages are here:
    // http://msdn.microsoft.com/library/azure/dd179357
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."<br />";
}
```

By using the `deleteSubscription` method, you can delete a subscription independently:

```
$serviceBusRestProxy->deleteSubscription("mytopic", "mysubscription");
```

Next steps

Now that you've learned the basics of Service Bus queues, see Queues, topics, and subscriptions for more information.

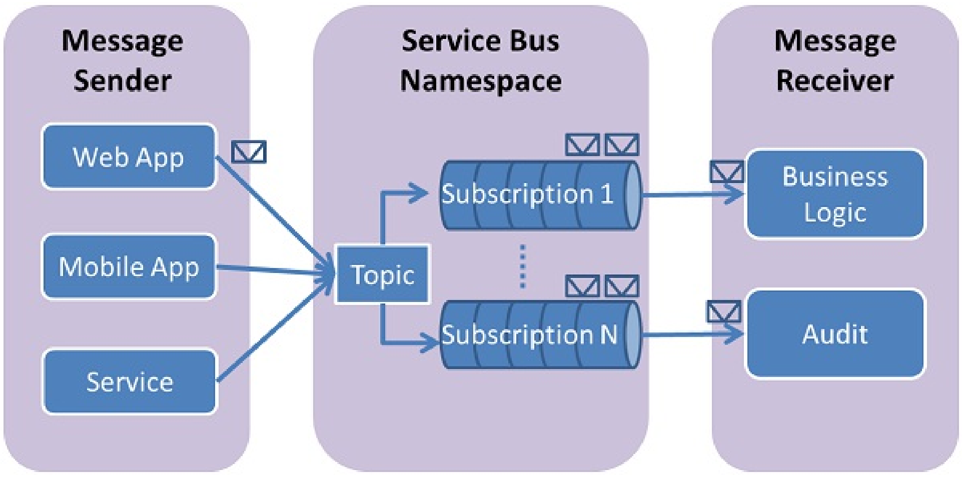
How to use Service Bus topics and subscriptions

1/17/2017 • 8 min to read • Edit on GitHub

This article describes how to use Service Bus topics and subscriptions. The samples are written in Python and use the Python Azure package. The scenarios covered include **creating topics and subscriptions**, **creating subscription filters**, **sending messages to a topic**, **receiving messages from a subscription**, and **deleting topics and subscriptions**. For more information about topics and subscriptions, see the Next Steps section.

What are Service Bus topics and subscriptions?

Service Bus topics and subscriptions support a *publish/subscribe* messaging communication model. When using topics and subscriptions, components of a distributed application do not communicate directly with each other; instead they exchange messages via a topic, which acts as an intermediary.



In contrast with Service Bus queues, in which each message is processed by a single consumer, topics and subscriptions provide a "one-to-many" form of communication, using a publish/subscribe pattern. It is possible to register multiple subscriptions to a topic. When a message is sent to a topic, it is then made available to each subscription to handle/process independently.

A subscription to a topic resembles a virtual queue that receives copies of the messages that were sent to the topic. You can optionally register filter rules for a topic on a per-subscription basis, which enables you to filter or restrict which messages to a topic are received by which topic subscriptions.

Service Bus topics and subscriptions enable you to scale and process a very large number of messages across many users and applications.

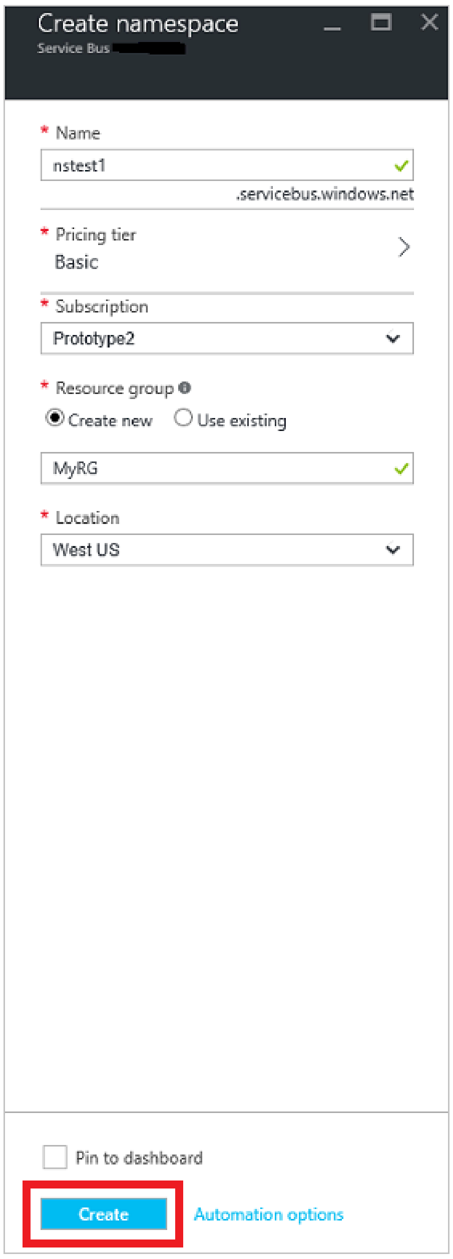
Create a namespace

To begin using Service Bus topics and subscriptions in Azure, you must first create a *service namespace*. A namespace provides a scoping container for addressing Service Bus resources within your application.

To create a namespace:

1. Log on to the Azure portal.
2. In the left navigation pane of the portal, click **New**, then click **Enterprise Integration**, and then click **Service Bus**.
3. In the **Create namespace** dialog, enter a namespace name. The system immediately checks to see if the name is available.

4. After making sure the namespace name is available, choose the pricing tier (Basic, Standard, or Premium).
5. In the **Subscription** field, choose an Azure subscription in which to create the namespace.
6. In the **Resource group** field, choose an existing resource group in which the namespace will live, or create a new one.
7. In **Location**, choose the country or region in which your namespace should be hosted.


Create namespace
Service Bus [redacted]

* Name
nstest1 ✓
.servicebus.windows.net

* Pricing tier
Basic

* Subscription
Prototype2

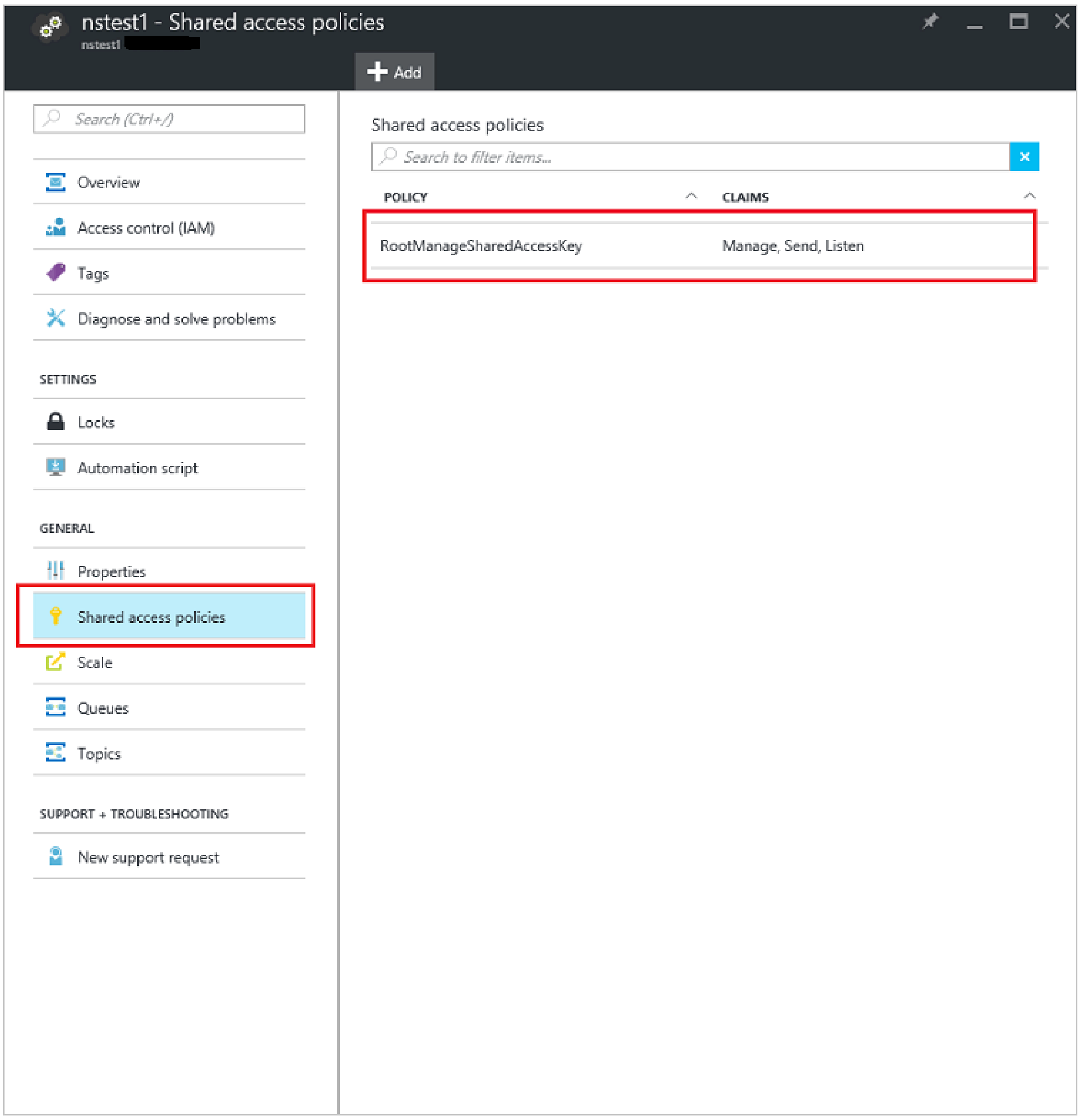
* Resource group
[✓ Create new] Use existing
MyRG ✓

* Location
West US

8. Click the **Create** button. The system now creates your namespace and enables it. You might have to wait several minutes as the system provisions resources for your account.

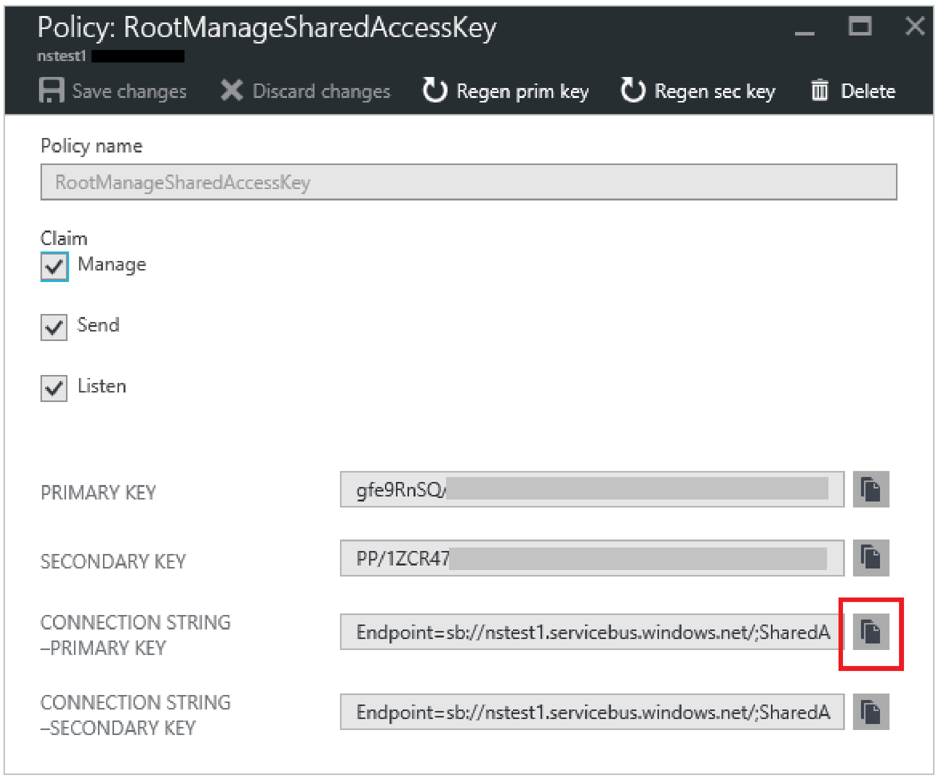
Obtain the credentials

1. In the list of namespaces, click the newly created namespace name.
2. In the **Service Bus namespace** blade, click **Shared access policies**.
3. In the **Shared access policies** blade, click **RootManageSharedAccessKey**.



POLICY	CLAIMS
RootManageSharedAccessKey	Manage, Send, Listen

4. In the **Policy: RootManageSharedAccessKey** blade, click the copy button next to **Connection string–primary key**, to copy the connection string to your clipboard for later use.


Policy: RootManageSharedAccessKey

nstest1 [redacted]

Policy name

RootManageSharedAccessKey

Claim

- [x] Manage
- [x] Send
- [x] Listen

PRIMARY KEY	gfe9RnSQ/[redacted]
SECONDARY KEY	PP/1ZCR47[redacted]
CONNECTION STRING –PRIMARY KEY	Endpoint=sb://nstest1.servicebus.windows.net/;SharedA
CONNECTION STRING –SECONDARY KEY	Endpoint=sb://nstest1.servicebus.windows.net/;SharedA

NOTE

If you need to install Python or the Python Azure package, please see the Python Installation Guide.

Create a topic

The **ServiceBusService** object enables you to work with topics. Add the following near the top of any Python file in which you wish to programmatically access Service Bus:

```
from azure.servicebus import ServiceBusService, Message, Topic, Rule, DEFAULT_RULE_NAME
```

The following code creates a **ServiceBusService** object. Replace `mynamespace`, `sharedaccesskeyname`, and `sharedaccesskey` with your actual namespace, Shared Access Signature (SAS) key name, and key value.

```
bus_service = ServiceBusService(
    service_namespace='mynamespace',
    shared_access_key_name='sharedaccesskeyname',
    shared_access_key_value='sharedaccesskey')
```

You can obtain the values for the SAS key name and value from the Azure portal.

```
bus_service.create_topic('mytopic')
```

create_topic also supports additional options, which enable you to override default topic settings such as message time to live or maximum topic size. The following example sets the maximum topic size to 5 GB, and a time to live (TTL) value of 1 minute:

```
topic_options = Topic()
topic_options.max_size_in_megabytes = '5120'
topic_options.default_message_time_to_live = 'PT1M'

bus_service.create_topic('mytopic', topic_options)
```

Create subscriptions

Subscriptions to topics are also created with the **ServiceBusService** object. Subscriptions are named and can have an optional filter that restricts the set of messages delivered to the subscription's virtual queue.

> **NOTE**
>
> Subscriptions are persistent and will continue to exist until either they, or the topic to which they are subscribed, are deleted.

Create a subscription with the default (MatchAll) filter

The **MatchAll** filter is the default filter that is used if no filter is specified when a new subscription is created. When the **MatchAll** filter is used, all messages published to the topic are placed in the subscription's virtual queue. The following example creates a subscription named 'AllMessages' and uses the default **MatchAll** filter.

```
bus_service.create_subscription('mytopic', 'AllMessages')
```

Create subscriptions with filters

You can also define filters that enable you to specify which messages sent to a topic should show up within a specific topic subscription.

The most flexible type of filter supported by subscriptions is a **SqlFilter**, which implements a subset of SQL92. SQL filters operate on the properties of the messages that are published to the topic. For more information about the expressions that can be used with a SQL filter, see the SqlFilter.SqlExpression syntax.

You can add filters to a subscription by using the **create_rule** method of the **ServiceBusService** object. This method allows you to add new filters to an existing subscription.

> **NOTE**
>
> Because the default filter is applied automatically to all new subscriptions, you must first remove the default filter or the **MatchAll** will override any other filters you may specify. You can remove the default rule by using the **delete_rule** method of the **ServiceBusService** object.

The following example creates a subscription named `HighMessages` with a **SqlFilter** that only selects messages that have a custom **messagenumber** property greater than 3:

```
bus_service.create_subscription('mytopic', 'HighMessages')

rule = Rule()
rule.filter_type = 'SqlFilter'
rule.filter_expression = 'messagenumber > 3'

bus_service.create_rule('mytopic', 'HighMessages', 'HighMessageFilter', rule)
bus_service.delete_rule('mytopic', 'HighMessages', DEFAULT_RULE_NAME)
```

Similarly, the following example creates a subscription named `LowMessages` with a **SqlFilter** that only selects messages that have a **messagenumber** property less than or equal to 3:

```
bus_service.create_subscription('mytopic', 'LowMessages')

rule = Rule()
rule.filter_type = 'SqlFilter'
rule.filter_expression = 'messagenumber <= 3'

bus_service.create_rule('mytopic', 'LowMessages', 'LowMessageFilter', rule)
bus_service.delete_rule('mytopic', 'LowMessages', DEFAULT_RULE_NAME)
```

Now, when a message is sent to `mytopic` it is always delivered to receivers subscribed to the **AllMessages** topic subscription, and selectively delivered to receivers subscribed to the **HighMessages** and **LowMessages** topic subscriptions (depending on the message content).

Send messages to a topic

To send a message to a Service Bus topic, your application must use the **send_topic_message** method of the **ServiceBusService** object.

The following example demonstrates how to send five test messages to `mytopic`. Note that the **messagenumber** property value of each message varies on the iteration of the loop (this determines which subscriptions receive it):

```
for i in range(5):
    msg = Message('Msg {0}'.format(i).encode('utf-8'), custom_properties={'messagenumber':i})
    bus_service.send_topic_message('mytopic', msg)
```

Service Bus topics support a maximum message size of 256 KB in the Standard tier and 1 MB in the Premium tier. The header, which includes the standard and custom application properties, can have a maximum size of 64 KB. There is no limit on the number of messages held in a topic but there is a cap on the total size of the messages held by a topic. This topic size is defined at creation time, with an upper limit of 5 GB. For more information about quotas, see Service Bus quotas.

Receive messages from a subscription

Messages are received from a subscription using the **receive_subscription_message** method on the **ServiceBusService** object:

```
msg = bus_service.receive_subscription_message('mytopic', 'LowMessages', peek_lock=False)
print(msg.body)
```

Messages are deleted from the subscription as they are read when the parameter **peek_lock** is set to **False**. You can read (peek) and lock the message without deleting it from the queue by setting the parameter **peek_lock** to **True**.

The behavior of reading and deleting the message as part of the receive operation is the simplest model, and works best for scenarios in which an 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. Because Service Bus will have marked the message as being consumed, then when the application restarts and begins consuming messages again, it will have missed the message that was consumed prior to the crash.

If the **peek_lock** parameter is set to **True**, the receive becomes a two stage operation, which makes it possible to support applications that cannot tolerate missing messages. When Service Bus receives a request, it finds the next message to be consumed, locks it to prevent other consumers 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 **delete** method on the **Message** object. The **delete** method marks the message as being consumed and removes it from the subscription.

```
msg = bus_service.receive_subscription_message('mytopic', 'LowMessages', peek_lock=True)
print(msg.body)

msg.delete()
```

How to handle application crashes and unreadable messages

Service Bus provides functionality to help you gracefully recover from errors in your application or difficulties processing a message. If a receiver application is unable to process the message for some reason, then it can call the **unlock** method on the **Message** object. This will cause Service Bus to unlock the message within the subscription and make it available to be received again, either by the same consuming application or by another consuming application.

There is also a timeout associated with a message locked within the subscription, and if the application fails to process the message before the lock timeout expires (for example, if the application crashes), then Service Bus unlocks the message automatically and makes it available to be received again.

In the event that the application crashes after processing the message but before the **delete** method is called, then 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 application developers should add additional logic to their application to handle duplicate message delivery. This is often achieved using the **MessageId** property of the message, which will remain constant across delivery attempts.

Delete topics and subscriptions

Topics and subscriptions are persistent, and must be explicitly deleted either through the Azure portal or programmatically. The following example shows how to delete the topic named `mytopic`:

```
bus_service.delete_topic('mytopic')
```

Deleting a topic also deletes any subscriptions that are registered with the topic. Subscriptions can also be deleted independently. The following code shows how to delete a subscription named `HighMessages` from the `mytopic` topic:

```
bus_service.delete_subscription('mytopic', 'HighMessages')
```

Next steps

Now that you've learned the basics of Service Bus topics, follow these links to learn more.

- See Queues, topics, and subscriptions.
- Reference for SqlFilter.SqlExpression.

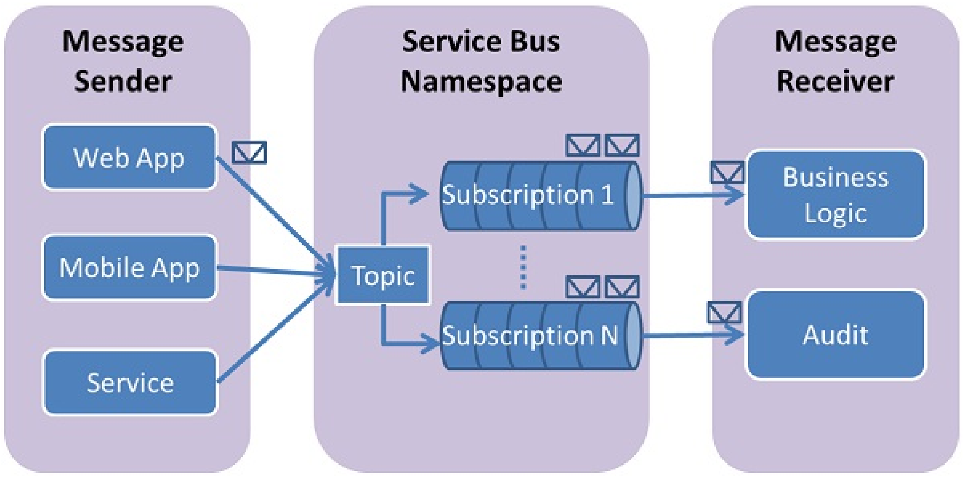
How to Use Service Bus Topics/Subscriptions

1/17/2017 • 11 min to read • Edit on GitHub

This article describes how to use Service Bus topics and subscriptions from Ruby applications. The scenarios covered include **creating topics and subscriptions, creating subscription filters, sending messages** to a topic, **receiving messages from a subscription**, and **deleting topics and subscriptions**. For more information on topics and subscriptions, see the Next Steps section.

What are Service Bus topics and subscriptions?

Service Bus topics and subscriptions support a *publish/subscribe* messaging communication model. When using topics and subscriptions, components of a distributed application do not communicate directly with each other; instead they exchange messages via a topic, which acts as an intermediary.



In contrast with Service Bus queues, in which each message is processed by a single consumer, topics and subscriptions provide a "one-to-many" form of communication, using a publish/subscribe pattern. It is possible to register multiple subscriptions to a topic. When a message is sent to a topic, it is then made available to each subscription to handle/process independently.

A subscription to a topic resembles a virtual queue that receives copies of the messages that were sent to the topic. You can optionally register filter rules for a topic on a per-subscription basis, which enables you to filter or restrict which messages to a topic are received by which topic subscriptions.

Service Bus topics and subscriptions enable you to scale and process a very large number of messages across many users and applications.

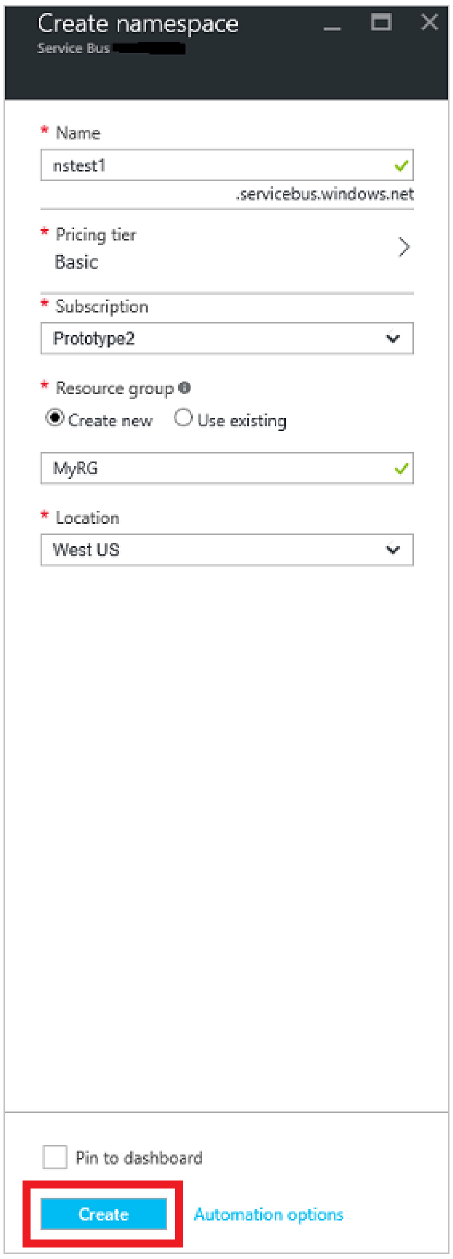
Create a namespace

To begin using Service Bus topics and subscriptions in Azure, you must first create a *service namespace*. A namespace provides a scoping container for addressing Service Bus resources within your application.

To create a namespace:

1. Log on to the Azure portal.
2. In the left navigation pane of the portal, click **New**, then click **Enterprise Integration**, and then click **Service Bus**.
3. In the **Create namespace** dialog, enter a namespace name. The system immediately checks to see if the name is available.

4. After making sure the namespace name is available, choose the pricing tier (Basic, Standard, or Premium).
5. In the **Subscription** field, choose an Azure subscription in which to create the namespace.
6. In the **Resource group** field, choose an existing resource group in which the namespace will live, or create a new one.
7. In **Location**, choose the country or region in which your namespace should be hosted.



Create namespace
Service Bus [redacted]

* Name
nstest1 ✓
.servicebus.windows.net

* Pricing tier
Basic

* Subscription
Prototype2

* Resource group
[✓ Create new] Use existing
MyRG ✓

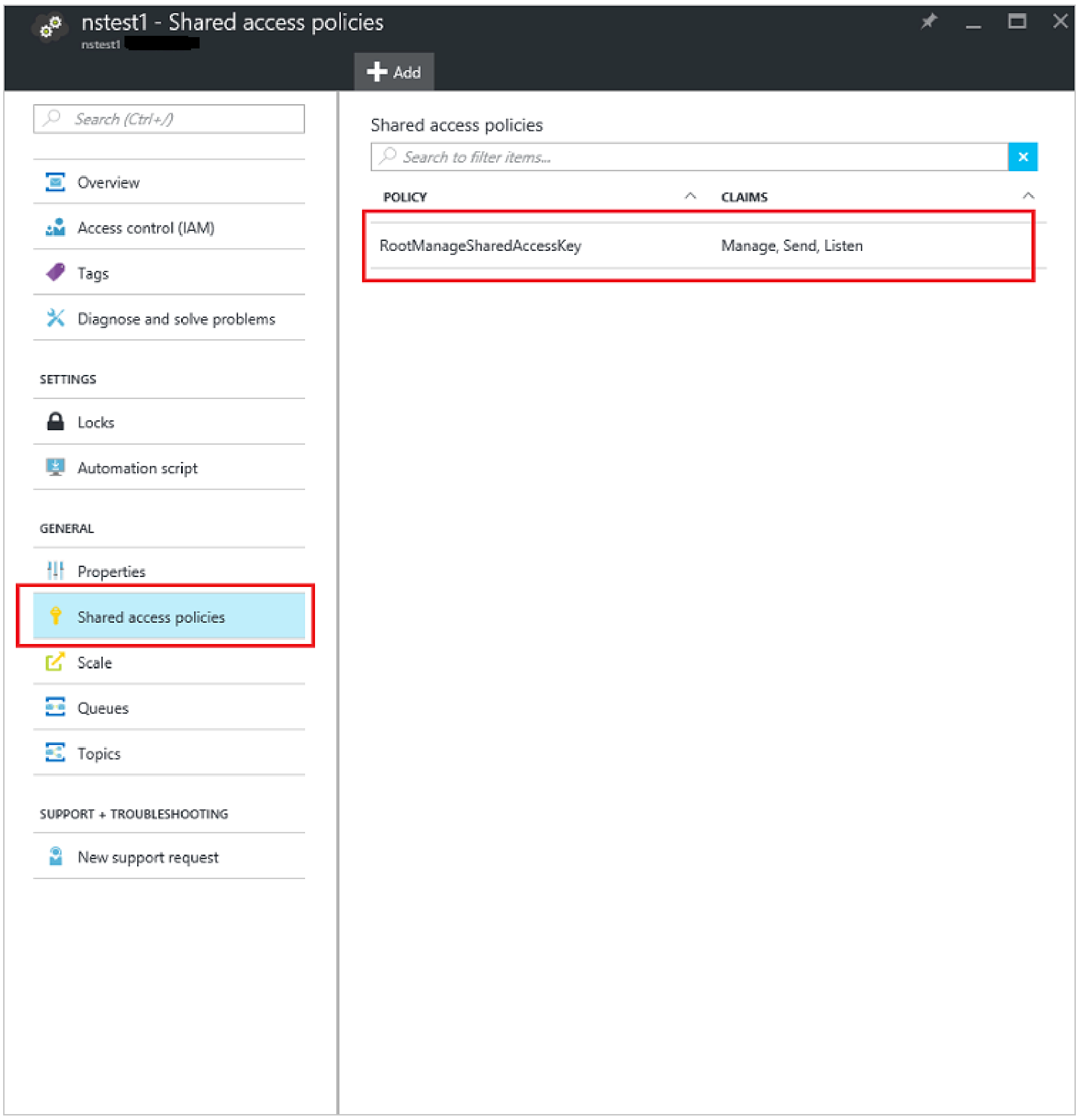
* Location
West US

Pin to dashboard

8. Click the **Create** button. The system now creates your namespace and enables it. You might have to wait several minutes as the system provisions resources for your account.

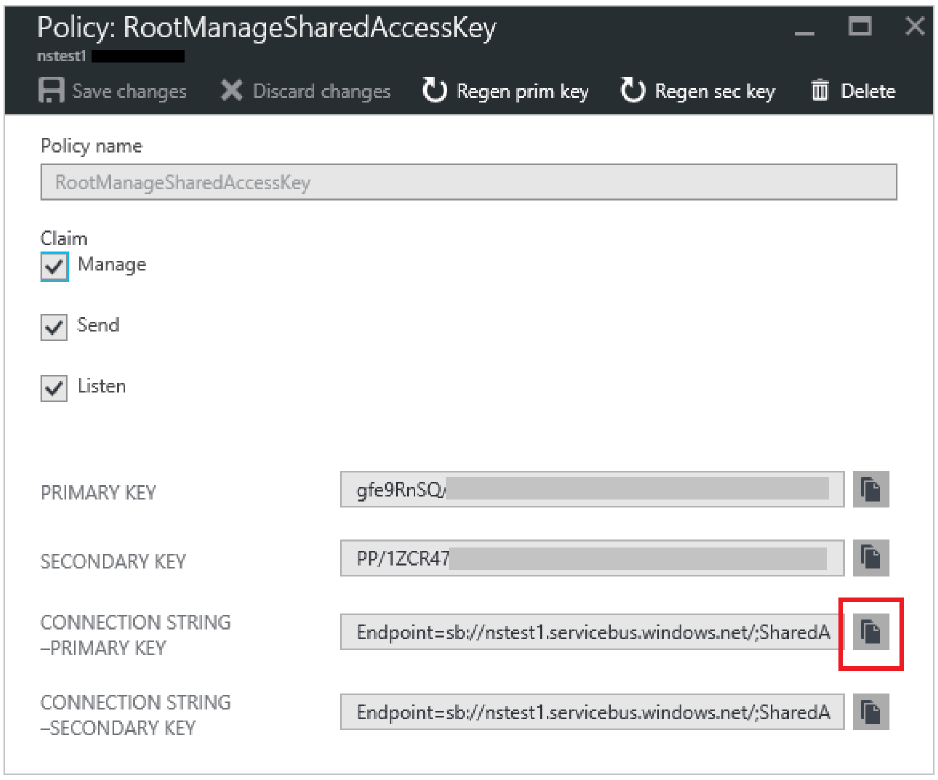
Obtain the credentials

1. In the list of namespaces, click the newly created namespace name.
2. In the **Service Bus namespace** blade, click **Shared access policies**.
3. In the **Shared access policies** blade, click **RootManageSharedAccessKey**.



POLICY	CLAIMS
RootManageSharedAccessKey	Manage, Send, Listen

4. In the **Policy: RootManageSharedAccessKey** blade, click the copy button next to **Connection string–primary key**, to copy the connection string to your clipboard for later use.


Policy: RootManageSharedAccessKey

nstest1 [redacted]

Policy name: RootManageSharedAccessKey

Claim
- [✓] Manage
- [✓] Send
- [✓] Listen

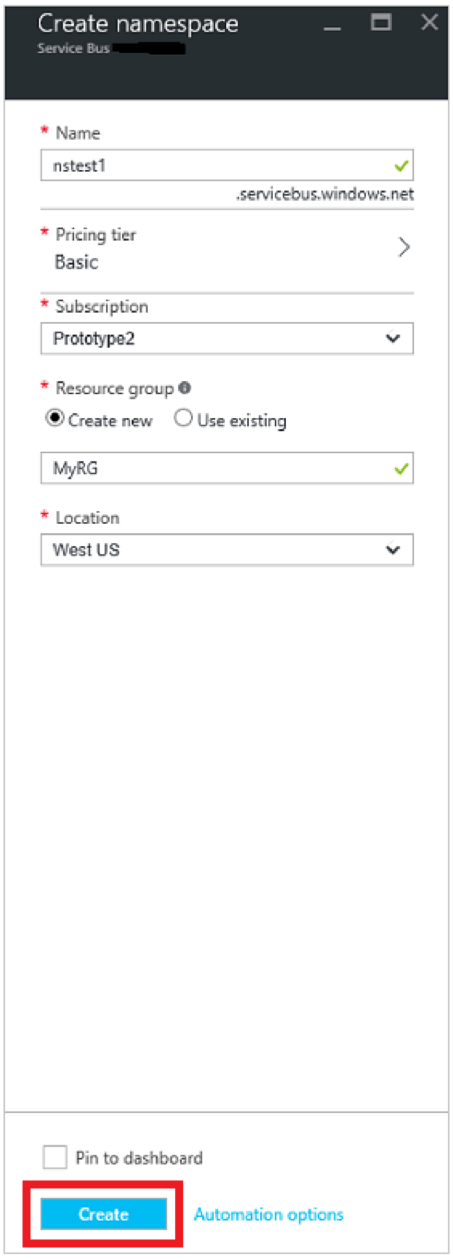
PRIMARY KEY	gfe9RnSQ/
SECONDARY KEY	PP/1ZCR47
CONNECTION STRING –PRIMARY KEY	Endpoint=sb://nstest1.servicebus.windows.net/;SharedA
CONNECTION STRING –SECONDARY KEY	Endpoint=sb://nstest1.servicebus.windows.net/;SharedA

Create a service namespace

To begin using Service Bus queues in Azure, you must first create a namespace. A namespace provides a scoping container for addressing Service Bus resources within your application.

To create a namespace:

1. Log on to the Azure portal.
2. In the left navigation pane of the portal, click **New**, then click **Enterprise Integration**, and then click **Service Bus**.
3. In the **Create namespace** dialog, enter a namespace name. The system immediately checks to see if the name is available.
4. After making sure the namespace name is available, choose the pricing tier (Basic, Standard, or Premium).
5. In the **Subscription** field, choose an Azure subscription in which to create the namespace.
6. In the **Resource group** field, choose an existing resource group in which the namespace will live, or create a new one.
7. In **Location**, choose the country or region in which your namespace should be hosted.


Create namespace
Service Bus [redacted]

* Name
nstest1 ✓
.servicebus.windows.net

* Pricing tier
Basic

* Subscription
Prototype2

* Resource group
[✓ Create new] Use existing

MyRG ✓

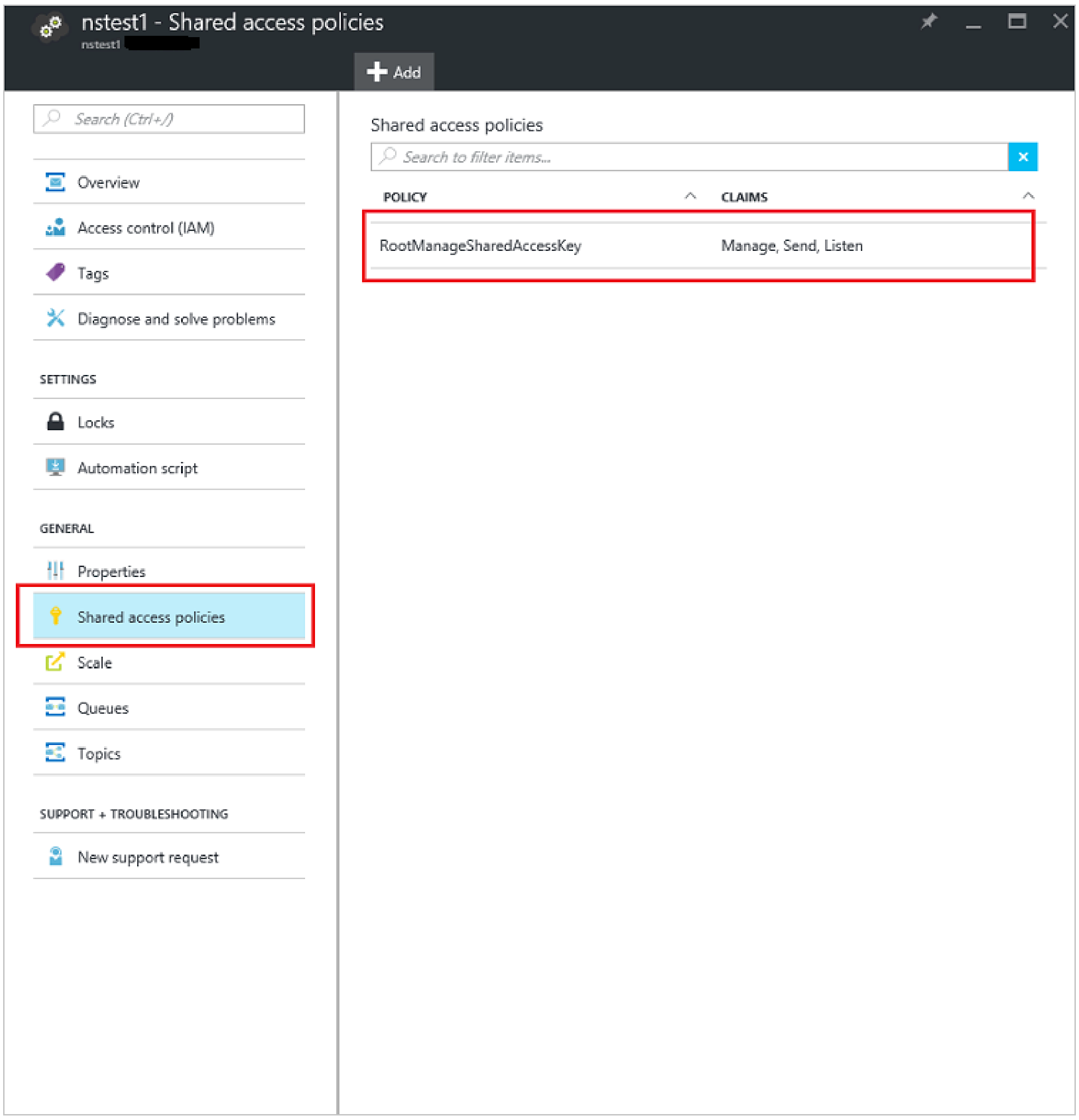
* Location
West US

☐ Pin to dashboard

8. Click **Create**. The system now creates your namespace and enables it. You might have to wait several minutes as the system provisions resources for your account.

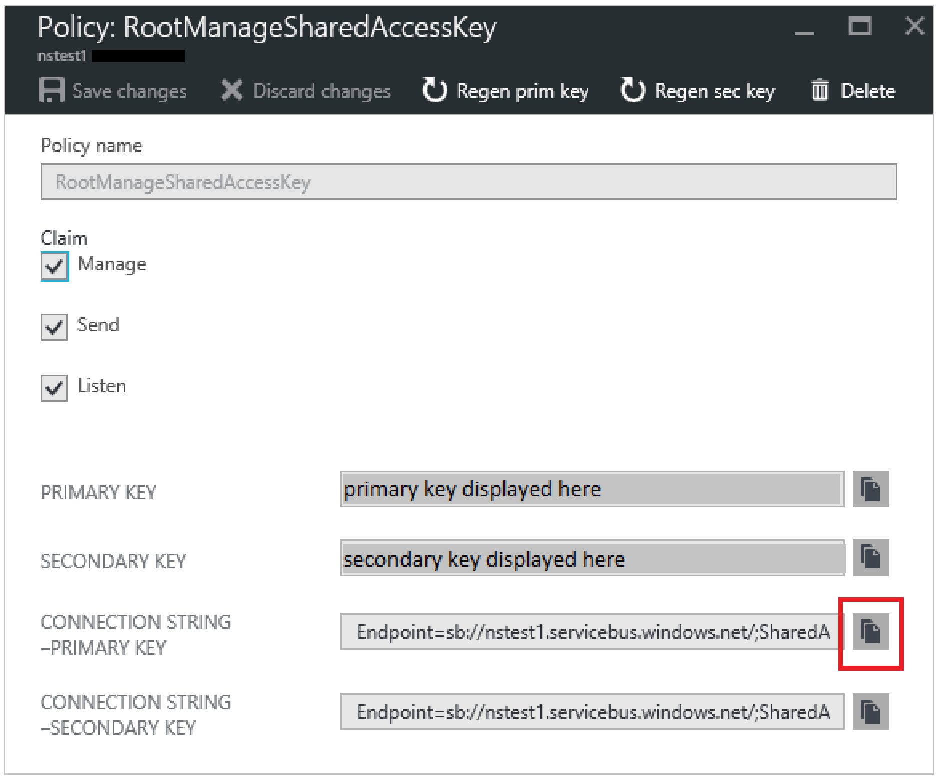
Obtain the management credentials

1. In the list of namespaces, click the newly created namespace name.
2. In the namespace blade, click **Shared access policies**.
3. In the **Shared access policies** blade, click **RootManageSharedAccessKey**.



POLICY	CLAIMS
RootManageSharedAccessKey	Manage, Send, Listen

4. In the **Policy: RootManageSharedAccessKey** blade, click the copy button next to **Connection string–primary key**, to copy the connection string to your clipboard for later use. Paste this value into Notepad or some other temporary location.


Policy: RootManageSharedAccessKey

nstest1 [redacted]

Save changes | Discard changes | Regen prim key | Regen sec key | Delete

Policy name: RootManageSharedAccessKey

Claim

- ✓ Manage
- ✓ Send
- ✓ Listen

PRIMARY KEY	primary key displayed here
SECONDARY KEY	secondary key displayed here
CONNECTION STRING –PRIMARY KEY	Endpoint=sb://nstest1.servicebus.windows.net/;SharedA
CONNECTION STRING –SECONDARY KEY	Endpoint=sb://nstest1.servicebus.windows.net/;SharedA

5. Repeat the previous step, copying and pasting the value of **Primary key** to a temporary location for later use.

Create a Ruby application

For instructions, see Create a Ruby Application on Azure.

Configure Your application to Use Service Bus

To use Service Bus, download and use the Ruby Azure package, which includes a set of convenience libraries that communicate with the storage REST services.

Use RubyGems to obtain the package

1. Use a command-line interface such as **PowerShell** (Windows), **Terminal** (Mac), or **Bash** (Unix).
2. Type "gem install azure" in the command window to install the gem and dependencies.

Import the package

Using your favorite text editor, add the following to the top of the Ruby file in which you intend to use storage:

```
require "azure"
```

Set up a Service Bus connection

The Azure module reads the environment variables **AZURE_SERVICEBUS_NAMESPACE** and **AZURE_SERVICEBUS_ACCESS_KEY** for information required to connect to your namespace. If these environment variables are not set, you must specify the namespace information before using **Azure::ServiceBusService** with the following code:

```
Azure.config.sb_namespace = "<your azure service bus namespace>"
Azure.config.sb_access_key = "<your azure service bus access key>"
```

Set the namespace value to the value you created rather than the entire URL. For example, use

"yourexamplenamespace", not "yourexamplenamespace.servicebus.windows.net".

Create a topic

The **Azure::ServiceBusService** object enables you to work with topics. The following code creates an **Azure::ServiceBusService** object. To create a topic, use the **create_topic()** method. The following example creates a topic or prints out the errors if there are any.

```
azure_service_bus_service = Azure::ServiceBusService.new
begin
  topic = azure_service_bus_service.create_queue("test-topic")
rescue
  puts $!
end
```

You can also pass a **Azure::ServiceBus::Topic** object with additional options, which allow you to override default topic settings such as message time to live or maximum queue size. The following example shows setting the maximum queue size to 5GB and time to live to 1 minute:

```
topic = Azure::ServiceBus::Topic.new("test-topic")
topic.max_size_in_megabytes = 5120
topic.default_message_time_to_live = "PT1M"

topic = azure_service_bus_service.create_topic(topic)
```

Create subscriptions

Topic subscriptions are also created with the **Azure::ServiceBusService** object. Subscriptions are named and can have an optional filter that restricts the set of messages delivered to the subscription's virtual queue.

Subscriptions are persistent and will continue to exist until either they, or the topic they are associated with, are deleted. If your application contains logic to create a subscription, it should first check if the subscription already exists by using the getSubscription method.

Create a subscription with the default (MatchAll) filter

The **MatchAll** filter is the default filter that is used if no filter is specified when a new subscription is created. When the **MatchAll** filter is used, all messages published to the topic are placed in the subscription's virtual queue. The following example creates a subscription named "all-messages" and uses the default **MatchAll** filter.

```
subscription = azure_service_bus_service.create_subscription("test-topic", "all-messages")
```

Create subscriptions with filters

You can also define filters that enable you to specify which messages sent to a topic should show up within a specific subscription.

The most flexible type of filter supported by subscriptions is the **Azure::ServiceBus::SqlFilter**, which implements a subset of SQL92. SQL filters operate on the properties of the messages that are published to the topic. For more details about the expressions that can be used with a SQL filter, review the SqlFilter.SqlExpression syntax.

You can add filters to a subscription by using the **create_rule()** method of the **Azure::ServiceBusService** object. This method enables you to add new filters to an existing subscription.

Since the default filter is applied automatically to all new subscriptions, you must first remove the default filter, or the **MatchAll** will override any other filters you may specify. You can remove the default rule by using the **delete_rule()** method on the **Azure::ServiceBusService** object.

The following example creates a subscription named "high-messages" with a **Azure::ServiceBus::SqlFilter** that only selects messages that have a custom **message_number** property greater than 3:

```
subscription = azure_service_bus_service.create_subscription("test-topic", "high-messages")
azure_service_bus_service.delete_rule("test-topic", "high-messages", "$Default")

rule = Azure::ServiceBus::Rule.new("high-messages-rule")
rule.topic = "test-topic"
rule.subscription = "high-messages"
rule.filter = Azure::ServiceBus::SqlFilter.new({
  :sql_expression => "message_number > 3" })
rule = azure_service_bus_service.create_rule(rule)
```

Similarly, the following example creates a subscription named "low-messages" with a **Azure::ServiceBus::SqlFilter** that only selects messages that have a **message_number** property less than or equal to 3:

```
subscription = azure_service_bus_service.create_subscription("test-topic", "low-messages")
azure_service_bus_service.delete_rule("test-topic", "low-messages", "$Default")

rule = Azure::ServiceBus::Rule.new("low-messages-rule")
rule.topic = "test-topic"
rule.subscription = "low-messages"
rule.filter = Azure::ServiceBus::SqlFilter.new({
  :sql_expression => "message_number <= 3" })
rule = azure_service_bus_service.create_rule(rule)
```

When a message is now sent to "test-topic", it is always be delivered to receivers subscribed to the "all-messages" topic subscription, and selectively delivered to receivers subscribed to the "high-messages" and "low-messages" topic subscriptions (depending upon the message content).

Send messages to a topic

To send a message to a Service Bus topic, your application must use the **send_topic_message()** method on the **Azure::ServiceBusService** object. Messages sent to Service Bus topics are instances of the **Azure::ServiceBus::BrokeredMessage** objects. **Azure::ServiceBus::BrokeredMessage** objects have a set of standard properties (such as **label** and **time_to_live**), a dictionary that is used to hold custom application-specific properties, and a body of string data. An application can set the body of the message by passing a string value to the **send_topic_message()** method and any required standard properties will be populated by default values.

The following example demonstrates how to send five test messages to "test-topic". Note that the **message_number** custom property value of each message varies on the iteration of the loop (this determines which subscription receives it):

```
5.times do |i|
  message = Azure::ServiceBus::BrokeredMessage.new("test message " + i,
    { :message_number => i })
  azure_service_bus_service.send_topic_message("test-topic", message)
end
```

Service Bus topics support a maximum message size of 256 KB in the Standard tier and 1 MB in the Premium tier. The header, which includes the standard and custom application properties, can have a maximum size of 64 KB. There is no limit on the number of messages held in a topic but there is a cap on the total size of the messages held by a topic. This topic size is defined at creation time, with an upper limit of 5 GB.

Receive messages from a subscription

Messages are received from a subscription using the **receive_subscription_message()** method on the **Azure::ServiceBusService** object. By default, messages are read(peak) and locked without deleting it from the subscription. You can read and delete the message from the subscription by setting the **peek_lock** option to **false**.

The default behavior makes the reading and deleting a two-stage operation, which also makes it possible to support applications that cannot tolerate missing messages. When Service Bus receives a request, it finds the next message to be consumed, locks it to prevent other consumers 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 **delete_subscription_message()** method and providing the message to be deleted as a parameter. The **delete_subscription_message()** method will mark the message as being consumed and remove it from the subscription.

If the **:peek_lock** parameter is set to **false**, reading and deleting the message becomes the simplest model, and works best for scenarios in which an 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. Because Service Bus will have marked the message as being consumed, then when the application restarts and begins consuming messages again, it will have missed the message that was consumed prior to the crash.

The following example demonstrates how messages can be received and processed using **receive_subscription_message()**. The example first receives and deletes a message from the "low-messages" subscription by using **:peek_lock** set to **false**, then it receives another message from the "high-messages" and then deletes the message using **delete_subscription_message()**:

```
message = azure_service_bus_service.receive_subscription_message(
  "test-topic", "low-messages", { :peek_lock => false })
message = azure_service_bus_service.receive_subscription_message(
  "test-topic", "high-messages")
azure_service_bus_service.delete_subscription_message(message)
```

Handle application crashes and unreadable messages

Service Bus provides functionality to help you gracefully recover from errors in your application or difficulties processing a message. If a receiver application is unable to process the message for some reason, then it can call the **unlock_subscription_message()** method on the **Azure::ServiceBusService** object. This causes Service Bus to unlock the message within the subscription and make it available to be received again, either by the same consuming application or by another consuming application.

There is also a timeout associated with a message locked within the subscription, 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 automatically and make it available to be received again.

In the event that the application crashes after processing the message but before the **delete_subscription_message()** method is called, then the message is 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 application developers should add additional logic to their application to handle duplicate message delivery. This logic is often achieved using the **message_id** property of the message, which will remain constant across delivery attempts.

Delete topics and subscriptions

Topics and subscriptions are persistent, and must be explicitly deleted either through the Azure portal or programmatically. The example below demonstrates how to delete the topic named "test-topic".

```
azure_service_bus_service.delete_topic("test-topic")
```

Deleting a topic also deletes any subscriptions that are registered with the topic. Subscriptions can also be deleted independently. The following code demonstrates how to delete the subscription named "high-messages" from the "test-topic" topic:

```
azure_service_bus_service.delete_subscription("test-topic", "high-messages")
```

Next steps

Now that you've learned the basics of Service Bus topics, follow these links to learn more.

- See Queues, topics, and subscriptions.
- API reference for SqlFilter.
- Visit the Azure SDK for Ruby repository on GitHub.

.NET multi-tier application using Azure Service Bus queues

2/28/2017 • 13 min to read • Edit on GitHub

Introduction

Developing for Microsoft Azure is easy using Visual Studio and the free Azure SDK for .NET. This tutorial walks you through the steps to create an application that uses multiple Azure resources running in your local environment. The steps assume you have no prior experience using Azure.

You will learn the following:

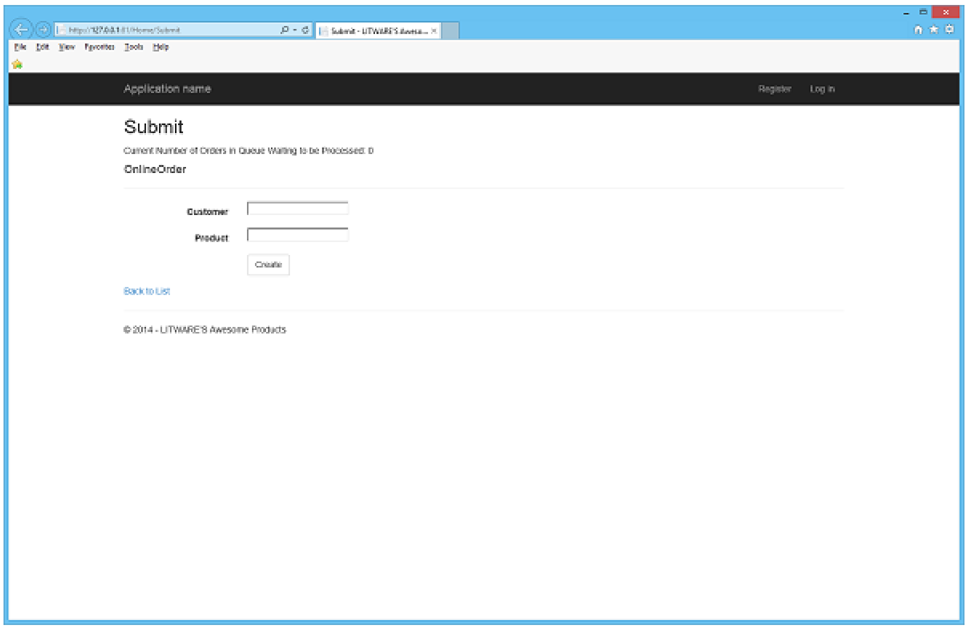
- How to enable your computer for Azure development with a single download and install.
- How to use Visual Studio to develop for Azure.
- How to create a multi-tier application in Azure using web and worker roles.
- How to communicate between tiers using Service Bus queues.

NOTE

To complete this tutorial, you need an Azure account. You can activate your MSDN subscriber benefits or sign up for a free account.

In this tutorial you'll build and run the multi-tier application in an Azure cloud service. The front end is an ASP.NET MVC web role and the back end is a worker-role that uses a Service Bus queue. You can create the same multi-tier application with the front end as a web project, that is deployed to an Azure website instead of a cloud service. For instructions about what to do differently on an Azure website front end, see the Next steps section. You can also try out the .NET on-premises/cloud hybrid application tutorial.

The following screen shot shows the completed application.



Submit

Current Number of Orders in Queue Waiting to be Processed: 0

OnlineOrder

Customer

Product

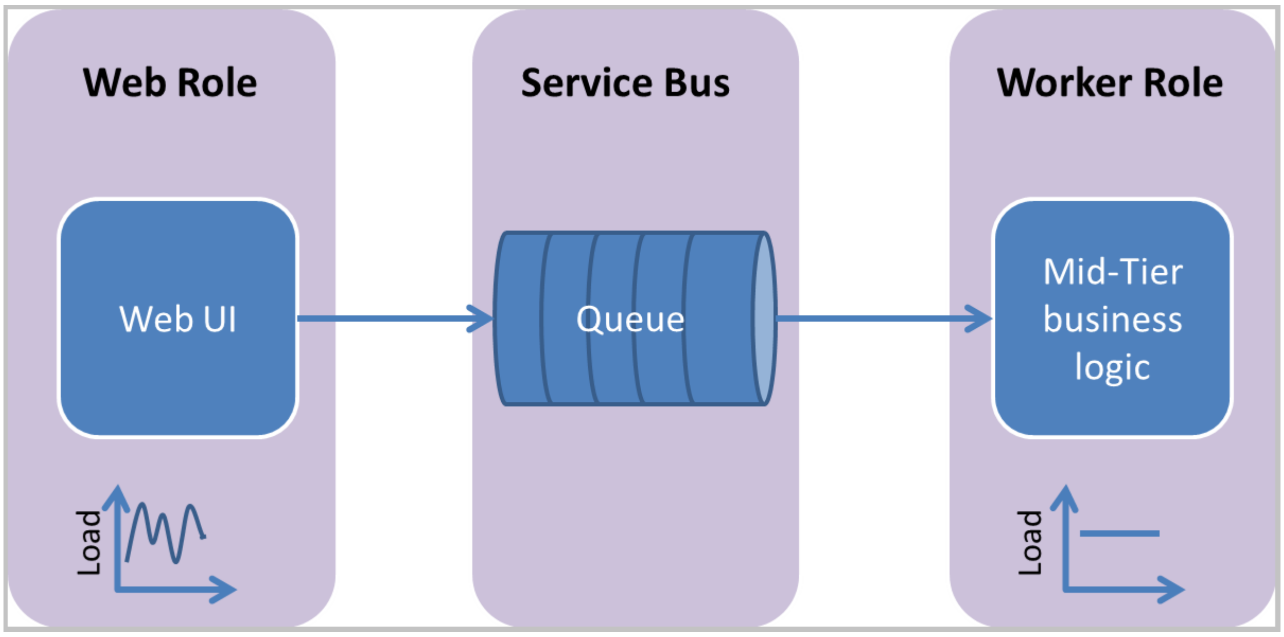
© 2014 - LITWARE'S Awesome Products

Scenario overview: inter-role communication

To submit an order for processing, the front-end UI component, running in the web role, must interact with the middle tier logic running in the worker role. This example uses Service Bus brokered messaging for the communication between the tiers.

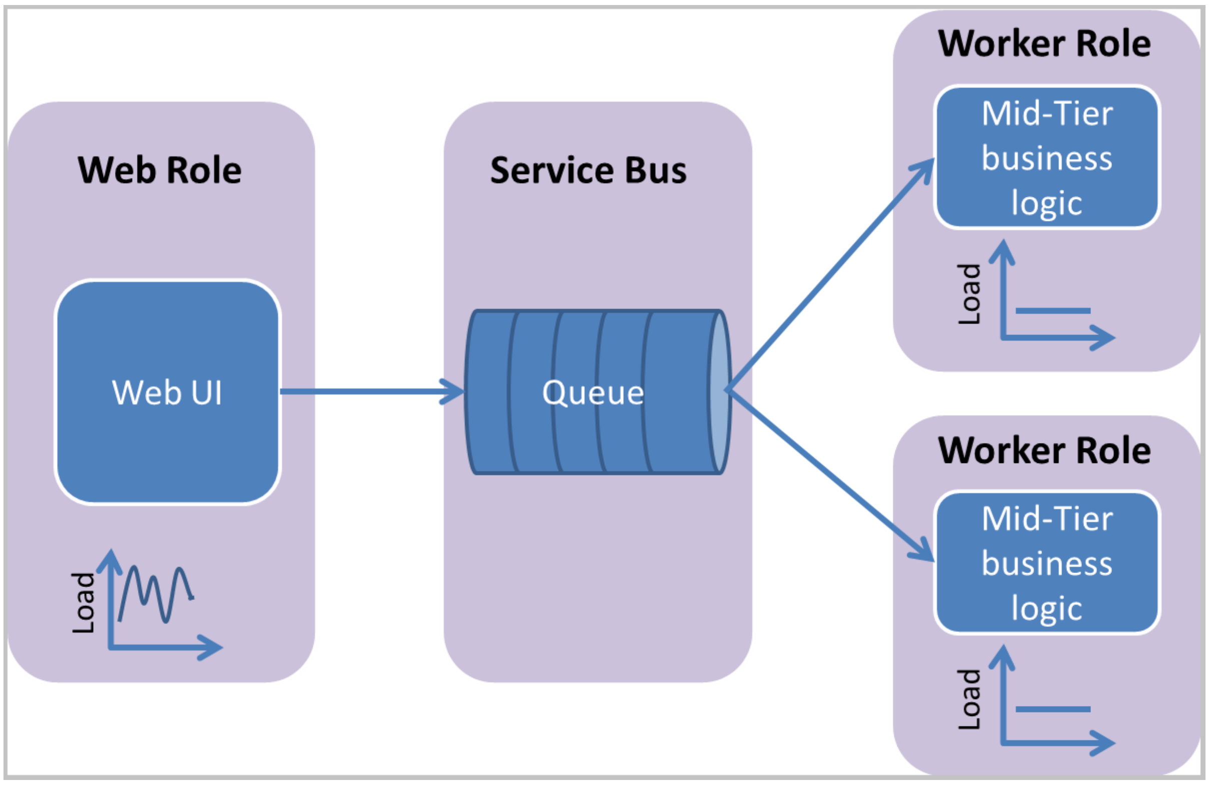
Using brokered messaging between the web and middle tiers decouples the two components. In contrast to direct messaging (that is, TCP or HTTP), the web tier does not connect to the middle tier directly; instead it pushes units of work, as messages, into Service Bus, which reliably retains them until the middle tier is ready to consume and process them.

Service Bus provides two entities to support brokered messaging: queues and topics. With queues, each message sent to the queue is consumed by a single receiver. Topics support the publish/subscribe pattern in which each published message is made available to a subscription registered with the topic. Each subscription logically maintains its own queue of messages. Subscriptions can also be configured with filter rules that restrict the set of messages passed to the subscription queue to those that match the filter. The following example uses Service Bus queues.



This communication mechanism has several advantages over direct messaging:

- **Temporal decoupling.** With the asynchronous messaging pattern, producers and consumers need not be online at the same time. Service Bus reliably stores messages until the consuming party is ready to receive them. This enables the components of the distributed application to be disconnected, either voluntarily, for example, for maintenance, or due to a component crash, without impacting the system as a whole. Furthermore, the consuming application might only need to come online during certain times of the day.
- **Load leveling.** In many applications, system load varies over time, while 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 (the worker) only needs to be provisioned to accommodate average load rather than peak load. The depth of the queue grows and contracts as the incoming load varies. This directly saves money in terms of the amount of infrastructure required to service the application load.
- **Load balancing.** As 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 enables optimal use of the worker machines even if the worker machines differ in terms of processing power, as they will pull messages at their own maximum rate. This pattern is often termed the *competing consumer* pattern.



The following sections discuss the code that implements this architecture.

Set up the development environment

Before you can begin developing Azure applications, get the tools and set up your development environment.

1. Install the Azure SDK for .NET from the SDK downloads page.
2. In the **.NET** column, click the version of Visual Studio you are using. The steps in this tutorial use Visual Studio 2015.
3. When prompted to run or save the installer, click **Run**.
4. In the **Web Platform Installer**, click **Install** and proceed with the installation.
5. Once the installation is complete, you will have everything necessary to start to develop the app. The SDK includes tools that let you easily develop Azure applications in Visual Studio.

Create a namespace

The next step is to create a service namespace, and obtain a Shared Access Signature (SAS) key. A namespace provides an application boundary for each application exposed through Service Bus. A SAS key is generated by the system when a namespace is created. The combination of namespace and SAS key provides the credentials for Service Bus to authenticate access to an application.

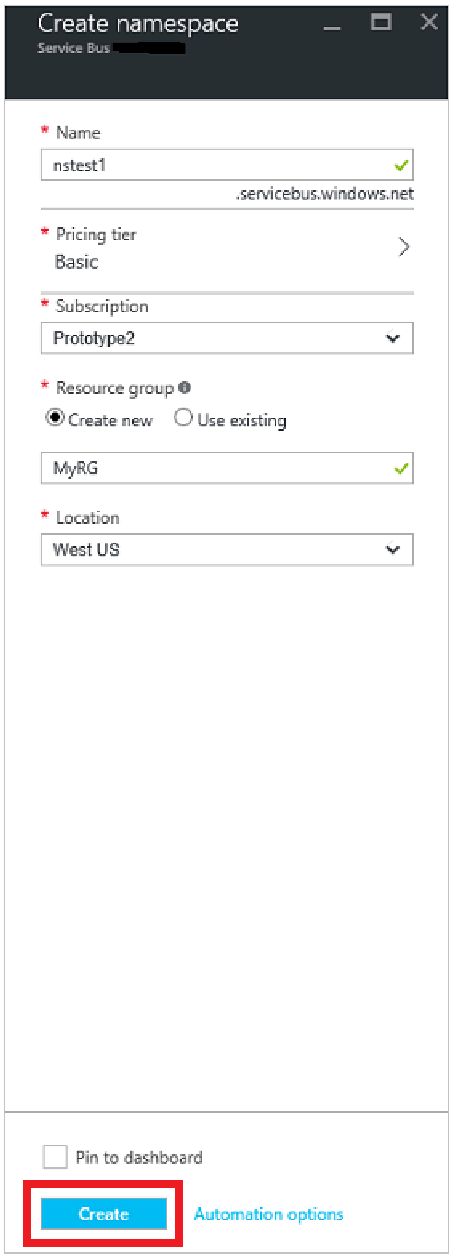
Create a service namespace

To begin using Service Bus queues in Azure, you must first create a namespace. A namespace provides a scoping container for addressing Service Bus resources within your application.

To create a namespace:

1. Log on to the Azure portal.
2. In the left navigation pane of the portal, click **New**, then click **Enterprise Integration**, and then click **Service Bus**.
3. In the **Create namespace** dialog, enter a namespace name. The system immediately checks to see if the name is available.

4. After making sure the namespace name is available, choose the pricing tier (Basic, Standard, or Premium).
5. In the **Subscription** field, choose an Azure subscription in which to create the namespace.
6. In the **Resource group** field, choose an existing resource group in which the namespace will live, or create a new one.
7. In **Location**, choose the country or region in which your namespace should be hosted.


Create namespace

Service Bus [redacted]

* Name

nstest1 ✓

.servicebus.windows.net

* Pricing tier

Basic

* Subscription

Prototype2

* Resource group

[✓ Create new] Use existing

MyRG ✓

* Location

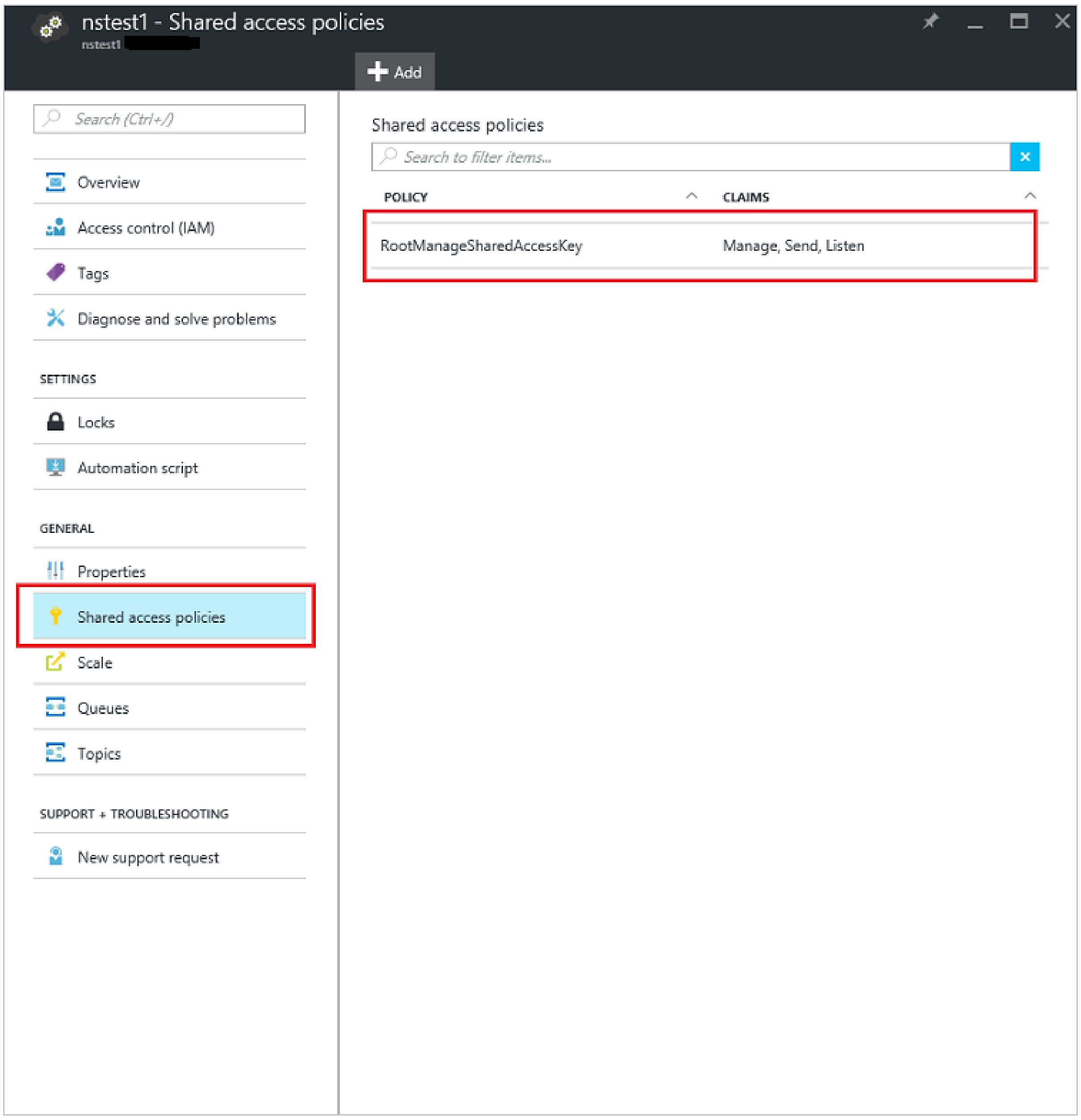
West US

Pin to dashboard

8. Click **Create**. The system now creates your namespace and enables it. You might have to wait several minutes as the system provisions resources for your account.

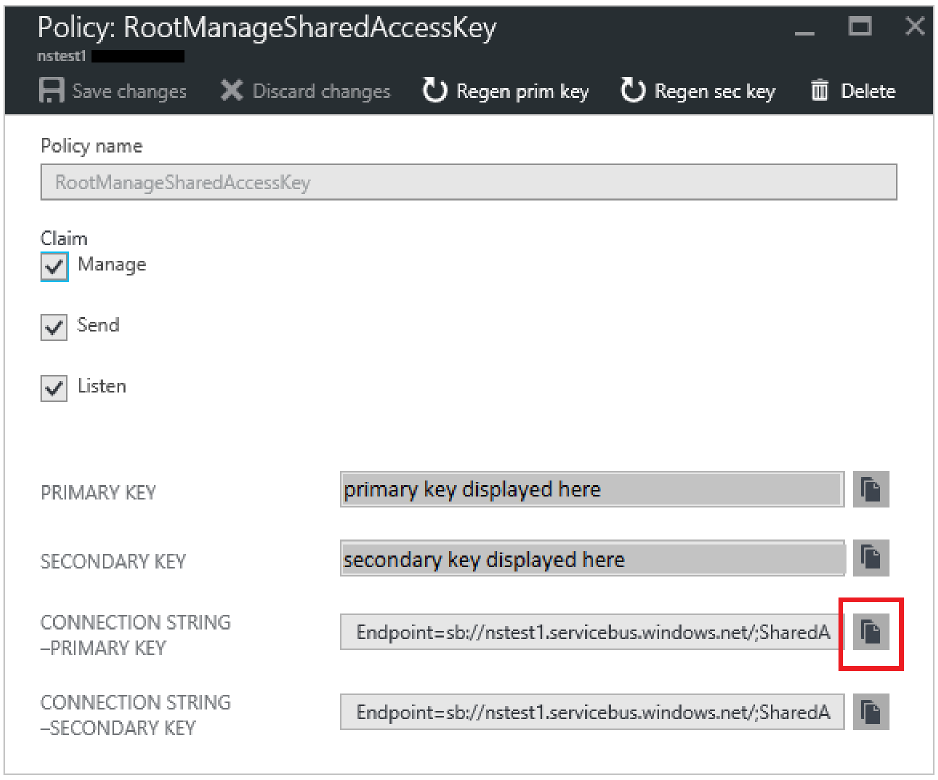
Obtain the management credentials

1. In the list of namespaces, click the newly created namespace name.
2. In the namespace blade, click **Shared access policies**.
3. In the **Shared access policies** blade, click **RootManageSharedAccessKey**.



POLICY	CLAIMS
RootManageSharedAccessKey	Manage, Send, Listen

4. In the **Policy: RootManageSharedAccessKey** blade, click the copy button next to **Connection string–primary key**, to copy the connection string to your clipboard for later use. Paste this value into Notepad or some other temporary location.


Policy: RootManageSharedAccessKey
nstest1 [redacted]
Save changes · Discard changes · Regen prim key · Regen sec key · Delete

Policy name
RootManageSharedAccessKey

Claim
✓ Manage
✓ Send
✓ Listen

PRIMARY KEY	primary key displayed here
SECONDARY KEY	secondary key displayed here
CONNECTION STRING –PRIMARY KEY	Endpoint=sb://nstest1.servicebus.windows.net/;SharedA
CONNECTION STRING –SECONDARY KEY	Endpoint=sb://nstest1.servicebus.windows.net/;SharedA

5. Repeat the previous step, copying and pasting the value of **Primary key** to a temporary location for later use.

Create a web role

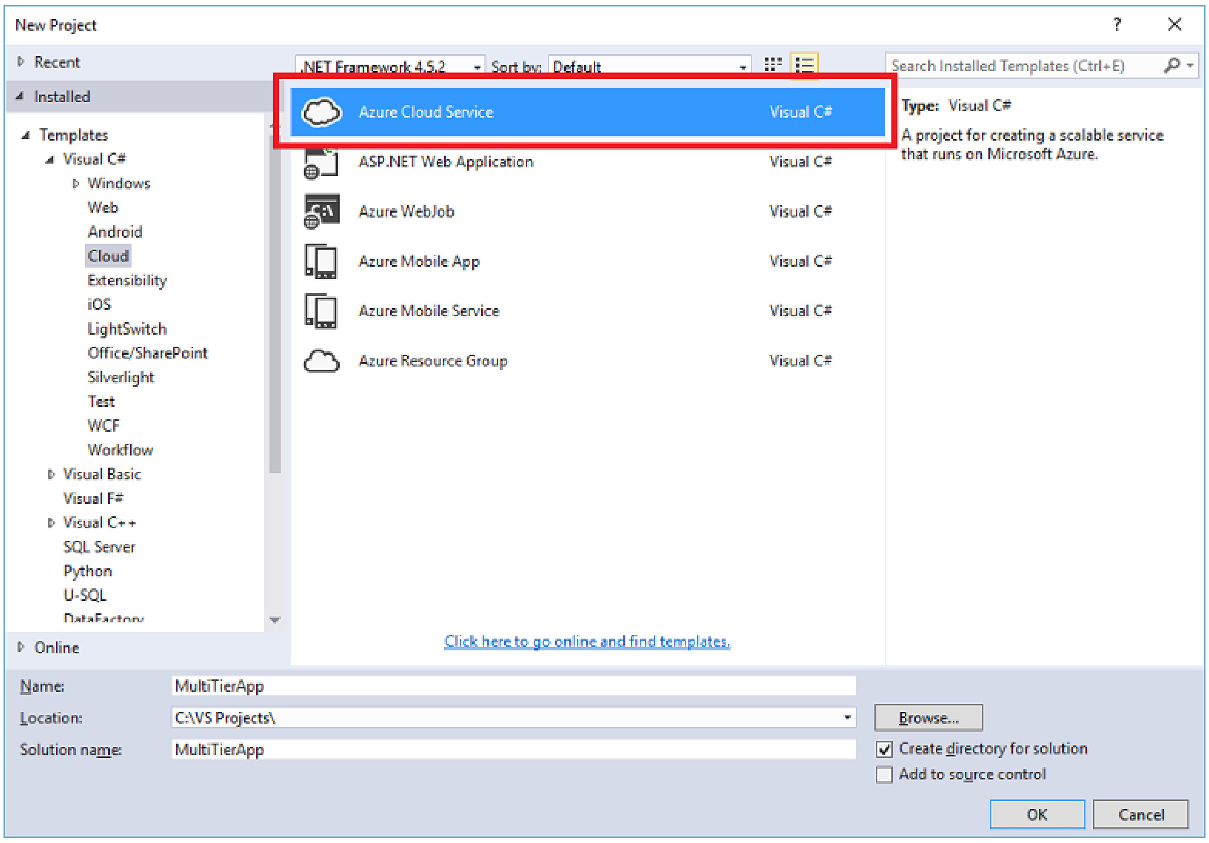
In this section, you build the front end of your application. First, you create the pages that your application displays. After that, add code that submits items to a Service Bus queue and displays status information about the queue.

Create the project

1. Using administrator privileges, start Microsoft Visual Studio. To start Visual Studio with administrator privileges, right-click the **Visual Studio** program icon, and then click **Run as administrator**. The Azure compute emulator, discussed later in this article, requires that Visual Studio be started with administrator privileges.

   In Visual Studio, on the **File** menu, click **New**, and then click **Project**.

2. From **Installed Templates**, under **Visual C#**, click **Cloud** and then click **Azure Cloud Service**. Name the project **MultiTierApp**. Then click **OK**.



.NET Framework 4.5.2 Sort by: Default

Search Installed Templates (Ctrl+E)

Templates
- Visual C#
  - Windows
  - Web
  - Android
  - Cloud
  - Extensibility
  - iOS
  - LightSwitch
  - Office/SharePoint
  - Silverlight
  - Test
  - WCF
  - Workflow
- Visual Basic
- Visual F#
- Visual C++
- SQL Server
- Python
- U-SQL
- DataFactory

Azure Cloud Service	Visual C#
ASP.NET Web Application	Visual C#
Azure WebJob	Visual C#
Azure Mobile App	Visual C#
Azure Mobile Service	Visual C#
Azure Resource Group	Visual C#

Type: Visual C#

A project for creating a scalable service that runs on Microsoft Azure.

Name: MultiTierApp

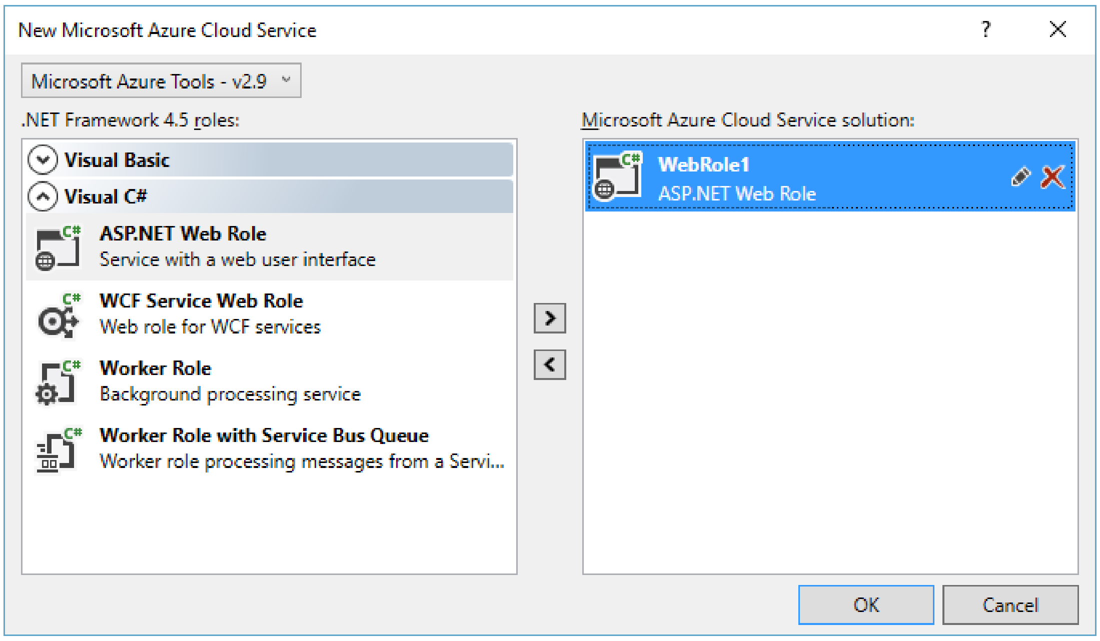
Location: C:\VS Projects\

Solution name: MultiTierApp

☑ Create directory for solution

☐ Add to source control

3. From **.NET Framework 4.5** roles, double-click **ASP.NET Web Role**.



Microsoft Azure Tools - v2.9

.NET Framework 4.5 roles:

Visual Basic

Visual C#

ASP.NET Web Role
Service with a web user interface

WCF Service Web Role
Web role for WCF services

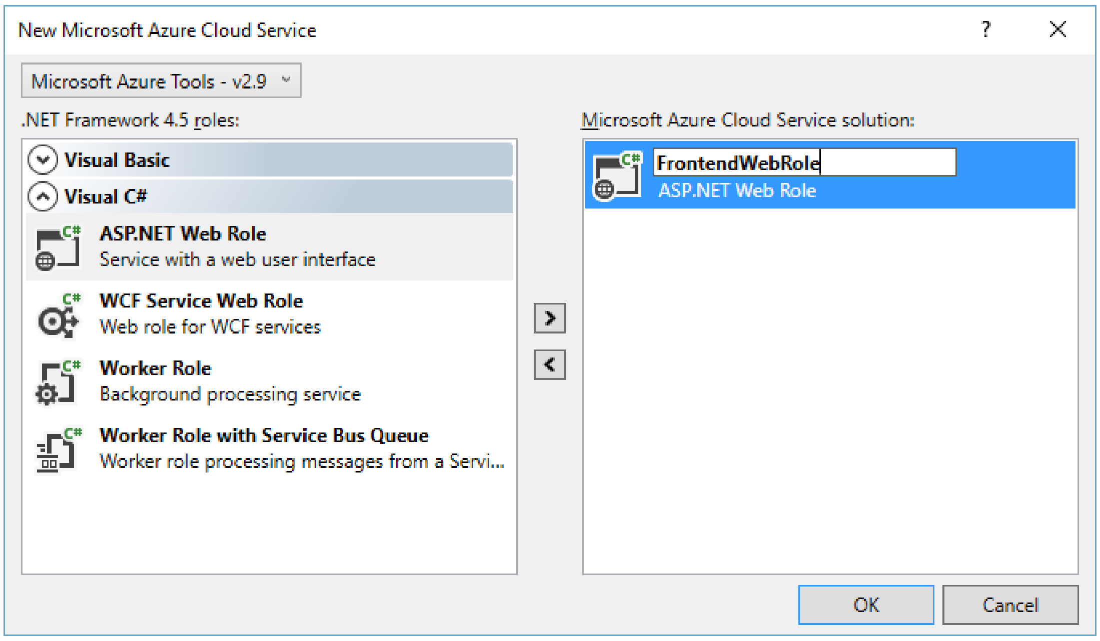
Worker Role
Background processing service

Worker Role with Service Bus Queue
Worker role processing messages from a Servi...

Microsoft Azure Cloud Service solution:

WebRole1
ASP.NET Web Role

4. Hover over **WebRole1** under **Azure Cloud Service solution**, click the pencil icon, and rename the web role to **FrontendWebRole**. Then click **OK**. (Make sure you enter "Frontend" with a lower-case 'e,' not "FrontEnd".)



New Microsoft Azure Cloud Service

Microsoft Azure Tools - v2.9

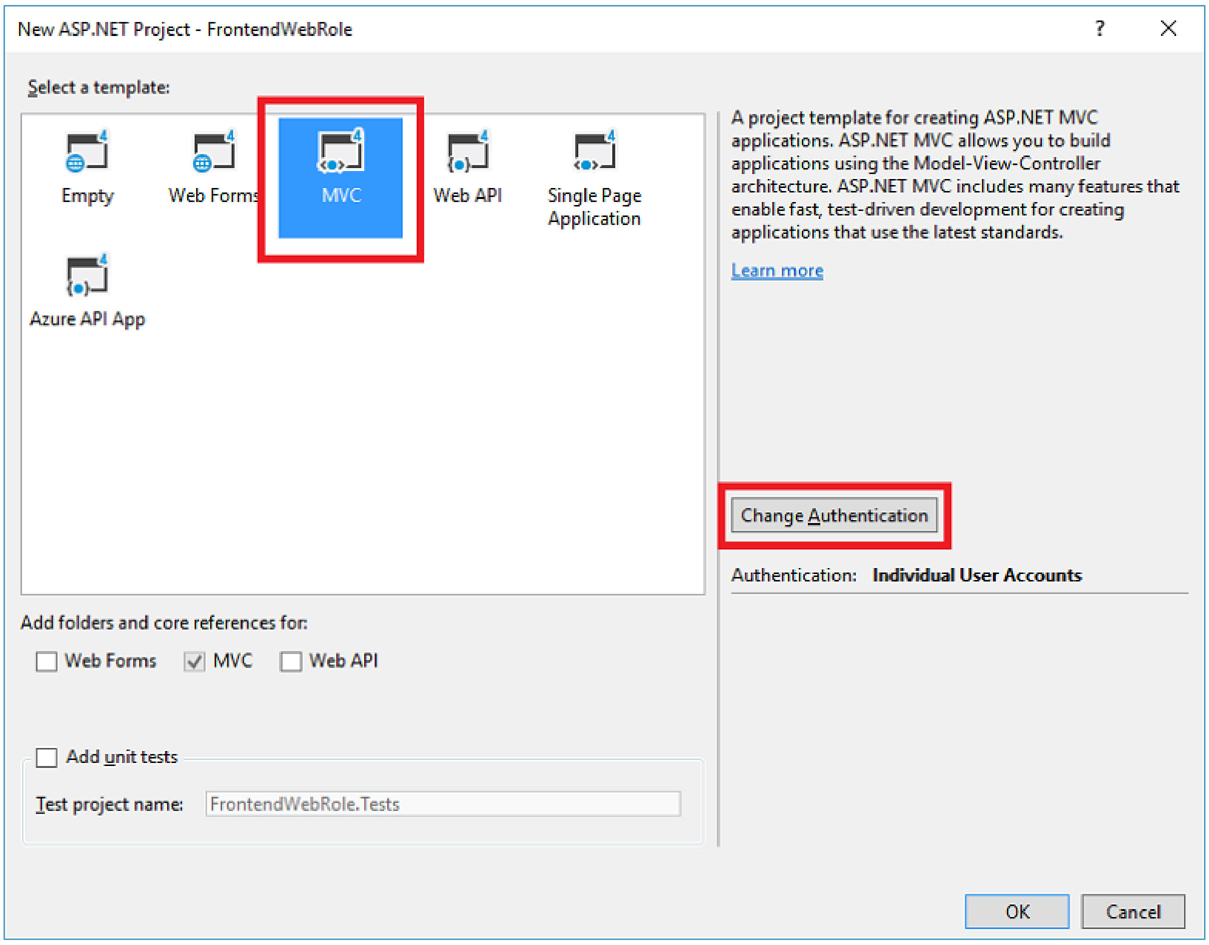
.NET Framework 4.5 roles:

- Visual Basic
- Visual C#
  - **ASP.NET Web Role** Service with a web user interface
  - **WCF Service Web Role** Web role for WCF services
  - **Worker Role** Background processing service
  - **Worker Role with Service Bus Queue** Worker role processing messages from a Servi...

Microsoft Azure Cloud Service solution:

FrontendWebRole
ASP.NET Web Role

5. From the **New ASP.NET Project** dialog box, in the **Select a template** list, click **MVC**.



New ASP.NET Project - FrontendWebRole

Select a template:

A project template for creating ASP.NET MVC applications. ASP.NET MVC allows you to build applications using the Model-View-Controller architecture. ASP.NET MVC includes many features that enable fast, test-driven development for creating applications that use the latest standards.

Learn more

Authentication: **Individual User Accounts**

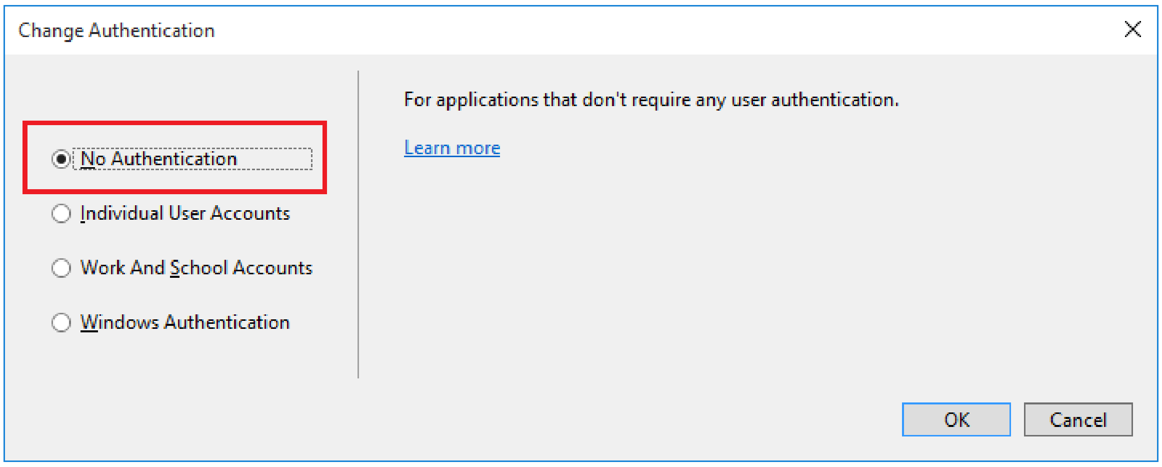
Add folders and core references for:

☐ Web Forms ☑ MVC ☐ Web API

☐ Add unit tests

Test project name: FrontendWebRole.Tests

6. Still in the **New ASP.NET Project** dialog box, click the **Change Authentication** button. In the **Change Authentication** dialog box, click **No Authentication**, and then click **OK**. For this tutorial, you're deploying an app that doesn't need a user login.



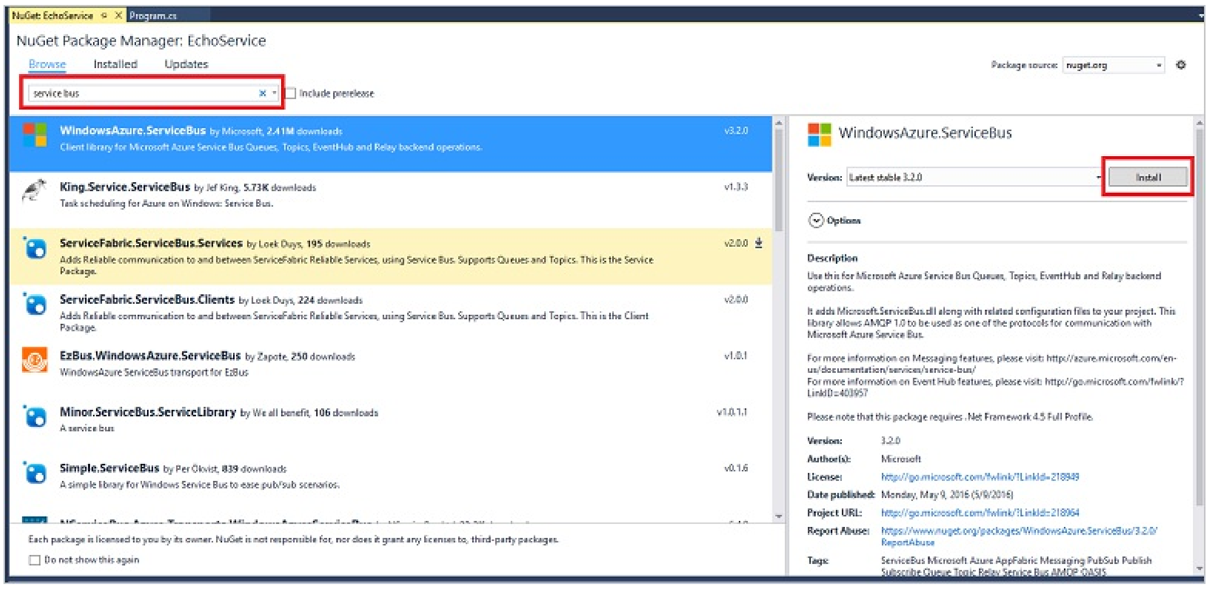
Change Authentication

(●) No Authentication
( ) Individual User Accounts
( ) Work And School Accounts
( ) Windows Authentication

For applications that don't require any user authentication.

Learn more

7. Back in the **New ASP.NET Project** dialog box, click **OK** to create the project.
8. In **Solution Explorer**, in the **FrontendWebRole** project, right-click **References**, then click **Manage NuGet Packages**.
9. Click the **Browse** tab, then search for `Microsoft Azure Service Bus`. Click **Install**, and accept the terms of use.



NuGet: EchoService   Program.cs

NuGet Package Manager: EchoService

service bus   [ ] Include prerelease

Package source: nuget.org

WindowsAzure.ServiceBus by Microsoft, 2.41M downloads   v3.2.0
Client library for Microsoft Azure Service Bus Queues, Topics, EventHub and Relay backend operations.

King.Service.ServiceBus by Jef King, 5.73K downloads   v1.3.3
Task scheduling for Azure on Windows: Service Bus.

ServiceFabric.ServiceBus.Services by Loek Duys, 195 downloads   v2.0.0
Adds Reliable communication to and between ServiceFabric Reliable Services, using Service Bus. Supports Queues and Topics. This is the Service Package.

ServiceFabric.ServiceBus.Clients by Loek Duys, 224 downloads   v2.0.0
Adds Reliable communication to and between ServiceFabric Reliable Services, using Service Bus. Supports Queues and Topics. This is the Client Package.

EzBus.WindowsAzure.ServiceBus by Zapote, 250 downloads   v1.0.1
WindowsAzure ServiceBus transport for EzBus

Minor.ServiceBus.ServiceLibrary by We all benefit, 106 downloads   v1.0.1.1
A service bus

Simple.ServiceBus by Per Ökvist, 839 downloads   v0.1.6
A simple library for Windows Service Bus to ease pub/sub scenarios.

Each package is licensed to you by its owner. NuGet is not responsible for, nor does it grant any licenses to, third-party packages.

[ ] Do not show this again

WindowsAzure.ServiceBus

Version: Latest stable 3.2.0   Install

Options

Description

Use this for Microsoft Azure Service Bus Queues, Topics, EventHub and Relay backend operations.

It adds Microsoft.ServiceBus.dll along with related configuration files to your project. This library allows AMQP 1.0 to be used as one of the protocols for communication with Microsoft Azure Service Bus.

For more information on Messaging features, please visit: http://azure.microsoft.com/en-us/documentation/services/service-bus/
For more information on Event Hub features, please visit: http://go.microsoft.com/fwlink/?LinkID=403957

Please note that this package requires .Net Framework 4.5 Full Profile.

Version:	3.2.0
Author(s):	Microsoft
License:	http://go.microsoft.com/fwlink/?LinkId=218949
Date published:	Monday, May 9, 2016 (5/9/2016)
Project URL:	http://go.microsoft.com/fwlink/?LinkId=218964
Report Abuse:	https://www.nuget.org/packages/WindowsAzure.ServiceBus/3.2.0/ReportAbuse
Tags:	ServiceBus Microsoft Azure AppFabric Messaging PubSub Publish Subscribe Queue Topic Relay Service Bus AMQP OASIS

Note that the required client assemblies are now referenced and some new code files have been added.

10. In **Solution Explorer**, right-click **Models** and click **Add**, then click **Class**. In the **Name** box, type the name **OnlineOrder.cs**. Then click **Add**.

Write the code for your web role

In this section, you create the various pages that your application displays.

1. In the OnlineOrder.cs file in Visual Studio, replace the existing namespace definition with the following code:

```
namespace FrontendWebRole.Models
{
    public class OnlineOrder
    {
        public string Customer { get; set; }
        public string Product { get; set; }
    }
}
```

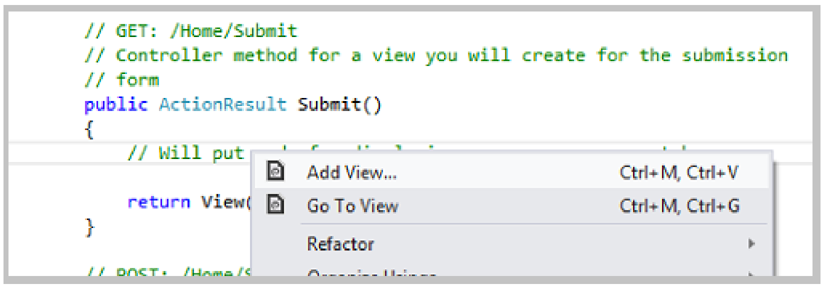
2. In **Solution Explorer**, double-click **Controllers\HomeController.cs**. Add the following **using** statements at the top of the file to include the namespaces for the model you just created, as well as Service Bus.

```
using FrontendWebRole.Models;
using Microsoft.ServiceBus.Messaging;
using Microsoft.ServiceBus;
```

3. Also in the HomeController.cs file in Visual Studio, replace the existing namespace definition with the following code. This code contains methods for handling the submission of items to the queue.

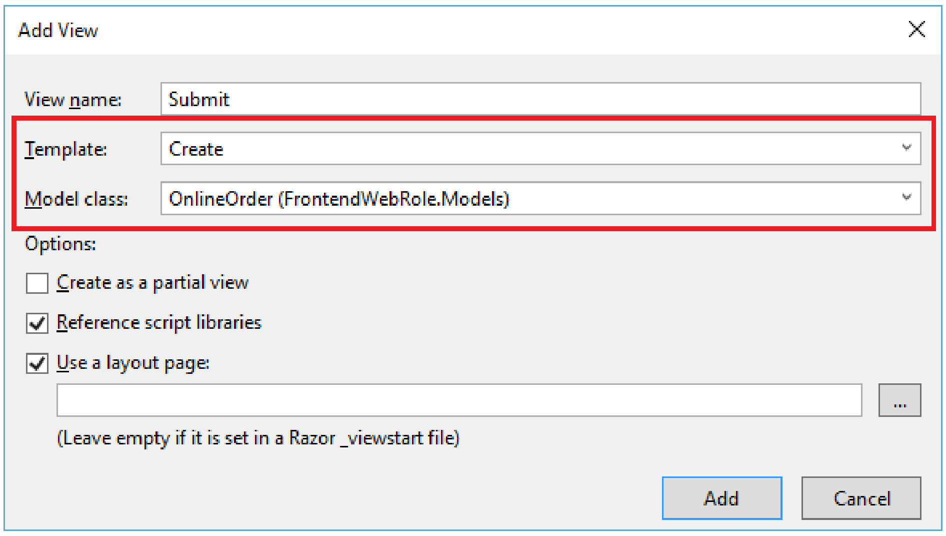
```
namespace FrontendWebRole.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            // Simply redirect to Submit, since Submit will serve as the
            // front page of this application.
            return RedirectToAction("Submit");
        }

        public ActionResult About()
        {
            return View();
        }

        // GET: /Home/Submit.
        // Controller method for a view you will create for the submission
        // form.
        public ActionResult Submit()
        {
            // Will put code for displaying queue message count here.

            return View();
        }

        // POST: /Home/Submit.
        // Controller method for handling submissions from the submission
        // form.
        [HttpPost]
        // Attribute to help prevent cross-site scripting attacks and
        // cross-site request forgery.
        [ValidateAntiForgeryToken]
        public ActionResult Submit(OnlineOrder order)
        {
            if (ModelState.IsValid)
            {
                // Will put code for submitting to queue here.

                return RedirectToAction("Submit");
            }
            else
            {
                return View(order);
            }
        }
    }
}
```

4. On the **Build** menu, click **Build Solution** to test the accuracy of your work so far.
5. Now, create the view for the `Submit()` method you created earlier. Right-click within the `Submit()` method (the overload of `Submit()` that takes no parameters), and then choose **Add View**.



```
// GET: /Home/Submit
// Controller method for a view you will create for the submission
// form
public ActionResult Submit()
{
    // Will put
    return View(
}
```

6. A dialog box appears for creating the view. In the **Template** list, choose **Create**. In the **Model class** list, click the **OnlineOrder** class.



Add View

View name: Submit

Template: Create

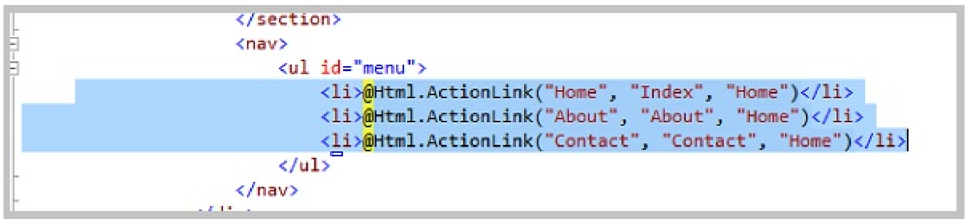
Model class: OnlineOrder (FrontendWebRole.Models)

Options:

- [ ] Create as a partial view
- [✓] Reference script libraries
- [✓] Use a layout page:

(Leave empty if it is set in a Razor _viewstart file)

7. Click **Add**.
8. Now, change the displayed name of your application. In **Solution Explorer**, double-click the **Views\Shared_Layout.cshtml** file to open it in the Visual Studio editor.
9. Replace all occurrences of **My ASP.NET Application** with **LITWARE'S Products**.
10. Remove the **Home**, **About**, and **Contact** links. Delete the highlighted code:

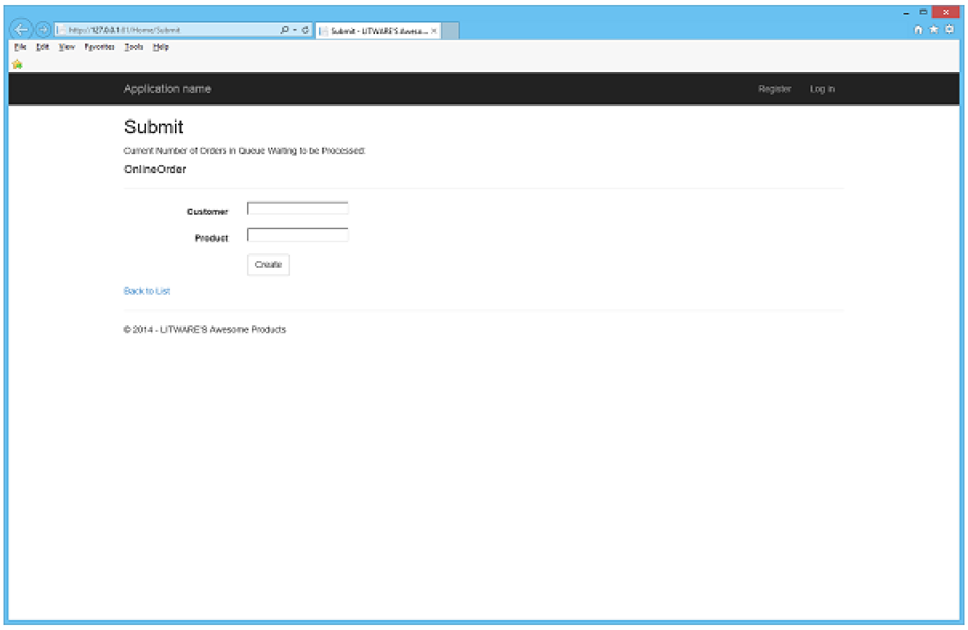


```
        </section>
        <nav>
            <ul id="menu">
                <li>@Html.ActionLink("Home", "Index", "Home")</li>
                <li>@Html.ActionLink("About", "About", "Home")</li>
                <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
            </ul>
        </nav>
```

11. Finally, modify the submission page to include some information about the queue. In **Solution Explorer**, double-click the **Views\Home\Submit.cshtml** file to open it in the Visual Studio editor. Add the following line after `<h2>Submit</h2>`. For now, the `ViewBag.MessageCount` is empty. You will populate it later.

```
<p>Current number of orders in queue waiting to be processed: @ViewBag.MessageCount</p>
```

12. You now have implemented your UI. You can press **F5** to run your application and confirm that it looks as expected.


http://127.0.0.1:81/Home/Submit
Submit - LITWARE'S Aweso...
File Edit View Favorites Tools Help
Application name
Register
Log in

Submit

Current Number of Orders in Queue Waiting to be Processed:

OnlineOrder

Customer

Product

© 2014 - LITWARE'S Awesome Products

Write the code for submitting items to a Service Bus queue

Now, add code for submitting items to a queue. First, you create a class that contains your Service Bus queue connection information. Then, initialize your connection from Global.aspx.cs. Finally, update the submission code you created earlier in HomeController.cs to actually submit items to a Service Bus queue.

1. In **Solution Explorer**, right-click **FrontendWebRole** (right-click the project, not the role). Click **Add**, and then click **Class**.
2. Name the class **QueueConnector.cs**. Click **Add** to create the class.
3. Now, add code that encapsulates the connection information and initializes the connection to a Service Bus queue. Replace the entire contents of QueueConnector.cs with the following code, and enter values for `your Service Bus namespace` (your namespace name) and `yourKey`, which is the **primary key** you previously obtained from the Azure portal.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using Microsoft.ServiceBus.Messaging;
using Microsoft.ServiceBus;

namespace FrontendWebRole
{
    public static class QueueConnector
    {
        // Thread-safe. Recommended that you cache rather than recreating it
        // on every request.
        public static QueueClient OrdersQueueClient;

        // Obtain these values from the portal.
        public const string Namespace = "your Service Bus namespace";

        // The name of your queue.
        public const string QueueName = "OrdersQueue";

        public static NamespaceManager CreateNamespaceManager()
        {
            // Create the namespace manager which gives you access to
            // management operations.
            var uri = ServiceBusEnvironment.CreateServiceUri(
                "sb", Namespace, String.Empty);
            var tP = TokenProvider.CreateSharedAccessSignatureTokenProvider(
                "RootManageSharedAccessKey", "yourKey");
            return new NamespaceManager(uri, tP);
        }

        public static void Initialize()
        {
            // Using Http to be friendly with outbound firewalls.
            ServiceBusEnvironment.SystemConnectivity.Mode =
                ConnectivityMode.Http;

            // Create the namespace manager which gives you access to
            // management operations.
            var namespaceManager = CreateNamespaceManager();

            // Create the queue if it does not exist already.
            if (!namespaceManager.QueueExists(QueueName))
            {
                namespaceManager.CreateQueue(QueueName);
            }

            // Get a client to the queue.
            var messagingFactory = MessagingFactory.Create(
                namespaceManager.Address,
                namespaceManager.Settings.TokenProvider);
            OrdersQueueClient = messagingFactory.CreateQueueClient(
                "OrdersQueue");
        }
    }
}
```

4. Now, ensure that your **Initialize** method gets called. In **Solution Explorer**, double-click **Global.asax\Global.asax.cs**.
5. Add the following line of code at the end of the **Application_Start** method.

```
FrontendWebRole.QueueConnector.Initialize();
```

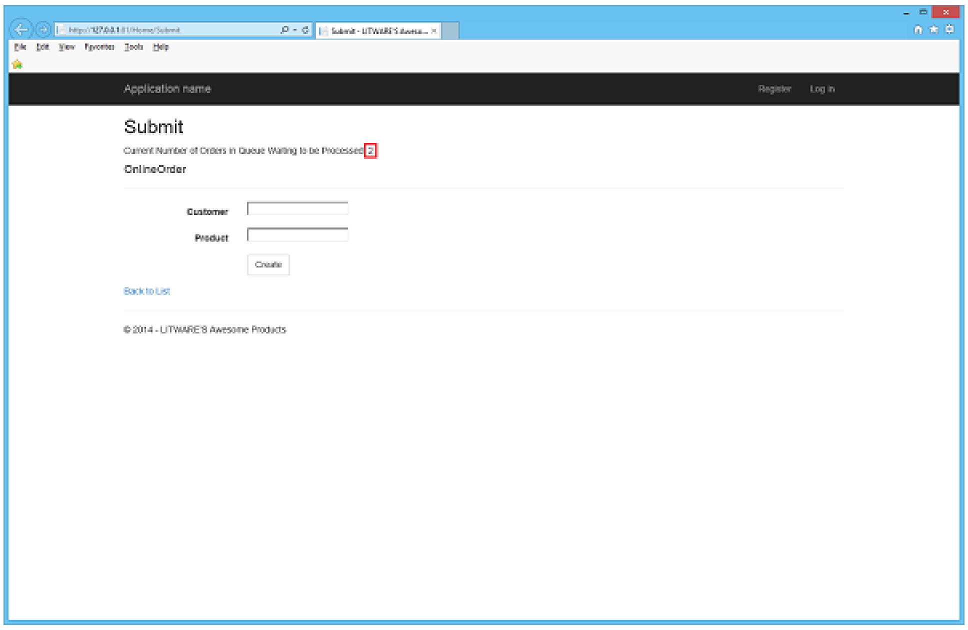
6. Finally, update the web code you created earlier, to submit items to the queue. In **Solution Explorer**, double-click **Controllers\HomeController.cs**.
7. Update the `Submit()` method (the overload that takes no parameters) as follows to get the message count for the queue.

```
public ActionResult Submit()
{
    // Get a NamespaceManager which allows you to perform management and
    // diagnostic operations on your Service Bus queues.
    var namespaceManager = QueueConnector.CreateNamespaceManager();

    // Get the queue, and obtain the message count.
    var queue = namespaceManager.GetQueue(QueueConnector.QueueName);
    ViewBag.MessageCount = queue.MessageCount;

    return View();
}
```

8. Update the `Submit(OnlineOrder order)` method (the overload that takes one parameter) as follows to submit order information to the queue.

```
public ActionResult Submit(OnlineOrder order)
{
    if (ModelState.IsValid)
    {
        // Create a message from the order.
        var message = new BrokeredMessage(order);

        // Submit the order.
        QueueConnector.OrdersQueueClient.Send(message);
        return RedirectToAction("Submit");
    }
    else
    {
        return View(order);
    }
}
```

9. You can now run the application again. Each time you submit an order, the message count increases.


http://127.0.0.1:81/Home/Submit
Submit - LITWARE'S Aweso...
File Edit View Favorites Tools Help
Application name
Register
Log in

Submit

Current Number of Orders in Queue Waiting to be Processed: 2

OnlineOrder

Customer

Product

Create

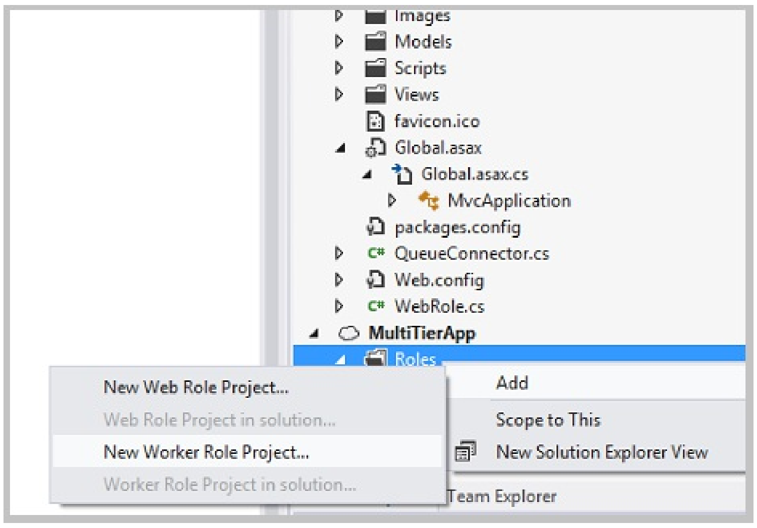
Back to List

© 2014 - LITWARE'S Awesome Products

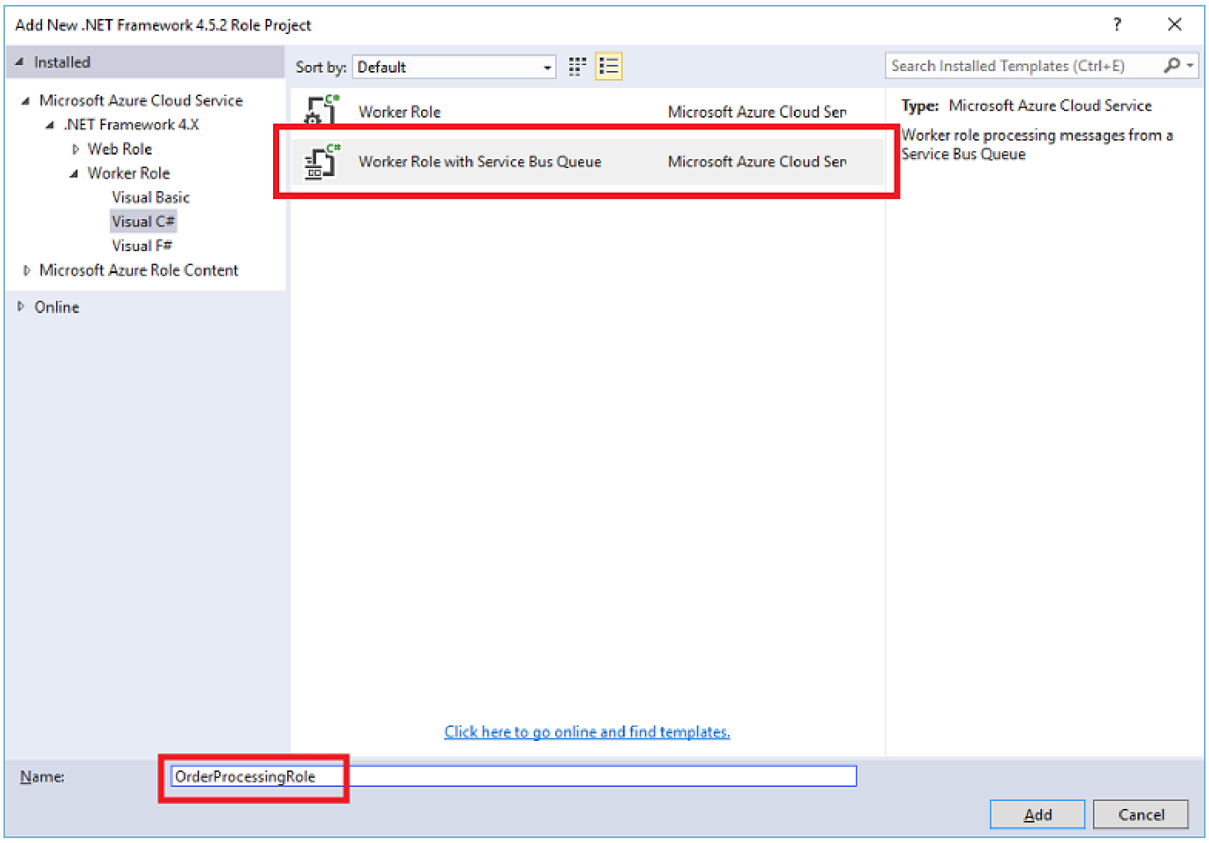
Create the worker role

You will now create the worker role that processes the order submissions. This example uses the **Worker Role with Service Bus Queue** Visual Studio project template. You already obtained the required credentials from the portal.

1. Make sure you have connected Visual Studio to your Azure account.
2. In Visual Studio, in **Solution Explorer** right-click the **Roles** folder under the **MultiTierApp** project.
3. Click **Add**, and then click **New Worker Role Project**. The **Add New Role Project** dialog box appears.



4. In the **Add New Role Project** dialog box, click **Worker Role with Service Bus Queue**.



Add New .NET Framework 4.5.2 Role Project
?
Installed
Microsoft Azure Cloud Service
.NET Framework 4.X
Web Role
Worker Role
Visual Basic
Visual C#
Visual F#
Microsoft Azure Role Content
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
Worker Role
Microsoft Azure Cloud Ser
Worker Role with Service Bus Queue
Microsoft Azure Cloud Ser
Click here to go online and find templates.
Add
Cancel

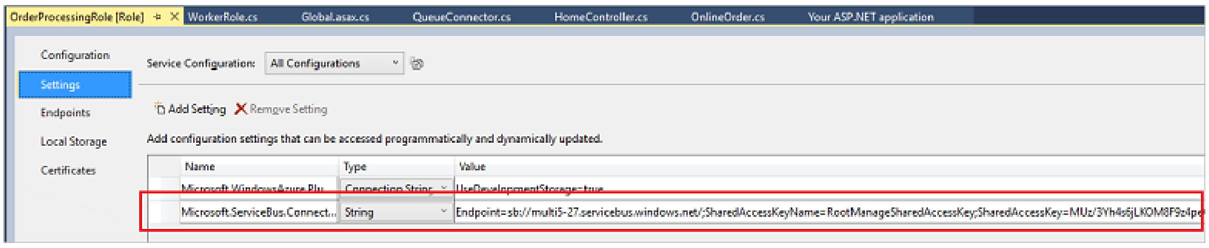
Type: Microsoft Azure Cloud Service

Worker role processing messages from a Service Bus Queue

Name: OrderProcessingRole

5. In the **Name** box, name the project **OrderProcessingRole**. Then click **Add**.
6. Copy the connection string that you obtained in step 9 of the "Create a Service Bus namespace" section to the clipboard.
7. In **Solution Explorer**, right-click the **OrderProcessingRole** you created in step 5 (make sure that you right-click **OrderProcessingRole** under **Roles**, and not the class). Then click **Properties**.

8. On the **Settings** tab of the **Properties** dialog box, click inside the **Value** box for **Microsoft.ServiceBus.ConnectionString**, and then paste the endpoint value you copied in step 6.



Service Configuration: All Configurations

Add configuration settings that can be accessed programmatically and dynamically updated.

Name	Type	Value
Microsoft.WindowsAzure.Plu	Connection String	UseDevelopmentStorage=true
Microsoft.ServiceBus.Connect...	String	Endpoint=sb://multi5-27.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=MUz/3Yh4s6jLKOM8F9z4pe

9. Create an **OnlineOrder** class to represent the orders as you process them from the queue. You can reuse a class you have already created. In **Solution Explorer**, right-click the **OrderProcessingRole** class (right-click the class icon, not the role). Click **Add**, then click **Existing Item**.
10. Browse to the subfolder for **FrontendWebRole\Models**, and then double-click **OnlineOrder.cs** to add it to this project.
11. In **WorkerRole.cs**, change the value of the **QueueName** variable from `"ProcessingQueue"` to `"OrdersQueue"` as shown in the following code.

```
// The name of your queue.
const string QueueName = "OrdersQueue";
```

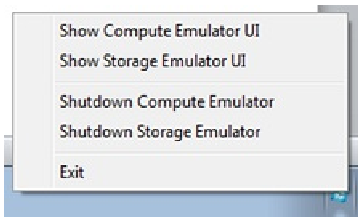
12. Add the following using statement at the top of the WorkerRole.cs file.

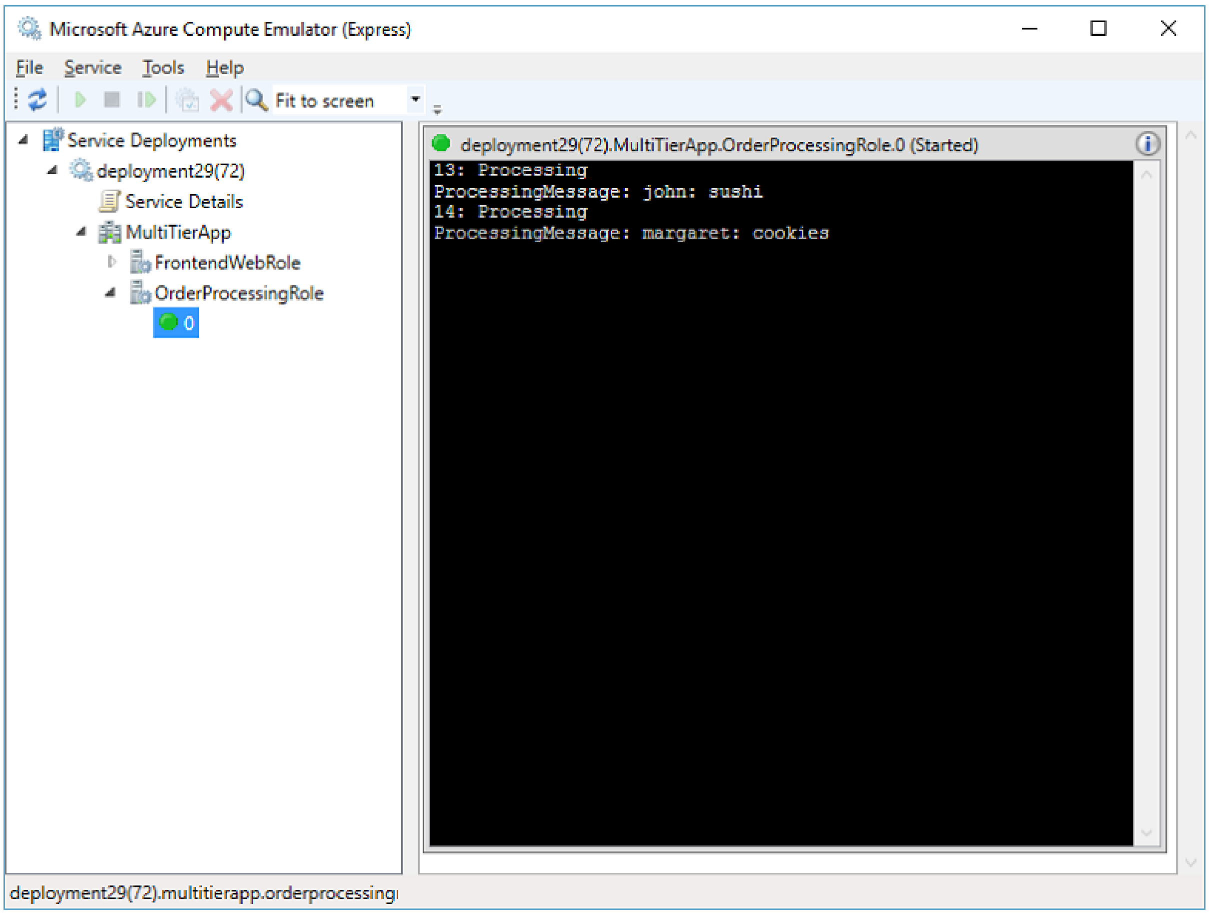
```
using FrontendWebRole.Models;
```

13. In the `Run()` function, inside the `OnMessage()` call, replace the contents of the `try` clause with the following code.

```
Trace.WriteLine("Processing", receivedMessage.SequenceNumber.ToString());
// View the message as an OnlineOrder.
OnlineOrder order = receivedMessage.GetBody<OnlineOrder>();
Trace.WriteLine(order.Customer + ": " + order.Product, "ProcessingMessage");
receivedMessage.Complete();
```

14. You have completed the application. You can test the full application by right-clicking the MultiTierApp project in Solution Explorer, selecting **Set as Startup Project**, and then pressing F5. Note that the message count does not increment, because the worker role processes items from the queue and marks them as complete. You can see the trace output of your worker role by viewing the Azure Compute Emulator UI. You can do this by right-clicking the emulator icon in the notification area of your taskbar and selecting **Show Compute Emulator UI**.




Microsoft Azure Compute Emulator (Express)
File Service Tools Help
Fit to screen
Service Deployments
deployment29(72)
Service Details
MultiTierApp
FrontendWebRole
OrderProcessingRole
0
deployment29(72).MultiTierApp.OrderProcessingRole.0 (Started)
deployment29(72).multitierapp.orderprocessing

```
13: Processing
ProcessingMessage: john: sushi
14: Processing
ProcessingMessage: margaret: cookies
```

Next steps

To learn more about Service Bus, see the following resources:

- Azure Service Bus
- Service Bus service page
- How to Use Service Bus Queues

To learn more about multi-tier scenarios, see:

- .NET Multi-Tier Application Using Storage Tables, Queues, and Blobs

Service Bus Premium and Standard messaging tiers

1/19/2017 • 2 min to read • Edit on GitHub

Service Bus Messaging, which includes entities such as queues and topics, combines enterprise messaging capabilities with rich publish-subscribe semantics at cloud scale. Service Bus Messaging is used as the communication backbone for many sophisticated cloud solutions.

The *Premium* tier of Service Bus Messaging addresses common customer requests around scale, performance, and availability for mission-critical applications. Although the feature sets are nearly identical, these two tiers of Service Bus Messaging are designed to serve different use cases.

Some high-level differences are highlighted in the following table.

PREMIUM	STANDARD
High throughput	Variable throughput
Predictable performance	Variable latency
Fixed pricing	Pay as you go variable pricing
Ability to scale workload up and down	N/A
Message size up to 1 MB	Message size up to 256 KB

Service Bus Premium Messaging provides resource isolation at the CPU and memory layer so that each customer workload runs in isolation. This resource container is called a *messaging unit*. Each premium namespace is allocated at least one messaging unit. You can purchase 1, 2, or 4 messaging units for each Service Bus Premium namespace. A single workload or entity can span multiple messaging units and the number of messaging units can be changed at will, although billing is in 24-hour or daily rate charges. The result is predictable and repeatable performance for your Service Bus-based solution.

Not only is this performance more predictable and available, but it is also faster. Service Bus Premium Messaging builds on the storage engine introduced in Azure Event Hubs. With Premium Messaging, peak performance is much faster than with the Standard tier.

Premium Messaging technical differences

The following are a few differences between Premium and Standard messaging tiers.

Partitioned queues and topics

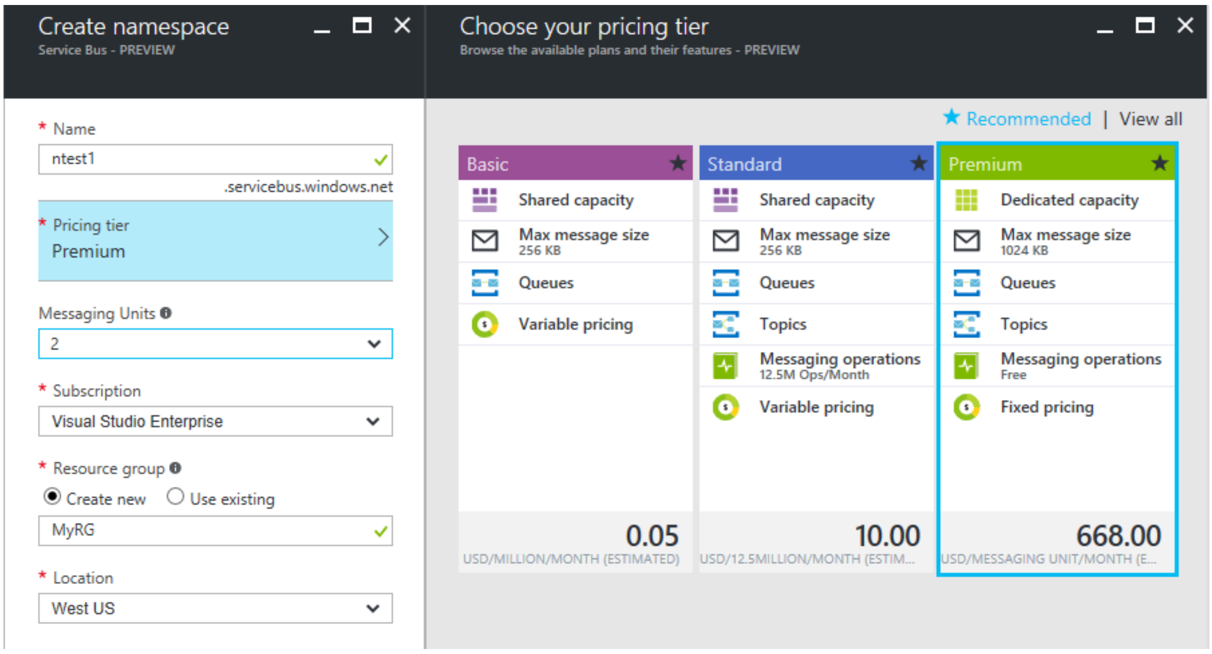
Partitioned queues and topics are supported in Premium Messaging, but they do not function the same way as in the Standard and Basic tiers of Service Bus Messaging. Premium Messaging does not use SQL as a data store and no longer has the possible resource competition associated with a shared platform. As a result, partitioning is not necessary for performance. Additionally, the partition count has been changed from 16 partitions in Standard Messaging to 2 partitions in Premium. Having two partitions ensures availability and is a more appropriate number for the Premium runtime environment. For more information about partitioning, see Partitioned queues and topics.

Express entities

Because Premium Messaging runs in a completely isolated runtime environment, express entities are not supported in Premium namespaces. For more information about the express feature, see the QueueDescription.EnableExpress property.

Get started with Premium Messaging

Getting started with Premium Messaging is straightforward and the process is similar to that of Standard Messaging. Begin by creating a namespace. Make sure you select **Premium** under **Pricing tier**.



Create namespace
Service Bus - PREVIEW

* Name
ntest1
.servicebus.windows.net

* Pricing tier
Premium

Messaging Units
2

* Subscription
Visual Studio Enterprise

* Resource group
(● Create new) ○ Use existing
MyRG

* Location
West US

Choose your pricing tier
Browse the available plans and their features - PREVIEW

Recommended | View all

Basic	Standard	Premium
Shared capacity	Shared capacity	Dedicated capacity
Max message size 256 KB	Max message size 256 KB	Max message size 1024 KB
Queues	Queues	Queues
Variable pricing	Topics	Topics
	Messaging operations 12.5M Ops/Month	Messaging operations Free
	Variable pricing	Fixed pricing
0.05 USD/MILLION/MONTH (ESTIMATED)	10.00 USD/12.5MILLION/MONTH (ESTIM...	668.00 USD/MESSAGING UNIT/MONTH (E...

You can also create Premium namespaces using Azure Resource Manager templates.

Next steps

To learn more about Service Bus Messaging, see the following topics.

- Introducing Azure Service Bus Premium Messaging (blog post)
- Introducing Azure Service Bus Premium Messaging (Channel9)
- Service Bus Messaging overview
- How to use Service Bus queues

Storage queues and Service Bus queues - compared and contrasted

2/27/2017 • 15 min to read • Edit on GitHub

This article analyzes the differences and similarities between the two types of queues offered by Microsoft Azure today: Storage queues and Service Bus queues. By using this information, you can compare and contrast the respective technologies and be able to make a more informed decision about which solution best meets your needs.

Introduction

Microsoft Azure supports two types of queue mechanisms: **Storage queues** and **Service Bus queues**.

Storage queues, which are part of the Azure storage infrastructure, feature a simple REST-based Get/Put/Peek interface, providing reliable, persistent messaging within and between services.

Service Bus queues are part of a broader Azure messaging infrastructure that supports queuing as well as publish/subscribe, and more advanced integration patterns. For more information about Service Bus queues/topics/subscriptions, see the overview of Service Bus.

While both queuing technologies exist concurrently, Storage queues were introduced first, as a dedicated queue storage mechanism built on top of the Azure storage services. Service Bus queues are built on top of the broader "messaging" infrastructure designed to integrate applications or application components that may span multiple communication protocols, data contracts, trust domains, and/or network environments.

Technology selection considerations

Both Storage queues and Service Bus queues are implementations of the message queuing service currently offered on Microsoft Azure. Each has a slightly different feature set, which means you can choose one or the other, or use both, depending on the needs of your particular solution or business/technical problem you are solving.

When determining which queuing technology fits the purpose for a given solution, solution architects and developers should consider the recommendations below. For more details, see the next section.

As a solution architect/developer, **you should consider using Storage queues** when:

- Your application must store over 80 GB of messages in a queue, where the messages have a lifetime shorter than 7 days.
- Your application wants to track progress for processing a message inside of the queue. This is useful if the worker processing a message crashes. A subsequent worker can then use that information to continue from where the prior worker left off.
- You require server side logs of all of the transactions executed against your queues.

As a solution architect/developer, **you should consider using Service Bus queues** when:

- Your solution must be able to receive messages without having to poll the queue. With Service Bus, this can be achieved through the use of the long-polling receive operation using the TCP-based protocols that Service Bus supports.
- Your solution requires the queue to provide a guaranteed first-in-first-out (FIFO) ordered delivery.
- You want a symmetric experience in Azure and on Windows Server (private cloud). For more information, see Service Bus for Windows Server.

- Your solution must be able to support automatic duplicate detection.
- You want your application to process messages as parallel long-running streams (messages are associated with a stream using the SessionId property on the message). In this model, each node in the consuming application competes for streams, as opposed to messages. When a stream is given to a consuming node, the node can examine the state of the application stream state using transactions.
- Your solution requires transactional behavior and atomicity when sending or receiving multiple messages from a queue.
- The time-to-live (TTL) characteristic of the application-specific workload can exceed the 7-day period.
- Your application handles messages that can exceed 64 KB but will not likely approach the 256 KB limit.
- You deal with a requirement to provide a role-based access model to the queues, and different rights/permissions for senders and receivers.
- Your queue size will not grow larger than 80 GB.
- You want to use the AMQP 1.0 standards-based messaging protocol. For more information about AMQP, see Service Bus AMQP Overview.
- You can envision an eventual migration from queue-based point-to-point communication to a message exchange pattern that enables seamless integration of additional receivers (subscribers), each of which receives independent copies of either some or all messages sent to the queue. The latter refers to the publish/subscribe capability natively provided by Service Bus.
- Your messaging solution must be able to support the "At-Most-Once" delivery guarantee without the need for you to build the additional infrastructure components.
- You would like to be able to publish and consume batches of messages.

Comparing Storage queues and Service Bus queues

The tables in the following sections provide a logical grouping of queue features and let you compare, at a glance, the capabilities available in both Storage queues and Service Bus queues.

Foundational capabilities

This section compares some of the fundamental queuing capabilities provided by Storage queues and Service Bus queues.

COMPARISON CRITERIA	STORAGE QUEUES	SERVICE BUS QUEUES
Ordering guarantee	**No** For more information, see the first note in the “Additional Information” section.	**Yes - First-In-First-Out (FIFO)** (through the use of messaging sessions)
Delivery guarantee	**At-Least-Once**	**At-Least-Once** **At-Most-Once**
Atomic operation support	**No**	**Yes**

COMPARISON CRITERIA	STORAGE QUEUES	SERVICE BUS QUEUES
Receive behavior	**Non-blocking** (completes immediately if no new message is found)	**Blocking with/without timeout** (offers long polling, or the "Comet technique") **Non-blocking** (through the use of .NET managed API only)
Push-style API	**No**	**Yes** OnMessage and **OnMessage** sessions .NET API.
Receive mode	**Peek & Lease**	**Peek & Lock** **Receive & Delete**
Exclusive access mode	**Lease-based**	**Lock-based**
Lease/Lock duration	**30 seconds (default)** **7 days (maximum)** (You can renew or release a message lease using the UpdateMessage API.)	**60 seconds (default)** You can renew a message lock using the RenewLock API.
Lease/Lock precision	**Message level** (each message can have a different timeout value, which you can then update as needed while processing the message, by using the UpdateMessage API)	**Queue level** (each queue has a lock precision applied to all of its messages, but you can renew the lock using the RenewLock API.)
Batched receive	**Yes** (explicitly specifying message count when retrieving messages, up to a maximum of 32 messages)	**Yes** (implicitly enabling a pre-fetch property or explicitly through the use of transactions)
Batched send	**No**	**Yes** (through the use of transactions or client-side batching)

Additional information

- Messages in Storage queues are typically first-in-first-out, but sometimes they can be out of order; for example, when a message's visibility timeout duration expires (for example, as a result of a client application crashing during processing). When the visibility timeout expires, the message becomes visible again on the queue for another worker to dequeue it. At that point, the newly visible message might be placed in the queue (to be dequeued again) after a message that was originally enqueued after it.
- The guaranteed FIFO pattern in Service Bus queues requires the use of messaging sessions. In the event that the application crashes while processing a message received in the **Peek & Lock** mode, the next time a queue receiver accepts a messaging session, it will start with the failed message after its time-to-live (TTL) period expires.

- Storage queues are designed to support standard queuing scenarios, such as decoupling application components to increase scalability and tolerance for failures, load leveling, and building process workflows.
- Service Bus queues support the *At-Least-Once* delivery guarantee. In addition, the *At-Most-Once* semantic can be supported by using session state to store the application state and by using transactions to atomically receive messages and update the session state.
- Storage queues provide a uniform and consistent programming model across queues, tables, and BLOBs – both for developers and for operations teams.
- Service Bus queues provide support for local transactions in the context of a single queue.
- The **Receive and Delete** mode supported by Service Bus provides the ability to reduce the messaging operation count (and associated cost) in exchange for lowered delivery assurance.
- Storage queues provide leases with the ability to extend the leases for messages. This allows the workers to maintain short leases on messages. Thus, if a worker crashes, the message can be quickly processed again by another worker. In addition, a worker can extend the lease on a message if it needs to process it longer than the current lease time.
- Storage queues offer a visibility timeout that you can set upon the enqueueing or dequeuing of a message. In addition, you can update a message with different lease values at run-time, and update different values across messages in the same queue. Service Bus lock timeouts are defined in the queue metadata; however, you can renew the lock by calling the RenewLock method.
- The maximum timeout for a blocking receive operation in Service Bus queues is 24 days. However, REST-based timeouts have a maximum value of 55 seconds.
- Client-side batching provided by Service Bus enables a queue client to batch multiple messages into a single send operation. Batching is only available for asynchronous send operations.
- Features such as the 200 TB ceiling of Storage queues (more when you virtualize accounts) and unlimited queues make it an ideal platform for SaaS providers.
- Storage queues provide a flexible and performant delegated access control mechanism.

Advanced capabilities

This section compares advanced capabilities provided by Storage queues and Service Bus queues.

COMPARISON CRITERIA	STORAGE QUEUES	SERVICE BUS QUEUES
Scheduled delivery	**Yes**	**Yes**
Automatic dead lettering	**No**	**Yes**
Increasing queue time-to-live value	**Yes** (via in-place update of visibility timeout)	**Yes** (provided via a dedicated API function)
Poison message support	**Yes**	**Yes**
In-place update	**Yes**	**Yes**
Server-side transaction log	**Yes**	**No**

COMPARISON CRITERIA	STORAGE QUEUES	SERVICE BUS QUEUES
Storage metrics	**Yes** **Minute Metrics**: provides real-time metrics for availability, TPS, API call counts, error counts, and more, all in real time (aggregated per minute and reported within a few minutes from what just happened in production. For more information, see About Storage Analytics Metrics.	**Yes** (bulk queries by calling GetQueues)
State management	**No**	**Yes** Microsoft.ServiceBus.Messaging.EntityStatus.Active, Microsoft.ServiceBus.Messaging.EntityStatus.Disabled, Microsoft.ServiceBus.Messaging.EntityStatus.SendDisabled, Microsoft.ServiceBus.Messaging.EntityStatus.ReceiveDisabled
Message auto-forwarding	**No**	**Yes**
Purge queue function	**Yes**	**No**
Message groups	**No**	**Yes** (through the use of messaging sessions)
Application state per message group	**No**	**Yes**
Duplicate detection	**No**	**Yes** (configurable on the sender side)
Browsing message groups	**No**	**Yes**
Fetching message sessions by ID	**No**	**Yes**

Additional information

- Both queuing technologies enable a message to be scheduled for delivery at a later time.
- Queue auto-forwarding enables thousands of queues to auto-forward their messages to a single queue, from which the receiving application consumes the message. You can use this mechanism to achieve security, control flow, and isolate storage between each message publisher.
- Storage queues provide support for updating message content. You can use this functionality for persisting state information and incremental progress updates into the message so that it can be processed from the last known checkpoint, instead of starting from scratch. With Service Bus queues, you can enable the same scenario through the use of message sessions. Sessions enable you to save and retrieve the application processing state (by using SetState and GetState).
- Dead lettering, which is only supported by Service Bus queues, can be useful for isolating messages that cannot be processed successfully by the receiving application or when messages cannot reach their destination due to an expired time-to-live (TTL) property. The TTL value specifies how long a message remains in the queue. With

Service Bus, the message will be moved to a special queue called $DeadLetterQueue when the TTL period expires.

- To find "poison" messages in Storage queues, when dequeuing a message the application examines the **DequeueCount** property of the message. If **DequeueCount** is above a given threshold, the application moves the message to an application-defined "dead letter" queue.
- Storage queues enable you to obtain a detailed log of all of the transactions executed against the queue, as well as aggregated metrics. Both of these options are useful for debugging and understanding how your application uses Storage queues. They are also useful for performance-tuning your application and reducing the costs of using queues.
- The concept of "message sessions" supported by Service Bus enables messages that belong to a certain logical group to be associated with a given receiver, which in turn creates a session-like affinity between messages and their respective receivers. You can enable this advanced functionality in Service Bus by setting the SessionID property on a message. Receivers can then listen on a specific session ID and receive messages that share the specified session identifier.
- The duplication detection functionality supported by Service Bus queues automatically removes duplicate messages sent to a queue or topic, based on the value of the MessageId property.

Capacity and quotas

This section compares Storage queues and Service Bus queues from the perspective of capacity and quotas that may apply.

COMPARISON CRITERIA	STORAGE QUEUES	SERVICE BUS QUEUES
Maximum queue size	**200 TB** (limited to a single storage account capacity)	**1 GB to 80 GB** (defined upon creation of a queue and enabling partitioning – see the “Additional Information” section)
Maximum message size	**64 KB** (48 KB when using **Base64** encoding) Azure supports large messages by combining queues and blobs – at which point you can enqueue up to 200GB for a single item.	**256 KB** or **1 MB** (including both header and body, maximum header size: 64 KB). Depends on the service tier.
Maximum message TTL	**7 days**	`TimeSpan.Max`
Maximum number of queues	**Unlimited**	**10,000** (per service namespace, can be increased)
Maximum number of concurrent clients	**Unlimited**	**Unlimited** (100 concurrent connection limit only applies to TCP protocol-based communication)

Additional information

- Service Bus enforces queue size limits. The maximum queue size is specified upon creation of the queue and can have a value between 1 and 80 GB. If the queue size value set on creation of the queue is reached, additional incoming messages will be rejected and an exception will be received by the calling code. For more information about quotas in Service Bus, see Service Bus Quotas.

- You can create Service Bus queues in 1, 2, 3, 4, or 5 GB sizes (the default is 1 GB). With partitioning enabled (which is the default), Service Bus creates 16 partitions for each GB you specify. As such, if you create a queue that is 5 GB in size, with 16 partitions the maximum queue size becomes (5 * 16) = 80 GB. You can see the maximum size of your partitioned queue or topic by looking at its entry on the Azure portal.
- With Storage queues, if the content of the message is not XML-safe, then it must be **Base64** encoded. If you **Base64**-encode the message, the user payload can be up to 48 KB, instead of 64 KB.
- With Service Bus queues, each message stored in a queue is composed of two parts: a header and a body. The total size of the message cannot exceed the maximum message size supported by the service tier.
- When clients communicate with Service Bus queues over the TCP protocol, the maximum number of concurrent connections to a single Service Bus queue is limited to 100. This number is shared between senders and receivers. If this quota is reached, subsequent requests for additional connections will be rejected and an exception will be received by the calling code. This limit is not imposed on clients connecting to the queues using REST-based API.
- If you require more than 10,000 queues in a single Service Bus namespace, you can contact the Azure support team and request an increase. To scale beyond 10,000 queues with Service Bus, you can also create additional namespaces using the Azure portal.

Management and operations

This section compares the management features provided by Storage queues and Service Bus queues.

COMPARISON CRITERIA	STORAGE QUEUES	SERVICE BUS QUEUES
Management protocol	**REST over HTTP/HTTPS**	**REST over HTTPS**
Runtime protocol	**REST over HTTP/HTTPS**	**REST over HTTPS** **AMQP 1.0 Standard (TCP with TLS)**
.NET API	**Yes** (.NET Storage Client API)	**Yes** (.NET Service Bus API)
Native C++	**Yes**	**Yes**
Java API	**Yes**	**Yes**
PHP API	**Yes**	**Yes**
Node.js API	**Yes**	**Yes**
Arbitrary metadata support	**Yes**	**No**
Queue naming rules	**Up to 63 characters long** (Letters in a queue name must be lowercase.)	**Up to 260 characters long** (Queue paths and names are case-insensitive.)
Get queue length function	**Yes** (Approximate value if messages expire beyond the TTL without being deleted.)	**Yes** (Exact, point-in-time value.)

COMPARISON CRITERIA	STORAGE QUEUES	SERVICE BUS QUEUES
Peek function	**Yes**	**Yes**

Additional information

- Storage queues provide support for arbitrary attributes that can be applied to the queue description, in the form of name/value pairs.
- Both queue technologies offer the ability to peek a message without having to lock it, which can be useful when implementing a queue explorer/browser tool.
- The Service Bus .NET brokered messaging APIs leverage full-duplex TCP connections for improved performance when compared to REST over HTTP, and they support the AMQP 1.0 standard protocol.
- Names of Storage queues can be 3-63 characters long, can contain lowercase letters, numbers, and hyphens. For more information, see Naming Queues and Metadata.
- Service Bus queue names can be up to 260 characters long and have less restrictive naming rules. Service Bus queue names can contain letters, numbers, periods, hyphens, and underscores.

Authentication and authorization

This section discusses the authentication and authorization features supported by Storage queues and Service Bus queues.

COMPARISON CRITERIA	STORAGE QUEUES	SERVICE BUS QUEUES
Authentication	**Symmetric key**	**Symmetric key**
Security model	Delegated access via SAS tokens.	SAS
Identity provider federation	**No**	**Yes**

Additional information

- Every request to either of the queuing technologies must be authenticated. Public queues with anonymous access are not supported. Using SAS, you can address this scenario by publishing a write-only SAS, read-only SAS, or even a full-access SAS.
- The authentication scheme provided by Storage queues involves the use of a symmetric key, which is a hash-based Message Authentication Code (HMAC), computed with the SHA-256 algorithm and encoded as a **Base64** string. For more information about the respective protocol, see Authentication for the Azure Storage Services. Service Bus queues support a similar model using symmetric keys. For more information, see Shared Access Signature Authentication with Service Bus.

Conclusion

By gaining a deeper understanding of the two technologies, you will be able to make a more informed decision on which queue technology to use, and when. The decision on when to use Storage queues or Service Bus queues clearly depends on a number of factors. These factors may depend heavily on the individual needs of your application and its architecture. If your application already uses the core capabilities of Microsoft Azure, you may prefer to choose Storage queues, especially if you require basic communication and messaging between services or need queues that can be larger than 80 GB in size.

Because Service Bus queues provide a number of advanced features, such as sessions, transactions, duplicate detection, automatic dead-lettering, and durable publish/subscribe capabilities, they may be a preferred choice if you are building a hybrid application or if your application otherwise requires these features.

Next steps

The following articles provide more guidance and information about using Storage queues or Service Bus queues.

- How to Use Service Bus Queues
- How to Use the Queue Storage Service
- Best practices for performance improvements using Service Bus brokered messaging
- Introducing Queues and Topics in Azure Service Bus
- The Developer's Guide to Service Bus
- Using the Queuing Service in Azure
- Understanding Azure Storage Billing – Bandwidth, Transactions, and Capacity

Best Practices for performance improvements using Service Bus Messaging

2/22/2017 • 16 min to read • Edit on GitHub

This article describes how to use Azure Service Bus Messaging to optimize performance when exchanging brokered messages. The first part of this topic describes the different mechanisms that are offered to help increase performance. The second part provides guidance on how to use Service Bus in a way that can offer the best performance in a given scenario.

Throughout this topic, the term "client" refers to any entity that accesses 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.

These sections introduce several concepts that Service Bus uses to help boost performance.

Protocols

Service Bus enables clients to send and receive messages via one of three protocols:

1. Advanced Message Queuing Protocol (AMQP)
2. Service Bus Messaging Protocol (SBMP)
3. HTTP

AMQP and SBMP are more efficient, because they maintain the connection to Service Bus as long as the messaging factory exists. It also implements batching and prefetching. Unless explicitly mentioned, all content in this topic assumes the use of AMQP or SBMP.

Reusing factories and clients

Service Bus client objects, such as QueueClient or MessageSender, are created through a 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 you can avoid by re-using the same factory and client objects for multiple operations. You can safely use the QueueClient object for sending messages from concurrent asynchronous operations and multiple threads.

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 schedules 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);

Task send1 = queueClient.SendAsync(m1).ContinueWith((t) =>
  {
    Console.WriteLine("Sent message #1");
  });
Task send2 = queueClient.SendAsync(m2).ContinueWith((t) =>
  {
    Console.WriteLine("Sent message #2");
  });
Task.WaitAll(send1, send2);
Console.WriteLine("All messages sent");
```

This is an example of an asynchronous receive operation:

```
Task receive1 = queueClient.ReceiveAsync().ContinueWith(ProcessReceivedMessage);
Task receive2 = queueClient.ReceiveAsync().ContinueWith(ProcessReceivedMessage);

Task.WaitAll(receive1, receive2);
Console.WriteLine("All messages received");

async void ProcessReceivedMessage(Task<BrokeredMessage> t)
{
  BrokeredMessage m = t.Result;
  Console.WriteLine("{0} received", m.Label);
  await m.CompleteAsync();
  Console.WriteLine("{0} complete", 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. To obtain higher throughput rates, 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 PeekLock. When operating in this mode, the client sends a request to receive a message from Service Bus. After the client has received the message, it sends a request to complete the message.

When setting the receive mode to 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.

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 dead-letter 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 or 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.

By default, a client uses a batch interval of 20ms. You can change the batch interval by setting the

BatchFlushInterval property before creating the messaging factory. This setting affects all clients that are created by this factory.

To disable batching, set the 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, or subscription, Service Bus 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, 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 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 QueueClient.PrefetchCount or 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 256 KB, 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, and each receiver can process up to 10 messages per second. The prefetch count should not exceed 20 X 3 X 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 both synchronous and asynchronous receive operations.

Express queues and topics

Express entities enable high throughput and reduced latency scenarios. With express entities, if a message is sent to a queue or topic, the message is not immediately stored in the messaging store. Instead, it is cached in memory. If a message remains in the queue for more than a few seconds, it is automatically written to stable storage, thus protecting it against loss due to an outage. Writing the message into a memory cache increases throughput and reduces latency because there is no access to stable storage at the time the message is sent. Messages that are consumed within a few seconds are not written to the messaging store. The following example creates an express topic.

```
TopicDescription td = new TopicDescription(TopicName);
td.EnableExpress = true;
namespaceManager.CreateTopic(td);
```

If a message containing critical information that must not be lost is sent to an express entity, the sender can force Service Bus to immediately persist the message to stable storage by setting the ForcePersistence property to **true**.

> **NOTE**
>
> Please note that express entities do not support transactions.

Use of partitioned queues or topics

Internally, Service Bus uses the same node and messaging store to process and store all messages for a messaging entity (queue or topic). A partitioned queue or topic, on the other hand, is distributed across multiple nodes and messaging stores. Partitioned queues and topics not only yield a higher throughput than regular queues and topics, they also exhibit superior availability. To create a partitioned entity, set the EnablePartitioning property to **true**, as shown in the following example. For more information about partitioned entities, see Partitioned messaging entities.

```
// Create partitioned queue.
QueueDescription qd = new QueueDescription(QueueName);
qd.EnablePartitioning = true;
namespaceManager.CreateQueue(qd);
```

Use of multiple queues

If it is not possible to use a partitioned queue or topic, or the expected load cannot be handled by a single partitioned queue or topic, you must use multiple messaging entities. When using multiple entities, create a dedicated client for each entity, instead of using the same client for all entities.

Development and testing features

Service Bus has one feature that is used specifically for development which **should never be used in production configurations**: TopicDescription.EnableFilteringMessagesBeforePublishing.

When new rules or filters are added to the topic, you can use TopicDescription.EnableFilteringMessagesBeforePublishing to verify that the new filter expression is working as expected.

Scenarios

The following sections describe typical messaging scenarios and outline the preferred Service Bus settings. Throughput rates are classified as small (less than 1 message/second), moderate (1 message/second or greater but less than 100 messages/second) and high (100 messages/second or greater). The number of clients are classified as small (5 or fewer), moderate (more than 5 but less than or equal to 20), and large (more than 20).

High-throughput queue

Goal: Maximize the throughput of a single queue. The number of senders and receivers is small.

- Use a partitioned queue for improved performance and availability.
- 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.

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.

- Use a partitioned queue for improved availability.
- 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.

Service Bus enables up to 1000 concurrent connections to a messaging entity (or 5000 using AMQP). This limit is enforced at the namespace level, and queues/topics/subscriptions are capped by the limit of concurrent connections per namespace. For queues, this number is shared between senders and receivers. If all 1000 connections are required for senders, you should replace the queue with a topic and a single subscription. A topic accepts up to 1000 concurrent connections from senders, whereas the subscription accepts an additional 1000 concurrent connections from receivers. If more than 1000 concurrent senders are required, the senders should send messages to the Service Bus protocol via HTTP.

To maximize throughput, do the following:

- Use a partitioned queue for improved performance and availability.
- 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.

Service Bus enables up to 1000 concurrent connections to an entity. If a queue requires more than 1000 receivers, you should replace the queue with a topic and multiple subscriptions. Each subscription can support up to 1000 concurrent connections. Alternatively, receivers can access the queue via the HTTP protocol.

To maximize throughput, do the following:

- Use a partitioned queue for improved performance and availability.
- 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.

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:

- Use a partitioned topic for improved performance and availability.
- 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.

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. Service Bus supports up to 2,000 subscriptions per topic.

To maximize throughput, do the following:

- Use a partitioned topic for improved performance and availability.
- 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.

Next steps

To learn more about optimizing Service Bus performance, see Partitioned messaging entities.

Asynchronous messaging patterns and high availability

1/17/2017 • 10 min to read • Edit on GitHub

Asynchronous messaging can be implemented in a variety of different ways. With queues, topics, and subscriptions, Azure Service Bus supports asynchronism via a store and forward mechanism. In normal (synchronous) operation, you send messages to queues and topics, and receive messages from queues and subscriptions. Applications you write depend on these entities always being available. When the entity health changes, due to a variety of circumstances, you need a way to provide a reduced capability entity that can satisfy most needs.

Applications typically use asynchronous messaging patterns to enable a number of communication scenarios. You can build applications in which clients can send messages to services, even when the service is not running. For applications that experience bursts of communications, a queue can help level the load by providing a place to buffer communications. Finally, you can get a simple but effective load balancer to distribute messages across multiple machines.

In order to maintain availability of any of these entities, consider a number of different ways in which these entities can appear unavailable for a durable messaging system. Generally speaking, we see the entity become unavailable to applications we write in the following different ways:

- Unable to send messages.
- Unable to receive messages.
- Unable to manage entities (create, retrieve, update, or delete entities).
- Unable to contact the service.

For each of these failures, different failure modes exist that enable an application to continue to perform work at some level of reduced capability. For example, a system that can send messages but not receive them can still receive orders from customers but cannot process those orders. This topic discusses potential issues that can occur, and how those issues are mitigated. Service Bus has introduced a number of mitigations which you must opt into, and this topic also discusses the rules governing the use of those opt-in mitigations.

Reliability in Service Bus

There are several ways to handle message and entity issues, and there are guidelines governing the appropriate use of those mitigations. To understand the guidelines, you must first understand what can fail in Service Bus. Due to the design of Azure systems, all of these issues tend to be short-lived. At a high level, the different causes of unavailability appear as follows:

- Throttling from an external system on which Service Bus depends. Throttling occurs from interactions with storage and compute resources.
- Issue for a system on which Service Bus depends. For example, a given part of storage can encounter issues.
- Failure of Service Bus on single subsystem. In this situation, a compute node can get into an inconsistent state and must restart itself, causing all entities it serves to load balance to other nodes. This in turn can cause a short period of slow message processing.
- Failure of Service Bus within an Azure datacenter. This is a "catastrophic failure" during which the system is unreachable for many minutes or a few hours.

NOTE

The term **storage** can mean both Azure Storage and SQL Azure.

Service Bus contains a number of mitigations for these issues. The following sections discuss each issue and their respective mitigations.

Throttling

With Service Bus, throttling enables cooperative message rate management. Each individual Service Bus node houses many entities. Each of those entities makes demands on the system in terms of CPU, memory, storage, and other facets. When any of these facets detects usage that exceeds defined thresholds, Service Bus can deny a given request. The caller receives a ServerBusyException and retries after 10 seconds.

As a mitigation, the code must read the error and halt any retries of the message for at least 10 seconds. Since the error can happen across pieces of the customer application, it is expected that each piece independently executes the retry logic. The code can reduce the probability of being throttled by enabling partitioning on a queue or topic.

Issue for an Azure dependency

Other components within Azure can occasionally have service issues. For example, when a system that Service Bus uses is being upgraded, that system can temporarily experience reduced capabilities. To work around these types of issues, Service Bus regularly investigates and implements mitigations. Side effects of these mitigations do appear. For example, to handle transient issues with storage, Service Bus implements a system that allows message send operations to work consistently. Due to the nature of the mitigation, a sent message can take up to 15 minutes to appear in the affected queue or subscription and be ready for a receive operation. Generally speaking, most entities will not experience this issue. However, given the number of entities in Service Bus within Azure, this mitigation is sometimes needed for a small subset of Service Bus customers.

Service Bus failure on a single subsystem

With any application, circumstances can cause an internal component of Service Bus to become inconsistent. When Service Bus detects this, it collects data from the application to aid in diagnosing what happened. Once the data is collected, the application is restarted in an attempt to return it to a consistent state. This process happens fairly quickly, and results in an entity appearing to be unavailable for up to a few minutes, though typical down times are much shorter.

In these cases, the client application generates a System.TimeoutException or MessagingException exception. Service Bus contains a mitigation for this issue in the form of automated client retry logic. Once the retry period is exhausted and the message is not delivered, you can explore using other features such as paired namespaces. Paired namespaces have other caveats that are discussed in that article.

Failure of Service Bus within an Azure datacenter

The most probable reason for a failure in an Azure datacenter is a failed upgrade deployment of Service Bus or a dependent system. As the platform has matured, the likelihood of this type of failure has diminished. A datacenter failure can also happen for reasons that include the following:

- Electrical outage (power supply and generating power disappear).
- Connectivity (internet break between your clients and Azure).

In both cases, a natural or man-made disaster caused the issue. To work around this and make sure that you can still send messages, you can use paired namespaces to enable messages to be sent to a second location while the primary location is made healthy again. For more information, see Best practices for insulating applications against Service Bus outages and disasters.

Paired namespaces

The paired namespaces feature supports scenarios in which a Service Bus entity or deployment within a data center becomes unavailable. While this event occurs infrequently, distributed systems still must be prepared to handle worst case scenarios. Typically, this event happens because some element on which Service Bus depends is experiencing a short-term issue. To maintain application availability during an outage, Service Bus users can use two separate namespaces, preferably in separate data centers, to host their messaging entities. The remainder of this section uses the following terminology:

- Primary namespace: The namespace with which your application interacts, for send and receive operations.
- Secondary namespace: The namespace that acts as a backup to the primary namespace. Application logic does not interact with this namespace.
- Failover interval: The amount of time to accept normal failures before the application switches from the primary namespace to the secondary namespace.

Paired namespaces support *send availability*. Send availability preserves the ability to send messages. To use send availability, your application must meet the following requirements:

1. Messages are only received from the primary namespace.
2. Messages sent to a given queue or topic might arrive out of order.
3. If your application uses sessions, messages within a session might arrive out of order. This is a break from normal functionality of sessions. This means that your application uses sessions to logically group messages. Session state is only maintained on the primary namespace.
4. Messages within a session might arrive out of order. This is a break from normal functionality of sessions. This means that your application uses sessions to logically group messages.
5. Session state is only maintained on the primary namespace.
6. The primary queue can come online and start accepting messages before the secondary queue delivers all messages into the primary queue.

The following sections discuss the APIs, how the APIs are implemented, and shows sample code that uses the feature. Note that there are billing implications associated with this feature.

The MessagingFactory.PairNamespaceAsync API

The paired namespaces feature includes the PairNamespaceAsync method on the Microsoft.ServiceBus.Messaging.MessagingFactory class:

```
public Task PairNamespaceAsync(PairedNamespaceOptions options);
```

When the task completes, the namespace pairing is also complete and ready to act upon for any MessageReceiver, QueueClient, or TopicClient created with the MessagingFactory instance. Microsoft.ServiceBus.Messaging.PairedNamespaceOptions is the base class for the different types of pairing that are available with a MessagingFactory object. Currently, the only derived class is one named SendAvailabilityPairedNamespaceOptions, which implements the send availability requirements. SendAvailabilityPairedNamespaceOptions has a set of constructors that build on each other. Looking at the constructor with the most parameters, you can understand the behavior of the other constructors.

```
public SendAvailabilityPairedNamespaceOptions(
    NamespaceManager secondaryNamespaceManager,
    MessagingFactory messagingFactory,
    int backlogQueueCount,
    TimeSpan failoverInterval,
    bool enableSyphon)
```

These parameters have the following meanings:

- *secondaryNamespaceManager*: An initialized NamespaceManager instance for the secondary namespace that

the PairNamespaceAsync method can use to set up the secondary namespace. The namespace manager is used to obtain the list of queues in the namespace and to make sure that the required backlog queues exist. If those queues do not exist, they are created. NamespaceManager requires the ability to create a token with the **Manage** claim.
- *messagingFactory*: The MessagingFactory instance for the secondary namespace. The MessagingFactory object is used to send and, if the EnableSyphon property is set to **true**, receive messages from the backlog queues.
- *backlogQueueCount*: The number of backlog queues to create. This value must be at least 1. When sending messages to the backlog, one of these queues is randomly chosen. If you set the value to 1, then only one queue can ever be used. When this happens and the one backlog queue generates errors, the client is not able to try a different backlog queue and may fail to send your message. We recommend setting this value to some larger value and default the value to 10. You can change this to a higher or lower value depending on how much data your application sends per day. Each backlog queue can hold up to 5 GB of messages.
- *failoverInterval*: The amount of time during which you will accept failures on the primary namespace before switching any single entity over to the secondary namespace. Failovers occur on an entity-by-entity basis. Entities in a single namespace frequently live in different nodes within Service Bus. A failure in one entity does not imply a failure in another. You can set this value to System.TimeSpan.Zero to failover to the secondary immediately after your first, non-transient failure. Failures that trigger the failover timer are any MessagingException in which the IsTransient property is false, or a System.TimeoutException. Other exceptions, such as UnauthorizedAccessException do not cause failover, because they indicate that the client is configured incorrectly. A ServerBusyException does not cause failover because the correct pattern is to wait 10 seconds, then send the message again.
- *enableSyphon*: Indicates that this particular pairing should also syphon messages from the secondary namespace back to the primary namespace. In general, applications that send messages should set this value to **false**; applications that receive messages should set this value to **true**. The reason for this is that frequently, there are fewer message receivers than message senders. Depending on the number of receivers, you can choose to have a single application instance handle the syphon duties. Using many receivers has billing implications for each backlog queue.

To use the code, create a primary MessagingFactory instance, a secondary MessagingFactory instance, a secondary NamespaceManager instance, and a SendAvailabilityPairedNamespaceOptions instance. The call can be as simple as the following:

```
SendAvailabilityPairedNamespaceOptions sendAvailabilityOptions = new
SendAvailabilityPairedNamespaceOptions(secondaryNamespaceManager, secondary);
primary.PairNamespaceAsync(sendAvailabilityOptions).Wait();
```

When the task returned by the PairNamespaceAsync method completes, everything is set up and ready to use. Before the task is returned, you may not have completed all of the background work necessary for the pairing to work right. As a result, you should not start sending messages until the task returns. If any failures occurred, such as bad credentials, or failure to create the backlog queues, those exceptions will be thrown once the task completes. Once the task returns, verify that the queues were found or created by examining the BacklogQueueCount property on your SendAvailabilityPairedNamespaceOptions instance. For the preceding code, that operation appears as follows:

```
if (sendAvailabilityOptions.BacklogQueueCount < 1)
{
    // Handle case where no queues were created.
}
```

Next steps

Now that you've learned the basics of asynchronous messaging in Service Bus, you can read more details about paired namespaces.

Best practices for insulating applications against Service Bus outages and disasters

1/17/2017 • 7 min to read • Edit on GitHub

Mission-critical applications must operate continuously, even in the presence of unplanned outages or disasters. This topic describes techniques you can use to protect Service Bus applications against a potential service outage or disaster.

An outage is defined as the temporary unavailability of Azure Service Bus. The outage can affect some components of Service Bus, such as a messaging store, or even the entire datacenter. After the problem has been fixed, Service Bus becomes available again. Typically, an outage does not cause loss of messages or other data. An example of a component failure is the unavailability of a particular messaging store. An example of a datacenter-wide outage is a power failure of the datacenter, or a faulty datacenter network switch. An outage can last from a few minutes to a few days.

A disaster is defined as the permanent loss of a Service Bus scale unit or datacenter. The datacenter may or may not become available again. Typically a disaster causes loss of some or all messages or other data. Examples of disasters are fire, flooding, or earthquake.

Current architecture

Service Bus uses multiple messaging stores to store messages that are sent to queues or topics. A non-partitioned queue or topic is assigned to one messaging store. If this messaging store is unavailable, all operations on that queue or topic will fail.

All Service Bus messaging entities (queues, topics, relays) reside in a service namespace, which is affiliated with a datacenter. Service Bus does not enable automatic geo-replication of data, nor does it allow a namespace to span multiple datacenters.

Protecting against ACS outages

If you are using ACS credentials, and ACS becomes unavailable, clients can no longer obtain tokens. Clients that have a token at the time ACS goes down can continue to use Service Bus until the tokens expire. The default token lifetime is 3 hours.

To protect against ACS outages, use Shared Access Signature (SAS) tokens. In this case, the client authenticates directly with Service Bus by signing a self-minted token with a secret key. Calls to ACS are no longer required. For more information about SAS tokens, see Service Bus authentication.

Protecting queues and topics against messaging store failures

A non-partitioned queue or topic is assigned to one messaging store. If this messaging store is unavailable, all operations on that queue or topic will fail. A partitioned queue, on the other hand, consists of multiple fragments. Each fragment is stored in a different messaging store. When a message is sent to a partitioned queue or topic, Service Bus assigns the message to one of the fragments. If the corresponding messaging store is unavailable, Service Bus writes the message to a different fragment, if possible. For more information about partitioned entities, see Partitioned messaging entities.

Protecting against datacenter outages or disasters

To allow for a failover between two datacenters, you can create a Service Bus service namespace in each datacenter. For example, the Service Bus service namespace **contosoPrimary.servicebus.windows.net** might be located in the United States North/Central region, and **contosoSecondary.servicebus.windows.net** might be located in the US South/Central region. If a Service Bus messaging entity must remain accessible in the presence of a datacenter outage, you can create that entity in both namespaces.

For more information, see the "Failure of Service Bus within an Azure datacenter" section in Asynchronous messaging patterns and high availability.

Protecting relay endpoints against datacenter outages or disasters

Geo-replication of relay endpoints allows a service that exposes a relay endpoint to be reachable in the presence of Service Bus outages. To achieve geo-replication, the service must create two relay endpoints in different namespaces. The namespaces must reside in different datacenters and the two endpoints must have different names. For example, a primary endpoint can be reached under **contosoPrimary.servicebus.windows.net/myPrimaryService**, while its secondary counterpart can be reached under **contosoSecondary.servicebus.windows.net/mySecondaryService**.

The service then listens on both endpoints, and a client can invoke the service via either endpoint. A client application randomly picks one of the relays as the primary endpoint, and sends its request to the active endpoint. If the operation fails with an error code, this failure indicates that the relay endpoint is not available. The application opens a channel to the backup endpoint and reissues the request. At that point the active and the backup endpoints switch roles: the client application considers the old active endpoint to be the new backup endpoint, and the old backup endpoint to be the new active endpoint. If both send operations fail, the roles of the two entities remain unchanged and an error is returned.

The Geo-replication with Service Bus relayed messages sample demonstrates how to replicate relays.

Protecting queues and topics against datacenter outages or disasters

To achieve resilience against datacenter outages when using brokered messaging, Service Bus supports two approaches: *active* and *passive* replication. For each approach, if a given queue or topic must remain accessible in the presence of a datacenter outage, you can create it in both namespaces. Both entities can have the same name. For example, a primary queue can be reached under **contosoPrimary.servicebus.windows.net/myQueue**, while its secondary counterpart can be reached under **contosoSecondary.servicebus.windows.net/myQueue**.

If the application does not require permanent sender-to-receiver communication, the application can implement a durable client-side queue to prevent message loss and to shield the sender from any transient Service Bus errors.

Active replication

Active replication uses entities in both namespaces for every operation. Any client that sends a message sends two copies of the same message. The first copy is sent to the primary entity (for example, **contosoPrimary.servicebus.windows.net/sales**), and the second copy of the message is sent to the secondary entity (for example, **contosoSecondary.servicebus.windows.net/sales**).

A client receives messages from both queues. The receiver processes the first copy of a message, and the second copy is suppressed. To suppress duplicate messages, the sender must tag each message with a unique identifier. Both copies of the message must be tagged with the same identifier. You can use the BrokeredMessage.MessageId or BrokeredMessage.Label properties, or a custom property to tag the message. The receiver must maintain a list of messages that it has already received.

The Geo-replication with Service Bus Brokered Messages sample demonstrates active replication of messaging entities.

NOTE

The active replication approach doubles the number of operations, therefore this approach can lead to higher cost.

Passive replication

In the fault-free case, passive replication uses only one of the two messaging entities. A client sends the message to the active entity. If the operation on the active entity fails with an error code that indicates the datacenter that hosts the active entity might be unavailable, the client sends a copy of the message to the backup entity. At that point the active and the backup entities switch roles: the sending client considers the old active entity to be the new backup entity, and the old backup entity is the new active entity. If both send operations fail, the roles of the two entities remain unchanged and an error is returned.

A client receives messages from both queues. Because there is a chance that the receiver receives two copies of the same message, the receiver must suppress duplicate messages. You can suppress duplicates in the same way as described for active replication.

In general, passive replication is more economical than active replication because in most cases only one operation is performed. Latency, throughput, and monetary cost are identical to the non-replicated scenario.

When using passive replication, in the following scenarios messages can be lost or received twice:

- **Message delay or loss**: Assume that the sender successfully sent a message m1 to the primary queue, and then the queue becomes unavailable before the receiver receives m1. The sender sends a subsequent message m2 to the secondary queue. If the primary queue is temporarily unavailable, the receiver receives m1 after the queue becomes available again. In case of a disaster, the receiver may never receive m1.
- **Duplicate reception**: Assume that the sender sends a message m to the primary queue. Service Bus successfully processes m but fails to send a response. After the send operation times out, the sender sends an identical copy of m to the secondary queue. If the receiver is able to receive the first copy of m before the primary queue becomes unavailable, the receiver receives both copies of m at approximately the same time. If the receiver is not able to receive the first copy of m before the primary queue becomes unavailable, the receiver initially receives only the second copy of m, but then receives a second copy of m when the primary queue becomes available.

The Geo-replication with Service Bus brokered messages sample demonstrates passive replication of messaging entities.

Next steps

To learn more about disaster recovery, see these articles:

- Azure SQL Database Business Continuity
- Azure resiliency technical guidance

Chaining Service Bus entities with auto-forwarding

1/17/2017 • 2 min to read • Edit on GitHub

The *auto-forwarding* feature enables you to chain a queue or subscription to another queue or topic that is part of the same namespace. When auto-forwarding is enabled, Service Bus automatically removes messages that are placed in the first queue or subscription (source) and puts them in the second queue or topic (destination). Note that it is still possible to send a message to the destination entity directly. Also, it is not possible to chain a subqueue, such as a deadletter queue, to another queue or topic.

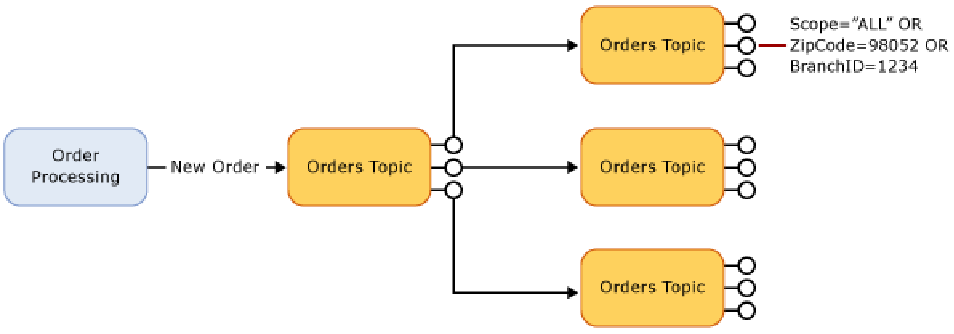
Using auto-forwarding

You can enable auto-forwarding by setting the QueueDescription.ForwardTo or SubscriptionDescription.ForwardTo properties on the QueueDescription or SubscriptionDescription objects for the source, as in the following example.

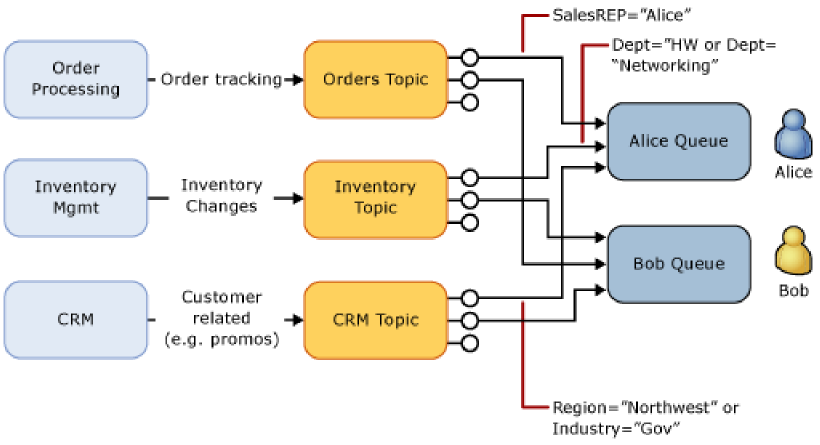
```
SubscriptionDescription srcSubscription = new SubscriptionDescription (srcTopic, srcSubscriptionName);
srcSubscription.ForwardTo = destTopic;
namespaceManager.CreateSubscription(srcSubscription));
```

The destination entity must exist at the time the source entity is created. If the destination entity does not exist, Service Bus returns an exception when asked to create the source entity.

You can use auto-forwarding to scale out an individual topic. Service Bus limits the number of subscriptions on a given topic to 2,000. You can accommodate additional subscriptions by creating second-level topics. Note that even if you are not bound by the Service Bus limitation on the number of subscriptions, adding a second level of topics can improve the overall throughput of your topic.



You can also use auto-forwarding to decouple message senders from receivers. For example, consider an ERP system that consists of three modules: Order Processing, Inventory Management, and Customer Relations Management. Each of these modules generates messages that are enqueued into a corresponding topic. Alice and Bob are sales representatives that are interested in all messages that relate to their customers. To receive those messages, Alice and Bob each create a personal queue and a subscription on each of the ERP topics that automatically forward all messages to their queue.



If Alice goes on vacation, her personal queue, rather than the ERP topic, fills up. In this scenario, because a sales representative has not received any messages, none of the ERP topics ever reach quota.

Auto-forwarding considerations

If the destination entity has accumulated many messages and exceeds the quota, or the destination entity is disabled, the source entity adds the messages to its dead-letter queue until there is space in the destination (or the entity is re-enabled). Those messages will continue to live in the dead-letter queue, so you must explicitly receive and process them from the dead-letter queue.

When chaining together individual topics to obtain a composite topic with many subscriptions, it is recommended that you have a moderate number of subscriptions on the first-level topic and many subscriptions on the second-level topics. For example, a first-level topic with 20 subscriptions, each of them chained to a second-level topic with 200 subscriptions, allows for higher throughput than a first-level topic with 200 subscriptions, each chained to a second-level topic with 20 subscriptions.

Service Bus bills one operation for each forwarded message. For example, sending a message to a topic with 20 subscriptions, each of them configured to auto-forward messages to another queue or topic, is billed as 21 operations if all first-level subscriptions receive a copy of the message.

To create a subscription that is chained to another queue or topic, the creator of the subscription must have **manage** permissions on both the source and the destination entity. Sending messages to the source topic only requires **send** permissions on the source topic.

Next steps

For detailed information about auto-forwarding, see the following reference topics:

- SubscriptionDescription.ForwardTo
- QueueDescription
- SubscriptionDescription

To learn more about Service Bus performance improvements, see Partitioned messaging entities.

Service Bus queues, topics, and subscriptions

2/27/2017 • 8 min to read • Edit on GitHub

Microsoft Azure Service Bus supports 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 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.

The messaging entities that form the core of the messaging capabilities in Service Bus are queues, topics and 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 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 grows and contracts 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. You perform management operations for Service Bus messaging entities (both queues and topics) via the Microsoft.ServiceBus.NamespaceManager class, which is constructed by supplying the base address of the Service Bus namespace and the user credentials. NamespaceManager provides methods to create, enumerate and delete messaging entities. After creating a Microsoft.ServiceBus.TokenProvider object from the SAS name and key, and a service namespace management object, you can use the Microsoft.ServiceBus.NamespaceManager.CreateQueue method to create the queue. For example:

```
// Create management credentials
TokenProvider credentials = TokenProvider.CreateSharedAccessSignatureTokenProvider(sasKeyName,sasKeyValue);
// 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 appears 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 then 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 ReceiveAndDelete mode, the receive operation is single-shot; that is, when Service Bus receives the request, it marks the message as being consumed and returns it to the application. **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. Because 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 PeekLock mode, the receive operation becomes two-stage, which makes it possible to support applications that cannot tolerate missing messages. When 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 Complete on the received message. When Service Bus sees the **Complete** call, it marks the message as being consumed.

If the application is unable to process the message for some reason, it can call the Abandon method on the received message (instead of Complete). This enables Service Bus to unlock the message and make it available to be received again, either by the same consumer or by another competing 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 unlocks the message and makes it available to be received again (essentially performing an Abandon operation by default).

Note that in the event that the application crashes after processing the message, but before the **Complete** request is issued, the message is redelivered to the application when it restarts. This is often called *At Least Once* processing; that is, each message is processed at least once. However, 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 remains 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 processed 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 identically to the way 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 maps 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 similar to creating a queue, as shown in the example in the previous section. Create the service URI, and then use the NamespaceManager class to create the namespace client. You can then create a topic using the CreateTopic method. For example:

```
TopicDescription dataCollectionTopic = namespaceClient.CreateTopic("DataCollectionTopic");
```

Next, add subscriptions as desired:

```
SubscriptionDescription myAgentSubscription = namespaceClient.CreateSubscription(myTopic.Path,
"Inventory");
SubscriptionDescription myAuditSubscription = namespaceClient.CreateSubscription(myTopic.Path,
"Dashboard");
```

You can 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 SubscriptionClient object instead of a 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>()));
}
```

Rules and actions

In many scenarios, messages that have specific characteristics must be processed in different 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 operates on the properties of the message, both the system properties (for example, **Label**) and custom application properties (for example, **StoreName**.) 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("IssueTrackingTopic", "Dashboard", new SqlFilter("StoreName =
'Store1'"));
```

With this subscription filter in place, only messages that have the `StoreName` property set to `Store1` are copied to the virtual queue for the `Dashboard` subscription.

For more information about possible filter values, see the documentation for the SqlFilter and SqlRuleAction classes. Also, see the Brokered Messaging: Advanced Filters and Topic Filters samples.

Next steps

See the following advanced topics for more information and examples of using Service Bus messaging.

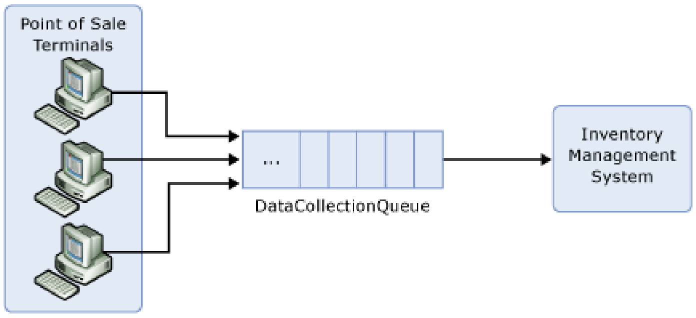
- Service Bus messaging overview
- Service Bus brokered messaging .NET tutorial
- Service Bus brokered messaging REST tutorial
- Topic Filters sample
- Brokered Messaging: Advanced Filters sample

Create applications that use Service Bus queues

2/16/2017 • 8 min to read • Edit on GitHub

This topic describes Service Bus queues and shows how to write a simple queue-based application that uses Service Bus.

Consider a scenario from the world of retail in which sales data from individual Point-of-Sale (POS) terminals must be routed to an inventory management system that uses the data to determine when stock has to be replenished. This solution uses Service Bus messaging for the communication between the terminals and the inventory management system, as illustrated in the following figure:



Each POS terminal reports its sales data by sending messages to the **DataCollectionQueue**. These messages remain in this queue until they are retrieved by the inventory management system. This pattern is often termed *asynchronous messaging*, because the POS terminal does not have to wait for a reply from the inventory management system to continue processing.

Why queuing?

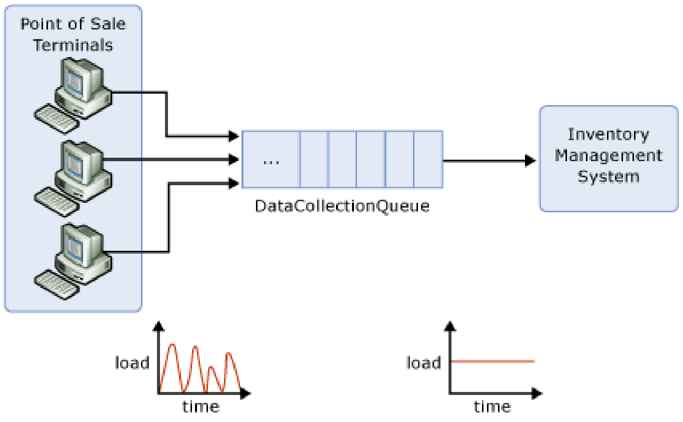
Before we look at the code that is required to set up this application, consider the advantages of using a queue in this scenario instead of having the POS terminals talk directly (synchronously) to the inventory management system.

Temporal decoupling

With the asynchronous messaging pattern, producers and consumers 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 means the components of the distributed application can be disconnected, either voluntarily; for example, for maintenance, or due to a component crash, without affecting the whole system. Furthermore, the consuming application may only have to be online during certain times of the day. For example, in this retail scenario, the inventory management system may only have to come online after the end of the business day.

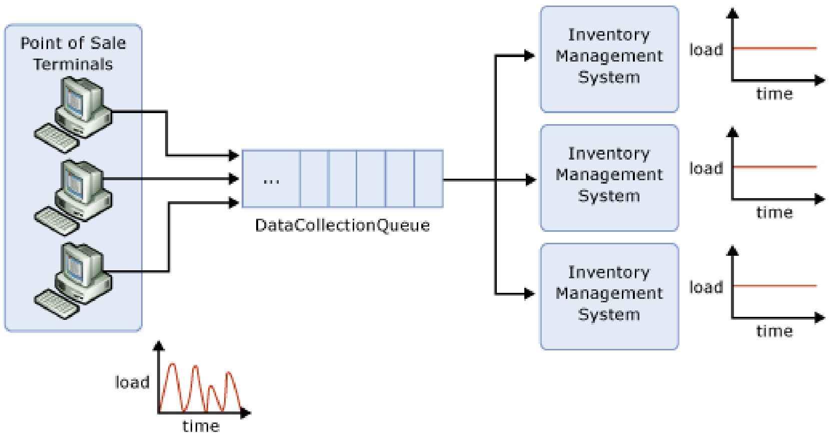
Load leveling

In many applications system load varies over time, whereas 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 (the worker) only has to be provisioned to service an average load rather than a 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.



Load balancing

As the load increases, more worker processes can be added to read from the worker queue. Each message is processed by only one of the worker processes. Furthermore, this pull-based load balancing allows for optimum usage 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.


Point of Sale Terminals
...
DataCollectionQueue
Inventory Management System
Inventory Management System
Inventory Management System
load
time
load
time
load
time
load
time

Loose coupling

Using message queuing to intermediate between message producers and consumers provides an intrinsic 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. Furthermore, the messaging topology can evolve without affecting the existing endpoints. We'll discuss this more when we talk about publish/subscribe.

Show me the code

The following section shows how to use Service Bus to build this application.

Sign up for an Azure account

You'll need an Azure account in order to start working with Service Bus. If you do not already have one, you can sign up for a free account here.

Create a namespace

Once you have a subscription, you can create a service namespace. Each namespace acts as a scoping container for a set of Service Bus entities. Give your new namespace a unique name across all Service Bus accounts.

Install the NuGet package

To use the Service Bus namespace, an application must reference the Service Bus assembly, specifically Microsoft.ServiceBus.dll. You can find this assembly as part of the Microsoft Azure SDK, and the download is available at the Azure SDK download page. However, the Service Bus NuGet package is the easiest way to get the Service Bus API and to configure your application with all of the Service Bus dependencies.

Create the queue

Management operations for Service Bus messaging entities (queues and publish/subscribe topics) are performed via the NamespaceManager class. Service Bus uses a Shared Access Signature (SAS) based security model. The TokenProvider class represents a security token provider with built-in factory methods returning some well-known token providers. We'll use a CreateSharedAccessSignatureTokenProvider method to hold the SAS credentials. The NamespaceManager instance is then constructed with the base address of the Service Bus namespace and the token provider.

The NamespaceManager class provides methods to create, enumerate and delete messaging entities. The code that is shown here shows how the NamespaceManager instance is created and used to create the **DataCollectionQueue** queue.

```
Uri uri = ServiceBusEnvironment.CreateServiceUri("sb",
                "test-blog", string.Empty);
string name = "RootManageSharedAccessKey";
string key = "abcdefghijklmopqrstuvwxyz";

TokenProvider tokenProvider =
    TokenProvider.CreateSharedAccessSignatureTokenProvider(name, key);
NamespaceManager namespaceManager =
    new NamespaceManager(uri, tokenProvider);
namespaceManager.CreateQueue("DataCollectionQueue");
```

Note that there are overloads of the CreateQueue method that enable properties of the queue to be tuned. For example, you can set the default time-to-live (TTL) value for messages sent to the queue.

Send messages to the queue

For run-time operations on Service Bus entities; for example, sending and receiving messages, an application must first create a MessagingFactory object. Similar to the NamespaceManager class, the MessagingFactory instance is created from the base address of the service namespace and the token provider.

```
 BrokeredMessage bm = new BrokeredMessage(salesData);
 bm.Label = "SalesReport";
 bm.Properties["StoreName"] = "Redmond";
 bm.Properties["MachineID"] = "POS_1";
```

Messages sent to, and received from Service Bus queues are instances of the BrokeredMessage class. This class consists of a set of standard properties (such as Label and TimeToLive), a dictionary that is used to hold application properties, and a body of arbitrary application data. An application can set the body by passing in any serializable object (the following example passes in a **SalesData** object that represents the sales data from the POS terminal), which will use the DataContractSerializer to serialize the object. Alternatively, a Stream object can be provided.

The easiest way to send messages to a given queue, in our case the **DataCollectionQueue**, is to use CreateMessageSender to create a MessageSender object directly from the MessagingFactory instance.

```
MessageSender sender = factory.CreateMessageSender("DataCollectionQueue");
sender.Send(bm);
```

Receiving messages from the queue

To receive messages from the queue, you can use a MessageReceiver object which you create directly from the MessagingFactory using CreateMessageReceiver. Message receivers can work in two different modes: **ReceiveAndDelete** and **PeekLock**. The ReceiveMode is set when the message receiver is created, as a parameter to the CreateMessageReceiver call.

When using the **ReceiveAndDelete** mode, the receive is a single-shot operation; that is, when Service Bus receives the request, it marks the message as being consumed and returns it to the application.

ReceiveAndDelete mode is the simplest model and works best for scenarios in which the application can tolerate not processing a message if a failure were to occur. To understand this, consider a scenario in which the consumer issues the receive request and then crashes before processing it. Since Service Bus marked the message as being consumed, when the application restarts and starts consuming messages again, it will have missed the message that was consumed before the crash.

In **PeekLock** mode, the receive becomes a two-stage operation, which makes it possible to support applications that cannot tolerate missing messages. When Service Bus receives the request, it finds the next message to be consumed, locks it to prevent other consumers 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 Complete on the received message. When Service Bus sees the Complete call, it marks the message as being consumed.

Two other outcomes are possible. First, if the application is unable to process the message for some reason, it can call Abandon on the received message (instead of Complete). This causes Service Bus to unlock the message and make it available to be received again, either by the same consumer or by another completing consumer. Second, there is a time-out associated with the lock and if the application cannot process the message before the lock time-out expires (for example, if the application crashes), then Service Bus will unlock the message and make it available to be received again (essentially performing an Abandon operation by default).

Note that if the application crashes after it processes the message but before the Complete request was issued, the message will be redelivered to the application when it restarts. This is often termed _At Least Once * processing. This means that 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. This can be achieved based on the MessageId property of the message. The value of this property remains constant across delivery attempts. This is termed *Exactly Once_ processing.

The code that is shown here receives and processes a message using the **PeekLock** mode, which is the default if no ReceiveMode value is explicitly provided.

```
MessageReceiver receiver = factory.CreateMessageReceiver("DataCollectionQueue");
BrokeredMessage receivedMessage = receiver.Receive();
try
{
    ProcessMessage(receivedMessage);
    receivedMessage.Complete();
}
catch (Exception e)
{
    receivedMessage.Abandon();
}
```

Use the queue client

The examples earlier in this section created MessageSender and MessageReceiver objects directly from the MessagingFactory to send and receive messages from the queue, respectively. An alternative approach is to use a QueueClient object, which supports both send and receive operations in addition to more advanced features, such as sessions.

```
QueueClient queueClient = factory.CreateQueueClient("DataCollectionQueue");
queueClient.Send(bm);

BrokeredMessage message = queueClient.Receive();

try
{
    ProcessMessage(message);
    message.Complete();
}
catch (Exception e)
{
    message.Abandon();
}
```

Next steps

Now that you've learned the basics of queues, see Create applications that use Service Bus topics and subscriptions to continue this discussion using the publish/subscribe capabilities of Service Bus topics and subscriptions.

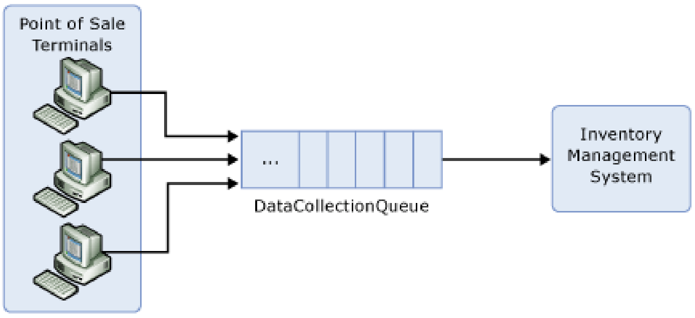
Create applications that use Service Bus topics and subscriptions

1/17/2017 • 7 min to read • Edit on GitHub

Azure Service Bus supports a set of cloud-based, message-oriented middleware technologies including reliable message queuing and durable publish/subscribe messaging. This article builds on the information provided in Create applications that use Service Bus queues and offers an introduction to the publish/subscribe capabilities offered by Service Bus topics.

Evolving retail scenario

This article continues the retail scenario used in Create applications that use Service Bus queues. Recall that sales data from individual Point of Sale (POS) terminals must be routed to an inventory management system which uses that data to determine when stock has to be replenished. Each POS terminal reports its sales data by sending messages to the **DataCollectionQueue** queue, where they remain until they are received by the inventory management system, as shown here:



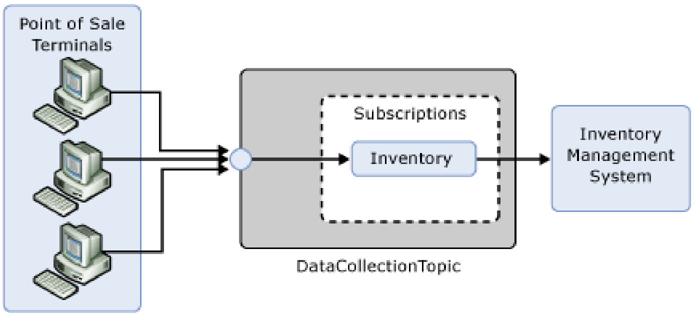
To evolve this scenario, a new requirement has been added to the system: the store owner wants to be able to monitor how the store is performing in real time.

To address this requirement, the system must "tap" off the sales data stream. We still want each message sent by the POS terminals to be sent to the inventory management system as before, but we want another copy of each message that we can use to present the dashboard view to the store owner.

In any situation such as this, in which you require each message to be consumed by multiple parties, you can use Service Bus *topics*. Topics provide a publish/subscribe pattern in which each published message is made available to one or more subscriptions registered with the topic. In contrast, with queues each message is received by a single consumer.

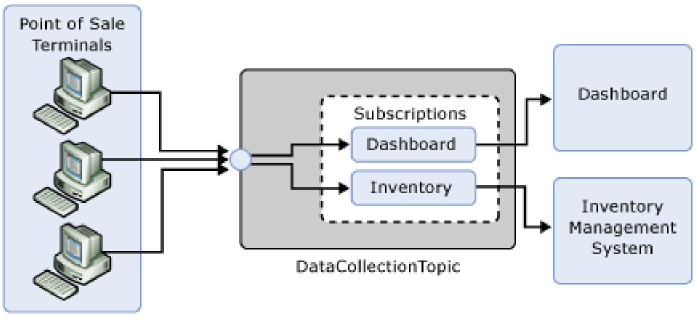
Messages are sent to a topic in the same way as they are sent to a queue. However, messages are not received from the topic directly; they are received from subscriptions. You can think of a subscription to a topic as a virtual queue that receives copies of the messages that are sent to that topic. Messages are received from a subscription the same way as they are received from a queue.

Going back to the retail scenario, the queue is replaced by a topic, and a subscription is added, which the inventory management system component can use. The system now appears as follows:



The configuration here performs identically to the previous queue-based design. That is, messages sent to the topic are routed to the **Inventory** subscription, from which the **Inventory Management System** consumes them.

In order to support the management dashboard, we create a second subscription on the topic, as shown here:



With this configuration, each message from the POS terminals is made available to both the **Dashboard** and **Inventory** subscriptions.

Show me the code

The article Create applications that use Service Bus queues describes how to sign up for an Azure account and create a service namespace. To use a Service Bus namespace, an application must reference the Service Bus assembly, specifically Microsoft.ServiceBus.dll. The easiest way to reference Service Bus dependencies is to install the Service Bus Nuget package. You can also find the assembly as part of the Azure SDK. The download is available at the Azure SDK download page.

Create the topic and subscriptions

Management operations for Service Bus messaging entities (queues and publish/subscribe topics) are performed via the NamespaceManager class. Appropriate credentials are required in order to create a **NamespaceManager** instance for a particular namespace. Service Bus uses a Shared Access Signature (SAS) based security model. The TokenProvider class represents a security token provider with built-in factory methods returning some well-known token providers. We'll use a CreateSharedAccessSignatureTokenProvider method to hold the SAS credentials. The NamespaceManager instance is then constructed with the base address of the Service Bus namespace and the token provider.

The NamespaceManager class provides methods to create, enumerate and delete messaging entities. The code that is shown here shows how the **NamespaceManager** instance is created and used to create the **DataCollectionTopic** topic.

```
Uri uri = ServiceBusEnvironment.CreateServiceUri("sb", "test-blog", string.Empty);
string name = "RootManageSharedAccessKey";
string key = "abcdefghijklmopqrstuvwxyz";

TokenProvider tokenProvider = TokenProvider.CreateSharedAccessSignatureTokenProvider(name, key);
NamespaceManager namespaceManager = new NamespaceManager(uri, tokenProvider);

namespaceManager.CreateTopic("DataCollectionTopic");
```

Note that there are overloads of the CreateTopic method that enable you to set properties of the topic. For example, you can set the default time-to-live (TTL) value for messages sent to the topic. Next, add the **Inventory** and **Dashboard** subscriptions.

```
namespaceManager.CreateSubscription("DataCollectionTopic", "Inventory");
namespaceManager.CreateSubscription("DataCollectionTopic", "Dashboard");
```

Send messages to the topic

For run-time operations on Service Bus entities; for example, sending and receiving messages, an application must first create a MessagingFactory object. Similar to the NamespaceManager class, the **MessagingFactory** instance is created from the base address of the service namespace and the token provider.

```
MessagingFactory factory = MessagingFactory.Create(uri, tokenProvider);
```

Messages sent to and received from Service Bus topics, are instances of the BrokeredMessage class. This class consists of a set of standard properties (such as Label and TimeToLive), a dictionary that is used to hold application properties, and a body of arbitrary application data. An application can set the body by passing in any serializable object (the following example passes in a **SalesData** object that represents the sales data from the POS terminal), which will use the DataContractSerializer to serialize the object. Alternatively, a Stream object can be provided.

```
BrokeredMessage bm = new BrokeredMessage(salesData);
bm.Label = "SalesReport";
bm.Properties["StoreName"] = "Redmond";
bm.Properties["MachineID"] = "POS_1";
```

The easiest way to send messages to the topic is to use CreateMessageSender to create a MessageSender object directly from the MessagingFactory instance.

```
MessageSender sender = factory.CreateMessageSender("DataCollectionTopic");
sender.Send(bm);
```

Receive messages from a subscription

Similar to using queues, to receive messages from a subscription you can use a MessageReceiver object which you create directly from the MessagingFactory using CreateMessageReceiver. You can use one of the two different receive modes (**ReceiveAndDelete** and **PeekLock**), as discussed in Create applications that use Service Bus queues.

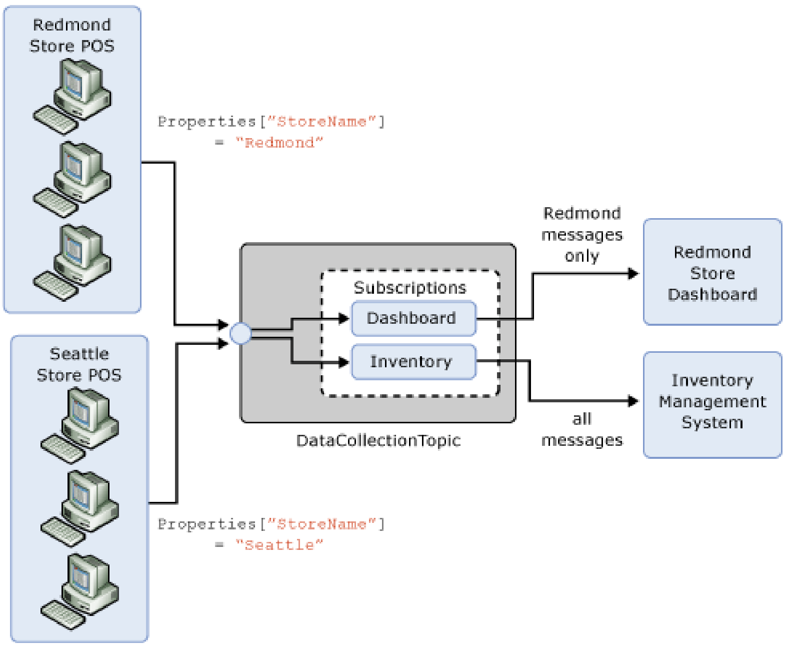
Note that when you create a **MessageReceiver** for subscriptions, the *entityPath* parameter is of the form `topicPath/subscriptions/subscriptionName`. Therefore, to create a **MessageReceiver** for the **Inventory** subscription of the **DataCollectionTopic** topic, *entityPath* must be set to `DataCollectionTopic/subscriptions/Inventory`. The code appears as follows:

```
MessageReceiver receiver = factory.CreateMessageReceiver("DataCollectionTopic/subscriptions/Inventory");
BrokeredMessage receivedMessage = receiver.Receive();
try
{
    ProcessMessage(receivedMessage);
    receivedMessage.Complete();
}
catch (Exception e)
{
    receivedMessage.Abandon();
}
```

Subscription filters

So far, in this scenario all messages sent to the topic are made available to all registered subscriptions. The key phrase here is "made available." While Service Bus subscriptions see all messages sent to the topic, you can copy only a subset of those messages to the virtual subscription queue. This is performed using subscription *filters*. When you create a subscription, you can supply a filter expression in the form of a SQL92 style predicate that operates over the properties of the message, both the system properties (for example, Label) and the application properties, such as **StoreName** in the previous example.

Evolving the scenario to illustrate this, a second store is to be added to our retail scenario. Sales data from all of the POS terminals from both stores still have to be routed to the centralized inventory management system, but a store manager using the dashboard tool is only interested in the performance of that store. You can use subscription filtering to achieve this. Note that when the POS terminals publish messages, they set the **StoreName** application property on the message. Given two stores, for example **Redmond** and **Seattle**, the POS terminals in the Redmond store stamp their sales data messages with a **StoreName** equal to **Redmond**, whereas the Seattle store POS terminals use a **StoreName** equal to **Seattle**. The store manager of the Redmond store only wants to see data from its POS terminals. The system appears as follows:



To set up this routing, you create the **Dashboard** subscription as follows:

```
SqlFilter dashboardFilter = new SqlFilter("StoreName = 'Redmond'");
namespaceManager.CreateSubscription("DataCollectionTopic", "Dashboard", dashboardFilter);
```

With this subscription filter, only messages that have the **StoreName** property set to **Redmond** will be copied to the virtual queue for the **Dashboard** subscription. There is much more to subscription filtering, however. Applications can have multiple filter rules per subscription in addition to the ability to modify the properties of a message as it passes to a subscription's virtual queue.

Summary

All of the reasons to use queuing described in Create applications that use Service Bus queues also apply to topics, specifically:

- Temporal decoupling – message producers and consumers do not have to be online at the same time.
- Load leveling – peaks in load are smoothed out by the topic enabling consuming applications to be provisioned for average load instead of peak load.

- Load balancing – similar to a queue, you can have multiple competing consumers listening on a single subscription, with each message handed off to only one of the consumers, thereby balancing load.
- Loose coupling – you can evolve the messaging network without affecting existing endpoints; for example, adding subscriptions or changing filters to a topic to allow for new consumers.

Next steps

See Create applications that use Service Bus queues for information about how to use queues in the POS retail scenario.

Service Bus authentication with Shared Access Signatures

2/15/2017 • 15 min to read • Edit on GitHub

Shared Access Signatures (SAS) are the primary security mechanism for Service Bus messaging. This article discusses SAS, how they work, and how to use them in a platform-agnostic way.

SAS authentication enables applications to authenticate to Service Bus using an access key configured on the namespace, or on the messaging entity (queue or topic) with which specific rights are associated. You can then use this key to generate a SAS token that clients can in turn use to authenticate to Service Bus.

SAS authentication support is included in the Azure SDK version 2.0 and later.

Overview of SAS

Shared Access Signatures are an authentication mechanism based on SHA-256 secure hashes or URIs. SAS is an extremely powerful mechanism that is used by all Service Bus services. In actual use, SAS has two components: a *shared access policy* and a *Shared Access Signature* (often called a *token*).

SAS authentication in Service Bus involves the configuration of a cryptographic key with associated rights on a Service Bus resource. Clients claim access to Service Bus resources by presenting a SAS token. This token consists of the resource URI being accessed, and an expiry signed with the configured key.

You can configure Shared Access Signature authorization rules on Service Bus relays, queues, and topics.

SAS authentication uses the following elements:

- Shared Access authorization rule: A 256-bit primary cryptographic key in Base64 representation, an optional secondary key, and a key name and associated rights (a collection of *Listen*, *Send*, or *Manage* rights).
- Shared Access Signature token: Generated using the HMAC-SHA256 of a resource string, consisting of the URI of the resource that is accessed and an expiry, with the cryptographic key. The signature and other elements described in the following sections are formatted into a string to form the SAS token.

Shared access policy

An important thing to understand about SAS is that it starts with a policy. For each policy, you decide on three pieces of information: **name**, **scope**, and **permissions**. The **name** is just that; a unique name within that scope. The scope is easy enough: it's the URI of the resource in question. For a Service Bus namespace, the scope is the fully qualified domain name (FQDN), such as `https://<yournamespace>.servicebus.windows.net/`.

The available permissions for a policy are largely self-explanatory:

- Send
- Listen
- Manage

After you create the policy, it is assigned a *Primary Key* and a *Secondary Key*. These are cryptographically strong keys. Don't lose them or leak them - they'll always be available in the Azure portal. You can use either of the generated keys, and you can regenerate them at any time. However, if you regenerate or change the primary key in the policy, any Shared Access Signatures created from it will be invalidated.

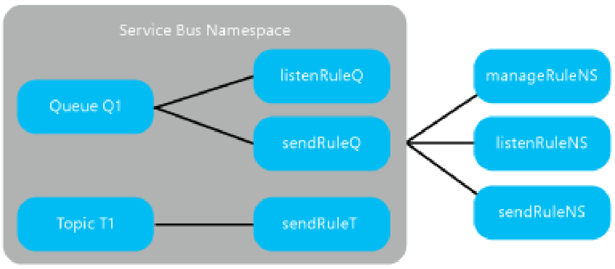
When you create a Service Bus namespace, a policy is automatically created for the entire namespace called

RootManageSharedAccessKey, and this policy has all permissions. You don't log on as **root**, so don't use this policy unless there's a really good reason. You can create additional policies in the **Configure** tab for the namespace in the portal. It's important to note that a single tree level in Service Bus (namespace, queue, etc.) can only have up to 12 policies attached to it.

Configuration for Shared Access Signature authentication

You can configure the SharedAccessAuthorizationRule rule on Service Bus namespaces, queues, or topics. Configuring a SharedAccessAuthorizationRule on a Service Bus subscription is currently not supported, but you can use rules configured on a namespace or topic to secure access to subscriptions. For a working sample that illustrates this procedure, see the Using Shared Access Signature (SAS) authentication with Service Bus Subscriptions sample.

A maximum of 12 such rules can be configured on a Service Bus namespace, queue, or topic. Rules that are configured on a Service Bus namespace apply to all entities in that namespace.



In this figure, the *manageRuleNS*, *sendRuleNS*, and *listenRuleNS* authorization rules apply to both queue Q1 and topic T1, while *listenRuleQ* and *sendRuleQ* apply only to queue Q1 and *sendRuleT* applies only to topic T1.

The key parameters of a SharedAccessAuthorizationRule are as follows:

PARAMETER	DESCRIPTION
KeyName	A string that describes the authorization rule.
PrimaryKey	A base64-encoded 256-bit primary key for signing and validating the SAS token.
SecondaryKey	A base64-encoded 256-bit secondary key for signing and validating the SAS token.
AccessRights	A list of access rights granted by the authorization rule. These rights can be any collection of Listen, Send, and Manage rights.

When a Service Bus namespace is provisioned, a SharedAccessAuthorizationRule, with KeyName set to **RootManageSharedAccessKey**, is created by default.

Generate a Shared Access Signature (token)

The policy itself is not the access token for Service Bus. It is the object from which the access token is generated - using either the primary or secondary key. Any client that has access to the signing keys specified in the shared access authorization rule can generate the SAS token. The token is generated by carefully crafting a string in the following format:

```
SharedAccessSignature sig=<signature-string>&se=<expiry>&skn=<keyName>&sr=<URL-encoded-resourceURI>
```

Where `signature-string` is the SHA-256 hash of the scope of the token (**scope** as described in the previous section) with a CRLF appended and an expiry time (in seconds since the epoch: `00:00:00 UTC` on 1 January 1970).

> **NOTE**
>
> To avoid a short token expiry time, it is recommended that you encode the expiry time value as at least a 32-bit unsigned integer, or preferably a long (64-bit) integer.

The hash looks similar to the following pseudo code and returns 32 bytes.

```
SHA-256('https://<yournamespace>.servicebus.windows.net/'+'\n'+ 1438205742)
```

The non-hashed values are in the **SharedAccessSignature** string so that the recipient can compute the hash with the same parameters, to be sure that it returns the same result. The URI specifies the scope, and the key name identifies the policy to be used to compute the hash. This is important from a security standpoint. If the signature doesn't match that which the recipient (Service Bus) calculates, then access is denied. At this point you can be sure that the sender had access to the key and should be granted the rights specified in the policy.

Note that you should use the encoded resource URI for this operation. The resource URI is the full URI of the Service Bus resource to which access is claimed. For example, `http://<namespace>.servicebus.windows.net/<entityPath>` or `sb://<namespace>.servicebus.windows.net/<entityPath>`; that is, `http://contoso.servicebus.windows.net/contosoTopics/T1/Subscriptions/S3`.

The shared access authorization rule used for signing must be configured on the entity specified by this URI, or by one of its hierarchical parents. For example, `http://contoso.servicebus.windows.net/contosoTopics/T1` or `http://contoso.servicebus.windows.net` in the previous example.

A SAS token is valid for all resources under the `<resourceURI>` used in the `signature-string`.

The KeyName in the SAS token refers to the **keyName** of the shared access authorization rule used to generate the token.

The *URL-encoded-resourceURI* must be the same as the URI used in the string-to-sign during the computation of the signature. It should be percent-encoded.

It is recommended that you periodically regenerate the keys used in the SharedAccessAuthorizationRule object. Applications should generally use the PrimaryKey to generate a SAS token. When regenerating the keys, you should replace the SecondaryKey with the old primary key, and generate a new key as the new primary key. This enables you to continue using tokens for authorization that were issued with the old primary key, and that have not yet expired.

If a key is compromised and you have to revoke the keys, you can regenerate both the PrimaryKey and the SecondaryKey of a SharedAccessAuthorizationRule, replacing them with new keys. This procedure invalidates all tokens signed with the old keys.

How to use Shared Access Signature authentication with Service Bus

The following scenarios include configuration of authorization rules, generation of SAS tokens, and client authorization.

For a full working sample of a Service Bus application that illustrates the configuration and uses SAS authorization, see Shared Access Signature authentication with Service Bus. A related sample that illustrates the use of SAS authorization rules configured on namespaces or topics to secure Service Bus subscriptions is available here: Using Shared Access Signature (SAS) authentication with Service Bus Subscriptions.

Access Shared Access Authorization rules on a namespace

Operations on the Service Bus namespace root require certificate authentication. You must upload a management certificate for your Azure subscription. To upload a management certificate, follow the steps here, using the Azure portal. For more information about Azure management certificates, see the Azure certificates overview.

The endpoint for accessing shared access authorization rules on a Service Bus namespace is as follows:

```
https://management.core.windows.net/{subscriptionId}/services/ServiceBus/namespaces/{namespace}/AuthorizationR
ules/
```

To create a SharedAccessAuthorizationRule object on a Service Bus namespace, execute a POST operation on this endpoint with the rule information serialized as JSON or XML. For example:

```
// Base address for accessing authorization rules on a namespace
string baseAddress =
@"https://management.core.windows.net/<subscriptionId>/services/ServiceBus/namespaces/<namespace>/Authorizatio
nRules/";

// Configure authorization rule with base64-encoded 256-bit key and Send rights
var sendRule = new SharedAccessAuthorizationRule("contosoSendAll",
    SharedAccessAuthorizationRule.GenerateRandomKey(),
    new[] { AccessRights.Send });

// Operations on the Service Bus namespace root require certificate authentication.
WebRequestHandler handler = new WebRequestHandler
{
    ClientCertificateOptions = ClientCertificateOption.Manual
};
// Access the management certificate by subject name
handler.ClientCertificates.Add(GetCertificate(<certificateSN>));

HttpClient httpClient = new HttpClient(handler)
{
    BaseAddress = new Uri(baseAddress)
};
httpClient.DefaultRequestHeaders.Accept.Add(
    new MediaTypeWithQualityHeaderValue("application/json"));
httpClient.DefaultRequestHeaders.Add("x-ms-version", "2015-01-01");

// Execute a POST operation on the baseAddress above to create an auth rule
var postResult = httpClient.PostAsJsonAsync("", sendRule).Result;
```

Similarly, use a GET operation on the endpoint to read the authorization rules configured on the namespace.

To update or delete a specific authorization rule, use the following endpoint:

```
https://management.core.windows.net/{subscriptionId}/services/ServiceBus/namespaces/{namespace}/AuthorizationR
ules/{KeyName}
```

Access Shared Access Authorization rules on an entity

You can access a Microsoft.ServiceBus.Messaging.SharedAccessAuthorizationRule object configured on a Service Bus queue or topic through the AuthorizationRules collection in the corresponding QueueDescription or TopicDescription.

The following code shows how to add authorization rules for a queue.

```
// Create an instance of NamespaceManager for the operation
NamespaceManager nsm = NamespaceManager.CreateFromConnectionString(
    <connectionString> );
QueueDescription qd = new QueueDescription( <qPath> );

// Create a rule with send rights with keyName as "contosoQSendKey"
// and add it to the queue description.
qd.Authorization.Add(new SharedAccessAuthorizationRule("contosoSendKey",
    SharedAccessAuthorizationRule.GenerateRandomKey(),
    new[] { AccessRights.Send }));

// Create a rule with listen rights with keyName as "contosoQListenKey"
// and add it to the queue description.
qd.Authorization.Add(new SharedAccessAuthorizationRule("contosoQListenKey",
    SharedAccessAuthorizationRule.GenerateRandomKey(),
    new[] { AccessRights.Listen }));

// Create a rule with manage rights with keyName as "contosoQManageKey"
// and add it to the queue description.
// A rule with manage rights must also have send and receive rights.
qd.Authorization.Add(new SharedAccessAuthorizationRule("contosoQManageKey",
    SharedAccessAuthorizationRule.GenerateRandomKey(),
    new[] {AccessRights.Manage, AccessRights.Listen, AccessRights.Send }));

// Create the queue.
nsm.CreateQueue(qd);
```

Use Shared Access Signature authorization

Applications using the Azure .NET SDK with the Service Bus .NET libraries can use SAS authorization through the SharedAccessSignatureTokenProvider class. The following code illustrates the use of the token provider to send messages to a Service Bus queue.

```
Uri runtimeUri = ServiceBusEnvironment.CreateServiceUri("sb",
    <yourServiceNamespace>, string.Empty);
MessagingFactory mf = MessagingFactory.Create(runtimeUri,
    TokenProvider.CreateSharedAccessSignatureTokenProvider(keyName, key));
QueueClient sendClient = mf.CreateQueueClient(qPath);

//Sending hello message to queue.
BrokeredMessage helloMessage = new BrokeredMessage("Hello, Service Bus!");
helloMessage.MessageId = "SAS-Sample-Message";
sendClient.Send(helloMessage);
```

Applications can also use SAS for authentication by using a SAS connection string in methods that accept connection strings.

Note that to use SAS authorization with Service Bus relays, you can use SAS keys configured on the Service Bus namespace. If you explicitly create a relay on the namespace (NamespaceManager with a RelayDescription) object, you can set the SAS rules just for that relay. To use SAS authorization with Service Bus subscriptions, you can use SAS keys configured on a Service Bus namespace or on a topic.

Use the Shared Access Signature (at HTTP level)

Now that you know how to create Shared Access Signatures for any entities in Service Bus, you are ready to perform an HTTP POST:

```
POST https://<yournamespace>.servicebus.windows.net/<yourentity>/messages
Content-Type: application/json
Authorization: SharedAccessSignature
sr=https%3A%2F%2F<yournamespace>.servicebus.windows.net%2F<yourentity>&sig=<yoursignature from code
above>&se=1438205742&skn=KeyName
ContentType: application/atom+xml;type=entry;charset=utf-8
```

Remember, this works for everything. You can create SAS for a queue, topic, or subscription.

If you give a sender or client a SAS token, they don't have the key directly, and they cannot reverse the hash to obtain it. As such, you have control over what they can access, and for how long. An important thing to remember is that if you change the primary key in the policy, any Shared Access Signatures created from it will be invalidated.

Use the Shared Access Signature (at AMQP level)

In the previous section, you saw how to use the SAS token with an HTTP POST request for sending data to the Service Bus. As you know, you can access Service Bus using the Advanced Message Queuing Protocol (AMQP) that is the preferred protocol to use for performance reasons, in many scenarios. The SAS token usage with AMQP is described in the document AMQP Claim-Based Security Version 1.0 that is in working draft since 2013 but well-supported by Azure today.

Before starting to send data to Service Bus, the publisher must send the SAS token inside an AMQP message to a well-defined AMQP node named **$cbs** (you can see it as a "special" queue used by the service to acquire and validate all the SAS tokens). The publisher must specify the **ReplyTo** field inside the AMQP message; this is the node in which the service replies to the publisher with the result of the token validation (a simple request/reply pattern between publisher and service). This reply node is created "on the fly," speaking about "dynamic creation of remote node" as described by the AMQP 1.0 specification. After checking that the SAS token is valid, the publisher can go forward and start to send data to the service.

The following steps show how to send the SAS token with AMQP protocol using the AMQP.Net Lite library. This is useful if you can't use the official Service Bus SDK (for example on WinRT, .Net Compact Framework, .Net Micro Framework and Mono) developing in C#. Of course, this library is useful to help understand how claims-based security works at the AMQP level, as you saw how it works at the HTTP level (with an HTTP POST request and the SAS token sent inside the "Authorization" header). If you don't need such deep knowledge about AMQP, you can use the official Service Bus SDK with .Net Framework applications, which will do it for you.

C#

```
/// <summary>
/// Send claim-based security (CBS) token
/// </summary>
/// <param name="shareAccessSignature">Shared access signature (token) to send</param>
private bool PutCbsToken(Connection connection, string sasToken)
{
    bool result = true;
    Session session = new Session(connection);

    string cbsClientAddress = "cbs-client-reply-to";
    var cbsSender = new SenderLink(session, "cbs-sender", "$cbs");
    var cbsReceiver = new ReceiverLink(session, cbsClientAddress, "$cbs");

    // construct the put-token message
    var request = new Message(sasToken);
    request.Properties = new Properties();
    request.Properties.MessageId = Guid.NewGuid().ToString();
    request.Properties.ReplyTo = cbsClientAddress;
    request.ApplicationProperties = new ApplicationProperties();
    request.ApplicationProperties["operation"] = "put-token";
    request.ApplicationProperties["type"] = "servicebus.windows.net:sastoken";
    request.ApplicationProperties["name"] = Fx.Format("amqp://{0}/{1}", sbNamespace, entity);
    cbsSender.Send(request);

    // receive the response
    var response = cbsReceiver.Receive();
    if (response == null || response.Properties == null || response.ApplicationProperties == null)
    {
        result = false;
    }
    else
    {
        int statusCode = (int)response.ApplicationProperties["status-code"];
        if (statusCode != (int)HttpStatusCode.Accepted && statusCode != (int)HttpStatusCode.OK)
        {
            result = false;
        }
    }

    // the sender/receiver may be kept open for refreshing tokens
    cbsSender.Close();
    cbsReceiver.Close();
    session.Close();

    return result;
}
```

The `PutCbsToken()` method receives the *connection* (AMQP connection class instance as provided by the AMQP .NET Lite library) that represents the TCP connection to the service and the *sasToken* parameter that is the SAS token to send.

> **NOTE**
>
> It's important that the connection is created with **SASL authentication mechanism set to EXTERNAL** (and not the default PLAIN with username and password used when you don't need to send the SAS token).

Next, the publisher creates two AMQP links for sending the SAS token and receiving the reply (the token validation result) from the service.

The AMQP message contains a set of properties, and more information than a simple message. The SAS token is the body of the message (using its constructor). The **"ReplyTo"** property is set to the node name for receiving the validation result on the receiver link (you can change its name if you want, and it will be created dynamically by the service). The last three application/custom properties are used by the service to indicate what kind of operation it has to execute. As described by the CBS draft specification, they must be the **operation name** ("put-token"), the **type of token** (in this case, a "servicebus.windows.net:sastoken"), and the **"name" of the audience** to which the token applies (the entire entity).

After sending the SAS token on the sender link, the publisher must read the reply on the receiver link. The reply is a simple AMQP message with an application property named **"status-code"** that can contain the same values as an HTTP status code.

Rights required for Service Bus operations

The following table shows the access rights required for various operations on Service Bus resources.

OPERATION	CLAIM REQUIRED	CLAIM SCOPE
Namespace		
Configure authorization rule on a namespace	Manage	Any namespace address
Service Registry		
Enumerate Private Policies	Manage	Any namespace address
Begin listening on a namespace	Listen	Any namespace address
Send messages to a listener at a namespace	Send	Any namespace address
Queue		
Create a queue	Manage	Any namespace address
Delete a queue	Manage	Any valid queue address
Enumerate queues	Manage	/$Resources/Queues
Get the queue description	Manage	Any valid queue address
Configure authorization rule for a queue	Manage	Any valid queue address
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

OPERATION	CLAIM REQUIRED	CLAIM SCOPE
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 namespace address
Delete a topic	Manage	Any valid topic address
Enumerate topics	Manage	/$Resources/Topics
Get the topic description	Manage	Any valid topic address
Configure authorization rule for a topic	Manage	Any valid topic address
Send to the topic	Send	Any valid topic address
Subscription		
Create a subscription	Manage	Any namespace address
Delete subscription	Manage	../myTopic/Subscriptions/mySubscription
Enumerate subscriptions	Manage	../myTopic/Subscriptions
Get subscription description	Manage	../myTopic/Subscriptions/mySubscription
Abandon or complete messages after receiving the message in peek-lock mode	Listen	../myTopic/Subscriptions/mySubscription
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
Rules		
Create a rule	Manage	../myTopic/Subscriptions/mySubscription

OPERATION	CLAIM REQUIRED	CLAIM SCOPE
Delete a rule	Manage	../myTopic/Subscriptions/mySubscription
Enumerate rules	Manage or Listen	../myTopic/Subscriptions/mySubscription/Rules

Next steps

To learn more about Service Bus messaging, see the following topics.

- Service Bus fundamentals
- Service Bus queues, topics, and subscriptions
- How to use Service Bus queues
- How to use Service Bus topics and subscriptions

Partitioned queues and topics

2/27/2017 • 9 min to read • Edit on GitHub

Azure Service Bus employs multiple message brokers to process messages and multiple messaging stores to store messages. A conventional queue or topic is handled by a single message broker and stored in one messaging store. Service Bus also enables queues or topics to be partitioned across multiple message brokers and messaging stores. This means that the overall throughput of a partitioned queue or topic is no longer limited by the performance of a single message broker or messaging store. In addition, a temporary outage of a messaging store does not render a partitioned queue or topic unavailable. Partitioned queues and topics can contain all advanced Service Bus features, such as support for transactions and sessions.

For more details about Service Bus internals, see the Service Bus architecture article.

How it works

Each partitioned queue or topic consists of multiple fragments. Each fragment is stored in a different messaging store and handled by a different message broker. When a message is sent to a partitioned queue or topic, Service Bus assigns the message to one of the fragments. The selection is done randomly by Service Bus or by using a partition key that the sender can specify.

When a client wants to receive a message from a partitioned queue, or from a subscription of a partitioned topic, Service Bus queries all fragments for messages, then returns the first message that is obtained from any of the messaging stores to the receiver. Service Bus caches the other messages and returns them when it receives additional receive requests. A receiving client is not aware of the partitioning; the client-facing behavior of a partitioned queue or topic (for example, read, complete, defer, deadletter, prefetching) is identical to the behavior of a regular entity.

There is no additional cost when sending a message to, or receiving a message from, a partitioned queue or topic.

Enable partitioning

To use partitioned queues and topics with Azure Service Bus, use the Azure SDK version 2.2 or later, or specify `api-version=2013-10` in your HTTP requests.

You can create Service Bus queues and topics in 1, 2, 3, 4, or 5 GB sizes (the default is 1 GB). With partitioning enabled, Service Bus creates 16 partitions for each GB you specify. As such, if you create a queue that's 5 GB in size, with 16 partitions the maximum queue size becomes (5 * 16) = 80 GB. You can see the maximum size of your partitioned queue or topic by looking at its entry on the Azure portal.

There are several ways to create a partitioned queue or topic. When you create the queue or topic from your application, you can enable partitioning for the queue or topic by respectively setting the QueueDescription.EnablePartitioning or TopicDescription.EnablePartitioning property to **true**. These properties must be set at the time the queue or topic is created. It is not possible to change these properties on an existing queue or topic. For example:

```
// Create partitioned topic
NamespaceManager ns = NamespaceManager.CreateFromConnectionString(myConnectionString);
TopicDescription td = new TopicDescription(TopicName);
td.EnablePartitioning = true;
ns.CreateTopic(td);
```

Alternatively, you can create a partitioned queue or topic in Visual Studio or in the Azure portal. When you create a new queue or topic in the portal, set the **Enable Partitioning** option in the **General settings** blade of the queue or topic **Settings** window to **true**. In Visual Studio, click the **Enable Partitioning** checkbox in the **New Queue** or **New Topic** dialog box.

Use of partition keys

When a message is enqueued into a partitioned queue or topic, Service Bus checks for the presence of a partition key. If it finds one, it selects the fragment based on that key. If it does not find a partition key, it selects the fragment based on an internal algorithm.

Using a partition key

Some scenarios, such as sessions or transactions, require messages to be stored in a specific fragment. All of these scenarios require the use of a partition key. All messages that use the same partition key are assigned to the same fragment. If the fragment is temporarily unavailable, Service Bus returns an error.

Depending on the scenario, different message properties are used as a partition key:

SessionId: If a message has the BrokeredMessage.SessionId property set, then Service Bus uses this property as the partition key. This way, all messages that belong to the same session are handled by the same message broker. This enables Service Bus to guarantee message ordering as well as the consistency of session states.

PartitionKey: If a message has the BrokeredMessage.PartitionKey property but not the BrokeredMessage.SessionId property set, then Service Bus uses the PartitionKey property as the partition key. If the message has both the SessionId and the PartitionKey properties set, both properties must be identical. If the PartitionKey property is set to a different value than the SessionId property, Service Bus returns an invalid operation exception. The PartitionKey property should be used if a sender sends non-session aware transactional messages. The partition key ensures that all messages that are sent within a transaction are handled by the same messaging broker.

MessageId: If the queue or topic has the QueueDescription.RequiresDuplicateDetection property set to **true** and the BrokeredMessage.SessionId or BrokeredMessage.PartitionKey properties are not set, then the BrokeredMessage.MessageId property serves as the partition key. (Note that the Microsoft .NET and AMQP libraries automatically assign a message ID if the sending application does not.) In this case, all copies of the same message are handled by the same message broker. This allows Service Bus to detect and eliminate duplicate messages. If the QueueDescription.RequiresDuplicateDetection property is not set to **true**, Service Bus does not consider the MessageId property as a partition key.

Not using a partition key

In the absence of a partition key, Service Bus distributes messages in a round-robin fashion to all the fragments of the partitioned queue or topic. If the chosen fragment is not available, Service Bus assigns the message to a different fragment. This way, the send operation succeeds despite the temporary unavailability of a messaging store.

To give Service Bus enough time to enqueue the message into a different fragment, the MessagingFactorySettings.OperationTimeout value specified by the client that sends the message must be greater than 15 seconds. It is recommended that you set the OperationTimeout property to the default value of 60 seconds.

Note that a partition key "pins" a message to a specific fragment. If the messaging store that holds this fragment is unavailable, Service Bus returns an error. In the absence of a partition key, Service Bus can choose a different fragment and the operation succeeds. Therefore, it is recommended that you do not supply a partition key unless it is required.

Advanced topics: use transactions with partitioned entities

Messages that are sent as part of a transaction must specify a partition key. This can be one of the following properties: BrokeredMessage.SessionId, BrokeredMessage.PartitionKey, or BrokeredMessage.MessageId. All messages that are sent as part of the same transaction must specify the same partition key. If you attempt to send a message without a partition key within a transaction, Service Bus returns an invalid operation exception. If you attempt to send multiple messages within the same transaction that have different partition keys, Service Bus returns an invalid operation exception. For example:

```
CommittableTransaction committableTransaction = new CommittableTransaction();
using (TransactionScope ts = new TransactionScope(committableTransaction))
{
    BrokeredMessage msg = new BrokeredMessage("This is a message");
    msg.PartitionKey = "myPartitionKey";
    messageSender.Send(msg);
    ts.Complete();
}
committableTransaction.Commit();
```

If any of the properties that serve as a partition key are set, Service Bus pins the message to a specific fragment. This behavior occurs whether or not a transaction is used. It is recommended that you do not specify a partition key if it is not necessary.

Using sessions with partitioned entities

To send a transactional message to a session-aware topic or queue, the message must have the BrokeredMessage.SessionId property set. If the BrokeredMessage.PartitionKey property is specified as well, it must be identical to the SessionId property. If they differ, Service Bus returns an invalid operation exception.

Unlike regular (non-partitioned) queues or topics, it is not possible to use a single transaction to send multiple messages to different sessions. If attempted, Service Bus returns an invalid operation exception. For example:

```
CommittableTransaction committableTransaction = new CommittableTransaction();
using (TransactionScope ts = new TransactionScope(committableTransaction))
{
    BrokeredMessage msg = new BrokeredMessage("This is a message");
    msg.SessionId = "mySession";
    messageSender.Send(msg);
    ts.Complete();
}
committableTransaction.Commit();
```

Automatic message forwarding with partitioned entities

Service Bus supports automatic message forwarding from, to, or between partitioned entities. To enable automatic message forwarding, set the QueueDescription.ForwardTo property on the source queue or subscription. If the message specifies a partition key (SessionId, PartitionKey, or MessageId), that partition key is used for the destination entity.

Considerations and guidelines

- **High consistency features**: If an entity uses features such as sessions, duplicate detection, or explicit control of partitioning key, then the messaging operations are always routed to specific fragments. If any of the fragments experience high traffic or the underlying store is unhealthy, those operations fail and availability is reduced. Overall, the consistency is still much higher than non-partitioned entities; only a subset of traffic is experiencing issues, as opposed to all the traffic.
- **Management**: Operations such as Create, Update and Delete must be performed on all the fragments of the

entity. If any fragment is unhealthy it could result in failures for these operations. For the Get operation, information such as message counts must be aggregated from all fragments. If any fragment is unhealthy, the entity availability status is reported as limited.

- **Low volume message scenarios**: For such scenarios, especially when using the HTTP protocol, you may have to perform multiple receive operations in order to obtain all the messages. For receive requests, the front end performs a receive on all the fragments and caches all the responses received. A subsequent receive request on the same connection would benefit from this caching and receive latencies will be lower. However, if you have multiple connections or use HTTP, that establishes a new connection for each request. As such, there is no guarantee that it would land on the same node. If all existing messages are locked and cached in another front end, the receive operation returns **null**. Messages eventually expire and you can receive them again. HTTP keep-alive is recommended.
- **Browse/Peek messages**: PeekBatch does not always return the number of messages specified in the MessageCount property. There are two common reasons for this. One reason is that the aggregated size of the collection of messages exceeds the maximum size of 256KB. Another reason is that if the queue or topic has the EnablePartitioning property set to **true**, a partition may not have enough messages to complete the requested number of messages. In general, if an application wants to receive a specific number of messages, it should call PeekBatch repeatedly until it gets that number of messages, or there are no more messages to peek. For more information, including code samples, see QueueClient.PeekBatch or SubscriptionClient.PeekBatch.

Latest added features

- Add or remove rule is now supported with partitioned entities. Different from non-partitioned entities, these operations are not supported under transactions.
- AMQP is now supported for sending and receiving messages to and from a partitioned entity.
- AMQP is now supported for the following operations: Batch Send, Batch Receive, Receive by Sequence Number, Peek, Renew Lock, Schedule Message, Cancel Scheduled Message, Add Rule, Remove Rule, Session Renew Lock, Set Session State, Get Session State, and Enumerate Sessions.

Partitioned entities limitations

Currently Service Bus imposes the following limitations on partitioned queues and topics:

- Partitioned queues and topics do not support sending messages that belong to different sessions in a single transaction.
- Service Bus currently allows up to 100 partitioned queues or topics per namespace. Each partitioned queue or topic counts towards the quota of 10,000 entities per namespace (does not apply to Premium tier).

Next steps

See the discussion of AMQP 1.0 support for Service Bus partitioned queues and topics to learn more about partitioning messaging entities.

Overview of Service Bus dead-letter queues

2/15/2017 • 3 min to read • Edit on GitHub

Service Bus queues and topic subscriptions provide a secondary sub-queue, called a *dead-letter queue* (DLQ). The dead-letter queue does not need to be explicitly created and cannot be deleted or otherwise managed independent of the main entity.

The purpose of the dead-letter queue is to hold messages that cannot be delivered to any receiver, or simply messages that could not be processed. Messages can then be removed from the DLQ and inspected. An application might, with help of an operator, correct issues and resubmit the message, log the fact that there was an error, and/or take corrective action.

From an API and protocol perspective, the DLQ is mostly similar to any other queue, except that messages can only be submitted via the dead-letter gesture of the parent entity. In addition, time-to-live is not observed, and you can't dead-letter a message from a DLQ. The dead-letter queue fully supports peek-lock delivery and transactional operations.

Note that there is no automatic cleanup of the DLQ. Messages remain in the DLQ until you explicitly retrieve them from the DLQ and call Complete() on the dead-letter message.

Moving messages to the DLQ

There are several activities in Service Bus that cause messages to get pushed to the DLQ from within the messaging engine itself. An application can also explicitly push messages to the DLQ.

As the message gets moved by the broker, two properties are added to the message as the broker calls its internal version of the DeadLetter method on the message: `DeadLetterReason` and `DeadLetterErrorDescription`.

Applications can define their own codes for the `DeadLetterReason` property, but the system sets the following values.

CONDITION	DEADLETTERREASON	DEADLETTERERRORDESCRIPTION
Always	HeaderSizeExceeded	The size quota for this stream has been exceeded.
!TopicDescription. EnableFilteringMessagesBeforePublishing and SubscriptionDescription. EnableDeadLetteringOnFilterEvaluationExceptions	exception.GetType().Name	exception.Message
EnableDeadLetteringOnMessageExpiration	TTLExpiredException	The message expired and was dead lettered.
SubscriptionDescription.RequiresSession	Session id is null.	Session enabled entity doesn't allow a message whose session identifier is null.
!dead letter queue	MaxTransferHopCountExceeded	Null
Application explicit dead lettering	Specified by application	Specified by application

Exceeding MaxDeliveryCount

Queues and subscriptions each have a QueueDescription.MaxDeliveryCount and SubscriptionDescription.MaxDeliveryCount property respectively; the default value is 10. Whenever a message has been delivered under a lock (ReceiveMode.PeekLock), but has been either explicitly abandoned or the lock has expired, the message's BrokeredMessage.DeliveryCount is incremented. When DeliveryCount exceeds MaxDeliveryCount, the message is moved to the DLQ, specifying the `MaxDeliveryCountExceeded` reason code.

This behavior cannot be disabled, but you can set MaxDeliveryCount to a very large number.

Exceeding TimeToLive

When the QueueDescription.EnableDeadLetteringOnMessageExpiration or SubscriptionDescription.EnableDeadLetteringOnMessageExpiration property is set to **true** (the default is **false**), all expiring messages are moved to the DLQ, specifying the `TTLExpiredException` reason code.

Note that expired messages are only purged and therefore moved to the DLQ when there is at least one active receiver pulling on the main queue or subscription; that behavior is by design.

Errors while processing subscription rules

When the SubscriptionDescription.EnableDeadLetteringOnFilterEvaluationExceptions property is enabled for a subscription, any errors that occur while a subscription's SQL filter rule executes are captured in the DLQ along with the offending message.

Application-level dead-lettering

In addition to the system-provided dead-lettering features, applications can use the DLQ to explicitly reject unacceptable messages. This may include messages that cannot be properly processed due to any sort of system issue, messages that hold malformed payloads, or messages that fail authentication when some message-level security scheme is used.

Dead-lettering in ForwardTo or SendVia scenarios

Messages will be sent to the transfer dead-letter queue under the following conditions:

- A message passes through more than 3 queues or topics that are chained together.
- The destination queue or topic is disabled or deleted.

To retrieve these dead-lettered messages, you can create a receiver using the FormatTransferDeadletterPath utility method.

Example

The following code snippet creates a message receiver. In the receive loop for the main queue, the code retrieves the message with Receive(TimeSpan.Zero), which asks the broker to instantly return any message readily available, or to return with no result. If the code receives a message, it immediately abandons it, which increments the `DeliveryCount`. Once the system moves the message to the DLQ, the main queue is empty and the loop exits, as ReceiveAsync returns **null**.

```
var receiver = await receiverFactory.CreateMessageReceiverAsync(queueName, ReceiveMode.PeekLock);
while(true)
{
    var msg = await receiver.ReceiveAsync(TimeSpan.Zero);
    if (msg != null)
    {
        Console.WriteLine("Picked up message; DeliveryCount {0}", msg.DeliveryCount);
        await msg.AbandonAsync();
    }
    else
    {
        break;
    }
}
```

Next steps

See the following articles for more information about Service Bus queues:

- Get started with Service Bus queues
- Azure Queues and Service Bus queues compared

Overview of Service Bus transaction processing

2/3/2017 • 3 min to read • Edit on GitHub

This article discusses the transaction capabilities of Azure Service Bus. Much of the discussion is illustrated by the Atomic Transactions with Service Bus sample. This article is limited to an overview of transaction processing and the *send via* feature in Service Bus, while the Atomic Transactions sample is broader and more complex in scope.

Transactions in Service Bus

A transaction groups two or more operations together into an *execution scope*. By nature, such a transaction must ensure that all operations belonging to a given group of operations either succeed or fail jointly. In this respect transactions act as one unit, which is often referred to as *atomicity*.

Service Bus is a transactional message broker and ensures transactional integrity for all internal operations against its message stores. All transfers of messages inside of Service Bus, such as moving messages to a dead-letter queue or automatic forwarding of messages between entities, are transactional. As such, if Service Bus accepts a message, it has already been stored and labeled with a sequence number. From then on, any message transfers within Service Bus are coordinated operations across entities, and will neither lead to loss (source succeeds and target fails) or to duplication (source fails and target succeeds) of the message.

Service Bus supports grouping operations against a single messaging entity (queue, topic, subscription) within the scope of a transaction. For example, you can send several messages to one queue from within a transaction scope, and the messages will only be committed to the queue's log when the transaction successfully completes.

Operations within a transaction scope

The operations that can be performed within a transaction scope are as follows:

- **QueueClient**, **MessageSender**, **TopicClient**: Send, SendAsync, SendBatch, SendBatchAsync
- **BrokeredMessage**: Complete, CompleteAsync, Abandon, AbandonAsync, Deadletter, DeadletterAsync, Defer, DeferAsync, RenewLock, RenewLockAsync

Receive operations are not included, because it is assumed that the application acquires messages using the ReceiveMode.PeekLock mode, inside some receive loop or with an OnMessage callback, and only then opens a transaction scope for processing the message.

The disposition of the message (complete, abandon, dead-letter, defer) then occurs within the scope of, and dependent on, the overall outcome of the transaction.

Transfers and "send via"

To enable transactional handover of data from a queue to a processor, and then to another queue, Service Bus supports *transfers*. In a transfer operation, a sender first sends a message to a "transfer queue" and the transfer queue immediately moves the message to the intended destination queue using the same robust transfer implementation that the auto-forward capability relies on. The message is never committed to the transfer queue's log in a way that it becomes visible for the transfer queue's consumers.

The power of this transactional capability becomes apparent when the transfer queue itself is the source of the sender's input messages. In other words, Service Bus can transfer the message to the destination queue "via" the transfer queue, while performing a complete (or defer, or dead-letter) operation on the input message, all in one atomic operation.

See it in code

To set up such transfers, you create a message sender that targets the destination queue via the transfer queue. You will also have a receiver that pulls messages from that same queue. For example:

```
var sender = this.messagingFactory.CreateMessageSender(destinationQueue, myQueueName);
var receiver = this.messagingFactory.CreateMessageReceiver(myQueueName);
```

A simple transaction then uses these elements, as in the following example:

```
var msg = receiver.Receive();

using (scope = new TransactionScope())
{
    // Do whatever work is required

    var newmsg = ... // package the result

    msg.Complete(); // mark the message as done
    sender.Send(newmsg); // forward the result

    scope.Complete(); // declare the transaction done
}
```

Next steps

See the following articles for more information about Service Bus queues:

- Chaining Service Bus entities with auto-forwarding
- Auto-forward sample
- Atomic Transactions with Service Bus sample
- Azure Queues and Service Bus queues compared
- How to use Service Bus queues

Service Bus diagnostic logs

3/1/2017 • 1 min to read • Edit on GitHub

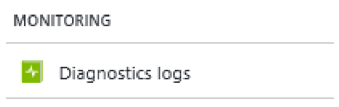
You can view two types of logs for Azure Service Bus:

- **Activity logs**. These logs have information about operations performed on a job. The logs are always turned on.
- **Diagnostic logs**. You can configure diagnostic logs, for richer insight into everything that happens with a job. Diagnostic logs cover activities from the time the job is created until the job is deleted, including updates and activities that occur while the job is running.

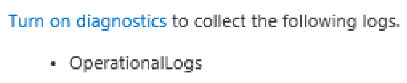
Turn on diagnostic logs

Diagnostics logs are **off** by default. To turn on diagnostic logs:

1. In the Azure portal, go to the streaming job blade.
2. Under **Monitoring**, go to the **Diagnostics logs** blade.



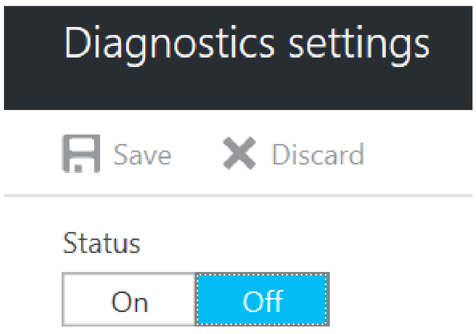
3. Select **Turn on diagnostics**.



Turn on diagnostics to collect the following logs.

- OperationalLogs

4. For **Status**, select **On**.



5. Set the archival target that you want, for example, a storage account, an event hub, or Azure Log Analytics.
6. Select the categories of logs that you want to collect, for example, **Execution** or **Authoring**.
7. Save the new diagnostics settings.

New settings take effect in about 10 minutes. After that, logs appear in the configured archival target, on the **Diagnostics logs** blade.

For more information about configuring diagnostics, see an overview of Azure diagnostic logs.

Diagnostic logs schema

All logs are stored in JavaScript Object Notation (JSON) format. Each entry has string fields that use the format described in the following example.

Operation logs example

Logs in the **OperationalLogs** category capture what is happening during Service Bus operation. Specifically, these logs capture the operation type, including queue creation, resources used, and the status of the operation.

Operation log JSON strings include elements listed in the following table:

NAME	DESCRIPTION
ActivityId	Internal ID, used for tracking
EventName	Operation name
resourceId	Azure Resource Manager resource ID
SubscriptionId	Subscription ID
EventTimeString	Operation time
EventProperties	Operation properties
Status	Operation status
Caller	Caller of operation (Azure portal or management client)
category	OperationalLogs

Here's an example of an operation log JSON string:

```
Example:
{
     "ActivityId": "6aa994ac-b56e-4292-8448-0767a5657cc7",
     "EventName": "Create Queue",
     "resourceId": "/SUBSCRIPTIONS/1A2109E3-9DA0-455B-B937-E35E36C1163C/RESOURCEGROUPS/DEFAULT-SERVICEBUS-CENTRALUS/PROVIDERS/MICROSOFT.SERVICEBUS/NAMESPACES/SHOEBOXEHNS-CY4001",
     "SubscriptionId": "1a2109e3-9da0-455b-b937-e35e36c1163c",
     "EventTimeString": "9/28/2016 8:40:06 PM +00:00",
     "EventProperties": "{\"SubscriptionId\":\"1a2109e3-9da0-455b-b937-e35e36c1163c\",\"Namespace\":\"shoeboxehns-cy4001\",\"Via\":\"https://shoeboxehns-cy4001.servicebus.windows.net/f8096791adb448579ee83d30e006a13e/?api-version=2016-07\",\"TrackingId\":\"5ee74c9e-72b5-4e98-97c4-08a62e56e221_G1\"}",
     "Status": "Succeeded",
     "Caller": "ServiceBus Client",
     "category": "OperationalLogs"
}
```

Next steps

- Introduction to Service Bus
- Get started with Service Bus

Using Service Bus from .NET with AMQP 1.0

1/20/2017 • 4 min to read • Edit on GitHub

Downloading the Service Bus SDK

AMQP 1.0 support is available in the Service Bus SDK version 2.1 or later. You can ensure you have the latest version by downloading the Service Bus bits from NuGet.

Configuring .NET applications to use AMQP 1.0

By default, the Service Bus .NET client library communicates with the Service Bus service using a dedicated SOAP-based protocol. To use AMQP 1.0 instead of the default protocol requires explicit configuration on the Service Bus connection string, as described in the next section. Other than this change, application code remains unchanged when using AMQP 1.0.

In the current release, there are a few API features that are not supported when using AMQP. These unsupported features are listed later in the section Unsupported features, restrictions, and behavioral differences. Some of the advanced configuration settings also have a different meaning when using AMQP.

Configuration using App.config

It is good practice for applications to use the App.config configuration file to store settings. For Service Bus applications, you can use App.config to store the settings for the Service Bus **ConnectionString** value. An example App.config file is as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <appSettings>
        <add key="Microsoft.ServiceBus.ConnectionString"

value="Endpoint=sb://[namespace].servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAc
cessKey=[SAS key];TransportType=Amqp" />
    </appSettings>
</configuration>
```

The value of the `Microsoft.ServiceBus.ConnectionString` setting is the Service Bus connection string that is used to configure the connection to Service Bus. The format is as follows:

```
Endpoint=sb://[namespace].servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=
[SAS key];TransportType=Amqp
```

Where `[namespace]` and `SharedAccessKey` are obtained from the Azure portal. For more information, see Get started with Service Bus queues.

When using AMQP, append the connection string with `;TransportType=Amqp`. This notation instructs the client library to make its connection to Service Bus using AMQP 1.0.

Message serialization

When using the default protocol, the default serialization behavior of the .NET client library is to use the DataContractSerializer type to serialize a BrokeredMessage instance for transport between the client library and the Service Bus service. When using the AMQP transport mode, the client library uses the AMQP type system for serialization of the brokered message into an AMQP message. This serialization enables the message to be received and interpreted by a receiving application that is potentially running on a different platform, for example, a Java application that uses the JMS API to access Service Bus.

When you construct a BrokeredMessage instance, you can provide a .NET object as a parameter to the constructor to serve as the body of the message. For objects that can be mapped to AMQP primitive types, the body is serialized into AMQP data types. If the object cannot be directly mapped into an AMQP primitive type; that is, a custom type defined by the application, then the object is serialized using the DataContractSerializer, and the serialized bytes are sent in an AMQP data message.

To facilitate interoperability with non-.NET clients, use only .NET types that can be serialized directly into AMQP types for the body of the message. The following table details those types and the corresponding mapping to the AMQP type system.

.NET BODY OBJECT TYPE	MAPPED AMQP TYPE	AMQP BODY SECTION TYPE
bool	boolean	AMQP Value
byte	ubyte	AMQP Value
ushort	ushort	AMQP Value
uint	uint	AMQP Value
ulong	ulong	AMQP Value
sbyte	byte	AMQP Value
short	short	AMQP Value
int	int	AMQP Value
long	long	AMQP Value
float	float	AMQP Value
double	double	AMQP Value
decimal	decimal128	AMQP Value
char	char	AMQP Value
DateTime	timestamp	AMQP Value
Guid	uuid	AMQP Value
byte[]	binary	AMQP Value
string	string	AMQP Value
System.Collections.IList	list	AMQP Value: items contained in the collection can only be those that are defined in this table.

.NET BODY OBJECT TYPE	MAPPED AMQP TYPE	AMQP BODY SECTION TYPE
System.Array	array	AMQP Value: items contained in the collection can only be those that are defined in this table.
System.Collections.IDictionary	map	AMQP Value: items contained in the collection can only be those that are defined in this table.Note: only String keys are supported.
Uri	Described string(see the following table)	AMQP Value
DateTimeOffset	Described long(see the following table)	AMQP Value
TimeSpan	Described long(see the following)	AMQP Value
Stream	binary	AMQP Data (may be multiple). The Data sections contain the raw bytes read from the Stream object.
Other Object	binary	AMQP Data (may be multiple). Contains the serialized binary of the object that uses the DataContractSerializer or a serializer supplied by the application.

.NET TYPE	MAPPED AMQP DESCRIBED TYPE	NOTES
Uri	`<type name=”uri” class=restricted source=”string”> <descriptor name=”com.microsoft:uri” /> </type>`	Uri.AbsoluteUri
DateTimeOffset	`<type name=”datetime-offset” class=restricted source=”long”> <descriptor name=”com.microsoft:datetime-offset” /></type>`	DateTimeOffset.UtcTicks
TimeSpan	`<type name=”timespan” class=restricted source=”long”> <descriptor name=”com.microsoft:timespan” /> </type>`	TimeSpan.Ticks

Unsupported features, restrictions, and behavioral differences

The following features of the Service Bus .NET API are not currently supported when using AMQP:

- Transactions
- Send via transfer destination

There are also some small differences in the behavior of the Service Bus .NET API when using AMQP, compared to the default protocol:

- The OperationTimeout property is ignored.
- `MessageReceiver.Receive(TimeSpan.Zero)` is implemented as `MessageReceiver.Receive(TimeSpan.FromSeconds(10))`.
- Completing messages by lock tokens can only be done by the message receivers that initially received the messages.

Controlling AMQP protocol settings

The .NET APIs expose several settings to control the behavior of the AMQP protocol:

- **MessageReceiver.PrefetchCount**: Controls the initial credit applied to a link. The default is 0.
- **MessagingFactorySettings.AmqpTransportSettings.MaxFrameSize**: Controls the maximum AMQP frame size offered during the negotiation at connection open time. The default is 65,536 bytes.
- **MessagingFactorySettings.AmqpTransportSettings.BatchFlushInterval**: If transfers are batchable, this value determines the maximum delay for sending dispositions. Inherited by senders/receivers by default. Individual sender/receiver can override the default, which is 20 milliseconds.
- **MessagingFactorySettings.AmqpTransportSettings.UseSslStreamSecurity**: Controls whether AMQP connections are established over an SSL connection. The default is **true**.

Next steps

Ready to learn more? Visit the following links:

- Service Bus AMQP overview
- AMQP 1.0 support for Service Bus partitioned queues and topics
- AMQP in Service Bus for Windows Server

Edit on GitHub

1 min to read •

AMQP 1.0 in Azure Service Bus and Event Hubs protocol guide

1/17/2017 • 23 min to read • Edit on GitHub

The Advanced Message Queueing Protocol 1.0 is a standardized framing and transfer protocol for asynchronously, securely, and reliably transferring messages between two parties. It is the primary protocol of Azure Service Bus Messaging and Azure Event Hubs. Both services also support HTTPS. The proprietary SBMP protocol that is also supported is being phased out in favor of AMQP.

AMQP 1.0 is the result of broad industry collaboration that brought together middleware vendors, such as Microsoft and Red Hat, with many messaging middleware users such as JP Morgan Chase representing the financial services industry. The technical standardization forum for the AMQP protocol and extension specifications is OASIS, and it has achieved formal approval as an international standard as ISO/IEC 19494.

Goals

This article briefly summarizes the core concepts of the AMQP 1.0 messaging specification along with a small set of draft extension specifications that are currently being finalized in the OASIS AMQP technical committee and explains how Azure Service Bus implements and builds on these specifications.

The goal is for any developer using any existing AMQP 1.0 client stack on any platform to be able to interact with Azure Service Bus via AMQP 1.0.

Common general purpose AMQP 1.0 stacks, such as Apache Proton or AMQP.NET Lite, already implement all core AMQP 1.0 gestures. Those foundational gestures are sometimes wrapped with a higher level API; Apache Proton even offers two, the imperative Messenger API and the reactive Reactor API.

In the following discussion, we will assume that the management of AMQP connections, sessions, and links and the handling of frame transfers and flow control are handled by the respective stack (such as Apache Proton-C) and do not require much if any specific attention from application developers. We will abstractly assume the existence of a few API primitives like the ability to connect, and to create some form of *sender* and *receiver* abstraction objects, which then have some shape of `send()` and `receive()` operations, respectively.

When discussing advanced capabilities of Azure Service Bus, such as message browsing or management of sessions, those will be explained in AMQP terms, but also as a layered pseudo-implementation on top of this assumed API abstraction.

What is AMQP?

AMQP is a framing and transfer protocol. Framing means that it provides structure for binary data streams that flow in either direction of a network connection. The structure provides delineation for distinct blocks of data – frames – to be exchanged between the connected parties. The transfer capabilities make sure that both communicating parties can establish a shared understanding about when frames shall be transferred, and when transfers shall be considered complete.

Unlike earlier expired draft versions produced by the AMQP working group that are still in use by a few message brokers, the working group's final, and standardized AMQP 1.0 protocol does not prescribe the presence of a message broker or any particular topology for entities inside a message broker.

The protocol can be used for symmetric peer-to-peer communication, for interaction with message brokers that support queues and publish/subscribe entities, as Azure Service Bus does. It can also be used for interaction with messaging infrastructure where the interaction patterns are different from regular queues, as is the case with Azure Event Hubs. An Event Hub acts like a queue when events are sent to it, but acts more like a serial storage service when events are read from it; it somewhat resembles a tape drive. The client picks an offset into the available data stream and is then served all events from that offset to the latest available.

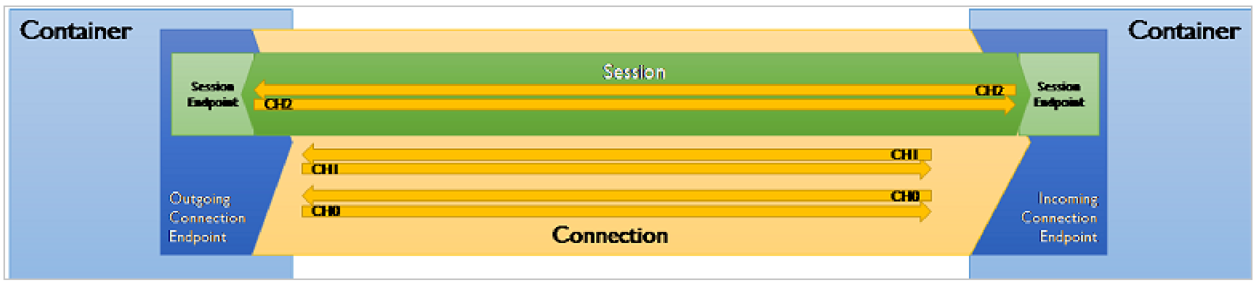
The AMQP 1.0 protocol is designed to be extensible, allowing further specifications to enhance its capabilities. The three extension specifications we will discuss in this document illustrate this. For communication over existing HTTPS/WebSockets infrastructure where configuring the native AMQP TCP ports may be difficult, a binding specification defines how to layer AMQP over WebSockets. For interacting with the messaging infrastructure in a request/response fashion for management purposes or to provide advanced functionality, the AMQP Management specification defines the required basic interaction primitives. For federated authorization model integration, the AMQP claims-based-security specification defines how to associate and renew authorization tokens associated with links.

Basic AMQP scenarios

This section explains basic usage of AMQP 1.0 with Azure Service Bus, which includes creating connections, sessions, and links, and transferring messages to and from Service Bus entities such as queues, topics, and subscriptions.

The most authoritative source to learn about how AMQP works is the AMQP 1.0 specification, but the specification was written to precisely guide implementation and not to teach the protocol. This section focuses on introducing as much terminology as needed for describing how Service Bus uses AMQP 1.0. For a more comprehensive introduction to AMQP, as well as a broader discussion of AMQP 1.0, you can review this video course.

Connections and sessions



AMQP calls the communicating programs *containers*; those contain *nodes*, which are the communicating entities inside of those containers. A queue can be such a node. AMQP allows for multiplexing, so a single connection can be used for many communication paths between nodes; for example, an application client can concurrently receive from one queue and send to another queue over the same network connection.

The network connection is thus anchored on the container. It is initiated by the container in the client role making an outbound TCP socket connection to a container in the receiver role which listens for and accepts inbound TCP connections. The connection handshake includes negotiating the protocol version, declaring or negotiating the use of Transport Level Security (TLS/SSL), and an authentication/authorization handshake at the connection scope that is based on SASL.

Azure Service Bus requires the use of TLS at all times. It supports connections over TCP port 5671, whereby the TCP connection is first overlaid with TLS before entering the AMQP protocol handshake, and also supports connections over TCP port 5672 whereby the server immediately offers a mandatory upgrade of connection to TLS using the AMQP-prescribed model. The AMQP WebSockets binding creates a tunnel over TCP port 443 that is then equivalent to AMQP 5671 connections.

After setting up the connection and TLS, Service Bus offers two SASL mechanism options:

- SASL PLAIN is commonly used for passing username and password credentials to a server. Service Bus does not have accounts, but named Shared Access Security rules, which confer rights and are associated with a key. The name of a rule is used as the user name and the key (as base64 encoded text) is used as the password. The rights associated with the chosen rule govern the operations allowed on the connection.

- SASL ANONYMOUS is used for bypassing SASL authorization when the client wants to use the claims-based-security (CBS) model that will be described later. With this option, a client connection can be established anonymously for a short time during which the client can only interact with the CBS endpoint and the CBS handshake must complete.

After the transport connection is established, the containers each declare the maximum frame size they are willing to handle, and after which idle timeout they'll unilaterally disconnect if there is no activity on the connection.

They also declare how many concurrent channels are supported. A channel is a unidirectional, outbound, virtual transfer path on top of the connection. A session takes a channel from each of the interconnected containers to form a bi-directional communication path.

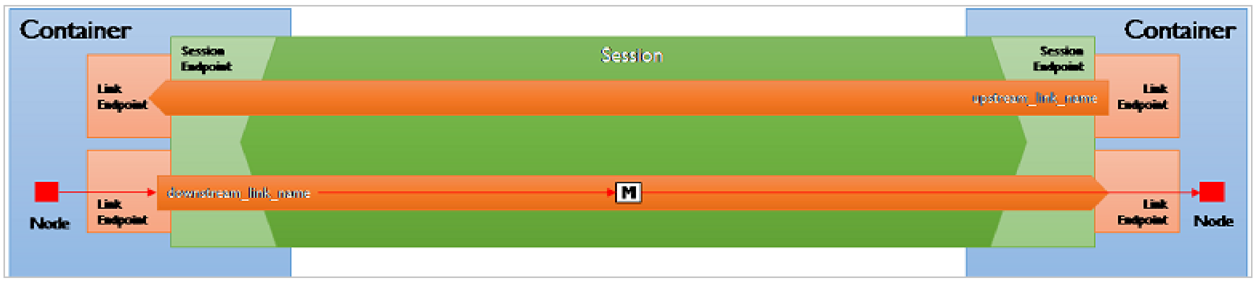
Sessions have a window-based flow control model; when a session is created, each party declares how many frames it is willing to accept into its receive window. As the parties exchange frames, transferred frames fill that window and transfers stop when the window is full and until the window gets reset or expanded using the *flow performative* (*performative* is the AMQP term for protocol-level gestures exchanged between the two parties).

This window-based model is roughly analogous to the TCP concept of window-based flow control, but at the session level inside the socket. The protocol's concept of allowing for multiple concurrent sessions exists so that high priority traffic could be rushed past throttled normal traffic, like on a highway express lane.

Azure Service Bus currently uses exactly one session for each connection. The Service Bus maximum frame-size is 262,144 bytes (256K bytes) for Service Bus Standard and Event Hubs. It is 1,048,576 (1 MB) for Service Bus Premium. Service Bus does not impose any particular session-level throttling windows, but resets the window regularly as part of link-level flow control (see the next section).

Connections, channels, and sessions are ephemeral. If the underlying connection collapses, connections, TLS tunnel, SASL authorization context, and sessions must be reestablished.

Links



AMQP transfers messages over links. A link is a communication path created over a session that enables transferring messages in one direction; the transfer status negotiation is over the link and bi-directional between the connected parties.

Links can be created by either container at any time and over an existing session, which makes AMQP different from many other protocols, including HTTP and MQTT, where the initiation of transfers and transfer path is an exclusive privilege of the party creating the socket connection.

The link-initiating container asks the opposite container to accept a link and it chooses a role of either sender or receiver. Therefore, either container can initiate creating unidirectional or bi-directional communication paths, with the latter modeled as pairs of links.

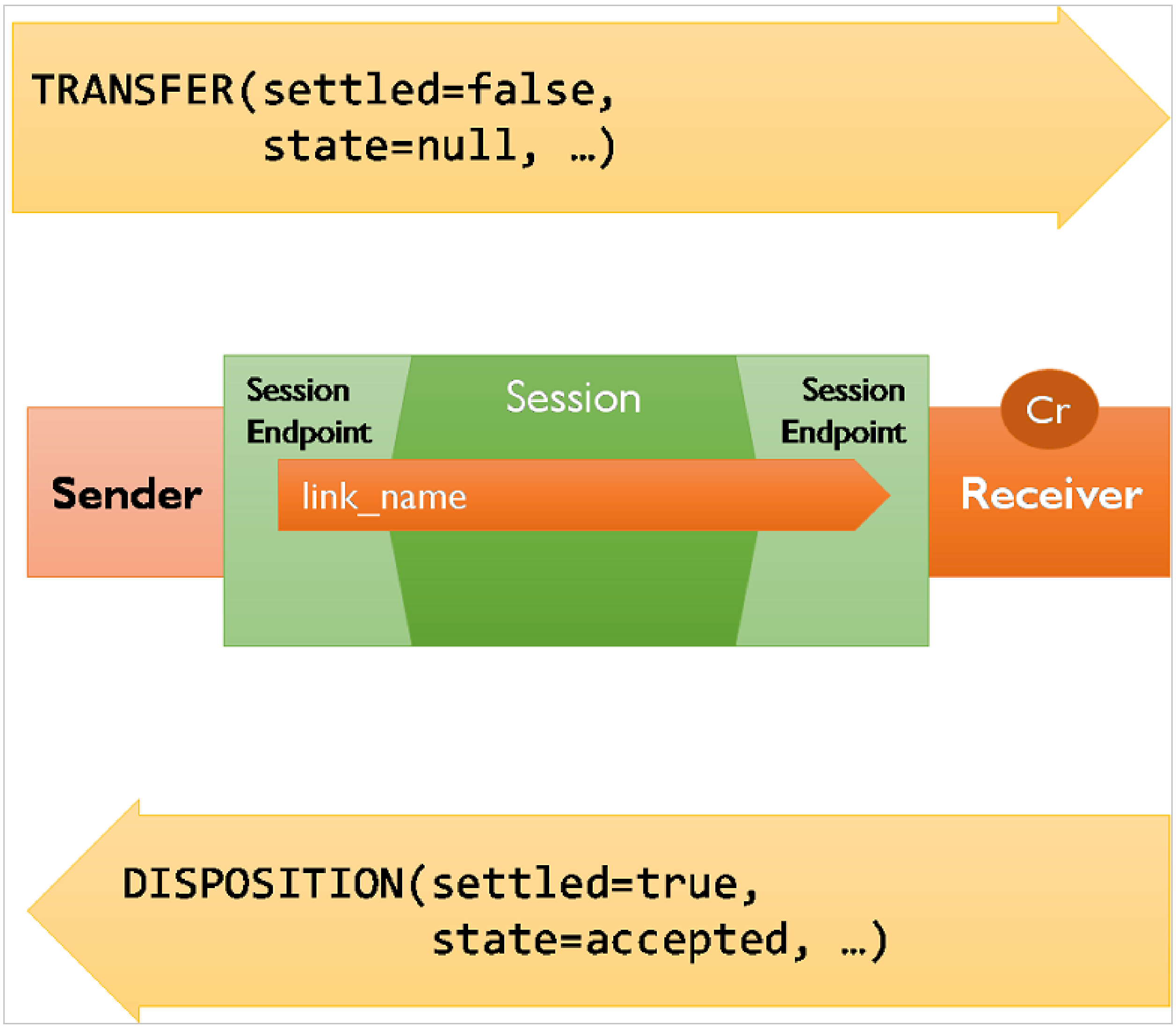
Links are named and associated with nodes. As stated in the beginning, nodes are the communicating entities inside a container.

In Azure Service Bus, a node is directly equivalent to a queue, a topic, a subscription, or a deadletter sub-queue of a queue or subscription. The node name used in AMQP is therefore the relative name of the entity inside of the

Service Bus namespace. If a queue is named **myqueue**, that's also its AMQP node name. A topic subscription follows the HTTP API convention by being sorted into a "subscriptions" resource collection and thus, a subscription **sub** or a topic **mytopic** has the AMQP node name **mytopic/subscriptions/sub**.

The connecting client is also required to use a local node name for creating links; Service Bus is not prescriptive about those node names and will not interpret them. AMQP 1.0 client stacks generally use a scheme to assure that these ephemeral node names are unique in the scope of the client.

Transfers



Once a link has been established, messages can be transferred over that link. In AMQP, a transfer is executed with an explicit protocol gesture (the *transfer* performative) that moves a message from sender to receiver over a link. A transfer is complete when it is "settled", meaning that both parties have established a shared understanding of the outcome of that transfer.

In the simplest case, the sender can choose to send messages "pre-settled," meaning that the client isn't interested in the outcome and the receiver will not provide any feedback about the outcome of the operation. This mode is supported by Azure Service Bus at the AMQP protocol level, but not exposed in any of the client APIs.

The regular case is that messages are being sent unsettled, and the receiver will then indicate acceptance or rejection using the *disposition* performative. Rejection occurs when the receiver cannot accept the message for any reason, and the rejection message contains information about the reason, which is an error structure defined by AMQP. If messages are rejected due to internal errors inside of Azure Service Bus, the service returns extra information inside that structure that can be used for providing diagnostics hints to support personnel if you are filing support requests. You'll learn more details about errors later.

A special form of rejection is the *released* state, which indicates that the receiver has no technical objection to the transfer, but also no interest in settling the transfer. That case exists, for instance, when a message is delivered to a Service Bus client, and the client chooses to "abandon" the message because it cannot perform the work resulting from processing the message while the message delivery itself is not at fault. A variation of that state is the *modified* state, which allows changes to the message as it is released. That state is not used by Service Bus at present.

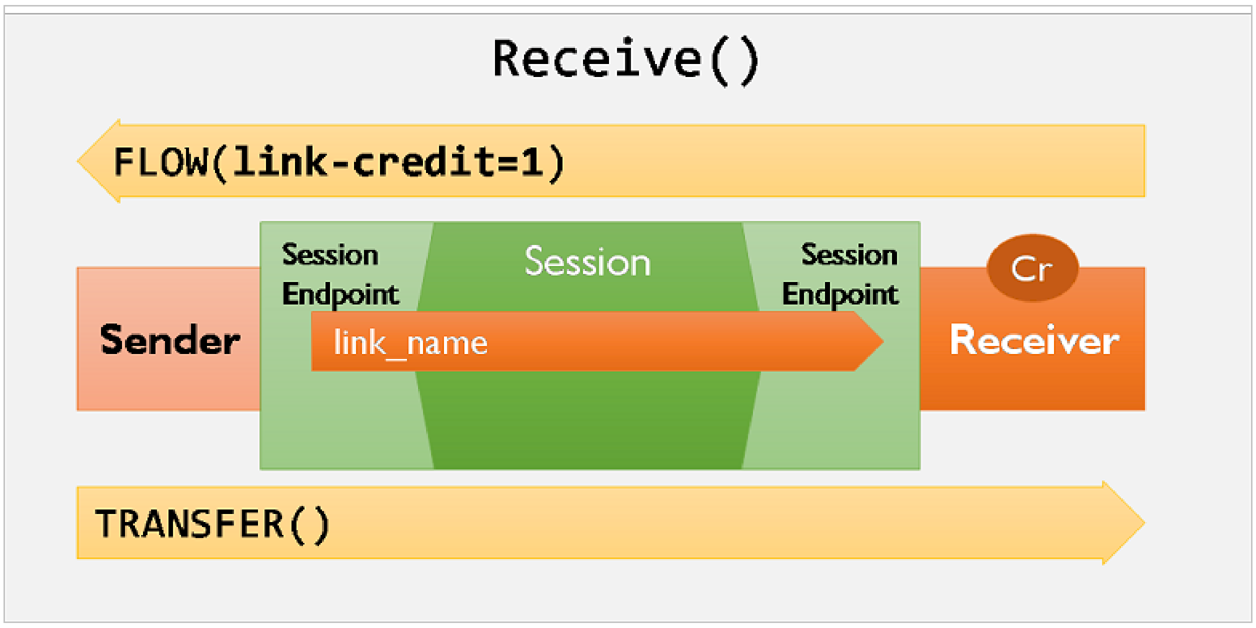
The AMQP 1.0 specification defines a further disposition state *received* that specifically helps to handle link recovery. Link recovery allows reconstituting the state of a link and any pending deliveries on top of a new connection and session, when the prior connection and session were lost.

Azure Service Bus does not support link recovery; if the client loses the connection to Service Bus with an unsettled message transfer pending, that message transfer is lost, and the client must reconnect, reestablish the link, and retry the transfer.

As such, Azure Service Bus and Event Hubs do support "at least once" transfers where the sender can be assured for the message having been stored and accepted, but it does not support "exactly once" transfers at the AMQP level, where the system would attempt to recover the link and continue to negotiate the delivery state to avoid duplication of the message transfer.

To compensate for possible duplicate sends, Azure Service Bus supports duplicate detection as an optional feature on queues and topics. Duplicate detection records the message IDs of all incoming messages during a user-defined time window, and silently drop all messages sent with the same message-IDs during that same window.

Flow control



In addition to the session-level flow control model that previously discussed, each link has its own flow control model. Session-level flow control protects the container from having to handle too many frames at once, link-level flow control puts the application in charge of how many messages it wants to handle from a link and when.

On a link, transfers can only happen when the sender has enough "link credit". Link credit is a counter set by the receiver using the *flow* performative, which is scoped to a link. When and while the sender is assigned link credit, it will attempt to use up that credit by delivering messages. Each message delivery decrements the remaining link credit by one. When the link credit is used up, deliveries stop.

When Service Bus is in the receiver role it will instantly provide the sender with ample link credit, so that messages can be sent immediately. As link credit is being used, Service Bus will occasionally send a *flow* performative to the sender to update the link credit balance.

In the sender role, Service Bus will eagerly send messages to use up any outstanding link credit.

A "receive" call at the API level translates into a *flow* performative being sent to Service Bus by the client, and

Service Bus will consume that credit by taking the first available, unlocked message from the queue, locking it and transferring it. If there is no message readily available for delivery, any outstanding credit by any link established with that particular entity will remain recorded in order of arrival and messages will be locked and transferred as they become available to use any outstanding credit.

The lock on a message is released when the transfer is settled into one of the terminal states *accepted*, *rejected*, or *released*. The message is removed from Service Bus when the terminal state is *accepted*. It remains in Service Bus and will be delivered to the next receiver when the transfer reaches any of the other states. Service Bus will automatically move the message into the entity's deadletter queue when it reaches the maximum delivery count allowed for the entity due to repeated rejections or releases.

Even though the official Service Bus APIs do not directly expose such an option today, a lower-level AMQP protocol client can use the link-credit model to turn the "pull-style" interaction of issuing one unit of credit for each receive request into a "push-style" model by issuing a very large number of link credits and then receive messages as they become available without any further interaction. Push is supported through the MessagingFactory.PrefetchCount or MessageReceiver.PrefetchCount property settings. When they are non-zero, the AMQP client uses it as the link credit.

In this context it's important to understand that the clock for the expiration of the lock on the message inside the entity starts when the message is taken from the entity and not when the message is being put on the wire. Whenever the client indicates readiness to receive messages by issuing link credit, it is therefore expected to be actively pulling messages across the network and be ready to handle them. Otherwise the message lock may have expired before the message is even delivered. The use of link-credit flow control should directly reflect the immediate readiness to deal with available messages dispatched to the receiver.

In summary, the following sections provide a schematic overview of the performative flow during different API interactions. Each section describes a different logical operation. Some of those interactions may be "lazy," meaning they may only be performed once required. Creating a message sender may not cause a network interaction until the first message is sent or requested.

The arrows show the performative flow direction.

Create Message Receiver

CLIENT	SERVICE BUS

--> attach(
name={link name},
handle={numeric handle},
role=**receiver**,
source={entity name},
target={client link id}
)
	Client attaches to entity as receiver

Service Bus replies attaching its end of the link	<-- attach(
	name={link name},
	handle={numeric handle},
	role=**sender**,
	source={entity name},
	target={client link id}
	)

Create Message Sender

CLIENT	SERVICE BUS

--> attach(
name={link name},
handle={numeric handle},
role=**sender**,
source={client link id},
target={entity name}
)
	No action

No action	<-- attach(
	name={link name},
	handle={numeric handle},
	role=**receiver**,
	source={client link id},
	target={entity name}
	)

Create Message Sender (Error)

CLIENT	SERVICE BUS

--> attach(
name={link name},
handle={numeric handle},
role=**sender**,
source={client link id},
target={entity name}
)
	No action

No action	<-- attach(
	name={link name},
	handle={numeric handle},
	role=**receiver**,
	source=null,
	target=null
	)
	
	<-- detach(
	handle={numeric handle},
	closed=**true**,
	error={error info}
	)

Close Message Receiver/Sender

CLIENT	SERVICE BUS

--> detach(
handle={numeric handle},
closed=**true**
)
	No action

No action	<-- detach(
	handle={numeric handle},
	closed=**true**
	)

Send (Success)

CLIENT	SERVICE BUS

--> transfer(
delivery-id={numeric handle},
delivery-tag={binary handle},
settled=**false**,,more=**false**,
state=**null**,
resume=**false**
)
	No action

No action	<-- disposition(
	role=receiver,
	first={delivery id},
	last={delivery id},
	settled=**true**,
	state=**accepted**
	)

Send (Error)

CLIENT	SERVICE BUS

--> transfer(
delivery-id={numeric handle},
delivery-tag={binary handle},
settled=**false**,,more=**false**,
state=**null**,
resume=**false**
)
	No action

No action	<-- disposition(
	role=receiver,
	first={delivery id},
	last={delivery id},
	settled=**true**,
	state=**rejected**(
	error={error info}
	)
	)

Receive

CLIENT	SERVICE BUS
--> flow(link-credit=1)	No action

No action	< transfer(
	delivery-id={numeric handle},
	delivery-tag={binary handle},
	settled=**false**,
	more=**false**,
	state=**null**,
	resume=**false**
	)

CLIENT	SERVICE BUS

--> disposition(
role=**receiver**,
first={delivery id},
last={delivery id},
settled=**true**,
state=**accepted**
)
	No action

Multi-Message Receive

CLIENT	SERVICE BUS
--> flow(link-credit=3)	No action

No action	< transfer(
	delivery-id={numeric handle},
	delivery-tag={binary handle},
	settled=**false**,
	more=**false**,
	state=**null**,
	resume=**false**
	)

No action	< transfer(
	delivery-id={numeric handle+1},
	delivery-tag={binary handle},
	settled=**false**,
	more=**false**,
	state=**null**,
	resume=**false**
	)

No action	< transfer(
	delivery-id={numeric handle+2},
	delivery-tag={binary handle},
	settled=**false**,
	more=**false**,
	state=**null**,
	resume=**false**
	)

--> disposition(
role=receiver,
first={delivery id},
last={delivery id+2},
settled=**true**,
state=**accepted**
)
	No action

Messages

The following sections explain which properties from the standard AMQP message sections are used by Service Bus and how they map to the official Service Bus APIs.

FIELD NAME	USAGE	API NAME
durable	-	-
priority	-	-
ttl	Time to live for this message	TimeToLive
first-acquirer	-	-
delivery-count	-	DeliveryCount

properties

FIELD NAME	USAGE	API NAME
message-id	Application-defined, free-form identifier for this message. Used for duplicate detection.	MessageId
user-id	Application-defined user identifier, not interpreted by Service Bus.	Not accessible through the Service Bus API.
to	Application-defined destination identifier, not interpreted by Service Bus.	To
subject	Application-defined message purpose identifier, not interpreted by Service Bus.	Label
reply-to	Application-defined reply-path indicator, not interpreted by Service Bus.	ReplyTo
correlation-id	Application-defined correlation identifier, not interpreted by Service Bus.	CorrelationId
content-type	Application-defined content-type indicator for the body, not interpreted by Service Bus.	ContentType
content-encoding	Application-defined content-encoding indicator for the body, not interpreted by Service Bus.	Not accessible through the Service Bus API.
absolute-expiry-time	Declares at which absolute instant the message will expire. Ignored on input (header ttl is observed), authoritative on output.	ExpiresAtUtc
creation-time	Declares at which time the message was created. Not used by Service Bus	Not accessible through the Service Bus API.

FIELD NAME	USAGE	API NAME
group-id	Application-defined identifier for a related set of messages. Used for Service Bus sessions.	SessionId
group-sequence	Counter identifying the relative sequence number of the message inside a session. Ignored by Service Bus.	Not accessible through the Service Bus API.
reply-to-group-id	-	ReplyToSessionId

Advanced Service Bus capabilities

This section covers advanced capabilities of Azure Service Bus that are based on draft extensions to AMQP currently being developed in the OASIS Technical Committee for AMQP. Azure Service Bus implements the latest status of these drafts and will adopt changes introduced as those drafts reach standard status.

> **NOTE**
> Service Bus Messaging advanced operations are supported through a request/response pattern. The details of these operations are described in the document AMQP 1.0 in Service Bus: request-response-based operations.

AMQP management

The AMQP Management specification is the first of the draft extensions we'll discuss here. This specification defines a set of protocol gestures layered on top of the AMQP protocol that allow management interactions with the messaging infrastructure over AMQP. The specification defines generic operations such as *create*, *read*, *update*, and *delete* for managing entities inside a messaging infrastructure and a set of query operations.

All those gestures require a request/response interaction between the client and the messaging infrastructure, and therefore the specification defines how to model that interaction pattern on top of AMQP: The client connects to the messaging infrastructure, initiates a session, and then creates a pair of links. On one link, the client acts as sender and on the other it acts as receiver, thus creating a pair of links that can act as a bi-directional channel.

LOGICAL OPERATION	CLIENT	SERVICE BUS

Create Request Response Path	--> attach(
	name={*link name*},
	handle={*numeric handle*},
	role=**sender**,
	source=**null**,
	target="myentity/$management"
	)
		No action

Create Request Response Path	No action	<-- attach(
		name={*link name*},
		handle={*numeric handle*},
		role=**receiver**,
		source=null,
		target="myentity"
		)

LOGICAL OPERATION	CLIENT	SERVICE BUS

Create Request Response Path	--> attach(
	name={*link name*},
	handle={*numeric handle*},
	role=**receiver**,
	source="myentity/$management",
	target="myclient$id"
	)	

Create Request Response Path	No action	<-- attach(
		name={*link name*},
		handle={*numeric handle*},
		role=**sender**,
		source="myentity",
		target="myclient$id"
		)

Having that pair of links in place, the request/response implementation is straightforward: A request is a message sent to an entity inside the messaging infrastructure that understands this pattern. In that request-message, the *reply-to* field in the *properties* section is set to the *target* identifier for the link onto which to deliver the response. The handling entity will process the request, and then deliver the reply over the link whose *target* identifier matches the indicated *reply-to* identifier.

The pattern obviously requires that the client container and the client-generated identifier for the reply destination are unique across all clients and, for security reasons, also difficult to predict.

The message exchanges used for the management protocol and for all other protocols that use the same pattern happen at the application level; they do not define new AMQP protocol-level gestures. That's intentional so that applications can take immediate advantage of these extensions with compliant AMQP 1.0 stacks.

Azure Service Bus does not currently implement any of the core features of the management specification, but the request/response pattern defined by the management specification is foundational for the claims-based-security feature and for nearly all of the advanced capabilities we will discuss in the following sections.

Claims-based authorization

The AMQP Claims-Based-Authorization (CBS) specification draft builds on the management specification's request/response pattern, and describes a generalized model for how to use federated security tokens with AMQP.

The default security model of AMQP discussed in the introduction is based on SASL and integrates with the AMQP connection handshake. Using SASL has the advantage that it provides an extensible model for which a set of mechanisms have been defined from which any protocol that formally leans on SASL can benefit. Amongst those mechanisms are "PLAIN" for transfer of usernames and passwords, "EXTERNAL" to bind to TLS-level security, "ANONYMOUS" to express the absence of explicit authentication/authorization, and a broad variety of additional mechanisms that allow passing authentication and/or authorization credentials or tokens.

AMQP's SASL integration has two drawbacks:

- All credentials and tokens are scoped to the connection. A messaging infrastructure may want to provide differentiated access control on a per-entity basis. For example, allowing the bearer of a token to send to queue A but not to queue B. With the authorization context anchored on the connection, it's not possible to use a single connection and yet use different access tokens for queue A and queue B.
- Access tokens are typically only valid for a limited time. This forces the user to periodically reacquire tokens and provides an opportunity to the token issuer to refuse issuing a fresh token if the user's access permissions have changed. AMQP connections may last for very long periods of time. The SASL model only provides a chance to set a token at connection time, which means that the messaging infrastructure either has to disconnect the client when the token expires or it needs to accept the risk of allowing continued communication with a client who's

access rights may have been revoked in the interim.

The AMQP CBS specification, implemented by Azure Service Bus, allows an elegant workaround for both of those issues: It allows a client to associate access tokens with each node, and to update those tokens before they expire, without interrupting the message flow.

CBS defines a virtual management node, named *$cbs*, to be provided by the messaging infrastructure. The management node accepts tokens on behalf of any other nodes in the messaging infrastructure.

The protocol gesture is a request/reply exchange as defined by the management specification. That means the client establishes a pair of links with the *$cbs* node and then passes a request on the outbound link, and then waits for the response on the inbound link.

The request message has the following application properties:

KEY	OPTIONAL	VALUE TYPE	VALUE CONTENTS
operation	No	string	**put-token**
type	No	string	The type of the token being put.
name	No	string	The "audience" to which the token applies.
expiration	Yes	timestamp	The expiry time of the token.

The *name* property identifies the entity with which the token shall be associated. In Service Bus it's the path to the queue, or topic/subscription. The *type* property identifies the token type:

TOKEN TYPE	TOKEN DESCRIPTION	BODY TYPE	NOTES
amqp:jwt	JSON Web Token (JWT)	AMQP Value (string)	Not yet available.
amqp:swt	Simple Web Token (SWT)	AMQP Value (string)	Only supported for SWT tokens issued by AAD/ACS
servicebus.windows.net:sastoken	Service Bus SAS Token	AMQP Value (string)	-

Tokens confer rights. Service Bus knows three fundamental rights: “Send” allows sending, “Listen” allows receiving, and “Manage” allows manipulating entities. SWT tokens issued by AAD/ACS explicitly include those rights as claims. Service Bus SAS tokens refer to rules configured on the namespace or entity, and those rules are configured with rights. Signing the token with the key associated with that rule thus makes the token express the respective rights. The token associated with an entity using *put-token* will permit the connected client to interact with the entity per the token rights. A link where the client takes on the *sender* role requires the “Send” right, taking on the *receiver* role requires the “Listen” right.

The reply message has the following *application-properties* values

KEY	OPTIONAL	VALUE TYPE	VALUE CONTENTS
status-code	No	int	HTTP response code **[RFC2616]**.

KEY	OPTIONAL	VALUE TYPE	VALUE CONTENTS
status-description	Yes	string	Description of the status.

The client can call *put-token* repeatedly and for any entity in the messaging infrastructure. The tokens are scoped to the current client and anchored on the current connection, meaning the server will drop any retained tokens when the connection drops.

The current Service Bus implementation only allows CBS in conjunction with the SASL method "ANONYMOUS". A SSL/TLS connection must always exist prior to the SASL handshake.

The ANONYMOUS mechanism must therefore be supported by the chosen AMQP 1.0 client. Anonymous access means that the initial connection handshake, including creating of the initial session happens without Service Bus knowing who is creating the connection.

Once the connection and session is established, attaching the links to the *$cbs* node and sending the *put-token* request are the only permitted operations. A valid token must be set successfully using a *put-token* request for some entity node within 20 seconds after the connection has been established, otherwise the connection is unilaterally dropped by Service Bus.

The client is subsequently responsible for keeping track of token expiration. When a token expires, Service Bus will promptly drop all links on the connection to the respective entity. To prevent this, the client can replace the token for the node with a new one at any time through the virtual *$cbs* management node with the same *put-token* gesture, and without getting in the way of the payload traffic that flows on different links.

Next steps

To learn more about AMQP, see the following links:

- Service Bus AMQP overview
- AMQP 1.0 support for Service Bus partitioned queues and topics
- AMQP in Service Bus for Windows Server

AMQP 1.0 in Microsoft Azure Service Bus: request-response-based operations

1/17/2017 • 12 min to read • Edit on GitHub

This topic defines the list of Microsoft Azure Service Bus request/response-based operations. This information is based on the AMQP Management Version 1.0 working draft.

For a detailed wire-level AMQP 1.0 protocol guide, which explains how Service Bus implements and builds on the OASIS AMQP technical specification, see the AMQP 1.0 in Azure Service Bus and Event Hubs protocol guide.

Concepts

Entity description

An entity description refers to either a Service Bus QueueDescription Class, TopicDescription Class, or SubscriptionDescription Class object.

Brokered message

Represents a message in Service Bus which is mapped to an AMQP message. The mapping is defined in the Service Bus AMQP protocol guide article.

Attach to entity management node

All the operations described in this document follow a request/response pattern, are scoped to an entity, and require attaching to an entity management node.

Create link for sending requests

Creates a link to the management node for sending requests.

```
requestLink = session.attach(
role: SENDER,
       target: { address: "<entity address>/$management" },
       source: { address: ""<my request link unique address>" }
)
```

Create link for receiving responses

Creates a link for receiving responses from the management node.

```
responseLink = session.attach(
role: RECEIVER,
    source: { address: "<entity address>/$management" }
       target: { address: "<my response link unique address>" }
)
```

Transfer a request message

Transfers a request message.

```
requestLink.sendTransfer(
        Message(
                properties: {
                        message-id: <request id>,
                        reply-to: "<my response link unique address>"
                },
                application-properties: {
                        "operation" -> "<operation>",
                },
        )
```

Receive a response message

Receives the response message from the response link.

```
responseMessage = responseLink.receiveTransfer()
```

The response message will be of the following form.

```
Message(
properties: {
        correlation-id: <request id>
    },
    application-properties: {
            "statusCode" -> <status code>,
            "statusDescription" -> <status description>,
           },
)
```

Service Bus entity address

Service Bus entities must be addressed as follows:

ENTITY TYPE	ADDRESS	EXAMPLE

queue	`<queue_name>`	`“myQueue”`
		
		`“site1/myQueue”`

topic	`<topic_name>`	`“myTopic”`
		
		`“site2/page1/myQueue”`

subscription	`<topic_name>/Subscriptions/<subscriptio`	`“myTopic/Subscriptions/MySub”`

Message operations

Message Renew Lock

Extends the lock of a message by the time specified in the entity description.

Request

The request message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
operation	string	Yes	`com.microsoft:renew-lock`

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
`com.microsoft:server-timeout`	uint	No	Operation server timeout in milliseconds.

The request message body must consist of an amqp-value section containing a map with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
`lock-tokens`	array of uuid	Yes	Message lock tokens to renew.

Response

The response message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
statusCode	int	Yes	HTTP response code [RFC2616] 200: OK – success, otherwise failed.
statusDescription	string	No	Description of the status.

The response message body must consist of an amqp-value section containing a map with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
expirations	array of timestamp	Yes	Message lock token new expiration corresponding to the request lock tokens.

Peek Message

Peeks messages without locking.

Request

The request message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
operation	string	Yes	`com.microsoft:peek-message`
`com.microsoft:server-timeout`	uint	No	Operation server timeout in milliseconds.

The request message body must consist of an **amqp-value** section containing a **map** with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
`from-sequence-number`	long	Yes	Sequence number from which to start peek.
`message-count`	int	Yes	Maximum number of messages to peek.

Response

The response message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
statusCode	int	Yes	HTTP response code [RFC2616] 200: OK – has more messages 0xcc: No content – no more messages
statusDescription	string	No	Description of the status.

The response message body must consist of an **amqp-value** section containing a **map** with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
messages	list of maps	Yes	List of messages in which every map represents a message.

The map representing a message must contain the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
message	array of byte	Yes	AMQP 1.0 wire-encoded message.

Schedule Message

Schedules messages.

Request

The request message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
operation	string	Yes	`com.microsoft:schedule-message`
`com.microsoft:server-timeout`	uint	No	Operation server timeout in milliseconds.

The request message body must consist of an **amqp-value** section containing a **map** with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
messages	list of maps	Yes	List of messages in which every map represents a message.

The map representing a message must contain the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
message-id	string	Yes	`amqpMessage.Properties.MessageId` as string
session-id	string	Yes	`amqpMessage.Properties.GroupId as string`
partition-key	string	Yes	`amqpMessage.MessageAnnotations.”x opt-partition-key"`
message	array of byte	Yes	AMQP 1.0 wire-encoded message.

Response

The response message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
statusCode	int	Yes	HTTP response code [RFC2616] 200: OK – success, otherwise failed.
statusDescription	string	No	Description of the status.

The response message body must consist of an **amqp-value** section containing a map with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
sequence-numbers	array of long	Yes	Sequence number of scheduled messages. Sequence number is used to cancel.

Cancel Scheduled Message

Cancels scheduled messages.

Request

The request message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
operation	string	Yes	`com.microsoft:cancel-scheduled-message`
`com.microsoft:server-timeout`	uint	No	Operation server timeout in milliseconds.

The request message body must consist of an **amqp-value** section containing a **map** with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
sequence-numbers	array of long	Yes	Sequence numbers of scheduled messages to cancel.

Response

The response message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
statusCode	int	Yes	HTTP response code [RFC2616] 200: OK – success, otherwise failed.
statusDescription	string	No	Description of the status.

The response message body must consist of an **amqp-value** section containing a map with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
sequence-numbers	array of long	Yes	Sequence number of scheduled messages. Sequence number is used to cancel.

Session Operations

Session Renew Lock

Extends the lock of a message by the time specified in the entity description.

Request

The request message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
operation	string	Yes	`com.microsoft:renew-session-lock`
`com.microsoft:server-timeout`	uint	No	Operation server timeout in milliseconds.

The request message body must consist of an **amqp-value** section containing a **map** with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
session-id	string	Yes	Session ID.

Response

The response message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
statusCode	int	Yes	HTTP response code [RFC2616] 200: OK – has more messages 0xcc: No content – no more messages

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
statusDescription	string	No	Description of the status.

The response message body must consist of an **amqp-value** section containing a map with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
expiration	timestamp	Yes	New expiration.

Peek Session Message

Peeks session messages without locking.

Request

The request message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
operation	string	Yes	`com.microsoft:peek-message`
`com.microsoft:server-timeout`	uint	No	Operation server timeout in milliseconds.

The request message body must consist of an **amqp-value** section containing a **map** with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
from-sequence-number	long	Yes	Sequence number from which to start peek.
message-count	int	Yes	Maximum number of messages to peek.
session-id	string	Yes	Session ID.

Response

The response message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
statusCode	int	Yes	HTTP response code [RFC2616] 200: OK – has more messages 0xcc: No content – no more messages
statusDescription	string	No	Description of the status.

The response message body must consist of an **amqp-value** section containing a map with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
messages	list of maps	Yes	List of messages in which every map represents a message.

The map representing a message must contain the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
message	array of byte	Yes	AMQP 1.0 wire-encoded message.

Set Session State

Sets the state of a session.

Request

The request message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
operation	string	Yes	`com.microsoft:peek-message`
`com.microsoft:server-timeout`	uint	No	Operation server timeout in milliseconds.

The request message body must consist of an **amqp-value** section containing a **map** with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
session-id	string	Yes	Session ID.
session-state	array of bytes	Yes	Opaque binary data.

Response

The response message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
statusCode	int	Yes	HTTP response code [RFC2616] 200: OK – success, otherwise failed
statusDescription	string	No	Description of the status.

Get Session State

Gets the state of a session.

Request

The request message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
operation	string	Yes	`com.microsoft:get-session-state`
`com.microsoft:server-timeout`	uint	No	Operation server timeout in milliseconds.

The request message body must consist of an **amqp-value** section containing a **map** with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
session-id	string	Yes	Session ID.

Response

The response message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
statusCode	int	Yes	HTTP response code [RFC2616] 200: OK – success, otherwise failed
statusDescription	string	No	Description of the status.

The response message body must consist of an **amqp-value** section containing a **map** with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
session-state	array of bytes	Yes	Opaque binary data.

Enumerate Sessions

Enumerates sessions on a messaging entity.

Request

The request message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
operation	string	Yes	`com.microsoft:get-message-sessions`
`com.microsoft:server-timeout`	uint	No	Operation server timeout in milliseconds.

The request message body must consist of an **amqp-value** section containing a **map** with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
last-updated-time	timestamp	Yes	Filter to include only sessions updated after a given time.
skip	int	Yes	Skip a number of sessions.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
top	int	Yes	Maximum number of sessions.

Response

The response message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
statusCode	int	Yes	HTTP response code [RFC2616] 200: OK – has more messages 0xcc: No content – no more messages
statusDescription	string	No	Description of the status.

The response message body must consist of an **amqp-value** section containing a **map** with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
skip	int	Yes	Number of skipped sessions if status code is 200.
sessions-ids	array of strings	Yes	Array of session IDs if status code is 200.

Rule operations

Add Rule

Request

The request message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
operation	string	Yes	`com.microsoft:add-rule`
`com.microsoft:server-timeout`	uint	No	Operation server timeout in milliseconds.

The request message body must consist of an **amqp-value** section containing a **map** with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
rule-name	string	Yes	Rule name, not including subscription and topic names.
rule-description	map	Yes	Rule description as specified in next section.

The **rule-description** map must include the following entries, where **sql-filter** and **correlation-filter** are mutually exclusive.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
sql-filter	map	Yes	`sql-filter`, as specified in the next section.
correlation-filter	map	Yes	`correlation-filter`, as specified in the next section.
sql-rule-action	map	Yes	`sql-rule-action`, as specified in the next section.

The sql-filter map must include the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
expression	string	Yes	Sql filter expression.

The **correlation-filter** map must include at least one of the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
correlation-id	string	No	
message-id	string	No	
to	string	No	
reply-to	string	No	
label	string	No	
session-id	string	No	
reply-to-session-id	string	No	
content-type	string	No	
properties	map	No	Maps to Service Bus BrokeredMessage.Properties.

The **sql-rule-action** map must include the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
expression	string	Yes	Sql action expression.

Response

The response message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
statusCode	int	Yes	HTTP response code [RFC2616] 200: OK – success, otherwise failed
statusDescription	string	No	Description of the status.

Remove Rule

Request

The request message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
operation	string	Yes	`com.microsoft:remove-rule`
`com.microsoft:server-timeout`	uint	No	Operation server timeout in milliseconds.

The request message body must consist of an **amqp-value** section containing a **map** with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
rule-name	string	Yes	Rule name, not including subscription and topic names.

Response

The response message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
statusCode	int	Yes	HTTP response code [RFC2616] 200: OK – success, otherwise failed
statusDescription	string	No	Description of the status.

Deferred message operations

Receive by sequence number

Receives deferred messages by sequence number.

Request

The request message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
operation	string	Yes	`com.microsoft:receive-by-sequence-number`

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
`com.microsoft:server-timeout`	uint	No	Operation server timeout in milliseconds.

The request message body must consist of an **amqp-value** section containing a **map** with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
sequence-numbers	array of long	Yes	Sequence numbers.
receiver-settle-mode	ubyte	Yes	Receiver settle mode as specified in AMQP core v1.0.

Response

The response message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
statusCode	int	Yes	HTTP response code [RFC2616] 200: OK – success, otherwise failed
statusDescription	string	No	Description of the status.

The response message body must consist of an **amqp-value** section containing a **map** with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
messages	list of maps	Yes	List of messages where every map represents a message.

The map representing a message must contain the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
lock-token	uuid	Yes	Lock token if `receiver-settle-mode` is 1.
message	array of byte	Yes	AMQP 1.0 wire-encoded message.

Update disposition status

Updates the disposition status of deferred messages.

Request

The request message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
operation	string	Yes	`com.microsoft:update-disposition`

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
`com.microsoft:server-timeout`	uint	No	Operation server timeout in milliseconds.

The request message body must consist of an **amqp-value** section containing a **map** with the following entries.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
disposition-status	string	Yes	completed abandoned suspended
lock-tokens	array of uuid	Yes	Message lock tokens to update disposition status.
deadletter-reason	string	No	May be set if disposition status is set to **suspended**.
deadletter-description	string	No	May be set if disposition status is set to **suspended**.
properties-to-modify	map	No	List of Service Bus brokered message properties to modify.

Response

The response message must include the following application properties.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
statusCode	int	Yes	HTTP response code [RFC2616] 200: OK – success, otherwise failed
statusDescription	string	No	Description of the status.

Next steps

to learn more about AMQP and Service bus, visit the following links:

- Service Bus AMQP overview
- AMQP 1.0 support for Service Bus partitioned queues and topics
- AMQP in Service Bus for Windows Server

Service Bus management libraries

1/18/2017 • 1 min to read • Edit on GitHub

The Service Bus management libraries can dynamically provision Service Bus namespaces and entities. This allows for complex deployments and messaging scenarios, enabling you to programmatically determine what entities to provision. These libraries are currently available for .NET.

Supported functionality

- Namespace creation, update, deletion
- Queue creation, update, deletion
- Topic creation, update, deletion
- Subscription creation, update, deletion

Prerequisites

To get started using the Service Bus management libraries, you must authenticate with Azure Active Directory (AAD). AAD requires that you authenticate as a service principal which provides access to your Azure resources. For information about creating a service principal, see one of these articles:

- Use the Azure Portal to create Active Directory application and service principal that can access resources
- Use Azure PowerShell to create a service principal to access resources
- Use Azure CLI to create a service principal to access resources

These tutorials will provide you with an `AppId` (Client ID), `TenantId`, and `ClientSecret` (Authentication Key), all of which are used for authentication by the management libraries. You must have 'Owner' permissions for the resource group on which you wish to run.

Programming pattern

The pattern to manipulate any Service Bus resource follows a common protocol:

1. Obtain a token from Azure Active Directory using the `Microsoft.IdentityModel.Clients.ActiveDirectory` library.

```
var context = new AuthenticationContext($"https://login.windows.net/{tenantId}");

var result = await context.AcquireTokenAsync(
    "https://management.core.windows.net/",
    new ClientCredential(clientId, clientSecret)
);
```

2. Create the `ServiceBusManagementClient` object.

```
var creds = new TokenCredentials(token);
var sbClient = new ServiceBusManagementClient(creds)
{
    SubscriptionId = SettingsCache["SubscriptionId"]
};
```

3. Set the CreateOrUpdate parameters to your specified values.

```
var queueParams = new QueueCreateOrUpdateParameters()
{
    Location = SettingsCache["DataCenterLocation"],
    EnablePartitioning = true
};
```

4. Execute the call.

```
await sbClient.Queues.CreateOrUpdateAsync(resourceGroupName, namespaceName, QueueName, queueParams);
```

Next steps

- .NET Management sample
- Microsoft.Azure.Management.ServiceBus Reference

Create Service Bus resources using Azure Resource Manager templates

1/23/2017 • 4 min to read • Edit on GitHub

This article describes how to create and deploy Service Bus resources using Azure Resource Manager templates, PowerShell, and the Service Bus resource provider.

Azure Resource Manager templates help you define the resources to deploy for a solution, and to specify parameters and variables that enable you to input values for different environments. The template consists of JSON and expressions that you can use to construct values for your deployment. For detailed information about writing Azure Resource Manager templates, and a discussion of the template format, see Authoring Azure Resource Manager templates.

NOTE
The examples in this article show how to use Azure Resource Manager to create a Service Bus namespace and messaging entity (queue). For other template examples, visit the Azure Quickstart Templates gallery and search for "Service Bus."

Service Bus Resource Manager templates

These Service Bus Azure Resource Manager templates are available for download and deployment. Click the following links for details about each one, with links to the templates on GitHub:

- Create a Service Bus namespace
- Create a Service Bus namespace with queue
- Create a Service Bus namespace with topic and subscription
- Create a Service Bus namespace with queue and authorization rule
- Create a Service Bus namespace with topic, subscription, and rule

Deploy with PowerShell

The following procedure describes how to use PowerShell to deploy an Azure Resource Manager template that creates a **Standard** tier Service Bus namespace, and a queue within that namespace. This example is based on the Create a Service Bus namespace with queue template. The approximate workflow is as follows:

1. Install PowerShell.
2. Create the template and (optionally) a parameter file.
3. In PowerShell, log in to your Azure account.
4. Create a new resource group if one does not exist.
5. Test the deployment.
6. If desired, set the deployment mode.
7. Deploy the template.

For complete information about deploying Azure Resource Manager templates, see Deploy resources with Azure Resource Manager templates.

Install PowerShell

Install Azure PowerShell by following the instructions in Get started with Azure PowerShell cmdlets.

Create a template

Clone or copy the 201-servicebus-create-queue template from GitHub:

```
{
    "$schema": "http://schema.management.azure.com/schemas/2014-04-01-preview/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "serviceBusNamespaceName": {
            "type": "string",
            "metadata": {
                "description": "Name of the Service Bus namespace"
            }
        },
        "serviceBusQueueName": {
            "type": "string",
            "metadata": {
                "description": "Name of the Queue"
            }
        },
        "serviceBusApiVersion": {
            "type": "string",
            "defaultValue": "2015-08-01",
            "metadata": {
                "description": "Service Bus ApiVersion used by the template"
            }
        }
    },
    "variables": {
        "location": "[resourceGroup().location]",
        "sbVersion": "[parameters('serviceBusApiVersion')]",
        "defaultSASKeyName": "RootManageSharedAccessKey",
        "authRuleResourceId": "[resourceId('Microsoft.ServiceBus/namespaces/authorizationRules',
parameters('serviceBusNamespaceName'), variables('defaultSASKeyName'))]"
    },
    "resources": [{
        "apiVersion": "[variables('sbVersion')]",
        "name": "[parameters('serviceBusNamespaceName')]",
        "type": "Microsoft.ServiceBus/Namespaces",
        "location": "[variables('location')]",
        "kind": "Messaging",
        "sku": {
            "name": "StandardSku",
            "tier": "Standard"
        },
        "resources": [{
            "apiVersion": "[variables('sbVersion')]",
            "name": "[parameters('serviceBusQueueName')]",
            "type": "Queues",
            "dependsOn": [
                "[concat('Microsoft.ServiceBus/namespaces/', parameters('serviceBusNamespaceName'))]"
            ],
            "properties": {
                "path": "[parameters('serviceBusQueueName')]"
            }
        }]
    }],
    "outputs": {
        "NamespaceConnectionString": {
            "type": "string",
            "value": "[listkeys(variables('authRuleResourceId'),
variables('sbVersion')).primaryConnectionString]"
        },
        "SharedAccessPolicyPrimaryKey": {
            "type": "string",
            "value": "[listkeys(variables('authRuleResourceId'), variables('sbVersion')).primaryKey]"
        }
    }
}
```

Create a parameters file (optional)

To use an optional parameters file, copy the 201-servicebus-create-queue file. Replace the value of `serviceBusNamespaceName` with the name of the Service Bus namespace you want to create in this deployment, and replace the value of `serviceBusQueueName` with the name of the queue you want to create.

```
{
    "$schema": "http://schema.management.azure.com/schemas/2015-01-01/deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "serviceBusNamespaceName": {
            "value": "<myNamespaceName>"
        },
        "serviceBusQueueName": {
            "value": "<myQueueName>"
        },
        "serviceBusApiVersion": {
            "value": "2015-08-01"
        }
    }
}
```

For more information, see the Parameters topic.

Log in to Azure and set the Azure subscription

From a PowerShell prompt, run the following command:

```
Login-AzureRmAccount
```

You are prompted to log on to your Azure account. After logging on, run the following command to view your available subscriptions.

```
Get-AzureRMSubscription
```

This command returns a list of available Azure subscriptions. Choose a subscription for the current session by running the following command. Replace `<YourSubscriptionId>` with the GUID for the Azure subscription you want to use.

```
Set-AzureRmContext -SubscriptionID <YourSubscriptionId>
```

Set the resource group

If you do not have an existing resource group, create a new resource group with the **New-AzureRmResourceGroup ** command. Provide the name of the resource group and location you want to use. For example:

```
New-AzureRmResourceGroup -Name MyDemoRG -Location "West US"
```

If successful, a summary of the new resource group is displayed.

```
ResourceGroupName : MyDemoRG
Location          : westus
ProvisioningState : Succeeded
Tags              :
ResourceId        : /subscriptions/<GUID>/resourceGroups/MyDemoRG
```

Test the deployment

Validate your deployment by running the `Test-AzureRmResourceGroupDeployment` cmdlet. When testing the deployment, provide parameters exactly as you would when executing the deployment.

```
Test-AzureRmResourceGroupDeployment -ResourceGroupName MyDemoRG -TemplateFile <path to template
file>\azuredeploy.json
```

Create the deployment

To create the new deployment, run the `New-AzureRmResourceGroupDeployment` cmdlet, and provide the necessary parameters when prompted. The parameters include a name for your deployment, the name of your resource group, and the path or URL to the template file. If the **Mode** parameter is not specified, the default value of **Incremental** is used. For more information, see Incremental and complete deployments.

The following command prompts you for the three parameters in the PowerShell window:

```
New-AzureRmResourceGroupDeployment -Name MyDemoDeployment -ResourceGroupName MyDemoRG -TemplateFile <path to
template file>\azuredeploy.json
```

To specify a parameters file instead, use the following command.

```
New-AzureRmResourceGroupDeployment -Name MyDemoDeployment -ResourceGroupName MyDemoRG -TemplateFile <path to
template file>\azuredeploy.json -TemplateParameterFile <path to parameters file>\azuredeploy.parameters.json
```

You can also use inline parameters when you run the deployment cmdlet. The command is as follows:

```
New-AzureRmResourceGroupDeployment -Name MyDemoDeployment -ResourceGroupName MyDemoRG -TemplateFile <path to
template file>\azuredeploy.json -parameterName "parameterValue"
```

To run a complete deployment, set the **Mode** parameter to **Complete**:

```
New-AzureRmResourceGroupDeployment -Name MyDemoDeployment -Mode Complete -ResourceGroupName MyDemoRG -
TemplateFile <path to template file>\azuredeploy.json
```

Verify the deployment

If the resources are deployed successfully, a summary of the deployment is displayed in the PowerShell window:

```
DeploymentName    : MyDemoDeployment
ResourceGroupName : MyDemoRG
ProvisioningState : Succeeded
Timestamp         : 4/19/2016 10:38:30 PM
Mode              : Incremental
TemplateLink      :
Parameters        :
                    Name             Type                       Value
                    ===============  =========================  ==========
                    serviceBusNamespaceName  String             <namespaceName>
                    serviceBusQueueName  String                 <queueName>
                    serviceBusApiVersion  String                2015-08-01
```

Next steps

You've now seen the basic workflow and commands for deploying an Azure Resource Manager template. For more detailed information, visit the following links:

- Azure Resource Manager overview
- Deploy resources with Resource Manager templates and Azure PowerShell
- Authoring Azure Resource Manager templates

Edit on GitHub

1 min to read •

Service Bus messaging exceptions

3/3/2017 • 9 min to read • Edit on GitHub

This article lists some exceptions generated by the Microsoft Azure Service Bus messaging APIs. This reference is subject to change, so check back for updates.

Exception categories

The messaging APIs generate exceptions that can fall into the following categories, along with the associated action you can take to try to fix them. Note that the meaning and causes of an exception can vary depending on the type of messaging entity (queues/topics or Event Hubs):

1. User coding error (System.ArgumentException, System.InvalidOperationException, System.OperationCanceledException, 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.MessagingException, Microsoft.ServiceBus.Messaging.ServerBusyException, Microsoft.ServiceBus.Messaging.MessagingCommunicationException). General action: retry the operation or notify users.
4. Other exceptions (System.Transactions.TransactionException, System.TimeoutException, Microsoft.ServiceBus.Messaging.MessageLockLostException, Microsoft.ServiceBus.Messaging.SessionLockLostException). General action: specific to the exception type; please refer to the table in the following section.

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
TimeoutException	The server did not respond to the requested operation within the specified time which is controlled by 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. See Timeout exceptions.	Retry might help in some cases; add retry logic to code.

EXCEPTION TYPE	DESCRIPTION/CAUSE/EXAMPLES	SUGGESTED ACTION	NOTE ON AUTOMATIC/IMMEDIATE RETRY
InvalidOperationException	The requested user operation is not allowed within the server or service. See the exception message for details. For example, 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.
OperationCanceledException	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.
UnauthorizedAccessException	The 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.
ArgumentException ArgumentNullException ArgumentOutOfRangeException	One or more arguments supplied to the method are invalid. The URI supplied to NamespaceManager or Create contains path segment(s). The URI scheme supplied to NamespaceManager or Create is invalid. The property value is larger than 32KB.	Check the calling code and make sure the arguments are correct.	Retry will not help.
MessagingEntityNotFoundException	Entity associated with the operation does not exist or it has been deleted.	Make sure the entity exists.	Retry will not help.
MessageNotFoundException	Attempt to receive a message with a particular sequence number. This message is not found.	Make sure the message has not been received already. Check the deadletter queue to see if the message has been deadlettered.	Retry will not help.
MessagingCommunicationException	Client is not able to establish a connection to Service Bus.	Make sure the supplied host name is correct and the host is reachable.	Retry might help if there are intermittent connectivity issues.
ServerBusyException	Service is not able to process the request at this time.	Client can wait for a period of time, then retry the operation.	Client may retry after certain interval. If a retry results in a different exception, check retry behavior of that exception.

EXCEPTION TYPE	DESCRIPTION/CAUSE/EXAMPLES	SUGGESTED ACTION	NOTE ON AUTOMATIC/IMMEDIATE RETRY
MessageLockLostException	Lock token associated with the message has expired, or the lock token is not found.	Dispose the message.	Retry will not help.
SessionLockLostException	Lock associated with this session is lost.	Abort the MessageSession object.	Retry will not help.
MessagingException	Generic messaging exception that may be thrown in the following cases: An attempt is made to create a 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. The server or service encountered an error during processing of the request. Please see the exception message for details. 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 value types of the properties and only use supported types. Check the IsTransient property. If it is **true**, you can retry the operation.	Retry behavior is undefined and might not help.
MessagingEntityAlreadyExistsException	Attempt to create an entity with a name that is already used by another entity in that service namespace.	Delete the existing entity or choose a different name for the entity to be created.	Retry will not help.
QuotaExceededException	The messaging entity has reached its maximum allowable size, or the maximum number of connections to a namespace has been exceeded.	Create space in the entity by receiving messages from the entity or its sub-queues. See QuotaExceededException.	Retry might help if messages have been removed in the meantime.
RuleActionException	Service Bus returns this exception if you attempt to create an invalid rule action. Service Bus attaches this exception to a deadlettered message if an error occurs while processing the rule action for that message.	Check the rule action for correctness.	Retry will not help.

EXCEPTION TYPE	DESCRIPTION/CAUSE/EXAMPLES	SUGGESTED ACTION	NOTE ON AUTOMATIC/IMMEDIATE RETRY
FilterException	Service Bus returns this exception if you attempt to create an invalid filter. Service Bus attaches this exception to a deadlettered message if an error occurred while processing the filter for that message.	Check the filter for correctness.	Retry will not help.
SessionCannotBeLockedException	Attempt to accept a session with a specific session ID, but the session is currently locked by another client.	Make sure the session is unlocked by other clients.	Retry might help if the session has been released in the interim.
TransactionSizeExceededException	Too many operations are part of the transaction.	Reduce the number of operations that are part of this transaction.	Retry will not help.
MessagingEntityDisabledException	Request for a runtime operation on a disabled entity.	Activate the entity.	Retry might help if the entity has been activated in the interim.
NoMatchingSubscriptionException	Service Bus returns this exception if you send a message to a topic that has pre-filtering enabled and none of the filters match.	Make sure at least one filter matches.	Retry will not help.
MessageSizeExceededException	A message payload exceeds the 256 KB limit. Note that the 256 KB limit is the total message size, which can include system properties and any .NET overhead.	Reduce the size of the message payload, then retry the operation.	Retry will not help.
TransactionException	The ambient transaction (*Transaction.Current*) is invalid. It may have been completed or aborted. Inner exception may provide additional information.		Retry will not help.
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.	Your application must handle this exception (as a special case), as the transaction may have already been committed.	-

QuotaExceededException

QuotaExceededException indicates that a quota for a specific entity has been exceeded.

Queues and topics

For queues and topics, this is often the size of the queue. The error message property will contain further details, as in the following example:

```
Microsoft.ServiceBus.Messaging.QuotaExceededException
Message: The maximum entity size has been reached or exceeded for Topic: ‘xxx-xxx-xxx’.
    Size of entity in bytes:1073742326, Max entity size in bytes:
1073741824..TrackingId:xxxxxxxxxxxxxxxxxxxxxxxxxx, TimeStamp:3/15/2013 7:50:18 AM
```

The message states that the topic exceeded its size limit, in this case 1 GB (the default size limit).

Namespaces

For namespaces, QuotaExceededException can indicate that an application has exceeded the maximum number of connections to a namespace. For example:

```
Microsoft.ServiceBus.Messaging.QuotaExceededException: ConnectionsQuotaExceeded for namespace xxx.
<tracking-id-guid>_G12 --->
System.ServiceModel.FaultException`1[System.ServiceModel.ExceptionDetail]:
ConnectionsQuotaExceeded for namespace xxx.
```

Common causes

There are two common causes for this error: the dead-letter queue, and non-functioning message receivers.

1. **Dead-letter queue** A reader is failing to complete messages and the messages are returned to the queue/topic when the lock expires. This can happen if the reader encounters an exception that prevents it from calling BrokeredMessage.Complete. After a message has been read 10 times, it moves to the dead-letter queue by default. This behavior is controlled by the QueueDescription.MaxDeliveryCount property and has a default value of 10. As messages pile up in the dead letter queue, they take up space.

   To resolve the issue, read and complete the messages from the dead-letter queue, as you would from any other queue. The QueueClient class even contains a FormatDeadLetterPath method to help format the dead-letter queue path.

2. **Receiver stopped** A receiver has stopped receiving messages from a queue or subscription. The way to identify this is to look at the QueueDescription.MessageCountDetails property, which shows the full breakdown of the messages. If the ActiveMessageCount property is high or growing, then the messages are not being read as fast as they're being written.

Event Hubs

Event Hubs has a limit of 20 consumer groups per Event Hub. When you attempt to create more, you receive a QuotaExceededException.

TimeoutException

A TimeoutException indicates that a user-initiated operation is taking longer than the operation timeout.

You should check the value of the ServicePointManager.DefaultConnectionLimit property, as hitting this limit can also cause a TimeoutException.

Queues and topics

For queues and topics, the timeout is specified either in the MessagingFactorySettings.OperationTimeout property, as part of the connection string, or through ServiceBusConnectionStringBuilder. The error message itself might vary, but it always contains the timeout value specified for the current operation.

Event Hubs

For Event Hubs, the timeout is specified either as part of the connection string, or through ServiceBusConnectionStringBuilder. The error message itself might vary, but it always contains the timeout value specified for the current operation.

Common causes

There are two common causes for this error: incorrect configuration, or a transient service error.

1. **Incorrect configuration** The operation timeout might be too small for the operational condition. The default value for the operation timeout in the client SDK is 60 seconds. Check to see if your code has the value set to something too small. Note that the condition of the network and CPU usage can affect the time it takes for a particular operation to complete, so the operation timeout should not be set to a very small value.
2. **Transient service error** Sometimes the Service Bus service can experience delays in processing requests; for example, during periods of high traffic. In such cases, you can retry your operation after a delay, until the operation is successful. If the same operation still fails after multiple attempts, please visit the Azure service status site to see if there are any known service outages.

Next steps

For the complete Service Bus and Event Hubs .NET API reference, see the Azure .NET API reference.

To learn more about Service Bus, see the following topics.

- Service Bus messaging overview
- Service Bus fundamentals
- Service Bus architecture

Service Bus quotas

1/17/2017 • 4 min to read • Edit on GitHub

This section lists basic quotas and throttling thresholds in Microsoft Azure Service Bus Messaging.

Messaging quotas

The following table lists quota information specific to Service Bus messaging. For information about pricing and other quotas for Service Bus, see the Service Bus Pricing overview.

QUOTA NAME	SCOPE	TYPE	BEHAVIOR WHEN EXCEEDED	VALUE
Maximum number of basic / standard namespaces per Azure subscription	Namespace	Static	Subsequent requests for additional basic / standard namespaces will be rejected by the portal.	100
Maximum number of premium namespaces per Azure subscription	Namespace	Static	Subsequent requests for additional premium namespaces will be rejected by the portal.	10
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 GB. If partitioning is enabled, the maximum queue/topic size is 80 GB.
Number of concurrent connections on a namespace	Namespace	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.	NetMessaging: 1,000 AMQP: 5,000
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.	Capped by the limit of concurrent connections per namespace.

QUOTA NAME	SCOPE	TYPE	BEHAVIOR WHEN EXCEEDED	VALUE
Number of concurrent receive requests on a queue/topic/subscription entity	Entity	Static	Subsequent receive requests will be rejected and an exception will be received by the calling code. This quota applies to the combined number of concurrent receive operations across all subscriptions on a topic.	5,000
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 Azure portal, an error message will be generated. If called from the management API, an exception will be received by the calling code.	10,000 The total number of topics plus queues in a service namespace must be less than or equal to 10,000. This is not applicable to Premium as all entities are partitioned.
Number of partitioned topics/queues per service namespace	System-wide	Static	Subsequent requests for creation of a new partitioned topic or queue on the service namespace will be rejected. As a result, if configured through the Azure portal, an error message will be generated. If called from the management API, a **QuotaExceededException** exception will be received by the calling code.	Basic and Standard Tiers - 100 Premium - 1,000 Each partitioned queue or topic counts towards the quota of 10,000 entities per namespace.
Maximum size of any messaging entity path: queue or topic	Entity	Static	-	260 characters
Maximum size of any messaging entity name: namespace, subscription, or subscription rule	Entity	Static	-	50 characters

QUOTA NAME	SCOPE	TYPE	BEHAVIOR WHEN EXCEEDED	VALUE
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 (Standard tier) / 1MB (Premium tier). **Note** Due to system overhead, this limit is usually slightly less. Maximum header size: 64KB Maximum number of header properties in property bag: **byte/int.MaxValue** Maximum size of property in property bag: No explicit limit. Limited by maximum header size.
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 cannot exceed 64K. This applies to the entire header of the BrokeredMessage, which has both user properties as well as system properties (such as SequenceNumber, Label, 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 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: 1024 (1K). Maximum length of rule action string: 1024 (1K). Maximum number of expressions per rule action: 32.
Number of SharedAccessAuthorizationRule rules per namespace, queue, or topic	Entity, namespace	Static	Subsequent requests for creation of additional rules will be rejected and an exception will be received by the calling code.	Maximum number of rules: 12. Rules that are configured on a Service Bus namespace apply to all queues and topics in that namespace.

SQLFilter syntax

1/23/2017 • 5 min to read • Edit on GitHub

A *SqlFilter* is an instance of the SqlFilter Class, and represents a SQL language-based filter expression that is evaluated against a BrokeredMessage. A SqlFilter supports a subset of the SQL-92 standard.

This topic lists details about SqlFilter grammar.

```
<predicate ::=
      { NOT <predicate> }
      | <predicate> AND <predicate>
      | <predicate> OR <predicate>
      | <expression> { = | <> | != | > | >= | < | <= } <expression>
      | <property> IS [NOT] NULL
      | <expression> [NOT] IN ( <expression> [, ...n] )
      | <expression> [NOT] LIKE <pattern> [ESCAPE <escape_char>]
      | EXISTS ( <property> )
      | ( <predicate> )
```

```
<expression> ::=
      <constant>
      | <function>
      | <property>
      | <expression> { + | - | * | / | % } <expression>
      | { + | - } <expression>
      | ( <expression> )
```

```
<property> :=
       [<scope> .] <property_name>
```

Arguments

- `<scope>` is an optional string indicating the scope of the `<property_name>`. Valid values are `sys` or `user`. The `sys` value indicates system scope where `<property_name>` is a public property name of the BrokeredMessage Class. `user` indicates user scope where `<property_name>` is a key of the BrokeredMessage Class dictionary. `user` scope is the default scope if `<scope>` is not specified.

Remarks

An attempt to access a non-existent system property is an error, while an attempt to access a non-existent user property is not an error. Instead, a non-existent user property is internally evaluated as an unknown value. An unknown value is treated specially during operator evaluation.

property_name

```
<property_name> ::=
     <identifier>
     | <delimited_identifier>

<identifier> ::=
     <regular_identifier> | <quoted_identifier> | <delimited_identifier>
```

Arguments

`<regular_identifier>` is a string represented by the following regular expression:

```
[[:IsLetter:]][_[:IsLetter:][:IsDigit:]]*
```

This means any string that starts with a letter and is followed by one or more underscore/letter/digit.

`[:IsLetter:]` means any Unicode character that is categorized as a Unicode letter. `System.Char.IsLetter(c)` returns `true` if `c` is a Unicode letter.

`[:IsDigit:]` means any Unicode character that is categorized as a decimal digit. `System.Char.IsDigit(c)` returns `true` if `c` is a Unicode digit.

A `<regular_identifier>` cannot be a reserved keyword.

`<delimited_identifier>` is any string that is enclosed with left/right square brackets ([]). A right square bracket is represented as two right square brackets. The following are examples of `<delimited_identifier>`:

```
[Property With Space]
[HR-EmployeeID]
```

`<quoted_identifier>` is any string that is enclosed with double quotation marks. A double quotation mark in identifier is represented as two double quotation marks. It is not recommended to use quoted identifiers because it can easily be confused with a string constant. Use a delimited identifier if possible. The following is an example of `<quoted_identifier>`:

```
"Contoso & Northwind"
```

pattern

```
<pattern> ::=
      <expression>
```

Remarks

`<pattern>` must be an expression that is evaluated as a string. It is used as a pattern for the LIKE operator. It can contain the following wildcard characters:

- `%`: Any string of zero or more characters.
- `_`: Any single character.

escape_char

```
<escape_char> ::=
      <expression>
```

Remarks

`<escape_char>` must be an expression that is evaluated as a string of length 1. It is used as an escape character for the LIKE operator.

For example, `property LIKE 'ABC\%' ESCAPE '\'` matches `ABC%` rather than a string that starts with `ABC`.

constant

```
<constant> ::=
      <integer_constant> | <decimal_constant> | <approximate_number_constant> | <boolean_constant> | NULL
```

Arguments

- `<integer_constant>` is a string of numbers that are not enclosed in quotation marks and do not contain decimal points. The values are stored as `System.Int64` internally, and follow the same range.

  The following are examples of long constants:

  ```
  1894
  2
  ```

- `<decimal_constant>` is a string of numbers that are not enclosed in quotation marks, and contain a decimal point. The values are stored as `System.Double` internally, and follow the same range/precision.

  In a future version, this number might be stored in a different data type to support exact number semantics, so you should not rely on the fact the underlying data type is `System.Double` for `<decimal_constant>`.

  The following are examples of decimal constants:

  ```
  1894.1204
  2.0
  ```

- `<approximate_number_constant>` is a number written in scientific notation. The values are stored as `System.Double` internally, and follow the same range/precision. The following are examples of approximate number constants:

  ```
  101.5E5
  0.5E-2
  ```

boolean_constant

```
<boolean_constant> :=
      TRUE | FALSE
```

Remarks

Boolean constants are represented by the keywords **TRUE** or **FALSE**. The values are stored as `System.Boolean`.

string_constant

```
<string_constant>
```

Remarks

String constants are enclosed in single quotation marks and include any valid Unicode characters. A single quotation mark embedded in a string constant is represented as two single quotation marks.

function

```
<function> :=
      newid() |
      property(name) | p(name)
```

Remarks

The `newid()` function returns a **System.Guid** generated by the `System.Guid.NewGuid()` method.

The `property(name)` function returns the value of the property referenced by `name`. The `name` value can be any valid expression that returns a string value.

Considerations

Consider the following SqlFilter semantics:

- Property names are case-insensitive.
- Operators follow C# implicit conversion semantics whenever possible.
- System properties are public properties exposed in BrokeredMessage instances.

  Consider the following `IS [NOT] NULL` semantics:

  - `property IS NULL` is evaluated as `true` if either the property doesn't exist or the property's value is `null`.

Property evaluation semantics:

- An attempt to evaluate a non-existent system property will throw a FilterException exception.
- A property that does not exist is internally evaluated as **unknown**.

  Unknown evaluation in arithmetic operators:

- For binary operators, if either the left and/or right side of operands is evaluated as **unknown**, then the result is **unknown**.
- For unary operators, if an operand is evaluated as **unknown**, then the result is **unknown**.

  Unknown evaluation in binary comparison operators:

- If either the left and/or right side of operands is evaluated as **unknown**, then the result is **unknown**.

  Unknown evaluation in `[NOT] LIKE`:

- If any operand is evaluated as **unknown** then the result is **unknown**.

  Unknown evaluation in `[NOT] IN`:

- If the left operand is evaluated as **unknown** then the result is **unknown**.

  Unknown evaluation in **AND** operator:

```
+---+---+---+---+
|AND| T | F | U |
+---+---+---+---+
| T | T | F | U |
+---+---+---+---+
| F | F | F | F |
+---+---+---+---+
| U | U | F | U |
+---+---+---+---+
```

Unknown evaluation in **OR** operator:

```
+---+---+---+---+
|OR | T | F | U |
+---+---+---+---+
| T | T | T | T |
+---+---+---+---+
| F | T | F | U |
+---+---+---+---+
| U | T | U | U |
+---+---+---+---+
```

Operator binding semantics:

- Comparison operators such as `>`, `>=`, `<`, `<=`, `!=`, and `=` follow the same semantics as the C# operator binding in data type promotions and implicit conversions.
- Arithmetic operators such as `+`, `-`, `*`, `/`, and `%` follow the same semantics as the C# operator binding in data type promotions and implicit conversions.

Next steps

- SQLFilter class
- SQLRuleAction class

SQLRuleAction syntax

1/17/2017 • 4 min to read • Edit on GitHub

A *SqlRuleAction* is an instance of the SqlRuleAction class, and represents set of actions written in SQL-language based syntax that is performed against a BrokeredMessage.

This topic lists details about the SQL rule action grammar.

```
<statements> ::=
    <statement> [, ...n]
```

```
<statement> ::=
    <action> [;]
    Remarks
    -------
    Semicolon is optional.
```

```
<action> ::=
    SET <property> = <expression>
    REMOVE <property>
```

```
<expression> ::=
    <constant>
    | <function>
    | <property>
    | <expression> { + | - | * | / | % } <expression>
    | { + | - } <expression>
    | ( <expression> )
```

```
<property> :=
    [<scope> .] <property_name>
```

Arguments

- `<scope>` is an optional string indicating the scope of the `<property_name>`. Valid values are `sys` or `user`. The `sys` value indicates system scope where `<property_name>` is a public property name of the BrokeredMessage Class. `user` indicates user scope where `<property_name>` is a key of the BrokeredMessage Class dictionary. `user` scope is the default scope if `<scope>` is not specified.

Remarks

An attempt to access a non-existent system property is an error, while an attempt to access a non-existent user property is not an error. Instead, a non-existent user property is internally evaluated as an unknown value. An unknown value is treated specially during operator evaluation.

property_name

```
<property_name> ::=
     <identifier>
     | <delimited_identifier>

<identifier> ::=
     <regular_identifier> | <quoted_identifier> | <delimited_identifier>
```

Arguments

`<regular_identifier>` is a string represented by the following regular expression:

```
[[:IsLetter:]][_[:IsLetter:][:IsDigit:]]*
```

This means any string that starts with a letter and is followed by one or more underscore/letter/digit.

`[:IsLetter:]` means any Unicode character that is categorized as a Unicode letter. `System.Char.IsLetter(c)` returns `true` if `c` is a Unicode letter.

`[:IsDigit:]` means any Unicode character that is categorized as a decimal digit. `System.Char.IsDigit(c)` returns `true` if `c` is a Unicode digit.

A `<regular_identifier>` cannot be a reserved keyword.

`<delimited_identifier>` is any string that is enclosed with left/right square brackets ([]). A right square bracket is represented as two right square brackets. The following are examples of `<delimited_identifier>`:

```
[Property With Space]
[HR-EmployeeID]
```

`<quoted_identifier>` is any string that is enclosed with double quotation marks. A double quotation mark in identifier is represented as two double quotation marks. It is not recommended to use quoted identifiers because it can easily be confused with a string constant. Use a delimited identifier if possible. The following is an example of `<quoted_identifier>`:

```
"Contoso & Northwind"
```

pattern

```
<pattern> ::=
      <expression>
```

Remarks

`<pattern>` must be an expression that is evaluated as a string. It is used as a pattern for the LIKE operator. It can contain the following wildcard characters:

- `%`: Any string of zero or more characters.
- `_`: Any single character.

escape_char

```
<escape_char> ::=
      <expression>
```

Remarks

`<escape_char>` must be an expression that is evaluated as a string of length 1. It is used as an escape character for the LIKE operator.

For example, `property LIKE 'ABC\%' ESCAPE '\'` matches `ABC%` rather than a string that starts with `ABC`.

constant

```
<constant> ::=
      <integer_constant> | <decimal_constant> | <approximate_number_constant> | <boolean_constant> | NULL
```

Arguments

- `<integer_constant>` is a string of numbers that are not enclosed in quotation marks and do not contain decimal points. The values are stored as `System.Int64` internally, and follow the same range.

  The following are examples of long constants:

  ```
  1894
  2
  ```

- `<decimal_constant>` is a string of numbers that are not enclosed in quotation marks, and contain a decimal point. The values are stored as `System.Double` internally, and follow the same range/precision.

  In a future version, this number might be stored in a different data type to support exact number semantics, so you should not rely on the fact the underlying data type is `System.Double` for `<decimal_constant>`.

  The following are examples of decimal constants:

  ```
  1894.1204
  2.0
  ```

- `<approximate_number_constant>` is a number written in scientific notation. The values are stored as `System.Double` internally, and follow the same range/precision. The following are examples of approximate number constants:

  ```
  101.5E5
  0.5E-2
  ```

boolean_constant

```
<boolean_constant> :=
      TRUE | FALSE
```

Remarks

Boolean constants are represented by the keywords `TRUE` or `FALSE`. The values are stored as `System.Boolean`.

string_constant

```
<string_constant>
```

Remarks

String constants are enclosed in single quotation marks and include any valid Unicode characters. A single quotation mark embedded in a string constant is represented as two single quotation marks.

function

```
<function> :=
      newid() |
      property(name) | p(name)
```

Remarks

The `newid()` function returns a **System.Guid** generated by the `System.Guid.NewGuid()` method.

The `property(name)` function returns the value of the property referenced by `name`. The `name` value can be any valid expression that returns a string value.

Considerations

- SET is used to create a new property or update the value of an existing property.
- REMOVE is used to remove a property.
- SET performs implicit conversion if possible when the expression type and the existing property type are different.
- Action fails if non-existent system properties were referenced.
- Action does not fail if non-existent user properties were referenced.
- A non-existent user property is evaluated as "Unknown" internally, following the same semantics as SQLFilter when evaluating operators.

Next steps

- SQLRuleAction class
- SQLFilter class

Service Bus pricing and billing

2/16/2017 • 7 min to read • Edit on GitHub

Service Bus is offered in Basic, Standard, and Premium tiers. You can choose a service tier for each Service Bus service namespace that you create, and this tier selection applies across all entities created within that namespace.

> **NOTE**
> For detailed information about current Service Bus pricing, see the Azure Service Bus pricing page, and the Service Bus FAQ.

Service Bus uses the following two meters for queues and topics/subscriptions:

1. **Messaging Operations**: Defined as API calls against queue or topic/subscription service endpoints. This meter will replace messages sent or received as the primary unit of billable usage for queues and topics/subscriptions.
2. **Brokered Connections**: Defined as the peak number of persistent connections open against queues, topics, or subscriptions during a given one-hour sampling period. This meter will only apply in the Standard tier, in which you can open additional connections (previously, connections were limited to 100 per queue/topic/subscription) for a nominal per-connection fee.

The **Standard** tier introduces graduated pricing for operations performed with queues and topics/subscriptions, resulting in volume-based discounts of up to 80% at the highest usage levels. There is also a Standard tier base charge of $10 per month, which enables you to perform up to 12.5 million operations per month at no additional cost.

The **Premium** tier provides resource isolation at the CPU and memory layer so that each customer workload runs in isolation. This resource container is called a *messaging unit*. Each premium namespace is allocated at least one messaging unit. You can purchase 1, 2, or 4 messaging units for each Service Bus Premium namespace. A single workload or entity can span multiple messaging units and the number of messaging units can be changed at will, although billing is in 24-hour or daily rate charges. The result is predictable and repeatable performance for your Service Bus-based solution. Not only is this performance more predictable and available, but it is also faster. Azure Service Bus Premium messaging builds on the storage engine introduced in Azure Event Hubs. With Premium messaging, peak performance is much faster than the Standard tier.

Note that the standard base charge is charged only once per month per Azure subscription. This means that after you create a single Standard or Premium tier Service Bus namespace, you will be able to create as many additional Standard or Premium tier namespaces as you want under that same Azure subscription, without incurring additional base charges.

All existing Service Bus namespaces created prior to November 1, 2014 were automatically placed into the Standard tier. This ensures that you continue to have access to all features currently available with Service Bus. Subsequently, you can use the Azure classic portal to downgrade to the Basic tier if desired.

The following table summarizes the functional differences between the Basic and Standard/Premium tiers.

CAPABILITY	BASIC	STANDARD/PREMIUM
Queues	Yes	Yes
Scheduled messages	Yes	Yes

CAPABILITY	BASIC	STANDARD/PREMIUM
Topics/Subscriptions	No	Yes
Relays	No	Yes
Transactions	No	Yes
De-Duplication	No	Yes
Sessions	No	Yes
Large messages	No	Yes
ForwardTo	No	Yes
SendVia	No	Yes
Brokered connections (included)	100 per Service Bus namespace	1,000 per Azure subscription
Brokered connections (overage allowed)	No	Yes (billable)

Messaging operations

As part of the new pricing model, billing for queues and topics/subscriptions is changing. These entities are transitioning from billing per message to billing per operation. An "operation" refers to any API call made against a queue or topic/subscription service endpoint. This includes management, send/receive, and session state operations.

OPERATION TYPE	DESCRIPTION
Management	Create, Read, Update, Delete (CRUD) against queues or topics/subscriptions.
Messaging	Sending and receiving messages with queues or topics/subscriptions.
Session state	Getting or setting session state on a queue or topic/subscription.

The following prices were effective starting November 1, 2014:

BASIC	COST
Operations	$0.05 per million operations

STANDARD	COST
Base charge	$10/month
First 12.5 million operations/month	Included

STANDARD	COST
12.5-100 million operations/month	$0.80 per million operations
100 million-2,500 million operations/month	$0.50 per million operations
Over 2,500 million operations/month	$0.20 per million operations

PREMIUM	COST
Daily	$11.13 fixed rate per Message Unit

Brokered connections

Brokered connections accommodate customer usage patterns that involve a large number of "persistently connected" senders/receivers against queues, topics, or subscriptions. Persistently connected senders/receivers are those that connect using either AMQP or HTTP with a non-zero receive timeout (for example, HTTP long polling). HTTP senders and receivers with an immediate timeout do not generate brokered connections.

Previously, queues and topics/subscriptions had a limit of 100 concurrent connections per URL. The current billing scheme removes the per-URL limit for queues and topics/subscriptions, and implements quotas and metering on brokered connections at the Service Bus namespace and Azure subscription levels.

The Basic tier includes, and is strictly limited to, 100 brokered connections per Service Bus namespace. Connections above this number will be rejected in the Basic tier. The Standard tier removes the per-namespace limit and counts aggregate brokered connection usage across the Azure subscription. In the Standard tier, 1,000 brokered connections per Azure subscription will be allowed at no extra cost (beyond the base charge). Using more than a total of 1,000 brokered connections across Standard-tier Service Bus namespaces in an Azure subscription will be billed on a graduated schedule, as shown in the following table.

BROKERED CONNECTIONS (STANDARD TIER)	COST
First 1,000/month	Included with base charge
1,000-100,000/month	$0.03 per connection/month
100,000-500,000/month	$0.025 per connection/month
Over 500,000/month	$0.015 per connection/month

NOTE
1,000 brokered connections are included with the Standard messaging tier (via the base charge) and can be shared across all queues, topics, and subscriptions within the associated Azure subscription.

NOTE
Billing is based on the peak number of concurrent connections and is prorated hourly based on 744 hours per month.

PREMIUM TIER
Brokered connections are not charged in the Premium tier.

For more information about brokered connections, see the FAQ section later in this topic.

WCF Relay

WCF Relays are available only in Standard tier namespaces. Otherwise, pricing and connection quotas for relays remain unchanged. This means that relays will continue to be charged on the number of messages (not operations), and relay hours.

WCF RELAY PRICING	COST
WCF Relay hours	$0.10 for every 100 relay hours
Messages	$0.01 for every 10,000 messages

FAQ

How is the WCF Relay Hours meter calculated?

See this topic.

What are brokered connections and how do I get charged for them?

A brokered connection is defined as one of the following:

1. An AMQP connection from a client to a Service Bus queue or topic/subscription.
2. An HTTP call to receive a message from a Service Bus topic or queue that has a receive timeout value greater than zero.

Service Bus charges for the peak number of concurrent brokered connections that exceed the included quantity (1,000 in the Standard tier). Peaks are measured on an hourly basis, prorated by dividing by 744 hours in a month, and added up over the monthly billing period. The included quantity (1,000 brokered connections per month) is applied at the end of the billing period against the sum of the prorated hourly peaks.

For example:

1. Each of 10,000 devices connects via a single AMQP connection, and receives commands from a Service Bus topic. The devices send telemetry events to an Event Hub. If all devices connect for 12 hours each day, the following connection charges apply (in addition to any other Service Bus topic charges): 10,000 connections * 12 hours * 31 days / 744 = 5,000 brokered connections. After the monthly allowance of 1,000 brokered connections, you would be charged for 4,000 brokered connections, at the rate of $0.03 per brokered connection, for a total of $120.
2. 10,000 devices receive messages from a Service Bus queue via HTTP, specifying a non-zero timeout. If all devices connect for 12 hours every day, you will see the following connection charges (in addition to any other Service Bus charges): 10,000 HTTP Receive connections * 12 hours per day * 31 days / 744 hours = 5,000 brokered connections.

Do brokered connection charges apply to queues and topics/subscriptions?

Yes. There are no connection charges for sending events using HTTP, regardless of the number of sending systems or devices. Receiving events with HTTP using a timeout greater than zero, sometimes called "long polling," generates brokered connection charges. AMQP connections generate brokered connection charges regardless of whether the connections are being used to send or receive. Note that 100 brokered connections are allowed at no charge in a Basic namespace. This is also the maximum number of brokered connections allowed for the Azure subscription. The first 1,000 brokered connections across all Standard namespaces in an Azure subscription are included at no extra charge (beyond the base charge). Because these allowances are enough to cover many service-to-service messaging scenarios, brokered connection charges usually only become relevant if you plan to use AMQP or HTTP long-polling with a large number of clients; for example, to achieve more efficient event streaming or enable bi-directional communication with many devices or application instances.

Next steps

- For more details about Service Bus pricing, see the Azure Service Bus pricing page.
- See the Service Bus FAQ for some common FAQs around Service bus pricing and billing.

Service Bus messaging samples

1/17/2017 • 5 min to read • Edit on GitHub

The Service Bus messaging samples demonstrate key features in Service Bus messaging (cloud service) and Service Bus for Windows Server. This article categorizes and describes the samples available, with links to each.

NOTE

Service Bus samples are not installed with the SDK. To obtain the samples, visit the Azure SDK samples page.

Additionally, there is an updated set of Service Bus messaging samples here (as of this writing, they are not described in this article).

For relay samples, see Service Bus relay samples.

Service Bus messaging

The following samples illustrate how to write applications that use Service Bus messaging.

Note that the messaging samples require a connection string to access your Service Bus namespace.

To obtain a connection string for Azure Service Bus

1. Log on to the Azure portal.
2. In the left-hand column, click **Service Bus**.
3. Click the name of your namespace in the list.
4. In the namespace blade, click **Shared access policies**.
5. In the **Shared access policies** blade, click **RootManageSharedAccessKey**.
6. Copy the connection string to the clipboard.

To obtain a connection string for Service Bus for Windows Server

1. Run the following PowerShell cmdlet:

```
get-sbClientConfiguration
```

2. Paste the connection string into the App.config file for the sample.

Getting started samples

These samples describe basic messaging functionality.

SAMPLE NAME	DESCRIPTION	MINIMUM SDK VERSION	AVAILABILITY
Getting Started: Messaging with Queues	Demonstrates how to use Microsoft Azure Service Bus to send and receive messages from a queue.	1.8	Microsoft Azure Service Bus; Service Bus for Windows Server

SAMPLE NAME	DESCRIPTION	MINIMUM SDK VERSION	AVAILABILITY
Getting Started: Messaging With Topics	Demonstrates how to use Microsoft Azure Service Bus to send and receive messages from a topic with multiple subscriptions.	1.8	Microsoft Azure Service Bus; Service Bus for Windows Server

Exploring features

The following samples demonstrate various features of Service Bus.

SAMPLE NAME	DESCRIPTION	MINIMUM SDK VERSION	AVAILABILITY
HTTP Token Providers	Demonstrates the different ways of authenticating an HTTP/REST client with Service Bus.	2.1	Microsoft Azure Service Bus; Service Bus for Windows Server
Service Bus HTTP Client	Demonstrates how to send messages to and receive messages from Service Bus via HTTP/REST.	2.3	Microsoft Azure Service Bus; Service Bus for Windows Server
Service Bus Autoforwarding	Demonstrates how to automatically forward messages from a queue, subscription, or deadletter queue into another queue or topic. It also demonstrates how to send a message into a queue or topic via a transfer queue.	2.3	Microsoft Azure Service Bus; Service Bus for Windows Server
Brokered Messaging: WCF Channel Session Sample	Demonstrates how to use Microsoft Azure Service Bus using Windows Communication Foundation (WCF) channels. The sample shows the use of WCF channels to send and receive messages via a Service Bus queue. The sample shows both session and non-session communication over the Service Bus.	1.8	Microsoft Azure Service Bus; Service Bus for Windows Server
Brokered Messaging: Transactions	Demonstrates how to use the Microsoft Azure Service Bus messaging features within a transaction scope in order to ensure batches of messaging operations are committed atomically.	1.8	Microsoft Azure Service Bus; Service Bus for Windows Server
Brokered Messaging: Management Operations Using REST	Demonstrates how to perform management operations on Service Bus using REST.	1.8	Microsoft Azure Service Bus; Service Bus for Windows Server

SAMPLE NAME	DESCRIPTION	MINIMUM SDK VERSION	AVAILABILITY
Resource Provider REST APIs	Demonstrates how to use the new Service Bus RDFE REST APIs to manage namespaces and messaging entities.	1.8	Microsoft Azure Service Bus; Service Bus for Windows Server
Brokered Messaging: WCF Service Session Sample	Demonstrates how to use Microsoft Azure Service Bus using the WCF service model. The sample shows the use of the WCF service model to accomplish session-based communication via a Service Bus queue.	1.8	Microsoft Azure Service Bus; Service Bus for Windows Server
Brokered Messaging: Request Response	Demonstrates how to use the Microsoft Azure Service Bus and the request/response functionality. The sample shows simple clients and servers communicating via a Service Bus queue.	1.8	Microsoft Azure Service Bus; Service Bus for Windows Server
Brokered Messaging: Dead Letter Queue	Demonstrates how to use Microsoft Azure Service Bus and the messaging "dead letter queue" functionality. The sample shows a simple sender and receiver communicating via a Service Bus queue.	1.8	Microsoft Azure Service Bus; Service Bus for Windows Server
Brokered Messaging: Deferred Messages	Demonstrates how to use the message deferral feature of Microsoft Azure Service Bus. The sample shows a simple sender and receiver communicating via a Service Bus queue.	1.8	Microsoft Azure Service Bus; Service Bus for Windows Server
Brokered Messaging: Session Messages	Demonstrates how to use Microsoft Azure Service Bus and the Messaging Session functionality. The sample shows simple senders and receivers communicating via a Service Bus queue.	1.8	Microsoft Azure Service Bus; Service Bus for Windows Server
Brokered Messaging: Request Response Topic	Demonstrates how to implement the request/response pattern using Microsoft Azure Service Bus topics and subscriptions. The sample shows simple clients and servers communicating via a Service Bus topic.	1.8	Microsoft Azure Service Bus; Service Bus for Windows Server

SAMPLE NAME	DESCRIPTION	MINIMUM SDK VERSION	AVAILABILITY
Brokered Messaging: Request Response Queue	Demonstrates how to use Microsoft Azure Service Bus and the request/response functionality. The sample shows simple clients and servers communicating via two Service Bus queues.	1.8	Microsoft Azure Service Bus; Service Bus for Windows Server
Brokered Messaging: Duplicate Detection	Demonstrates how to use Microsoft Azure Service Bus duplicate message detection with queues. It creates two queues, one with duplicate detection enabled and other one without duplicate detection.	1.8	Microsoft Azure Service Bus; Service Bus for Windows Server
Brokered Messaging: Async Messaging	Demonstrates how to use Microsoft Azure Service Bus to send and receive messages asynchronously from a queue. The queue provides decoupled, asynchronous communication between a sender and any number of receivers (here, a single receiver).	1.8	Microsoft Azure Service Bus; Service Bus for Windows Server
Brokered Messaging: Advanced Filters	Demonstrates how to use Microsoft Azure Service Bus publish/subscribe advanced filters. It creates a topic and 3 subscriptions with different filter definitions, sends messages to the topic, and receives all messages from subscriptions.	1.8	Microsoft Azure Service Bus; Service Bus for Windows Server
Brokered Messaging: Messages Prefetch	Demonstrates how to use the Microsoft Azure Service Bus messages prefetch feature. It demonstrates how to use the messages prefetch feature upon receive.	1.8	Microsoft Azure Service Bus; Service Bus for Windows Server

Service Bus reference tools

The following samples demonstrate various other features of the service.

SAMPLE NAME	DESCRIPTION	MINIMUM SDK VERSION	AVAILABILITY

SAMPLE NAME	DESCRIPTION	MINIMUM SDK VERSION	AVAILABILITY
Service Bus Explorer	The Service Bus Explorer allows users to connect to a Service Bus service namespace and manage messaging entities in an easy manner. The tool provides advanced features such as import/export functionality, and the ability to test messaging entities and relay services.	1.8	Microsoft Azure Service Bus; Service Bus for Windows Server
Authorization: SBAzTool	This sample demonstrates how to create and manage service identities in Microsoft Azure Active Directory Access Control (also known as Access Control Service or ACS) for use with Service Bus.	N/A	Microsoft Azure Service Bus

Next steps

See the following topics for conceptual overviews of Service Bus.

- Service Bus messaging overview
- Service Bus architecture
- Service Bus fundamentals