

Table of Contents

[Service Bus Messaging Documentation](#)

[Overview](#)

[What is Service Bus Messaging?](#)

[Service Bus fundamentals](#)

[Service Bus architecture](#)

[FAQ](#)

[Quickstarts](#)

[Create a namespace](#)

[Use queues](#)

[.NET](#)

[Java](#)

[Node.js](#)

[PHP](#)

[Python](#)

[Ruby](#)

[REST](#)

[Use topics and subscriptions](#)

[.NET](#)

[Java](#)

[Node.js](#)

[PHP](#)

[Python](#)

[Ruby](#)

[How To](#)

[Plan and design](#)

[Managed Service Identity \(preview\)](#)

[Role-Based Access Control \(preview\)](#)

[Premium messaging](#)

[Compare Azure Queues and Service Bus queues](#)

[Optimize performance](#)

[Geo-disaster recovery and geo-replication](#)

[Asynchronous messaging and high availability](#)

[Handling outages and disasters](#)

Develop

[Build a multi-tier Service Bus application](#)

[Message handling](#)

[Authentication and authorization](#)

[Topic filters and actions](#)

[Partitioned queues and topics](#)

[Message sessions](#)

[AMQP](#)

[Advanced features](#)

[End-to-end tracing and diagnostics](#)

Manage

[Monitor Service Bus with Azure Monitoring](#)

[Service Bus management libraries](#)

[Diagnostic logs](#)

[Suspend and reactivate messaging entities](#)

[Use Azure Resource Manager templates](#)

[Using Azure PowerShell to provision entities](#)

Reference

.NET

[Microsoft.ServiceBus.Messaging \(.NET Framework\)](#)

[Microsoft.Azure.ServiceBus \(.NET Standard\)](#)

Java

[Azure PowerShell](#)

REST

[Exceptions](#)

[Quotas](#)

[SQLFilter syntax](#)

[SQLRuleAction syntax](#)

Resources

[Azure Roadmap](#)

[Blog](#)

[MSDN forums](#)

[Pricing](#)

[Pricing calculator](#)

[Pricing details](#)

[Samples](#)

[ServiceBus360](#)

[Service Bus Explorer](#)

[Service updates](#)

[Stack Overflow](#)

[Videos](#)

The messaging service provides dependable information delivery as a brokered or third-party communication mechanism.

[Learn about Service Bus Messaging](#)

[Azure Service Bus Messaging Video Library](#)

[Create a namespace using the portal](#)

Reference

Command-Line

[Azure PowerShell](#)

Languages

[.NET Framework](#)

[.NET Standard](#)

[Java](#)

REST

[REST API Reference](#)

OTHERS

[Exceptions](#)

[Quotas](#)

[SQLFilter syntax](#)

[SQLRuleAction syntax](#)

Service Bus messaging: flexible data delivery in the cloud

12/21/2017 • 4 min to read • [Edit Online](#)

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 Azure Relay 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 with [queues, topics, and subscriptions](#) 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" (for example, a queue) 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 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 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](#)
- [Get started with Service Bus queues](#)
- [How to use Service Bus topics and subscriptions](#)

Azure Service Bus

1/31/2018 • 9 min to read • [Edit Online](#)

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 looks 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 needed is a connection between applications, and queues are not required. Azure 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 needed within that namespace. Figure 1 shows this architecture:

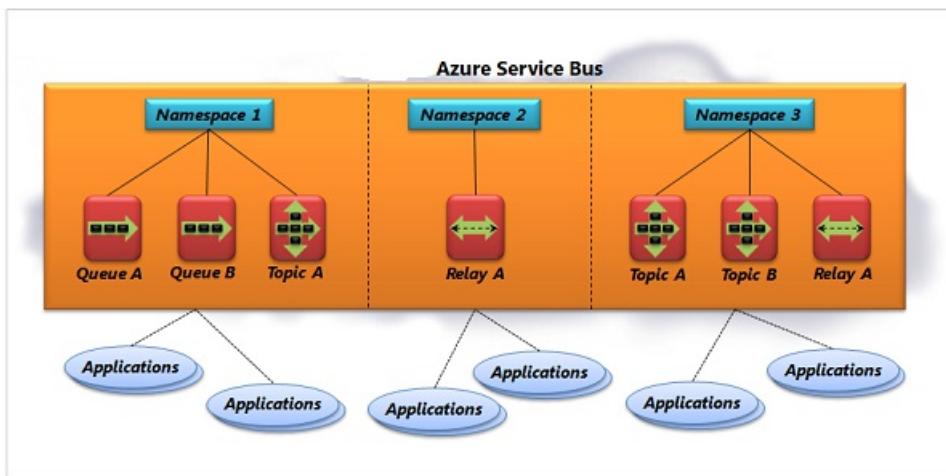


Figure 1: Service Bus provides a multi-tenant service for connecting applications through the cloud.

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 each other.

To use any of these objects in the relay scenario, Windows applications can use Windows Communication Foundation (WCF). This service is known as [WCF Relay](#). 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 connect these devices to each 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:

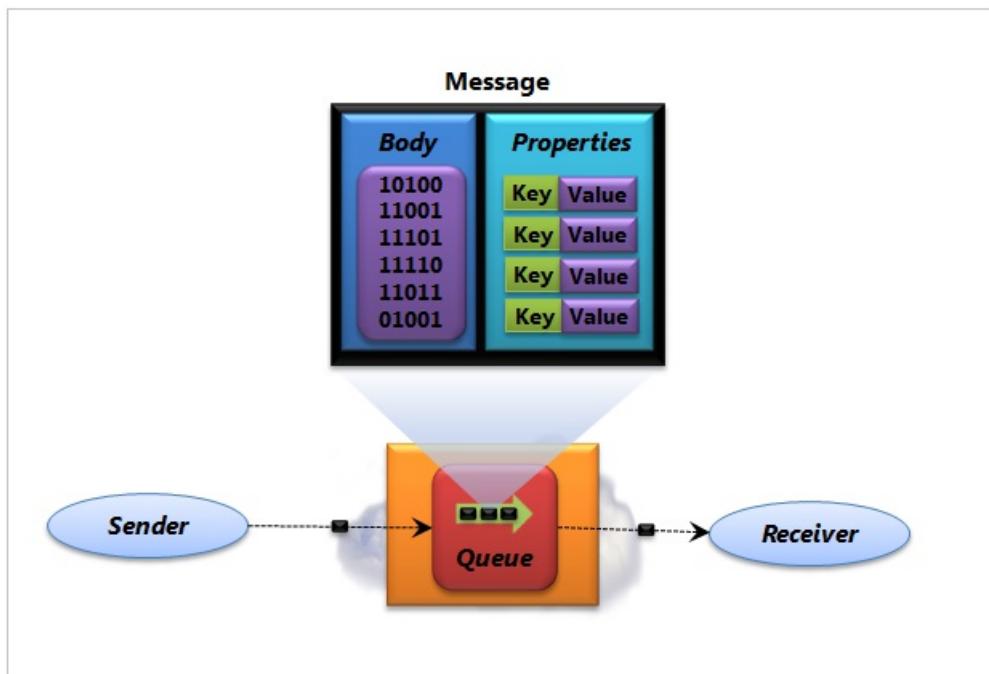


Figure 2: Service Bus queues provide one-way asynchronous queuing.

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, remain empty.

A receiver can read a message from a Service Bus queue in two different ways. The first option, called [ReceiveAndDelete](#), receives a message from the queue and immediately deletes it. This option is simple, but if the receiver crashes before it finishes processing the message, the message is 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 methods 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.

Note 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 event. 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 if both are not 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, topics are better. Figure 3 illustrates this idea:

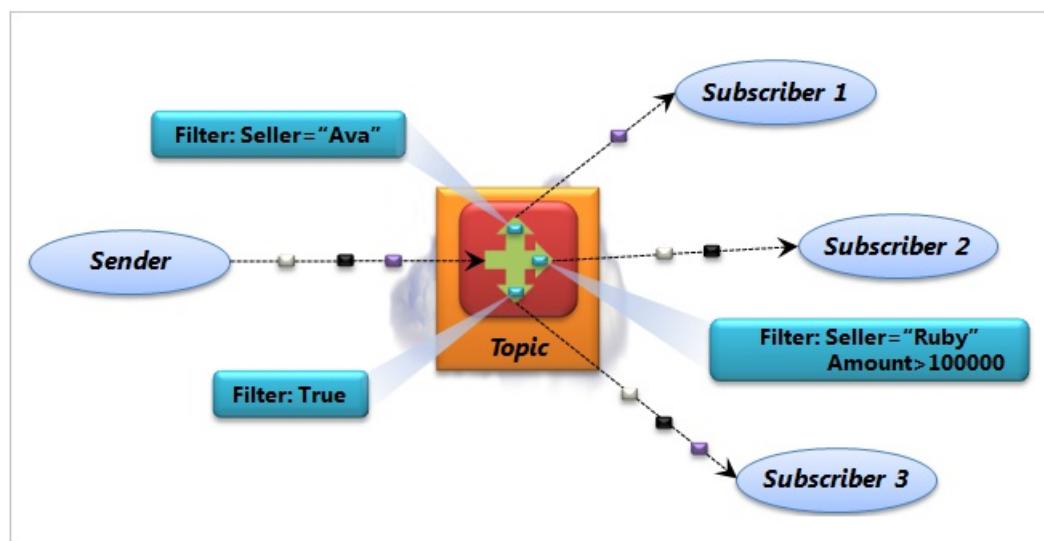


Figure 3: Based on the filter a subscribing application specifies, it can receive some or all the messages sent to a Service Bus topic.

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 difference is that topics enable each receiving application to create its own *subscription* and optionally define a *filter*. A subscriber then sees 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 large 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 connection? 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:

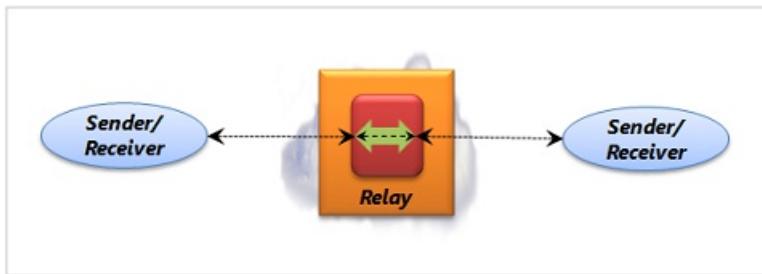


Figure 4: Service Bus relay provides synchronous, two-way communication between applications.

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 computer on which each application is running 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 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 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 communication 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 architecture

12/21/2017 • 2 min to read • [Edit Online](#)

This article describes the message processing architecture of [Azure Service Bus](#).

Service Bus scale units

Service Bus is organized by *scale units*. A scale unit is a unit of deployment and contains all components required to run the service. Each region deploys one or more Service Bus scale units.

A Service Bus namespace is mapped to a scale unit. The scale unit handles all types of Service Bus entities (queues, topics, subscriptions). A Service Bus scale unit consists of the following components:

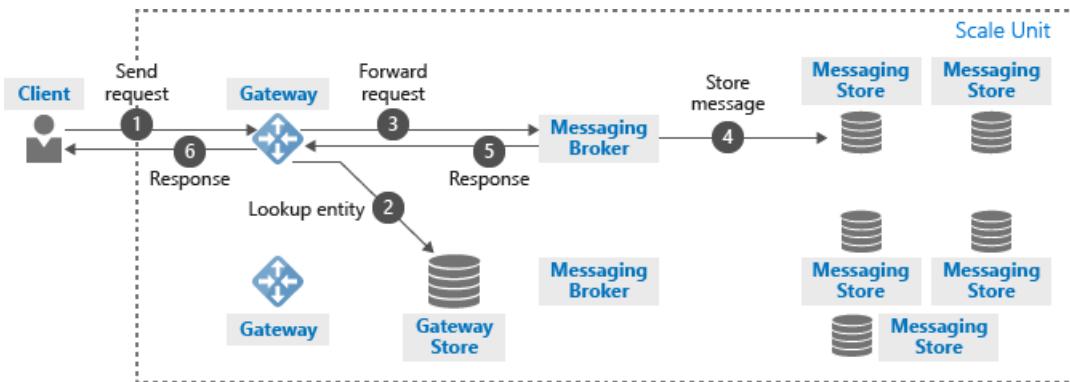
- **A set of gateway nodes.** Gateway nodes authenticate incoming requests. Each gateway node has a public IP address.
- **A set of messaging broker nodes.** Messaging broker nodes process requests concerning messaging entities.
- **One gateway store.** The gateway store holds the data for every entity that is defined in this scale unit. The gateway store is implemented on top of a SQL Database instance.
- **Multiple messaging stores.** Messaging stores hold the messages of all queues, topics and subscriptions that are defined in this scale unit. It also contains all subscription data. Unless [partitioning messaging entities](#) is enabled, a queue or topic is mapped to one messaging store. Subscriptions are stored in the same messaging store as their parent topic. Except for Service Bus [Premium Messaging](#), the messaging stores are implemented on top of [SQL Database](#) instances.

Containers

Each messaging entity is assigned a specific container. A container is a logical construct that uses one messaging store to store all relevant data for this container. Each container is assigned to a messaging broker node. Typically, there are more containers than messaging broker nodes. Therefore, each messaging broker node loads multiple containers. The distribution of containers to a messaging broker node is organized such that all messaging broker nodes are equally loaded. If the load pattern changes (for example, one of the containers gets very busy), or if a messaging broker node becomes temporarily unavailable, the containers are redistributed among the messaging broker nodes.

Processing of incoming messaging requests

When a client sends a request to Service Bus, the Azure load balancer routes it to any of the gateway nodes. The gateway node authorizes the request. If the request concerns a messaging entity (queue, topic, subscription), the gateway node looks up the entity in the gateway store and determines in which messaging store the entity is located. It then looks up which messaging broker node is currently servicing this container, and sends the request to that messaging broker node. The messaging broker node processes the request and updates the entity state in the container store. The messaging broker node then sends the response back to the gateway node, which sends an appropriate response back to the client that issued the original request.



Next steps

Now that you've read an overview of Service Bus architecture, visit the following links for more information:

- [Service Bus messaging overview](#)
- [Service Bus fundamentals](#)
- [Get started with Service Bus queues](#)

Service Bus FAQ

1/18/2018 • 5 min to read • [Edit Online](#)

This article discusses some frequently asked questions about Microsoft Azure Service Bus. You can also visit the [Azure Support FAQs](#) 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 a namespace is necessary to use Service Bus and is 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 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 partitioned 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?

See [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. Consider this limitation when you provision your entities, as these properties cannot be modified without creating a new replacement entity.

- Partitioning
- Sessions
- Duplicate detection

- Express entity

Pricing

This section answers some frequently asked questions about the Service Bus pricing structure.

The [Service Bus pricing and billing](#) article explains the billing meters in Service Bus. For specific information about Service Bus pricing options, see [Service Bus pricing details](#).

You can also visit the [Azure Support FAQs](#) for general Azure pricing information.

How do you charge for Service Bus?

For complete information about Service Bus pricing, 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 the [Service Bus 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, you can contact customer service at any time so that we can understand your needs and adjust these limits appropriately. For Service Bus, the aggregate usage quota 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 provide e-mail notification and make multiple attempts to contact a customer before taking any action.

Customers exceeding these quotas are still 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. You can find more details about these quotas in the [Service Bus quotas overview](#).

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?

You can move a namespace from one Azure subscription to another, using either the [Azure portal](#) or PowerShell

commands. In order to execute the operation, the namespace must already be active. The user executing the commands must be an administrator on both the source and target subscriptions.

Portal

To use the Azure portal to migrate Service Bus namespaces to another subscription, follow the directions [here](#).

PowerShell

The following sequence of PowerShell 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-aaaaaaaaaaa'
$res = Find-AzureRmResource -ResourceNameContains mynamespace -ResourceType 'Microsoft.ServiceBus/namespaces'
Move-AzureRmResource -DestinationResourceGroupName 'targetRG' -DestinationSubscriptionId 'ffffffff-ffff-ffff-ffff-ffffffff'
-ResourceId $res.ResourceId
```

Next steps

To learn more about Service Bus, see the following articles:

- [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

2/23/2018 • 2 min to read • [Edit Online](#)

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

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

Create a namespace in the Azure portal

To begin using Service Bus messaging entities in Azure, you must first create a namespace with a name that is unique across Azure. 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 **+ Create a resource**, 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

* Name
sbnstest1 .servicebus.windows.net

* Pricing tier
Standard >

* Subscription
Prototype3

* Resource group ⓘ
 Create new Use existing
MyRG

* Location
East US

Pin to dashboard

Create Automation options

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

Creating a new namespace automatically generates an initial Shared Access Signature (SAS) rule with an associated pair of primary and secondary keys that each grant full control over all aspects of the namespace. See [Service Bus authentication and authorization](#) for information about how to create further rules with more constrained rights for regular senders and receivers. To copy the initial rule, follow these steps:

1. Click **All resources**, then click the newly created namespace name.
2. In the namespace window, click **Shared access policies**.
3. In the **Shared access policies** screen, click **RootManageSharedAccessKey**.

The screenshot shows the Azure portal interface for a Service Bus namespace named 'sbnstest1'. The left sidebar contains navigation links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (which is expanded), Shared access policies (highlighted with a red box), Scale, Properties, Locks, Automation script, Entities (Queues and Topics), Monitoring (Diagnostics logs and Metrics (preview)), and Support + TROUBLESHOOTING (New support request). The main content area displays the 'Shared access policies' blade. It includes a search bar, an 'Add' button, and a table with two columns: POLICY and CLAIMS. A policy named 'RootManageSharedAccessKey' is listed, showing claims 'Manage, Send, Listen'. Both the policy name and the claims row are highlighted with a red box.

4. In the **Policy: RootManageSharedAccessKey** window, 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.

The screenshot shows the 'SAS Policy: RootManageSharedAccessKey' configuration dialog. At the top are Save, Discard, Delete, and More buttons. Below are checkboxes for Manage, Send, and Listen, all of which are checked. Under 'Primary Key', there is a text input field containing 'Primary key here' with a copy icon to its right. Under 'Secondary Key', there is a text input field containing 'Secondary key here' with a copy icon to its right. Under 'Primary Connection String', there is a text input field containing 'Endpoint=sb://sbnstest1.servicebus.windows.n...' with a copy icon to its right. The 'copy' icon for the primary connection string is highlighted with a red box.

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 Service Bus messaging.

Get started with Service Bus queues

12/11/2017 • 10 min to read • [Edit Online](#)

This tutorial covers the following steps:

1. Create a Service Bus namespace, using the Azure portal.
2. Create a Service Bus queue, using the Azure portal.
3. Write a .NET Core console application to send a set of messages to the queue.
4. Write a .NET Core console application to receive those messages from the queue.

Prerequisites

1. [Visual Studio 2017 Update 3 \(version 15.3, 26730.01\)](#) or later.
2. [.NET Core SDK](#), version 2.0 or later.
3. 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

NOTE

You can also create a Service Bus namespace and messaging entities using [PowerShell](#). For more information, see [Use PowerShell to manage Service Bus resources](#).

If you have already created a Service Bus Messaging namespace, jump to the [Create a queue using the Azure portal](#) section.

To begin using Service Bus messaging entities in Azure, you must first create a namespace with a name that is unique across Azure. 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 **+ Create a resource**, 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

* Name
sbnstest1 .servicebus.windows.net

* Pricing tier
Standard >

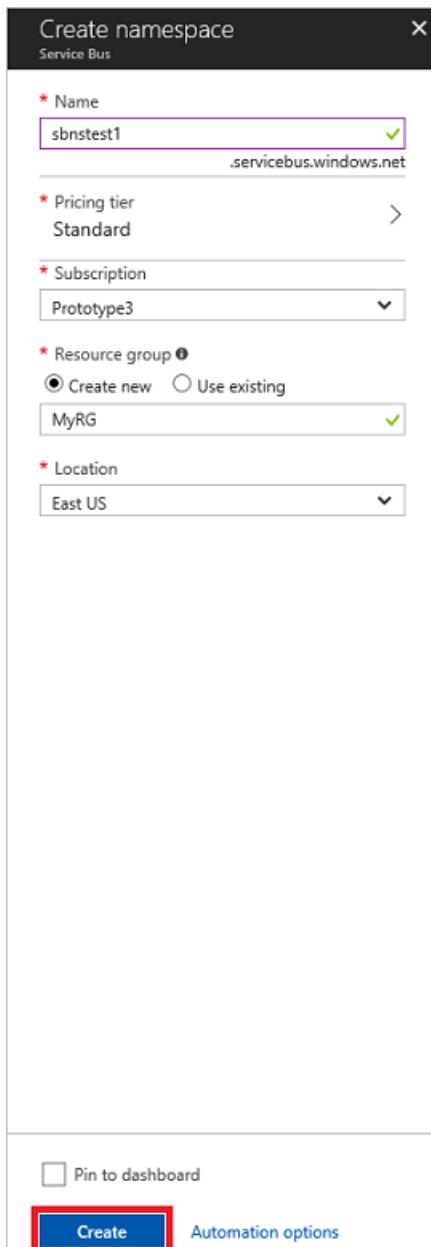
* Subscription
Prototype3

* Resource group ⓘ
 Create new Use existing
MyRG

* Location
East US

Pin to dashboard

Create Automation options



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

Creating a new namespace automatically generates an initial Shared Access Signature (SAS) rule with an associated pair of primary and secondary keys that each grant full control over all aspects of the namespace. See [Service Bus authentication and authorization](#) for information about how to create further rules with more constrained rights for regular senders and receivers. To copy the initial rule, follow these steps:

1. Click **All resources**, then click the newly created namespace name.
2. In the namespace window, click **Shared access policies**.
3. In the **Shared access policies** screen, click **RootManageSharedAccessKey**.

The screenshot shows the Azure portal interface for a Service Bus entity named 'sbnstest1'. The left sidebar contains navigation links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, SETTINGS (with 'Shared access policies' selected and highlighted with a red box), Properties, Locks, Automation script, ENTITIES (Queues, Topics), MONITORING (Diagnostics logs, Metrics (preview)), and SUPPORT + TROUBLESHOOTING (New support request). The main pane displays the 'Shared access policies' configuration for the selected policy 'RootManageSharedAccessKey'. The policy details show the following:

POLICY	CLAIMS
RootManageSharedAccessKey	Manage, Send, Listen

4. In the **Policy: RootManageSharedAccessKey** window, 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.

The screenshot shows the 'SAS Policy: RootManageSharedAccessKey' configuration window. The top bar includes Save, Discard, Delete, and More options. The policy settings are as follows:

- Manage:
- Send:
- Listen:

Below these settings are four fields with copy icons (blue folder icons with a white arrow):

- Primary Key: 
- Secondary Key: 
- Primary Connection String: 
- Secondary Connection String: 

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 have already created a Service Bus queue, 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 **All services**).
3. Click the namespace in which you would like to create the queue. In this case, it is **sbnstest1**.

The screenshot shows the Azure Service Bus namespace 'sbnstest1' overview page. The left sidebar contains several sections: Overview (highlighted in blue), Activity log, Access control (IAM), Tags, Diagnose and solve problems, SETTINGS (Shared access policies, Scale, Properties, Locks, Automation script), ENTITIES (Queues, Topics), MONITORING (Diagnostics logs, Metrics (preview)), and SUPPORT + TROUBLESHOOTING (New support request). The 'Queues' link is highlighted with a red box.

4. In the namespace window, click **Queues**, then in the **Queues** window, click **+ Queue**.

sbnstest1 - Queues
Service Bus

Search (Ctrl+ /) <> **+ Queue** Refresh

Overview Activity log Access control (IAM) Tags Diagnose and solve problems

SETTINGS Shared access policies Scale Properties Locks Automation script

ENTITIES **Queues** Topics

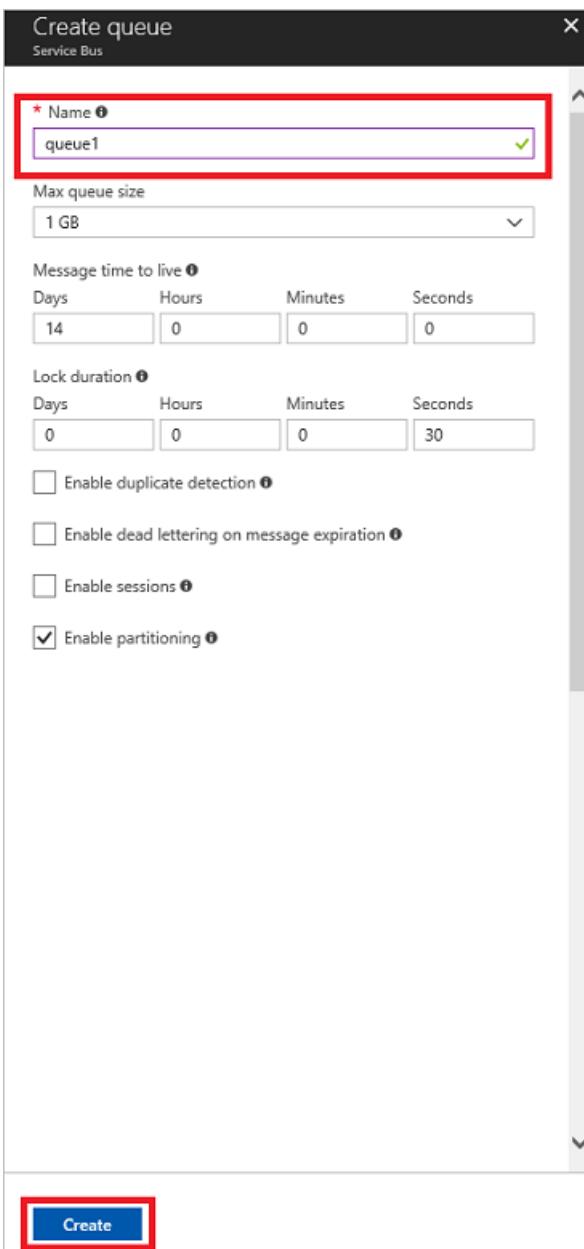
MONITORING Diagnostics logs Metrics (preview)

SUPPORT + TROUBLESHOOTING New support request

Search to filter items...

NAME	STATUS	MAX SIZE
no queues yet.		

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



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

3. Send messages to the queue

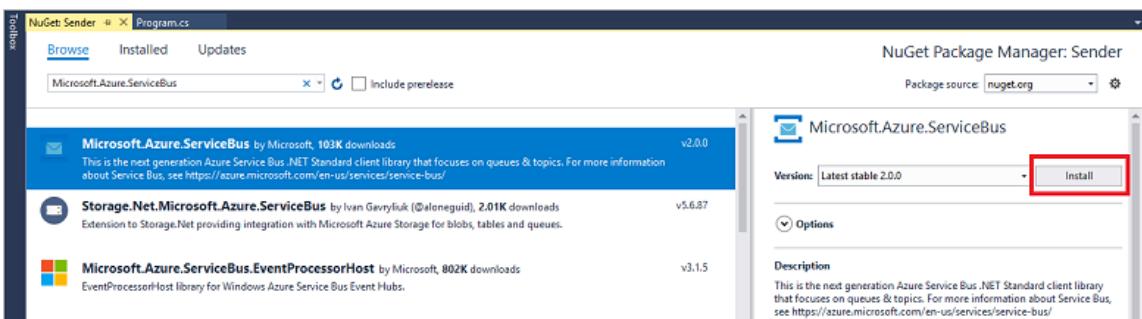
To send messages to the queue, write a C# console application using Visual Studio.

Create a console application

Launch Visual Studio and create a new **Console App (.NET Core)** project.

Add the Service Bus NuGet package

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



Write code to send messages to the queue

1. In Program.cs, add the following `using` statements at the top of the namespace definition, before the class declaration:

```
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Azure.ServiceBus;
```

2. Within the `Program` class, declare the following variables. Set the `ServiceBusConnectionString` variable to the connection string that you obtained when creating the namespace, and set `QueueName` to the name that you used when creating the queue:

```
const string ServiceBusConnectionString = "<your_connection_string>";
const string QueueName = "<your_queue_name>";
static IQueueClient queueClient;
```

3. Replace the default contents of `Main()` with the following line of code:

```
MainAsync().GetAwaiter().GetResult();
```

4. Directly after `Main()`, add the following asynchronous `MainAsync()` method that calls the send messages method:

```
static async Task MainAsync()
{
    const int numberOfMessages = 10;
    queueClient = new QueueClient(ServiceBusConnectionString, QueueName);

    Console.WriteLine("=====");
    Console.WriteLine("Press ENTER key to exit after sending all the messages.");
    Console.WriteLine("=====");

    // Send messages.
    await SendMessagesAsync(numberOfMessages);

    Console.ReadKey();

    await queueClient.CloseAsync();
}
```

5. Directly after the `MainAsync()` method, add the following `SendMessagesAsync()` method that performs the work of sending the number of messages specified by `numberOfMessagesToSend` (currently set to 10):

```
static async Task SendMessagesAsync(int numberOfMessagesToSend)
{
    try
    {
        for (var i = 0; i < numberOfMessagesToSend; i++)
        {
            // Create a new message to send to the queue.
            string messageBody = $"Message {i}";
            var message = new Message(Encoding.UTF8.GetBytes(messageBody));

            // Write the body of the message to the console.
            Console.WriteLine($"Sending message: {messageBody}");

            // Send the message to the queue.
            await queueClient.SendAsync(message);
        }
    }
    catch (Exception exception)
    {
        Console.WriteLine($"{DateTime.Now} :: Exception: {exception.Message}");
    }
}
```

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

```

namespace CoreSenderApp
{
    using System;
    using System.Text;
    using System.Threading;
    using System.Threading.Tasks;
    using Microsoft.Azure.ServiceBus;

    class Program
    {
        // Connection String for the namespace can be obtained from the Azure portal under the
        // 'Shared Access policies' section.
        const string ServiceBusConnectionString = "<your_connection_string>";
        const string QueueName = "<your_queue_name>";
        static IQueueClient queueClient;

        static void Main(string[] args)
        {
            MainAsync().GetAwaiter().GetResult();
        }

        static async Task MainAsync()
        {
            const int numberOfMessages = 10;
            queueClient = new QueueClient(ServiceBusConnectionString, QueueName);

            Console.WriteLine("=====");
            Console.WriteLine("Press ENTER key to exit after receiving all the messages.");
            Console.WriteLine("=====");

            // Send Messages
            await SendMessagesAsync(numberOfMessages);

            Console.ReadKey();

            await queueClient.CloseAsync();
        }

        static async Task SendMessagesAsync(int numberOfMessagesToSend)
        {
            try
            {
                for (var i = 0; i < numberOfMessagesToSend; i++)
                {
                    // Create a new message to send to the queue
                    string messageBody = $"Message {i}";
                    var message = new Message(Encoding.UTF8.GetBytes(messageBody));

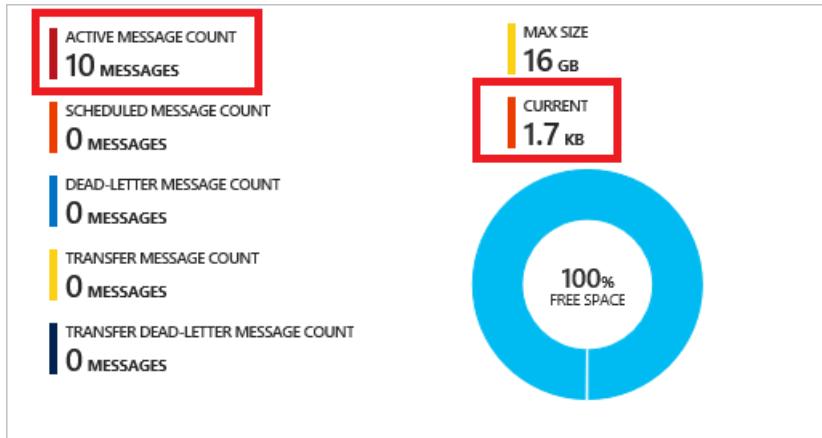
                    // Write the body of the message to the console
                    Console.WriteLine($"Sending message: {messageBody}");

                    // Send the message to the queue
                    await queueClient.SendAsync(message);
                }
            }
            catch (Exception exception)
            {
                Console.WriteLine($"{DateTime.Now} :: Exception: {exception.Message}");
            }
        }
    }
}

```

7. Run the program, and check the Azure portal: click the name of your queue in the namespace **Overview** window. The queue **Essentials** screen is displayed. Notice that the **Active Message Count** value for the queue is now **10**. Each time you run the sender application without retrieving the

messages (as described in the next section), this value increases by 10. Also note that the current size of the queue increments the **Current** value in the **Essentials** window each time the app adds messages to the queue.



4. Receive messages from the queue

To receive the messages you just sent, create another .NET Core console application and install the **Microsoft.Azure.ServiceBus** NuGet package, similar to the previous sender application.

Write code to receive messages from the queue

1. In Program.cs, add the following `using` statements at the top of the namespace definition, before the class declaration:

```
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Azure.ServiceBus;
```

2. Within the `Program` class, declare the following variables. Set the `ServiceBusConnectionString` variable to the connection string that you obtained when creating the namespace, and set `QueueName` to the name that you used when creating the queue:

```
const string ServiceBusConnectionString = "<your_connection_string>";
const string QueueName = "<your_queue_name>";
static IQueueClient queueClient;
```

3. Replace the default contents of `Main()` with the following line of code:

```
MainAsync().GetAwaiter().GetResult();
```

4. Directly after `Main()`, add the following asynchronous `MainAsync()` method that calls the `RegisterOnMessageHandlerAndReceiveMessages()` method:

```

static async Task MainAsync()
{
    queueClient = new QueueClient(ServiceBusConnectionString, QueueName);

    Console.WriteLine("====");
    Console.WriteLine("Press ENTER key to exit after receiving all the messages.");
    Console.WriteLine("====");

    // Register the queue message handler and receive messages in a loop
    RegisterOnMessageHandlerAndReceiveMessages();

    Console.ReadKey();

    await queueClient.CloseAsync();
}

```

5. Directly after the `MainAsync()` method, add the following method that registers the message handler and receives the messages sent by the sender application:

```

static void RegisterOnMessageHandlerAndReceiveMessages()
{
    // Configure the message handler options in terms of exception handling, number of concurrent
    // messages to deliver, etc.
    var messageHandlerOptions = new MessageHandlerOptions(ExceptionReceivedHandler)
    {
        // Maximum number of concurrent calls to the callback ProcessMessagesAsync(), set to 1 for
        // simplicity.
        // Set it according to how many messages the application wants to process in parallel.
        MaxConcurrentCalls = 1,

        // Indicates whether the message pump should automatically complete the messages after
        // returning from user callback.
        // False below indicates the complete operation is handled by the user callback as in
        // ProcessMessagesAsync().
        AutoComplete = false
    };

    // Register the function that processes messages.
    queueClient.RegisterMessageHandler(ProcessMessagesAsync, messageHandlerOptions);
}

```

6. Directly after the previous method, add the following `ProcessMessagesAsync()` method to process the received messages:

```

static async Task ProcessMessagesAsync(Message message, CancellationToken token)
{
    // Process the message.
    Console.WriteLine($"Received message: SequenceNumber:{message.SystemProperties.SequenceNumber}
Body:{Encoding.UTF8.GetString(message.Body)}");

    // Complete the message so that it is not received again.
    // This can be done only if the queue Client is created in ReceiveMode.PeekLock mode (which is
    // the default).
    await queueClient.CompleteAsync(message.SystemProperties.LockToken);

    // Note: Use the cancellationToken passed as necessary to determine if the queueClient has
    // already been closed.
    // If queueClient has already been closed, you can choose to not call CompleteAsync() or
    // AbandonAsync() etc.
    // to avoid unnecessary exceptions.
}

```

7. Finally, add the following method to handle any exceptions that might occur:

```
// Use this handler to examine the exceptions received on the message pump.
static Task ExceptionReceivedHandler(ExceptionReceivedEventArgs exceptionReceivedEventArgs)
{
    Console.WriteLine($"Message handler encountered an exception
{exceptionReceivedEventArgs.Exception}.");
    var context = exceptionReceivedEventArgs.ExceptionReceivedContext;
    Console.WriteLine("Exception context for troubleshooting:");
    Console.WriteLine($"- Endpoint: {context.Endpoint}");
    Console.WriteLine($"- Entity Path: {context.EntityPath}");
    Console.WriteLine($"- Executing Action: {context.Action}");
    return Task.CompletedTask;
}
```

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

```
namespace CoreReceiverApp
{
    using System;
    using System.Text;
    using System.Threading;
    using System.Threading.Tasks;
    using Microsoft.Azure.ServiceBus;

    class Program
    {
        // Connection String for the namespace can be obtained from the Azure portal under the
        // 'Shared Access policies' section.
        const string ServiceBusConnectionString = "<your_connection_string>";
        const string QueueName = "<your_queue_name>";
        static IQueueClient queueClient;

        static void Main(string[] args)
        {
            MainAsync().GetAwaiter().GetResult();
        }

        static async Task MainAsync()
        {
            queueClient = new QueueClient(ServiceBusConnectionString, QueueName);

            Console.WriteLine("=====");
            Console.WriteLine("Press ENTER key to exit after receiving all the messages.");
            Console.WriteLine("=====");

            // Register QueueClient's MessageHandler and receive messages in a loop
            RegisterOnMessageHandlerAndReceiveMessages();

            Console.ReadKey();

            await queueClient.CloseAsync();
        }

        static void RegisterOnMessageHandlerAndReceiveMessages()
        {
            // Configure the MessageHandler Options in terms of exception handling, number of
            concurrent messages to deliver etc.
            var messageHandlerOptions = new MessageHandlerOptions(ExceptionReceivedHandler)
            {
                // Maximum number of Concurrent calls to the callback `ProcessMessagesAsync`, set
                to 1 for simplicity.
                // Set it according to how many messages the application wants to process in
                parallel.
                MaxConcurrentCalls = 1,
```

```

        // Indicates whether MessagePump should automatically complete the messages after
        // returning from User Callback.
        // False below indicates the Complete will be handled by the User Callback as in
`ProcessMessagesAsync` below.
        AutoComplete = false
    };

    // Register the function that will process messages
    queueClient.RegisterMessageHandler(ProcessMessagesAsync, messageHandlerOptions);
}

static async Task ProcessMessagesAsync(Message message, CancellationToken token)
{
    // Process the message
    Console.WriteLine($"Received message: SequenceNumber:
{message.SystemProperties.SequenceNumber} Body:{Encoding.UTF8.GetString(message.Body)}");

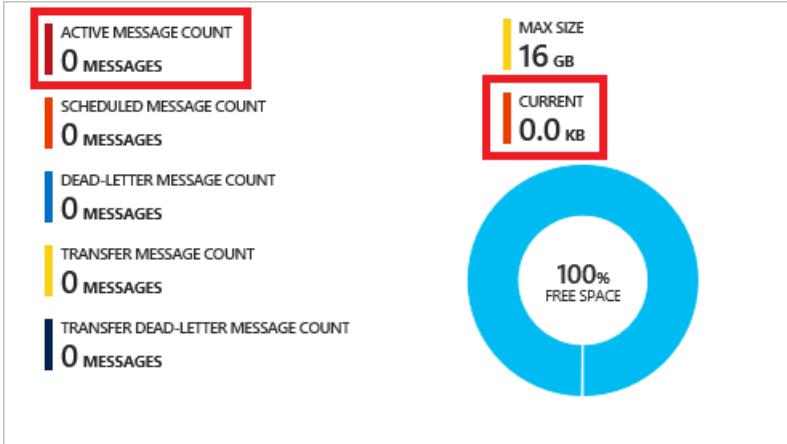
    // Complete the message so that it is not received again.
    // This can be done only if the queueClient is created in ReceiveMode.PeekLock mode
    // (which is default).
    await queueClient.CompleteAsync(message.SystemProperties.LockToken);

    // Note: Use the cancellationToken passed as necessary to determine if the queueClient
    has already been closed.
    // If queueClient has already been Closed, you may chose to not call CompleteAsync() or
    AbandonAsync() etc. calls
    // to avoid unnecessary exceptions.
}

static Task ExceptionReceivedHandler(ExceptionReceivedEventArgs exceptionReceivedEventArgs)
{
    Console.WriteLine($"Message handler encountered an exception
{exceptionReceivedEventArgs.Exception}.");
    var context = exceptionReceivedEventArgs.ExceptionReceivedContext;
    Console.WriteLine("Exception context for troubleshooting:");
    Console.WriteLine($"- Endpoint: {context.Endpoint}");
    Console.WriteLine($"- Entity Path: {context.EntityPath}");
    Console.WriteLine($"- Executing Action: {context.Action}");
    return Task.CompletedTask;
}
}
}

```

- Run the program, and check the portal again. Notice that the **Active Message Count** and **Current** values are now **0**.



Congratulations! You have now created a queue, sent a set of messages to that queue, and received those messages from the same queue.

Next steps

Check out our [GitHub repository with samples](#) that demonstrate some of the more advanced features of Service Bus messaging.

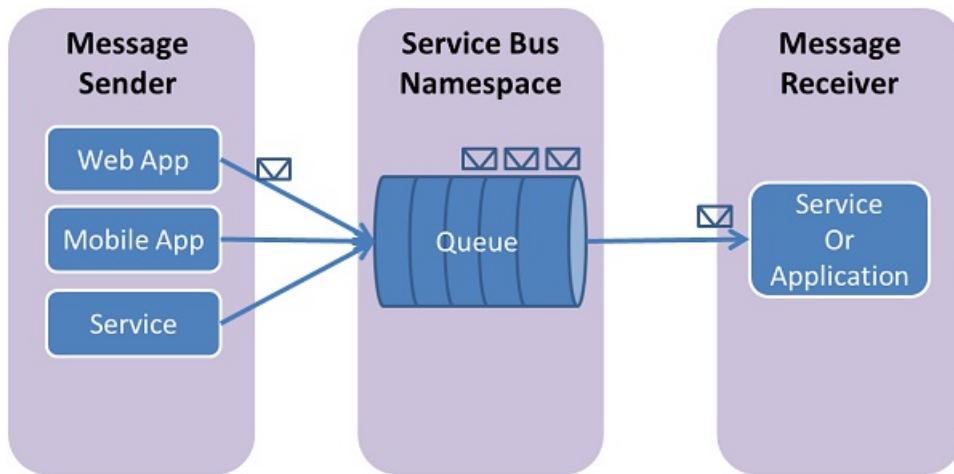
How to use Service Bus queues with Java

8/10/2017 • 9 min to read • [Edit Online](#)

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.

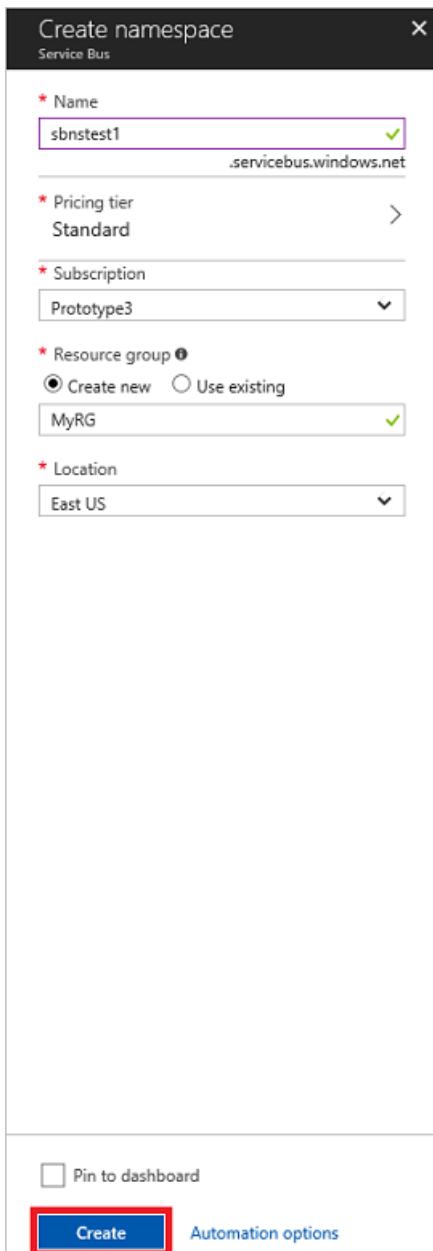
Using queues enables you to scale your applications more easily, and enable more resiliency to your architecture.

To begin using Service Bus messaging entities in Azure, you must first create a namespace with a name that is unique across Azure. 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 **+ Create a resource**, 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.



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

Creating a new namespace automatically generates an initial Shared Access Signature (SAS) rule with an associated pair of primary and secondary keys that each grant full control over all aspects of the namespace. See [Service Bus authentication and authorization](#) for information about how to create further rules with more constrained rights for regular senders and receivers. To copy the initial rule, follow these steps:

1. Click **All resources**, then click the newly created namespace name.
2. In the namespace window, click **Shared access policies**.
3. In the **Shared access policies** screen, click **RootManageSharedAccessKey**.

The screenshot shows the Azure portal interface for a Service Bus entity named 'sbnstest1'. The left sidebar contains various navigation options like Overview, Activity log, Access control (IAM), Tags, and Diagnose and solve problems. Under the 'SETTINGS' section, 'Shared access policies' is selected and highlighted with a red box. The main content area displays a table with two columns: 'POLICY' and 'CLAIMS'. A single row is present, showing 'RootManageSharedAccessKey' in the 'POLICY' column and 'Manage, Send, Listen' in the 'CLAIMS' column, also highlighted with a red box.

4. In the **Policy: RootManageSharedAccessKey** window, 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.

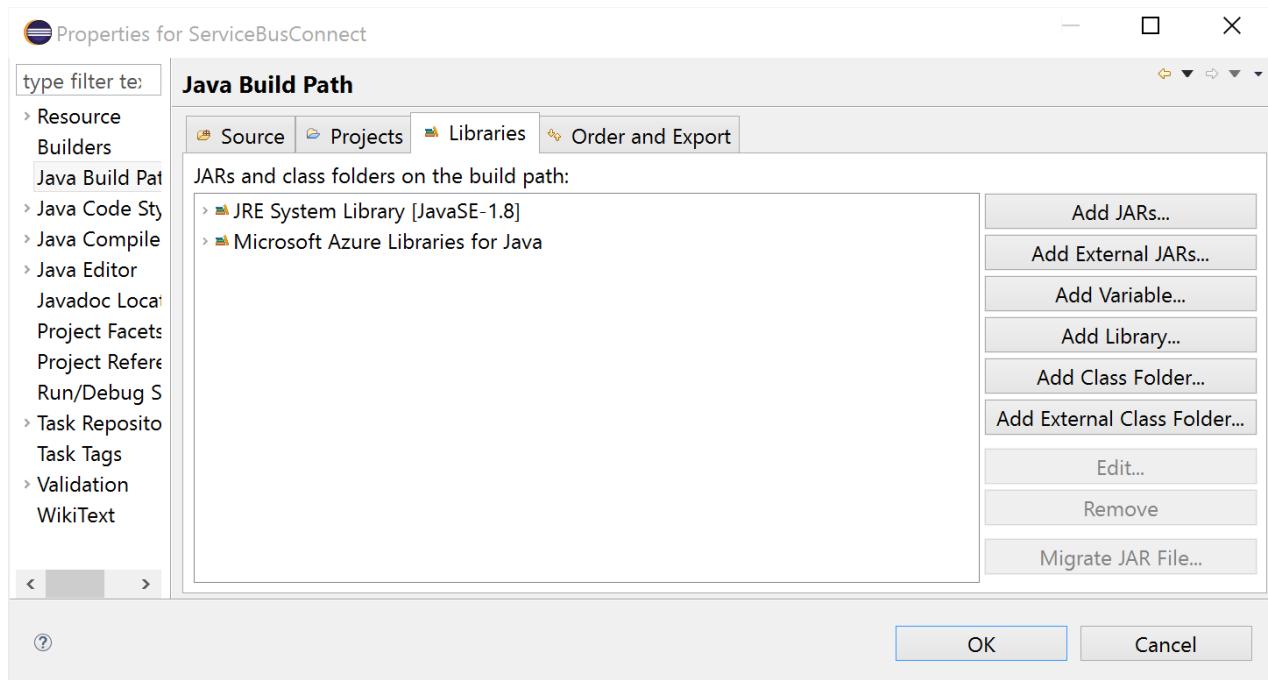
This screenshot shows the 'SAS Policy: RootManageSharedAccessKey' configuration dialog. At the top, there are buttons for Save, Discard, Delete, and More. Below these are three checked checkboxes: 'Manage', 'Send', and 'Listen'. Underneath these are four input fields:

- Primary Key:** A text input field containing 'Primary key here' with a copy icon to its right.
- Secondary Key:** A text input field containing 'Secondary key here' with a copy icon to its right.
- Primary Connection String:** A text input field containing 'Endpoint=sb://sbnstest1.servicebus.windows.n...' with a copy icon to its right, which is highlighted with a red box.
- Secondary Connection String:** A text input field containing 'Endpoint=sb://sbnstest1.servicebus.windows.n...' with a copy icon to its right.

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:



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 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 **deleteMessage** request is issued, then 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 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 **get messageId** 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, see the [Java Developer Center](#).

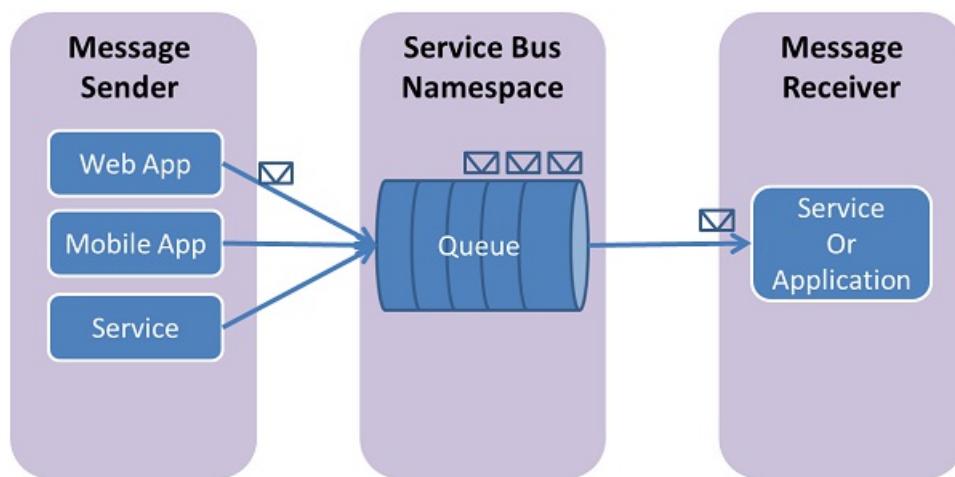
How to use Service Bus queues with Node.js

9/19/2017 • 9 min to read • [Edit Online](#)

This article describes how to use Service Bus queues with 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.

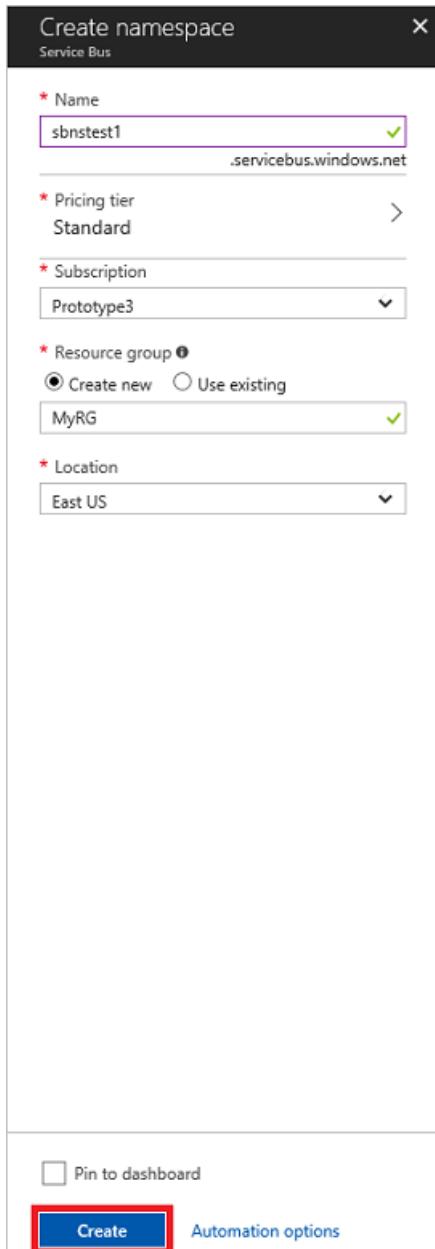
Using queues enables you to scale your applications more easily, and enable more resiliency to your architecture.

To begin using Service Bus messaging entities in Azure, you must first create a namespace with a name that is unique across Azure. 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 + **Create a resource**, 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.



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

Creating a new namespace automatically generates an initial Shared Access Signature (SAS) rule with an associated pair of primary and secondary keys that each grant full control over all aspects of the namespace. See [Service Bus authentication and authorization](#) for information about how to create further rules with more constrained rights for regular senders and receivers. To copy the initial rule, follow these steps:

1. Click **All resources**, then click the newly created namespace name.
2. In the namespace window, click **Shared access policies**.
3. In the **Shared access policies** screen, click **RootManageSharedAccessKey**.

POLICY	CLAIMS
RootManageSharedAccessKey	Manage, Send, Listen

4. In the **Policy: RootManageSharedAccessKey** window, 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.

SAS Policy: RootManageSharedAccessKey

Save Discard Delete More

Manage
Send
Listen

Primary Key: Primary key here

Secondary Key: Secondary key here

Primary Connection String: Endpoint=sb://sbnstest1.servicebus.windows.n...

Secondary Connection String: Endpoint=sb://sbnstest1.servicebus.windows.n...

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 variable **AZURE_SERVICEBUS_CONNECTION_STRING** to obtain information required to connect to Service Bus. If this environment variable is 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 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
  }
});
```

The `createServiceBusService` method 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 code 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 marks the message as being consumed and removes 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](#)

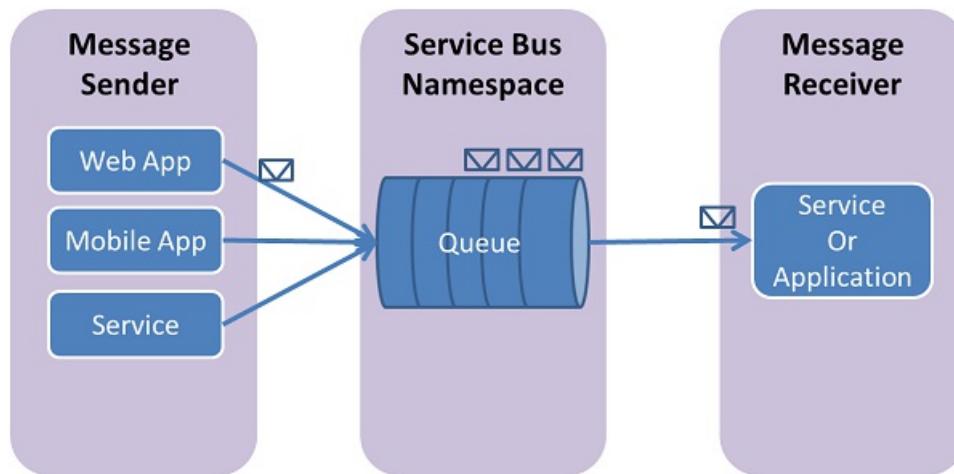
How to use Service Bus queues with PHP

8/10/2017 • 8 min to read • [Edit Online](#)

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. Create a file named **composer.json** in the root of your project and add the following code to it:

```
{  
    "require": {  
        "microsoft/azure-storage": "*"  
    }  
}
```

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

```
php composer.phar install
```

Alternatively go to the [Azure Storage PHP Client Library](#) on GitHub to clone the source code.

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];SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=[Primary 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 is passed directly.

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

use WindowsAzure\Common\ServicesBuilder;

$connectionString = "Endpoint=[yourEndpoint];SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=[Primary Key]";

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

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:
    // https://docs.microsoft.com/rest/api/storageservices/Common-REST-API-Error-Codes
    $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.

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:
    // https://docs.microsoft.com/rest/api/storageservices/Common-REST-API-Error-Codes
    $code = $e->getCode();
    $error_message = $e->getMessage();
    echo $code.": ".$error_message."  
";
}
```

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.

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:
    // https://docs.microsoft.com/rest/api/storageservices/Common-REST-API-Error-Codes
    $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 `get messageId` 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](#).

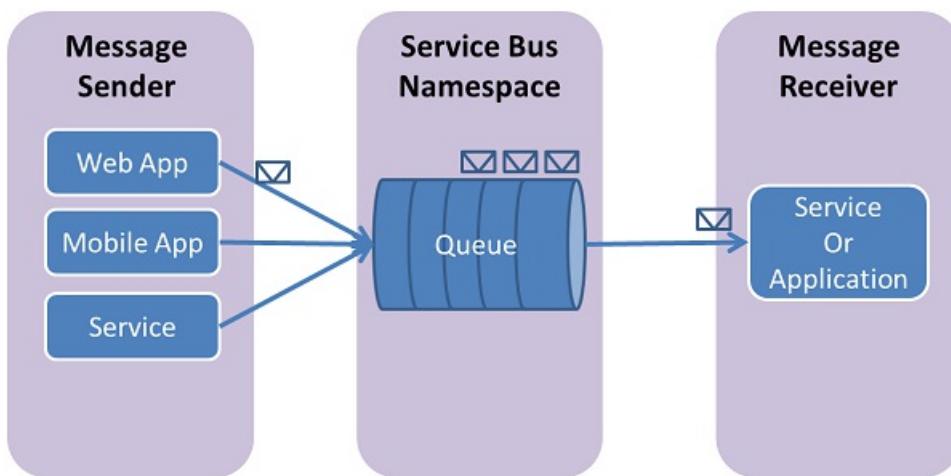
How to use Service Bus queues with Python

4/30/2018 • 7 min to read • [Edit Online](#)

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.

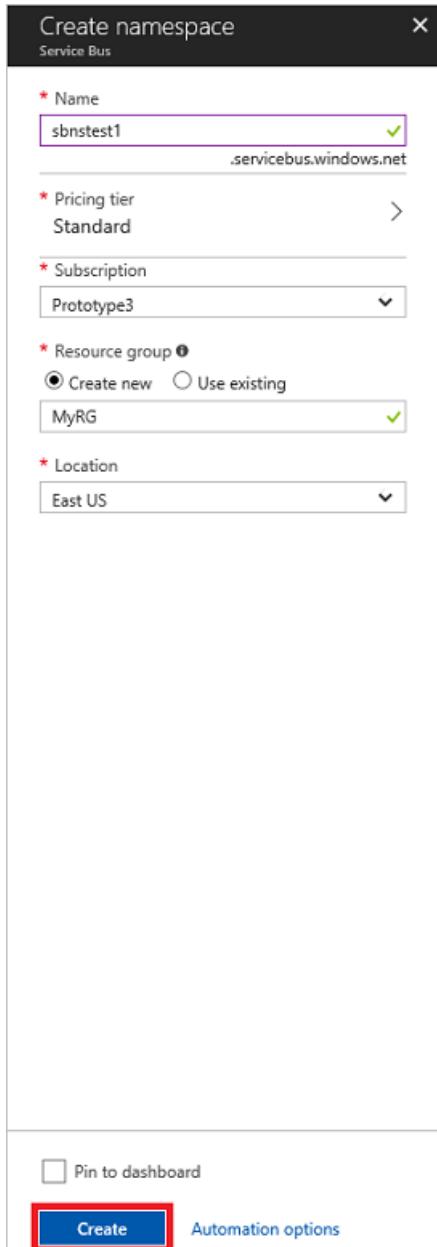
To begin using Service Bus messaging entities in Azure, you must first create a namespace with a name that is unique across Azure. A namespace provides a scoping container for addressing Service Bus resources within your application.

To create a namespace:

1. Log on to the [Azure portal](#).
2. In the left navigation pane of the portal, click **+ Create a resource**, 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.



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

Creating a new namespace automatically generates an initial Shared Access Signature (SAS) rule with an associated pair of primary and secondary keys that each grant full control over all aspects of the namespace. See [Service Bus authentication and authorization](#) for information about how to create further rules with more constrained rights for regular senders and receivers. To copy the initial rule, follow these steps:

1. Click **All resources**, then click the newly created namespace name.
2. In the namespace window, click **Shared access policies**.
3. In the **Shared access policies** screen, click **RootManageSharedAccessKey**.

The screenshot shows the Azure Service Bus Shared access policies page. On the left, there's a navigation menu with sections like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, SETTINGS (with Shared access policies highlighted by a red box), ENTITIES (Queues, Topics), MONITORING (Diagnostics logs, Metrics (preview)), and SUPPORT + TROUBLESHOOTING (New support request). In the main area, a search bar at the top has '+ Add' and a magnifying glass icon. Below it, a table lists a single policy: 'RootManageSharedAccessKey' under the 'POLICY' column and 'Manage, Send, Listen' under the 'CLAIMS' column. The entire row for this policy is also highlighted with a red box.

4. In the **Policy: RootManageSharedAccessKey** window, 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.

This screenshot shows the 'SAS Policy: RootManageSharedAccessKey' configuration window. At the top, there are buttons for Save, Discard, Delete, and More. Below that, three checkboxes are checked: 'Manage', 'Send', and 'Listen'. Underneath these are four input fields: 'Primary Key' containing 'Primary key here' with a copy icon, 'Secondary Key' containing 'Secondary key here' with a copy icon, 'Primary Connection String' containing 'Endpoint=sb://sbnstest1.servicebus.windows.n...' with a copy icon highlighted by a red box, and 'Secondary Connection String' containing 'Endpoint=sb://sbnstest1.servicebus.windows.n...' with a copy icon.

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 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')
```

The `create_queue` method 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 5 GB, 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](#)

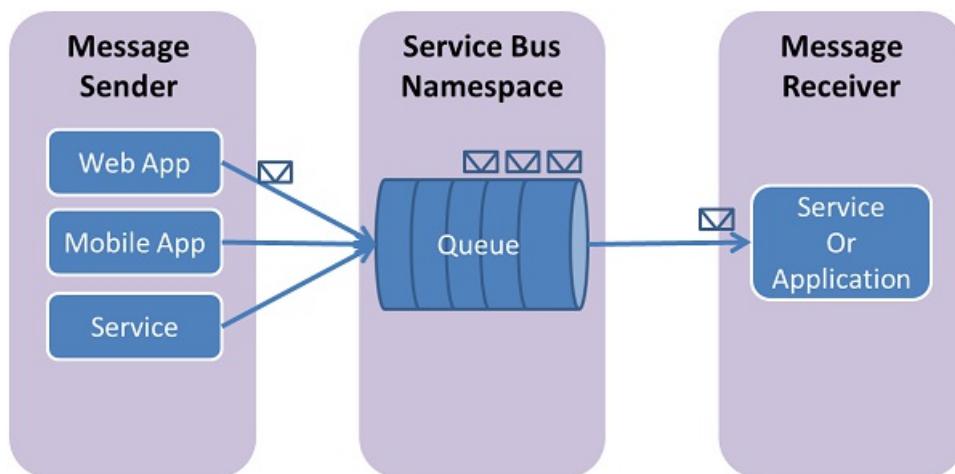
How to use Service Bus queues with Ruby

8/21/2017 • 8 min to read • [Edit Online](#)

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.

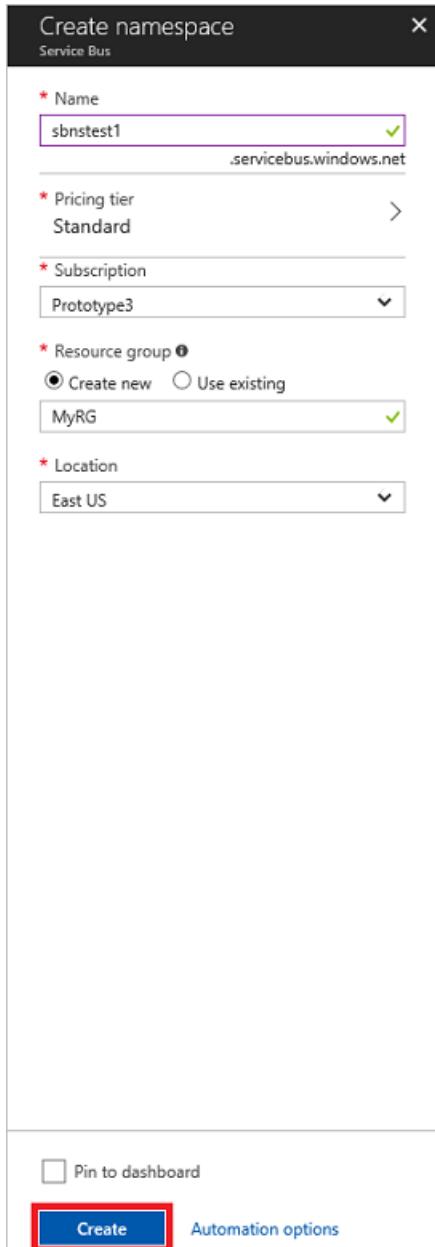
To begin using Service Bus messaging entities in Azure, you must first create a namespace with a name that is unique across Azure. A namespace provides a scoping container for addressing Service Bus resources within your application.

To create a namespace:

1. Log on to the [Azure portal](#).
2. In the left navigation pane of the portal, click **+ Create a resource**, 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.



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

Creating a new namespace automatically generates an initial Shared Access Signature (SAS) rule with an associated pair of primary and secondary keys that each grant full control over all aspects of the namespace. See [Service Bus authentication and authorization](#) for information about how to create further rules with more constrained rights for regular senders and receivers. To copy the initial rule, follow these steps:

1. Click **All resources**, then click the newly created namespace name.
2. In the namespace window, click **Shared access policies**.
3. In the **Shared access policies** screen, click **RootManageSharedAccessKey**.

The screenshot shows the Azure portal interface for a Service Bus entity named 'sbnstest1'. The left sidebar has sections for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, SETTINGS (with 'Shared access policies' highlighted by a red box), ENTITIES (Queues, Topics), MONITORING (Diagnostics logs, Metrics (preview)), and SUPPORT + TROUBLESHOOTING (New support request). The main content area shows a table for 'RootManageSharedAccessKey' with columns for POLICY and CLAIMS. The policy name is 'RootManageSharedAccessKey' and the claims are 'Manage, Send, Listen'.

4. In the **Policy: RootManageSharedAccessKey** window, 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.

The screenshot shows the 'SAS Policy: RootManageSharedAccessKey' configuration window. It includes buttons for Save, Discard, Delete, and More. Under the 'Manage' section, there are checkboxes for Manage, Send, and Listen, all of which are checked. Below this are fields for Primary Key ('Primary key here'), Secondary Key ('Secondary key here'), Primary Connection String ('Endpoint=sb://sbnstest1.servicebus.windows.n...'), and Secondary Connection String ('Endpoint=sb://sbnstest1.servicebus.windows.n...'). Each field has a copy icon (a blue square with a white arrow) to its right.

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 Azure Ruby 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

Use the following code to set the values of namespace, name of the key, key signer and host:

```
Azure.configure do |config|
  config_sb_namespace = '<your azure service bus namespace>'
  config_sb_sas_key_name = '<your azure service bus access keyname>'
  config_sb_sas_key = '<your azure service bus access key>'
end
signer = Azure::ServiceBus::Auth::SharedAccessSigner.new
sb_host = "https://#{Azure.sb_namespace}.servicebus.windows.net"
```

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::ServiceBus::ServiceBusService.new(sb_host, { signer: signer})
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 5 GB 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 are 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 has 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 call 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 is redelivered to the application when it restarts. This process 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](#)

Get started with Service Bus topics

2/16/2018 • 10 min to read • [Edit Online](#)

This tutorial covers the following steps:

1. Create a Service Bus namespace, using the Azure portal.
2. Create a Service Bus topic, using the Azure portal.
3. Create a Service Bus subscription to that topic, using the Azure portal.
4. Write a .NET Core console application to send a set of messages to the topic.
5. Write a .NET Core console application to receive those messages from the subscription.

Prerequisites

1. [Visual Studio 2017 Update 3 \(version 15.3, 26730.01\)](#) or later.
2. [NET Core SDK](#), version 2.0 or later.
3. 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

NOTE

You can also create a Service Bus namespace and messaging entities using [PowerShell](#). For more information, see [Use PowerShell to manage Service Bus resources](#).

If you have already created a Service Bus Messaging namespace, jump to the [Create a topic using the Azure portal](#) section.

To begin using Service Bus messaging entities in Azure, you must first create a namespace with a name that is unique across Azure. 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 **+ Create a resource**, 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

* Name
sbnstest1 .servicebus.windows.net

* Pricing tier
Standard >

* Subscription
Prototype3

* Resource group i
 Create new Use existing
MyRG

* Location
East US

Pin to dashboard

Create Automation options

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

Creating a new namespace automatically generates an initial Shared Access Signature (SAS) rule with an associated pair of primary and secondary keys that each grant full control over all aspects of the namespace. See [Service Bus authentication and authorization](#) for information about how to create further rules with more constrained rights for regular senders and receivers. To copy the initial rule, follow these steps:

1. Click **All resources**, then click the newly created namespace name.
2. In the namespace window, click **Shared access policies**.
3. In the **Shared access policies** screen, click **RootManageSharedAccessKey**.

The screenshot shows the Azure Service Bus Shared access policies page. The left sidebar lists various settings like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Scale, Properties, Locks, Automation script, Queues, Topics, Diagnostics logs, Metrics (preview), and New support request. The 'Shared access policies' link is highlighted with a red box. The main pane displays a table with columns 'POLICY' and 'CLAIMS'. A policy named 'RootManageSharedAccessKey' is listed, and its 'CLAIMS' row is also highlighted with a red box.

4. In the **Policy: RootManageSharedAccessKey** window, 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.

The screenshot shows the 'SAS Policy: RootManageSharedAccessKey' configuration window. It includes checkboxes for Manage, Send, and Listen. Below these are fields for Primary Key (placeholder: Primary key here), Secondary Key (placeholder: Secondary key here), Primary Connection String (placeholder: Endpoint=sb://sbnstest1.servicebus.windows.n...), and Secondary Connection String (placeholder: Endpoint=sb://sbnstest1.servicebus.windows.n...). Each string field has a copy icon to its right.

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

2. Create a topic using the Azure portal

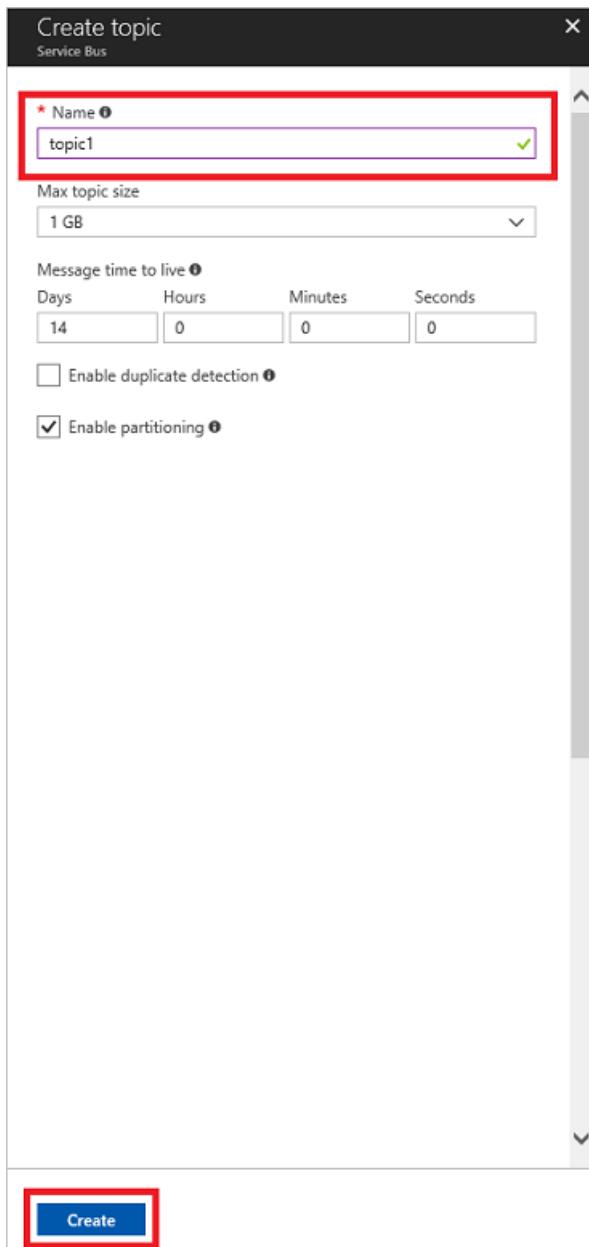
1. Log on to the [Azure portal](#).
2. In the left navigation pane of the portal, click **Service Bus** (if you don't see **Service Bus**, click **All services**, or click on **All resources**). Click the namespace in which you would like to create the topic.
3. The namespace overview window opens. Click **Topics**:

The screenshot shows the Azure Service Bus namespace overview for a resource named 'sbnatest1'. The left sidebar contains several sections: Overview (highlighted in blue), Activity log, Access control (IAM), Tags, Diagnose and solve problems, SETTINGS (Shared access policies, Scale, Properties, Locks, Automation script), ENTITIES (Queues, Topics - highlighted with a red box), MONITORING (Diagnostics logs, Metrics (preview)), and SUPPORT + TROUBLESHOOTING (New support request).

4. Click **+ Topic**.

The screenshot shows the Azure Service Bus Topics blade for a service named 'sbnstest1'. The left sidebar contains navigation links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Shared access policies, Scale, Properties, Locks, Automation script), Entities (Queues, Topics), Monitoring (Diagnostics logs, Metrics (preview)), and Support + Troubleshooting (New support request). The 'Topics' link in the Entities section is highlighted with a blue box. The main content area displays a table titled 'Topics' with columns: NAME, STATUS, MAX SIZE, SUBSCRIPTION COUNT, and ENABLE PARTITIONING. A search bar and refresh button are at the top of the table. The message 'no topics yet.' is displayed below the table.

5. Enter a name for the topic. Leave the other options with their default values.



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

3. Create a subscription to the topic

1. In the portal resources pane, click the namespace you created in step 1, then click **Topics**, and then click name of the topic you created in step 2.
2. At the top of the overview pane, click **+ Subscription** to add a subscription to this topic.

3. Enter a name for the subscription. Leave the other options with their default values.

4. Send messages to the topic

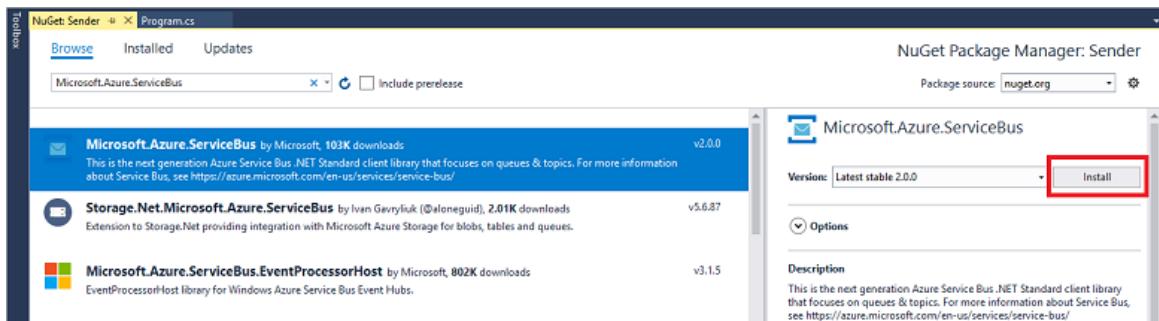
To send messages to the topic, write a C# console application using Visual Studio.

Create a console application

Launch Visual Studio and create a new **Console App (.NET Core)** project.

Add the Service Bus NuGet package

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



Write code to send messages to the topic

1. In Program.cs, add the following `using` statements at the top of the namespace definition, before the class declaration:

```
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Azure.ServiceBus;
```

2. Within the `Program` class, declare the following variables. Set the `ServiceBusConnectionString` variable to the connection string that you obtained when creating the namespace, and set `TopicName` to the name that you used when creating the topic:

```
const string ServiceBusConnectionString = "<your_connection_string>";
const string TopicName = "<your_topic_name>";
static ITopicClient topicClient;
```

3. Replace the default contents of `Main()` with the following line of code:

```
MainAsync().GetAwaiter().GetResult();
```

4. Directly after `Main()`, add the following asynchronous `MainAsync()` method that calls the send messages method:

```
static async Task MainAsync()
{
    const int numberOfMessages = 10;
    topicClient = new TopicClient(ServiceBusConnectionString, TopicName);

    Console.WriteLine("=====");
    Console.WriteLine("Press ENTER key to exit after sending all the messages.");
    Console.WriteLine("=====");

    // Send messages.
    await SendMessagesAsync(numberOfMessages);

    Console.ReadKey();

    await topicClient.CloseAsync();
}
```

5. Directly after the `MainAsync()` method, add the following `SendMessagesAsync()` method that performs the work of sending the number of messages specified by `numberOfMessagesToSend` (currently set to 10):

```
static async Task SendMessagesAsync(int numberOfMessagesToSend)
{
    try
    {
        for (var i = 0; i < numberOfMessagesToSend; i++)
        {
            // Create a new message to send to the topic.
            string messageBody = $"Message {i}";
            var message = new Message(Encoding.UTF8.GetBytes(messageBody));

            // Write the body of the message to the console.
            Console.WriteLine($"Sending message: {messageBody}");

            // Send the message to the topic.
            await topicClient.SendAsync(message);
        }
    }
    catch (Exception exception)
    {
        Console.WriteLine($"{DateTime.Now} :: Exception: {exception.Message}");
    }
}
```

6. Here is what your sender Program.cs file should look like.

```

namespace CoreSenderApp
{
    using System;
    using System.Text;
    using System.Threading;
    using System.Threading.Tasks;
    using Microsoft.Azure.ServiceBus;

    class Program
    {
        const string ServiceBusConnectionString = "<your_connection_string>";
        const string TopicName = "<your_topic_name>";
        static ITopicClient topicClient;

        static void Main(string[] args)
        {
            MainAsync().GetAwaiter().GetResult();
        }

        static async Task MainAsync()
        {
            const int numberOfMessages = 10;
            topicClient = new TopicClient(ServiceBusConnectionString, TopicName);

            Console.WriteLine("=====");
            Console.WriteLine("Press ENTER key to exit after sending all the messages.");
            Console.WriteLine("=====");

            // Send messages.
            await SendMessagesAsync(numberOfMessages);

            Console.ReadKey();

            await topicClient.CloseAsync();
        }

        static async Task SendMessagesAsync(int numberOfMessagesToSend)
        {
            try
            {
                for (var i = 0; i < numberOfMessagesToSend; i++)
                {
                    // Create a new message to send to the topic
                    string messageBody = $"Message {i}";
                    var message = new Message(Encoding.UTF8.GetBytes(messageBody));

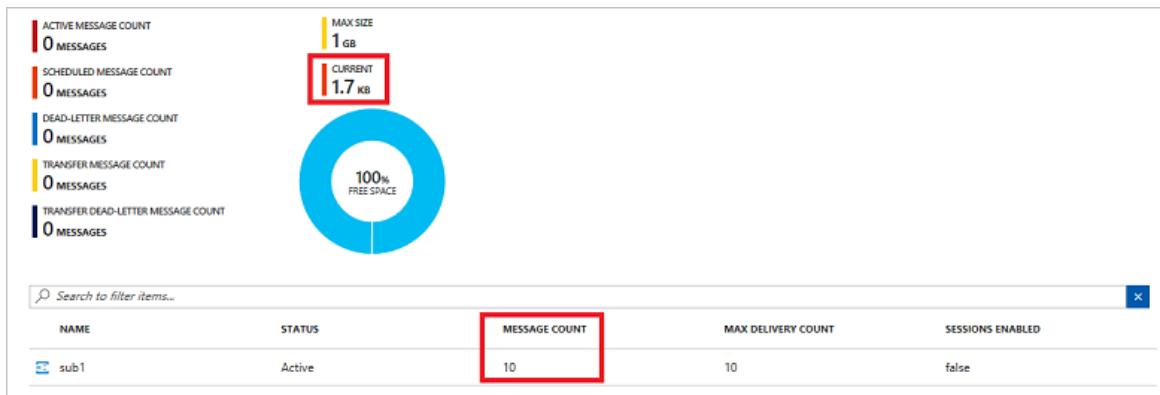
                    // Write the body of the message to the console
                    Console.WriteLine($"Sending message: {messageBody}");

                    // Send the message to the topic
                    await topicClient.SendAsync(message);
                }
            }
            catch (Exception exception)
            {
                Console.WriteLine($"{DateTime.Now} :: Exception: {exception.Message}");
            }
        }
    }
}

```

- Run the program, and check the Azure portal: click the name of your topic in the namespace **Overview** window. The topic **Essentials** screen is displayed. In the subscription listed near the bottom of the window, notice that the **Message Count** value for the subscription is now **10**. Each time you run the sender application without retrieving the messages (as described in the next section), this value increases.

by 10. Also note that the current size of the topic increments the **Current** value in the **Essentials** window each time the app adds messages to the topic.



5. Receive messages from the subscription

To receive the messages you just sent, create another .NET Core console application and install the **Microsoft.Azure.ServiceBus** NuGet package, similar to the previous sender application.

Write code to receive messages from the subscription

1. In Program.cs, add the following `using` statements at the top of the namespace definition, before the class declaration:

```
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Azure.ServiceBus;
```

2. Within the `Program` class, declare the following variables. Set the `ServiceBusConnectionString` variable to the connection string that you obtained when creating the namespace, set `TopicName` to the name that you used when creating the topic, and set `SubscriptionName` to the name that you used when creating the subscription to the topic:

```
const string ServiceBusConnectionString = "<your_connection_string>";
const string TopicName = "<your_topic_name>";
const string SubscriptionName = "<your_subscription_name>";
static ISubscriptionClient subscriptionClient;
```

3. Replace the default contents of `Main()` with the following line of code:

```
MainAsync().GetAwaiter().GetResult();
```

4. Directly after `Main()`, add the following asynchronous `MainAsync()` method that calls the `RegisterOnMessageHandlerAndReceiveMessages()` method:

```

static async Task MainAsync()
{
    subscriptionClient = new SubscriptionClient(ServiceBusConnectionString, TopicName,
SubscriptionName);

    Console.WriteLine("=====");
    Console.WriteLine("Press ENTER key to exit after receiving all the messages.");
    Console.WriteLine("=====");

    // Register subscription message handler and receive messages in a loop
    RegisterOnMessageHandlerAndReceiveMessages();

    Console.ReadKey();

    await subscriptionClient.CloseAsync();
}

```

5. Directly after the `MainAsync()` method, add the following method that registers the message handler and receives the messages sent by the sender application:

```

static void RegisterOnMessageHandlerAndReceiveMessages()
{
    // Configure the message handler options in terms of exception handling, number of concurrent
    // messages to deliver, etc.
    var messageHandlerOptions = new MessageHandlerOptions(ExceptionReceivedHandler)
    {
        // Maximum number of concurrent calls to the callback ProcessMessagesAsync(), set to 1 for
        // simplicity.
        // Set it according to how many messages the application wants to process in parallel.
        MaxConcurrentCalls = 1,

        // Indicates whether the message pump should automatically complete the messages after
        // returning from user callback.
        // False below indicates the complete operation is handled by the user callback as in
        ProcessMessagesAsync().
        AutoComplete = false
    };

    // Register the function that processes messages.
    subscriptionClient.RegisterMessageHandler(ProcessMessagesAsync, messageHandlerOptions);
}

```

6. Directly after the previous method, add the following `ProcessMessagesAsync()` method to process the received messages:

```

static async Task ProcessMessagesAsync(Message message, CancellationToken token)
{
    // Process the message.
    Console.WriteLine($"Received message: SequenceNumber:{message.SystemProperties.SequenceNumber}
Body:{Encoding.UTF8.GetString(message.Body)}");

    // Complete the message so that it is not received again.
    // This can be done only if the subscriptionClient is created in ReceiveMode.PeekLock mode (which
    // is the default).
    await subscriptionClient.CompleteAsync(message.SystemProperties.LockToken);

    // Note: Use the cancellationToken passed as necessary to determine if the subscriptionClient has
    // already been closed.
    // If subscriptionClient has already been closed, you can choose to not call CompleteAsync() or
    AbandonAsync() etc.
    // to avoid unnecessary exceptions.
}

```

7. Finally, add the following method to handle any exceptions that might occur:

```
// Use this handler to examine the exceptions received on the message pump.
static Task ExceptionReceivedHandler(ExceptionReceivedEventArgs exceptionReceivedEventArgs)
{
    Console.WriteLine($"Message handler encountered an exception
{exceptionReceivedEventArgs.Exception}.");
    var context = exceptionReceivedEventArgs.ExceptionReceivedContext;
    Console.WriteLine("Exception context for troubleshooting:");
    Console.WriteLine($"- Endpoint: {context.Endpoint}");
    Console.WriteLine($"- Entity Path: {context.EntityPath}");
    Console.WriteLine($"- Executing Action: {context.Action}");
    return Task.CompletedTask;
}
```

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

```
namespace CoreReceiverApp
{
    using System;
    using System.Text;
    using System.Threading;
    using System.Threading.Tasks;
    using Microsoft.Azure.ServiceBus;

    class Program
    {
        const string ServiceBusConnectionString = "<your_connection_string>";
        const string TopicName = "<your_topic_name>";
        const string SubscriptionName = "<your_subscription_name>";
        static ISubscriptionClient subscriptionClient;

        static void Main(string[] args)
        {
            MainAsync().GetAwaiter().GetResult();
        }

        static async Task MainAsync()
        {
            subscriptionClient = new SubscriptionClient(ServiceBusConnectionString, TopicName,
SubscriptionName);

            Console.WriteLine("=====");
            Console.WriteLine("Press ENTER key to exit after receiving all the messages.");
            Console.WriteLine("=====");

            // Register subscription message handler and receive messages in a loop.
            RegisterOnMessageHandlerAndReceiveMessages();

            Console.ReadKey();

            await subscriptionClient.CloseAsync();
        }

        static void RegisterOnMessageHandlerAndReceiveMessages()
        {
            // Configure the message handler options in terms of exception handling, number of
concurrent messages to deliver, etc.
            var messageHandlerOptions = new MessageHandlerOptions(ExceptionReceivedHandler)
            {
                // Maximum number of concurrent calls to the callback ProcessMessagesAsync(), set to
1 for simplicity.
                // Set it according to how many messages the application wants to process in
parallel.
                MaxConcurrentCalls = 1,
```

```

        // Indicates whether MessagePump should automatically complete the messages after
        // returning from User Callback.
        // False below indicates the Complete will be handled by the User Callback as in
`ProcessMessagesAsync` below.
        AutoComplete = false
    };

    // Register the function that processes messages.
    subscriptionClient.RegisterMessageHandler(ProcessMessagesAsync, messageHandlerOptions);
}

static async Task ProcessMessagesAsync(Message message, CancellationToken token)
{
    // Process the message.
    Console.WriteLine($"Received message: SequenceNumber:
{message.SystemProperties.SequenceNumber} Body:{Encoding.UTF8.GetString(message.Body)}");

    // Complete the message so that it is not received again.
    // This can be done only if the subscriptionClient is created in ReceiveMode.PeekLock
    mode (which is the default).
    await subscriptionClient.CompleteAsync(message.SystemProperties.LockToken);

    // Note: Use the cancellationToken passed as necessary to determine if the
    subscriptionClient has already been closed.
    // If subscriptionClient has already been closed, you can choose to not call
    CompleteAsync() or AbandonAsync() etc.
    // to avoid unnecessary exceptions.
}

static Task ExceptionReceivedHandler(ExceptionReceivedEventArgs exceptionReceivedEventArgs)
{
    Console.WriteLine($"Message handler encountered an exception
{exceptionReceivedEventArgs.Exception}.");
    var context = exceptionReceivedEventArgs.ExceptionReceivedContext;
    Console.WriteLine("Exception context for troubleshooting:");
    Console.WriteLine($"- Endpoint: {context.Endpoint}");
    Console.WriteLine($"- Entity Path: {context.EntityPath}");
    Console.WriteLine($"- Executing Action: {context.Action}");
    return Task.CompletedTask;
}
}
}

```

9. Run the program, and check the portal again. Notice that the **Message Count** and **Current** values are now **0**.



Congratulations! Using the .NET Standard library, you have now created a topic and subscription, sent 10 messages, and received those messages.

Next steps

Check out our [GitHub repository with samples](#) that demonstrate some of the more advanced features of

Service Bus messaging.

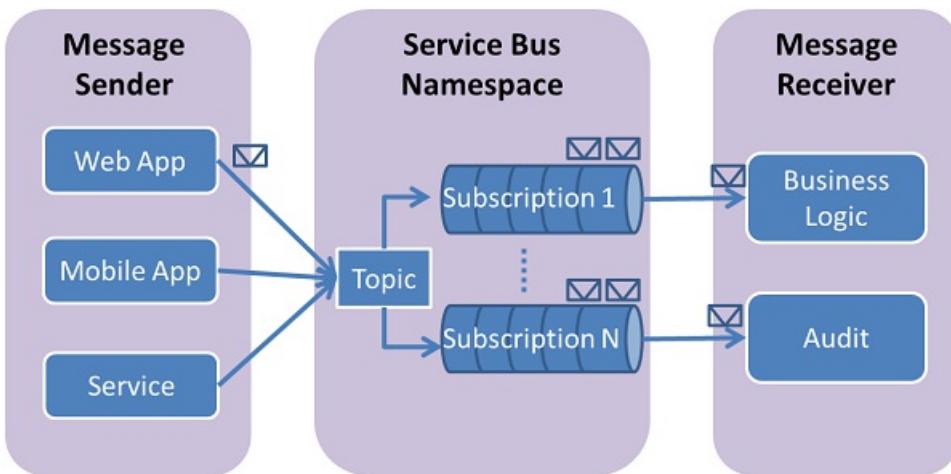
How to use Service Bus topics and subscriptions with Java

4/9/2018 • 11 min to read • [Edit Online](#)

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 large number of messages across a large number of users and applications.

Create a service namespace

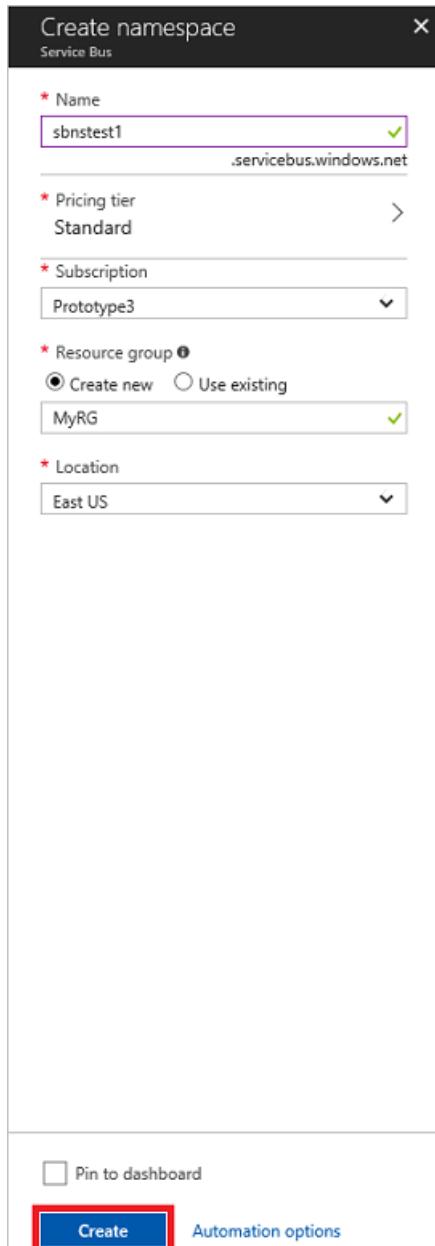
To begin using Service Bus topics and subscriptions in Azure, you must first create a *namespace*, which provides a scoping container for addressing Service Bus resources within your application.

To create a namespace:

To begin using Service Bus messaging entities in Azure, you must first create a namespace with a name that is unique across Azure. 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 **+ Create a resource**, 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.



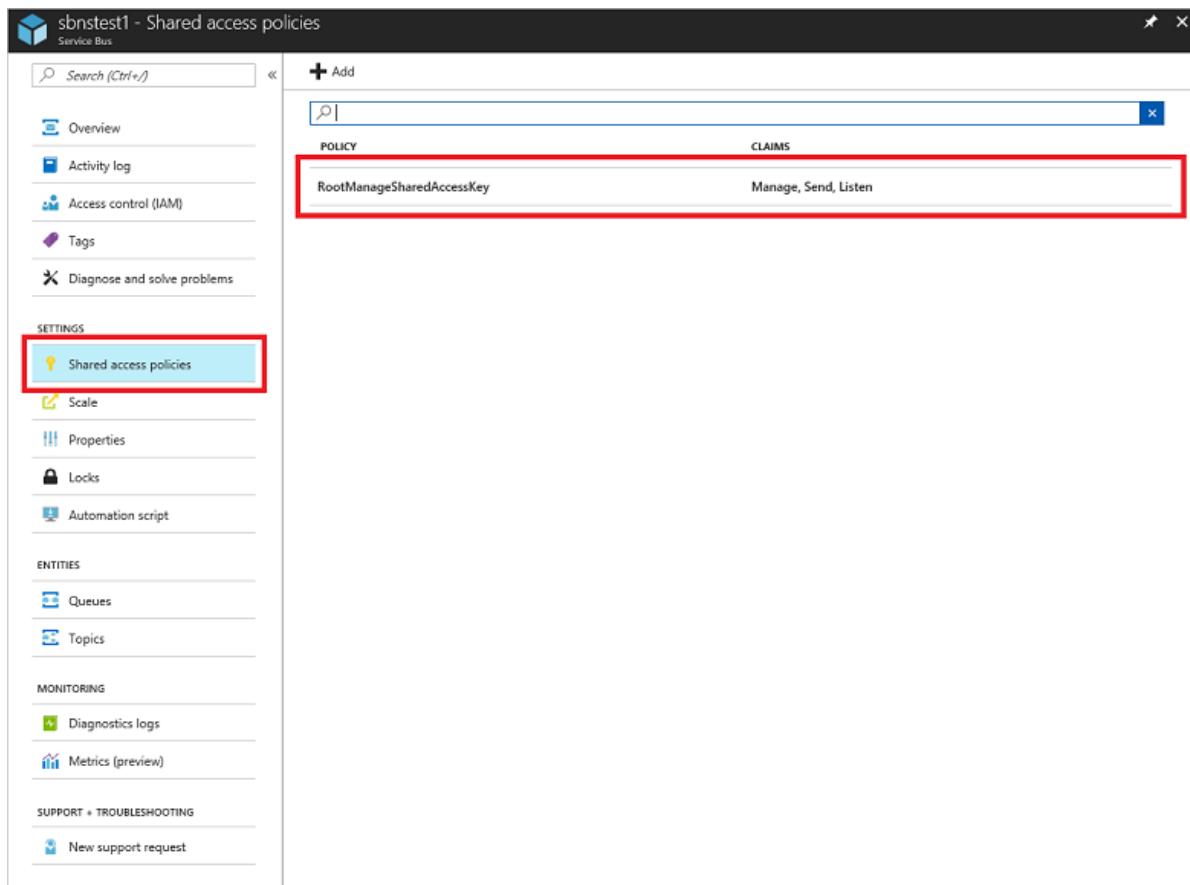
8. Click **Create**. The system now creates your namespace and enables it. You might have to wait several minutes as the system provisions resources for your account.

Obtain the management credentials

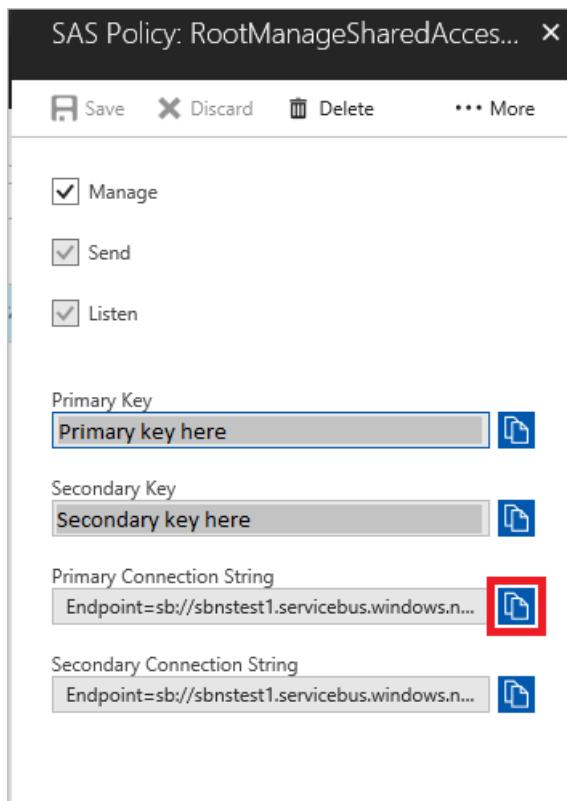
Creating a new namespace automatically generates an initial Shared Access Signature (SAS) rule with an associated pair of primary and secondary keys that each grant full control over all aspects of the namespace. See [Service Bus authentication and authorization](#) for information about how to create further rules with more

constrained rights for regular senders and receivers. To copy the initial rule, follow these steps:

1. Click **All resources**, then click the newly created namespace name.
2. In the namespace window, click **Shared access policies**.
3. In the **Shared access policies** screen, click **RootManageSharedAccessKey**.



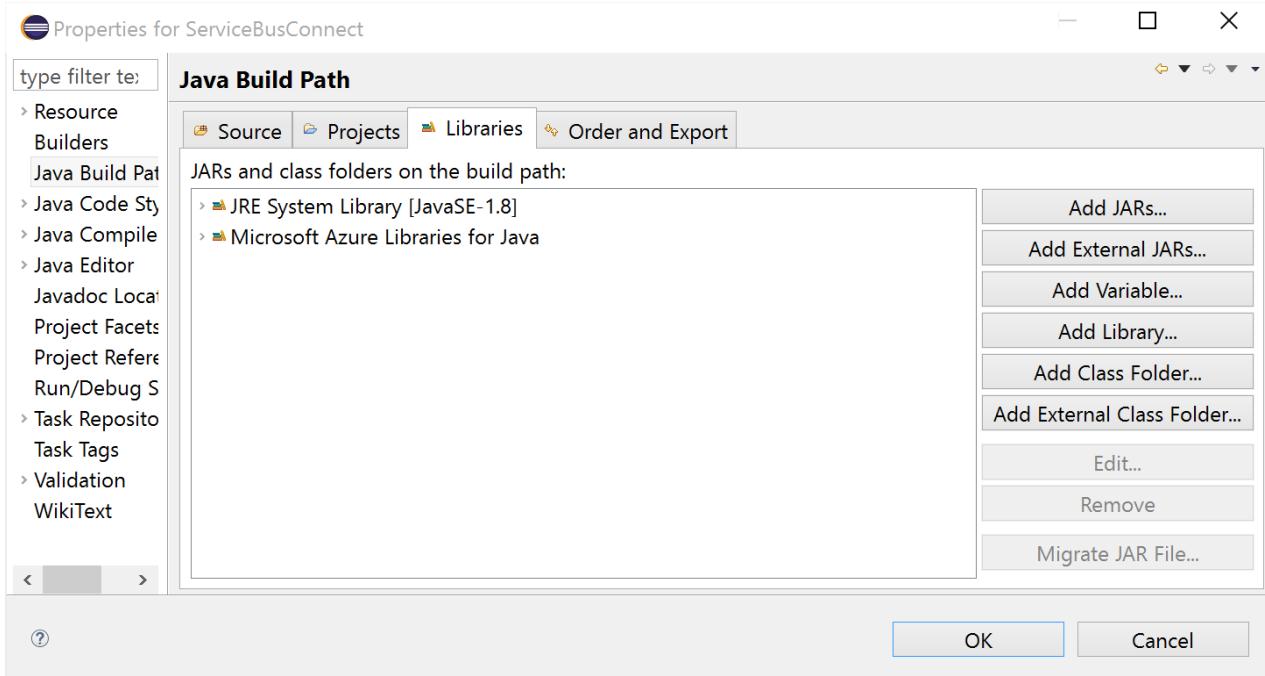
4. In the **Policy: RootManageSharedAccessKey** window, click the copy button next to **Connection string-primary key**, to copy the connection string to your clipboard for later use. Paste this value into Notepad or some other temporary location.



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:



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);

```

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

If no filter is specified when a new subscription is created, the **MatchAll** filter is the default filter that is used. 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 is always 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` is then 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 previous code snippet. Note how the **MessageNumber** property value of each message varies on the iteration of the loop (this value 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 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 limit 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** (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, 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 if a failure occurs. For example, 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 being consumed, then when the application restarts and begins consuming messages again, it has 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 marks the message as being consumed and removes it from the topic.

The following example demonstrates how messages can be received and processed using **PeekLock** (the default mode). The example performs a loop and processes messages in the `HighMessages` subscription and then exits when there are no more messages (alternatively, it can 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 causes 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 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** request is issued, then the message is redelivered to the application when it restarts. This process 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 **get messageId** method of the message, which remains 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 also deletes 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.

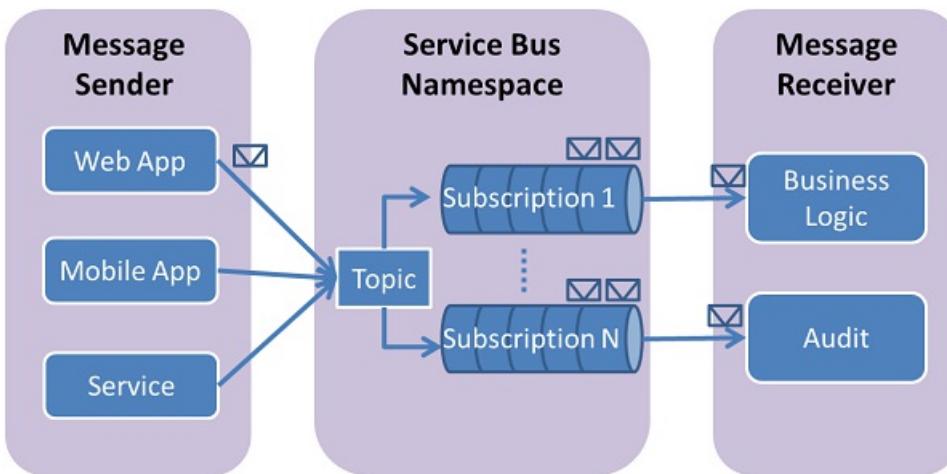
How to Use Service Bus topics and subscriptions with Node.js

11/28/2017 • 12 min to read • [Edit Online](#)

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, where 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. Filter rules enable 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 large number of messages across many users and applications.

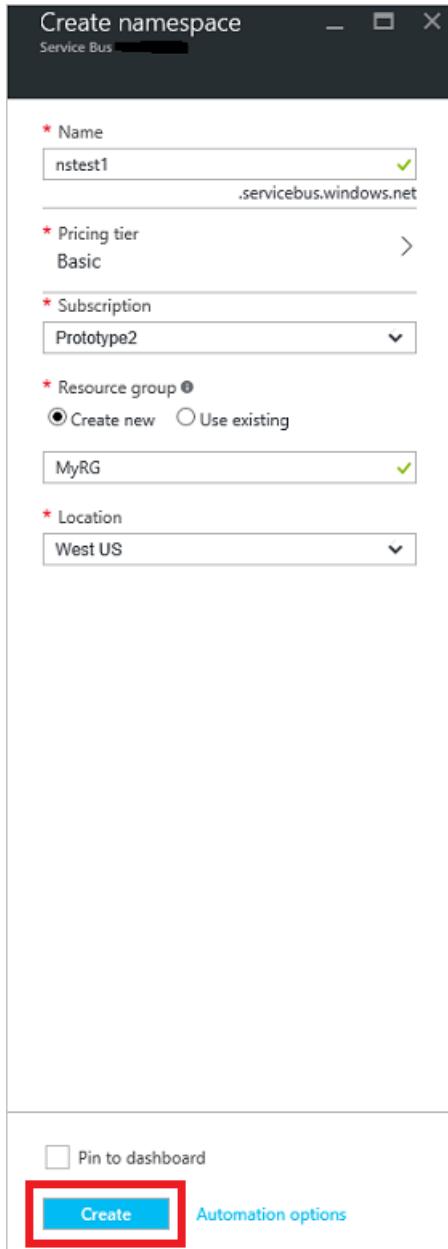
Create a namespace

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 **Create a resource**, 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 lives, or create a new one.
7. In **Location**, choose the country or region in which your namespace should be hosted.



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.

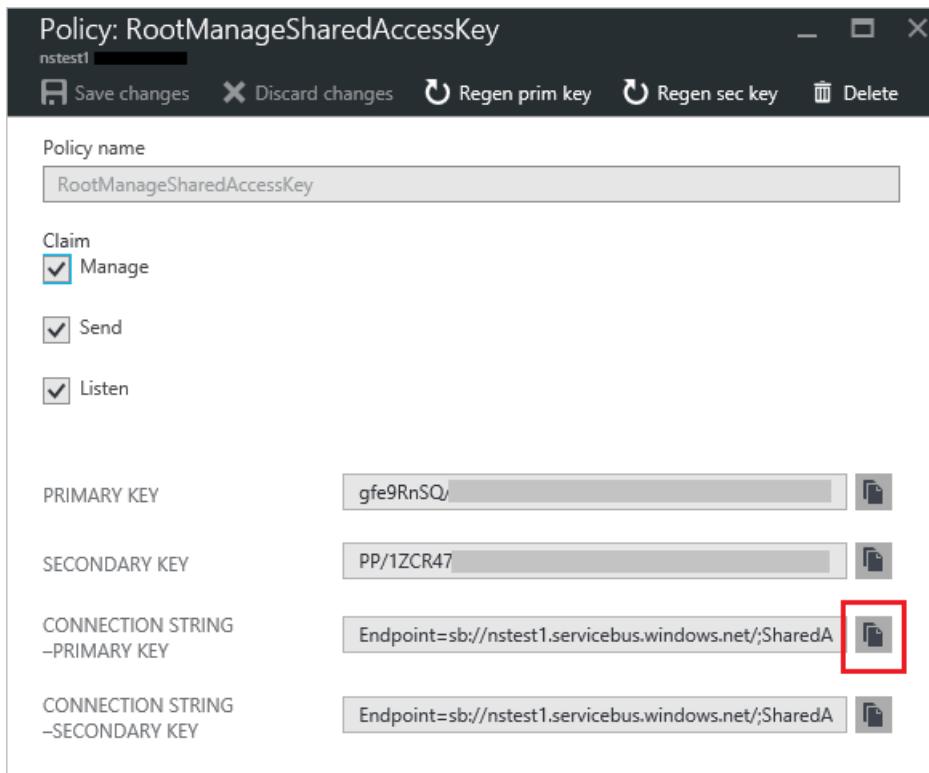
Obtain the credentials

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

The screenshot shows the 'Shared access policies' blade for a storage account named 'nctest1'. On the left, a navigation pane lists 'Overview', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'SETTINGS' (with 'Locks' and 'Automation script'), 'GENERAL' (with 'Properties' selected and 'Shared access policies' highlighted), 'SCALE' (with 'Scale' selected), 'QUEUES' (with 'Queues' selected), 'TOPICS' (with 'Topics' selected), and 'SUPPORT + TROUBLESHOOTING' (with 'New support request' selected). The main pane displays a table titled 'Shared access policies' with one row. The row contains the policy name 'RootManageSharedAccessKey' and the claim 'Manage, Send, Listen'. A red box highlights both the policy name and the claims column.

POLICY	CLAIMS
RootManageSharedAccessKey	Manage, Send, Listen

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



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
└── xmlhttp@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 variable `AZURE_SERVICEBUS_CONNECTION_STRING` for the connection string that you obtained from the earlier step, "Obtain the credentials". If this environment variable is not set, you must specify the account information when calling `createServiceBusService`.

For an example of setting the environment variables for an Azure Cloud Service, see [Node.js Cloud Service with Storage](#).

For an example of setting the environment variables 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.');
    }
});
```

The `createServiceBusService` method 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 invoke `finalCallback` otherwise, to end 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 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.

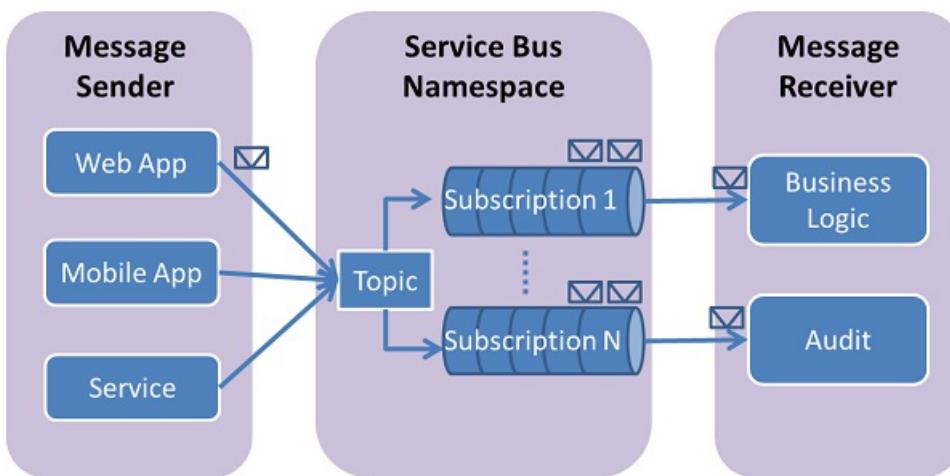
How to use Service Bus topics and subscriptions with PHP

10/9/2017 • 12 min to read • [Edit Online](#)

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, where 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. Filter rules enable 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 large number of messages across many users and applications.

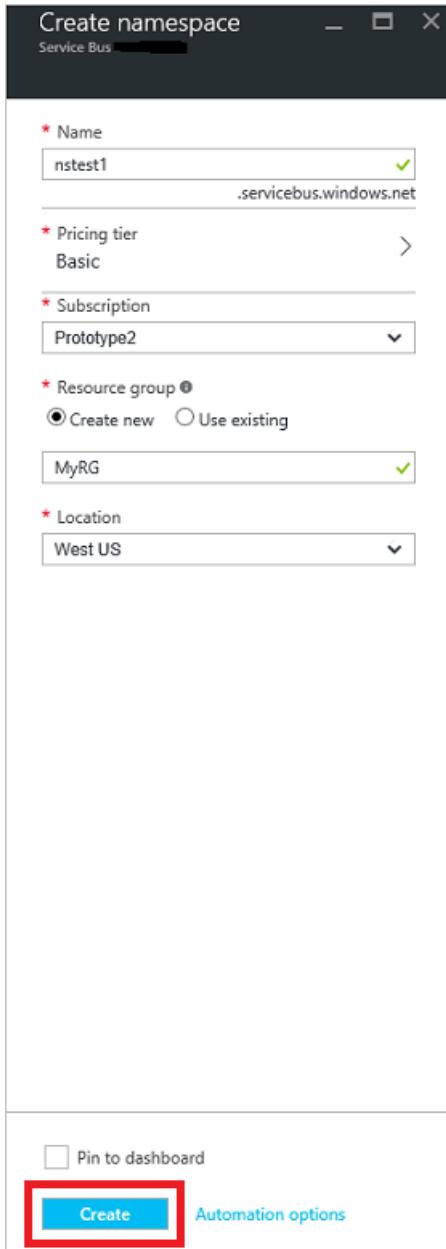
Create a namespace

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 **Create a resource**, 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 lives, or create a new one.
7. In **Location**, choose the country or region in which your namespace should be hosted.



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.

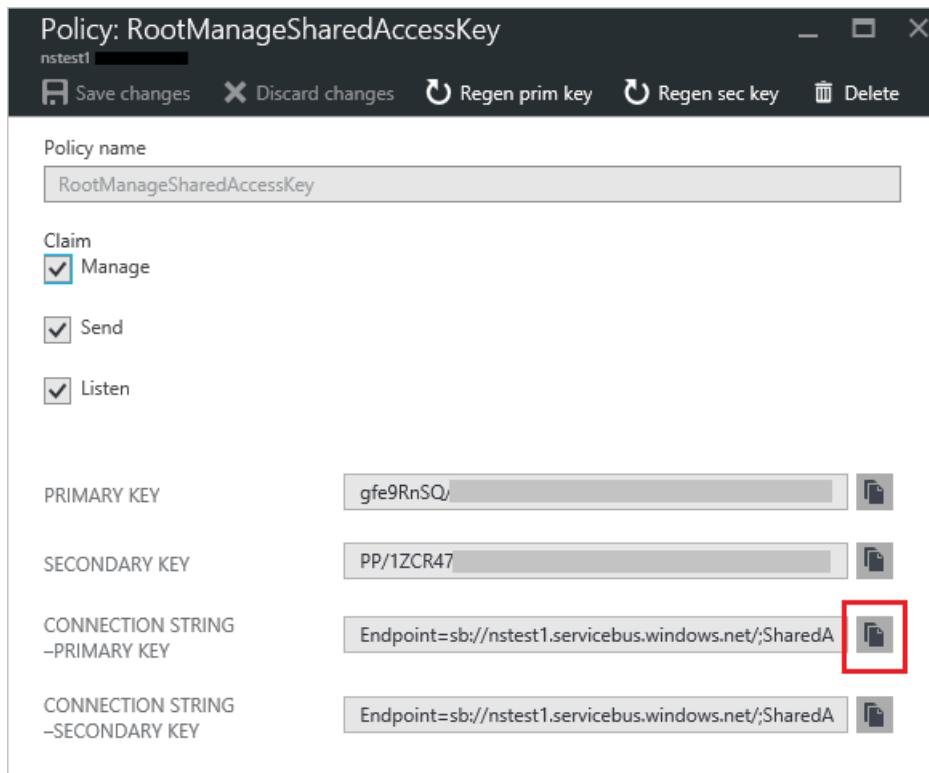
Obtain the credentials

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

The screenshot shows the 'Shared access policies' blade for a storage account named 'nctest1'. The left sidebar contains navigation links: Overview, Access control (IAM), Tags, Diagnose and solve problems, SETTINGS (Locks, Automation script), GENERAL (Properties, Shared access policies, Scale, Queues, Topics), and SUPPORT + TROUBLESHOOTING (New support request). The 'Shared access policies' link is highlighted with a red box. The main pane displays a table of shared access policies. A second red box highlights the first row in the table.

POLICY	CLAIMS
RootManageSharedAccessKey	Manage, Send, Listen

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



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. Create a file named **composer.json** in the root of your project and add the following code to it:

```
{  
    "require": {  
        "microsoft/azure-storage": "*"  
    }  
}
```

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

```
php composer.phar install
```

Alternatively go to the [Azure Storage PHP Client Library](#) on GitHub to clone the source code.

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 is always 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];SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=[Primary 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];SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=[Primary 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:
    // https://docs.microsoft.com/rest/api/storageservices/Common-REST-API-Error-Codes
    $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:
    // https://docs.microsoft.com/rest/api/storageservices/Common-REST-API-Error-Codes
    $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:
    // https://docs.microsoft.com/rest/api/storageservices/Common-REST-API-Error-Codes
    $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 marks 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:
    // https://docs.microsoft.com/rest/api/storageservices/Common-REST-API-Error-Codes
    $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 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 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 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 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 `get messageId` 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->deleteSubscription` 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:
    // https://docs.microsoft.com/rest/api/storageservices/Common-REST-API-Error-Codes
    $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.

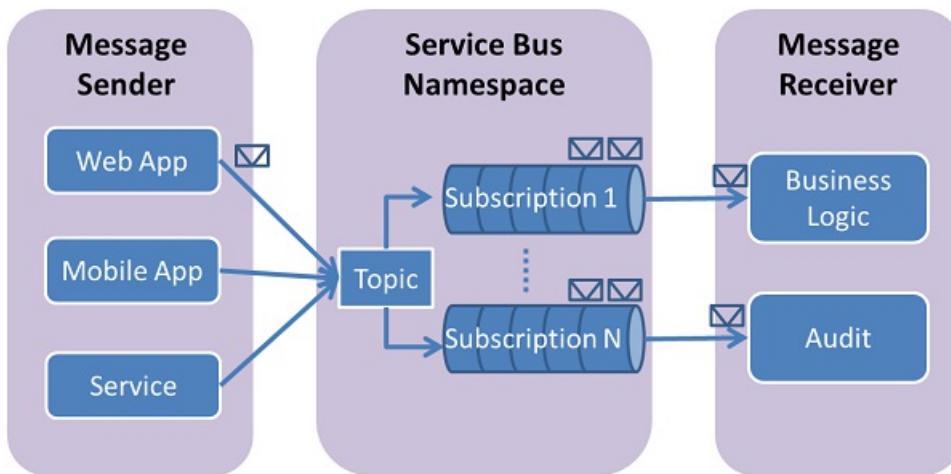
How to use Service Bus topics and subscriptions with Python

4/20/2018 • 8 min to read • [Edit Online](#)

This article describes how to use Service Bus topics and subscriptions. The samples are written in Python and use the [Azure Python SDK 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, where 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. Filter rules enable 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 large number of messages across many users and applications.

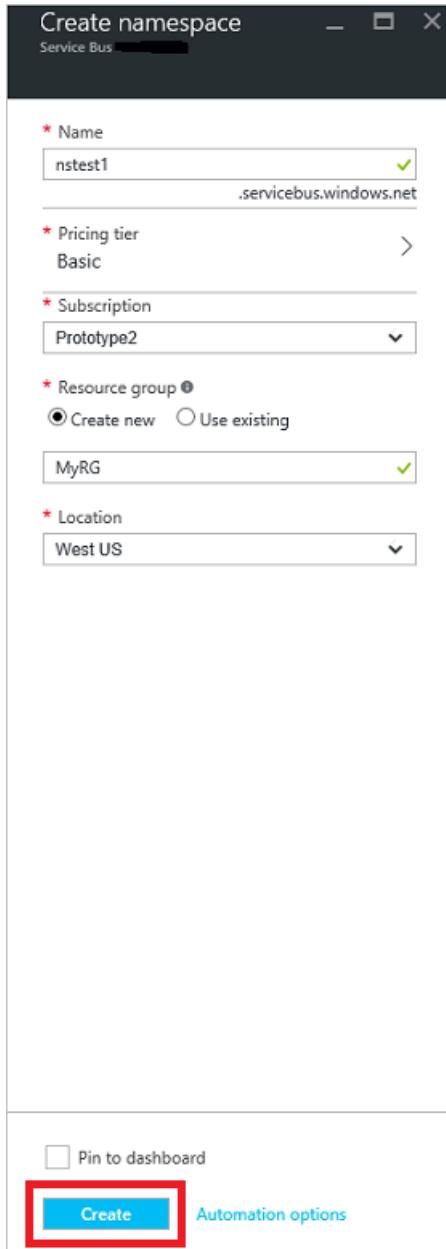
Create a namespace

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 **Create a resource**, 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 lives, or create a new one.
7. In **Location**, choose the country or region in which your namespace should be hosted.



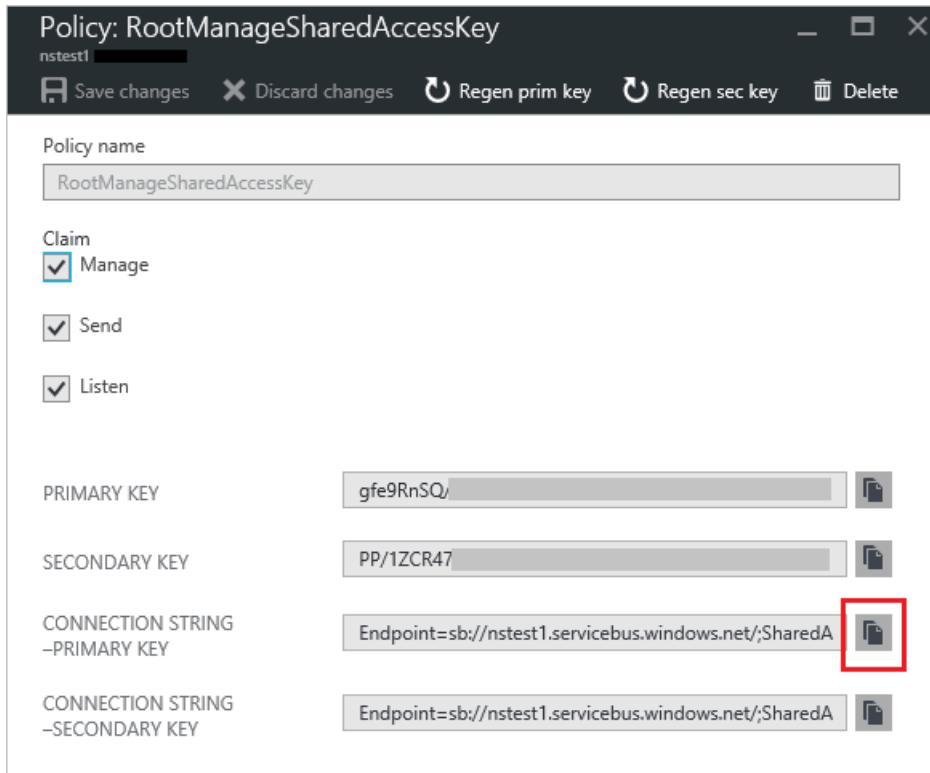
8. Click the **Create** button. The system now creates your namespace and enables it. You might have to wait several minutes as the system provisions resources for your account.

Obtain the credentials

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

The screenshot shows the Azure portal interface for managing shared access policies. On the left, a sidebar lists various management options like Overview, Access control (IAM), Tags, and Diagnose and solve problems. Under the GENERAL section, the 'Properties' item is selected and highlighted with a red box. Below it, the 'Shared access policies' item is also highlighted with a red box. The main content area displays a table titled 'Shared access policies' with one entry: 'RootManageSharedAccessKey' under the POLICY column and 'Manage, Send, Listen' under the CLAIMS column. A search bar at the top right allows filtering of items.

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



NOTE

If you need to install Python or the [Azure Python package](#), see the [Python Installation Guide](#).

Create a topic

The **ServiceBusService** object enables you to work with topics. 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, 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')
```

The `create_topic` method 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 one 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`. 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 if there is a failure. To understand this behavior, 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)
if msg.body is not None:
    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 method 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 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 behavior 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. To do so, you can use the **MessageId** property of the message, which remains 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](#).

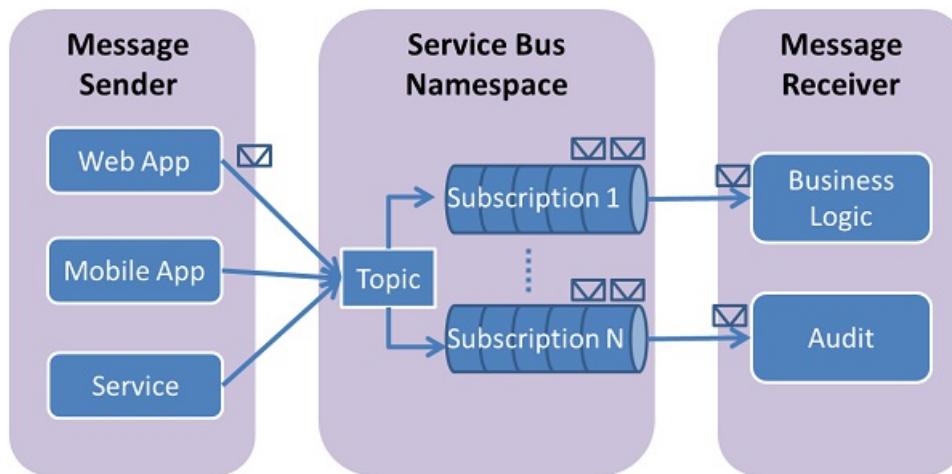
How to use Service Bus topics and subscriptions with Ruby

8/10/2017 • 12 min to read • [Edit Online](#)

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, where 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. Filter rules enable 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 large number of messages across many users and applications.

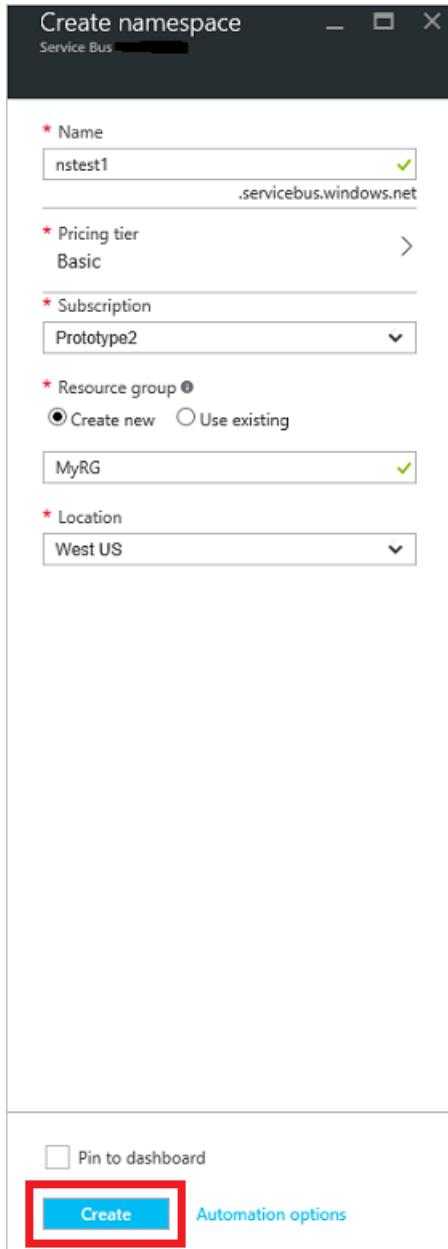
Create a namespace

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 **Create a resource**, 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 lives, or create a new one.
7. In **Location**, choose the country or region in which your namespace should be hosted.



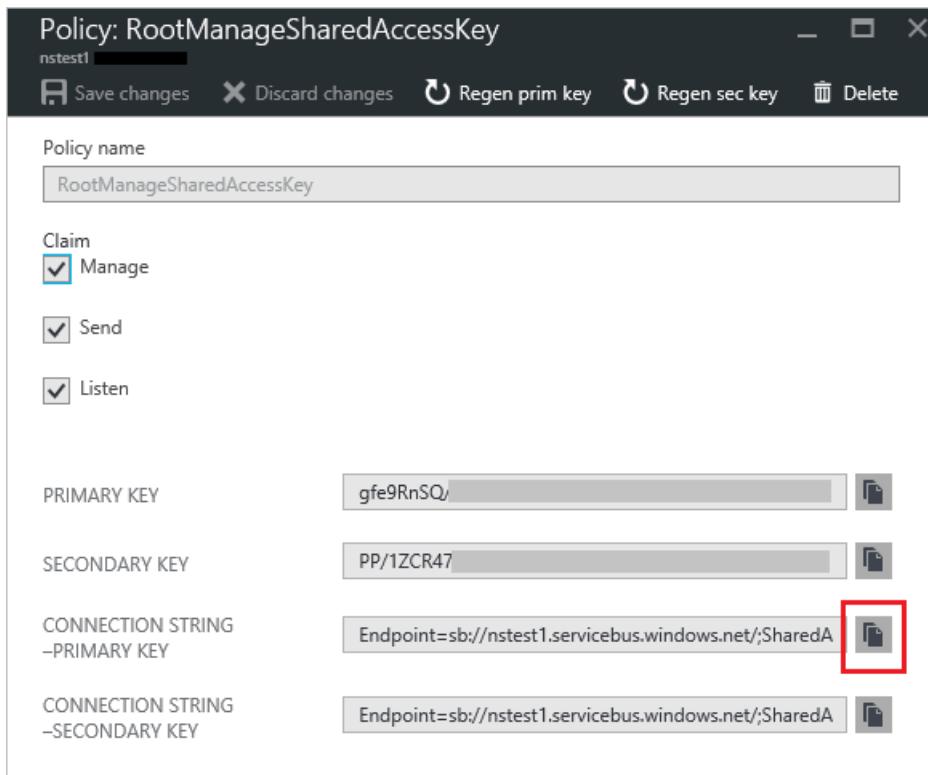
8. Click the **Create** button. The system now creates your namespace and enables it. You might have to wait several minutes as the system provisions resources for your account.

Obtain the credentials

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

The screenshot shows the Azure portal interface for a storage account named 'nstest1'. The left sidebar contains navigation links: Overview, Access control (IAM), Tags, Diagnose and solve problems, Locks, Automation script, Properties (which is selected and highlighted with a red box), Scale, Queues, Topics, and New support request. The main content area is titled 'Shared access policies' and lists one policy: 'RootManageSharedAccessKey'. This policy has the claim 'Manage, Send, Listen'. A red box highlights both the 'RootManageSharedAccessKey' row and the 'Claims' column header.

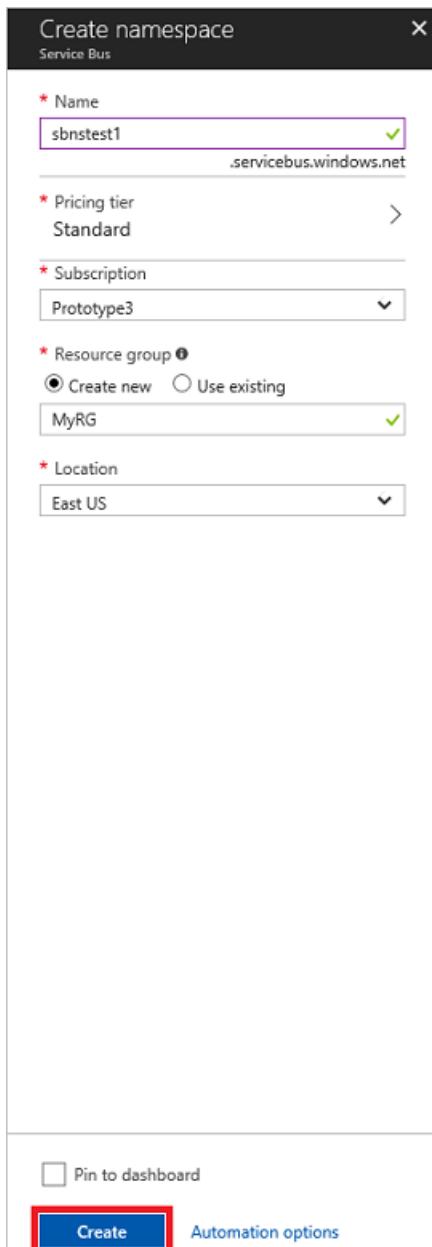
4. In the **Policy: RootManageSharedAccessKey** pane, click the copy button next to **Connection string-primary key**, to copy the connection string to your clipboard for later use.



To begin using Service Bus messaging entities in Azure, you must first create a namespace with a name that is unique across Azure. 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 **+ Create a resource**, 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.



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

Creating a new namespace automatically generates an initial Shared Access Signature (SAS) rule with an associated pair of primary and secondary keys that each grant full control over all aspects of the namespace. See [Service Bus authentication and authorization](#) for information about how to create further rules with more constrained rights for regular senders and receivers. To copy the initial rule, follow these steps:

1. Click **All resources**, then click the newly created namespace name.
2. In the namespace window, click **Shared access policies**.
3. In the **Shared access policies** screen, click **RootManageSharedAccessKey**.

The screenshot shows the Azure portal interface for a Service Bus entity named 'sbnstest1'. The left sidebar has sections for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, SETTINGS (with 'Shared access policies' highlighted by a red box), ENTITIES (Queues, Topics), MONITORING (Diagnostics logs, Metrics (preview)), and SUPPORT + TROUBLESHOOTING (New support request). The main content area shows a table for 'RootManageSharedAccessKey' with columns for POLICY and CLAIMS. The policy name is 'RootManageSharedAccessKey' and the claims are 'Manage, Send, Listen'.

4. In the **Policy: RootManageSharedAccessKey** window, 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.

The screenshot shows the 'SAS Policy: RootManageSharedAccessKey' configuration window. It includes buttons for Save, Discard, Delete, and More. Under the 'Manage' section, there are checkboxes for Manage, Send, and Listen, all of which are checked. Below this are fields for 'Primary Key' (placeholder 'Primary key here') and 'Secondary Key' (placeholder 'Secondary key here'), each with a copy icon. At the bottom are fields for 'Primary Connection String' (value 'Endpoint=sb://sbnstest1.servicebus.windows.n...') and 'Secondary Connection String' (value 'Endpoint=sb://sbnstest1.servicebus.windows.n...'), each also with a copy icon.

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 Azure Ruby 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

Use the following code to set the values of namespace, name of the key, key, signer and host:

```
Azure.configure do |config|
  config_sb_namespace = '<your azure service bus namespace>'
  config_sb_sas_key_name = '<your azure service bus access keyname>'
  config_sb_sas_key = '<your azure service bus access key>'
end
signer = Azure::ServiceBus::Auth::SharedAccessSigner.new
sb_host = "https://#{Azure.sb_namespace}.servicebus.windows.net"
```

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::ServiceBus::ServiceBusService.new(sb_host, { signer: signer})
begin
  topic = azure_service_bus_service.create_queue("test-topic")
rescue
  puts $!
end
```

You can also pass an **Azure::ServiceBus::Topic** object with additional options, which enable 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 5 GB 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](#) 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 an **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 an **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 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)
```

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_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.

Managed Service Identity (preview)

3/9/2018 • 4 min to read • [Edit Online](#)

A Managed Service Identity (MSI) is a cross-Azure feature that enables you to create a secure identity associated with the deployment under which your application code runs. You can then associate that identity with access-control roles that grant custom permissions for accessing specific Azure resources that your application needs.

With MSI, the Azure platform manages this runtime identity. You do not need to store and protect access keys in your application code or configuration, either for the identity itself, or for the resources you need to access. A Service Bus client app running inside an Azure App Service application or in a virtual machine with enabled MSI support does not need to handle SAS rules and keys, or any other access tokens. The client app only needs the endpoint address of the Service Bus Messaging namespace. When the app connects, Service Bus binds the MSI context to the client in an operation that is shown in an example later in this article.

Once it is associated with a managed service identity, a Service Bus client can perform all authorized operations. Authorization is granted by associating an MSI with Service Bus roles.

Service Bus roles and permissions

For the initial public preview release, you can only add a managed service identity to the "Owner" or "Contributor" roles of a Service Bus namespace, which grants the identity full control on all entities in the namespace. However, management operations that change the namespace topology are initially supported only through Azure Resource Manager and not through the native Service Bus REST management interface. This support also means that you cannot use the .NET Framework client [NamespaceManager](#) object within a managed service identity.

Use Service Bus with a Managed Service Identity

The following section describes the steps required to create and deploy a sample application that runs under a managed service identity, how to grant that identity access to a Service Bus Messaging namespace, and how the application interacts with Service Bus entities using that identity.

This introduction describes a web application hosted in [Azure App Service](#). The steps required for a VM-hosted application are similar.

Create an App Service web application

The first step is to create an App Service ASP.NET application. If you're not familiar with how to do this in Azure, follow [this how-to guide](#). However, instead of creating an MVC application as shown in the tutorial, create a Web Forms application.

Set up the managed service identity

Once you create the application, navigate to the newly created web app in the Azure portal (also shown in the how-to), then navigate to the **Managed Service Identity** page, and enable the feature:

The screenshot shows the Azure portal's left sidebar with various settings like Deployment options, Continuous Delivery (Preview), Application settings, Authentication / Authorization, Managed service identity (which is highlighted with a red box), and Backups. The main content area is titled 'Managed service identity' and contains instructions about using a managed Azure Active Directory identity for communication with other services. It includes a 'Register with Azure Active Directory' section with an 'Off' to 'On' toggle switch, and 'Save' and 'Discard' buttons.

Once you've enabled the feature, a new service identity is created in your Azure Active Directory, and configured into the App Service host.

Create a new Service Bus Messaging namespace

Next, [create a Service Bus Messaging namespace](#) in one of the Azure regions that have preview support for RBAC: **US East**, **US East 2**, or **West Europe**.

Navigate to the namespace **Access Control (IAM)** page on the portal, and then click **Add** to add the managed service identity to the **Owner** role. To do so, search for the name of the web application in the **Add permissions** panel **Select** field, and then click **Save**.

The screenshot shows the Azure portal's left sidebar with Service Bus options like Overview, Activity log, and Access control (IAM). The Access control (IAM) option is highlighted with a red box. The main content area is titled 'Add permissions' and shows a 'Role' dropdown set to 'Owner' and a 'Select' field with a placeholder 'Enter name here'. Below it, a list of 'Selected members' shows two items. At the bottom are 'Save' and 'Discard' buttons, both highlighted with red boxes.

The web application's managed service identity now has access to the Service Bus namespace, and to the queue you previously created.

Run the app

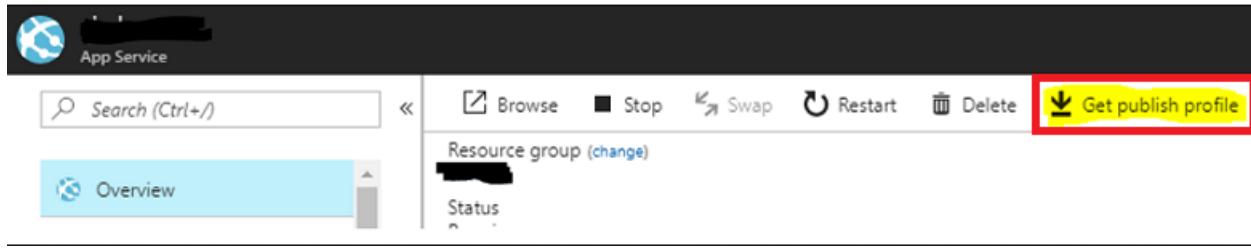
Now modify the default page of the ASP.NET application you created. You can also use the web application code from [this GitHub repository](#).

The Default.aspx page is your landing page. The code can be found in the Default.aspx.cs file. The result is a minimal web application with a few entry fields, and with **send** and **receive** buttons that connect to Service Bus to either send or receive messages.

Note how the **MessagingFactory** object is initialized. Instead of using the Shared Access Token (SAS) token provider, the code creates a token provider for the managed service identity with the `TokenProvider.CreateManagedServiceIdentityTokenProvider(ServiceAudience.ServiceBusAudience)` call. As such, there are no secrets to retain and use. The flow of the managed service identity context to Service Bus and the

authorization handshake are automatically handled by the token provider, which is a simpler model than using SAS.

Once you have made these changes, publish and run the application. An easy way to obtain the correct publishing data is to download and then import a publishing profile in Visual Studio:



To send or receive messages, enter the name of the namespace and the name of the entity you created, then click either **send** or **receive**.

Note that the managed service identity only works inside the Azure environment, and only in the App Service deployment in which you configured it. Also note that managed service identities do not work with App Service deployment slots at this time.

Next steps

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

- [Service Bus fundamentals](#)
- [Service Bus queues, topics, and subscriptions](#)
- [Get started with Service Bus queues](#)
- [How to use Service Bus topics and subscriptions](#)

Active Directory Role-Based Access Control (preview)

12/20/2017 • 4 min to read • [Edit Online](#)

Microsoft Azure provides integrated access control management for resources and applications based on Azure Active Directory (Azure AD). With Azure AD, you can either manage user accounts and applications specifically for your Azure based applications, or you can federate your existing Active Directory infrastructure with Azure AD for company-wide single-sign-on that also spans Azure resources and Azure hosted applications. You can then assign those Azure AD user and application identities to global and service-specific roles in order to grant access to Azure resources.

For Azure Service Bus, the management of namespaces and all related resources through the Azure portal and the Azure resource management API is already protected using the *role-based access control* (RBAC) model. RBAC for runtime operations is a feature now in public preview.

An application that uses Azure AD RBAC does not need to handle SAS rules and keys or any other access tokens specific to Service Bus. The client app interacts with Azure AD to establish an authentication context, and acquires an access token for Service Bus. With domain user accounts that require interactive login, the application never handles any credentials directly.

Service Bus roles and permissions

For the initial public preview, you can only add Azure AD accounts and service principals to the "Owner" or "Contributor" roles of a Service Bus Messaging namespace. This operation grants the identity full control over all entities in the namespace. Management operations that change the namespace topology are initially only supported though Azure resource management and not through the native Service Bus REST management interface. This support also means that the .NET Framework client [NamespaceManager](#) object cannot be used with an Azure AD account.

Use Service Bus with an Azure AD domain user account

The following section describes the steps required to create and run a sample application that prompts for an interactive Azure AD user to log on, how to grant Service Bus access to that user account, and how to use that identity to access Event Hubs.

This introduction describes a simple console application, the [code for which is on Github](#).

Create an Active Directory user account

This first step is optional. Every Azure subscription is automatically paired with an Azure Active Directory tenant and if you have access to an Azure subscription, your user account is already registered. That means you can just use your account.

If you still want to create a specific account for this scenario, [follow these steps](#). You must have permission to create accounts in the Azure Active Directory tenant, which may not be the case for larger enterprise scenarios.

Create a Service Bus namespace

Next, [create a Service Bus Messaging namespace](#) in one of the Azure regions that have preview support for RBAC: **US East**, **US East 2**, or **West Europe**.

Once the namespace is created, navigate to its **Access Control (IAM)** page on the portal, and then click **Add** to add the Azure AD user account to the Owner role. If you use your own user account and you created the namespace, you are already in the Owner role. To add a different account to the role, search for the name of the

web application in the **Add permissions** panel **Select** field, and then click the entry. Then click **Save**.

The screenshot shows the Azure portal interface. On the left, the sidebar has items like Overview, Activity log, and Access control (IAM) highlighted with a red box. The main area shows the 'Access control (IAM)' blade with a 'Search (Ctrl+ /)' bar. A red box highlights the '+ Add' button. The 'Add permissions' dialog is open on the right, showing a 'Role' dropdown set to 'Owner' and an 'Assign access to' dropdown set to 'Azure AD user, group, or application'. A red box highlights the 'Select' dropdown with 'Enter name here' and a green checkmark. Below it, 'Selected members:' shows a list with one item. At the bottom of the dialog, a red box highlights the 'Save' button.

The user account now has access to the Service Bus namespace, and to the queue you previously created.

Register the application

Before you can run the sample application, register it in Azure AD and approve the consent prompt that permits the application to access Azure Service Bus on its behalf.

Because the sample application is a console application, you must register a native application and add API permissions for **Microsoft.ServiceBus** to the "required permissions" set. Native applications also need a **redirect-URI** in Azure AD which serves as an identifier; the URI does not need to be a network destination. Use <http://servicebus.microsoft.com> for this example, because the sample code already uses that URI.

The detailed registration steps are explained in [this tutorial](#). Follow the steps to register a **Native** app, and then follow the update instructions to add the **Microsoft.ServiceBus** API to the required permissions. As you follow the steps, make note of the **TenantId** and the **ApplicationId**, as you will need these values to run the application.

Run the app

Before you can run the sample, edit the App.config file and, depending on your scenario, set the following values:

- `tenantId` : Set to **TenantId** value.
- `clientId` : Set to **ApplicationId** value.
- `clientSecret` : If you want to log on using the client secret, create it in Azure AD. Also, use a web app or API instead of a native app. Also, add the app under **Access Control (IAM)** in the namespace you previously created.
- `serviceBusNamespaceFQDN` : Set to the full DNS name of your newly created Service Bus namespace; for example, `example.servicebus.windows.net` .
- `queueName` : Set to the name of the queue you created.
- The redirect URI you specified in your app in the previous steps.

When you run the console application, you are prompted to select a scenario; click **Interactive User Login** by typing its number and pressing ENTER. The application displays a login window, asks for your consent to access Service Bus, and then uses the service to run through the send/receive scenario using the login identity.

Next steps

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

- [Service Bus fundamentals](#)

- [Service Bus queues, topics, and subscriptions](#)
- [Get started with Service Bus queues](#)
- [How to use Service Bus topics and subscriptions](#)

Service Bus Premium and Standard messaging tiers

5/1/2018 • 2 min to read • [Edit Online](#)

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. The Premium tier is recommended for production scenarios. 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 level 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 sections discuss a few differences between Premium and Standard messaging tiers.

Partitioned queues and topics

Partitioned queues and topics are not supported in Premium Messaging. For more information about partitioning, see [Partitioned queues and topics](#).

Express entities

Because Premium messaging runs in a completely isolated run-time environment, express entities are not supported in Premium namespaces. For more information about the express feature, see the [QueueDescription.EnableExpress](#) property.

If you have code running under Standard messaging and want to port it to the Premium tier, make sure the [EnableExpress](#) property is set to **false** (the default value).

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](#) in the [Azure portal](#). Make sure you select **Premium** under **Choose your pricing tier**.

The screenshot shows two overlapping windows. The left window is titled "Create namespace" under "Service Bus". It has fields for Name (nctest1), Pricing tier (Premium), Messaging Units (1), Subscription (Prototype3), Resource group (MyRG), and Location (East US). The right window is titled "Choose your pricing tier". It compares three tiers: Basic, Standard, and Premium. The Premium tier is highlighted with a red box around the "Recommended For Production Workloads" badge. The table below shows the estimated costs per month:

Tier	Capacity	Cost
Basic	Shared capacity	0.05 USD/MILLION/MONTH (ESTIMATED)
Standard	Shared capacity	10.00 USD/12.5MILLION/MONTH (ESTIM...)
Premium	Dedicated capacity	690.06 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](#)
- [Get started with Service Bus queues](#)

Storage queues and Service Bus queues – compared and contrasted

4/12/2018 • 15 min to read • [Edit Online](#)

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

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 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 by 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 these recommendations. 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.
- 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.
- 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 Azure 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
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)

COMPARISON CRITERIA	STORAGE QUEUES	SERVICE BUS QUEUES
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
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)

COMPARISON CRITERIA	STORAGE QUEUES	SERVICE BUS QUEUES
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 greater than 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	500 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 200 GB 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	Infinite (as of api-version 2017-07-27)	TimeSpan.Max
Maximum number of queues	Unlimited	10,000 (per service namespace)
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](#).
- In the [Standard tier](#), you can create Service Bus queues in 1, 2, 3, 4, or 5 GB sizes (the default is 1 GB). In the Premium tier, you can create queues up to 80 GB in size. In Standard tier, 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](#). In the Premium tier, only 2 partitions are created per queue.
- 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.)
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.

- [Get started with 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 \(blog post\)](#)
- [The Developer's Guide to Service Bus](#)

- Using the Queuing Service in Azure

Best Practices for performance improvements using Service Bus Messaging

2/13/2018 • 16 min to read • [Edit Online](#)

This article describes how to use [Azure Service Bus](#) to optimize performance when exchanging brokered messages. The first part of this article 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 subscription. 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.

NOTE

There is no risk of losing messages with batching, even if there is a Service Bus failure at the end of a 20ms batching interval.

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 $20 \times 3 \times 10 = 600$. 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, and are supported only in the Standard messaging tier. Entities created in [Premium namespaces](#) do not support the express option. 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

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](#).

Azure Service Bus Geo-disaster recovery

4/9/2018 • 5 min to read • [Edit Online](#)

When entire Azure regions or datacenters (if no [availability zones](#) are used) experience downtime, it is critical for data processing to continue to operate in a different region or datacenter. As such, *Geo-disaster recovery* and *Geo-replication* are important features for any enterprise. Azure Service Bus supports both geo-disaster recovery and geo-replication, at the namespace level.

The Geo-disaster recovery feature is globally available for the Service Bus Premium SKU.

Outages and disasters

It's important to note the distinction between "outages" and "disasters." An *outage* is the temporary unavailability of Azure Service Bus, and can affect some components of the service, such as a messaging store, or even the entire datacenter. However, after the problem is fixed, Service Bus becomes available again. Typically, an outage does not cause the loss of messages or other data. An example of such an outage might be a power failure in the datacenter. Some outages are only short connection losses due to transient or network issues.

A *disaster* is defined as the permanent, or longer-term loss of a Service Bus cluster, Azure region, or datacenter. The region or datacenter may or may not become available again, or may be down for hours or days. Examples of such disasters are fire, flooding, or earthquake. A disaster that becomes permanent might cause the loss of some messages, events, or other data. However, in most cases there should be no data loss and messages can be recovered once the data center is back up.

The Geo-disaster recovery feature of Azure Service Bus is a disaster recovery solution. The concepts and workflow described in this article apply to disaster scenarios, and not to transient, or temporary outages. For a detailed discussion of disaster recovery in Microsoft Azure, see [this article](#).

Basic concepts and terms

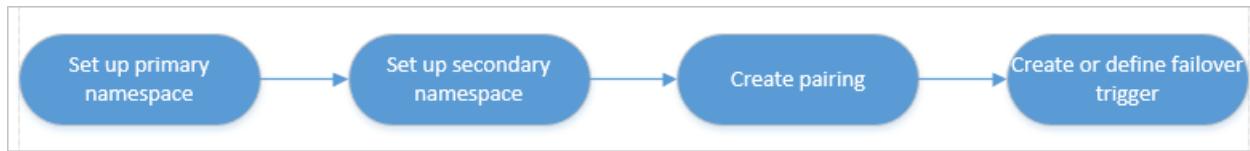
The disaster recovery feature implements metadata disaster recovery, and relies on primary and secondary disaster recovery namespaces. Note that the Geo-disaster recovery feature is available for the [Premium SKU](#) only. You do not need to make any connection string changes, as the connection is made via an alias.

The following terms are used in this article:

- **Alias:** The name for a disaster recovery configuration that you set up. The alias provides a single stable Fully Qualified Domain Name (FQDN) connection string. Applications use this alias connection string to connect to a namespace.
- **Primary/secondary namespace:** The namespaces that correspond to the alias. The primary namespace is "active" and receives messages (this can be an existing or new namespace). The secondary namespace is "passive" and does not receive messages. The metadata between both is in sync, so both can seamlessly accept messages without any application code or connection string changes. To ensure that only the active namespace receives messages, you must use the alias.
- **Metadata:** Entities such as queues, topics, and subscriptions; and their properties of the service that are associated with the namespace. Note that only entities and their settings are replicated automatically. Messages are not replicated.
- **Failover:** The process of activating the secondary namespace.

Setup and failover flow

The following section is an overview of the failover process, and explains how to set up the initial failover.



Setup

You first create or use an existing primary namespace, and a new secondary namespace, then pair the two. This pairing gives you an alias that you can use to connect. Because you use an alias, you do not have to change connection strings. Only new namespaces can be added to your failover pairing. Finally, you should add some monitoring to detect if a failover is necessary. In most cases, the service is one part of a large ecosystem, thus automatic failovers are rarely possible, as very often failovers must be performed in sync with the remaining subsystem or infrastructure.

Example

In one example of this scenario, consider a Point of Sale (POS) solution that emits either messages or events. Service Bus passes those events to some mapping or reformatting solution, which then forwards mapped data to another system for further processing. At that point, all of these systems might be hosted in the same Azure region. The decision on when and what part to fail over depends on the flow of data in your infrastructure.

You can automate failover either with monitoring systems, or with custom-built monitoring solutions. However, such automation takes extra planning and work, which is out of the scope of this article.

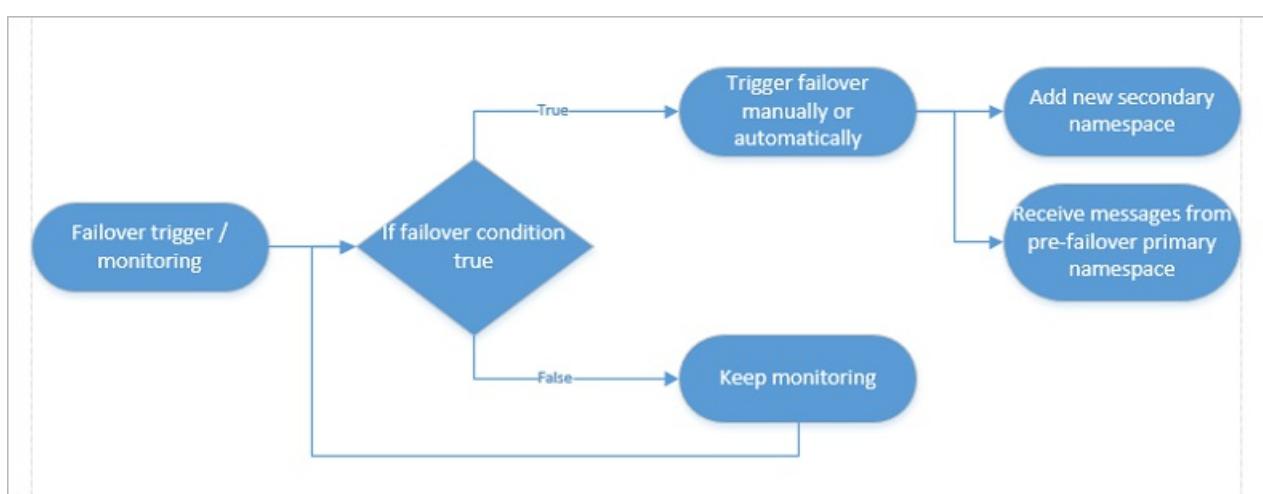
Failover flow

If you initiate the failover, two steps are required:

1. If another outage occurs, you want to be able to failover again. Therefore, set up another passive namespace and update the pairing.
2. Pull messages from the former primary namespace once it is available again. After that, use that namespace for regular messaging outside of your geo-recovery setup, or delete the old primary namespace.

NOTE

Only fail forward semantics are supported. In this scenario, you fail over and then re-pair with a new namespace. Failing back is not supported; for example, in a SQL cluster.



Management

If you made a mistake; for example, you paired the wrong regions during the initial setup, you can break the pairing of the two namespaces at any time. If you want to use the paired namespaces as regular namespaces, delete the alias.

Use existing namespace as alias

If you have a scenario in which you cannot change the connections of producers and consumers, you can reuse your namespace name as the alias name. See the [sample code on GitHub here](#).

Samples

The [samples on GitHub](#) show how to set up and initiate a failover. These samples demonstrate the following concepts:

- A .Net sample and settings required in Azure Active Directory to use Azure Resource Manager with Service Bus to setup and enable Geo-disaster recovery.
- Steps required to execute the sample code.
- How to use an existing namespace as alias.
- Steps to alternatively enable Geo-disaster recovery via PowerShell or CLI.
- [Send and receive](#) from the current primary or secondary namespace using the alias.

Considerations

Note the following considerations to keep in mind with this release:

1. In your failover planning, you should also consider the time factor. For example, if you lose connectivity for longer than 15 to 20 minutes, you might decide to initiate the failover.
2. The fact that no data is replicated means that currently active sessions are not replicated. Additionally, duplicate detection and scheduled messages may not work. New sessions, new scheduled messages and new duplicates will work.
3. Failing over a complex distributed infrastructure should be [rehearsed](#) at least once.
4. Synchronizing entities can take some time, approximately 50-100 entities per minute. Subscriptions and rules also count as entities.

Next steps

- See the Geo-disaster recovery [REST API reference here](#).
- Run the Geo-disaster recovery [sample on GitHub](#).
- See the Geo-disaster recovery [sample that sends messages to an alias](#).

To learn more about Service Bus messaging, see the following articles:

- [Service Bus fundamentals](#)
- [Service Bus queues, topics, and subscriptions](#)
- [Get started with Service Bus queues](#)
- [How to use Service Bus topics and subscriptions](#)
- [Rest API](#)

Asynchronous messaging patterns and high availability

1/30/2018 • 10 min to read • [Edit Online](#)

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. 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.
4. Session state is only maintained on the primary namespace.
5. 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, read more details about [paired namespaces](#).

Best practices for insulating applications against Service Bus outages and disasters

1/30/2018 • 6 min to read • [Edit Online](#)

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 now supports [Geo-disaster recovery](#) and [Geo-replication](#) at the namespace level.

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.

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.

Geo-replication

Service Bus supports Geo-disaster recovery and Geo-replication, at the namespace level. For more information, see [Azure Service Bus Geo-disaster recovery](#). The disaster recovery feature, available for the **Premium SKU** only, implements metadata disaster recovery, and relies on primary and secondary disaster recovery namespaces.

Next steps

To learn more about disaster recovery, see these articles:

- [Azure Service Bus Geo-disaster recovery](#)
- [Azure SQL Database Business Continuity](#)
- [Designing resilient applications for Azure](#)

.NET multi-tier application using Azure Service Bus queues

1/18/2018 • 14 min to read • [Edit Online](#)

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.

You will learn the following:

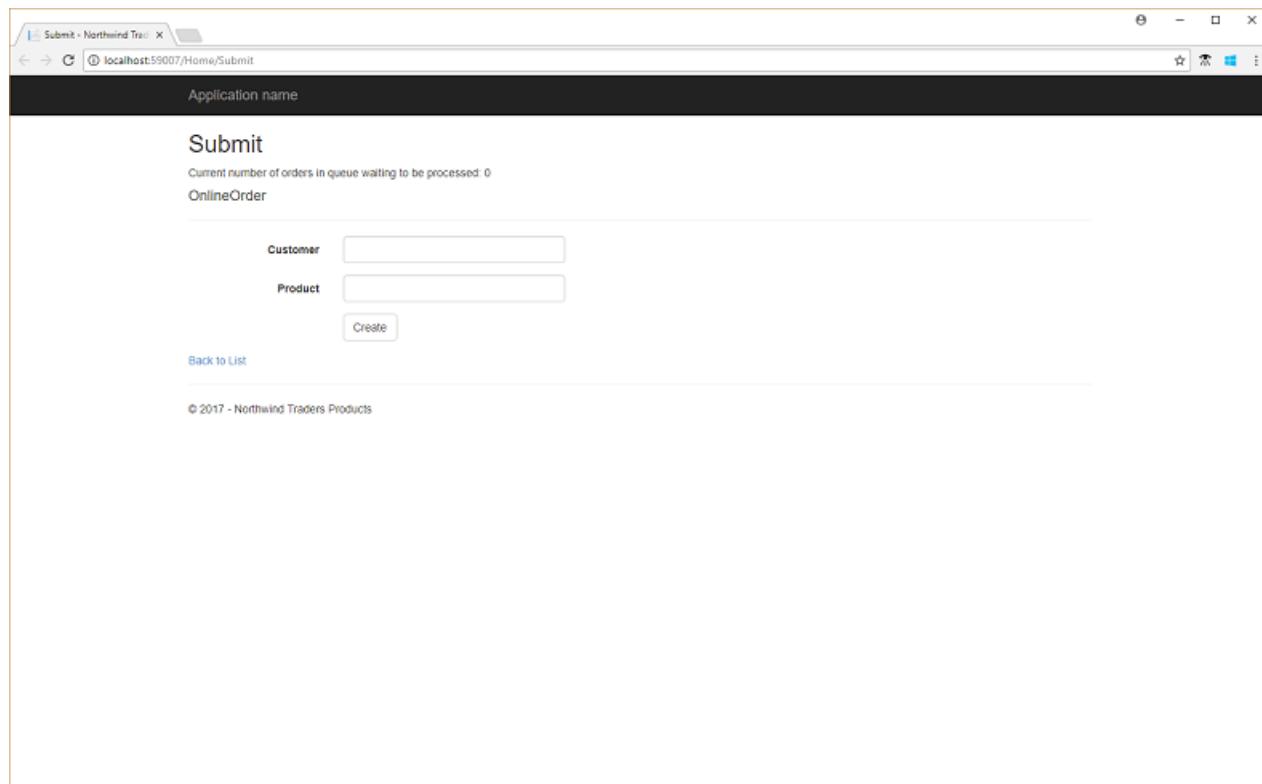
- How to enable your computer for Azure development with a single download and install.
- How to use Visual Studio to develop for Azure.
- How to create a multi-tier application in Azure using web and worker roles.
- How to communicate between tiers using Service Bus queues.

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. You can also try out the [.NET on-premises/cloud hybrid application](#) tutorial.

The following screen shot shows the completed application.

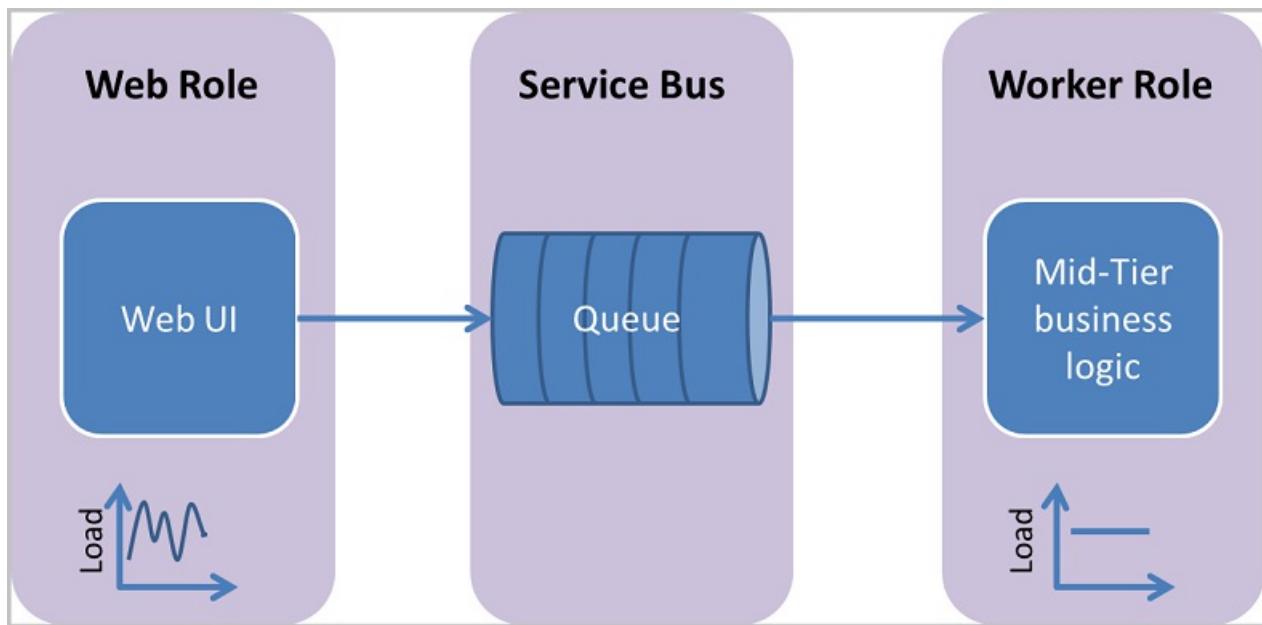


Scenario overview: inter-role communication

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 messaging for the communication between the tiers.

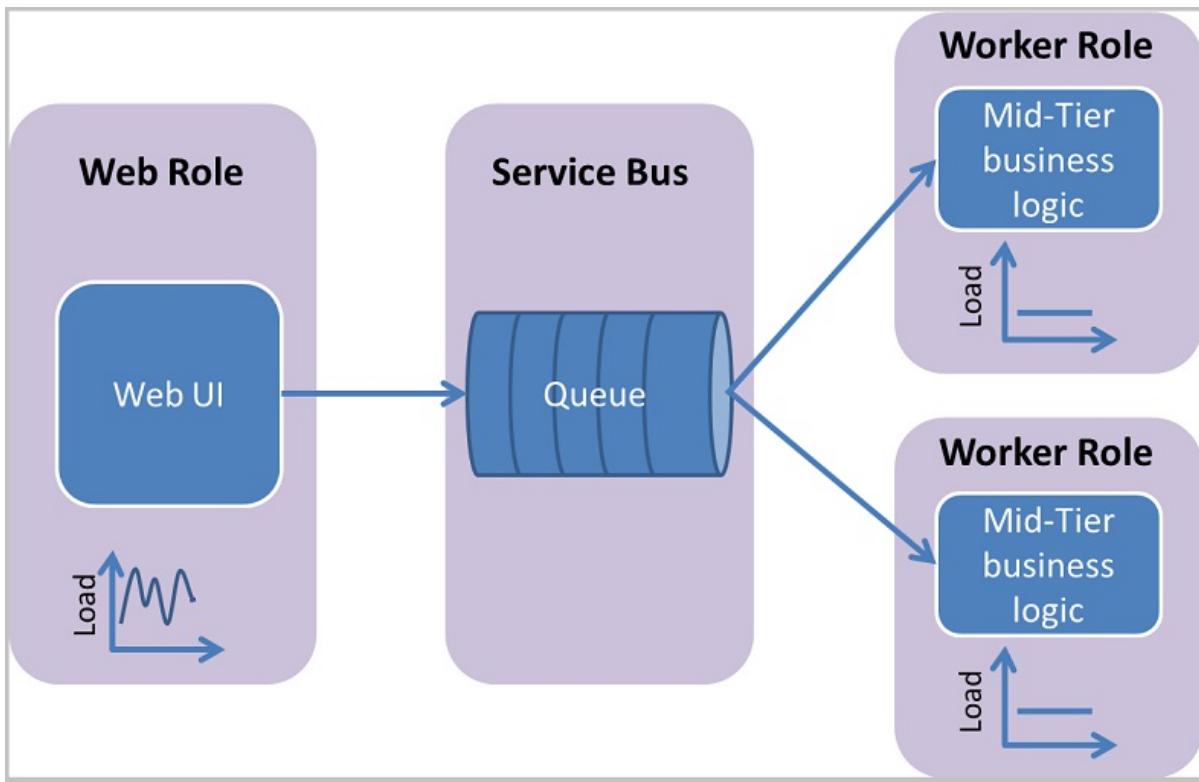
Using Service Bus messaging between the web and middle tiers decouples the two components. In contrast to direct messaging (that is, TCP or HTTP), the web tier does not connect to the middle tier directly; instead it pushes units of work, as messages, into Service Bus, which reliably retains them until 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, but they also work with Visual Studio 2017.
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 *namespace*, and obtain a [Shared Access Signature \(SAS\)](#) key for that namespace. 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 name and SAS key provides the credentials for Service Bus to authenticate access to an application.

To begin using Service Bus messaging entities in Azure, you must first create a namespace with a name that is unique across Azure. 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 **+ Create a resource**, 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.

The screenshot shows the 'Create namespace' dialog box for Service Bus. The 'Name' field contains 'sbnstest1'. The 'Pricing tier' is set to 'Standard'. The 'Subscription' dropdown shows 'Prototype3'. Under 'Resource group', the 'Create new' option is selected, and 'MyRG' is chosen. The 'Location' is set to 'East US'. At the bottom, there is a 'Pin to dashboard' checkbox and a 'Create' button, which is highlighted with a red border.

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

Creating a new namespace automatically generates an initial Shared Access Signature (SAS) rule with an associated pair of primary and secondary keys that each grant full control over all aspects of the namespace. See [Service Bus authentication and authorization](#) for information about how to create further rules with more constrained rights for regular senders and receivers. To copy the initial rule, follow these steps:

1. Click **All resources**, then click the newly created namespace name.
2. In the namespace window, click **Shared access policies**.
3. In the **Shared access policies** screen, click **RootManageSharedAccessKey**.

The screenshot shows the Azure Service Bus Shared access policies page. The 'Shared access policies' section is highlighted with a red box. A specific policy named 'RootManageSharedAccessKey' is selected, and its details are shown in the main pane. The 'CLAIMS' section shows 'Manage, Send, Listen'. The left sidebar lists various management options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Scale, Properties, Locks, Automation script, Queues, Topics, Diagnostics logs, Metrics (preview), and New support request.

4. In the **Policy: RootManageSharedAccessKey** window, 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.

The screenshot shows the 'SAS Policy: RootManageSharedAccessKey' configuration window. It includes checkboxes for Manage, Send, and Listen. Below these are fields for Primary Key (placeholder: 'Primary key here'), Secondary Key (placeholder: 'Secondary key here'), Primary Connection String (placeholder: 'Endpoint=sb://sbnstest1.servicebus.windows.n...', with a copy icon), and Secondary Connection String (placeholder: 'Endpoint=sb://sbnstest1.servicebus.windows.n...', with a copy icon).

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

Create a web role

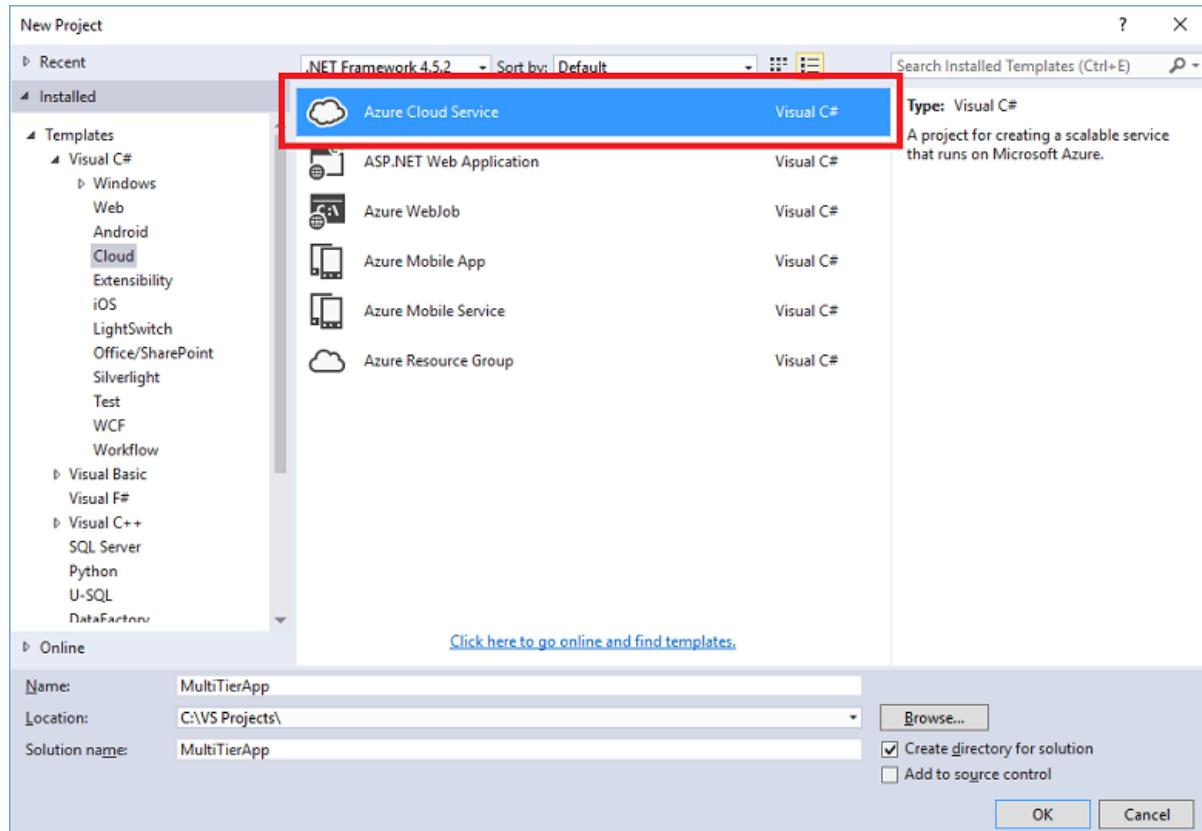
In this section, you build the front end of your application. First, you create the pages that your application displays. After that, add code that submits items to a Service Bus queue and displays status information about the queue.

Create the project

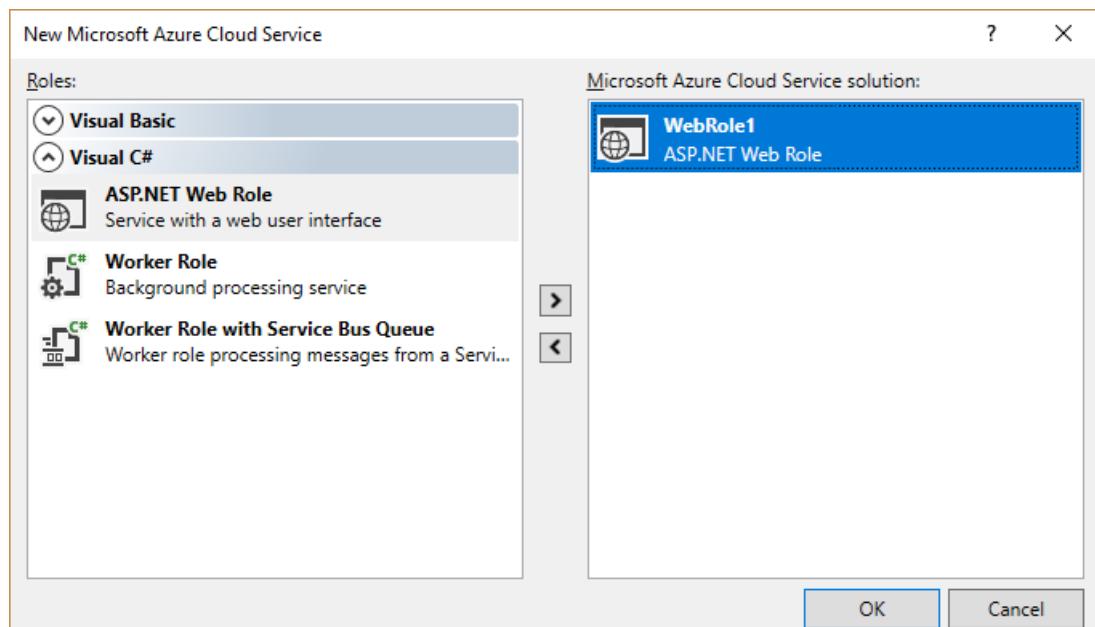
1. Using administrator privileges, start Visual Studio: right-click the **Visual Studio** program icon, and then click **Run as administrator**. The Azure compute emulator, discussed later in this article, requires that Visual Studio be started with administrator privileges.

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**.

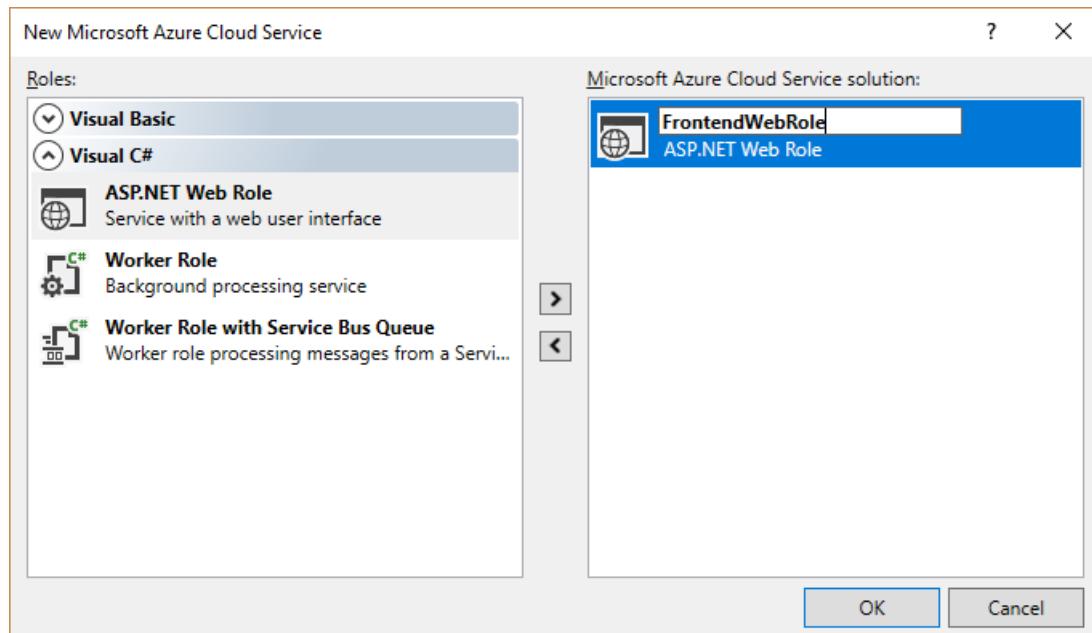


3. From the **Roles** pane, double-click **ASP.NET Web Role**.

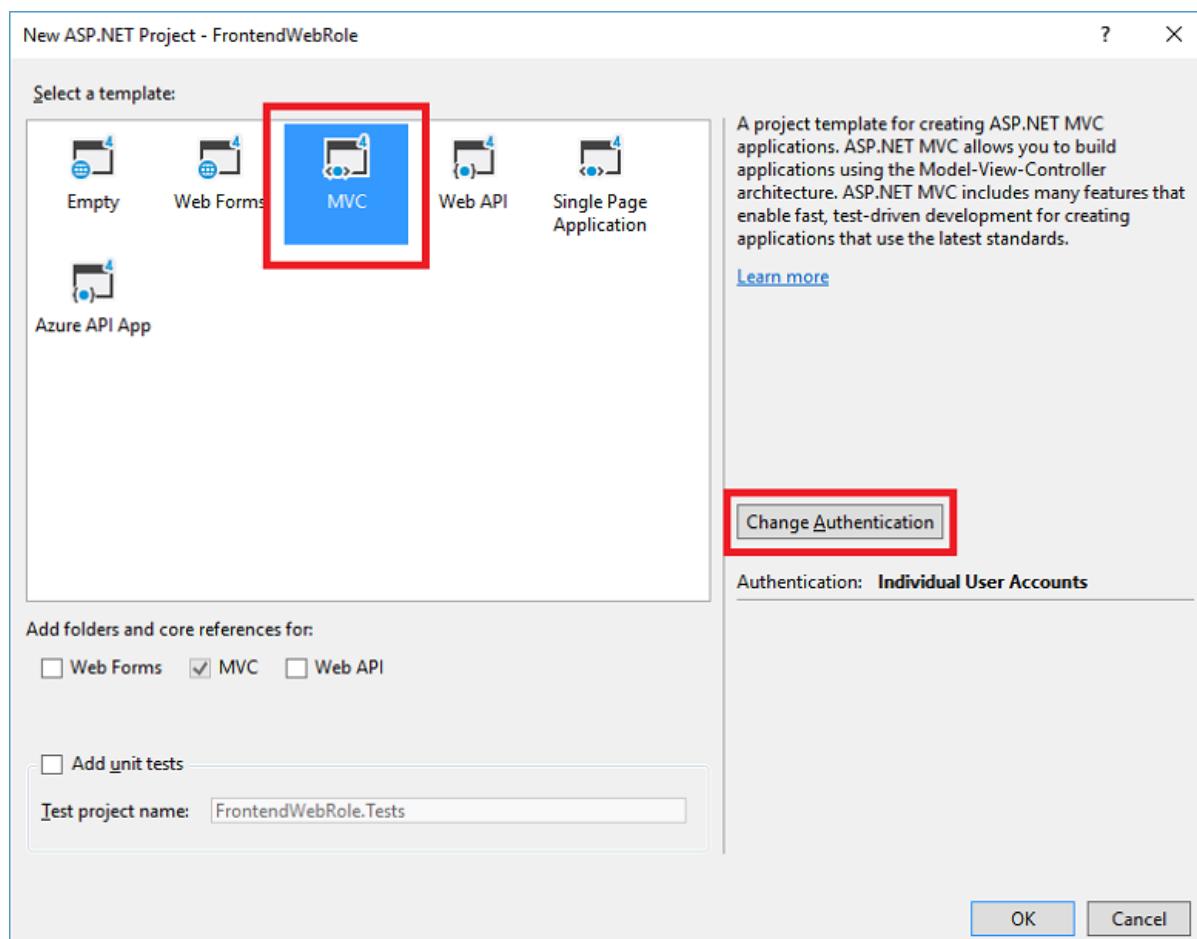


4. Hover over **WebRole1** under **Azure Cloud Service solution**, click the pencil icon, and rename the web

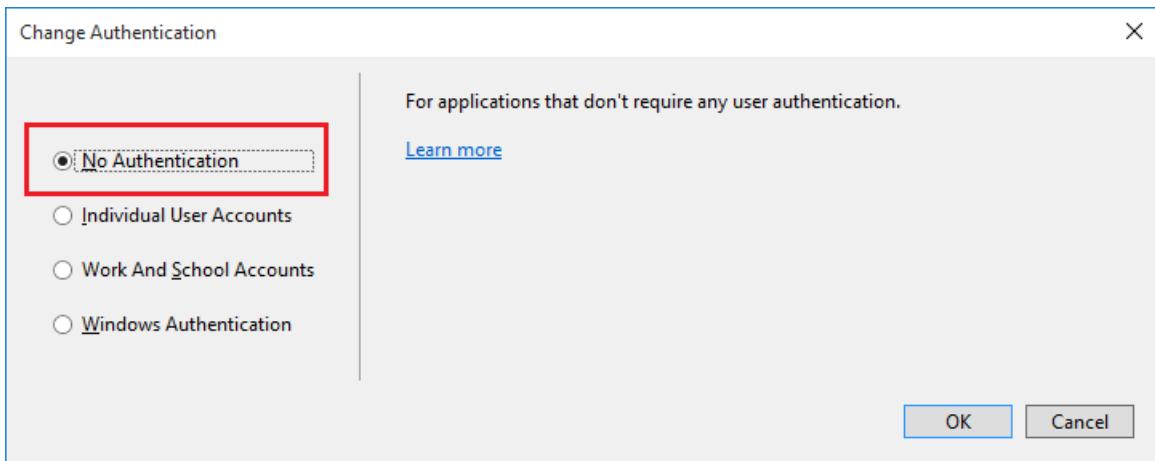
role to **FrontendWebRole**. Then click **OK**. (Make sure you enter "Frontend" with a lower-case 'e,' not "FrontEnd".)



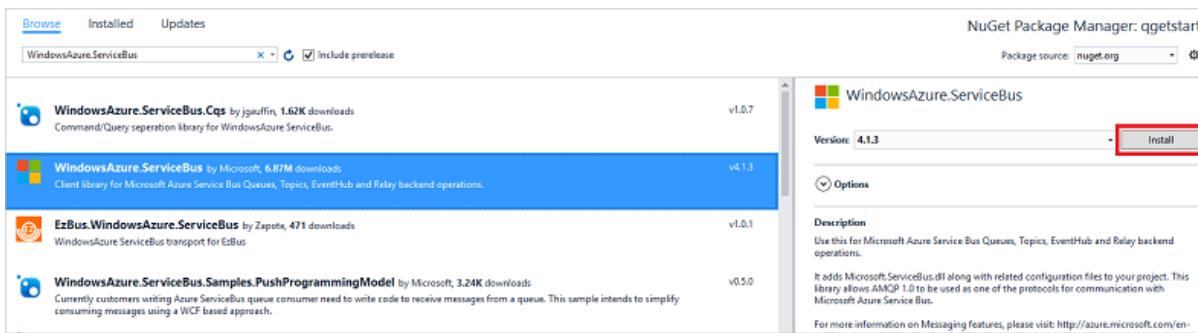
5. From the **New ASP.NET Project** dialog box, in the **Select a template** list, click **MVC**.



6. Still in the **New ASP.NET Project** dialog box, click the **Change Authentication** button. In the **Change Authentication** dialog box, ensure that **No Authentication** is selected, and then click **OK**. For this tutorial, you're deploying an app that doesn't need a user login.



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 **WindowsAzure.ServiceBus**. Select the **WindowsAzure.ServiceBus** package, click **Install**, and accept the terms of use.



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();
        }

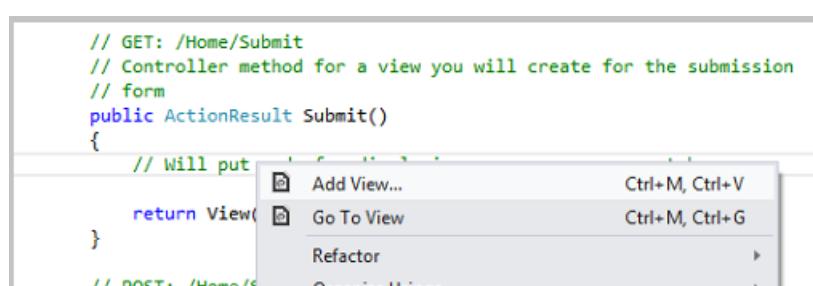
        // GET: /Home/Submit.
        // Controller method for a view you will create for the submission
        // form.
        public ActionResult Submit()
        {
            // Will put code for displaying queue message count here.

            return View();
        }

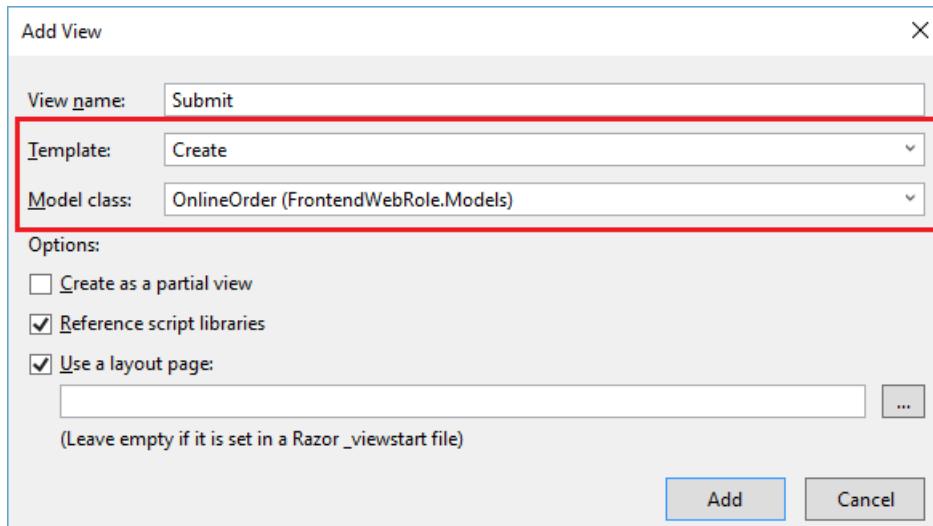
        // POST: /Home/Submit.
        // Controller method for handling submissions from the submission
        // form.
        [HttpPost]
        // Attribute to help prevent cross-site scripting attacks and
        // cross-site request forgery.
        [ValidateAntiForgeryToken]
        public ActionResult Submit(OnlineOrder order)
        {
            if (ModelState.IsValid)
            {
                // Will put code for submitting to queue here.

                return RedirectToAction("Submit");
            }
            else
            {
                return View(order);
            }
        }
    }
}
```

4. From 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**.



6. A dialog box appears for creating the view. In the **Template** list, choose **Create**. In the **Model class** list, select the **OnlineOrder** class.



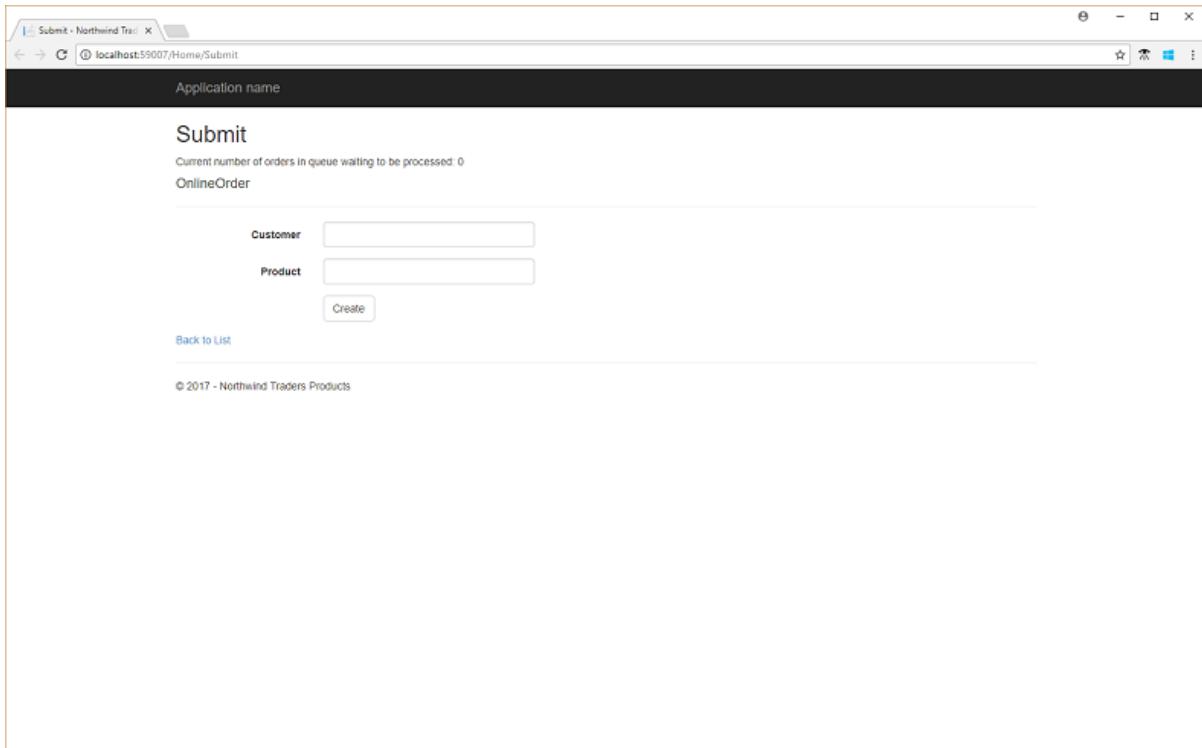
7. Click **Add**.
8. Now, change the displayed name of your application. In **Solution Explorer**, double-click the **Views\Shared_Layout.cshtml** file to open it in the Visual Studio editor.
9. Replace all occurrences of **My ASP.NET Application** with **Northwind Traders Products**.
10. Remove the **Home**, **About**, and **Contact** links. Delete the highlighted code:



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.



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.asax.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.

```
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.

Submit

Application name

Customer

Product

Create

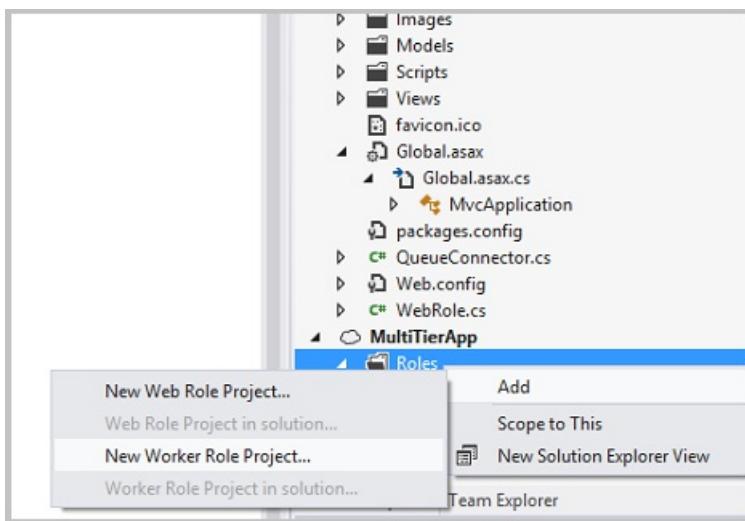
Back to List

© 2017 - Northwind Traders Products

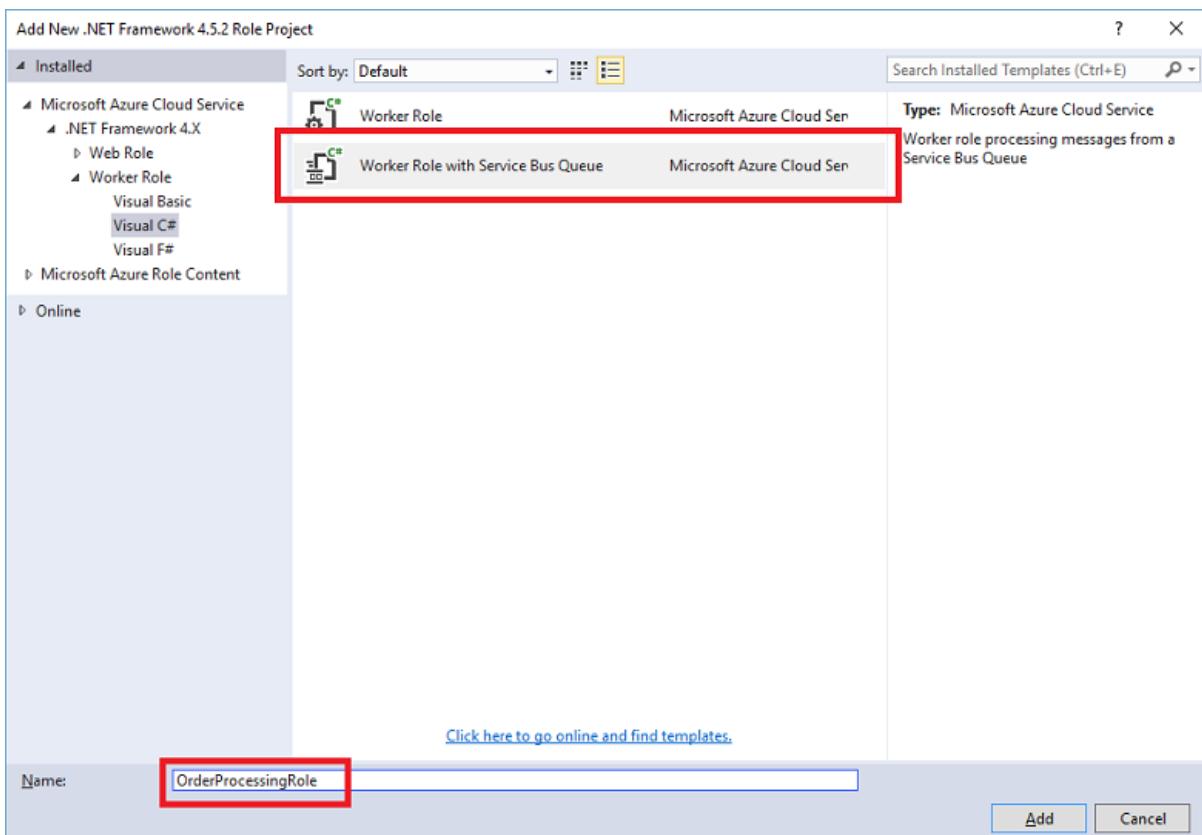
Create the worker role

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.

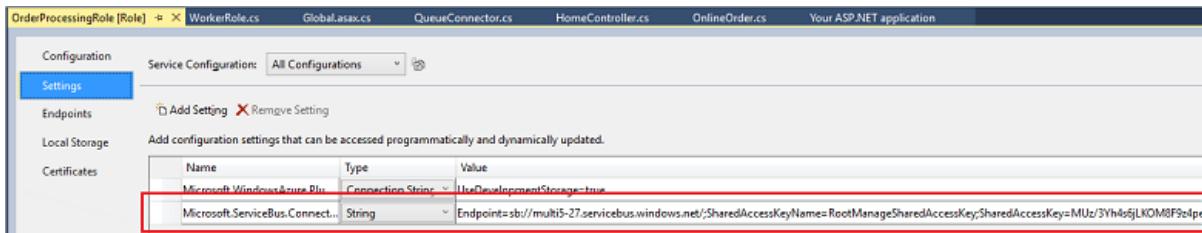
1. Make sure you have connected Visual Studio to your Azure account.
2. In Visual Studio, in **Solution Explorer** right-click the **Roles** folder under the **MultiTierApp** project.
3. Click **Add**, and then click **New Worker Role Project**. The **Add New Role Project** dialog box appears.



4. In the **Add New Role Project** dialog box, click **Worker Role with Service Bus Queue**.



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.



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";
```

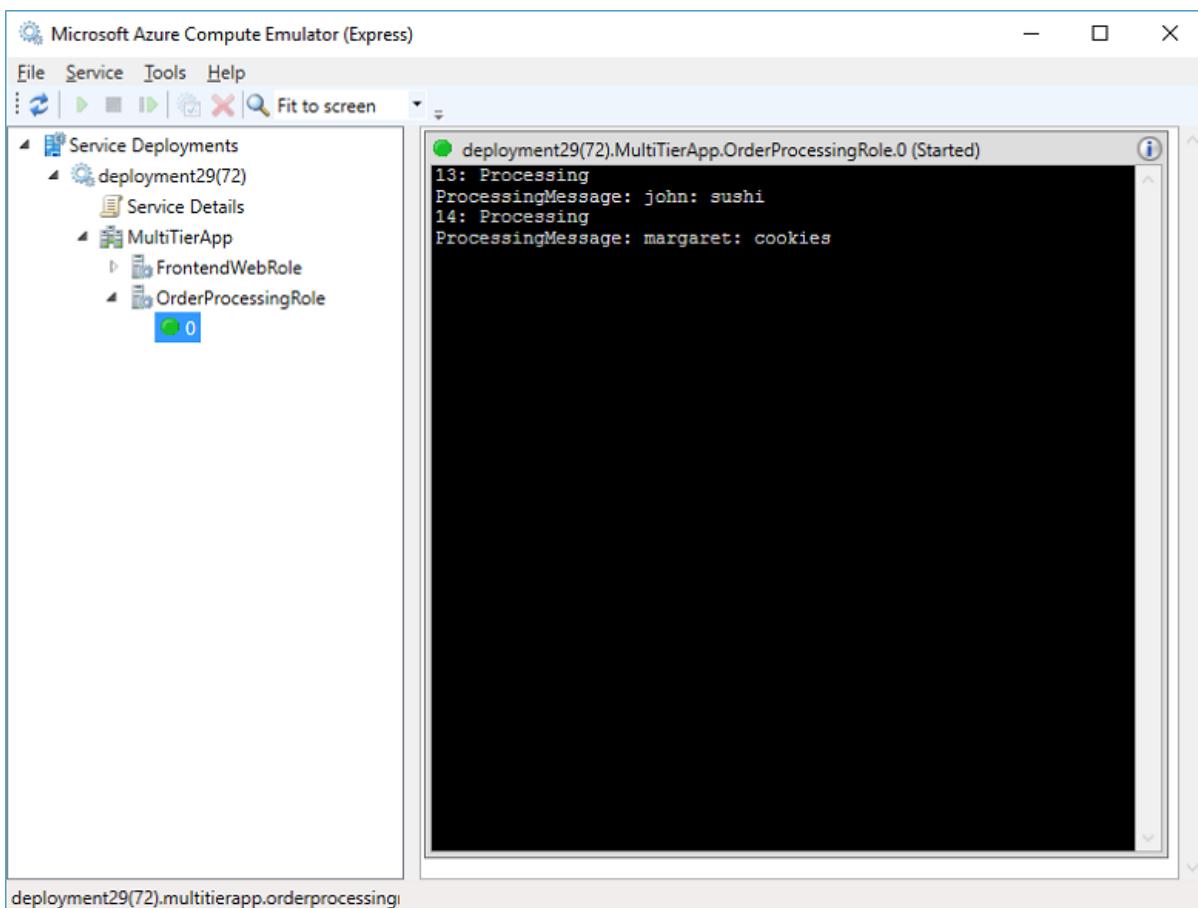
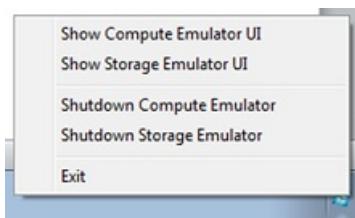
12. Add the following using statement at the top of the WorkerRole.cs file.

```
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**.



Next steps

To learn more about Service Bus, see the following resources:

- [Service Bus fundamentals](#)
- [Get started using Service Bus queues](#)
- [Service Bus service page](#)

To learn more about multi-tier scenarios, see:

- [.NET Multi-Tier Application Using Storage Tables, Queues, and Blobs](#)

Service Bus queues, topics, and subscriptions

11/7/2017 • 8 min to read • [Edit Online](#)

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 workload. 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 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 example:

```
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.

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](#)
- [Brokered Messaging: Advanced Filters sample](#)

Messages, payloads, and serialization

1/26/2018 • 9 min to read • [Edit Online](#)

Microsoft Azure Service Bus handles messages. Messages carry a payload as well as metadata, in the form of key-value pair properties, describing the payload and giving handling instructions to Service Bus and applications. Occasionally, that metadata alone is sufficient to carry the information that the sender wants to communicate to receivers, and the payload remains empty.

The object model of the official Service Bus clients for .NET and Java reflect the abstract Service Bus message structure, which is mapped to and from the wire protocols Service Bus supports.

A Service Bus message consists of a binary payload section that Service Bus never handles in any form on the service-side, and two sets of properties. The *broker properties* are predefined by the system. These predefined properties either control message-level functionality inside the broker, or they map to common and standardized metadata items. The *user properties* are a collection of key-value pairs that can be defined and set by the application.

The predefined broker properties are listed in the following table. The names are used with all official client APIs and also in the [BrokerProperties](#) JSON object of the HTTP protocol mapping.

The equivalent names used at the AMQP protocol level are listed in parentheses.

PROPERTY NAME	DESCRIPTION
ContentType (content-type)	Optional descriptor of the message payload, with a descriptor following the format of RFC2045, Section 5; for example, <code>application/json</code> .
CorrelationId (correlation-id)	Enables an application to specify a context for the message for the purposes of correlation; for example, reflecting the MessageId of a message that is being replied to.
DeadLetterSource	Only set in messages that have been dead-lettered and subsequently auto-forwarded from the dead-letter queue to another entity. Indicates the entity in which the message was dead-lettered. This property is read-only.
DeliveryCount	Number of deliveries that have been attempted for this message. The count is incremented when a message lock expires, or the message is explicitly abandoned by the receiver. This property is read-only.
EnqueuedSequenceNumber	For messages that have been auto-forwarded, this property reflects the sequence number that had first been assigned to the message at its original point of submission. This property is read-only.
EnqueuedTimeUtc	The UTC instant at which the message has been accepted and stored in the entity. This value can be used as an authoritative and neutral arrival time indicator when the receiver does not want to trust the sender's clock. This property is read-only.

PROPERTY NAME	DESCRIPTION
ExpiresAtUtc (absolute-expiry-time)	The UTC instant at which the message is marked for removal and no longer available for retrieval from the entity due to its expiration. Expiry is controlled by the TimeToLive property and this property is computed from <code>EnqueuedTimeUtc + TimeToLive</code> . This property is read-only.
ForcePersistence	For queues or topics that have the EnableExpress flag set, this property can be set to indicate that the message must be persisted to disk before it is acknowledged. This is the standard behavior for all non-express entities.
Label (subject)	This property enables the application to indicate the purpose of the message to the receiver in a standardized fashion, similar to an email subject line.
LockedUntilUtc	For messages retrieved under a lock (peek-lock receive mode, not pre-settled) this property reflects the UTC instant until which the message is held locked in the queue/subscription. When the lock expires, the DeliveryCount is incremented and the message is again available for retrieval. This property is read-only.
LockToken	The lock token is a reference to the lock that is being held by the broker in <i>peek-lock</i> receive mode. The token can be used to pin the lock permanently through the Deferral API and, with that, take the message out of the regular delivery state flow. This property is read-only.
MessageId (message-id)	The message identifier is an application-defined value that uniquely identifies the message and its payload. The identifier is a free-form string and can reflect a GUID or an identifier derived from the application context. If enabled, the duplicate detection feature identifies and removes second and further submissions of messages with the same MessageId .
PartitionKey	For partitioned entities , setting this value enables assigning related messages to the same internal partition, so that submission sequence order is correctly recorded. The partition is chosen by a hash function over this value and cannot be chosen directly. For session-aware entities, the SessionId property overrides this value.
ReplyTo (reply-to)	This optional and application-defined value is a standard way to express a reply path to the receiver of the message. When a sender expects a reply, it sets the value to the absolute or relative path of the queue or topic it expects the reply to be sent to.
ReplyToSessionId (reply-to-group-id)	This value augments the ReplyTo information and specifies which SessionId should be set for the reply when sent to the reply entity.
ScheduledEnqueueTimeUtc	For messages that are only made available for retrieval after a delay, this property defines the UTC instant at which the message will be logically enqueued, sequenced, and therefore made available for retrieval.

PROPERTY NAME	DESCRIPTION
SequenceNumber	The sequence number is a unique 64-bit integer assigned to a message as it is accepted and stored by the broker and functions as its true identifier. For partitioned entities, the topmost 16 bits reflect the partition identifier. Sequence numbers monotonically increase and are gapless. They roll over to 0 when the 48-64 bit range is exhausted. This property is read-only.
SessionId (group-id)	For session-aware entities, this application-defined value specifies the session affiliation of the message. Messages with the same session identifier are subject to summary locking and enable exact in-order processing and demultiplexing. For entities that are not session-aware, this value is ignored.
Size	Reflects the stored size of the message in the broker log as a count of bytes, as it counts towards the storage quota. This property is read-only.
State	Indicates the state of the message in the log. This property is only relevant during message browsing ("peek"), to determine whether a message is "active" and available for retrieval as it reaches the top of the queue, whether it is deferred, or is waiting to be scheduled. This property is read-only.
TimeToLive	This value is the relative duration after which the message expires, starting from the instant the message has been accepted and stored by the broker, as captured in EnqueueTimeUtc . When not set explicitly, the assumed value is the DefaultTimeToLive for the respective queue or topic. A message-level TimeToLive value cannot be longer than the entity's DefaultTimeToLive setting. If it is longer, it is silently adjusted.
To (to)	This property is reserved for future use in routing scenarios and currently ignored by the broker itself. Applications can use this value in rule-driven auto-forward chaining scenarios to indicate the intended logical destination of the message.
ViaPartitionKey	If a message is sent via a transfer queue in the scope of a transaction, this value selects the transfer queue partition.

The abstract message model enables a message to be posted to a queue via HTTP (actually always HTTPS) and can be retrieved via AMQP. In either case, the message looks normal in the context of the respective protocol. The broker properties are translated as needed, and the user properties are mapped to the most appropriate location on the respective protocol message model. In HTTP, user properties map directly to and from HTTP headers; in AMQP they map to and from the **application-properties** map.

Message routing and correlation

A subset of the broker properties described previously, specifically **To**, **ReplyTo**, **ReplyToSessionId**, **MessageId**, **CorrelationId**, and **SessionId**, are used to help applications route messages to particular destinations. To illustrate this, consider a few patterns:

- **Simple request/reply:** A publisher sends a message into a queue and expects a reply from the message consumer. To receive the reply, the publisher owns a queue into which it expects replies to be delivered. The address of that queue is expressed in the **ReplyTo** property of the outbound message. When the consumer

responds, it copies the **MessageId** of the handled message into the **CorrelationId** property of the reply message and delivers the message to the destination indicated by the **ReplyTo** property. One message can yield multiple replies, depending on the application context.

- **Multicast request/reply:** As a variation of the prior pattern, a publisher sends the message into a topic and multiple subscribers become eligible to consume the message. Each of the subscribers might respond in the fashion described previously. This pattern is used in discovery or roll-call scenarios and the respondent typically identifies itself with a user property or inside the payload. If **ReplyTo** points to a topic, such a set of discovery responses can be distributed to an audience.
- **Multiplexing:** This session feature enables multiplexing of streams of related messages through a single queue or subscription such that each session (or group) of related messages, identified by matching **SessionId** values, are routed to a specific receiver while the receiver holds the session under lock. Read more about the details of sessions [here](#).
- **Multiplexed request/reply:** This session feature enables multiplexed replies, allowing several publishers to share a reply queue. By setting **ReplyToSessionId**, the publisher can instruct the consumer(s) to copy that value into the **SessionId** property of the reply message. The publishing queue or topic does not need to be session-aware. As the message is sent, the publisher can then specifically wait for a session with the given **SessionId** to materialize on the queue by conditionally accepting a session receiver.

Routing inside of a Service Bus namespace can be realized using auto-forward chaining and topic subscription rules. Routing across namespaces can be realized [using Azure LogicApps](#). As indicated in the previous list, the **To** property is reserved for future use and may eventually be interpreted by the broker with a specially enabled feature. Applications that wish to implement routing should do so based on user properties and not lean on the **To** property; however, doing so now will not cause compatibility issues.

Payload serialization

When in transit or stored inside of Service Bus, the payload is always an opaque, binary block. The **ContentType** property enables applications to describe the payload, with the suggested format for the property values being a MIME content-type description according to IETF RFC2045; for example, `application/json; charset=utf-8`.

Unlike the Java or .NET Standard variants, the .NET Framework version of the Service Bus API supports creating **BrokeredMessage** instances by passing arbitrary .NET objects into the constructor.

When using the legacy SBMP protocol, those objects are then serialized with the default binary serializer, or with a serializer that is externally supplied. When using the AMQP protocol, the object is serialized into an AMQP object. The receiver can retrieve those objects with the **GetBody()** method, supplying the expected type. With AMQP, the objects are serialized into an AMQP graph of **ArrayList** and **IDictionary<string,object>** objects, and any AMQP client can decode them.

While this hidden serialization magic is convenient, applications should take explicit control of object serialization and turn their object graphs into streams before including them into a message, and do the reverse on the receiver side. This yields interoperable results. It should also be noted that while AMQP has a powerful binary encoding model, it is tied to the AMQP messaging ecosystem and HTTP clients will have trouble decoding such payloads.

We generally recommend JSON and Apache Avro as payload formats for structured data.

The .NET Standard and Java API variants only accept byte arrays, which means that the application must handle object serialization control.

Next steps

To learn more about Service Bus messaging, see the following topics:

- [Service Bus fundamentals](#)
- [Service Bus queues, topics, and subscriptions](#)

- [Get started with Service Bus queues](#)
- [How to use Service Bus topics and subscriptions](#)

Message transfers, locks, and settlement

1/26/2018 • 8 min to read • [Edit Online](#)

The central capability of a message broker such as Service Bus is to accept messages into a queue or topic and hold them available for later retrieval. *Send* is the term that is commonly used for the transfer of a message into the message broker. *Receive* is the term commonly used for the transfer of a message to a retrieving client.

When a client sends a message, it usually wants to know whether the message has been properly transferred to and accepted by the broker or whether some sort of error occurred. This positive or negative acknowledgment settles the client and the broker understanding about the transfer state of the message and is thus referred to as *settlement*.

Likewise, when the broker transfers a message to a client, the broker and client want to establish an understanding of whether the message has been successfully processed and can therefore be removed, or whether the message delivery or processing failed, and thus the message might have to be delivered again.

Settling send operations

Using any of the supported Service Bus API clients, send operations into Service Bus are always explicitly settled, meaning that the API operation waits for an acceptance result from Service Bus to arrive, and then completes the send operation.

If the message is rejected by Service Bus, the rejection contains an error indicator and text with a "tracking-id" inside of it. The rejection also includes information about whether the operation can be retried with any expectation of success. In the client, this information is turned into an exception and raised to the caller of the send operation. If the message has been accepted, the operation silently completes.

When using the AMQP protocol, which is the exclusive protocol for the .NET Standard client and the Java client and [which is an option for the .NET Framework client](#), message transfers and settlements are pipelined and completely asynchronous, and it is recommended that you use the asynchronous programming model API variants.

A sender can put several messages on the wire in rapid succession without having to wait for each message to be acknowledged, as would otherwise be the case with the SBMP protocol or with HTTP 1.1. Those asynchronous send operations complete as the respective messages are accepted and stored, on partitioned entities or when send operation to different entities overlap. The completions might also occur out of the original send order.

The strategy for handling the outcome of send operations can have immediate and significant performance impact for your application. The examples in this section are written in C# and apply equivalently for Java Futures.

If the application produces bursts of messages, illustrated here with a plain loop, and were to await the completion of each send operation before sending the next message, synchronous or asynchronous API shapes alike, sending 10 messages only completes after 10 sequential full round trips for settlement.

With an assumed 70 millisecond TCP roundtrip latency distance from an on-premises site to Service Bus and giving just 10 ms for Service Bus to accept and store each message, the following loop takes up at least 8 seconds, not counting payload transfer time or potential route congestion effects:

```

for (int i = 0; i < 100; i++)
{
    // creating the message omitted for brevity
    await client.SendAsync(...);
}

```

If the application starts the 10 asynchronous send operations in immediate succession and awaits their respective completion separately, the round trip time for those 10 send operations overlaps. The 10 messages are transferred in immediate succession, potentially even sharing TCP frames, and the overall transfer duration largely depends on the network-related time it takes to get the messages transferred to the broker.

Making the same assumptions as for the prior loop, the total overlapped execution time for the following loop might stay well under one second:

```

var tasks = new List<Task>();
for (int i = 0; i < 100; i++)
{
    tasks.Add(client.SendAsync(...));
}
await Task.WhenAll(tasks.ToArray());

```

It is important to note that all asynchronous programming models use some form of memory-based, hidden work queue that holds pending operations. When [SendAsync](#) (C#) or [Send](#) (Java) return, the send task is queued up in that work queue but the protocol gesture only commences once it is the task's turn to run. For code that tends to push bursts of messages and where reliability is a concern, care should be taken that not too many messages are put "in flight" at once, because all sent messages take up memory until they have factually been put onto the wire.

Semaphores, as shown in the following code snippet in C#, are synchronization objects that enable such application-level throttling when needed. This use of a semaphore allows for at most 10 messages to be in flight at once. One of the 10 available semaphore locks is taken before the send and it is released as the send completes. The 11th pass through the loop waits until at least one of the prior sends has completed, and then makes its lock available:

```

var semaphore = new SemaphoreSlim(10);

var tasks = new List<Task>();
for (int i = 0; i < 100; i++)
{
    await semaphore.WaitAsync();

    tasks.Add(client.SendAsync(...).ContinueWith((t)=>semaphore.Release()));
}
await Task.WhenAll(tasks.ToArray());

```

Applications should **never** initiate an asynchronous send operation in a "fire and forget" manner without retrieving the outcome of the operation. Doing so can load the internal and invisible task queue up to memory exhaustion, and prevent the application from detecting send errors:

```

for (int i = 0; i < 100; i++)
{
    client.SendAsync(message); // DON'T DO THIS
}

```

With a low-level AMQP client, Service Bus also accepts "pre-settled" transfers. A pre-settled transfer is a fire-and-forget operation for which the outcome, either way, is not reported back to the client and the message is considered

settled when sent. The lack of feedback to the client also means that there is no actionable data available for diagnostics, which means that this mode does not qualify for help via Azure support.

Settling receive operations

For receive operations, the Service Bus API clients enable two different explicit modes: *Receive-and-Delete* and *Peek-Lock*.

The [Receive-and-Delete](#) mode tells the broker to consider all messages it sends to the receiving client as settled when sent. That means that the message is considered consumed as soon as the broker has put it onto the wire. If the message transfer fails, the message is lost.

The upside of this mode is that the receiver does not need to take further action on the message and is also not slowed by waiting for the outcome of the settlement. If the data contained in the individual messages have low value and/or are only meaningful for a very short time, this mode is a reasonable choice.

The [Peek-Lock](#) mode tells the broker that the receiving client wants to settle received messages explicitly. The message is made available for the receiver to process, while held under an exclusive lock in the service so that other, competing receivers cannot see it. The duration of the lock is initially defined at the queue or subscription level and can be extended by the client owning the lock, via the [RenewLock](#) operation.

When a message is locked, other clients receiving from the same queue or subscription can take on locks and retrieve the next available messages not under active lock. When the lock on a message is explicitly released or when the lock expires, the message pops back up at or near the front of the retrieval order for redelivery.

When the message is repeatedly released by receivers or they let the lock elapse for a defined number of times ([maxDeliveryCount](#)), the message is automatically removed from the queue or subscription and placed into the associated dead-letter queue.

The receiving client initiates settlement of a received message with a positive acknowledgment when it calls [Complete](#) at the API level. This indicates to the broker that the message has been successfully processed and the message is removed from the queue or subscription. The broker replies to the receiver's settlement intent with a reply that indicates whether the settlement could be performed.

When the receiving client fails to process a message but wants the message to be redelivered, it can explicitly ask for the message to be released and unlocked instantly by calling [Abandon](#) or it can do nothing and let the lock elapse.

If a receiving client fails to process a message and knows that redelivering the message and retrying the operation will not help, it can reject the message, which moves it into the dead-letter queue by calling [DeadLetter](#), which also allows setting a custom property including a reason code that can be retrieved with the message from the dead-letter queue.

A special case of settlement is deferral, which is discussed in a separate article.

The [Complete](#) or [Deadletter](#) operations as well as the [RenewLock](#) operations may fail due to network issues, if the held lock has expired, or there are other service-side conditions that prevent settlement. In one of the latter cases, the service sends a negative acknowledgment that surfaces as an exception in the API clients. If the reason is a broken network connection, the lock is dropped since Service Bus does not support recovery of existing AMQP links on a different connection.

If [Complete](#) fails, which occurs typically at the very end of message handling and in some cases after minutes of processing work, the receiving application can decide whether it preserves the state of the work and ignores the same message when it is delivered a second time, or whether it tosses out the work result and retries as the message is redelivered.

The typical mechanism for identifying duplicate message deliveries is by checking the message-id, which can and

should be set by the sender to a unique value, possibly aligned with an identifier from the originating process. A job scheduler would likely set the message-id to the identifier of the job it is trying to assign to a worker with the given worker, and the worker would ignore the second occurrence of the job assignment if that job is already done.

Next steps

To learn more about Service Bus messaging, see the following topics:

- [Service Bus fundamentals](#)
- [Service Bus queues, topics, and subscriptions](#)
- [Get started with Service Bus queues](#)
- [How to use Service Bus topics and subscriptions](#)

Message sequencing and timestamps

1/26/2018 • 2 min to read • [Edit Online](#)

Sequencing and timestamping are two features that are always enabled on all Service Bus entities and surface through the [SequenceNumber](#) and [EnqueuedTimeUtc](#) properties of received or browsed messages.

For those cases in which absolute order of messages is significant and/or in which a consumer needs a trustworthy unique identifier for messages, the broker stamps messages with a gap-free, increasing sequence number relative to the queue or topic. For partitioned entities, the sequence number is issued relative to the partition.

The **SequenceNumber** value is a unique 64-bit integer assigned to a message as it is accepted and stored by the broker and functions as its internal identifier. For partitioned entities, the topmost 16 bits reflect the partition identifier. Sequence numbers roll over to zero when the 48/64 bit range is exhausted.

The sequence number can be trusted as a unique identifier since it is assigned by a central and neutral authority and not by clients. It also represents the true order of arrival, and is more precise than a time stamp as an order criterion, because time stamps may not have a high enough resolution at extreme message rates and may be subject to (however minimal) clock skew in situations where the broker ownership transitions between nodes.

The absolute arrival order matters, for example, in business scenarios in which a limited number of offered goods are served on a first-come-first-served basis while supplies last; concert ticket sales are an example.

The time-stamping capability acts as a neutral and trustworthy authority that accurately captures the UTC time of arrival of a message, reflected in the [EnqueuedTimeUtc](#) property. The value is useful if a business scenario depends on deadlines, such as whether a work item was submitted on a certain date before midnight, but the processing is far behind the queue backlog.

Scheduled messages

You can submit messages to a queue or topic for delayed processing; for example, to schedule a job to become available for processing by a system at a certain time. This capability realizes a reliable distributed time-based scheduler.

Scheduled messages do not materialize in the queue until the defined enqueue time. Before that time, scheduled messages can be canceled. Cancellation deletes the message.

You can schedule messages either by setting the [ScheduledEnqueueTimeUtc](#) property when sending a message through the regular send path, or explicitly with the [ScheduleMessageAsync](#) API. The latter immediately returns the scheduled message's **SequenceNumber**, which you can later use to cancel the scheduled message if needed. Scheduled messages and their sequence numbers can also be discovered using [message browsing](#).

The **SequenceNumber** for a scheduled message is only valid while the message is in this state. As the message transitions to the active state, the message is appended to the queue as if had been enqueued at the current instant, which includes assigning a new **SequenceNumber**.

Because the feature is anchored on individual messages and messages can only be enqueued once, Service Bus does not support recurring schedules for messages.

Next steps

To learn more about Service Bus messaging, see the following topics:

- [Service Bus fundamentals](#)

- [Service Bus queues, topics, and subscriptions](#)
- [Get started with Service Bus queues](#)
- [How to use Service Bus topics and subscriptions](#)

Message expiration (Time to Live)

1/26/2018 • 4 min to read • [Edit Online](#)

The payload inside of a message, or a command or inquiry that a message conveys to a receiver, is almost always subject to some form of application-level expiration deadline. After such a deadline, the content is no longer delivered, or the requested operation is no longer executed.

For development and test environments in which queues and topics are often used in the context of partial runs of applications or application parts, it's also desirable for stranded test messages to be automatically garbage collected so that the next test run can start clean.

The expiration for any individual message can be controlled by setting the [TimeToLive](#) system property, which specifies a relative duration. The expiration becomes an absolute instant when the message is enqueued into the entity. At that time, the [ExpiresAtUtc](#) property takes on the value ([EnqueuedTimeUtc](#) + [TimeToLive](#)).

Past the [ExpiresAtUtc](#) instant, messages become ineligible for retrieval. The expiration does not affect messages that are currently locked for delivery; those messages are still handled normally. If the lock expires or the message is abandoned, the expiration takes immediate effect.

While the message is under lock, the application might be in possession of a message that has expired. Whether the application is willing to go ahead with processing or chooses to abandon the message is up to the implementer.

Entity-level expiration

All messages sent into a queue or topic are subject to a default expiration that is set at the entity level with the [defaultMessageTimeToLive](#) property and which can also be set in the portal during creation and adjusted later. The default expiration is used for all messages sent to the entity where [TimeToLive](#) is not explicitly set. The default expiration also functions as a ceiling for the [TimeToLive](#) value. Messages that have a longer [TimeToLive](#) expiration than the default value are silently adjusted to the [defaultMessageTimeToLive](#) value before being enqueued.

Expired messages can optionally be moved to a [dead-letter queue](#) by setting the [EnableDeadLetteringOnMessageExpiration](#) property, or checking the respective box in the portal. If the option is left disabled, expired messages are dropped. Expired messages moved to the dead-letter queue can be distinguished from other dead-lettered messages by evaluating the [DeadletterReason](#) property that the broker stores in the user properties section; the value is [TTLExpiredException](#) in this case.

In the aforementioned case in which the message is protected from expiration while under lock and if the flag is set on the entity, the message is moved to the dead-letter queue as the lock is abandoned or expires. However, it is not moved if the message is successfully settled, which then assumes that the application has successfully handled it, in spite of the nominal expiration.

The combination of [TimeToLive](#) and automatic (and transactional) dead-lettering on expiry are a valuable tool for establishing confidence in whether a job given to a handler or a group of handlers under a deadline is retrieved for processing as the deadline is reached.

For example, consider a web site that needs to reliably execute jobs on a scale-constrained backend, and which occasionally experiences traffic spikes or wants to be insulated against availability episodes of that backend. In the regular case, the server-side handler for the submitted user data pushes the information into a queue and subsequently receives a reply confirming successful handling of the transaction into a reply queue. If there is a traffic spike and the backend handler cannot process its backlog items in time, the expired jobs are returned on the dead-letter queue. The interactive user can be notified that the requested operation will take a little longer than

usual, and the request can then be put on a different queue for a processing path where the eventual processing result is sent to the user by email.

Temporary entities

Service Bus queues, topics, and subscriptions can be created as temporary entities, which are automatically removed when they have not been used for a specified period of time.

Automatic cleanup is useful in development and test scenarios in which entities are created dynamically and are not cleaned up after use, due to some interruption of the test or debugging run. It is also useful when an application creates dynamic entities, such as a reply queue, for receiving responses back into a web server process, or into another relatively short-lived object where it is difficult to reliably clean up those entities when the object instance disappears.

The feature is enabled using the `autoDeleteOnIdle` property, which is set to the duration for which an entity must be idle (unused) before it is automatically deleted. The minimum duration is 5 minutes.

The **autoDeleteOnIdle** property must be set through an Azure Resource Manager operation or via the .NET Framework client `NamespaceManager` APIs. It cannot be set through the portal.

Next steps

To learn more about Service Bus messaging, see the following topics:

- [Service Bus fundamentals](#)
- [Service Bus queues, topics, and subscriptions](#)
- [Get started with Service Bus queues](#)
- [How to use Service Bus topics and subscriptions](#)

Azure Service Bus to Event Grid integration overview

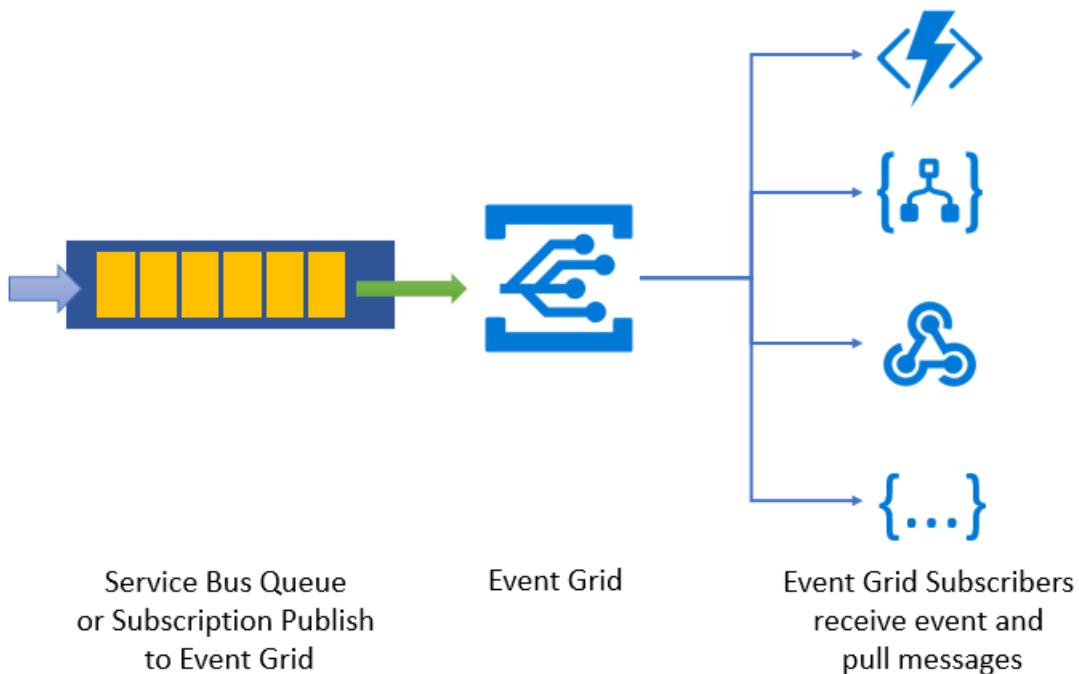
4/25/2018 • 4 min to read • [Edit Online](#)

Azure Service Bus has launched a new integration to Azure Event Grid. The key scenario of this feature is that Service Bus queues or subscriptions with a low volume of messages do not need to have a receiver that polls for messages continuously.

Service Bus can now emit events to Event Grid when there are messages in a queue or a subscription when no receivers are present. You can create Event Grid subscriptions to your Service Bus namespaces, listen to these events, and then react to the events by starting a receiver. With this feature, you can use Service Bus in reactive programming models.

To enable the feature, you need the following items:

- A Service Bus Premium namespace with at least one Service Bus queue or a Service Bus topic with at least one subscription.
- Contributor access to the Service Bus namespace.
- Additionally, you need an Event Grid subscription for the Service Bus namespace. This subscription receives a notification from Event Grid that there are messages to be picked up. Typical subscribers could be the Logic Apps feature of Azure App Service, Azure Functions, or a webhook contacting a web app. The subscriber then processes the messages.



Verify that you have contributor access

Go to your Service Bus namespace, and then select **Access control (IAM)**, as shown here:

Home > All resources > DemoNamespaceSB - Access control (IAM)

 DemoNamespaceSB - Access control (IAM)
Service Bus

Search (Ctrl+ /) << Add Remove Roles

Overview
Activity log
Access control (IAM)
Tags
Diagnose and solve problems

Name **CONTRIBUTOR**

Scope All scopes

25 items (22 Users, 3 Service Principals)

NAME

Events and event schemas

Service Bus today sends events for two scenarios:

- [ActiveMessagesWithNoListenersAvailable](#)
- [DeadletterMessagesAvailable](#)

Additionally, Service Bus uses the standard Event Grid security and [authentication mechanisms](#).

For more information, see [Azure Event Grid event schemas](#).

Active Messages Available event

This event is generated if you have active messages in a queue or a subscription and there are no receivers listening.

The schema for this event is as follows:

```
{
  "topic": "/subscriptions/<subscription id>/resourcegroups/DemoGroup/providers/Microsoft.ServiceBus/namespaces/<YOUR SERVICE BUS NAMESPACE WILL SHOW HERE>",
  "subject": "topics/<service bus topic>/subscriptions/<service bus subscription>",
  "eventType": "Microsoft.ServiceBus.ActiveMessagesAvailableWithNoListeners",
  "eventTime": "2018-02-14T05:12:53.4133526Z",
  "id": "dede87b0-3656-419c-acaf-70c95ddc60f5",
  "data": {
    "namespaceName": "YOUR SERVICE BUS NAMESPACE WILL SHOW HERE",
    "requestUri": "https://YOUR-SERVICE-BUS-NAMESPACE-WILL-SHOW-HERE.servicebus.windows.net/TOPIC-NAME/subscriptions/SUBSCRIPTIONNAME/messages/head",
    "entityType": "subscriber",
    "queueName": "QUEUE NAME IF QUEUE",
    "topicName": "TOPIC NAME IF TOPIC",
    "subscriptionName": "SUBSCRIPTION NAME"
  },
  "dataVersion": "1",
  "metadataVersion": "1"
}
```

Dead-letter Messages Available event

You get at least one event per Dead Letter queue, which has messages and no active receivers.

The schema for this event is as follows:

```
[{
  "topic": "/subscriptions/<subscription
id>/resourcegroups/DemoGroup/providers/Microsoft.ServiceBus/namespaces/<YOUR SERVICE BUS NAMESPACE WILL SHOW
HERE>",
  "subject": "topics/<service bus topic>/subscriptions/<service bus subscription>",
  "eventType": "Microsoft.ServiceBus.DeadletterMessagesAvailableWithNoListener",
  "eventTime": "2018-02-14T05:12:53.4133526Z",
  "id": "dede87b0-3656-419c-acaf-70c95ddc60f5",
  "data": {
    "namespaceName": "YOUR SERVICE BUS NAMESPACE WILL SHOW HERE",
    "requestUri": "https://YOUR-SERVICE-BUS-NAMESPACE-WILL-SHOW-HERE.servicebus.windows.net/TOPIC-
NAME/subscriptions/SUBSCRIPTIONNAME/$deadletterqueue/messages/head",
    "entityType": "subscriber",
    "queueName": "QUEUE NAME IF QUEUE",
    "topicName": "TOPIC NAME IF TOPIC",
    "subscriptionName": "SUBSCRIPTION NAME"
  },
  "dataVersion": "1",
  "metadataVersion": "1"
}]
```

How many events are emitted, and how often?

If you have multiple queues and topics or subscriptions in the namespace, you get at least one event per queue and one per subscription. The events are emitted immediately if there are no messages in the Service Bus entity and a new message arrives. Or the events are emitted every two minutes unless Service Bus detects an active receiver. Message browsing does not interrupt the events.

By default, Service Bus emits events for all entities in the namespace. If you want to get events for specific entities only, see the next section.

Use filters to limit where you get events from

If you want to get events only from, for example, one queue or one subscription within your namespace, you can use the *Begins with* or *Ends with* filters that are provided by Event Grid. In some interfaces, the filters are called *Pre* and *Suffix* filters. If you want to get events for multiple, but not all, queues and subscriptions, you can create multiple Event Grid subscriptions and provide a filter for each.

Create Event Grid subscriptions for Service Bus namespaces

You can create Event Grid subscriptions for Service Bus namespaces in three different ways:

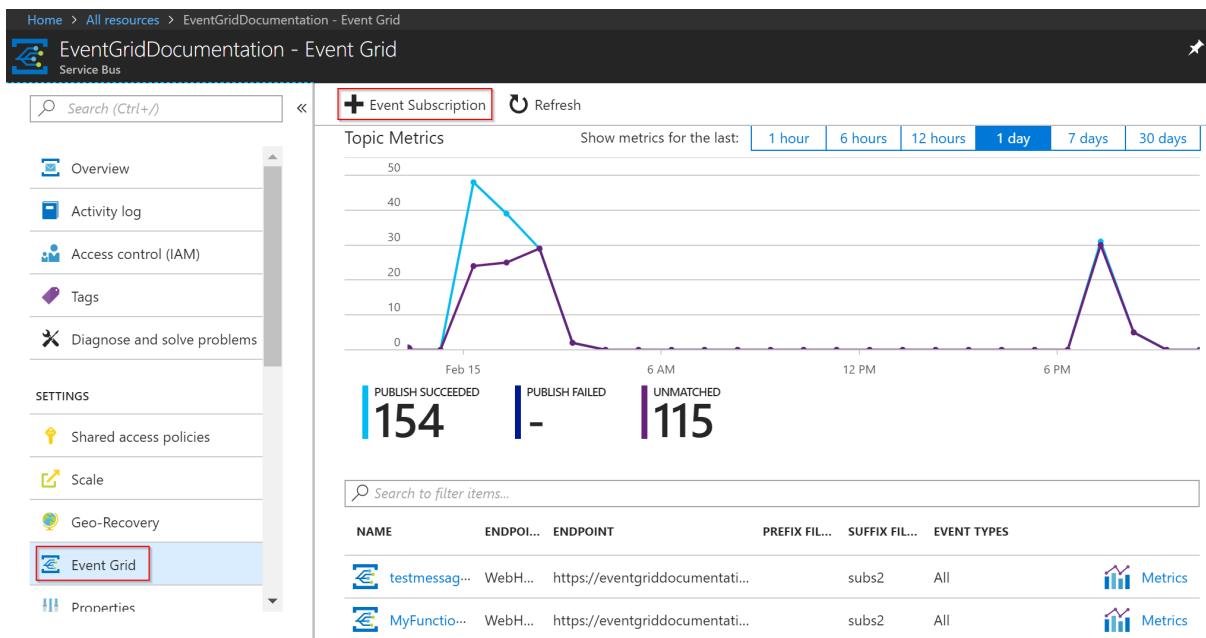
- In the [Azure portal](#)
- In [Azure CLI](#)
- In [PowerShell](#)

Azure portal instructions

To create a new Event Grid subscription, do the following:

1. In the Azure portal, go to your namespace.
2. In the left pane, select the **Event Grid**.
3. Select **Event Subscription**.

The following image displays a namespace that has a few Event Grid subscriptions:



The following image shows how to subscribe to a function or a webhook without any specific filtering:

*

Name

AnotherSubscription



Subscribe to all event types

Subscriber Type

Web Hook



*

Subscriber Endpoint

https://eventgriddocumentation.azurewe... ✓



Prefix Filter

Sample-workitems/{name}

Optional

Suffix Filter

.jpg

Optional

Create

Azure CLI instructions

First, make sure that you have Azure CLI version 2.0 or later installed. [Download the installer](#). Select **Windows + X**, and then open a new PowerShell console with administrator permissions. Alternatively, you can use a command shell within the Azure portal.

Execute the following code:

```
az login

az account set -s "THE SUBSCRIPTION YOU WANT TO USE"

$namespaceid=(az resource show --namespace Microsoft.ServiceBus --resource-type namespaces --name "<yourNamespace>" --resource-group "<Your Resource Group Name>" --query id --output tsv)

az eventgrid event-subscription create --resource-id $namespaceid --name "<YOUR EVENT GRID SUBSCRIPTION NAME (CAN BE ANY NOT EXISTING)>" --endpoint "<your_function_url>" --subject-ends-with "<YOUR SERVICE BUS SUBSCRIPTION NAME>"
```

PowerShell instructions

Make sure you have Azure PowerShell installed. [Download the installer](#). Select **Windows + X**, and then open a new PowerShell console with Administrator permissions. Alternatively, you can use a command shell within the Azure portal.

```
Connect-AzureRmAccount

Select-AzureRmSubscription -SubscriptionName "<YOUR SUBSCRIPTION NAME>

# This might be installed already
Install-Module AzureRM.ServiceBus

$NSID = (Get-AzureRmServiceBusNamespace -ResourceGroupName "<YOUR RESOURCE GROUP NAME>" -Name "<YOUR NAMESPACE NAME>").Id

New-AzureRmEventGridSubscription -EventSubscriptionName "<YOUR EVENT GRID SUBSCRIPTION NAME (CAN BE ANY NOT EXISTING)>" -ResourceId $NSID -Endpoint "<YOUR FUNCTION URL>" -SubjectEndsWith "<YOUR SERVICE BUS SUBSCRIPTION NAME>"
```

From here, you can explore the other setup options or [test that events are flowing](#).

Next steps

- Get Service Bus and Event Grid [examples](#).
- Learn more about [Event Grid](#).
- Learn more about [Azure Functions](#).
- Learn more about [Logic Apps](#).
- Learn more about [Service Bus](#).

Azure Service Bus to Azure Event Grid integration examples

4/25/2018 • 5 min to read • [Edit Online](#)

In this article, you learn how to set up an Azure function and a logic app, which both receive messages based on receiving an event from Azure Event Grid. You'll do the following:

- Create a simple [test Azure function](#) for debugging and viewing the initial flow of events from the Event Grid. Perform this step regardless of whether you perform the others.
- Create an [Azure function to receive and process Azure Service Bus messages](#) based on Event Grid events.
- Utilize the [The Logic Apps feature of Azure App Service](#).

The example that you create assumes that the Service Bus topic has two subscriptions. The example also assumes that the Event Grid subscription was created to send events for only one Service Bus subscription.

In the example, you send messages to the Service Bus topic and then verify that the event has been generated for this Service Bus subscription. The function or logic app receives the messages from the Service Bus subscription and then completes it.

Prerequisites

Before you begin, make sure that you have completed the steps in the next two sections.

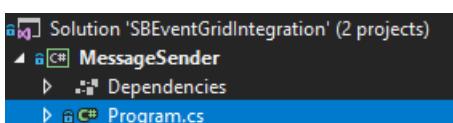
Create a Service Bus namespace

Create a Service Bus Premium namespace, and create a Service Bus topic that has two subscriptions.

Send a message to the Service Bus topic

You can use any method to send a message to your Service Bus topic. The sample code at the end of this procedure assumes that you are using Visual Studio 2017.

1. Clone [the GitHub azure-service-bus repository](#).
2. In Visual Studio, go to the `\samples\DotNet\Microsoft.ServiceBus.Messaging\ServiceBusEventGridIntegration` folder, and then open the `SBEVENTGRIDINTEGRATION.sln` file.
3. Go to the **MessageSender** project, and then select **Program.cs**.



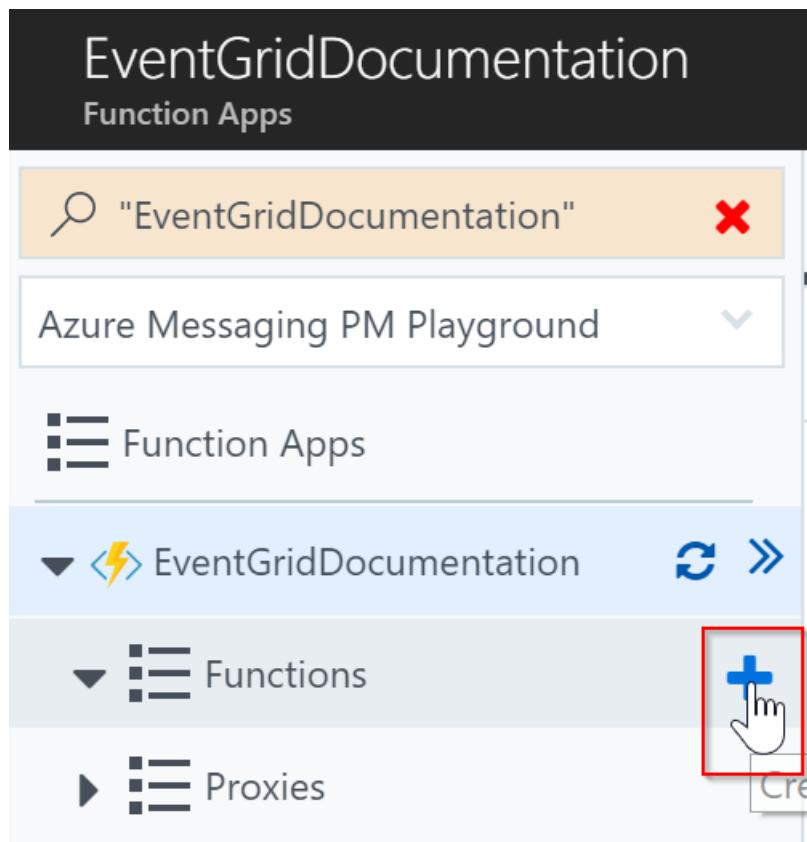
4. Fill in your topic name and connection string, and then execute the following console application code:

```
const string ServiceBusConnectionString = "YOUR CONNECTION STRING";
const string TopicName = "YOUR TOPIC NAME";
```

Set up a test function

Before you work through the entire scenario, set up at least a small test function, which you can use to debug and observe what events are flowing.

1. In the Azure portal, create a new Azure Functions application. To learn the basics of Azure Functions, see [Azure Functions documentation](#).
2. In your newly created function, select the plus sign (+) to add an HTTP trigger function:



The **Get started quickly with a premade function** window opens.



Get started quickly with a premade function

1. Choose a scenario

</>

Webhook + API



Timer



Data processing

2. Choose a language

CSharp

JavaScript

FSharp

Java

For PowerShell, Python, and Batch, [create your own custom function](#).

Create this function

3. Select the **Webhook + API** button, select **CSharp**, and then select **Create this function**.

4. Into the function, paste the following code:

```

#r "Newtonsoft.Json"
using System.Net;
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;

public static async Task<HttpResponseMessage> Run(HttpRequestMessage req, TraceWriter log)
{
    log.Info("C# HTTP trigger function processed a request.");
    // parse query parameter
    var content = req.Content;

    string jsonContent = await content.ReadAsStringAsync();
    log.Info($"Received Event with payload: {jsonContent}");

    IEnumerable<string> headerValues;
    if (req.Headers.TryGetValues("Aeg-Event-Type", out headerValues))
    {
        var validationHeaderValue = headerValues.FirstOrDefault();
        if(validationHeaderValue == "SubscriptionValidation")
        {
            var events = JsonConvert.DeserializeObject<GridEvent[]>(jsonContent);
            var code = events[0].Data["validationCode"];
            return req.CreateResponse(HttpStatusCode.OK,
                new { validationResponse = code });
        }
    }

    return jsonContent == null
        ? req.CreateResponse(HttpStatusCode.BadRequest, "Pass a name on the query string or in the request
body")
        : req.CreateResponse(HttpStatusCode.OK, "Hello " + jsonContent);
}

public class GridEvent
{
    public string Id { get; set; }
    public string EventType { get; set; }
    public string Subject { get; set; }
    public DateTime EventTime { get; set; }
    public Dictionary<string, string> Data { get; set; }
    public string Topic { get; set; }
}

```

5. Select **Save and run**.

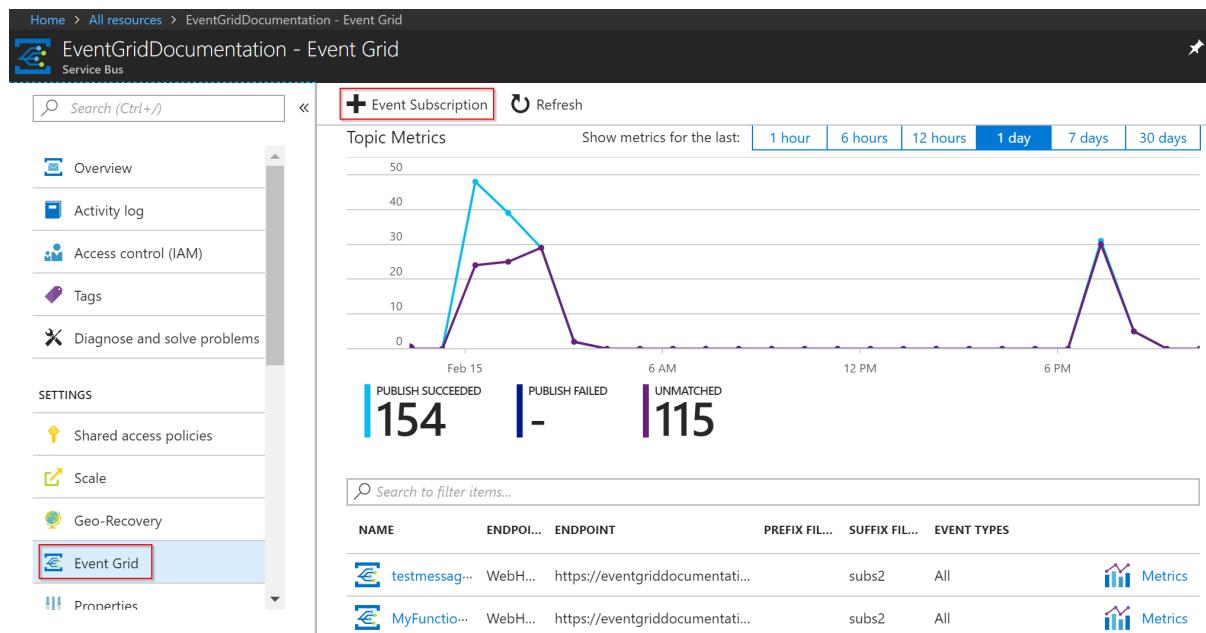
Connect the function and namespace via Event Grid

In this section, you tie together the function and the Service Bus namespace. For this example, use the Azure portal.

To understand how to use PowerShell or Azure CLI to do this procedure, see [Azure Service Bus to Azure Event Grid integration overview](#).

To create an Azure Event Grid subscription, do the following:

1. In the Azure portal, go to your namespace and then, in the left pane, select **Event Grid**. Your namespace window opens, with two Event Grid subscriptions displayed in the right pane.



2. Select **Event Subscription**.

The **Event Subscription** window opens. The following image displays a form for subscribing to an Azure function or a webhook without applying filters.

* Name

✓

Subscribe to all event types

Subscriber Type

▼

* Subscriber Endpoint

✓

Prefix Filter

Optional

Suffix Filter

Optional

Create

3. Complete the form as shown and, in the **Suffix Filter** box, remember to enter the relevant filter.
4. Select **Create**.
5. Send a message to your Service Bus topic, as mentioned in the "Prerequisites" section, and then verify that events are flowing via the Azure Functions Monitoring feature.

The next step is to tie together the function and the Service Bus namespace. For this example, use the Azure portal. To understand how to use PowerShell or Azure CLI to perform this step, see [Azure Service Bus to Azure Event Grid integration overview](#).

The screenshot shows the Azure portal interface for an Azure Function named "HttpTriggerCSharp1". On the left, there's a sidebar with navigation links like Home, All resources, Function Apps, Functions, and Monitor (which is highlighted with a red box). The main area has two sections: "Invocation log" and "Invocation details".

Invocation log:

Function	Status	Details: Last ran (duration)
HttpTriggerCSharp1 (Method: POST, Uri: ...)	✓	a minute ago (31 ms)
HttpTriggerCSharp1 (Method: POST, Uri: ...)	✓	an hour ago (16 ms)
HttpTriggerCSharp1 (Method: POST, Uri: ...)	✓	an hour ago (31 ms)
HttpTriggerCSharp1 (Method: POST, Uri: ...)	✓	2 hours ago (16 ms)
HttpTriggerCSharp1 (Method: POST, Uri: ...)	✓	3 hours ago (31 ms)
HttpTriggerCSharp1 (Method: POST, Uri: ...)	✓	3 hours ago (391 ms)
HttpTriggerCSharp1 (Method: POST, Uri: ...)	✓	3 hours ago (31 ms)
HttpTriggerCSharp1 (Method: POST, Uri: ...)	✗	3 hours ago (63 ms)

Invocation details:

- Parameter: req Method: POST, Uri: https://[REDACTED].azurewebsites.net/HttpTriggerCSharp1
- Parameter: log
- Parameter: _context 5f09d2df-e2d2-4a34-b978-a275dd7
- Parameter: \$return response

Logs:

```

https://eventgriddocumentation.servicebus.windows.net/mytopic/subscriptions/subs2/messages/head,
{
  "entityType": "subscriber",
  "queueName": null,
  "topicName": "[REDACTED]",
  "subscriptionName": "[REDACTED]"
},
"dataVersion": "1",
"metadataVersion": "1"
```

```

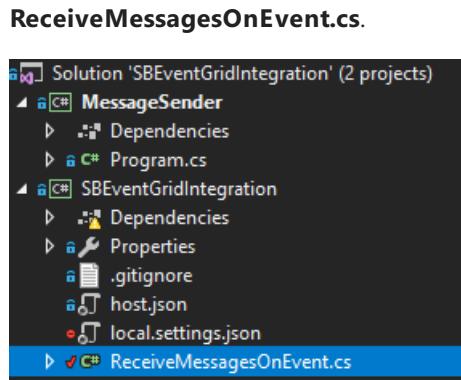
## Receive messages by using Azure Functions

In the preceding section, you observed a simple test and debugging scenario and ensured that events are flowing.

In this section, you'll learn how to receive and process messages after you receive an event.

You'll add an Azure function, as shown in the following example, because the Service Bus functions within Azure Functions do not yet natively support the new Event Grid integration.

1. In the same Visual Studio Solution that you opened in the prerequisites, select



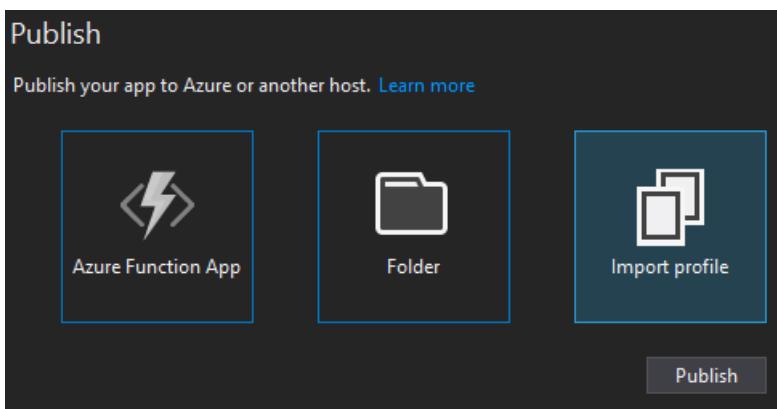
2. Enter your connection string in the following code:

```
const string ServiceBusConnectionString = "YOUR CONNECTION STRING";
```

3. In the Azure portal, download the publishing profile for the Azure function that you created in the "Set up a test function" section.

The screenshot shows the Azure portal's Function Apps overview page. On the left, there's a search bar with the text "EventGridDocumentation" and a red 'X' button. Below it is a list of function apps, with "EventGridDocumentation" expanded to show its functions. Under "Functions", "HttpTriggerCSharp1" is listed. On the right, the "Overview" tab is selected, showing the status as "Running" with a green checkmark. The "Platform features" tab is also visible. At the top right, there are buttons for "Stop", "Swap", "Restart", and "Download publish profile", with the "Download publish profile" button highlighted by a red box.

4. In Visual Studio, right-click **SBEventGridIntegration**, and then select **Publish**.
5. In the **Publish** pane for the publishing profile that you downloaded previously, select **Import profile**, and then select **Publish**.



6. After you've published the new Azure function, create a new Azure Event Grid subscription that points to the new Azure function.  
In the **Ends with** box, be sure to apply the correct filter, which should be your Service Bus subscription name.
7. Send a message to the Azure Service Bus topic that you created previously, and then monitor the Azure Functions log in the Azure portal to ensure that events are flowing and that messages are being received.

The screenshot shows the Azure portal's Function Apps overview for "EventGridDocumentation". The left sidebar has a "Monitor" button highlighted with a red box. The main area displays the "Invocation log" for the "ReceiveMessagesOnEvent" function. It shows two entries: "ReceiveMessagesOnEvent (Method: POST, Uri: ...)" with a status of "Success count since Feb 1st" (1) and "Invocation log" (Refresh). Below this is a table with columns "Function", "Status", and "Details: Last ran (duration)". The first entry has a status of "Success" and a duration of "a few seconds ago (1,625 ms)". The second entry also has a status of "Success" and a duration of "2 minutes ago (781 ms)".

## Receive messages by using Logic Apps

Connect a logic app with Azure Service Bus and Azure Event Grid by doing the following:

1. Create a new logic app in the Azure portal, and select **Event Grid** as the start action.



The Logic Apps designer window opens.

A screenshot of the 'When a resource event occurs (Preview)' configuration screen. It shows several input fields:

- \* Subscription: A dropdown menu with a blacked-out placeholder.
- \* Resource Type: A dropdown menu set to 'Microsoft.ServiceBus.Namespaces' with an 'X' button to clear it.
- \* Resource Name: A dropdown menu set to 'EventGridDocumentation' with a downward arrow.
- Prefix Filter: A text input field containing 'A filter like: Sample-workitems/{name}'.
- Suffix Filter: A text input field containing 'subs2'.
- Subscription Name: A text input field containing 'Name to use for the new Event Grid subscription.'

2. Add your information by doing the following:

- a. In the **Resource Name** box, enter your own namespace name.
- b. Under **Advanced options**, in the **Suffix Filter** box, enter filter for your subscription.

3. Add a Service Bus receive action to receive messages from a topic subscription.

The final action is shown in the following image:

A screenshot of the 'Get messages from a topic subscription (peek-lock)' configuration screen. It shows the following inputs:

- \* Topic name: A dropdown menu set to 'mytopic' with a downward arrow.
- \* Topic subscription name: A dropdown menu set to 'subs2' with a downward arrow.
- Maximum message count: A text input field set to '20'.

Below these fields is a link 'Show advanced options ▾'. At the bottom, it says 'Connected to GetMessagesFromTopic. [Change connection.](#)'

4. Add a complete event, as shown in the following image:

For each

\* Select an output from previous steps  
Body x

**Complete the message in a topic subscription**

\* Topic name  
mytopic

\* Topic subscription name  
subs2

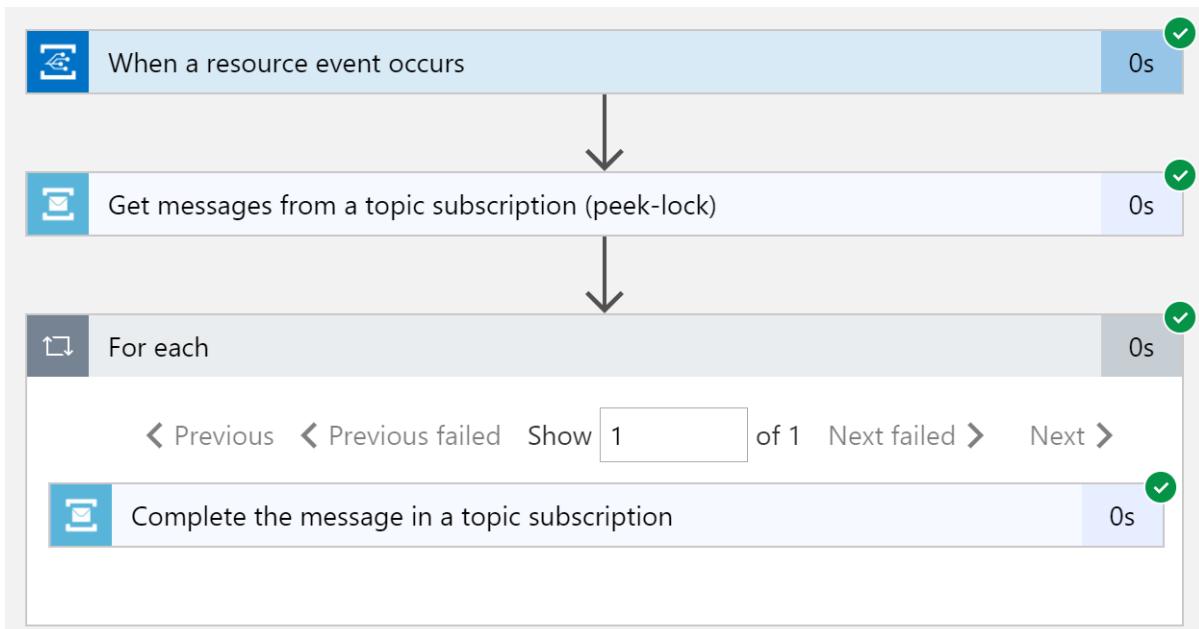
\* Lock token of the message  
Lock Token x

- Save the logic app, and send a message to your Service Bus topic, as mentioned in the "Prerequisites" section.

Observe the logic app execution. To view more data for the execution, select **Overview**, and then view the data under **Runs history**.

The screenshot shows the Azure Logic App Overview page. On the left, there's a sidebar with navigation links: Overview (highlighted with a red box), Activity log, Access control (IAM), Tags, DEVELOPMENT TOOLS (Logic App Designer), and Logic App Code View. The main area has a toolbar with Run Trigger, Refresh, Edit, Delete, and Disable buttons. Below the toolbar is a 'Runs history' section. It includes filters for Status (All), Start time (dropdown), Pick a date (calendar icon), and Pick a time. A text input field says 'Specify the run identifier to open monitor view directly' with a search icon. A table lists a single run entry: STATUS: Succ..., START TIME: 2/14/2018, 5:55..., IDENTIFIER: [REDACTED], DURATION: 800 Millis... This row is also highlighted with a red box.

| STATUS  | START TIME         | IDENTIFIER | DURATION      |
|---------|--------------------|------------|---------------|
| Succ... | 2/14/2018, 5:55... | [REDACTED] | 800 Millis... |



## Next steps

- Learn more about [Azure Event Grid](#).
- Learn more about [Azure Functions](#).
- Learn more about the [Logic Apps feature of Azure App Service](#).
- Learn more about [Azure Service Bus](#).

# Service Bus authentication and authorization

11/15/2017 • 2 min to read • [Edit Online](#)

Applications gain access to Azure Service Bus resources using Shared Access Signature (SAS) token authentication. With SAS, applications present a token to Service Bus that has been signed with a symmetric key known both to the token issuer and Service Bus (hence "shared") and that key is directly associated with a rule granting specific access rights, like the permission to receive/listen or send messages. SAS rules are either configured on the namespace, or directly on entities such as a queue or topic, allowing for fine grained access control.

SAS tokens can either be generated by a Service Bus client directly, or they can be generated by some intermediate token issuing endpoint with which the client interacts. For example, a system may require the client to call an Active Directory authorization protected web service endpoint to prove its identity and system access rights, and the web service then returns the appropriate Service Bus token. This SAS token can be easily generated using the Service Bus token provider included in the Azure SDK.

## IMPORTANT

If you are using Azure Active Directory Access Control (also known as Access Control Service or ACS) with Service Bus, note that the support for this method is now limited and you should migrate your application to use SAS. For more information, see [this blog post](#) and [this article](#).

## Shared Access Signature authentication

[SAS authentication](#) enables you to grant a user access to Service Bus resources, with specific rights. SAS authentication in Service Bus involves the configuration of a cryptographic key with associated rights on a Service Bus resource. Clients can then gain access to that resource by presenting a SAS token, which consists of the resource URI being accessed and an expiry signed with the configured key.

You can configure keys for SAS on a Service Bus namespace. The key applies to all messaging entities within that namespace. You can also configure keys on Service Bus queues and topics. SAS is also supported on [Azure Relay](#).

To use SAS, you can configure a [SharedAccessAuthorizationRule](#) object on a namespace, queue, or topic. This rule consists of the following elements:

- *KeyName*: identifies the rule.
- *PrimaryKey*: a cryptographic key used to sign/validate SAS tokens.
- *SecondaryKey*: a cryptographic key used to sign/validate SAS tokens.
- *Rights*: represents the collection of **Listen**, **Send**, or **Manage** rights granted.

Authorization rules configured at the namespace level can grant access to all entities in a namespace for clients with tokens signed using the corresponding key. You can configure up to 12 such authorization rules on a Service Bus namespace, queue, or topic. By default, a [SharedAccessAuthorizationRule](#) with all rights is configured for every namespace when it is first provisioned.

To access an entity, the client requires a SAS token generated using a specific [SharedAccessAuthorizationRule](#). The SAS token is generated using the HMAC-SHA256 of a resource string that consists of the resource URI to which access is claimed, and an expiry with a cryptographic key associated with the authorization rule.

SAS authentication support for Service Bus is included in the Azure .NET SDK versions 2.0 and later. SAS

includes support for a [SharedAccessAuthorizationRule](#). All APIs that accept a connection string as a parameter include support for SAS connection strings.

## Next steps

- Continue reading [Service Bus authentication with Shared Access Signatures](#) for more details about SAS.
- How to [migrate from Azure Active Directory Access Control \(ACS\) to Shared Access Signature authorization](#).
- [Changes To ACS Enabled namespaces](#).
- For corresponding information about Azure Relay authentication and authorization, see [Azure Relay authentication and authorization](#).

# Migrate from Azure Active Directory Access Control Service to Shared Access Signature authorization

12/21/2017 • 4 min to read • [Edit Online](#)

Service Bus applications have previously had a choice of using two different authorization models: the [Shared Access Signature \(SAS\)](#) token model provided directly by Service Bus, and a federated model where the management of authorization rules is managed inside by the [Azure Active Directory](#) Access Control Service (ACS), and tokens obtained from ACS are passed to Service Bus for authorizing access to the desired features.

The ACS authorization model has long been superseded by [SAS authorization](#) as the preferred model, and all documentation, guidance, and samples exclusively use SAS today. Moreover, it is no longer possible to create new Service Bus namespaces that are paired with ACS.

SAS has the advantage in that it is not immediately dependent on another service, but can be used directly from a client without any intermediaries by giving the client access to the SAS rule name and rule key. SAS can also be easily integrated with an approach in which a client has to first pass an authorization check with another service and then is issued a token. The latter approach is similar to the ACS usage pattern, but enables issuing access tokens based on application-specific conditions that are difficult to express in ACS.

For all existing applications that are dependent on ACS, we urge customers to migrate their applications to rely on SAS instead.

## Migration scenarios

ACS and Service Bus are integrated through the shared knowledge of a *signing key*. The signing key is used by an ACS namespace to sign authorization tokens, and it's used by Service Bus to verify that the token has been issued by the paired ACS namespace. The ACS namespace holds service identities and authorization rules. The authorization rules define which service identity or which token issued by an external identity provider gets which type of access to a part of the Service Bus namespace graph, in the form of a longest-prefix match.

For example, an ACS rule might grant the **Send** claim on the path prefix `/` to a service identity, which means that a token issued by ACS based on that rule grants the client rights to send to all entities in the namespace. If the path prefix is `/abc`, the identity is restricted to sending to entities named `abc` or organized beneath that prefix. It is assumed that readers of this migration guidance are already familiar with these concepts.

The migration scenarios fall into three broad categories:

1. **Unchanged defaults.** Some customers use a [SharedSecretTokenProvider](#) object, passing the automatically generated **owner** service identity and its secret key for the ACS namespace, paired with the Service Bus namespace, and do not add new rules.
2. **Custom service identities with simple rules.** Some customers add new service identities and grant each new service identity **Send**, **Listen**, and **Manage** permissions for one specific entity.
3. **Custom service identities with complex rules.** Very few customers have complex rule sets in which externally issued tokens are mapped to rights on Relay, or where a single service identity is assigned differentiated rights on several namespace paths through multiple rules.

For assistance with the migration of complex rule sets, you can contact [Azure support](#). The other two scenarios enable straightforward migration.

### Unchanged defaults

If your application has not changed ACS defaults, you can replace all [SharedSecretTokenProvider](#) usage with a [SharedAccessSignatureTokenProvider](#) object, and use the namespace preconfigured [RootManageSharedAccessKey](#) instead of the ACS **owner** account. Note that even with the ACS **owner** account, this configuration was (and still is) not generally recommended, because this account/rule provides full management authority over the namespace, including permission to delete any entities.

### Simple rules

If the application uses custom service identities with simple rules, the migration is straightforward in the case where an ACS service identity was created to provide access control on a specific queue. This scenario is often the case in SaaS-style solutions where each queue is used as a bridge to a tenant site or branch office, and the service identity is created for that particular site. In this case, the respective service identity can be migrated to a Shared Access Signature rule, directly on the queue. The service identity name can become the SAS rule name and the service identity key can become the SAS rule key. The rights of the SAS rule are then configured equivalent to the respectively applicable ACS rule for the entity.

You can make this new and additional configuration of SAS in-place on any existing namespace that is federated with ACS, and the migration away from ACS is subsequently performed by using [SharedAccessSignatureTokenProvider](#) instead of [SharedSecretTokenProvider](#). The namespace does not need to be unlinked from ACS.

### Complex rules

SAS rules are not meant to be accounts, but are named signing keys associated with rights. As such, scenarios in which the application creates many service identities and grants them access rights for several entities or the whole namespace still require a token-issuing intermediary. You can obtain guidance for such an intermediary by [contacting support](#).

## Next steps

To learn more about Service Bus authentication, see the following topics:

- [Service Bus authentication and authorization](#)
- [Service Bus authentication with Shared Access Signatures](#)
- [Service Bus fundamentals](#)

# Service Bus access control with Shared Access Signatures

4/30/2018 • 14 min to read • [Edit Online](#)

*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 guards access to Service Bus based on authorization rules. Those are configured either on a namespace, or a messaging entity (relay, queue, or topic). An authorization rule has a name, is associated with specific rights, and carries a pair of cryptographic keys. You use the rule's name and key via the Service Bus SDK or in your own code to generate a SAS token. A client can then pass the token to Service Bus to prove authorization for the requested operation.

## Overview of SAS

Shared Access Signatures are a claims-based authorization mechanism using simple tokens. Using SAS, keys are never passed on the wire. Keys are used to cryptographically sign information that can later be verified by the service. SAS can be used similar to a username and password scheme where the client is in immediate possession of an authorization rule name and a matching key. SAS can also be used similar to a federated security model, where the client receives a time-limited and signed access token from a security token service without ever coming into possession of the signing key.

SAS authentication in Service Bus is configured with named [Shared Access Authorization Rules](#) having associated access rights, and a pair of primary and secondary cryptographic keys. The keys are 256-bit values in Base64 representation. You can configure rules at the namespace level, on Service Bus [relays](#), [queues](#), and [topics](#).

The [Shared Access Signature](#) token contains the name of the chosen authorization rule, the URI of the resource that shall be accessed, an expiry instant, and an HMAC-SHA256 cryptographic signature computed over these fields using either the primary or the secondary cryptographic key of the chosen authorization rule.

## Shared Access Authorization Policies

Each Service Bus namespace and each Service Bus entity has a Shared Access Authorization policy made up of rules. The policy at the namespace level applies to all entities inside the namespace, irrespective of their individual policy configuration.

For each authorization policy rule, you decide on three pieces of information: **name**, **scope**, and **rights**. 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 rights conferred by the policy rule can be a combination of:

- 'Send' - Confers the right to send messages to the entity
- 'Listen' - Confers the right to listen (relay) or receive (queue, subscriptions) and all related message handling
- 'Manage' - Confers the right to manage the topology of the namespace, including creating and deleting entities

The 'Manage' right includes the 'Send' and 'Receive' rights.

A namespace or entity policy can hold up to 12 Shared Access Authorization rules, providing room for three sets

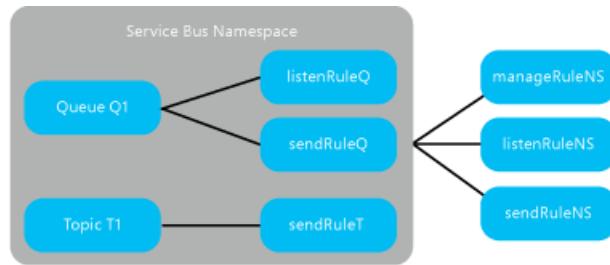
of rules, each covering the basic rights and the combination of Send and Listen. This limit underlines that the SAS policy store is not intended to be a user or service account store. If your application needs to grant access to Service Bus based on user or service identities, it should implement a security token service that issues SAS tokens after an authentication and access check.

An authorization rule 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. If you regenerate or change a key in the policy, all previously issued tokens based on that key become instantly invalid. However, ongoing connections created based on such tokens will continue to work until the token expires.

When you create a Service Bus namespace, a policy rule named **RootManageSharedAccessKey** is automatically created for the namespace. This policy has Manage permissions for the entire namespace. It's recommended that you treat this rule like an administrative **root** account and don't use it in your application. You can create additional policy rules in the **Configure** tab for the namespace in the portal, via Powershell or Azure CLI.

## 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.



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.

## Generate a Shared Access Signature token

Any client that has access to name of an authorization rule name and one of its signing keys can generate a SAS token. The token is generated by crafting a string in the following format:

```
SharedAccessSignature sig=<signature-string>&se=<expiry>&skn=<keyName>&sr=<URL-encoded-resourceURI>
```

- `se` - Token expiry instant. Integer reflecting seconds since the epoch `00:00:00 UTC` on 1 January 1970 (UNIX epoch) when the token expires.
- `skn` - Name of the authorization rule.
- `sr` - URI of the resource being accessed.
- `sig` - Signature.

The `signature-string` is the SHA-256 hash computed over the resource URI (**scope** as described in the previous section) and the string representation of the token expiry instant, separated by CRLF.

The hash computation looks similar to the following pseudo code and returns a 256-bit/32-byte hash value.

```
SHA-256('https://<yournamespace>.servicebus.windows.net/'+'\n'+ 1438205742)
```

The token contains the non-hashed values so that the recipient can recompute the hash with the same parameters, verifying that the issuer is in possession of a valid signing key.

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 URI must be [percent-encoded](#).

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 prefixed with the `<resourceURI>` used in the `signature-string`.

## Regenerating keys

It is recommended that you periodically regenerate the keys used in the [SharedAccessAuthorizationRule](#) object. The primary and secondary key slots exist so that you can rotate keys gradually. If your application generally uses the primary key, you can copy the primary key into the secondary key slot, and only then regenerate the primary key. The new primary key value can then be configured into the client applications, which have continued access using the old primary key in the secondary slot. Once all clients are updated, you can regenerate the secondary key to finally retire the old primary key.

If you know or suspect that 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.

## Shared Access Signature authentication with Service Bus

The scenarios described as follows 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 an entity

With Service Bus .NET Framework libraries, 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. Alternative to the usage shown here, you can also pass a previously issued token to the token provider factory method.

```

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);

```

You can also use the token provider directly for issuing tokens to pass to other clients.

Connection strings can include a rule name (*SharedAccessKeyName*) and rule key (*SharedAccessKey*) or a previously issued token (*SharedAccessSignature*). When those are present in the connection string passed to any constructor or factory method accepting a connection string, the SAS token provider is automatically created and populated.

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 are 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 [AMQPNet 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://{}/{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 ANONYMOUS** (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             |
|----------------------------------------------------------------------------|----------------|-------------------------|
| <b>Namespace</b>                                                           |                |                         |
| Configure authorization rule on a namespace                                | Manage         | Any namespace address   |
| <b>Service Registry</b>                                                    |                |                         |
| 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   |
| <b>Queue</b>                                                               |                |                         |
| 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                 |
| Schedule a message for later delivery; for example, <a href="#">ScheduleMessageAsync()</a> | Listen         | Any valid queue address                 |
| <b>Topic</b>                                                                               |                |                                         |
| 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                 |
| <b>Subscription</b>                                                                        |                |                                         |
| 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 |
| <b>Rules</b>                                                                               |                |                                         |

| OPERATION       | CLAIM REQUIRED   | CLAIM SCOPE                                   |
|-----------------|------------------|-----------------------------------------------|
| Create a rule   | Manage           | ../myTopic/Subscriptions/mySubscription       |
| 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](#)

# Topic filters and actions

1/26/2018 • 3 min to read • [Edit Online](#)

Subscribers can define which messages they want to receive from a topic. These messages are specified in the form of one or more named subscription rules. Each rule consists of a condition that selects particular messages and an action that annotates the selected message. For each matching rule condition, the subscription produces a copy of the message, which may be differently annotated for each matching rule.

Each newly created topic subscription has an initial default subscription rule. If you don't explicitly specify a filter condition for the rule, the applied filter is the **true** filter that enables all messages to be selected into the subscription. The default rule has no associated annotation action.

Service Bus supports three filter conditions:

- *Boolean filters* - The **TrueFilter** and **FalseFilter** either cause all arriving messages (**true**) or none of the arriving messages (**false**) to be selected for the subscription.
- *SQL Filters* - A **SqlFilter** holds a SQL-like conditional expression that is evaluated in the broker against the arriving messages' user-defined properties and system properties. All system properties must be prefixed with `sys.` in the conditional expression. The [SQL-language subset for filter conditions](#) tests for the existence of properties (EXISTS), as well as for null-values (IS NULL), logical NOT/AND/OR, relational operators, simple numeric arithmetic, and simple text pattern matching with LIKE.
- *Correlation Filters* - A **CorrelationFilter** holds a set of conditions that are matched against one or more of an arriving message's user and system properties. A common use is to match against the **CorrelationId** property, but the application can also choose to match against **ContentType**, **Label**, **MessageId**, **ReplyTo**, **ReplyToSessionId**, **SessionId**, **To**, and any user-defined properties. A match exists when an arriving message's value for a property is equal to the value specified in the correlation filter. For string expressions, the comparison is case-sensitive. When specifying multiple match properties, the filter combines them as a logical AND condition, meaning for the filter to match, all conditions must match.

All filters evaluate message properties. Filters cannot evaluate the message body.

Complex filter rules require processing capacity. In particular, the use of SQL filter rules results in lower overall message throughput at the subscription, topic, and namespace level. Whenever possible, applications should choose correlation filters over SQL-like filters, since they are much more efficient in processing and therefore have less impact on throughput.

## Actions

With SQL filter conditions, and only with those, you can define an action that can annotate the message by adding, removing, or replacing properties and their values. The action [uses a SQL-like expression](#) that loosely leans on the SQL UPDATE statement syntax. The action is performed on the message after it has been matched and before the message is selected into the topic. The changes to the message properties are private to the message copied into the subscription.

## Usage patterns

The simplest usage scenario for a topic is that every subscription gets a copy of each message sent to a topic, which enables a broadcast pattern.

Filters and actions enable two further groups of patterns: partitioning and routing.

Partitioning uses filters to distribute messages across several existing topic subscriptions in a predictable and mutually exclusive manner. The partitioning pattern is used when a system is scaled out to handle many different contexts in functionally identical compartments that each hold a subset of the overall data; for example, customer profile information. With partitioning, a publisher submits the message into a topic without requiring any knowledge of the partitioning model. The message then is moved to the correct subscription from which it can then be retrieved by the partition's message handler.

Routing uses filters to distribute messages across topic subscriptions in a predictable fashion, but not necessarily exclusive. In conjunction with the [auto forwarding](#) feature, topic filters can be used to create complex routing graphs within a Service Bus namespace for message distribution within an Azure region. With Azure Functions or Azure Logic Apps acting as a bridge between Azure Service Bus namespaces, you can create complex global topologies with direct integration into line of business applications.

## Next steps

To learn more about Service Bus messaging, see the following topics:

- [Service Bus fundamentals](#)
- [Service Bus queues, topics, and subscriptions](#)
- [SQLFilter syntax](#)
- [How to use Service Bus topics and subscriptions](#)

# Partitioned queues and topics

11/15/2017 • 11 min to read • [Edit Online](#)

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 *partitions* enable queues and topics, or *messaging entities*, to be partitioned across multiple message brokers and messaging stores. This means that the overall throughput of a partitioned entity 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 information about Service Bus internals, see the [Service Bus architecture](#) article.

Partitioning is enabled by default at entity creation on all queues and topics in both Standard and Premium messaging. You can create Standard messaging tier entities without partitioning, but queues and topics in a Premium namespace are always partitioned; this option cannot be disabled.

It is not possible to change the partitioning option on an existing queue or topic in either Standard or Premium tiers, you can only set the option when you create the entity.

## 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 to 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` or later in your HTTP requests.

### Standard

In the Standard messaging tier, 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 copies (16 partitions) of the entity 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](#), in the **Overview** blade for that entity.

### Premium

In a Premium tier namespace, you can create Service Bus queues and topics in 1, 2, 3, 4, 5, 10, 20, 40, or 80 GB

sizes (the default is 1 GB). With partitioning enabled by default, Service Bus creates two partitions per entity. You can see the maximum size of your partitioned queue or topic by looking at its entry on the [Azure portal](#), in the **Overview** blade for that entity.

For more information about partitioning in the Premium messaging tier, see [Service Bus Premium and Standard messaging tiers](#).

### Create a partitioned entity

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. As stated previously, 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 the [Azure portal](#) or in Visual Studio. When you create a queue or topic in the portal, the **Enable partitioning** option in the queue or topic **Create** dialog box is checked by default. You can only disable this option in a Standard tier entity; in the Premium tier partitioning is always enabled. 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 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 enables 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. However, you will not achieve the guaranteed ordering that a partition key provides.

For a more in-depth discussion of the tradeoff between availability (no partition key) and consistency (using a partition key), see [this article](#). This information applies equally to partitioned Service Bus entities.

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. For more information, see this [discussion of availability and consistency](#).
- **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 256 KB. 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 the [QueueClient.PeekBatch](#) or [SubscriptionClient.PeekBatch](#) API documentation.

## 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

Read about the core concepts of the AMQP 1.0 messaging specification in the [AMQP 1.0 protocol guide](#).

# Message sessions: first in, first out (FIFO)

3/22/2018 • 5 min to read • [Edit Online](#)

Microsoft Azure Service Bus sessions enable joint and ordered handling of unbounded sequences of related messages. To realize a FIFO guarantee in Service Bus, use Sessions. Service Bus is not prescriptive about the nature of the relationship between the messages, and also does not define a particular model for determining where a message sequence starts or ends.

Any sender can create a session when submitting messages into a topic or queue by setting the **SessionId** property to some application-defined identifier that is unique to the session. At the AMQP 1.0 protocol level, this value maps to the *group-id* property.

On session-aware queues or subscriptions, sessions come into existence when there is at least one message with the session's **SessionId**. Once a session exists, there is no defined time or API for when the session expires or disappears. Theoretically, a message can be received for a session today, the next message in a year's time, and if the **SessionId** matches, the session is the same from the Service Bus perspective.

Typically, however, an application has a clear notion of where a set of related messages starts and ends. Service Bus does not set any specific rules.

An example of how to delineate a sequence for transferring a file is to set the **Label** property for the first message to **start**, for intermediate messages to **content**, and for the last message to **end**. The relative position of the content messages can be computed as the current message *SequenceNumber* delta from the **start** message *SequenceNumber*.

The session feature in Service Bus enables a specific receive operation, in the form of **MessageSession** in the C# and Java APIs. You enable the feature by setting the **requiresSession** property on the queue or subscription via Azure Resource Manager, or by setting the flag in the portal. This is required before you attempt to use the related API operations.

In the portal, set the flag with the following check box:

Create queue

qtest2

\* Name  
mynewqueue ✓

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

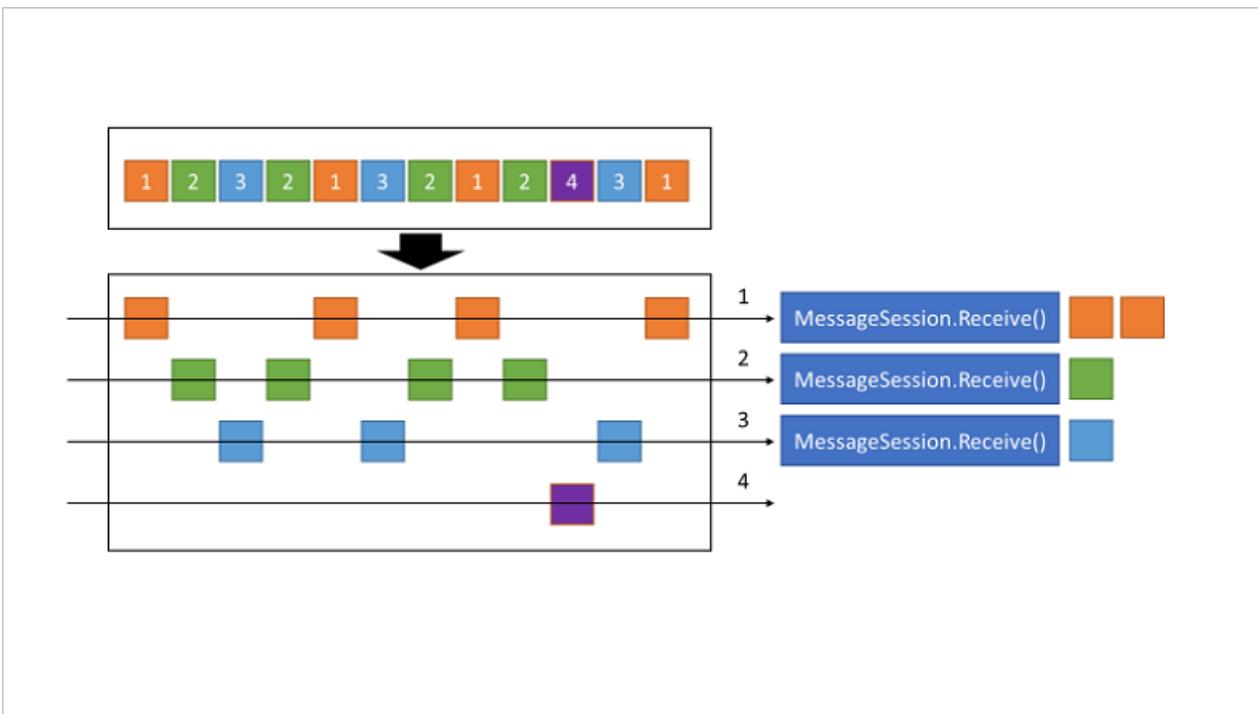
Enable partitioning

**Create**

The APIs for sessions exist on queue and subscription clients. There is an imperative model that controls when sessions and messages are received, and a handler-based model, similar to *OnMessage*, that hides the complexity of managing the receive loop.

## Session features

Sessions provide concurrent de-multiplexing of interleaved message streams while preserving and guaranteeing ordered delivery.



A [MessageSession](#) receiver is created by the client accepting a session. The client calls [QueueClient.AcceptMessageSession](#) or [QueueClient.AcceptMessageSessionAsync](#) in C#. In the reactive callback model, it registers a session handler.

When the [MessageSession](#) object is accepted and while it is held by a client, that client holds an exclusive lock on all messages with that session's [SessionId](#) that exist in the queue or subscription, and also on all messages with that **SessionId** that still arrive while the session is held.

The lock is released when [Close](#) or [CloseAsync](#) are called, or when the lock expires in cases in which the application is unable to perform the close operation. The session lock should be treated like an exclusive lock on a file, meaning that the application should close the session as soon as it no longer needs it and/or does not expect any further messages.

When multiple concurrent receivers pull from the queue, the messages belonging to a particular session are dispatched to the specific receiver that currently holds the lock for that session. With that operation, an interleaved message stream residing in one queue or subscription is cleanly de-multiplexed to different receivers and those receivers can also live on different client machines, since the lock management happens service-side, inside Service Bus.

The previous illustration shows three concurrent session receivers. One Session with `SessionId = 4` has no active, owning client, which means that no messages are delivered from this specific session. A session acts in many ways like a a sub queue.

The session lock held by the session receiver is an umbrella for the message locks used by the *peek-lock* settlement mode. A receiver cannot have two messages concurrently "in flight," but the messages must be processed in order. A new message can only be obtained when the prior message has been completed or dead-lettered. Abandoning a message causes the same message to be served again with the next receive operation.

## Message session state

When workflows are processed in high-scale, high-availability cloud systems, the workflow handler associated with a particular session must be able to recover from unexpected failures and also be capable of resuming partially completed work on a different process or machine from where the work began.

The session state facility enables an application-defined annotation of a message session inside the broker, so that the recorded processing state relative to that session becomes instantly available when the session is acquired by a

new processor.

From the Service Bus perspective, the message session state is an opaque binary object that can hold data of the size of one message, which is 256 KB for Service Bus Standard, and 1 MB for Service Bus Premium. The processing state relative to a session can be held inside the session state, or the session state can point to some storage location or database record that holds such information.

The APIs for managing session state, [SetState](#) and [GetState](#), can be found on the [MessageSession](#) object in both the C# and Java APIs. A session that had previously no session state set returns a **null** reference for [GetState](#). Clearing the previously set session state is done with [SetState\(null\)](#).

Note that session state remains as long as it is not cleared up (returning **null**), even if all messages in a session are consumed.

All existing sessions in a queue or subscription can be enumerated with the [SessionBrowser](#) method in the Java API and with [GetMessageSessions](#) on the [QueueClient](#) and [SubscriptionClient](#) in the .NET client.

The session state held in a queue or in a subscription counts towards that entity's storage quota. When the application is finished with a session, it is therefore recommended for the application to clean up its retained state to avoid external management cost.

## Next steps

- [A complete example](#) of sending and receiving session-based messages from Service Bus queues using the .NET Standard library.
- [A sample](#) that uses the .NET Framework client to handle session-aware messages.

To learn more about Service Bus messaging, see the following topics:

- [Service Bus fundamentals](#)
- [Service Bus queues, topics, and subscriptions](#)
- [Get started with Service Bus queues](#)
- [How to use Service Bus topics and subscriptions](#)

# AMQP 1.0 support in Service Bus

1/30/2018 • 5 min to read • [Edit Online](#)

Both the Azure Service Bus cloud service and on-premises [Service Bus for Windows Server \(Service Bus 1.1\)](#) support the Advanced Message Queueing Protocol (AMQP) 1.0. AMQP enables you to build cross-platform, hybrid applications using an open standard protocol. You can construct applications using components that are built using different languages and frameworks, and that run on different operating systems. All these components can connect to Service Bus and seamlessly exchange structured business messages efficiently and at full fidelity.

## Introduction: What is AMQP 1.0 and why is it important?

Traditionally, message-oriented middleware products have used proprietary protocols for communication between client applications and brokers. This means that once you've selected a particular vendor's messaging broker, you must use that vendor's libraries to connect your client applications to that broker. This results in a degree of dependence on that vendor, since porting an application to a different product requires code changes in all the connected applications.

Furthermore, connecting messaging brokers from different vendors is tricky. This typically requires application-level bridging to move messages from one system to another and to translate between their proprietary message formats. This is a common requirement; for example, when you must provide a new unified interface to older disparate systems, or integrate IT systems following a merger.

The software industry is a fast-moving business; new programming languages and application frameworks are introduced at a sometimes bewildering pace. Similarly, the requirements of IT systems evolve over time and developers want to take advantage of the latest platform features. However, sometimes the selected messaging vendor does not support these platforms. Because messaging protocols are proprietary, it's not possible for others to provide libraries for these new platforms. Therefore, you must use approaches such as building gateways or bridges that enable you to continue to use the messaging product.

The development of the Advanced Message Queueing Protocol (AMQP) 1.0 was motivated by these issues. It originated at JP Morgan Chase, who, like most financial services firms, are heavy users of message-oriented middleware. The goal was simple: to create an open-standard messaging protocol that made it possible to build message-based applications using components built using different languages, frameworks, and operating systems, all using best-of-breed components from a range of suppliers.

## AMQP 1.0 technical features

AMQP 1.0 is an efficient, reliable, wire-level messaging protocol that you can use to build robust, cross-platform, messaging applications. The protocol has a simple goal: to define the mechanics of the secure, reliable, and efficient transfer of messages between two parties. The messages themselves are encoded using a portable data representation that enables heterogeneous senders and receivers to exchange structured business messages at full fidelity. The following is a summary of the most important features:

- **Efficient:** AMQP 1.0 is a connection-oriented protocol that uses a binary encoding for the protocol instructions and the business messages transferred over it. It incorporates sophisticated flow-control schemes to maximize the utilization of the network and the connected components. That said, the protocol was designed to strike a balance between efficiency, flexibility and interoperability.
- **Reliable:** The AMQP 1.0 protocol allows messages to be exchanged with a range of reliability guarantees, from fire-and-forget to reliable, exactly-once acknowledged delivery.
- **Flexible:** AMQP 1.0 is a flexible protocol that can be used to support different topologies. The same protocol

can be used for client-to-client, client-to-broker, and broker-to-broker communications.

- **Broker-model independent:** The AMQP 1.0 specification does not make any requirements on the messaging model used by a broker. This means that it's possible to easily add AMQP 1.0 support to existing messaging brokers.

## AMQP 1.0 is a Standard (with a capital 'S')

AMQP 1.0 is an international standard, approved by ISO and IEC as ISO/IEC 19464:2014.

AMQP 1.0 has been in development since 2008 by a core group of more than 20 companies, both technology suppliers and end-user firms. During that time, user firms have contributed their real-world business requirements and the technology vendors have evolved the protocol to meet those requirements. Throughout the process, vendors have participated in workshops in which they collaborated to validate the interoperability between their implementations.

In October 2011, the development work transitioned to a technical committee within the Organization for the Advancement of Structured Information Standards (OASIS) and the OASIS AMQP 1.0 Standard was released in October 2012. The following firms participated in the technical committee during the development of the standard:

- **Technology vendors:** Axway Software, Huawei Technologies, IIT Software, INETCO Systems, Kaazing, Microsoft, Mitre Corporation, Primeton Technologies, Progress Software, Red Hat, SITA, Software AG, Solace Systems, VMware, WSO2, Zenika.
- **User firms:** Bank of America, Credit Suisse, Deutsche Boerse, Goldman Sachs, JPMorgan Chase.

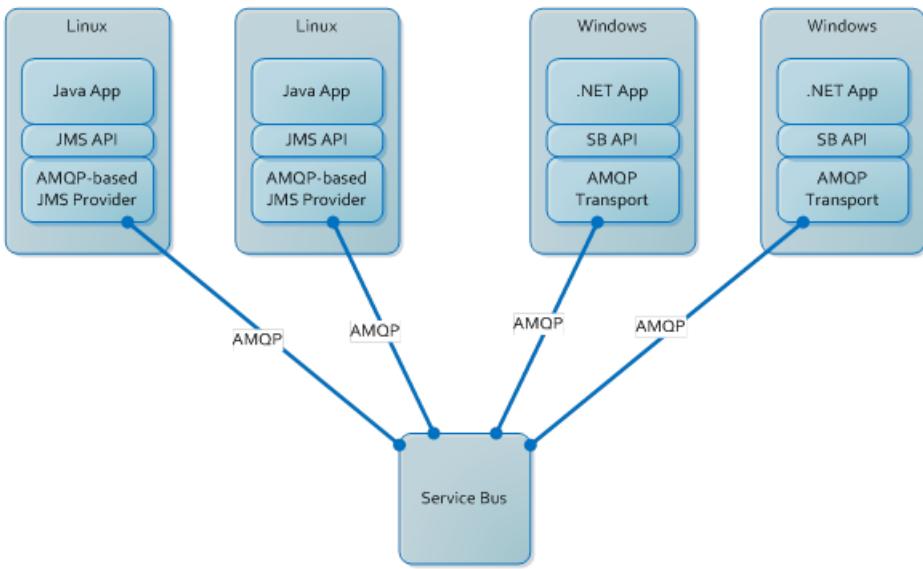
Some of the commonly cited benefits of open standards include:

- Less chance of vendor lock-in
- Interoperability
- Broad availability of libraries and tooling
- Protection against obsolescence
- Availability of knowledgeable staff
- Lower and manageable risk

## AMQP 1.0 and Service Bus

AMQP 1.0 support in Azure Service Bus means that you can now leverage the Service Bus queuing and publish/subscribe brokered messaging features from a range of platforms using an efficient binary protocol. Furthermore, you can build applications comprised of components built using a mix of languages, frameworks, and operating systems.

The following figure illustrates an example deployment in which Java clients running on Linux, written using the standard Java Message Service (JMS) API and .NET clients running on Windows, exchange messages via Service Bus using AMQP 1.0.



**Figure 1: Example deployment scenario showing cross-platform messaging using Service Bus and AMQP 1.0**

At this time the following client libraries are known to work with Service Bus:

| LANGUAGE | LIBRARY                                                                           |
|----------|-----------------------------------------------------------------------------------|
| Java     | Apache Qpid Java Message Service (JMS) client<br>IIT Software SwiftMQ Java client |
| C        | Apache Qpid Proton-C                                                              |
| PHP      | Apache Qpid Proton-PHP                                                            |
| Python   | Apache Qpid Proton-Python                                                         |
| C#       | AMQP .Net Lite                                                                    |

**Figure 2: Table of AMQP 1.0 client libraries**

## Summary

- AMQP 1.0 is an open, reliable messaging protocol that you can use to build cross-platform, hybrid applications. AMQP 1.0 is an OASIS standard.
- AMQP 1.0 support is now available in Azure Service Bus as well as Service Bus for Windows Server (Service Bus 1.1). Pricing is the same as for the existing protocols.

## Next steps

Ready to learn more? Visit the following links:

- [Using Service Bus from .NET with AMQP](#)
- [Using Service Bus from Java with AMQP](#)
- [Installing Apache Qpid Proton-C on an Azure Linux VM](#)
- [AMQP in Service Bus for Windows Server](#)

# Use Service Bus from .NET with AMQP 1.0

4/26/2018 • 4 min to read • [Edit Online](#)

AMQP 1.0 support is available in the Service Bus package version 2.1 or later. You can ensure you have the latest version by downloading the Service Bus bits from [NuGet](#).

## Configure .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 a 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 Service Bus connection string. 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;SharedAccessKey=[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 `SAS key` are obtained from the [Azure portal](#) when you create a Service Bus namespace. For more information, see [Create a Service Bus namespace using the Azure portal](#).

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.
System.Array	array	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.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	<pre>&lt;type name="uri" class=restricted source="string"&gt; &lt;descriptor name="com.microsoft:uri" /&gt; &lt;/type&gt;</pre>	Uri.AbsoluteUri
DateTimeOffset	<pre>&lt;type name="datetime-offset" class=restricted source="long"&gt; &lt;descriptor name="com.microsoft:datetime-offset" /&gt;&lt;/type&gt;</pre>	DateTimeOffset.UtcTicks
TimeSpan	<pre>&lt;type name="timespan" class=restricted source="long"&gt; &lt;descriptor name="com.microsoft:timespan" /&gt; &lt;/type&gt;</pre>	TimeSpan.Ticks

## Behavioral differences

There are 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.

## Control 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 protocol guide](#)

4 min to read •

# How to use the Java Message Service (JMS) API with Service Bus and AMQP 1.0

8/10/2017 • 8 min to read • [Edit Online](#)

The Advanced Message Queuing Protocol (AMQP) 1.0 is an efficient, reliable, wire-level messaging protocol that you can use to build robust, cross-platform messaging applications.

Support for AMQP 1.0 in Service Bus means that you can use the queuing and publish/subscribe brokered messaging features from a range of platforms using an efficient binary protocol. Furthermore, you can build applications comprised of components built using a mix of languages, frameworks, and operating systems.

This article explains how to use Service Bus messaging features (queues and publish/subscribe topics) from Java applications using the popular Java Message Service (JMS) API standard. There is a [companion article](#) that explains how to do the same using the Service Bus .NET API. You can use these two guides together to learn about cross-platform messaging using AMQP 1.0.

## Get started with Service Bus

This guide assumes that you already have a Service Bus namespace containing a queue named **queue1**. If you do not, then you can [create the namespace and queue](#) using the [Azure portal](#). For more information about how to create Service Bus namespaces and queues, see [Get started with Service Bus queues](#).

### NOTE

Partitioned queues and topics also support AMQP. For more information, see [Partitioned messaging entities](#) and [AMQP 1.0 support for Service Bus partitioned queues and topics](#).

## Downloading the AMQP 1.0 JMS client library

For information about where to download the latest version of the Apache Qpid JMS AMQP 1.0 client library, visit <https://qpid.apache.org/download.html>.

You must add the following four JAR files from the Apache Qpid JMS AMQP 1.0 distribution archive to the Java CLASSPATH when building and running JMS applications with Service Bus:

- geronimo-jms\_1.1\_spec-1.0.jar
- qpid-amqp-1-0-client-[version].jar
- qpid-amqp-1-0-client-jms-[version].jar
- qpid-amqp-1-0-common-[version].jar

## Coding Java applications

### Java Naming and Directory Interface (JNDI)

JMS uses the Java Naming and Directory Interface (JNDI) to create a separation between logical names and physical names. Two types of JMS objects are resolved using JNDI: ConnectionFactory and Destination. JNDI uses a provider model into which you can plug different directory services to handle name resolution duties. The Apache Qpid JMS AMQP 1.0 library comes with a simple properties file-based JNDI Provider that is configured using a properties file of the following format:

```

servicebus.properties - sample JNDI configuration

Register a ConnectionFactory in JNDI using the form:
connectionfactory.[jndi_name] = [ConnectionURL]
connectionfactory.SBCF = amqps://[SASPolicyName]:[SASPolicyKey]@[namespace].servicebus.windows.net

Register some queues in JNDI using the form
queue.[jndi_name] = [physical_name]
topic.[jndi_name] = [physical_name]
queue.QUEUE = queue1

```

#### Configure the ConnectionFactory

The entry used to define a **ConnectionFactory** in the Qpid properties file JNDI provider is of the following format:

```
connectionfactory.[jndi_name] = [ConnectionURL]
```

Where **[jndi\_name]** and **[ConnectionURL]** have the following meanings:

- **[jndi\_name]**: The logical name of the ConnectionFactory. This is the name that will be resolved in the Java application using the JNDI InitialContext.lookup() method.
- **[ConnectionURL]**: A URL that provides the JMS library with the information required to the AMQP broker.

The format of the **ConnectionURL** is as follows:

```
amqps://[SASPolicyName]:[SASPolicyKey]@[namespace].servicebus.windows.net
```

Where **[namespace]**, **[SASPolicyName]** and **[SASPolicyKey]** have the following meanings:

- **[namespace]**: The Service Bus namespace.
- **[SASPolicyName]**: The Queue Shared Access Signature policy name.
- **[SASPolicyKey]**: The Queue Shared Access Signature policy key.

#### NOTE

You must URL-encode the password manually. A useful URL-encoding utility is available at [http://www.w3schools.com/tags/ref\\_urlencode.asp](http://www.w3schools.com/tags/ref_urlencode.asp).

#### Configure destinations

The entry used to define a destination in the Qpid properties file JNDI provider is of the following format:

```
queue.[jndi_name] = [physical_name]
```

or

```
topic.[jndi_name] = [physical_name]
```

Where **[jndi\_name]** and **[physical\_name]** have the following meanings:

- **[jndi\_name]**: The logical name of the destination. This is the name that will be resolved in the Java application using the JNDI InitialContext.lookup() method.
- **[physical\_name]**: The name of the Service Bus entity to which the application sends or receives messages.

## NOTE

When receiving from a Service Bus topic subscription, the physical name specified in JNDI should be the name of the topic. The subscription name is provided when the durable subscription is created in the JMS application code. The [Service Bus AMQP 1.0 Developer's Guide](#) provides more details on working with Service Bus topics from JMS.

## Write the JMS application

There are no special APIs or options required when using JMS with Service Bus. However, there are a few restrictions that will be covered later. As with any JMS application, the first thing required is configuration of the JNDI environment, to be able to resolve a **ConnectionFactory** and destinations.

### Configure the JNDI InitialContext

The JNDI environment is configured by passing a hashtable of configuration information into the constructor of the javax.naming.InitialContext class. The two required elements in the hashtable are the class name of the Initial Context Factory and the Provider URL. The following code shows how to configure the JNDI environment to use the Qpid properties file based JNDI Provider with a properties file named **servicebus.properties**.

```
Hashtable<String, String> env = new Hashtable<String, String>();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"org.apache.qpid.amqp_1_0.jms.jndi.PropertiesFileInitialContextFactory");
env.put(Context.PROVIDER_URL, "servicebus.properties");
InitialContext context = new InitialContext(env);
```

## A simple JMS application using a Service Bus queue

The following example program sends JMS TextMessages to a Service Bus queue with the JNDI logical name of QUEUE, and receives the messages back.

```
// SimpleSenderReceiver.java

import javax.jms.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Hashtable;
import java.util.Random;

public class SimpleSenderReceiver implements MessageListener {
 private static boolean runReceiver = true;
 private Connection connection;
 private Session sendSession;
 private Session receiveSession;
 private MessageProducer sender;
 private MessageConsumer receiver;
 private static Random randomGenerator = new Random();

 public SimpleSenderReceiver() throws Exception {
 // Configure JNDI environment
 Hashtable<String, String> env = new Hashtable<String, String>();
 env.put(Context.INITIAL_CONTEXT_FACTORY,
"org.apache.qpid.amqp_1_0.jms.jndi.PropertiesFileInitialContextFactory");
 env.put(Context.PROVIDER_URL, "servicebus.properties");
 Context context = new InitialContext(env);

 // Look up ConnectionFactory and Queue
 ConnectionFactory cf = (ConnectionFactory) context.lookup("SBCF");
 Destination queue = (Destination) context.lookup("QUEUE");

 // Create Connection
 connection = cf.createConnection();
```

```

// Create sender-side Session and MessageProducer
sendSession = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
sender = sendSession.createProducer(queue);

if (runReceiver) {
 // Create receiver-side Session, MessageConsumer, and MessageListener
 receiveSession = connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);
 receiver = receiveSession.createConsumer(queue);
 receiver.setMessageListener(this);
 connection.start();
}
}

public static void main(String[] args) {
 try {

 if ((args.length > 0) && args[0].equalsIgnoreCase("sendonly")) {
 runReceiver = false;
 }

 SimpleSenderReceiver simpleSenderReceiver = new SimpleSenderReceiver();
 System.out.println("Press [enter] to send a message. Type 'exit' + [enter] to quit.");
 BufferedReader commandLine = new java.io.BufferedReader(new InputStreamReader(System.in));

 while (true) {
 String s = commandLine.readLine();
 if (s.equalsIgnoreCase("exit")) {
 simpleSenderReceiver.close();
 System.exit(0);
 } else {
 simpleSenderReceiver.sendMessage();
 }
 }
 } catch (Exception e) {
 e.printStackTrace();
 }
}

private void sendMessage() throws JMSException {
 TextMessage message = sendSession.createTextMessage();
 message.setText("Test AMQP message from JMS");
 long randomMessageID = randomGenerator.nextLong() >>>1;
 message.setJMSMessageID("ID:" + randomMessageID);
 sender.send(message);
 System.out.println("Sent message with JMSMessageID = " + message.getJMSMessageID());
}

public void close() throws JMSException {
 connection.close();
}

public void onMessage(Message message) {
 try {
 System.out.println("Received message with JMSMessageID = " + message.getJMSMessageID());
 message.acknowledge();
 } catch (Exception e) {
 e.printStackTrace();
 }
}
}

```

## Run the application

Running the application produces output of the form:

```
> java SimpleSenderReceiver
Press [enter] to send a message. Type 'exit' + [enter] to quit.

Sent message with JMSMessageID = ID:2867600614942270318
Received message with JMSMessageID = ID:2867600614942270318

Sent message with JMSMessageID = ID:7578408152750301483
Received message with JMSMessageID = ID:7578408152750301483

Sent message with JMSMessageID = ID:956102171969368961
Received message with JMSMessageID = ID:956102171969368961
exit
```

## Cross-platform messaging between JMS and .NET

This guide showed how to send and receive messages to and from Service Bus using JMS. However, one of the key benefits of AMQP 1.0 is that it enables applications to be built from components written in different languages, with messages exchanged reliably and at full fidelity.

Using the sample JMS application described above and a similar .NET application taken from a companion article, [Using Service Bus from .NET with AMQP 1.0](#), you can exchange messages between .NET and Java. Read this article for more information about the details of cross-platform messaging using Service Bus and AMQP 1.0.

### JMS to .NET

To demonstrate JMS to .NET messaging:

- Start the .NET sample application without any command-line arguments.
- Start the Java sample application with the "sendonly" command-line argument. In this mode, the application will not receive messages from the queue, it will only send.
- Press **Enter** a few times in the Java application console, which will cause messages to be sent.
- These messages are received by the .NET application.

### Output from JMS application

```
> java SimpleSenderReceiver sendonly
Press [enter] to send a message. Type 'exit' + [enter] to quit.
Sent message with JMSMessageID = ID:4364096528752411591
Sent message with JMSMessageID = ID:459252991689389983
Sent message with JMSMessageID = ID:1565011046230456854
exit
```

### Output from .NET application

```
> SimpleSenderReceiver.exe
Press [enter] to send a message. Type 'exit' + [enter] to quit.
Received message with MessageID = 4364096528752411591
Received message with MessageID = 459252991689389983
Received message with MessageID = 1565011046230456854
exit
```

### .NET to JMS

To demonstrate .NET to JMS messaging:

- Start the .NET sample application with the "sendonly" command-line argument. In this mode, the application will not receive messages from the queue, it will only send.
- Start the Java sample application without any command-line arguments.
- Press **Enter** a few times in the .NET application console, which will cause messages to be sent.

- These messages are received by the Java application.

#### Output from .NET application

```
> SimpleSenderReceiver.exe sendonly
Press [enter] to send a message. Type 'exit' + [enter] to quit.
Sent message with MessageID = d64e681a310a48a1ae0ce7b017bf1cf3
Sent message with MessageID = 98a39664995b4f74b32e2a0ecccc46bb
Sent message with MessageID = acbca67f03c346de9b7893026f97ddeb
exit
```

#### Output from JMS application

```
> java SimpleSenderReceiver
Press [enter] to send a message. Type 'exit' + [enter] to quit.
Received message with JMSMessageID = ID:d64e681a310a48a1ae0ce7b017bf1cf3
Received message with JMSMessageID = ID:98a39664995b4f74b32e2a0ecccc46bb
Received message with JMSMessageID = ID:acbca67f03c346de9b7893026f97ddeb
exit
```

## Unsupported features and restrictions

The following restrictions exist when using JMS over AMQP 1.0 with Service Bus, namely:

- Only one **MessageProducer** or **MessageConsumer** is allowed per **Session**. If you need to create multiple **MessageProducers** or **MessageConsumers** in an application, create a dedicated **Session** for each of them.
- Volatile topic subscriptions are not currently supported.
- **MessageSelectors** are not currently supported.
- Temporary destinations; for example, **TemporaryQueue**, **TemporaryTopic** are not currently supported, along with the **QueueRequestor** and **TopicRequestor** APIs that use them.
- Transacted sessions and distributed transactions are not supported.

## Summary

This how-to guide showed how to use Service Bus brokered messaging features (queues and publish/subscribe topics) from Java using the popular JMS API and AMQP 1.0.

You can also use Service Bus AMQP 1.0 from other languages, including .NET, C, Python, and PHP. Components built using these different languages can exchange messages reliably and at full fidelity using the AMQP 1.0 support in Service Bus.

## Next steps

- [AMQP 1.0 support in Azure Service Bus](#)
- [How to use AMQP 1.0 with the Service Bus .NET API](#)
- [Service Bus AMQP 1.0 Developer's Guide](#)
- [Get started with Service Bus queues](#)
- [Java Developer Center](#)

# AMQP 1.0 in Azure Service Bus and Event Hubs protocol guide

4/30/2018 • 27 min to read • [Edit Online](#)

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 protocols. 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 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 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 features are 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, called *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, enabling further specifications to enhance its capabilities. The three extension specifications discussed 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 the 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 is 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 an 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 (256 K 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.

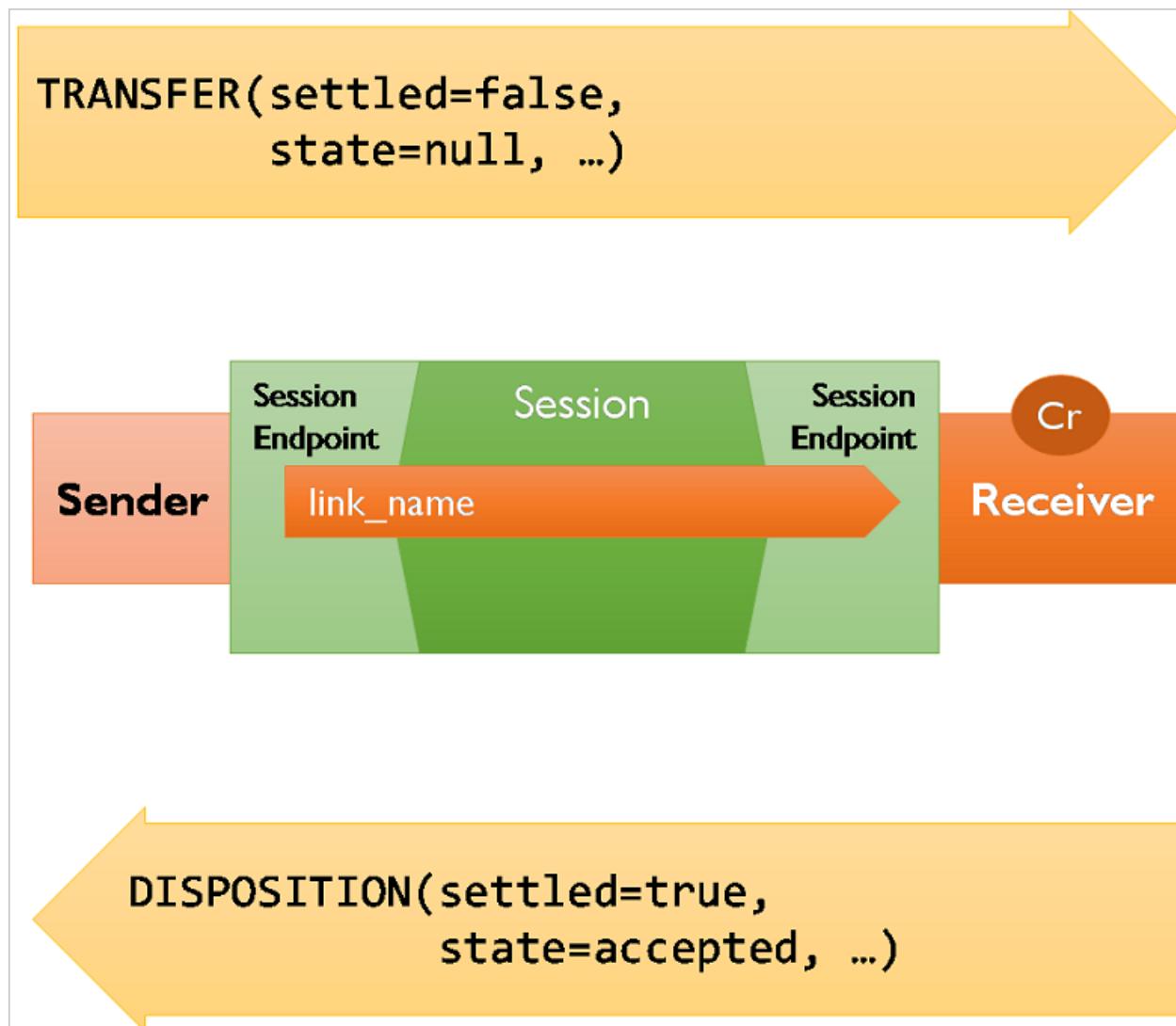
In Service Bus, a node is directly equivalent to a queue, a topic, a subscription, or a deadletter subqueue of a queue

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 does 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 does not provide any feedback about the outcome of the operation. This mode is supported by 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 then indicates 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 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 example, 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; 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 called *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.

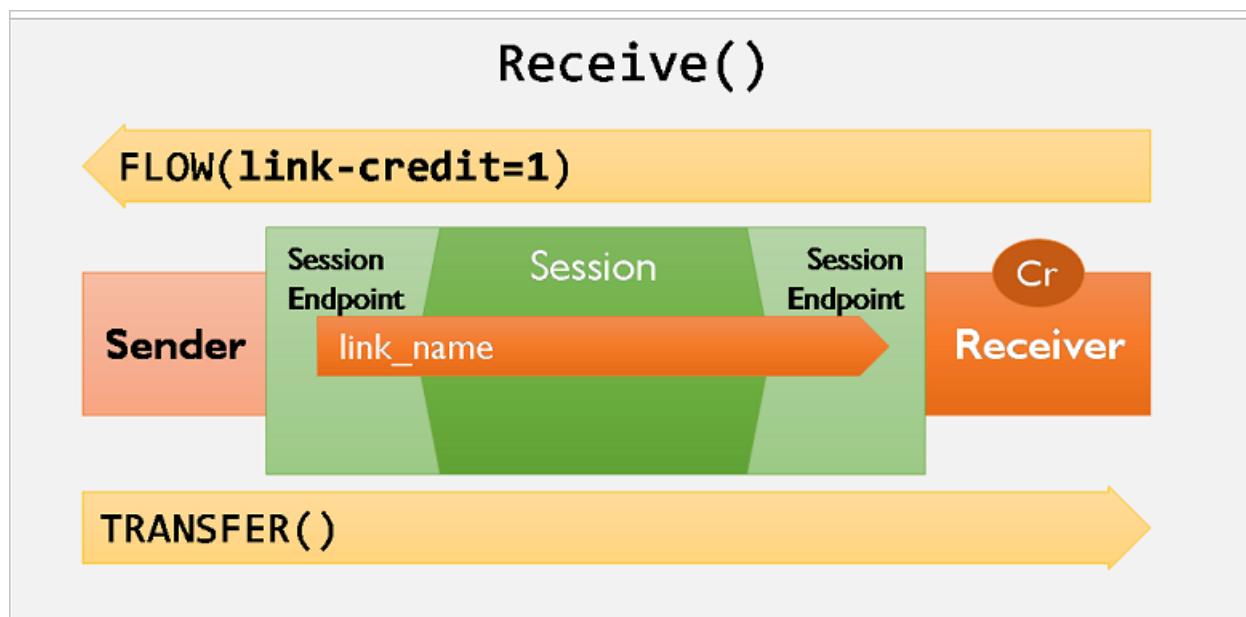
Service Bus does not support link recovery; if the client loses the connection to Service Bus with an unsettled message transfer pending, that message transfer is lost, and the client must reconnect, reestablish the link, and retry the transfer.

As such, Service Bus and Event Hubs support "at least once" transfers where the sender can be assured for the message having been stored and accepted, but do 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, 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, then silently drops 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 the sender is assigned link credit, it attempts to use up that credit by delivering messages. Each message delivery decrements the remaining link credit by 1. When the link credit is used up, deliveries stop.

When Service Bus is in the receiver role, it instantly provides the sender with ample link credit, so that messages can be sent immediately. As link credit is used, Service Bus occasionally sends a *flow* performative to the sender to update the link credit balance.

In the sender role, Service Bus sends 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 consumes 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 remains recorded in order of arrival, and messages are 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 is delivered to the next receiver when the transfer reaches any of the other states. Service Bus automatically moves 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 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 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, not when the message is 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 when required. Creating a message sender may not cause a network interaction until the first message is sent or requested.

The arrows in the following table show the performative flow direction.

#### Create message receiver

CLIENT	SERVICE BUS
--> attach( name={link name}, handle={numeric handle}, role= <b>receiver</b> , 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= <b>sender</b> , source={entity name}, target={client link id} )

#### Create message sender

CLIENT	SERVICE BUS
--> attach( name={link name}, handle={numeric handle}, <b>role=sender</b> , source={client link id}, target={entity name} )	No action
No action	<-- attach( name={link name}, handle={numeric handle}, <b>role=receiver</b> , source={client link id}, target={entity name} )

#### Create message sender (error)

CLIENT	SERVICE BUS
--> attach( name={link name}, handle={numeric handle}, <b>role=sender</b> , source={client link id}, target={entity name} )	No action
No action	<-- attach( name={link name}, handle={numeric handle}, <b>role=receiver</b> , source=null, target=null )  <-- detach( handle={numeric handle}, <b>closed=true</b> , error={error info} )

#### Close message receiver/sender

CLIENT	SERVICE BUS
--> detach( handle={numeric handle}, <b>closed=true</b> )	No action
No action	<-- detach( handle={numeric handle}, <b>closed=true</b> )

#### 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= <b>receiver</b> , first={delivery id}, last={delivery id}, settled= <b>true</b> , state= <b>accepted</b> )	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= <b>false</b> , more= <b>false</b> , state= <b>null</b> , resume= <b>false</b> )
No action	< transfer( delivery-id={numeric handle+1}, delivery-tag={binary handle}, settled= <b>false</b> , more= <b>false</b> , state= <b>null</b> , resume= <b>false</b> )
No action	< transfer( delivery-id={numeric handle+2}, delivery-tag={binary handle}, settled= <b>false</b> , more= <b>false</b> , state= <b>null</b> , resume= <b>false</b> )
--> disposition( role=receiver, first={delivery id}, last={delivery id+2}, settled= <b>true</b> , state= <b>accepted</b> )	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 Service Bus API set.

Any property that application needs to defines should be mapped to AMQP's `application-properties` map.

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 expires. 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.	<a href="#">SessionId</a>
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	-	<a href="#">ReplyToSessionId</a>

#### Message annotations

There are few other service bus message properties which are not part of AMQP message properties, and are passed along as `MessageAnnotations` on the message.

ANNOTATION MAP KEY	USAGE	API NAME
x-opt-scheduled-enqueue-time	Declares at which time the message should appear on the entity	<a href="#">ScheduledEnqueueTime</a>
x-opt-partition-key	Application-defined key that dictates which partition the message should land in.	<a href="#">PartitionKey</a>
x-opt-via-partition-key	Application-defined partition-key value when a transaction is to be used to send messages via a transfer queue.	<a href="#">ViaPartitionKey</a>
x-opt-enqueued-time	Service-defined UTC time representing the actual time of enqueueing the message. Ignored on input.	<a href="#">EnqueuedTimeUtc</a>
x-opt-sequence-number	Service-defined unique number assigned to a message.	<a href="#">SequenceNumber</a>
x-opt-offset	Service-defined enqueued sequence number of the message.	<a href="#">EnqueuedSequenceNumber</a>
x-opt-locked-until	Service-defined. The date and time until which the message will be locked in the queue/subscription.	<a href="#">LockedUntilUtc</a>
x-opt-deadletter-source	Service-Defined. If the message is received from dead letter queue, the source of the original message.	<a href="#">DeadLetterSource</a>

#### Transaction capability

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. The operations are grouped by an identifier `txnid`.

For transactional interaction, the client acts a `transaction controller` which controls the operations that should be grouped together. Service Bus Service acts as a `transactional resource` and performs work as requested by the `transaction controller`.

The client and service communicate over a `control link` which is established by the client. The `declare` and `discharge` messages are sent by the controller over the control link to allocate and complete transactions respectively (they do not represent the demarcation of transactional work). The actual send/receive is not performed on this link. Each transactional operation requested is explicitly identified with the desired `txnid` and therefore may occur on any link on the Connection. If the control link is closed while there exist non-discharged transactions it created, then all such transactions are immediately rolled back, and attempts to perform further transactional work on them will lead to failure. Messages on control link must not be pre settled.

Every connection has to initiate its own control link to be able to start and end transactions. The service defines a special target that functions as a `coordinator`. The client/controller establishes a control link to this target. Control link is outside the boundary of an entity, i.e., same control link can be used to initiate and discharge transactions for multiple entities.

#### Starting a transaction

To begin transactional work, the controller must obtain a `txnid` from the coordinator. It does this by sending a `declare` type message. If the declaration is successful, the coordinator responds with a disposition outcome of `declared` which carries the assigned `txnid`.

CLIENT (CONTROLLER)		SERVICE BUS (COORDINATOR)
<pre>attach(   name={link name},   ...   role=sender,   target=Coordinator )</pre>	----->	
	<-----	<pre>attach(   name={link name},   ...   target=Coordinator() )</pre>
<pre>transfer(   delivery-id=0, ...   { AmqpValue (Declare0) }</pre>	----->	
	<-----	<pre>disposition(   first=0, last=0,   state=Declared(     txnid={transaction id}   )) )</pre>

#### Discharging a transaction

The controller will conclude the transactional work by sending a `discharge` message to the coordinator. The controller indicates that it wishes to commit or rollback the transactional work by setting the `fail` flag on the discharge body. If the coordinator is unable to complete the discharge, the message is rejected with this outcome carrying the `transaction-error`.

Note: fail=true refers to Rollback of a transaction, and fail=false refers to Commit.

CLIENT (CONTROLLER)		SERVICE BUS (COORDINATOR)
<pre>transfer(   delivery-id=0, ...   { AmqpValue (Declare0) }</pre>	----->	

CLIENT (CONTROLLER)		SERVICE BUS (COORDINATOR)
	<-----	disposition( first=0, last=0, state=Declared( txnid={transaction id} ))
	... Transactional work on other links ...	
transfer( delivery-id=57, ...) { AmqpValue ( <b>Discharge(txn-id=0,</b> <b>fail=false))</b> }	----->	
	<-----	disposition( first=57, last=57, state= <b>Accepted()</b> )

#### Sending a message in a transaction

All transactional work is done with the transactional delivery state `transactional-state` that carries the txn-id. In the case of sending messages, the transactional-state is carried by the message's transfer frame.

CLIENT (CONTROLLER)		SERVICE BUS (COORDINATOR)
transfer( delivery-id=0, ...) { AmqpValue (Declare())}	----->	
	<-----	disposition( first=0, last=0, state=Declared( txnid={transaction id} ))
transfer( handle=1, delivery-id=1, <b>state=</b> <b>TransactionalState(</b> <b>txnid=0)</b> { payload }	----->	
	<-----	disposition( first=1, last=1, state= <b>TransactionalState(</b> <b>txnid=0,</b> <b>outcome=Accepted()</b> )

#### Disposing a message in a transaction

Message disposition includes operations like `complete` / `Abandon` / `DeadLetter` / `Defer`. To perform these operations within a transaction, pass the `transactional-state` with the disposition.

CLIENT (CONTROLLER)		SERVICE BUS (COORDINATOR)
transfer( delivery-id=0, ...) { AmqpValue (Declare())}	----->	
	<-----	disposition( first=0, last=0, state=Declared( txn-id={transaction id} ))
	<-----	transfer( handle=2, delivery-id=11, state=null) { payload }
disposition( first=11, last=11, state= <b>TransactionalState</b> ( <b>txnid=0</b> , <b>outcome=Accepted()</b> ))	----->	

## 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. Service Bus implements the latest versions of these drafts and adopts 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 article [AMQP 1.0 in Service Bus: request-response-based operations](#).

### AMQP management

The AMQP management specification is the first of the draft extensions discussed in this article. This specification defines a set of protocols 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= <b>sender</b> , source= <b>null</b> , target="myentity/\$management" )	No action

LOGICAL OPERATION	CLIENT	SERVICE BUS
Create Request Response Path	No action	<-- attach( name={link name}, handle={numeric handle}, role=receiver, source=null, target="myentity" )
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 processes the request, and then delivers 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.

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 discussed in the following sections.

### Claims-based authorization

The AMQP Claims-Based-Authorization (CBS) specification draft builds on the management specification 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. Among 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 validity requires 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 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 Service Bus, enables 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	<b>put-token</b>
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
amqpjwt	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 about three fundamental rights: "Send" enables sending, "Listen" enables receiving, and "Manage" enables 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` permits 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 [ <a href="#">RFC2616</a> ].
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 drops 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 promptly drops 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.

### Send-via functionality

[Send-via / Transfer sender](#) is a functionality that lets service bus forward a given message to a destination entity through another entity. This is mainly used to perform operations across entities in a single transaction.

With this functionality, you create a sender and establish the link to the `via-entity`. While establishing the link, additional information is passed to establish the true destination of the messages/transfers on this link. Once the attach has been successful, all the messages sent on this link will be automatically forwarded to the *destination-entity* through *via-entity*.

Note: Authentication has to be performed for both *via-entity* and *destination-entity* before establishing this link.

CLIENT	SERVICE BUS
<pre>attach(     name={link name},     role=sender,     source={client link id},     target=<b>{via-entity}</b>,     properties=map [(         com.microsoft:transfer-destination-         address=         {destination-entity} )] )</pre>	<pre>-----&gt;</pre>

CLIENT		SERVICE BUS
	<-----	attach( name={link name}, role=receiver, source={client link id}, target={via-entity}, properties=map {[ com.microsoft:transfer-destination- address= {destination-entity} ]} )

## Next steps

To learn more about AMQP, 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](#)

# AMQP 1.0 in Microsoft Azure Service Bus: request-response-based operations

4/26/2018 • 14 min to read • [Edit Online](#)

This article 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](#).

## 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.

A transaction-state can be added optionally for operations which supports transaction.

```

requestLink.sendTransfer(
 Message(
 properties: {
 message-id: <request id>,
 reply-to: "<my response link unique address>"
 },
 application-properties: {
 "operation" -> "<operation>",
 }
),
 [Optional] State = transactional-state: {
 txn-id: <txn-id>
 }
)

```

## Receive a response message

Receives the response message from the response link.

```
responseMessage = responseLink.receiveTransfer()
```

The response message is in 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/<subscription_id>	“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
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.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
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. This operation supports transaction.

#### 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	No	amqpMessage.Properties.GroupId as string
partition-key	string	No	amqpMessage.MessageAnnotations."x-opt-partition-key"
via-partition-key	string	No	amqpMessage.MessageAnnotations."x-opt-via-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

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
statusCode	int	Yes	<p>HTTP response code [RFC2616]</p> <p>200: OK – has more messages</p> <p>0xcc: No content – no more messages</p>
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	<p>HTTP response code [RFC2616]</p> <p>200: OK – has more messages</p> <p>0xcc: No content – no more messages</p>

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
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.

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
skip	int	Yes	Skip a number of sessions.
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	<code>sql-filter</code> , as specified in the next section.
correlation-filter	map	Yes	<code>correlation-filter</code> , as specified in the next section.
sql-rule-action	map	Yes	<code>sql-rule-action</code> , 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 <a href="#">BrokeredMessage.Properties</a> .

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.

## Get Rules

### Request

The request message must include the following application properties:

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
operation	string	Yes	com.microsoft:enumerate-rules
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
top	int	Yes	The number of rules to fetch in the page.
skip	int	Yes	The number of rules to skip. Defines the starting index (+1) on the list of rules.

### Response

The response message includes the following properties:

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
statusCode	int	Yes	HTTP response code [RFC2616]  200: OK – success, otherwise failed
rules	array of map	Yes	Array of rules. Each rule is represented by a map.

Each map entry in the array includes the following properties:

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
rule-description	array of described objects	Yes	<code>com.microsoft:rule-description:list</code> with AMQP described code 0x0000013700000004

`com.microsoft.rule-description:list` is an array of described objects. The array includes the following:

INDEX	VALUE TYPE	REQUIRED	VALUE CONTENTS
0	array of described objects	Yes	<code>filter</code> as specified below.
1	array of described object	Yes	<code>ruleAction</code> as specified below.
2	string	Yes	name of the rule.

`filter` can be of either of the following types:

DESCRIPTOR NAME	DESCRIPTOR CODE	VALUE
<code>com.microsoft:sql-filter:list</code>	0x0000013700000006	SQL filter
<code>com.microsoft:correlation-filter:list</code>	0x0000013700000009	Correlation filter
<code>com.microsoft:true-filter:list</code>	0x0000013700000007	True filter representing 1=1
<code>com.microsoft:false-filter:list</code>	0x0000013700000008	False filter representing 1=0

`com.microsoft:sql-filter:list` is a described array which includes:

INDEX	VALUE TYPE	REQUIRED	VALUE CONTENTS
0	string	Yes	Sql Filter expression

`com.microsoft:correlation-filter:list` is a described array which includes:

INDEX (IF EXISTS)	VALUE TYPE	VALUE CONTENTS
0	string	Correlation ID
1	string	Message ID
2	string	To
3	string	Reply To
4	string	Label
5	string	Session ID
6	string	Reply To Session ID
7	string	Content Type
8	Map	Map of application defined properties

`ruleAction` can be either of the following types:

DESCRIPTOR NAME	DESCRIPTOR CODE	VALUE
<code>com.microsoft:empty-rule-action:list</code>	0x0000013700000005	Empty Rule Action - No rule action present
<code>com.microsoft:sql-rule-action:list</code>	0x0000013700000006	SQL Rule Action

`com.microsoft:sql-rule-action:list` is an array of described objects whose first entry is a string which contains the SQL rule action's expression.

## 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	<code>com.microsoft:receive-by-sequence-number</code>
<code>com.microsoft:server-timeout</code>	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	<b>Receiver settle</b> 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 <b>receiver-settle-mode</b> is 1.
message	array of byte	Yes	AMQP 1.0 wire-encoded message.

## Update disposition status

Updates the disposition status of deferred messages. This operation supports transactions.

### Request

The request message must include the following application properties:

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
operation	string	Yes	com.microsoft:update-disposition
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

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 <b>suspended</b> .
deadletter-description	string	No	May be set if disposition status is set to <b>suspended</b> .
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 protocol guide](#)
- [AMQP in Service Bus for Windows Server](#)

# Overview of Service Bus dead-letter queues

1/31/2018 • 3 min to read • [Edit Online](#)

Azure 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.

This article describes dead-letter queues in Service Bus. Much of the discussion is illustrated by the [Dead-Letter queues sample](#) on GitHub.

## The dead-letter queue

The purpose of the dead-letter queue is to hold messages that cannot be delivered to any receiver, or 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 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 operation 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 move 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.

Condition	DeadLetterReason	DeadLetterErrorDescription
!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 [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 moved to the DLQ when there is at least one active receiver pulling from 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 can 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.
- The destination queue or topic exceeds the maximum entity size.

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](#)

# Prefetch Azure Service Bus messages

1/30/2018 • 3 min to read • [Edit Online](#)

When *Prefetch* is enabled in any of the official Service Bus clients, the receiver quietly acquires more messages, up to the [PrefetchCount](#) limit, beyond what the application initially asked for.

A single initial [Receive](#) or [ReceiveAsync](#) call therefore acquires a message for immediate consumption that is returned as soon as available. The client then acquires further messages in the background, to fill the prefetch buffer.

## Enable prefetch

With .NET, you enable the Prefetch feature by setting the [PrefetchCount](#) property of a **MessageReceiver**, **QueueClient**, or **SubscriptionClient** to a number greater than zero. Setting the value to zero turns off prefetch.

You can easily add this setting to the receive-side of the [QueuesGettingStarted](#) or [ReceiveLoop](#) samples' settings to see the effect in those contexts.

While messages are available in the prefetch buffer, any subsequent **Receive/ReceiveAsync** calls are immediately fulfilled from the buffer, and the buffer is replenished in the background as space becomes available. If there are no messages available for delivery, the receive operation empties the buffer and then waits or blocks, as expected.

Prefetch also works in the same way with the [OnMessage](#) and [OnMessageAsync](#) APIs.

## If it is faster, why is Prefetch not the default option?

Prefetch speeds up the message flow by having a message readily available for local retrieval when and before the application asks for one. This throughput gain is the result of a trade-off that the application author must make explicitly:

With the [ReceiveAndDelete](#) receive mode, all messages that are acquired into the prefetch buffer are no longer available in the queue, and only reside in the in-memory prefetch buffer until they are received into the application through the **Receive/ReceiveAsync** or **OnMessage/OnMessageAsync** APIs. If the application terminates before the messages are received into the application, those messages are irrecoverably lost.

In the [PeekLock](#) receive mode, messages fetched into the Prefetch buffer are acquired into the buffer in a locked state, and have the timeout clock for the lock ticking. If the prefetch buffer is large, and processing takes so long that message locks expire while residing in the prefetch buffer or even while the application is processing the message, there might be some confusing events for the application to handle.

The application might acquire a message with an expired or imminently expiring lock. If so, the application might process the message, but then find that it cannot complete it due to a lock expiration. The application can check the [LockedUntilUtc](#) property (which is subject to clock skew between the broker and local machine clock). If the message lock has expired, the application must ignore the message; no API call on or with the message should be made. If the message is not expired but expiration is imminent, the lock can be renewed and extended by another default lock period by calling [message.RenewLock\(\)](#)

If the lock silently expires in the prefetch buffer, the message is treated as abandoned and is again made available for retrieval from the queue. That might cause it to be fetched into the prefetch buffer; placed at the end. If the prefetch buffer cannot usually be worked through during the message expiration, this causes messages to be repeatedly prefetched but never effectively delivered in a usable (validly locked) state, and are eventually moved to the dead-letter queue once the maximum delivery count is exceeded.

If you need a high degree of reliability for message processing, and processing takes significant work and time, it is recommended that you use the prefetch feature conservatively, or not at all.

If you need high throughout and message processing is commonly cheap, prefetch yields significant throughput benefits.

The maximum prefetch count and the lock duration configured on the queue or subscription need to be balanced such that the lock timeout at least exceeds the cumulative expected message processing time for the maximum size of the prefetch buffer, plus one message. At the same time, the lock timeout ought not to be so long that messages can exceed their maximum [TimeToLive](#) when they are accidentally dropped, thus requiring their lock to expire before being redelivered.

## Next steps

To learn more about Service Bus messaging, see the following topics:

- [Service Bus fundamentals](#)
- [Service Bus queues, topics, and subscriptions](#)
- [Get started with Service Bus queues](#)
- [How to use Service Bus topics and subscriptions](#)

# Duplicate detection

1/26/2018 • 2 min to read • [Edit Online](#)

If an application encounters a fatal error immediately after it sends a message, and the restarted application instance erroneously believes that the prior message delivery did not occur, a subsequent send causes the same message to appear in the system twice.

It is also possible for an error at the client or network level to occur a moment earlier, and for a sent message to be committed into the queue, with the acknowledgment not successfully returned to the client. This scenario leaves the client in doubt about the outcome of the send operation.

Duplicate detection takes the doubt out of these situations by enabling the sender re-send the same message, and the queue or topic discards any duplicate copies.

Enabling duplicate detection helps keep track of the application-controlled *MessageId* of all messages sent into a queue or topic during a specified time window. If any new message is sent carrying a *MessageId* that has already been logged during the time window, the message is reported as accepted (the send operation succeeds), but the newly sent message is instantly ignored and dropped. No other parts of the message other than the *MessageId* are considered.

Application control of the identifier is essential, because only that allows the application to tie the *MessageId* to a business process context from which it can be predictably reconstructed in case of a failure.

For a business process in which multiple messages are sent in the course of handling some application context, the *MessageId* may be a composite of the application-level context identifier, such as a purchase order number, and the subject of the message; for example, **12345.2017/payment**.

The *MessageId* can always be some GUID, but anchoring the identifier to the business process yields predictable repeatability, which is desired for leveraging the duplicate detection feature effectively.

## Enable duplicate detection

In the portal, the feature is turned on during entity creation with the **Enable duplicate detection** check box, which is off by default. The setting for creating new topics is equivalent.

Create queue

qtest2

\* Name  
mynewqueue

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

**Create**

The screenshot shows the 'Create queue' dialog box in the Azure portal. The 'Name' field is filled with 'mynewqueue'. The 'Message time to live (default)' is set to '14 days'. The 'Lock duration' is set to '30 seconds'. Under 'Advanced options', the 'Enable duplicate detection' checkbox is checked and highlighted with a red box. Other options like 'Enable sessions' and 'Enable partitioning' are also present but not checked. At the bottom is a large blue 'Create' button.

Programmatically, you set the flag with the [QueueDescription.requiresDuplicateDetection](#) property on the full framework .NET API. With the Azure Resource Manager API, the value is set with the [queueProperties.requiresDuplicateDetection](#) property.

The duplicate detection time history defaults to 30 seconds for queues and topics, with a maximum value of 7 days. You can change this setting in the queue and topic properties window in the Azure portal.

The screenshot shows the Azure portal interface for a Service Bus queue named 'mynewqueue'. The left sidebar has a navigation menu with options like Overview, Diagnose and solve problems, Properties (which is selected and highlighted in blue), Shared access policies, Locks, and Automation script. Under SUPPORT + TROUBLESHOOTING, there's a New support request option. The main content area shows settings for the queue. A red box highlights the 'Duplicate detection history' section, which includes a dropdown for 'Time' set to 'minutes' and a value of '10'. Other settings shown include Message time to live (default: 14 days), Lock duration (1 minute), Maximum Delivery Count (10), Maximum size (1 GB), and Queue state (Active). There are 'Save changes' and 'Discard changes' buttons at the top.

Programmatically, you can configure the size of the duplicate detection window during which message-ids are retained, using the [QueueDescription.DuplicateDetectionHistoryTimeWindow](#) property with the full .NET Framework API. With the Azure Resource Manager API, the value is set with the [queueProperties.duplicateDetectionHistoryTimeWindow](#) property.

Note that enabling duplicate detection and the size of the window directly impact the queue (and topic) throughput, since all recorded message-ids must be matched against the newly submitted message identifier.

Keeping the window small means that fewer message-ids must be retained and matched, and throughput is impacted less. For high throughput entities that require duplicate detection, you should keep the window as small as possible.

## Next steps

To learn more about Service Bus messaging, see the following topics:

- [Service Bus fundamentals](#)
- [Service Bus queues, topics, and subscriptions](#)
- [Get started with Service Bus queues](#)
- [How to use Service Bus topics and subscriptions](#)

# Message counters

1/26/2018 • 1 min to read • [Edit Online](#)

You can retrieve the count of messages held in queues and subscriptions by using Azure Resource Manager and the Service Bus [NamespaceManager](#) APIs in the .NET Framework SDK.

With PowerShell, you can obtain the count as follows:

```
(Get-AzureRmServiceBusQueue -ResourceGroup mygrp -NamespaceName myns -QueueName myqueue).CountDetails
```

## Message count details

Knowing the active message count is useful in determining whether a queue builds up a backlog that requires more resources to process than what has currently been deployed. The following counter details are available in the [MessageCountDetails](#) class:

- [ActiveMessageCount](#): Messages in the queue or subscription that are in the active state and ready for delivery.
- [DeadLetterMessageCount](#): Messages in the dead-letter queue.
- [ScheduledMessageCount](#): Messages in the scheduled state.
- [TransferDeadLetterMessageCount](#): Messages that failed transfer into another queue or topic and have been moved into the transfer dead-letter queue.
- [TransferMessageCount](#): Messages pending transfer into another queue or topic.

If an application wants to scale resources based on the length of the queue, it should do so with a very measured pace. The acquisition of the message counters is an expensive operation inside the message broker, and executing it frequently directly and adversely impacts the entity performance.

## Next steps

To learn more about Service Bus messaging, see the following topics:

- [Service Bus fundamentals](#)
- [Service Bus queues, topics, and subscriptions](#)
- [Get started with Service Bus queues](#)
- [How to use Service Bus topics and subscriptions](#)

# Message deferral

1/26/2018 • 2 min to read • [Edit Online](#)

When a queue or subscription client receives a message that it is willing to process, but for which processing is not currently possible due to special circumstances inside of the application, it has the option of "deferring" retrieval of the message to a later point. The message remains in the queue or subscription, but it is set aside.

Deferral is a feature specifically created for workflow processing scenarios. Workflow frameworks may require certain operations to be processed in a particular order, and may have to postpone processing of some received messages until prescribed prior work that is informed by other messages has been completed.

A simple illustrative example for this is an order processing sequence in which a payment notification from an external payment provider appears in a system before the matching purchase order has been propagated from the store front to the fulfillment system. In that case, the fulfillment system might defer processing the payment notification until there is an order with which to associate it. In rendezvous scenarios, where messages from different sources drive a workflow forward, the real-time execution order may indeed be correct, but the messages reflecting the outcomes may arrive out of order.

Ultimately, deferral aids in re-ordering messages from the arrival order into an order in which they can be processed, while leaving those messages safely in the message store for which processing needs to be postponed.

## Message deferral APIs

The API is [BrokeredMessage.Defer](#) or [BrokeredMessage.DeferAsync](#) in the .NET Framework client, [MessageReceiver.DeferAsync](#) in the .NET Standard client, and [mesageReceiver.defer](#) or [messageReceiver.deferSync](#) in the Java client.

Deferred messages remain in the main queue along with all other active messages (unlike dead-letter messages that live in a sub-queue), but they can no longer be received using the regular Receive/ReceiveAsync functions. Deferred messages can be discovered via [message browsing](#) if an application loses track of them.

To retrieve a deferred message, its owner is responsible for remembering the [SequenceNumber](#) as it defers it. Any receiver that knows the sequence number of a deferred message can later receive the message explicitly with `Receive(sequenceNumber)`.

If a message cannot be processed because a particular resource for handling that message is temporarily unavailable but message processing should not be summarily suspended, a way to put that message on the side for a few minutes is to remember the [SequenceNumber](#) in a [scheduled message](#) to be posted in a few minutes, and re-retrieve the deferred message when the scheduled message arrives. Note that if a message handler depends on a database for all operations and that database is temporarily unavailable, it should not use deferral, but rather suspend receiving messages altogether until the database is available again.

Deferring messages does not impact the message expiration, meaning that deferred messages still expire at the initially scheduled time and are then moved into the dead-letter queue, if so configured.

## Next steps

To learn more about Service Bus messaging, see the following topics:

- [Service Bus fundamentals](#)
- [Service Bus queues, topics, and subscriptions](#)
- [Get started with Service Bus queues](#)

- How to use Service Bus topics and subscriptions

# Message browsing

1/26/2018 • 2 min to read • [Edit Online](#)

Message browsing ("peeking") enables a Service Bus client to enumerate all messages residing in a queue or subscription, typically for diagnostic and debugging purposes.

The peek operations return all messages that exist in the queue or subscription message log, not only those available for immediate acquisition with `Receive()` or the `OnMessage()` loop. The `State` property of each message tells you whether the message is active (available to be received), `deferred`, or `scheduled`.

Consumed and expired messages are cleaned up by an asynchronous "garbage collection" run and not necessarily exactly when messages expire, and therefore `Peek` may indeed return messages that have already expired and will be removed or dead-lettered when a receive operation is next invoked on the queue or subscription.

This is especially important to keep in mind when attempting to recover deferred messages from the queue. A message for which the `ExpiresAtUtc` instant has passed is no longer eligible for regular retrieval by any other means, even when it is being returned by Peek. Returning these messages is deliberate, since Peek is a diagnostics tool reflecting the current state of the log.

Peek also returns messages that were locked and are currently being processed by other receivers, but have not yet been completed. However, because Peek returns a disconnected snapshot, the lock state of a message cannot be observed on peeked messages, and the `LockedUntilUtc` and `LockToken` properties throw an `InvalidOperationException` when the application attempts to read them.

## Peek APIs

The `Peek/PeekAsync` and `PeekBatch/PeekBatchAsync` methods exist in all .NET and Java client libraries and on all receiver objects: **MessageReceiver**, **MessageSession**, **QueueClient**, and **SubscriptionClient**. Peek works on all queues and subscriptions and their respective dead-letter queues.

When called repeatedly, the Peek method enumerates all messages that exist in the queue or subscription log, in sequence number order, from the lowest available sequence number to the highest. This is the order in which messages were enqueued; it is not the order in which messages might eventually be retrieved.

`PeekBatch` retrieves multiple messages and returns them as an enumeration. If no messages are available, the enumeration object is empty, not null.

You can also seed an overload of the method with a `SequenceNumber` at which to start, and then call the parameterless method overload to enumerate further. `PeekBatch` functions equivalently, but retrieves a set of messages all at once.

## Next steps

To learn more about Service Bus messaging, see the following topics:

- [Service Bus fundamentals](#)
- [Service Bus queues, topics, and subscriptions](#)
- [Get started with Service Bus queues](#)
- [How to use Service Bus topics and subscriptions](#)

# Chaining Service Bus entities with auto-forwarding

2/23/2018 • 3 min to read • [Edit Online](#)

The Service Bus *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.

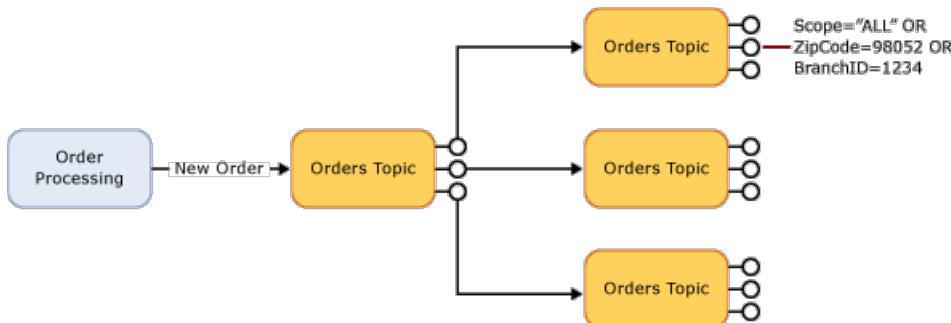
## Using auto-forwarding

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:

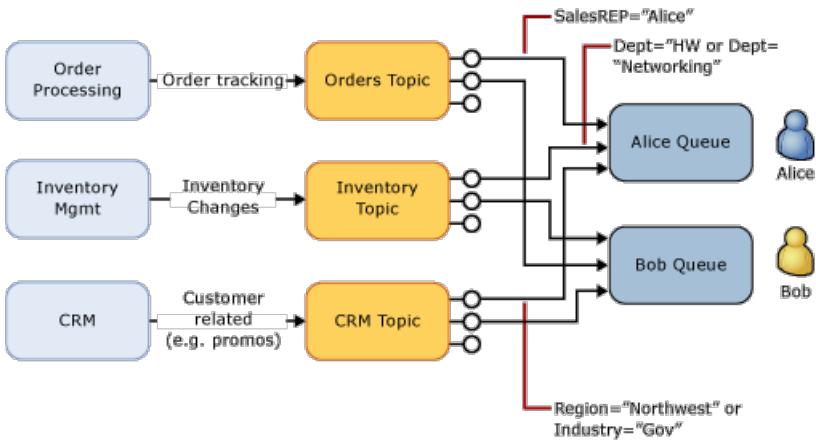
```
SubscriptionDescription srcSubscription = new SubscriptionDescription (srcTopic, srcSubscriptionName);
srcSubscription.ForwardTo = destTopic;
namespaceManager.CreateSubscription(srcSubscription));
```

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. 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 accumulates too 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 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:

- [ForwardTo](#)
- [QueueDescription](#)
- [SubscriptionDescription](#)

To learn more about Service Bus performance improvements, see

- [Best Practices for performance improvements using Service Bus Messaging](#)
- [Partitioned messaging entities](#).

# Overview of Service Bus transaction processing

2/23/2018 • 3 min to read • [Edit Online](#)

This article discusses the transaction capabilities of Microsoft 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 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:

- [How to use 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](#)

# Paired namespace implementation details and cost implications

12/21/2017 • 5 min to read • [Edit Online](#)

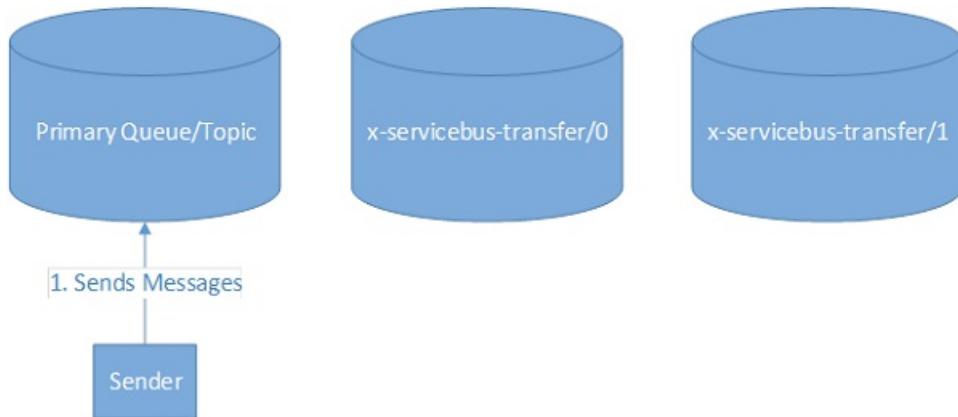
The [PairNamespaceAsync](#) method, using a [SendAvailabilityPairedNamespaceOptions](#) instance, performs visible tasks on your behalf. Because there are cost considerations when using the feature, it is useful to understand those tasks so that you expect the behavior when it happens. The API engages the following automatic behavior on your behalf:

- Creation of backlog queues.
- Creation of a [MessageSender](#) object that talks to queues or topics.
- When a messaging entity becomes unavailable, sends ping messages to the entity in an attempt to detect when that entity becomes available again.
- Optionally creates of a set of “message pumps” that move messages from the backlog queues to the primary queues.
- Coordinates closing/faulting of the primary and secondary [MessagingFactory](#) instances.

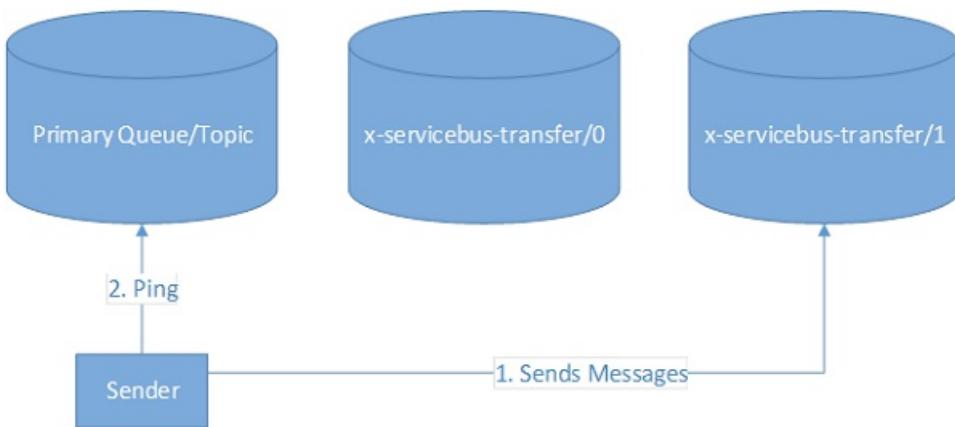
At a high level, the feature works as follows: when the primary entity is healthy, no behavior changes occur. When the [FailoverInterval](#) duration elapses, and the primary entity sees no successful sends after a non-transient [MessagingException](#) or a [TimeoutException](#), the following behavior occurs:

1. Send operations to the primary entity are disabled and the system pings the primary entity until pings can be successfully delivered.
2. A random backlog queue is selected.
3. [BrokeredMessage](#) objects are routed to the chosen backlog queue.
4. If a send operation to the chosen backlog queue fails, that queue is pulled from the rotation and a new queue is selected. All senders on the [MessagingFactory](#) instance learn of the failure.

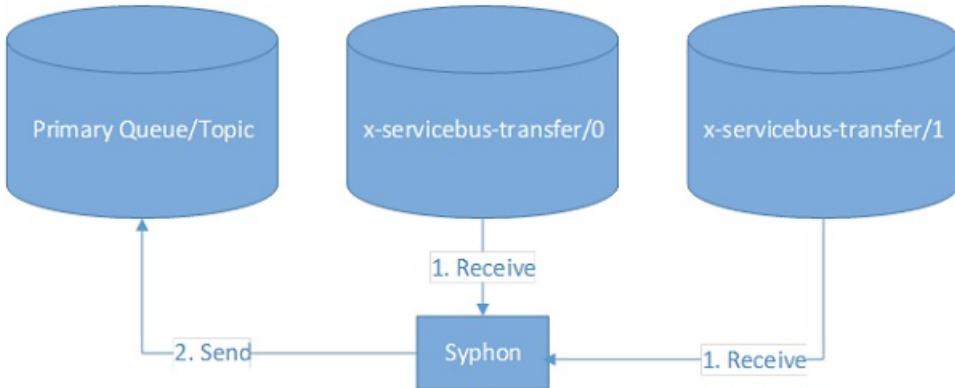
The following figures depict the sequence. First, the sender sends messages.



Upon failure to send to the primary queue, the sender begins sending messages to a randomly chosen backlog queue. Simultaneously, it starts a ping task.



At this point the messages are still in the secondary queue and have not been delivered to the primary queue. Once the primary queue is healthy again, at least one process should be running the siphon. The siphon delivers the messages from all the various backlog queues to the proper destination entities (queues and topics).



The remainder of this topic discusses the specific details of how these pieces work.

## Creation of backlog queues

The [SendAvailabilityPairedNamespaceOptions](#) object passed to the [PairNamespaceAsync](#) method indicates the number of backlog queues you want to use. Each backlog queue is then created with the following properties explicitly set (all other values are set to the [QueueDescription](#) defaults):

PATH	[PRIMARY NAMESPACE]/X-SERVICEBUS-TRANSFER/[INDEX] WHERE [INDEX] IS A VALUE IN [0, BACKLOGQUEUCOUNT)
MaxSizeInMegabytes	5120
MaxDeliveryCount	int.MaxValue
DefaultMessageTimeToLive	TimeSpan.MaxValue
AutoDeleteOnIdle	TimeSpan.MaxValue
LockDuration	1 minute

PATH	[PRIMARY NAMESPACE]/X-SERVICEBUS-TRANSFER/[INDEX] WHERE [INDEX] IS A VALUE IN [0, BACKLOGQUEUCOUNT]
EnableDeadLetteringOnMessageExpiration	true
EnableBatchedOperations	true

For example, the first backlog queue created for namespace **contoso** is named `contoso/x-servicebus-transfer/0`.

When creating the queues, the code first checks to see if such a queue exists. If the queue does not exist, then the queue is created. The code does not clean up "extra" backlog queues. Specifically, if the application with the primary namespace **contoso** requests five backlog queues but a backlog queue with the path

`contoso/x-servicebus-transfer/7` exists, that extra backlog queue is still present but is not used. The system explicitly allows extra backlog queues to exist that would not be used. As the namespace owner, you are responsible for cleaning up any unused/unwanted backlog queues. The reason for this decision is that Service Bus cannot know what purposes exist for all the queues in your namespace. Furthermore, if a queue exists with the given name but does not meet the assumed [QueueDescription](#), then your reasons are your own for changing the default behavior. No guarantees are made for modifications to the backlog queues by your code. Make sure to test your changes thoroughly.

## Custom MessageSender

When sending, all messages go through an internal [MessageSender](#) object that behaves normally when everything works, and redirects to the backlog queues when things "break." Upon receiving a non-transient failure, a timer starts. After a [TimeSpan](#) period consisting of the [FailoverInterval](#) property value during which no successful messages are sent, the failover is engaged. At this point, the following things happen for each entity:

- A ping task executes every [PingPrimaryInterval](#) to check if the entity is available. Once this task succeeds, all client code that uses the entity immediately starts sending new messages to the primary namespace.
- Future requests to send to that same entity from any other senders will result in the [BrokeredMessage](#) being sent to be modified to sit in the backlog queue. The modification removes some properties from the [BrokeredMessage](#) object and stores them elsewhere. The following properties are cleared and added under a new alias, allowing Service Bus and the SDK to process messages uniformly:

OLD PROPERTY NAME	NEW PROPERTY NAME
SessionId	x-ms-sessionid
TimeToLive	x-ms-timetolive
ScheduledEnqueueTimeUtc	x-ms-path

The original destination path is also stored within the message as a property named x-ms-path. This design allows messages for many entities to coexist in a single backlog queue. The properties are translated back by the syphon.

The custom [MessageSender](#) object can encounter issues when messages approach the 256-KB limit and failover is engaged. The custom [MessageSender](#) object stores messages for all queues and topics together in the backlog queues. This object mixes messages from many primaries together within the backlog queues. To handle load balancing among many clients that do not know each other, the SDK randomly picks one backlog queue for each [QueueClient](#) or [TopicClient](#) you create in code.

## Pings

A ping message is an empty [BrokeredMessage](#) with its [ContentType](#) property set to application/vnd.ms-

servicebus-ping and a [TimeToLive](#) value of 1 second. This ping has one special characteristic in Service Bus: the server never delivers a ping when any caller requests a [BrokeredMessage](#). Thus, you never have to learn how to receive and ignore these messages. Each entity (unique queue or topic) per [MessagingFactory](#) instance per client will be pinged when they are considered to be unavailable. By default, this happens once per minute. Ping messages are considered to be regular Service Bus messages, and can result in charges for bandwidth and messages. As soon as the clients detect that the system is available, the messages stop.

## The syphon

At least one executable program in the application should be actively running the syphon. The syphon performs a long poll receive that lasts 15 minutes. When all entities are available and you have 10 backlog queues, the application that hosts the syphon calls the receive operation 40 times per hour, 960 times per day, and 28800 times in 30 days. When the syphon is actively moving messages from the backlog to the primary queue, each message experiences the following charges (standard charges for message size and bandwidth apply in all stages):

1. Send to the backlog.
2. Receive from the backlog.
3. Send to the primary.
4. Receive from the primary.

## Close/fault behavior

Within an application that hosts the syphon, once the primary or secondary [MessagingFactory](#) faults or is closed without its partner also being faulted or closed and the syphon detects this state, the syphon acts. If the other [MessagingFactory](#) is not closed within 5 seconds, the syphon faults the still open [MessagingFactory](#).

## Next steps

See [Asynchronous messaging patterns and high availability](#) for a detailed discussion of Service Bus asynchronous messaging.

# Distributed tracing and correlation through Service Bus messaging

1/9/2018 • 9 min to read • [Edit Online](#)

One of the common problems in microservices development is the ability to trace operation from a client through all the services that are involved in processing. It's useful for debugging, performance analysis, A/B testing, and other typical diagnostics scenarios. One part of this problem is tracking logical pieces of work. It includes message processing result and latency and external dependency calls. Another part is correlation of these diagnostics events beyond process boundaries.

When a producer sends a message through a queue, it typically happens in the scope of some other logical operation, initiated by some other client or service. The same operation is continued by consumer once it receives a message. Both producer and consumer (and other services that process the operation), presumably emit telemetry events to trace the operation flow and result. In order to correlate such events and trace operation end-to-end, each service that reports telemetry has to stamp every event with a trace context.

Microsoft Azure Service Bus messaging has defined payload properties that producers and consumers should use to pass such trace context. The protocol is based on the [HTTP Correlation protocol](#).

PROPERTY NAME	DESCRIPTION
Diagnostic-Id	Unique identifier of an external call from producer to the queue. Refer to <a href="#">Request-Id in HTTP protocol</a> for the rationale, considerations, and format
Correlation-Context	Operation context, which is propagated across all services involved in operation processing. For more information, see <a href="#">Correlation-Context in HTTP protocol</a>

## Service Bus .NET Client auto-tracing

Starting with version 3.0.0 [Microsoft Azure ServiceBus Client for .NET](#) provides tracing instrumentation points that can be hooked by tracing systems, or piece of client code. The instrumentation allows tracking all calls to the Service Bus messaging service from client side. If message processing is done with the [message handler pattern](#), message processing is also instrumented

### Tracking with Azure Application Insights

[Microsoft Application Insights](#) provides rich performance monitoring capabilities including automagical request and dependency tracking.

Depending on your project type, install Application Insights SDK:

- [ASP.NET](#) version 2.5-beta2 or higher
- [ASP.NET Core](#) version 2.2.0-beta2 or higher. These links provide details on installing SDK, creating resources, and configuring SDK (if needed). For non-ASP.NET applications, refer to [Azure Application Insights for Console Applications](#) article.

If you use [message handler pattern](#) to process messages, you are done: all Service Bus calls done by your service are automatically tracked and correlated with other telemetry items. Otherwise refer to the following example for manual message processing tracking.

## Trace message processing

```
private readonly TelemetryClient telemetryClient;

async Task ProcessAsync(Message message)
{
 var activity = message.ExtractActivity();

 // If you are using Microsoft.ApplicationInsights package version 2.6-beta or higher, you should call
 StartOperation<RequestTelemetry>(activity) instead
 using (var operation = telemetryClient.StartOperation<RequestTelemetry>("Process", activity.RootId,
 activity.ParentId))
 {
 telemetryClient.TrackTrace("Received message");
 try
 {
 // process message
 }
 catch (Exception ex)
 {
 telemetryClient.TrackException(ex);
 operation.Telemetry.Success = false;
 throw;
 }

 telemetryClient.TrackTrace("Done");
 }
}
```

In this example, `RequestTelemetry` is reported for each processed message, having a timestamp, duration, and result (success). The telemetry also has a set of correlation properties. Nested traces and exceptions reported during message processing are also stamped with correlation properties representing them as 'children' of the `RequestTelemetry`.

In case you make calls to supported external components during message processing, they are also automatically tracked and correlated. Refer to [Track custom operations with Application Insights .NET SDK](#) for manual tracking and correlation.

## Tracking without tracing system

In case your tracing system does not support automagical Service Bus calls tracking you may be looking into adding such support into a tracing system or into your application. This section describes diagnostics events sent by Service Bus .NET client.

Service Bus .NET Client is instrumented using .NET tracing primitives `System.Diagnostics.Activity` and `System.Diagnostics.DiagnosticSource`.

`Activity` serves as a trace context while `DiagnosticSource` is a notification mechanism.

If there is no listener for the `DiagnosticSource` events, instrumentation is off, keeping zero instrumentation costs. `DiagnosticSource` gives all control to the listener:

- listener controls which sources and events to listen to
- listener controls event rate and sampling
- events are sent with a payload that provides full context so you can access and modify `Message` object during the event

Familiarize yourself with [DiagnosticSource User Guide](#) before proceeding with implementation.

Let's create a listener for Service Bus events in ASP.NET Core app that writes logs with `Microsoft.Extension.Logger`. It uses `System.Reactive.Core` library to subscribe to `DiagnosticSource` (it's also easy to subscribe to `DiagnosticSource` without it)

```

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory factory,
IApplicationLifetime applicationLifetime)
{
 // configuration...

 var serviceBusLogger = factory.CreateLogger("Microsoft.Azure.ServiceBus");

 IDisposable innerSubscription = null;
 IDisposable outerSubscription = DiagnosticListener.AllListeners.Subscribe(delegate (DiagnosticListener
listener)
 {
 // subscribe to the Service Bus DiagnosticSource
 if (listener.Name == "Microsoft.Azure.ServiceBus")
 {
 // receive event from Service Bus DiagnosticSource
 innerSubscription = listener.Subscribe(delegate (KeyValuePair<string, object> evnt)
 {
 // Log operation details once it's done
 if (evnt.Key.EndsWith("Stop"))
 {
 Activity currentActivity = Activity.Current;
 TaskStatus status = (TaskStatus)evnt.Value.GetProperty("Status");
 serviceBusLogger.LogInformation($"Operation {currentActivity.OperationName} is finished,
Duration={currentActivity.Duration}, Status={status}, Id={currentActivity.Id}, StartTime=
{currentActivity.StartTimeUtc}");
 }
 });
 }
 });

 applicationLifetime.ApplicationStopping.Register(() =>
 {
 outerSubscription?.Dispose();
 innerSubscription?.Dispose();
 });
}

```

In this example, listener logs duration, result, unique identifier, and start time for each Service Bus operation.

#### Events

For every operation, two events are sent: 'Start' and 'Stop'. Most probably, you are only interested in 'Stop' events. They provide the result of operation, as well as start time and duration as an Activity properties.

Event payload provides a listener with the context of the operation, it replicates API incoming parameters and return value. 'Stop' event payload has all the properties of 'Start' event payload, so you can ignore 'Start' event completely.

All events also have 'Entity' and 'Endpoint' properties, they are omitted in below table

- `string Entity` -- Name of the entity (queue, topic, etc.)
- `Uri Endpoint` - Service Bus endpoint URL

Each 'Stop' event has `Status` property with `TaskStatus` async operation was completed with, that is also omitted in the following table for simplicity.

Here is the full list of instrumented operations:

OPERATION NAME	TRACKED API	SPECIFIC PAYLOAD PROPERTIES
Microsoft.Azure.ServiceBus.Send	<a href="#">MessageSender.SendAsync</a>	IList Messages - List of messages being sent

OPERATION NAME	TRACKED API	SPECIFIC PAYLOAD PROPERTIES
Microsoft.Azure.ServiceBus.ScheduleMessage	<a href="#">MessageSender.ScheduleMessageAsync</a>	Message Message - Message being processed DateTimeOffset ScheduleEnqueueTimeUtc - Scheduled message offset long SequenceNumber - Sequence number of scheduled message ('Stop' event payload)
Microsoft.Azure.ServiceBus.Cancel	<a href="#">MessageSender.CancelScheduledMessageAsync</a>	long SequenceNumber - Sequence number of the message to be canceled
Microsoft.Azure.ServiceBus.Receive	<a href="#">MessageReceiver.ReceiveAsync</a>	int RequestedMessageCount - The maximum number of messages that could be received. IList Messages - List of received messages ('Stop' event payload)
Microsoft.Azure.ServiceBus.Peek	<a href="#">MessageReceiver.PeekAsync</a>	int FromSequenceNumber - The starting point from which to browse a batch of messages. int RequestedMessageCount - The number of messages to retrieve. IList Messages - List of received messages ('Stop' event payload)
Microsoft.Azure.ServiceBus.ReceiveDeferred	<a href="#">MessageReceiver.ReceiveDeferredMessageAsync</a>	IEnumerable SequenceNumbers - The list containing the sequence numbers to receive. IList Messages - List of received messages ('Stop' event payload)
Microsoft.Azure.ServiceBus.Complete	<a href="#">MessageReceiver.CompleteAsync</a>	IList LockTokens - The list containing the lock tokens of the corresponding messages to complete.
Microsoft.Azure.ServiceBus.Abandon	<a href="#">MessageReceiver.AbandonAsync</a>	string LockToken - The lock token of the corresponding message to abandon.
Microsoft.Azure.ServiceBus.Defer	<a href="#">MessageReceiver.DeferAsync</a>	string LockToken - The lock token of the corresponding message to defer.
Microsoft.Azure.ServiceBus.DeadLetter	<a href="#">MessageReceiver.DeadLetterAsync</a>	string LockToken - The lock token of the corresponding message to dead letter.
Microsoft.Azure.ServiceBus.RenewLock	<a href="#">MessageReceiver.RenewLockAsync</a>	string LockToken - The lock token of the corresponding message to renew lock on. DateTime LockedUntilUtc - New lock token expiry date and time in UTC format. ('Stop' event payload)
Microsoft.Azure.ServiceBus.Process	Message Handler lambda function provided in <a href="#">IReceiverClient.RegisterMessageHandler</a>	Message Message - Message being processed.

OPERATION NAME	TRACKED API	SPECIFIC PAYLOAD PROPERTIES
Microsoft.Azure.ServiceBus.ProcessSession	Message Session Handler lambda function provided in <a href="#">IQueueClient.RegisterSessionHandler</a>	Message Message - Message being processed. IMessageSession Session - Session being processed
Microsoft.Azure.ServiceBus.AddRule	<a href="#">SubscriptionClient.AddRuleAsync</a>	RuleDescription Rule - The rule description that provides the rule to add.
Microsoft.Azure.ServiceBus.RemoveRule	<a href="#">SubscriptionClient.RemoveRuleAsync</a>	string RuleName - Name of the rule to remove.
Microsoft.Azure.ServiceBus.GetRules	<a href="#">SubscriptionClient.GetRulesAsync</a>	IEnumerable Rules- All rules associated with the subscription. ('Stop' payload only)
Microsoft.Azure.ServiceBus.AcceptMessageSession	<a href="#">ISessionClient.AcceptMessageSessionAsync</a>	string SessionId - The sessionId present in the messages.
Microsoft.Azure.ServiceBus.GetSessionState	<a href="#">IMessageSession.GetStateAsync</a>	string SessionId - The sessionId present in the messages. byte [] State - Session state ('Stop' event payload)
Microsoft.Azure.ServiceBus.SetSessionState	<a href="#">IMessageSession.SetStateAsync</a>	string SessionId - The sessionId present in the messages. byte [] State - Session state
Microsoft.Azure.ServiceBus.RenewSessionLock	<a href="#">IMessageSession.RenewSessionLockAsync</a>	string SessionId - The sessionId present in the messages.
Microsoft.Azure.ServiceBus.Exception	any instrumented API	Exception Exception - Exception instance

In every event, you can access `Activity.Current` that holds current operation context.

#### Logging additional properties

`Activity.Current` provides detailed context of current operation and its parents. For more information, see [Activity documentation](#) for more details. Service Bus instrumentation provides additional information in the `Activity.Current.Tags` - they hold `MessageId` and `SessionId` whenever they are available.

Activities that track 'Receive', 'Peek' and 'ReceiveDeferred' event also may have `RelatedTo` tag. It holds distinct list of `Diagnostic-Id` (s) of messages that were received as a result. Such operation may result in several unrelated messages to be received. Also, the `Diagnostic-Id` is not known when operation starts, so 'Receive' operations could be correlated to 'Process' operations using this Tag only. It's useful when analyzing performance issues to check how long it took to receive the message.

Efficient way to log Tags is to iterate over them, so adding Tags to the preceding example looks like

```

Activity currentActivity = Activity.Current;
TaskStatus status = (TaskStatus)evnt.Value.GetProperty("Status");

var tagsList = new StringBuilder();
foreach (var tag in currentActivity.Tags)
{
 tagsList.Append($"{tag.Key}={tag.Value}");
}

serviceBusLogger.LogInformation($"{{currentActivity.OperationName}} is finished, Duration={{currentActivity.Duration}}, Status={{status}}, Id={{currentActivity.Id}}, StartTime={{currentActivity.StartTimeUtc}} {{tagsList}}");

```

## Filtering and sampling

In some cases, it's desirable to log only part of the events to reduce performance overhead or storage consumption. You could log 'Stop' events only (as in preceding example) or sample percentage of the events.

`DiagnosticSource` provide way to achieve it with `.IsEnabled` predicate. For more information, see [Context-Based Filtering in DiagnosticSource](#).

`.IsEnabled` may be called multiple times for a single operation to minimize performance impact.

`.IsEnabled` is called in following sequence:

1. `.IsEnabled(<OperationName>, string entity, null)` for example, `.IsEnabled("Microsoft.Azure.ServiceBus.Send", "MyQueue1")`. Note there is no 'Start' or 'Stop' at the end. Use it to filter out particular operations or queues. If callback returns `false`, events for the operation are not sent
  - For the 'Process' and 'ProcessSession' operations, you also receive `.IsEnabled(<OperationName>, string entity, Activity activity)` callback. Use it to filter events based on `activity.Id` or `Tags` properties.
2. `.IsEnabled(<OperationName>.Start)` for example, `.IsEnabled("Microsoft.Azure.ServiceBus.Send.Start")`. Checks whether 'Start' event should be fired. The result only affects 'Start' event, but further instrumentation does not depend on it.

There is no `.IsEnabled` for 'Stop' event.

If some operation result is exception, `.IsEnabled("Microsoft.Azure.ServiceBus.Exception")` is called. You could only subscribe to 'Exception' events and prevent the rest of the instrumentation. In this case, you still have to handle such exceptions. Since other instrumentation is disabled, you should not expect trace context to flow with the messages from consumer to producer.

You can use `.IsEnabled` also implement sampling strategies. Sampling based on the `Activity.Id` or `Activity.RootId` ensures consistent sampling across all tiers (as long as it is propagated by tracing system or by your own code).

In presence of multiple `DiagnosticSource` listeners for the same source, it's enough for just one listener to accept the event, so `.IsEnabled` is not guaranteed to be called,

## Next steps

- [Service Bus fundamentals](#)
- [Application Insights Correlation](#)
- [Application Insights Monitor Dependencies](#) to see if REST, SQL, or other external resources are slowing you down.
- [Track custom operations with Application Insights .NET SDK](#)

# Azure Service Bus metrics in Azure Monitor (preview)

4/9/2018 • 3 min to read • [Edit Online](#)

Service Bus metrics gives you the state of resources in your Azure subscription. With a rich set of metrics data, you can assess the overall health of your Service Bus resources, not only at the namespace level, but also at the entity level. These statistics can be important as they help you to monitor the state of Service Bus. Metrics can also help troubleshoot root-cause issues without needing to contact Azure support.

Azure Monitor provides unified user interfaces for monitoring across various Azure services. For more information, see [Monitoring in Microsoft Azure](#) and the [Retrieve Azure Monitor metrics with .NET sample](#) on GitHub.

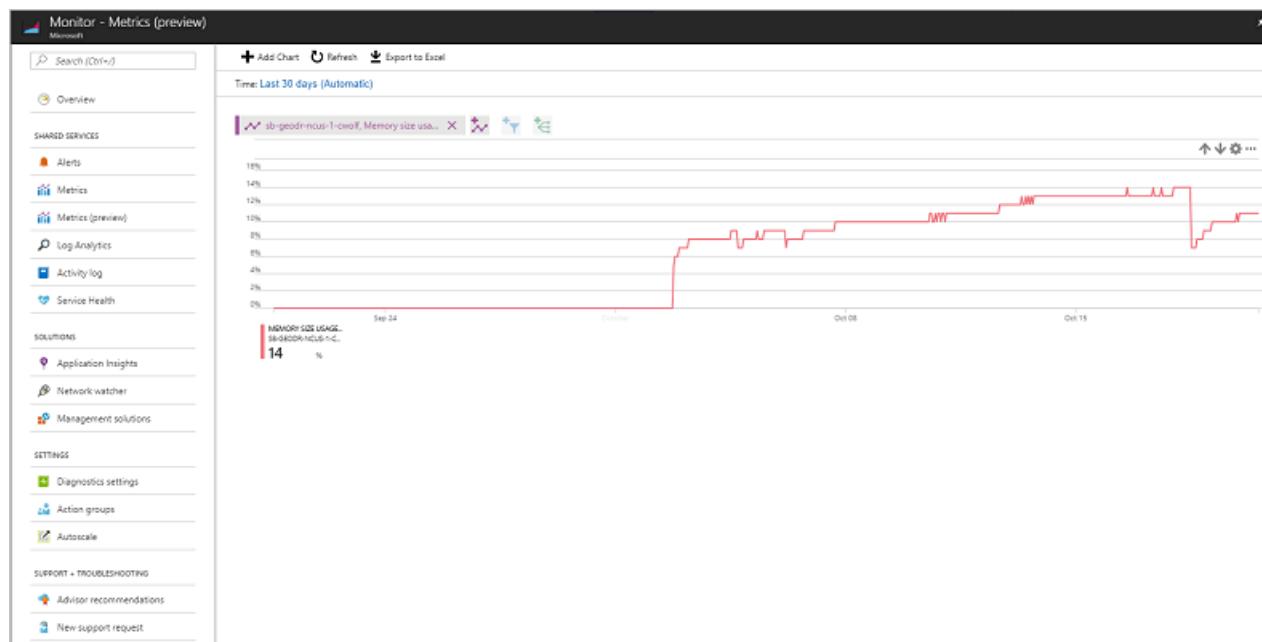
## Access metrics

Azure Monitor provides multiple ways to access metrics. You can either access metrics through the [Azure portal](#), or use the Azure Monitor APIs (REST and .NET) and analysis solutions such as Log Analytics and Event Hubs. For more information, see [Azure Monitor metrics](#).

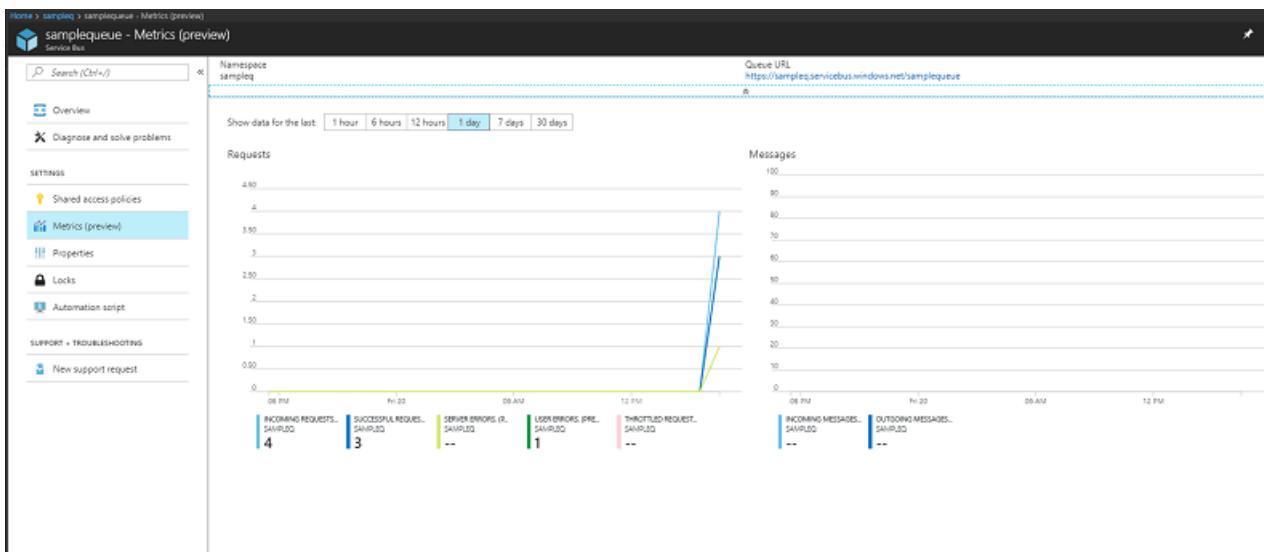
Metrics are enabled by default, and you can access the most recent 30 days of data. If you need to retain data for a longer period of time, you can archive metrics data to an Azure Storage account. This is configured in [diagnostic settings](#) in Azure Monitor.

## Access metrics in the portal

You can monitor metrics over time in the [Azure portal](#). The following example shows how to view successful requests and incoming requests at the account level:



You can also access metrics directly via the namespace. To do so, select your namespace and then click **Metrics (Preview)**. To display metrics filtered to the scope of the entity, select the entity and then click **Metrics (preview)**.



For metrics supporting dimensions, you must filter with the desired dimension value.

## Billing

Using metrics in Azure Monitor is free while in preview. However, if you use additional solutions that ingest metrics data, you may be billed by these solutions. For example, you are billed by Azure Storage if you archive metrics data to an Azure Storage account. You are also billed by Log Analytics if you stream metrics data to Log Analytics for advanced analysis.

The following metrics give you an overview of the health of your service.

### NOTE

We are deprecating several metrics as they are moved under a different name. This might require you to update your references. Metrics marked with the "deprecated" keyword will not be supported going forward.

All metrics values are sent to Azure Monitor every minute. The time granularity defines the time interval for which metrics values are presented. The supported time interval for all Service Bus metrics is 1 minute.

## Request metrics

Counts the number of data and management operations requests.

METRIC NAME	DESCRIPTION
Incoming Requests (preview)	<p>The number of requests made to the Service Bus service over a specified period.</p> <p>Unit: Count Aggregation Type: Total Dimension: EntityName</p>
Successful Requests (preview)	<p>The number of successful requests made to the Service Bus service over a specified period.</p> <p>Unit: Count Aggregation Type: Total Dimension: EntityName</p>

METRIC NAME	DESCRIPTION
Server Errors (preview)	<p>The number of requests not processed due to an error in the Service Bus service over a specified period.</p> <p>Unit: Count Aggregation Type: Total Dimension: EntityName</p>
User Errors (preview)	<p>The number of requests not processed due to user errors over a specified period.</p> <p>Unit: Count Aggregation Type: Total Dimension: EntityName</p>
Throttled Requests (preview)	<p>The number of requests that were throttled because the usage was exceeded.</p> <p>Unit: Count Aggregation Type: Total Dimension: EntityName</p>

## Message metrics

METRIC NAME	DESCRIPTION
Incoming Messages (preview)	<p>The number of events or messages sent to Service Bus over a specified period.</p> <p>Unit: Count Aggregation Type: Total Dimension: EntityName</p>
Outgoing Messages (preview)	<p>The number of events or messages received from Service Bus over a specified period.</p> <p>Unit: Count Aggregation Type: Total Dimension: EntityName</p>

## Connection metrics

METRIC NAME	DESCRIPTION
ActiveConnections (preview)	<p>The number of active connections on a namespace as well as on an entity.</p> <p>Unit: Count Aggregation Type: Total Dimension: EntityName</p>
Connections Opened (preview)	<p>The number of open connections.</p> <p>Unit: Count Aggregation Type: Total Dimension: EntityName</p>

METRIC NAME	DESCRIPTION
Connections Closed (preview)	The number of closed connections.  Unit: Count Aggregation Type: Total Dimension: EntityName

## Resource usage metrics

METRIC NAME	DESCRIPTION
CPU usage per namespace (preview)	The percentage CPU usage of the namespace.  Unit: Percent Aggregation Type: Maximum Dimension: EntityName
Memory size usage per namespace (preview)	The percentage memory usage of the namespace.  Unit: Percent Aggregation Type: Maximum Dimension: EntityName

## Metrics dimensions

Azure Service Bus supports the following dimensions for metrics in Azure Monitor. Adding dimensions to your metrics is optional. If you do not add dimensions, metrics are specified at the namespace level.

DIMENSION NAME	DESCRIPTION
EntityName	Service Bus supports messaging entities under the namespace.

## Next steps

See the [Azure Monitoring overview](#).

# Service Bus management libraries

2/6/2018 • 1 min to read • [Edit Online](#)

The Azure Service Bus management libraries can dynamically provision Service Bus namespaces and entities. This enables complex deployments and messaging scenarios, and makes it possible 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 the Azure Active Directory (Azure AD) service. Azure AD 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 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 AD using the **Microsoft.IdentityModel.Clients.ActiveDirectory** library:

```
var context = new AuthenticationContext($"https://login.microsoftonline.com/{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 API reference](#)

# Service Bus diagnostic logs

2/6/2018 • 1 min to read • [Edit Online](#)

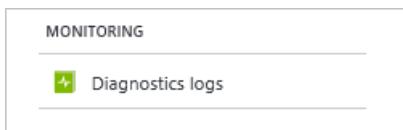
You can view two types of logs for Azure Service Bus:

- **Activity logs.** These logs contain information about operations performed on a job. The logs are always enabled.
- **Diagnostic logs.** You can configure diagnostic logs for richer information about everything that happens within 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

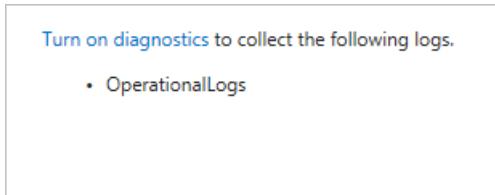
Diagnostics logs are disabled by default. To enable diagnostic logs, perform the following steps:

1. In the [Azure portal](#), under **Monitoring + Management**, click **Diagnostics logs**.

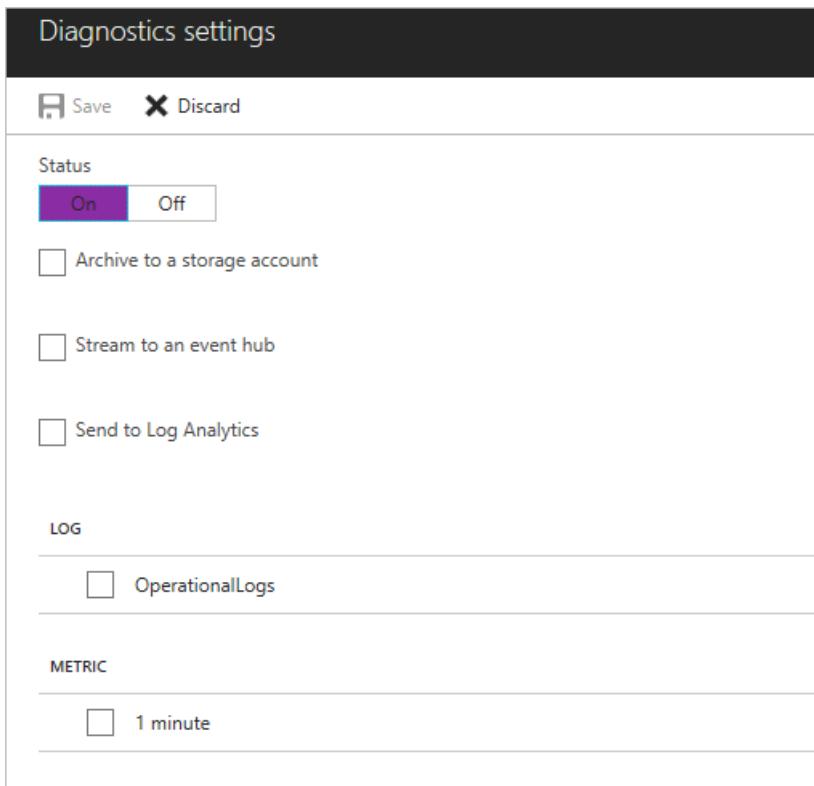


2. Click the resource you want to monitor.

3. Click **Turn on diagnostics**.



4. For **Status**, click **On**.



5. Set the archive target that you want; for example, a storage account, an event hub, or Azure Log Analytics.
6. 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 the [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 section.

## Operational logs schema

Logs in the **OperationalLogs** category capture what happens during Service Bus operations. Specifically, these logs capture the operation type, including queue creation, resources used, and the status of the operation.

Operational 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

NAME	DESCRIPTION
Status	Operation status
Caller	Caller of operation (Azure portal or management client)
category	OperationalLogs

Here's an example of an operational log JSON string:

```
{
 "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

See the following links to learn more about Service Bus:

- [Introduction to Service Bus](#)
- [Get started with Service Bus](#)

# Suspend and reactivate messaging entities (disable)

1/26/2018 • 1 min to read • [Edit Online](#)

Queues, topics, and subscriptions can be temporarily suspended. Suspension puts the entity into a disabled state in which all messages are maintained in storage. However, messages cannot be removed or added, and the respective protocol operations yield errors.

Suspending an entity is typically done for urgent administrative reasons. One scenario is having deployed a faulty receiver that takes messages off the queue, fails processing, and yet incorrectly completes the messages and removes them. If that behavior is diagnosed, the queue can be disabled for receives until corrected code is deployed and further data loss caused by the faulty code can be prevented.

A suspension or reactivation can be performed either by the user or by the system. The system only suspends entities due to grave administrative reasons such as hitting the subscription spending limit. System-disabled entities cannot be reactivated by the user, but are restored when the cause of the suspension has been addressed.

In the portal, the **Properties** section for the respective entity enables changing the state; the following screen shot shows the toggle for a queue:

The screenshot shows the 'mynewqueue - Properties' page in the Azure portal. The left sidebar lists navigation options: Overview, Diagnose and solve problems, Shared access policies, Properties (which is selected and highlighted in blue), Locks, and Automation script. The main content area is titled 'Save changes' and 'Discard changes'. It contains several configuration settings: Message time to live (default) set to 14 days, Lock duration set to 01 minutes, Duplicate detection history set to 10 minutes, Maximum Delivery Count set to 10, and Maximum size (dropdown menu). At the bottom, there is a 'Queue state' section with two buttons: 'Active' (highlighted with a red border) and 'Disabled'. A checkbox for 'Move expired messages to the dead-letter subqueue' is also present.

The portal only permits completely disabling queues. You can also disable the send and receive operations separately using the Service Bus [NamespaceManager](#) APIs in the .NET Framework SDK, or with an Azure Resource Manager template through Azure CLI or Azure PowerShell.

## Suspension states

The states that can be set for a queue are:

- **Active:** The queue is active.

- **Disabled**: The queue is suspended.
- **SendDisabled**: The queue is partially suspended, with receive being permitted.
- **ReceiveDisabled**: The queue is partially suspended, with send being permitted.

For subscriptions and topics, only **Active** and **Disabled** can be set.

The [EntityStatus](#) enumeration also defines a set of transitional states that can only be set by the system. The PowerShell command to disable a queue is shown in the following example. The reactivation command is equivalent, setting `Status` to **Active**.

```
$q = Get-AzureRmServiceBusQueue -ResourceGroup mygrp -NamespaceName myns -QueueName myqueue

$q.Status = "Disabled"

Set-AzureRmServiceBusQueue -ResourceGroup mygrp -NamespaceName myns -QueueName myqueue -QueueObj $q
```

## Next steps

To learn more about Service Bus messaging, see the following topics:

- [Service Bus fundamentals](#)
- [Service Bus queues, topics, and subscriptions](#)
- [Get started with Service Bus queues](#)
- [How to use Service Bus topics and subscriptions](#)

# Create Service Bus resources using Azure Resource Manager templates

4/18/2018 • 4 min to read • [Edit Online](#)

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 is written in JSON and consists of 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 [structure and syntax of 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 [Getting started with Azure PowerShell](#).

## Create a template

Clone the repository 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"
 }
 }
 },
 "variables": {
 "defaultSASKeyName": "RootManageSharedAccessKey",
 "authRuleResourceId": "[resourceId('Microsoft.ServiceBus/namespaces/authorizationRules',
parameters('serviceBusNamespaceName'), variables('defaultSASKeyName'))]",
 "sbVersion": "2017-04-01"
 },
 "resources": [
 {
 "apiVersion": "2017-04-01",
 "name": "[parameters('serviceBusNamespaceName')]",
 "type": "Microsoft.ServiceBus/Namespace",
 "location": "[resourceGroup().location]",
 "sku": {
 "name": "Standard"
 },
 "properties": {},
 "resources": [
 {
 "apiVersion": "2017-04-01",
 "name": "[parameters('serviceBusQueueName')]",
 "type": "Queues",
 "dependsOn": [
 "[concat('Microsoft.ServiceBus/namespaces/', parameters('serviceBusNamespaceName'))]"
],
 "properties": {
 "lockDuration": "PT5M",
 "maxSizeInMegabytes": "1024",
 "requiresDuplicateDetection": "false",
 "requiresSession": "false",
 "defaultMessageTimeToLive": "P10675199DT2H48M5.4775807S",
 "deadLetteringOnMessageExpiration": "false",
 "duplicateDetectionHistoryTimeWindow": "PT10M",
 "maxDeliveryCount": "10",
 "autoDeleteOnIdle": "P10675199DT2H48M5.4775807S",
 "enablePartitioning": "false",
 "enableExpress": "false"
 }
 }
]
 }
],
 "outputs": {
 "NamespaceConnectionString": {
 "type": "string",
 "value": "[listkeys(variables('authRuleResourceId'), variables('sbVersion')).primaryConnectionString]"
 },
 "SharedAccessPolicyPrimaryKey": {
 "type": "string",
 "value": "[listkeys(variables('authRuleResourceId'), variables('sbVersion')).primaryConnectionString]"
 }
 }
}
```

```
 "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": "2017-04-01"
 }
 }
}
```

For more information, see the [Parameters](#) article.

## Log in to Azure and set the Azure subscription

From a PowerShell prompt, run the following command:

```
Connect-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 : 4/19/2017 10:38:30 PM
Mode : Incremental
TemplateLink :
Parameters :
 Name Type Value
 ====== ===== =====
 serviceBusNamespaceName String <namespaceName>
 serviceBusQueueName String <queueName>
 serviceBusApiVersion String 2017-04-01
```

## 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](#)

# Create a Service Bus namespace using an Azure Resource Manager template

4/12/2018 • 2 min to read • [Edit Online](#)

This article describes how to use an Azure Resource Manager template that creates a Service Bus namespace of type **Messaging** with a Standard SKU. The article also defines the parameters that are specified for the execution of the deployment. You can use this template for your own deployments, or customize it to meet your requirements.

For more information about creating templates, see [Authoring Azure Resource Manager templates](#).

For the complete template, see the [Service Bus namespace template](#) on GitHub.

## NOTE

The following Azure Resource Manager templates are available for download and deployment.

- [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](#)

To check for the latest templates, visit the [Azure Quickstart Templates](#) gallery and search for Service Bus.

## What will you deploy?

With this template, you deploy a Service Bus namespace with a [Standard or Premium SKU](#).

To run the deployment automatically, click the following button:



## Parameters

With Azure Resource Manager, you define parameters for values you want to specify when the template is deployed. The template includes a section called `Parameters` that contains all of the parameter values. You should define a parameter for those values that vary based on the project you are deploying or based on the environment you are deploying to. Do not define parameters for values that always stay the same. Each parameter value is used in the template to define the resources that are deployed.

This template defines the following parameters:

### serviceBusNamespaceName

The name of the Service Bus namespace to create.

```
"serviceBusNamespaceName": {
 "type": "string",
 "metadata": {
 "description": "Name of the Service Bus namespace"
 }
}
```

## serviceBusSKU

The name of the Service Bus [SKU](#) to create.

```
"serviceBusSKU": {
 "type": "string",
 "allowedValues": [
 "Standard",
 "Premium"
],
 "defaultValue": "Standard",
 "metadata": {
 "description": "The messaging tier for service Bus namespace"
 }
}
```

The template defines the values that are permitted for this parameter (Standard or Premium). If no value is specified, the resource manager assigns a default value (Standard).

For more information about Service Bus pricing, see [Service Bus pricing and billing](#).

## serviceBusApiVersion

The Service Bus API version of the template.

```
"serviceBusApiVersion": {
 "type": "string",
 "defaultValue": "2017-04-01",
 "metadata": {
 "description": "Service Bus ApiVersion used by the template"
 }
}
```

# Resources to deploy

## Service Bus namespace

Creates a standard Service Bus namespace of type **Messaging**.

```
"resources": [
 {
 "apiVersion": "[parameters('serviceBusApiVersion')]",
 "name": "[parameters('serviceBusNamespaceName')]",
 "type": "Microsoft.ServiceBus/Namespaces",
 "location": "[variables('location')]",
 "kind": "Messaging",
 "sku": {
 "name": "Standard",
 },
 "properties": {
 }
 }
]
```

# Commands to run deployment

To deploy the resources to Azure, you must be logged in to your Azure account and you must use the Azure Resource Manager module. To learn about using Azure Resource Manager with either Azure PowerShell or Azure CLI, see:

- [Using Azure PowerShell with Azure Resource Manager](#)
- [Using the Azure CLI for Mac, Linux, and Windows with Azure Resource Management](#)

The following examples assume you already have a resource group in your account with the specified name.

## PowerShell

```
New-AzureRmResourceGroupDeployment -ResourceGroupName <resource-group-name> -TemplateFile
https://raw.githubusercontent.com/azure/azure-quickstart-templates/master/101-servicebus-create-
namespace/azureddeploy.json
```

## Azure CLI

```
azure config mode arm

azure group deployment create <my-resource-group> <my-deployment-name> --template-uri
https://raw.githubusercontent.com/azure/azure-quickstart-templates/master/101-servicebus-create-
namespace/azureddeploy.json
```

## Next steps

Now that you've created and deployed resources using Azure Resource Manager, learn how to manage these resources by reading these articles:

- [Manage Service Bus with PowerShell](#)
- [Manage Service Bus resources with the Service Bus Explorer](#)

# Create a Service Bus namespace and a queue using an Azure Resource Manager template

4/12/2018 • 2 min to read • [Edit Online](#)

This article shows how to use an Azure Resource Manager template that creates a Service Bus namespace and a queue within that namespace. The article explains how to specify which resources are deployed and how to define parameters that are specified when the deployment is executed. You can use this template for your own deployments, or customize it to meet your requirements.

For more information about creating templates, please see [Authoring Azure Resource Manager templates](#).

For the complete template, see the [Service Bus namespace and queue template](#) on GitHub.

## NOTE

The following Azure Resource Manager templates are available for download and deployment.

- [Create a Service Bus namespace with queue and authorization rule](#)
- [Create a Service Bus namespace with topic and subscription](#)
- [Create a Service Bus namespace](#)
- [Create a Service Bus namespace with topic, subscription, and rule](#)

To check for the latest templates, visit the [Azure Quickstart Templates](#) gallery and search for **Service Bus**.

## What will you deploy?

With this template, you deploy a Service Bus namespace with a queue.

[Service Bus queues](#) offer First In, First Out (FIFO) message delivery to one or more competing consumers.

To run the deployment automatically, click the following button:



## Parameters

With Azure Resource Manager, you define parameters for values you want to specify when the template is deployed. The template includes a section called `Parameters` that contains all of the parameter values. You should define a parameter for those values that will vary based on the project you are deploying or based on the environment you are deploying to. Do not define parameters for values that will always stay the same. Each parameter value is used in the template to define the resources that are deployed.

The template defines the following parameters.

### `serviceBusNamespaceName`

The name of the Service Bus namespace to create.

```
"serviceBusNamespaceName": {
 "type": "string",
 "metadata": {
 "description": "Name of the Service Bus namespace"
 }
}
```

## serviceBusQueueName

The name of the queue created in the Service Bus namespace.

```
"serviceBusQueueName": {
 "type": "string"
}
```

## serviceBusApiVersion

The Service Bus API version of the template.

```
"serviceBusApiVersion": {
 "type": "string",
 "defaultValue": "2017-04-01",
 "metadata": {
 "description": "Service Bus ApiVersion used by the template"
 }
}
```

# Resources to deploy

Creates a standard Service Bus namespace of type **Messaging**, with a queue.

```
"resources": [
 {
 "apiVersion": "[variables('sbVersion')]",
 "name": "[parameters('serviceBusNamespaceName')]",
 "type": "Microsoft.ServiceBus/Namespaces",
 "location": "[variables('location')]",
 "kind": "Messaging",
 "sku": {
 "name": "Standard",
 },
 "resources": [
 {
 "apiVersion": "[variables('sbVersion')]",
 "name": "[parameters('serviceBusQueueName')]",
 "type": "Queues",
 "dependsOn": [
 "[concat('Microsoft.ServiceBus/namespaces/', parameters('serviceBusNamespaceName'))]"
],
 "properties": {
 "path": "[parameters('serviceBusQueueName')]"
 }
 }
]
 }]
]
```

# Commands to run deployment

To deploy the resources to Azure, you must be logged in to your Azure account and you must use the Azure Resource Manager module. To learn about using Azure Resource Manager with either Azure PowerShell or Azure CLI, see:

- [Using Azure PowerShell with Azure Resource Manager](#)

- [Using the Azure CLI for Mac, Linux, and Windows with Azure Resource Management.](#)

The following examples assume you already have a resource group in your account with the specified name.

## PowerShell

```
New-AzureRmResourceGroupDeployment -ResourceGroupName <resource-group-name> -TemplateFile
<https://raw.githubusercontent.com/azure/azure-quickstart-templates/master/201-servicebus-create-
queue/azuredeploy.json>
```

## Azure CLI

```
azure config mode arm

azure group deployment create <my-resource-group> <my-deployment-name> --template-uri
<https://raw.githubusercontent.com/azure/azure-quickstart-templates/master/201-servicebus-create-
queue/azuredeploy.json>
```

## Next steps

Now that you've created and deployed resources using Azure Resource Manager, learn how to manage these resources by viewing these articles:

- [Manage Service Bus with PowerShell](#)
- [Manage Service Bus resources with the Service Bus Explorer](#)

# Create a Service Bus namespace with topic and subscription using an Azure Resource Manager template

4/12/2018 • 2 min to read • [Edit Online](#)

This article shows how to use an Azure Resource Manager template that creates a Service Bus namespace and a topic and subscription within that namespace. The article explains how to specify which resources are deployed and how to define parameters that are specified when the deployment is executed. You can use this template for your own deployments, or customize it to meet your requirements.

For more information about creating templates, please see [Authoring Azure Resource Manager templates](#).

For the complete template, see the [Service Bus namespace with topic and subscription](#) template.

## NOTE

The following Azure Resource Manager templates are available for download and deployment.

- [Create a Service Bus namespace](#)
- [Create a Service Bus namespace with queue](#)
- [Create a Service Bus namespace with queue and authorization rule](#)
- [Create a Service Bus namespace with topic, subscription, and rule](#)

To check for the latest templates, visit the [Azure Quickstart Templates](#) gallery and search for **Service Bus**.

## What will you deploy?

With this template, you deploy a Service Bus namespace with topic and subscription.

[Service Bus topics and subscriptions](#) provide a one-to-many form of communication, in a *publish/subscribe* pattern.

To run the deployment automatically, click the following button:



## Parameters

With Azure Resource Manager, you define parameters for values you want to specify when the template is deployed. The template includes a section called `Parameters` that contains all of the parameter values. You should define a parameter for those values that vary based on the project you are deploying or based on the environment you are deploying to. Do not define parameters for values that always stay the same. Each parameter value is used in the template to define the resources that are deployed.

The template defines the following parameters.

### **serviceBusNamespaceName**

The name of the Service Bus namespace to create.

```
"serviceBusNamespaceName": {
 "type": "string"
}
```

### **serviceBusTopicName**

The name of the topic created in the Service Bus namespace.

```
"serviceBusTopicName": {
 "type": "string"
}
```

### **serviceBusSubscriptionName**

The name of the subscription created in the Service Bus namespace.

```
"serviceBusSubscriptionName": {
 "type": "string"
}
```

### **serviceBusApiVersion**

The Service Bus API version of the template.

```
"serviceBusApiVersion": {
 "type": "string",
 "defaultValue": "2017-04-01",
 "metadata": {
 "description": "Service Bus ApiVersion used by the template"
 }
}
```

## Resources to deploy

Creates a standard Service Bus namespace of type **Messaging**, with topic and subscription.

```

"resources": [
 {
 "apiVersion": "[variables('sbVersion')]",
 "name": "[parameters('serviceBusNamespaceName')]",
 "type": "Microsoft.ServiceBus/Namespaces",
 "location": "[variables('location')]",
 "kind": "Messaging",
 "sku": {
 "name": "Standard",
 },
 "resources": [
 {
 "apiVersion": "[variables('sbVersion')]",
 "name": "[parameters('serviceBusTopicName')]",
 "type": "Topics",
 "dependsOn": [
 "[concat('Microsoft.ServiceBus/namespaces/', parameters('serviceBusNamespaceName'))]"
],
 "properties": {
 "path": "[parameters('serviceBusTopicName')]"
 },
 "resources": [
 {
 "apiVersion": "[variables('sbVersion')]",
 "name": "[parameters('serviceBusSubscriptionName')]",
 "type": "Subscriptions",
 "dependsOn": [
 "[parameters('serviceBusTopicName')]"
],
 "properties": {}
 }
]
 }
]
 }
]

```

## Commands to run deployment

To deploy the resources to Azure, you must be logged in to your Azure account and you must use the Azure Resource Manager module. To learn about using Azure Resource Manager with either Azure PowerShell or Azure CLI, see:

- [Using Azure PowerShell with Azure Resource Manager](#)
- [Using the Azure CLI for Mac, Linux, and Windows with Azure Resource Management.](#)

The following examples assume you already have a resource group in your account with the specified name.

## PowerShell

```

New-AzureResourceGroupDeployment -Name <deployment-name> -ResourceGroupName <resource-group-name> -
TemplateUri <https://raw.githubusercontent.com/azure/azure-quickstart-templates/master/201-servicebus-create-
topic-and-subscription/azuredploy.json>

```

## Azure CLI

```

azure config mode arm

azure group deployment create <my-resource-group> <my-deployment-name> --template-uri
<https://raw.githubusercontent.com/azure/azure-quickstart-templates/master/201-servicebus-create-topic-and-
subscription/azuredploy.json>

```

## Next steps

Now that you've created and deployed resources using Azure Resource Manager, learn how to manage these resources by viewing these articles:

- [Manage Service Bus with PowerShell](#)
- [Manage Service Bus resources with the Service Bus Explorer](#)

# Create a Service Bus authorization rule for namespace and queue using an Azure Resource Manager template

4/12/2018 • 3 min to read • [Edit Online](#)

This article shows how to use an Azure Resource Manager template that creates an [authorization rule](#) for a Service Bus namespace and queue. The article explains how to specify which resources are deployed and how to define parameters that are specified when the deployment is executed. You can use this template for your own deployments, or customize it to meet your requirements.

For more information about creating templates, please see [Authoring Azure Resource Manager templates](#).

For the complete template, see the [Service Bus authorization rule template](#) on GitHub.

## NOTE

The following Azure Resource Manager templates are available for download and deployment.

- [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 topic, subscription, and rule](#)

To check for the latest templates, visit the [Azure Quickstart Templates](#) gallery and search for **Service Bus**.

## What will you deploy?

With this template, you deploy a Service Bus authorization rule for a namespace and messaging entity (in this case, a queue).

This template uses [Shared Access Signature \(SAS\)](#) for authentication. SAS 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.

To run the deployment automatically, click the following button:



## Parameters

With Azure Resource Manager, you define parameters for values you want to specify when the template is deployed. The template includes a section called `Parameters` that contains all of the parameter values. You should define a parameter for those values that will vary based on the project you are deploying or based on the environment you are deploying to. Do not define parameters for values that will always stay the same. Each parameter value is used in the template to define the resources that are deployed.

The template defines the following parameters.

**serviceBusNamespaceName**

The name of the Service Bus namespace to create.

```
"serviceBusNamespaceName": {
 "type": "string"
}
```

#### **namespaceAuthorizationRuleName**

The name of the authorization rule for the namespace.

```
"namespaceAuthorizationRuleName": {
 "type": "string"
}
```

#### **serviceBusQueueName**

The name of the queue in the Service Bus namespace.

```
"serviceBusQueueName": {
 "type": "string"
}
```

#### **serviceBusApiVersion**

The Service Bus API version of the template.

```
"serviceBusApiVersion": {
 "type": "string",
 "defaultValue": "2017-04-01",
 "metadata": {
 "description": "Service Bus ApiVersion used by the template"
 }
}
```

## Resources to deploy

Creates a standard Service Bus namespace of type **Messaging**, and a Service Bus authorization rule for namespace and entity.

```

"resources": [
 {
 "apiVersion": "[variables('sbVersion')]",
 "name": "[parameters('serviceBusNamespaceName')]",
 "type": "Microsoft.ServiceBus/namespaces",
 "location": "[variables('location')]",
 "kind": "Messaging",
 "sku": {
 "name": "Standard",
 },
 "resources": [
 {
 "apiVersion": "[variables('sbVersion')]",
 "name": "[parameters('serviceBusQueueName')]",
 "type": "Queues",
 "dependsOn": [
 "[concat('Microsoft.ServiceBus/namespaces/', parameters('serviceBusNamespaceName'))]"
],
 "properties": {
 "path": "[parameters('serviceBusQueueName')]"
 },
 "resources": [
 {
 "apiVersion": "[variables('sbVersion')]",
 "name": "[parameters('queueAuthorizationRuleName')]",
 "type": "authorizationRules",
 "dependsOn": [
 "[parameters('serviceBusQueueName')]"
],
 "properties": {
 "Rights": ["Listen"]
 }
 }
]
 }
]
 },
 {
 "apiVersion": "[variables('sbVersion')]",
 "name": "[variables('namespaceAuthRuleName')]",
 "type": "Microsoft.ServiceBus/namespaces/authorizationRules",
 "dependsOn": "[["concat('Microsoft.ServiceBus/namespaces/',
parameters('serviceBusNamespaceName'))"],",
 "location": "[resourceGroup().location]",
 "properties": {
 "Rights": ["Send"]
 }
 }
]

```

## Commands to run deployment

To deploy the resources to Azure, you must be logged in to your Azure account and you must use the Azure Resource Manager module. To learn about using Azure Resource Manager with either Azure PowerShell or Azure CLI, see:

- [Using Azure PowerShell with Azure Resource Manager](#)
- [Using the Azure CLI for Mac, Linux, and Windows with Azure Resource Management.](#)

The following examples assume you already have a resource group in your account with the specified name.

### PowerShell

```
New-AzureRmResourceGroupDeployment -ResourceGroupName <resource-group-name> -TemplateFile
<https://raw.githubusercontent.com/azure/azure-quickstart-templates/master/301-servicebus-create-authrule-
namespace-and-queue/azuredploy.json>
```

## Azure CLI

```
azure config mode arm

azure group deployment create <my-resource-group> <my-deployment-name> --template-uri
<https://raw.githubusercontent.com/azure/azure-quickstart-templates/master/301-servicebus-create-authrule-
namespace-and-queue/azuredploy.json>
```

## Next steps

Now that you've created and deployed resources using Azure Resource Manager, learn how to manage these resources by viewing these articles:

- [Manage Service Bus with PowerShell](#)
- [Manage Service Bus resources with the Service Bus Explorer](#)
- [Service Bus authentication and authorization](#)

# Create a Service Bus namespace with topic, subscription, and rule using an Azure Resource Manager template

4/12/2018 • 3 min to read • [Edit Online](#)

This article shows how to use an Azure Resource Manager template that creates a Service Bus namespace with a topic, subscription, and rule (filter). The article explains how to specify which resources are deployed and how to define parameters that are specified when the deployment is executed. You can use this template for your own deployments, or customize it to meet your requirements.

For more information about creating templates, see [Authoring Azure Resource Manager templates](#).

For more information about practice and patterns on Azure resources naming conventions, see [Recommended naming conventions for Azure resources](#).

For the complete template, see the [Service Bus namespace with topic, subscription, and rule](#) template.

## NOTE

The following Azure Resource Manager templates are available for download and deployment.

- [Create a Service Bus namespace with queue and authorization rule](#)
- [Create a Service Bus namespace with queue](#)
- [Create a Service Bus namespace](#)
- [Create a Service Bus namespace with topic and subscription](#)

To check for the latest templates, visit the [Azure Quickstart Templates](#) gallery and search for Service Bus.

## What will you deploy?

With this template, you deploy a Service Bus namespace with topic, subscription, and rule (filter).

[Service Bus topics and subscriptions](#) provide a one-to-many form of communication, in a *publish/subscribe* pattern. When using topics and subscriptions, components of a distributed application do not communicate directly with each other, instead they exchange messages via topic that acts as an intermediary. A subscription to a topic resembles a virtual queue that receives copies of messages that were sent to the topic. A filter on subscription enables you to specify which messages sent to a topic should appear within a specific topic subscription.

## What are rules (filters)?

In many scenarios, messages that have specific characteristics must be processed in different ways. To enable this custom processing, you can configure subscriptions to find messages that have specific properties and then perform modifications to those properties. Although 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. To learn more about rules (filters), see [Rules and actions](#).

To run the deployment automatically, click the following button:



## Parameters

With Azure Resource Manager, you should define parameters for values you want to specify when the template is deployed. The template includes a section called **Parameters** that contains all the parameter values. You should define a parameter for those values that vary based on the project you are deploying or based on the environment you are deploying to. Do not define parameters for values that always stay the same. Each parameter value is used in the template to define the resources that are deployed.

The template defines the following parameters:

### **serviceBusNamespaceName**

The name of the Service Bus namespace to create.

```
"serviceBusNamespaceName": {
 "type": "string"
}
```

### **serviceBusTopicName**

The name of the topic created in the Service Bus namespace.

```
"serviceBusTopicName": {
 "type": "string"
}
```

### **serviceBusSubscriptionName**

The name of the subscription created in the Service Bus namespace.

```
"serviceBusSubscriptionName": {
 "type": "string"
}
```

### **serviceBusRuleName**

The name of the rule(filter) created in the Service Bus namespace.

```
"serviceBusRuleName": {
 "type": "string",
}
```

### **serviceBusApiVersion**

The Service Bus API version of the template.

```
"serviceBusApiVersion": {
 "type": "string",
 "defaultValue": "2017-04-01",
 "metadata": {
 "description": "Service Bus ApiVersion used by the template"
 }
}
```

## Resources to deploy

Creates a standard Service Bus namespace of type **Messaging**, with topic and subscription and rules.

```

"resources": [
 {
 "apiVersion": "[variables('sbVersion')]",
 "name": "[parameters('serviceBusNamespaceName')]",
 "type": "Microsoft.ServiceBus/Namespaces",
 "location": "[variables('location')]",
 "sku": {
 "name": "Standard",
 },
 "resources": [
 {
 "apiVersion": "[variables('sbVersion')]",
 "name": "[parameters('serviceBusTopicName')]",
 "type": "Topics",
 "dependsOn": [
 "[concat('Microsoft.ServiceBus/namespaces/', parameters('serviceBusNamespaceName'))]"
],
 "properties": {
 "path": "[parameters('serviceBusTopicName')]"
 },
 "resources": [
 {
 "apiVersion": "[variables('sbVersion')]",
 "name": "[parameters('serviceBusSubscriptionName')]",
 "type": "Subscriptions",
 "dependsOn": [
 "[parameters('serviceBusTopicName')]"
],
 "properties": {},
 "resources": [
 {
 "apiVersion": "[variables('sbVersion')]",
 "name": "[parameters('serviceBusRuleName')]",
 "type": "Rules",
 "dependsOn": [
 "[parameters('serviceBusSubscriptionName')]"
],
 "properties": {
 "filter": {
 "sqlExpression": "StoreName = 'Store1'"
 },
 "action": {
 "sqlExpression": "set FilterTag = 'true'"
 }
 }
 }
]
 }
]
 }
]
 }
]

```

## Commands to run deployment

To deploy the resources to Azure, you must be logged in to your Azure account and you must use the Azure Resource Manager module. To learn about using Azure Resource Manager with either Azure PowerShell or Azure CLI, see:

- [Using Azure PowerShell with Azure Resource Manager](#)
- [Using the Azure CLI for Mac, Linux, and Windows with Azure Resource Management.](#)

The following examples assume you already have a resource group in your account with the specified name.

## PowerShell

```
New-AzureResourceGroupDeployment -Name <deployment-name> -ResourceGroupName <resource-group-name> -
TemplateUri <https://raw.githubusercontent.com/azure/azure-quickstart-templates/master/201-servicebus-create-
topic-subscription-rule/azuredeploy.json>
```

## Azure CLI

```
azure config mode arm

azure group deployment create <my-resource-group> <my-deployment-name> --template-uri
<https://raw.githubusercontent.com/azure/azure-quickstart-templates/master/201-servicebus-create-topic-
subscription-rule/azuredeploy.json>
```

## Next steps

Now that you've created and deployed resources using Azure Resource Manager, learn how to manage these resources by viewing these articles:

- [Manage Azure Service Bus](#)
- [Manage Service Bus with PowerShell](#)
- [Manage Service Bus resources with the Service Bus Explorer](#)

# Use PowerShell to manage Service Bus resources

12/21/2017 • 4 min to read • [Edit Online](#)

Microsoft Azure PowerShell is a scripting environment that you can use to control and automate the deployment and management of Azure services. This article describes how to use the [Service Bus Resource Manager PowerShell module](#) to provision and manage Service Bus entities (namespaces, queues, topics, and subscriptions) using a local Azure PowerShell console or script.

You can also manage Service Bus entities using Azure Resource Manager templates. For more information, see the article [Create Service Bus resources using Azure Resource Manager templates](#).

## Prerequisites

Before you begin, you'll need the following prerequisites:

- An Azure subscription. For more information about obtaining a subscription, see [purchase options, member offers](#), or [free account](#).
- A computer with Azure PowerShell. For instructions, see [Get started with Azure PowerShell cmdlets](#).
- A general understanding of PowerShell scripts, NuGet packages, and the .NET Framework.

## Get started

The first step is to use PowerShell to log in to your Azure account and Azure subscription. Follow the instructions in [Get started with Azure PowerShell cmdlets](#) to log in to your Azure account, and retrieve and access the resources in your Azure subscription.

## Provision a Service Bus namespace

When working with Service Bus namespaces, you can use the [Get-AzureRmServiceBusNamespace](#), [New-AzureRmServiceBusNamespace](#), [Remove-AzureRmServiceBusNamespace](#), and [Set-AzureRmServiceBusNamespace](#) cmdlets.

This example creates a few local variables in the script; `$Namespace` and `$Location`.

- `$Namespace` is the name of the Service Bus namespace with which we want to work.
- `$Location` identifies the data center in which we provision the namespace.
- `$CurrentNamespace` stores the reference namespace that we retrieve (or create).

In an actual script, `$Namespace` and `$Location` can be passed as parameters.

This part of the script does the following:

1. Attempts to retrieve a Service Bus namespace with the specified name.
2. If the namespace is found, it reports what was found.
3. If the namespace is not found, it creates the namespace and then retrieves the newly created namespace.

```

Query to see if the namespace currently exists
$CurrentNamespace = Get-AzureRMServiceBusNamespace -ResourceGroup $ResGrpName -NamespaceName
$Namespace

Check if the namespace already exists or needs to be created
if ($CurrentNamespace)
{
 Write-Host "The namespace $Namespace already exists in the $Location region."
 # Report what was found
 Get-AzureRMServiceBusNamespace -ResourceGroup $ResGrpName -NamespaceName $Namespace
}
else
{
 Write-Host "The $Namespace namespace does not exist."
 Write-Host "Creating the $Namespace namespace in the $Location region..."
 New-AzureRmServiceBusNamespace -ResourceGroup $ResGrpName -NamespaceName $Namespace -Location
$Location
 $CurrentNamespace = Get-AzureRMServiceBusNamespace -ResourceGroup $ResGrpName -NamespaceName
$Namespace
 Write-Host "The $Namespace namespace in Resource Group $ResGrpName in the $Location region has
been successfully created."
}

}

```

### Create a namespace authorization rule

The following example shows how to manage namespace authorization rules using the [New-AzureRmServiceBusNamespaceAuthorizationRule](#), [Get-AzureRmServiceBusNamespaceAuthorizationRule](#), [Set-AzureRmServiceBusNamespaceAuthorizationRule](#), and [Remove-AzureRmServiceBusNamespaceAuthorizationRule](#) cmdlets.

```

Query to see if rule exists
$CurrentRule = Get-AzureRmServiceBusNamespaceAuthorizationRule -ResourceGroup $ResGrpName -NamespaceName
$Namespace -AuthorizationRuleName $AuthRule

Check if the rule already exists or needs to be created
if ($CurrentRule)
{
 Write-Host "The $AuthRule rule already exists for the namespace $Namespace."
}
else
{
 Write-Host "The $AuthRule rule does not exist."
 Write-Host "Creating the $AuthRule rule for the $Namespace namespace..."
 New-AzureRmServiceBusNamespaceAuthorizationRule -ResourceGroup $ResGrpName -NamespaceName $Namespace -
 AuthorizationRuleName $AuthRule -Rights @("Listen","Send")
 $CurrentRule = Get-AzureRmServiceBusNamespaceAuthorizationRule -ResourceGroup $ResGrpName -NamespaceName
$Namespace -AuthorizationRuleName $AuthRule
 Write-Host "The $AuthRule rule for the $Namespace namespace has been successfully created."

 Write-Host "Setting rights on the namespace"
 $authRuleObj = Get-AzureRmServiceBusNamespaceAuthorizationRule -ResourceGroup $ResGrpName -NamespaceName
$Namespace -AuthorizationRuleName $AuthRule

 Write-Host "Remove Send rights"
 $authRuleObj.Rights.Remove("Send")
 Set-AzureRmServiceBusNamespaceAuthorizationRule -ResourceGroup $ResGrpName -NamespaceName $Namespace -
AuthRuleObj $authRuleObj

 Write-Host "Add Send and Manage rights to the namespace"
 $authRuleObj.Rights.Add("Send")
 Set-AzureRmServiceBusNamespaceAuthorizationRule -ResourceGroup $ResGrpName -NamespaceName $Namespace -
AuthRuleObj $authRuleObj
 $authRuleObj.Rights.Add("Manage")
 Set-AzureRmServiceBusNamespaceAuthorizationRule -ResourceGroup $ResGrpName -NamespaceName $Namespace -
AuthRuleObj $authRuleObj

 Write-Host "Show value of primary key"
 $CurrentKey = Get-AzureRmServiceBusNamespaceKey -ResourceGroup $ResGrpName -NamespaceName $Namespace -
AuthorizationRuleName $AuthRule

 Write-Host "Remove this authorization rule"
 Remove-AzureRmServiceBusNamespaceAuthorizationRule -ResourceGroup $ResGrpName -NamespaceName $Namespace -
AuthorizationRuleName $AuthRule
}

```

## Create a queue

To create a queue or topic, perform a namespace check using the script in the previous section. Then, create the queue:

```

Check if queue already exists
$CurrentQ = Get-AzureRmServiceBusQueue -ResourceGroup $ResGrpName -NamespaceName $Namespace -QueueName $QueueName

if($CurrentQ)
{
 Write-Host "The queue $QueueName already exists in the $Location region."
}
else
{
 Write-Host "The $QueueName queue does not exist."
 Write-Host "Creating the $QueueName queue in the $Location region..."
 New-AzureRmServiceBusQueue -ResourceGroup $ResGrpName -NamespaceName $Namespace -QueueName $QueueName -EnablePartitioning $True
 $CurrentQ = Get-AzureRmServiceBusQueue -ResourceGroup $ResGrpName -NamespaceName $Namespace -QueueName $QueueName
 Write-Host "The $QueueName queue in Resource Group $ResGrpName in the $Location region has been successfully created."
}

```

## Modify queue properties

After executing the script in the preceding section, you can use the [Set-AzureRmServiceBusQueue](#) cmdlet to update the properties of a queue, as in the following example:

```

$CurrentQ.DeadLetteringOnMessageExpiration = $True
$CurrentQ.MaxDeliveryCount = 7
$CurrentQ.MaxValueInMegabytes = 2048
$CurrentQ.EnableExpress = $True

Set-AzureRmServiceBusQueue -ResourceGroup $ResGrpName -NamespaceName $Namespace -QueueName $QueueName -QueueObj $CurrentQ

```

## Provisioning other Service Bus entities

You can use the [Service Bus PowerShell module](#) to provision other entities, such as topics and subscriptions. These cmdlets are syntactically similar to the queue creation cmdlets demonstrated in the previous section.

## Next steps

- See the complete Service Bus Resource Manager PowerShell module documentation [here](#). This page lists all available cmdlets.
- For information about using Azure Resource Manager templates, see the article [Create Service Bus resources using Azure Resource Manager templates](#).
- Information about [Service Bus .NET management libraries](#).

There are some alternate ways to manage Service Bus entities, as described in these blog posts:

- [How to create Service Bus queues, topics and subscriptions using a PowerShell script](#)
- [How to create a Service Bus Namespace and an Event Hub using a PowerShell script](#)
- [Service Bus PowerShell Scripts](#)

# Service Bus messaging exceptions

1/31/2018 • 7 min to read • [Edit Online](#)

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:

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
<a href="#">TimeoutException</a>	The server did not respond to the requested operation within the specified time which is controlled by <a href="#">OperationTimeout</a> . 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 <a href="#">Timeout exceptions</a> .	Retry might help in some cases; add retry logic to code.

Exception Type	Description/Cause/Examples	Suggested Action	Note on Automatic/Immediate Retry
<a href="#">InvalidOperationException</a>	The requested user operation is not allowed within the server or service. See the exception message for details. For example, <a href="#">Complete()</a> generates this exception if the message was received in <a href="#">ReceiveAndDelete</a> mode.	Check the code and the documentation. Make sure the requested operation is valid.	Retry will not help.
<a href="#">OperationCanceledException</a>	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.
<a href="#">UnauthorizedAccessException</a>	The <a href="#">TokenProvider</a> 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.
<a href="#">ArgumentException</a> <a href="#">ArgumentNullException</a> <a href="#">ArgumentOutOfRangeException</a>	<p>One or more arguments supplied to the method are invalid.</p> <p>The URI supplied to <a href="#">NamespaceManager</a> or <a href="#">Create</a> contains path segment(s).</p> <p>The URI scheme supplied to <a href="#">NamespaceManager</a> or <a href="#">Create</a> is invalid.</p> <p>The property value is larger than 32KB.</p>	Check the calling code and make sure the arguments are correct.	Retry will not help.
<a href="#">MessagingEntityNotFoundException</a>	Entity associated with the operation does not exist or it has been deleted.	Make sure the entity exists.	Retry will not help.
<a href="#">MessageNotFoundException</a>	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.
<a href="#">MessagingCommunicationException</a>	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.
<a href="#">ServerBusyException</a>	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
<a href="#">MessageLockLostException</a>	Lock token associated with the message has expired, or the lock token is not found.	Dispose the message.	Retry will not help.
<a href="#">SessionLockLostException</a>	Lock associated with this session is lost.	Abort the <a href="#">MessageSession</a> object.	Retry will not help.
<a href="#">MessagingException</a>	<p>Generic messaging exception that may be thrown in the following cases:</p> <p>An attempt is made to create a <a href="#">QueueClient</a> using a name or path that belongs to a different entity type (for example, a topic).</p> <p>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.</p>	<p>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 <a href="#">IsTransient</a> property. If it is <b>true</b>, you can retry the operation.</p>	Retry behavior is undefined and might not help.
<a href="#">MessagingEntityAlreadyExistsException</a>	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.
<a href="#">QuotaExceededException</a>	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 <a href="#">QuotaExceededException</a> .	Retry might help if messages have been removed in the meantime.
<a href="#">RuleActionException</a>	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 ( <i>Transaction.Current</i> ) 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 contains 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:xxxxxxxxxxxxxxxxxxxxxx, 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. You can use the [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 are being written.

## 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.

## Next steps

For the complete Service Bus .NET API reference, see the [Azure .NET API reference](#).

To learn more about [Service Bus](#), see the following articles:

- [Service Bus messaging overview](#)
- [Service Bus fundamentals](#)

- Service Bus architecture

# Service Bus quotas

2/1/2018 • 3 min to read • [Edit Online](#)

This section lists basic quotas and throttling thresholds in 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	Notes	Value
Maximum number of basic / standard namespaces per Azure subscription	Namespace	Subsequent requests for additional basic / standard namespaces are rejected by the portal.	100
Maximum number of premium namespaces per Azure subscription	Namespace	Subsequent requests for additional premium namespaces are rejected by the portal.	10
Queue/topic size	Entity	Defined upon creation of the queue/topic.  Subsequent incoming messages are rejected and an exception is received by the calling code.	1, 2, 3, 4 or 5 GB.  If <a href="#">partitioning</a> is enabled, the maximum queue/topic size is 80 GB.
Number of concurrent connections on a namespace	Namespace	Subsequent requests for additional connections are rejected and an exception is received by the calling code. REST operations do not count towards concurrent TCP connections.	NetMessaging: 1,000  AMQP: 5,000
Number of concurrent receive requests on a queue/topic/subscription entity	Entity	Subsequent receive requests are rejected and an exception is received by the calling code. This quota applies to the combined number of concurrent receive operations across all subscriptions on a topic.	5,000

Quota Name	Scope	Notes	Value
Number of topics/queues per service namespace	Namespace	Subsequent requests for creation of a new topic or queue on the service namespace are rejected. As a result, if configured through the <a href="#">Azure portal</a> , an error message is generated. If called from the management API, an exception is 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	Namespace	Subsequent requests for creation of a new partitioned topic or queue on the service namespace are rejected. As a result, if configured through the <a href="#">Azure portal</a> , an error message is generated. If called from the management API, a <b>QuotaExceededException</b> exception is received by the calling code.	Basic and Standard Tiers - 100  <b>Premium</b> - 1,000 (per messaging unit)  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	-	260 characters
Maximum size of any messaging entity name: namespace, subscription, or subscription rule	Entity	-	50 characters
Message size for a queue/topic/subscription entity	Entity	Incoming messages that exceed these quotas are rejected and an exception is received by the calling code.	Maximum message size: 256 KB ( <a href="#">Standard tier</a> ) / 1 MB ( <a href="#">Premium tier</a> ).  <b>Note</b> Due to system overhead, this limit is usually slightly less.  Maximum header size: 64 KB  Maximum number of header properties in property bag: <b>byte/int.MaxValue</b>  Maximum size of property in property bag: No explicit limit. Limited by maximum header size.

Quota Name	Scope	Notes	Value
Message property size for a queue/topic/subscription entity	Entity	A <b>SerializationException</b> exception is generated.	Maximum message property size for each property is 32 K. Cumulative size of all properties cannot exceed 64 K. This applies to the entire header of the <a href="#">BrokeredMessage</a> , which has both user properties as well as system properties (such as <a href="#">SequenceNumber</a> , <a href="#">Label</a> , <a href="#">MessageId</a> , and so on).
Number of subscriptions per topic	Entity	Subsequent requests for creating additional subscriptions for the topic are rejected. As a result, if configured through the portal, an error message is shown. If called from the management API an exception is received by the calling code.	2,000
Number of SQL filters per topic	Entity	Subsequent requests for creation of additional filters on the topic are rejected and an exception is received by the calling code.	2,000
Number of correlation filters per topic	Entity	Subsequent requests for creation of additional filters on the topic are rejected and an exception is received by the calling code.	100,000
Size of SQL filters/actions	Namespace	Subsequent requests for creation of additional filters are rejected and an exception is 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 <a href="#">SharedAccessAuthorizationRule</a> rules per namespace, queue, or topic	Entity, namespace	Subsequent requests for creation of additional rules are rejected and an exception is 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.

QUOTA NAME	SCOPE	NOTES	VALUE
Number of messages per transaction	Transaction	Additional incoming messages are rejected and an exception stating "Cannot send more than 100 messages in a single transaction" is received by the calling code.	100  For both <b>Send()</b> and <b>SendAsync()</b> operations.

# SQLFilter syntax

2/6/2018 • 5 min to read • [Edit Online](#)

A `SqlFilter` object 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 grammar 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.

These 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 throws 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 \(.NET Framework\)](#)
- [SQLFilter class \(.NET Standard\)](#)
- [SQLRuleAction class](#)

# SQLRuleAction syntax

2/6/2018 • 4 min to read • [Edit Online](#)

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 article 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

12/21/2017 • 5 min to read • [Edit Online](#)

Azure Service Bus is offered in 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 2 meters for queues and topics/subscriptions:

1. **Messaging Operations:** Defined as API calls against queue or topic/subscription service endpoints. This meter replaces 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 only applies 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.

Note that the Standard tier base charge is charged only once per month per Azure subscription. This means that after you create a single Standard tier Service Bus namespace, you can create as many additional Standard namespaces as you want under that same Azure subscription, without incurring additional base charges.

The [Service Bus pricing](#) table summarizes the functional differences between the Standard and Premium tiers.

## Messaging operations

Queues and topics/subscriptions are billed per "operation," not per message. 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	Send and receive messages with queues or topics/subscriptions.

OPERATION TYPE	DESCRIPTION
Session state	Get or set session state on a queue or topic/subscription.

For cost details, see the prices listed on the [Service Bus pricing](#) page.

## Brokered connections

*Brokered connections* accommodate 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.

For connection quotas and other service limits, see the [Service Bus quotas](#) article. For more information about brokered connections, see the [FAQ](#) section later in this article.

The Standard tier removes the per-namespace brokered connection limit and counts aggregate brokered connection usage across the Azure subscription. For more information, see the [Brokered connections](#) table.

### 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.

## FAQ

### 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. 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 complete details about Service Bus pricing, see the [Service Bus pricing page](#).
- See the [Service Bus FAQ](#) for some common FAQs about Service bus pricing and billing.

# Service Bus messaging samples

12/21/2017 • 1 min to read • [Edit Online](#)

The Service Bus messaging samples demonstrate key features in [Service Bus messaging](#). Currently, you can find the samples in two places:

- [Service Bus messaging samples on GitHub](#): a newer set of samples, hosted on GitHub. See the [readme file](#) in the repo for descriptions of these .NET samples. The samples are continuously updated, so check back often for updates.
- [MSDN samples page](#): older samples that live in the MSDN code gallery. Although these samples still work, they are not maintained and may be outdated with respect to current recommended programming practices.

## Service Bus Explorer

In addition, the [Service Bus Explorer](#) is a sample hosted on GitHub that enables you to connect to a Service Bus service namespace and easily manage messaging entities. The tool provides advanced features such as import/export functionality, and the ability to test messaging entities and relay services. You can find the full Service Bus Explorer source and documentation on [GitHub](#).

## Next steps

Sample locations are here:

- [GitHub samples](#)
- [Service Bus Explorer](#)

See the following topics for conceptual overviews of Service Bus.

- [Service Bus messaging overview](#)
- [Service Bus architecture](#)
- [Service Bus fundamentals](#)