

# Contents

## [Service Bus Messaging Documentation](#)

### [Overview](#)

[What is Service Bus Messaging?](#)

[Compare messaging services](#)

### [Quickstarts](#)

[Service Bus queues](#)

[Create a Service Bus queue](#)

[Azure portal](#)

[Azure PowerShell](#)

[Azure CLI](#)

[Azure Resource Manager template](#)

[Send and receive messages](#)

[.NET](#)

[Java](#)

[Node.js](#)

[azure/service-bus package](#)

[azure-sb package](#)

[PHP](#)

[Python](#)

[Ruby](#)

[Services Bus topics and subscriptions](#)

[Create topics and subscriptions](#)

[Azure portal](#)

[Azure Resource Manager template](#)

[Publish and subscribe for messages](#)

[.NET](#)

[Java](#)

[Node.js](#)

[azure/service-bus package](#)

[azure-sb package](#)

[PHP](#)

[Python](#)

[Ruby](#)

## Tutorials

[Update inventory - PowerShell](#)

[Update inventory - Portal](#)

[Update inventory - CLI](#)

## Samples

[Service Bus samples](#)

## Concepts

[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](#)

[Throttling](#)

## Security

[Authentication and authorization](#)

[Use Shared Access Signatures](#)

[Migrate from ACS to SAS](#)

[Authentication with Shared Access Signatures](#)

[Use Azure Active Directory](#)

[Authenticate with managed identities for Azure resources](#)

[Authenticate from an application](#)

[Built-in security controls](#)

## How-to guides

[Create a namespace](#)

[Develop](#)

[Message handling](#)

[Queues, topics, and subscriptions](#)

- [Messages, payloads, and serialization](#)
- [Message transfers, locks, and settlement](#)
- [Message sequencing and timestamps](#)
- [Message expiration \(Time to Live\)](#)
- [Message handling using Azure Event Grid](#)
- [Azure Event Grid examples](#)
- [Topic filters and actions](#)
- [Migrate from Standard to Premium namespaces](#)
- [Partitioned queues and topics](#)
- [Message sessions](#)
- [AMQP](#)
  - [AMQP overview](#)
  - [.NET](#)
  - [Java Message Service \(JMS\) and AMQP](#)
  - [AMQP protocol guide](#)
  - [AMQP request-response-based operations](#)
  - [AMQP errors](#)
- [Advanced features](#)
  - [Use firewalls](#)
  - [Virtual Network service endpoints](#)
  - [Dead-letter queues](#)
  - [Prefetch messages](#)
  - [Duplicate message detection](#)
  - [Message counters](#)
  - [Message deferral](#)
  - [Message browsing](#)
  - [Chain entities with auto-forwarding](#)
  - [Transaction processing](#)
- [End-to-end tracing and diagnostics](#)
- [Build a multi-tier Service Bus application](#)
- [Manage](#)
  - [Use Azure PowerShell to provision entities](#)

- [Monitor Service Bus with Azure Monitor](#)
- [Service Bus management libraries](#)
- [Diagnostic logs](#)
- [Suspend and reactivate messaging entities](#)
- [Use Azure Resource Manager templates](#)
  - [Create a namespace](#)
  - [Create an authorization rule for namespace and queue](#)
  - [Create a namespace with topic, subscription, and rule](#)
- [Secure](#)
  - [Configure customer-managed keys for encryption at rest](#)
- [Troubleshoot](#)
  - [Troubleshooting guide](#)
  - [Resource Manager exceptions](#)
- [Reference](#)
  - [.NET](#)
    - [Microsoft.ServiceBus.Messaging \(.NET Framework\)](#)
    - [Microsoft.Azure.ServiceBus \(.NET Standard\)](#)
  - [Java](#)
  - [Python](#)
  - [Node.js](#)
  - [PHP](#)
  - [Ruby](#)
  - [Go](#)
  - [Azure PowerShell](#)
  - [REST](#)
    - [Resource Manager template](#)
    - [Quotas](#)
    - [SQLFilter syntax](#)
    - [SQLRuleAction syntax](#)
  - [Resources](#)
  - [FAQ](#)
  - [Build your skills with Microsoft Learn](#)

[Azure Roadmap](#)

[Blog](#)

[MSDN forums](#)

[Pricing](#)

[Pricing calculator](#)

[Serverless360](#)

[Service Bus Explorer](#)

[Service updates](#)

[Stack Overflow](#)

[Videos](#)

# What is Azure Service Bus?

1/24/2020 • 4 minutes to read • [Edit Online](#)

Microsoft Azure Service Bus is a fully managed enterprise integration message broker. Service Bus can decouple applications and services. Service Bus offers a reliable and secure platform for asynchronous transfer of data and state.

Data is transferred between different applications and services using *messages*. A message is in binary format and can contain JSON, XML, or just text. For more information, see [Integration Services](#).

Some common messaging scenarios are:

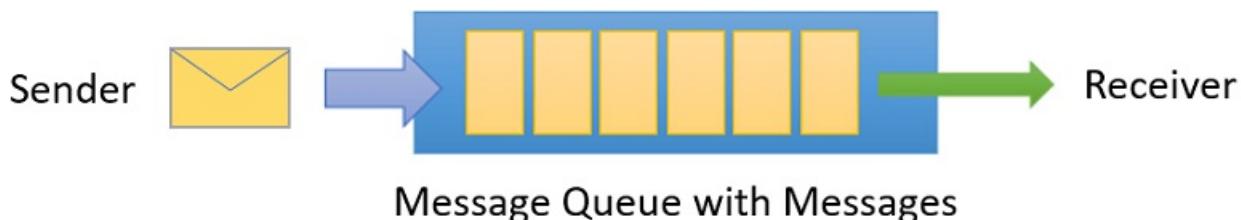
- *Messaging*. Transfer business data, such as sales or purchase orders, journals, or inventory movements.
- *Decouple applications*. Improve reliability and scalability of applications and services. Client and service don't have to be online at the same time.
- *Topics and subscriptions*. Enable 1:n relationships between publishers and subscribers.
- *Message sessions*. Implement workflows that require message ordering or message deferral.

## Namespaces

A namespace is a container for all messaging components. Multiple queues and topics can be in a single namespace, and namespaces often serve as application containers.

## Queues

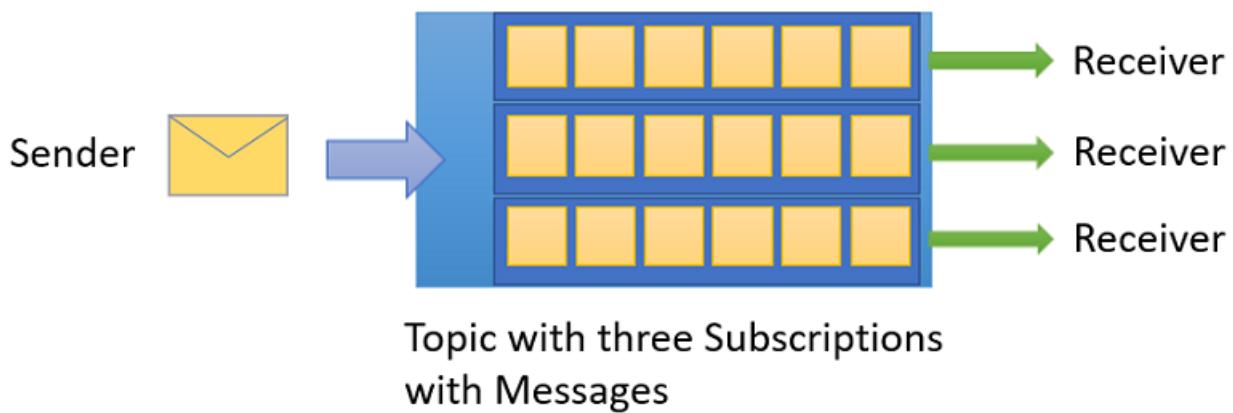
Messages are sent to and received from *queues*. Queues store messages until the receiving application is available to receive and process them.



Messages in queues are ordered and timestamped on arrival. Once accepted, the message is held safely in redundant storage. Messages are delivered in *pull* mode, only delivering messages when requested.

## Topics

You can also use *topics* to send and receive messages. While a queue is often used for point-to-point communication, topics are useful in publish/subscribe scenarios.



Topics can have multiple, independent subscriptions. A subscriber to a topic can receive a copy of each message sent to that topic. Subscriptions are named entities. Subscriptions persist, but can expire or autodelete.

You may not want individual subscriptions to receive all messages sent to a topic. If so, you can use *rules* and *filters* to define conditions that trigger optional *actions*. You can filter specified messages and set or modify message properties. For more information, see [Topic filters and actions](#).

## Advanced features

Service Bus includes advanced features that enable you to solve more complex messaging problems. The following sections describe several of these features.

### **Message sessions**

To create a first-in, first-out (FIFO) guarantee in Service Bus, use sessions. Message sessions enable joint and ordered handling of unbounded sequences of related messages. For more information, see [Message sessions: first in, first out \(FIFO\)](#).

### **Autoforwarding**

The autoforwarding feature chains a queue or subscription to another queue or topic. They must be part of the same namespace. With autoforwarding, Service Bus automatically removes messages from a queue or subscription and puts them in a different queue or topic. For more information, see [Chaining Service Bus entities with autoforwarding](#).

### **Dead-letter queue**

Service Bus supports a dead-letter queue (DLQ). A DLQ holds messages that can't be delivered to any receiver. It holds messages that can't be processed. Service Bus lets you remove messages from the DLQ and inspect them. For more information, see [Overview of Service Bus dead-letter queues](#).

### **Scheduled delivery**

You can submit messages to a queue or topic for delayed processing. You can schedule a job to become available for processing by a system at a certain time. For more information, see [Scheduled messages](#).

### **Message deferral**

A queue or subscription client can defer retrieval of a message until a later time. This deferral might be because of special circumstances in the application. The message remains in the queue or subscription, but it's set aside. For more information, see [Message deferral](#).

### **Batching**

Client-side batching enables a queue or topic client to delay sending 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. For more information, see [Client-side batching](#).

### **Transactions**

A transaction groups two or more operations together into an *execution scope*. Service Bus supports grouping operations against a single messaging entity within the scope of a single transaction. A message entity can be a queue, topic, or subscription. For more information, see [Overview of Service Bus transaction processing](#).

## Filtering and actions

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. For each matching rule condition, the subscription produces a copy of the message, which can be differently annotated for each matching rule. For more information, see [Topic filters and actions](#).

## Autodelete on idle

Autodelete on idle enables you to specify an idle interval after which a queue is automatically deleted. The minimum duration is 5 minutes. For more information, see the [QueueDescription.AutoDeleteOnIdle Property](#).

## Duplicate detection

An error could cause the client to have a doubt about the outcome of a send operation. Duplicate detection enables the sender to resend the same message. Another option is for the queue or topic to discard any duplicate copies. For more information, see [Duplicate detection](#).

## Security protocols

Service Bus supports security protocols such as [Shared Access Signatures](#) (SAS), [Role Based Access Control](#) (RBAC) and [Managed identities for Azure resources](#).

## Geo-disaster recovery

When Azure regions or datacenters experience downtime, Geo-disaster recovery enables data processing to continue operating in a different region or datacenter. For more information, see [Azure Service Bus Geo-disaster recovery](#).

## Security

Service Bus supports standard [AMQP 1.0](#) and [HTTP/REST](#) protocols.

## Client libraries

Service Bus supports client libraries for [.NET](#), [Java](#), and [JMS](#).

## Integration

Service Bus fully integrates with the following Azure services:

- [Event Grid](#)
- [Logic Apps](#)
- [Azure Functions](#)
- [Dynamics 365](#)
- [Azure Stream Analytics](#)

## Next steps

To get started using Service Bus messaging, see the following articles:

- To compare Azure messaging services, see [Comparison of services](#).
- Try the quickstarts for [.NET](#), [Java](#), or [JMS](#).
- To manage Service Bus resources, see [Service Bus Explorer](#).
- To learn more about Standard and Premium tiers and their pricing, see [Service Bus pricing](#).
- To learn about performance and latency for the Premium tier, see [Premium Messaging](#).



# Choose between Azure messaging services - Event Grid, Event Hubs, and Service Bus

1/16/2020 • 4 minutes to read • [Edit Online](#)

Azure offers three services that assist with delivering event messages throughout a solution. These services are:

- [Event Grid](#)
- [Event Hubs](#)
- [Service Bus](#)

Although they have some similarities, each service is designed for particular scenarios. This article describes the differences between these services, and helps you understand which one to choose for your application. In many cases, the messaging services are complementary and can be used together.

## Event vs. message services

There's an important distinction to note between services that deliver an event and services that deliver a message.

### Event

An event is a lightweight notification of a condition or a state change. The publisher of the event has no expectation about how the event is handled. The consumer of the event decides what to do with the notification. Events can be discrete units or part of a series.

Discrete events report state change and are actionable. To take the next step, the consumer only needs to know that something happened. The event data has information about what happened but doesn't have the data that triggered the event. For example, an event notifies consumers that a file was created. It may have general information about the file, but it doesn't have the file itself. Discrete events are ideal for [serverless](#) solutions that need to scale.

Series events report a condition and are analyzable. The events are time-ordered and interrelated. The consumer needs the sequenced series of events to analyze what happened.

### Message

A message is raw data produced by a service to be consumed or stored elsewhere. The message contains the data that triggered the message pipeline. The publisher of the message has an expectation about how the consumer handles the message. A contract exists between the two sides. For example, the publisher sends a message with the raw data, and expects the consumer to create a file from that data and send a response when the work is done.

## Comparison of services

SERVICE	PURPOSE	TYPE	WHEN TO USE
Event Grid	Reactive programming	Event distribution (discrete)	React to status changes
Event Hubs	Big data pipeline	Event streaming (series)	Telemetry and distributed data streaming
Service Bus	High-value enterprise messaging	Message	Order processing and financial transactions

## **Event Grid**

Event Grid is an eventing backplane that enables event-driven, reactive programming. It uses a publish-subscribe model. Publishers emit events, but have no expectation about which events are handled. Subscribers decide which events they want to handle.

Event Grid is deeply integrated with Azure services and can be integrated with third-party services. It simplifies event consumption and lowers costs by eliminating the need for constant polling. Event Grid efficiently and reliably routes events from Azure and non-Azure resources. It distributes the events to registered subscriber endpoints.

The event message has the information you need to react to changes in services and applications. Event Grid isn't a data pipeline, and doesn't deliver the actual object that was updated.

Event Grid supports dead-lettering for events that aren't delivered to an endpoint.

It has the following characteristics:

- dynamically scalable
- low cost
- serverless
- at least once delivery

## **Event Hubs**

Azure Event Hubs is a big data pipeline. It facilitates the capture, retention, and replay of telemetry and event stream data. The data can come from many concurrent sources. Event Hubs allows telemetry and event data to be made available to a variety of stream-processing infrastructures and analytics services. It is available either as data streams or bundled event batches. This service provides a single solution that enables rapid data retrieval for real-time processing as well as repeated replay of stored raw data. It can capture the streaming data into a file for processing and analysis.

It has the following characteristics:

- low latency
- capable of receiving and processing millions of events per second
- at least once delivery

## **Service Bus**

Service Bus is intended for traditional enterprise applications. These enterprise applications require transactions, ordering, duplicate detection, and instantaneous consistency. Service Bus enables [cloud-native](#) applications to provide reliable state transition management for business processes. When handling high-value messages that cannot be lost or duplicated, use Azure Service Bus. Service Bus also facilitates highly secure communication across hybrid cloud solutions and can connect existing on-premises systems to cloud solutions.

Service Bus is a brokered messaging system. It stores messages in a "broker" (for example, a queue) until the consuming party is ready to receive the messages.

It has the following characteristics:

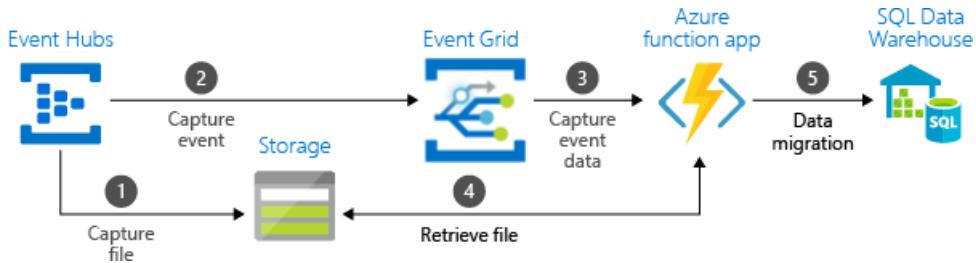
- reliable asynchronous message delivery (enterprise messaging as a service) that requires polling
- advanced messaging features like FIFO, batching/sessions, transactions, dead-lettering, temporal control, routing and filtering, and duplicate detection
- at least once delivery
- optional in-order delivery

# **Use the services together**

In some cases, you use the services side by side to fulfill distinct roles. For example, an e-commerce site can use

Service Bus to process the order, Event Hubs to capture site telemetry, and Event Grid to respond to events like an item was shipped.

In other cases, you link them together to form an event and data pipeline. You use Event Grid to respond to events in the other services. For an example of using Event Grid with Event Hubs to migrate data to a data warehouse, see [Stream big data into a data warehouse](#). The following image shows the workflow for streaming the data.



## Next steps

See the following articles:

- [Asynchronous messaging options in Azure](#)
- [Events, Data Points, and Messages - Choosing the right Azure messaging service for your data.](#)
- [Storage queues and Service Bus queues - compared and contrasted](#)
- To get started with Event Grid, see [Create and route custom events with Azure Event Grid](#).
- To get started with Event Hubs, see [Create an Event Hubs namespace and an event hub using the Azure portal](#).
- To get started with Service Bus, see [Create a Service Bus namespace using the Azure portal](#).

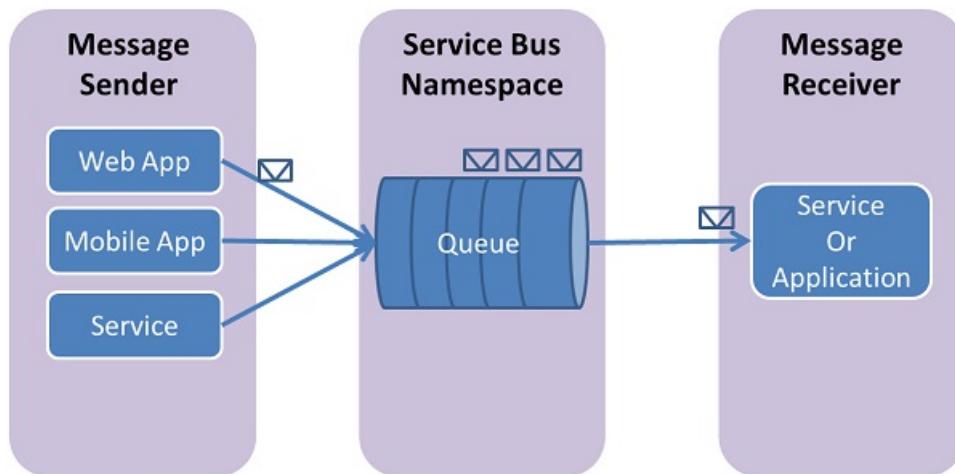
# Quickstart: Use Azure portal to create a Service Bus queue

1/17/2020 • 8 minutes to read • [Edit Online](#)

This quickstart describes how to send and receive messages to and from a Service Bus queue, using the [Azure portal](#) to create a messaging namespace and a queue within that namespace, and to obtain the authorization credentials on that namespace. The procedure then shows how to send and receive messages from this queue using the [.NET Standard library](#).

## 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.

## Prerequisites

To complete this tutorial, make sure you have installed:

- An Azure subscription. If you don't have an Azure subscription, you can create a [free account](#) before you begin.
- [Visual Studio 2017 Update 3 \(version 15.3, 26730.01\)](#) or later. You use Visual Studio to build a sample that sends messages to and receives message from a queue. The sample is to test the queue you created using

PowerShell.

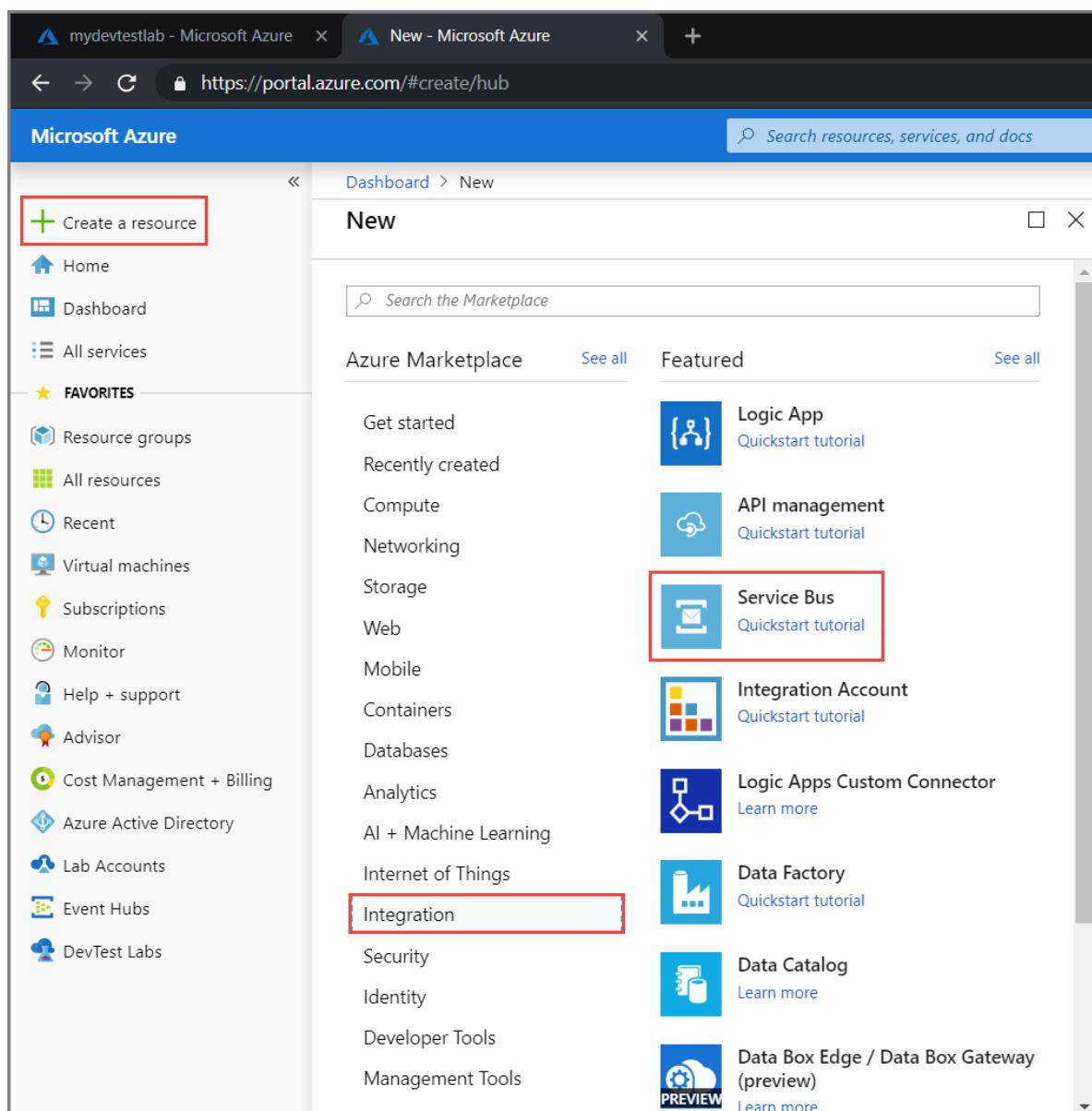
- [NET Core SDK](#), version 2.0 or later.

## 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. Sign in to the [Azure portal](#)
2. In the left navigation pane of the portal, select **+ Create a resource**, select **Integration**, and then select **Service Bus**.



3. In the **Create namespace** dialog, do the following steps:

- a. Enter a **name for the namespace**. The system immediately checks to see if the name is available. For a list of rules for naming namespaces, see [Create Namespace REST API](#).
- b. Select the pricing tier (Basic, Standard, or Premium) for the namespace. If you want to use [topics and subscriptions](#), choose either Standard or Premium. Topics/subscriptions are not supported in the Basic pricing tier.

- c. If you selected the **Premium** pricing tier, follow these steps:
- Specify the number of **messaging units**. The premium tier provides resource isolation at the CPU and memory level so that each workload runs in isolation. This resource container is called a messaging unit. A premium namespace has at least one messaging unit. You can select 1, 2, or 4 messaging units for each Service Bus Premium namespace. For more information, see [Service Bus Premium Messaging](#).
  - Specify whether you want to make the namespace **zone redundant**. The zone redundancy provides enhanced availability by spreading replicas across availability zones within one region at no additional cost. For more information, see [Availability zones in Azure](#).
  - For **Subscription**, choose an Azure subscription in which to create the namespace.
  - For **Resource group**, choose an existing resource group in which the namespace will live, or create a new one.
  - For **Location**, choose the region in which your namespace should be hosted.
  - Select **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.

The screenshot shows the 'Create namespace' wizard in the Azure portal. The steps are as follows:

- Name:** mysbusns (with a green checkmark)
- Pricing tier:** Standard
- Subscription:** Visual Studio Ultimate with MSDN
- Resource group:** (New) sbusrg (with a dropdown arrow) or Create new
- Location:** West US

A large blue 'Create' button is located at the bottom of the form.

4. Confirm that the service bus namespace is deployed successfully. To see the notifications, select the **bell icon (Alerts)** on the toolbar. Select the **name of the resource group** in the notification as shown in the image. You see the resource group that contains the service bus namespace.

The screenshot shows the Azure Notifications page. At the top, there are several icons: a back arrow, a refresh, a bell (which is highlighted with a red box), a gear, a question mark, and a smiley face. To the right, it says '@ho...' and 'DEFAULT DIRECTORY' with a user profile icon. Below the header, the title 'Notifications' is displayed. A blue link 'More events in the activity log →' is on the left, and a 'Dismiss all ...' button is on the right. A single notification card is shown: a green checkmark icon followed by the text 'Deployment succeeded'. Below that, it says 'Deployment 'mysbusns' to resource group 'sbusrsg'' was successful. A timestamp 'a few seconds ago' is at the bottom right of the card.

5. On the **Resource group** page for your resource group, select your **service bus namespace**.

The screenshot shows the Azure Resource Group Overview page for 'sbusrsg'. The left sidebar has sections like Overview, Activity log, Access control (IAM), Tags, Events, Settings, Deployments, Policies, Properties, Locks, Automation script, Monitoring, Insights (preview), Alerts, Metrics, and Diagnostic settings. The main area shows 'Subscription (change) : Visual Studio Ultimate with MSDN' and 'Deployments : 1 Succeeded'. It lists one item: 'NAME' 'mysbusns', 'TYPE' 'Service Bus Namespace', and 'LOCATION' 'West US'. The 'mysbusns' row is highlighted with a red box.

6. You see the home page for your service bus namespace.

The screenshot shows the Azure Service Bus Namespace home page for 'mysbusns'. The left sidebar includes Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Shared access policies, Scale, Geo-Recovery, Properties, Locks, Automation script), Entities (Queues, Topics), Monitoring (Alerts, Metrics, Diagnostic settings), Support + troubleshooting, Resource health, and New support request. The main area displays 'Resource group (change)' 'sbusrsg', 'Status' 'Active', 'Location' 'West US', and 'Subscription (change)' 'Visual Studio Ultimate with MSDN'. It shows 'Incoming Requests' (0), 'Successful Requests' (0), 'Server Errors' (0), 'User Errors' (0), and 'Throttled Requests' (0). A chart titled 'Messages' shows data from 1 hour ago to 2:15 PM. Below the chart is a table for Queues and Topics, which is currently empty. A section for 'Search to filter items...' is also present.

## Get the connection string

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 rules with more constrained rights for regular senders and receivers. To copy the primary and secondary keys for your namespace, 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 'Shared access policies' section for the 'mysbusns' Service Bus Namespace. On the left, a sidebar lists 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Settings' (with 'Shared access policies' selected), 'Scale', 'Geo-Recovery', 'Properties', 'Locks', and 'Automation script'. The main area displays a table with one row for 'RootManageSharedAccessKey'. The 'POLICY' column contains 'RootManageSharedAccessKey' and the 'CLAIMS' column contains 'Manage, Send, Listen'. A red box highlights the 'RootManageSharedAccessKey' row.

4. In the **Policy: RootManageSharedAccessKey** window, click the copy button next to **Primary Connection String**, 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. It includes buttons for 'Save', 'Discard', 'Delete', and 'More'. Under 'Manage' settings, 'Send' and 'Listen' are checked. Below these are fields for 'Primary Key' (containing '<Primary key>') and 'Secondary Key' (containing '<Secondary key>'). At the bottom, there are two connection string fields: 'Primary Connection String' (containing 'Endpoint=sb://mysbusns.servicebus.windows.net/S...') and 'Secondary Connection String' (containing 'Endpoint=sb://mysbusns.servicebus.windows.net/S...'). Each connection string field has a blue 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.

# Create a queue in the Azure portal

1. On the **Service Bus Namespace** page, select **Queues** in the left navigational menu.
2. On the **Queues** page, select **+ Queue** on the toolbar.
3. Enter a **name** for the queue, and leave the other values with their defaults.
4. Now, select **Create**.

The screenshot shows the Azure portal interface for creating a queue. On the left, the navigation menu is open, with 'Queues' selected under the 'Entities' section. A red box highlights the 'Queues' link. On the right, a 'Create queue' dialog box is open. It has a 'Name' field containing 'myqueue' with a checkmark. Other settings include 'Max queue size' at 1 GB, 'Message time to live' set to 14 days, 0 hours, 0 minutes, and 0 seconds, and 'Lock duration' set to 0 days, 0 hours, 0 minutes, and 30 seconds. There are several checkboxes for advanced features: 'Enable duplicate detection', 'Enable dead lettering on message expiration', 'Enable sessions', and 'Enable partitioning', with 'Enable partitioning' checked. A red box highlights the 'Create' button at the bottom of the dialog.

## Send and receive messages

### NOTE

The sample used in this section to send and receive messages is a .NET sample. For samples to send/receive messages using other programming languages, see [Service Bus samples](#).

For step-by-step instructions for sending/receiving messages using various programming languages, see the following quick starts:

- [.NET](#)
- [Java](#)
- [Node.js using azure/service-bus package](#)
- [Node.js using azure-sb package](#)
- [PHP](#)
- [Python](#)
- [Ruby](#)

After the namespace and queue are provisioned, and you have the necessary credentials, you are ready to send and receive messages. You can examine the code in [this GitHub sample folder](#).

To run the code, do the following:

1. Clone the [Service Bus GitHub repository](#) by issuing the following command:

```
git clone https://github.com/Azure/azure-service-bus.git
```

2. Navigate to the sample folder

```
azure-service-bus\samples\DotNet\GettingStarted\BasicSendReceiveQuickStart\BasicSendReceiveQuickStart
```

3. Copy the connection string and queue name you obtained in the Obtain the management credentials section.

4. At a command prompt, type the following command:

```
dotnet build
```

5. Navigate to the `bin\Debug\netcoreapp2.0` folder.

6. Type the following command to run the program. Be sure to replace `myConnectionString` with the value you previously obtained, and `myQueueName` with the name of the queue you created:

```
dotnet BasicSendReceiveQuickStart.dll -ConnectionString "myConnectionString" -QueueName "myQueueName"
```

7. Observe 10 messages being sent to the queue, and subsequently received from the queue:

```
Administrator: Command Prompt
QueueName: spqueue
=====
Press any key to exit after receiving all the messages.
=====
Sending message: Message 0
Sending message: Message 1
Sending message: Message 2
Received message: SequenceNumber:18295873486192641 Body:Message 1
Sending message: Message 3
Sending message: Message 4
Received message: SequenceNumber:22799473113563137 Body:Message 2
Sending message: Message 5
Sending message: Message 6
Sending message: Message 7
Sending message: Message 8
Sending message: Message 9
Received message: SequenceNumber:13792273858822145 Body:Message 0
Received message: SequenceNumber:27303072740933633 Body:Message 3
Received message: SequenceNumber:31806672368304129 Body:Message 4
Received message: SequenceNumber:36310271995674625 Body:Message 5
Received message: SequenceNumber:40813871623045121 Body:Message 6
Received message: SequenceNumber:45317471250415617 Body:Message 7
Received message: SequenceNumber:49821070877786113 Body:Message 8
Received message: SequenceNumber:54324670505156609 Body:Message 9

C:\Program Files\Git\azure-service-bus\samples\DotNet\GettingStarted\BasicSendReceiveQuickStart\BasicSendReceiveQuickStart\bin\Debug\netcoreapp2.0>
```

## Clean up resources

You can use the portal to remove the resource group, namespace, and queue.

## Understand the sample code

This section contains more details about what the sample code does.

### Get connection string and queue

The connection string and queue name are passed to the `Main()` method as command-line arguments. `Main()` declares two string variables to hold these values:

```

static void Main(string[] args)
{
    string ServiceBusConnectionString = "";
    string QueueName = "";

    for (int i = 0; i < args.Length; i++)
    {
        var p = new Program();
        if (args[i] == "-ConnectionString")
        {
            Console.WriteLine($"ConnectionString: {args[i+1]}");
            ServiceBusConnectionString = args[i + 1];
        }
        else if(args[i] == "-QueueName")
        {
            Console.WriteLine($"QueueName: {args[i+1]}");
            QueueName = args[i + 1];
        }
    }

    if (ServiceBusConnectionString != "" && QueueName != "")
        MainAsync(ServiceBusConnectionString, QueueName).GetAwaiter().GetResult();
    else
    {
        Console.WriteLine("Specify -ConnectionString and -QueueName to execute the example.");
        Console.ReadKey();
    }
}

```

The `Main()` method then starts the asynchronous message loop, `MainAsync()`.

## Message loop

The `MainAsync()` method creates a queue client with the command-line arguments, calls a receiving message handler named `RegisterOnMessageHandlerAndReceiveMessages()`, and sends the set of messages:

```

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

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

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

    // Send Messages
    await SendMessagesAsync(numberOfMessages);

    Console.ReadKey();

    await queueClient.CloseAsync();
}

```

The `RegisterOnMessageHandlerAndReceiveMessages()` method simply sets a few message handler options, then calls the queue client's `RegisterMessageHandler()` method, which starts the receiving:

```

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);
}

```

## Send messages

The message creation and send operations occur in the `SendMessagesAsync()` method:

```

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}");
    }
}

```

## Process messages

The `ProcessMessagesAsync()` method acknowledges, processes, and completes the receipt of the 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.
    await queueClient.CompleteAsync(message.SystemProperties.LockToken);
}

```

#### **NOTE**

You can manage Service Bus resources with [Service Bus Explorer](#). The Service Bus Explorer allows users to connect to a Service Bus namespace and administer messaging entities in an easy manner. The tool provides advanced features like import/export functionality or the ability to test topic, queues, subscriptions, relay services, notification hubs and events hubs.

## Next steps

In this article, you created a Service Bus namespace and other resources required to send and receive messages from a queue. To learn more about writing code to send and receive messages, continue to the tutorials in the [\*\*Send and receive messages\*\*](#) section.

[Send and receive messages](#)

# Quickstart: Use Azure PowerShell to create a Service Bus queue

12/20/2019 • 6 minutes to read • [Edit Online](#)

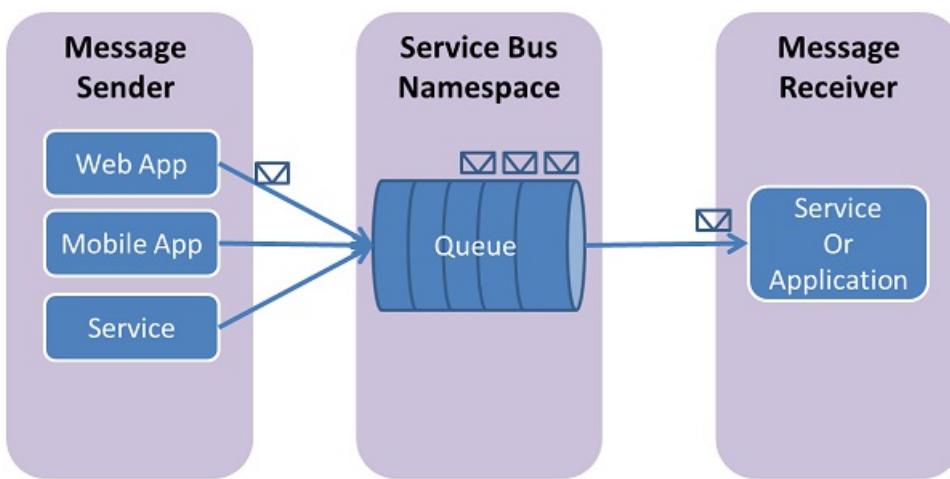
This quickstart describes how to send and receive messages to and from a Service Bus queue, using PowerShell to create a messaging namespace and a queue within that namespace, and to obtain the authorization credentials on that namespace. The procedure then shows how to send and receive messages from this queue using the [.NET Standard library](#).

## NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

## 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.

# Prerequisites

To complete this tutorial, make sure you have installed:

- An Azure subscription. If you don't have an Azure subscription, create a [free account](#) before you begin.
- [Visual Studio 2017 Update 3 \(version 15.3, 26730.01\)](#) or later. You use Visual Studio to build a sample that sends messages to and receives message from a queue. The sample is to test the queue you created in the portal.
- [NET Core SDK](#), version 2.0 or later.

This quickstart requires that you are running the latest version of Azure PowerShell. If you need to install or upgrade, see [Install and Configure Azure PowerShell](#). If you are familiar with Azure Cloud Shell, you could use it without installing Azure PowerShell on your machine. For details about Azure Cloud Shell, see [Overview of Azure Cloud Shell](#)

## Sign in to Azure

1. First, install the Service Bus PowerShell module, if you haven't already:

```
Install-Module Az.ServiceBus
```

2. Run the following command to sign in to Azure:

```
Login-AzAccount
```

3. Issue the following commands to set the current subscription context, or to see the currently active subscription:

```
Select-AzSubscription -SubscriptionName "MyAzureSubName"  
Get-AzContext
```

## Provision resources

From the PowerShell prompt, issue the following commands to provision Service Bus resources. Be sure to replace all placeholders with the appropriate values:

```
# Create a resource group  
New-AzResourceGroup -Name my-resourcegroup -Location eastus  
  
# Create a Messaging namespace  
New-AzServiceBusNamespace -ResourceGroupName my-resourcegroup -NamespaceName namespace-name -Location eastus  
  
# Create a queue  
New-AzServiceBusQueue -ResourceGroupName my-resourcegroup -NamespaceName namespace-name -Name queue-name -  
EnablePartitioning $False  
  
# Get primary connection string (required in next step)  
Get-AzServiceBusKey -ResourceGroupName my-resourcegroup -Namespace namespace-name -Name  
RootManageSharedAccessKey
```

After the `Get-AzServiceBusKey` cmdlet runs, copy and paste the connection string and the queue name you selected, to a temporary location such as Notepad. You will need it in the next step.

## Send and receive messages

After the namespace and queue are created, and you have the necessary credentials, you are ready to send and receive messages. You can examine the code in [this GitHub sample folder](#).

To run the code, do the following:

1. Clone the [Service Bus GitHub repository](#) by issuing the following command:

```
git clone https://github.com/Azure/azure-service-bus.git
```

2. Navigate to the sample folder

```
azure-service-bus\samples\DotNet\GettingStarted\BasicSendReceiveQuickStart\BasicSendReceiveQuickStart
```

3. If you have not done so already, obtain the connection string using the following PowerShell cmdlet. Be sure to replace `my-resourcegroup` and `namespace-name` with your specific values:

```
Get-AzServiceBusKey -ResourceGroupName my-resourcegroup -Namespace namespace-name -  
Name RootManageSharedAccessKey
```

4. At the PowerShell prompt, type the following command:

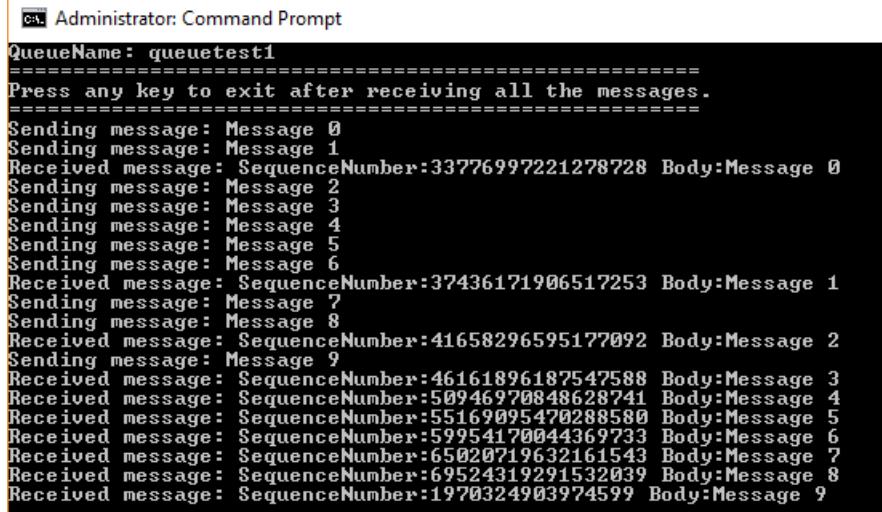
```
dotnet build
```

5. Navigate to the `bin\Debug\netcoreapp2.0` folder.

6. Type the following command to run the program. Be sure to replace `myConnectionString` with the value you previously obtained, and `myQueueName` with the name of the queue you created:

```
dotnet BasicSendReceiveQuickStart.dll -ConnectionString "myConnectionString" -QueueName "myQueueName"
```

7. Observe 10 messages being sent to the queue, and subsequently received from the queue:



```
Administrator: Command Prompt  
QueueName: queuetest1  
=====  
Press any key to exit after receiving all the messages.  
=====  
Sending message: Message 0  
Sending message: Message 1  
Received message: SequenceNumber:33776997221278728 Body:Message 0  
Sending message: Message 2  
Sending message: Message 3  
Sending message: Message 4  
Sending message: Message 5  
Sending message: Message 6  
Received message: SequenceNumber:37436171906517253 Body:Message 1  
Sending message: Message 7  
Sending message: Message 8  
Received message: SequenceNumber:41658296595177092 Body:Message 2  
Sending message: Message 9  
Received message: SequenceNumber:46161896187547588 Body:Message 3  
Received message: SequenceNumber:50946970848628741 Body:Message 4  
Received message: SequenceNumber:55169095470288580 Body:Message 5  
Received message: SequenceNumber:59954170044369733 Body:Message 6  
Received message: SequenceNumber:65020719632161543 Body:Message 7  
Received message: SequenceNumber:69524319291532039 Body:Message 8  
Received message: SequenceNumber:1970324903974599 Body:Message 9
```

## Clean up resources

Run the following command to remove the resource group, namespace, and all related resources:

```
Remove-AzResourceGroup -Name my-resourcegroup
```

# Understand the sample code

This section contains more details about what the sample code does.

## Get connection string and queue

The connection string and queue name are passed to the `Main()` method as command-line arguments. `Main()` declares two string variables to hold these values:

```
static void Main(string[] args)
{
    string ServiceBusConnectionString = "";
    string QueueName = "";

    for (int i = 0; i < args.Length; i++)
    {
        var p = new Program();
        if (args[i] == "-ConnectionString")
        {
            Console.WriteLine($"ConnectionString: {args[i+1]}");
            ServiceBusConnectionString = args[i + 1];
        }
        else if(args[i] == "-QueueName")
        {
            Console.WriteLine($"QueueName: {args[i+1]}");
            QueueName = args[i + 1];
        }
    }

    if (ServiceBusConnectionString != "" && QueueName != "")
        MainAsync(ServiceBusConnectionString, QueueName).GetAwaiter().GetResult();
    else
    {
        Console.WriteLine("Specify -ConnectionString and -QueueName to execute the example.");
        Console.ReadKey();
    }
}
```

The `Main()` method then starts the asynchronous message loop, `MainAsync()`.

## Message loop

The `MainAsync()` method creates a queue client with the command-line arguments, calls a receiving message handler named `RegisterOnMessageHandlerAndReceiveMessages()`, and sends the set of messages:

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

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

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

    // Send Messages
    await SendMessagesAsync(numberOfMessages);

    Console.ReadKey();

    await queueClient.CloseAsync();
}
```

The `RegisterOnMessageHandlerAndReceiveMessages()` method simply sets a few message handler options, then calls the queue client's `RegisterMessageHandler()` method, which starts the receiving:

```
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);
}
```

## Send messages

The message creation and send operations occur in the `SendMessagesAsync()` method:

```
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}");
    }
}
```

## Process messages

The `ProcessMessagesAsync()` method acknowledges, processes, and completes the receipt of the 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.
    await queueClient.CompleteAsync(message.SystemProperties.LockToken);
}
```

#### NOTE

You can manage Service Bus resources with [Service Bus Explorer](#). The Service Bus Explorer allows users to connect to a Service Bus namespace and administer messaging entities in an easy manner. The tool provides advanced features like import/export functionality or the ability to test topic, queues, subscriptions, relay services, notification hubs and events hubs.

## Next steps

In this article, you created a Service Bus namespace and other resources required to send and receive messages from a queue. To learn more about writing code to send and receive messages, continue to the tutorials in the [Send and receive messages](#) section.

[Send and receive messages](#)

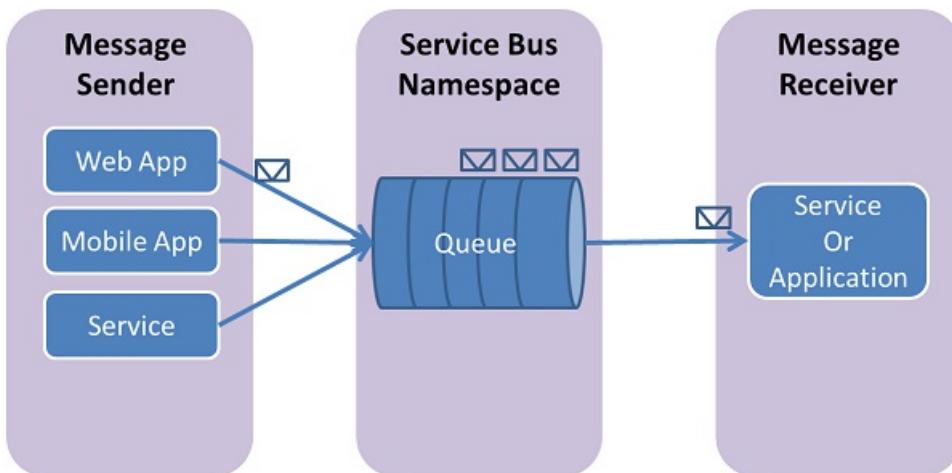
# Quickstart: Use the Azure CLI to create a Service Bus queue

5/30/2019 • 9 minutes to read • [Edit Online](#)

This quickstart describes how to send and receive messages with Service Bus by using the Azure CLI and the Service Bus Java library. Finally, if you're interested in more technical details, you can [read an explanation](#) of the key elements of the sample code.

## 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.

## Prerequisites

If you don't have an Azure subscription, you can create a [free account](#) before you begin.

## Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

OPTION	EXAMPLE/LINK
Select <b>Try It</b> in the upper-right corner of a code block. Selecting <b>Try It</b> doesn't automatically copy the code to Cloud Shell.	
Go to <a href="https://shell.azure.com">https://shell.azure.com</a> , or select the <b>Launch Cloud Shell</b> button to open Cloud Shell in your browser.	
Select the <b>Cloud Shell</b> button on the menu bar at the upper right in the <a href="#">Azure portal</a> .	

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

## Sign in to Azure

If you use the **Try It** button to launch the Cloud Shell, sign in to Azure using your credentials.

If you launched the Cloud Shell in your Web browser either directly or in the Azure portal, switch to **Bash** if you see **PowerShell** in the top-left corner of the Cloud Shell.

## Use the Azure CLI to create resources

In Cloud Shell, from the Bash prompt issue the following commands to provision Service Bus resources. Be sure to replace all placeholders with the appropriate values: The Java sample program expects the queue name to be `BasicQueue`, so do not change it. You may want to copy/paste commands one-by-one so that you can replace the values before you run them.

```
# Create a resource group
resourceGroupName="myResourceGroup"

az group create --name $resourceGroupName --location eastus

# Create a Service Bus messaging namespace with a unique name
namespaceName=myNameSpace$RANDOM
az servicebus namespace create --resource-group $resourceGroupName --name $namespaceName --location eastus

# Create a Service Bus queue
az servicebus queue create --resource-group $resourceGroupName --namespace-name $namespaceName --name
BasicQueue

# Get the connection string for the namespace
connectionString=$(az servicebus namespace authorization-rule keys list --resource-group $resourceGroupName --
namespace-name $namespaceName --name RootManageSharedAccessKey --query primaryConnectionString --output tsv)
```

After the last command runs, copy and paste the connection string, and the queue name you selected, to a temporary location such as Notepad. You will need it in the next step.

## Send and receive messages

After you've created the namespace and queue, and you have the necessary credentials, you are ready to send and receive messages. You can examine the code in [this GitHub sample folder](#).

1. Clone the [Service Bus GitHub repository](#) on your computer by issuing the following command:

```
git clone https://github.com/Azure/azure-service-bus.git
```

2. Change your current directory to the sample folder, using forward slashes as path separators:

```
cd azure-service-bus/samples/Java/azure-servicebus/QueuesGettingStarted
```

3. Issue the following command to build the application:

```
mvn clean package -DskipTests
```

4. To run the program, issue the following command after replacing the connection string with the value you copied earlier:

```
java -jar ./target/queuesgettingstarted-1.0.0-jar-with-dependencies.jar -c "<SERVICE BUS NAMESPACE CONNECTION STRING>"
```

5. Observe 10 messages being sent to the queue. Ordering of messages is not guaranteed, but you can see the messages sent, then acknowledged and received, along with the payload data:

```
Message sending: Id = 0
Message sending: Id = 1
Message sending: Id = 2
Message sending: Id = 3
Message sending: Id = 4
Message sending: Id = 5
Message sending: Id = 6
Message sending: Id = 7
Message sending: Id = 8
Message sending: Id = 9
Message acknowledged: Id = 9
Message acknowledged: Id = 3
Message received:
    MessageId = 9,
    SequenceNumber = 54324670505156609,
    EnqueuedTimeUtc = 2019-02-25T18:15:20.972Z,
    ExpiresAtUtc = 2019-02-25T18:17:20.972Z,
    ContentType = "application/json",
    Content: [ firstName = Nikolaus, name = Kopernikus ]

Message acknowledged: Id = 2
Message acknowledged: Id = 5
Message acknowledged: Id = 1
Message acknowledged: Id = 8
Message acknowledged: Id = 7
Message acknowledged: Id = 0
Message acknowledged: Id = 6
Message acknowledged: Id = 4
Message received:
    MessageId = 3,
    SequenceNumber = 58828270132527105,
    EnqueuedTimeUtc = 2019-02-25T18:15:20.972Z,
    ExpiresAtUtc = 2019-02-25T18:17:20.972Z,
```

```
        ContentType = "application/json",
        Content: [ firstName = Steven, name = Hawking ]\n\n    Message received:
        messageId = 2,
        SequenceNumber = 9288674231451649,
        EnqueuedTimeUtc = 2019-02-25T18:15:21.012Z,
        ExpiresAtUtc = 2019-02-25T18:17:21.012Z,
        ContentType = "application/json",
        Content: [ firstName = Marie, name = Curie ]\n\n    Message received:
        messageId = 1,
        SequenceNumber = 22799473113563137,
        EnqueuedTimeUtc = 2019-02-25T18:15:21.025Z,
        ExpiresAtUtc = 2019-02-25T18:17:21.025Z,
        ContentType = "application/json",
        Content: [ firstName = Werner, name = Heisenberg ]\n\n    Message received:
        messageId = 8,
        SequenceNumber = 67835469387268097,
        EnqueuedTimeUtc = 2019-02-25T18:15:21.028Z,
        ExpiresAtUtc = 2019-02-25T18:17:21.028Z,
        ContentType = "application/json",
        Content: [ firstName = Johannes, name = Kepler ]\n\n    Message received:
        messageId = 7,
        SequenceNumber = 4785074604081153,
        EnqueuedTimeUtc = 2019-02-25T18:15:21.020Z,
        ExpiresAtUtc = 2019-02-25T18:17:21.020Z,
        ContentType = "application/json",
        Content: [ firstName = Galileo, name = Galilei ]\n\n    Message received:
        messageId = 5,
        SequenceNumber = 13792273858822145,
        EnqueuedTimeUtc = 2019-02-25T18:15:21.027Z,
        ExpiresAtUtc = 2019-02-25T18:17:21.027Z,
        ContentType = "application/json",
        Content: [ firstName = Niels, name = Bohr ]\n\n    Message received:
        messageId = 0,
        SequenceNumber = 18295873486192641,
        EnqueuedTimeUtc = 2019-02-25T18:15:21.021Z,
        ExpiresAtUtc = 2019-02-25T18:17:21.021Z,
        ContentType = "application/json",
        Content: [ firstName = Albert, name = Einstein ]\n\n    Message received:
        messageId = 6,
        SequenceNumber = 281474976710657,
        EnqueuedTimeUtc = 2019-02-25T18:15:21.019Z,
        ExpiresAtUtc = 2019-02-25T18:17:21.019Z,
        ContentType = "application/json",
        Content: [ firstName = Michael, name = Faraday ]\n\n    Message received:
        messageId = 4,
        SequenceNumber = 63331869759897601,
        EnqueuedTimeUtc = 2019-02-25T18:15:20.964Z,
        ExpiresAtUtc = 2019-02-25T18:17:20.964Z,
        ContentType = "application/json",
        Content: [ firstName = Isaac, name = Newton ]
```

## Clean up resources

In the Azure Cloud Shell, run the following command to remove the resource group, namespace, and all related resources:

```
az group delete --resource-group myResourceGroup
```

## Understand the sample code

This section contains more details about key sections of the sample code. You can browse the code, located in the GitHub repository [here](#).

### Get connection string

The runApp method reads the connection string value from the arguments to the program.

```
public static void main(String[] args) {

    System.exit(runApp(args, (connectionString) -> {
        QueuesGettingStarted app = new QueuesGettingStarted();
        try {
            app.run(connectionString);
            return 0;
        } catch (Exception e) {
            System.out.printf("%s", e.toString());
            return 1;
        }
    }));
}

static final String SB_SAMPLES_CONNECTIONSTRING = "SB_SAMPLES_CONNECTIONSTRING";

public static int runApp(String[] args, Function<String, Integer> run) {
    try {

        String connectionString = null;

        // parse connection string from command line
        Options options = new Options();
        options.addOption(new Option("c", true, "Connection string"));
        CommandLineParser clp = new DefaultParser();
        CommandLine cl = clp.parse(options, args);
        if (cl.getOptionValue("c") != null) {
            connectionString = cl.getOptionValue("c");
        }

        // get overrides from the environment
        String env = System.getenv(SB_SAMPLES_CONNECTIONSTRING);
        if (env != null) {
            connectionString = env;
        }

        if (connectionString == null) {
            HelpFormatter formatter = new HelpFormatter();
            formatter.printHelp("run jar with", "", options, "", true);
            return 2;
        }
        return run.apply(connectionString);
    } catch (Exception e) {
        System.out.printf("%s", e.toString());
        return 3;
    }
}
```

## Create queue clients to send and receive

To send and receive messages, the `run()` method creates queue client instances, which are constructed from the connection string and the queue name. This code creates two queue clients, one each for sending and receiving:

```
public void run(String connectionString) throws Exception {

    // Create a QueueClient instance for receiving using the connection string builder
    // We set the receive mode to "PeekLock", meaning the message is delivered
    // under a lock and must be acknowledged ("completed") to be removed from the queue
    QueueClient receiveClient = new QueueClient(new ConnectionStringBuilder(connectionString, "BasicQueue"),
        ReceiveMode.PEEKLOCK);
    // We are using single thread executor as we are only processing one message at a time
    ExecutorService executorService = Executors.newSingleThreadExecutor();
    this.registerReceiver(receiveClient, executorService);

    // Create a QueueClient instance for sending and then asynchronously send messages.
    // Close the sender once the send operation is complete.
    QueueClient sendClient = new QueueClient(new ConnectionStringBuilder(connectionString, "BasicQueue"),
        ReceiveMode.PEEKLOCK);
    this.sendMessagesAsync(sendClient).thenRunAsync(() -> sendClient.closeAsync());

    // wait for ENTER or 10 seconds elapsing
    waitForEnter(10);

    // shut down receiver to close the receive loop
    receiveClient.close();
    executorService.shutdown();
}
```

## Construct and send messages

The `sendMessagesAsync()` method creates a set of 10 messages and asynchronously sends them using the queue client:

```

CompletableFuture<Void> sendMessagesAsync(QueueClient sendClient) {
    List<HashMap<String, String>> data =
        GSON.fromJson(
            "[" +
                "{\"name' = 'Einstein', 'firstName' = 'Albert'}," +
                "{\"name' = 'Heisenberg', 'firstName' = 'Werner'}," +
                "{\"name' = 'Curie', 'firstName' = 'Marie'}," +
                "{\"name' = 'Hawking', 'firstName' = 'Steven'}," +
                "{\"name' = 'Newton', 'firstName' = 'Isaac'}," +
                "{\"name' = 'Bohr', 'firstName' = 'Niels'}," +
                "{\"name' = 'Faraday', 'firstName' = 'Michael'}," +
                "{\"name' = 'Galilei', 'firstName' = 'Galileo'}," +
                "{\"name' = 'Kepler', 'firstName' = 'Johannes'}," +
                "{\"name' = 'Copernicus', 'firstName' = 'Nikolaus'}" +
            "]",
            new TypeToken<List<HashMap<String, String>>>() {}.getType());
}

List<CompletableFuture> tasks = new ArrayList<>();
for (int i = 0; i < data.size(); i++) {
    final String messageId = Integer.toString(i);
    Message message = new Message(GSON.toJson(data.get(i), Map.class).getBytes(UTF_8));
    message.setContentType("application/json");
    message.setLabel("Scientist");
    message.setMessageId(messageId);
    message.setTimeToLive(Duration.ofMinutes(2));
    System.out.printf("\nMessage sending: Id = %s", message.getMessageId());
    tasks.add(
        sendClient.sendAsync(message).thenRunAsync(() -> {
            System.out.printf("\n\tMessage acknowledged: Id = %s", message.getMessageId());
        }));
}
return CompletableFuture.allOf(tasks.toArray(new CompletableFuture<?>[tasks.size()]));
}

```

## Receive messages

The `registerReceiver()` method registers the `RegisterMessageHandler` callback and also sets some message handler options:

```

void registerReceiver(QueueClient queueClient, ExecutorService executorService) throws Exception {

    // register the RegisterMessageHandler callback with executor service
    queueClient.registerMessageHandler(new IMessageHandler() {
        // callback invoked when the message handler loop has obtained a
        message
        public CompletableFuture<Void> onMessageAsync(IMessage message) {
            // receives message is passed to callback
            if (message.getLabel() != null &&
                message.getContentType() != null &&
                message.getLabel().contentEquals("Scientist") &&

            message.getContentType().contentEquals("application/json")) {

                byte[] body = message.getBody();
                Map scientist = GSON.fromJson(new String(body, UTF_8),
                    Map.class);

                System.out.printf(
                    "\n\t\t\t\tMessage received:
\n\t\t\t\t\tMessageId = %s, \n\t\t\t\t\tSequenceNumber = %s, \n\t\t\t\t\tEnqueuedUtc = %s," +
                    "\n\t\t\t\tExpiresAtUtc = %s,
\n\t\t\t\tContentType = \"%s\", \n\t\t\t\tContent: [ firstName = %s, name = %s ]\n",
                    message.getMessageId(),
                    message.getSequenceNumber(),
                    message.getEnqueuedTimeUtc(),
                    message.getExpiresAtUtc(),
                    message.getContentType(),
                    scientist != null ? scientist.get("firstName") :
                    "",
                    scientist != null ? scientist.get("name") : "");
            }
            return CompletableFuture.completedFuture(null);
        }
    }

    // callback invoked when the message handler has an exception to
    report
    public void notifyException(Throwable throwable, ExceptionPhase
exceptionPhase) {
        System.out.printf(exceptionPhase + "-" +
throwable.getMessage());
    }
},
// 1 concurrent call, messages are auto-completed, auto-renew duration
new MessageHandlerOptions(1, true, Duration.ofMinutes(1)),
executorService);
}

```

#### NOTE

You can manage Service Bus resources with [Service Bus Explorer](#). The Service Bus Explorer allows users to connect to a Service Bus namespace and administer messaging entities in an easy manner. The tool provides advanced features like import/export functionality or the ability to test topic, queues, subscriptions, relay services, notification hubs and events hubs.

## Next steps

In this article, you created a Service Bus namespace and other resources required to send and receive messages from a queue. To learn more about writing code to send and receive messages, continue to the tutorials in the [Send and receive messages](#) section.

Send and receive messages

# Quickstart: Create a Service Bus namespace and a queue using an Azure Resource Manager template

12/23/2019 • 3 minutes 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**.

## NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

## 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": "2017-04-01",  
        "name": "[parameters('serviceBusNamespaceName')]",  
        "type": "Microsoft.ServiceBus/namespaces",  
        "location": "[parameters('location')]",  
        "sku": {  
            "name": "Standard"  
        },  
        "properties": {},  
        "resources": [{  
            "apiVersion": "[variables('sbVersion')]",  
            "name": "[parameters('serviceBusQueueName')]",  
            "type": "Queues",  
            "dependsOn": [  
                "[concat('Microsoft.ServiceBus/namespaces/', parameters('serviceBusNamespaceName'))]"  
            ],  
            "properties": {  
                "path": "[parameters('serviceBusQueueName')]"  
            }  
        }]  
    }]  
}
```

For JSON syntax and properties, see [namespaces](#) and [queues](#).

## Commands to run deployment

To deploy the resources to Azure, you must be signed 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:

- [Manage Azure resources by using Azure PowerShell](#)
- [Manage Azure resources by using Azure CLI.](#)

The following examples assume you already have a resource group in your account with the specified name.

### PowerShell

```
New-AzResourceGroupDeployment -ResourceGroupName <resource-group-name> -TemplateFile  
<https://raw.githubusercontent.com/azure/azure-quickstart-templates/master/201-servicebus-create-  
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/201-servicebus-create-  
queue/azuredploy.json>
```

## Next steps

See the following topic that shows how to create an authorization rule for the namespace/queue: [Create a Service Bus authorization rule for namespace and queue using an Azure Resource Manager template](#)

Learn how to manage these resources by viewing these articles:

- [Manage Service Bus with PowerShell](#)
- [Manage Service Bus resources with the Service Bus Explorer](#)

# Get started with Service Bus queues

1/24/2020 • 7 minutes to read • [Edit Online](#)

In this tutorial, you create .NET Core console applications to send messages to and receive messages from a Service Bus queue.

## Prerequisites

- [Visual Studio 2019](#).
- [NET Core SDK](#), version 2.0 or later.
- An Azure subscription. To complete this tutorial, you need an Azure account. You can activate your [MSDN subscriber benefits](#) or sign up for a [free account](#).
- If you don't have a queue to work with, follow steps in the [Use Azure portal to create a Service Bus queue](#) article to create a queue.
  - Read the quick overview of Service Bus queues.
  - Create a Service Bus namespace.
  - Get the connection string.
  - Create a Service Bus queue.

## Send messages to the queue

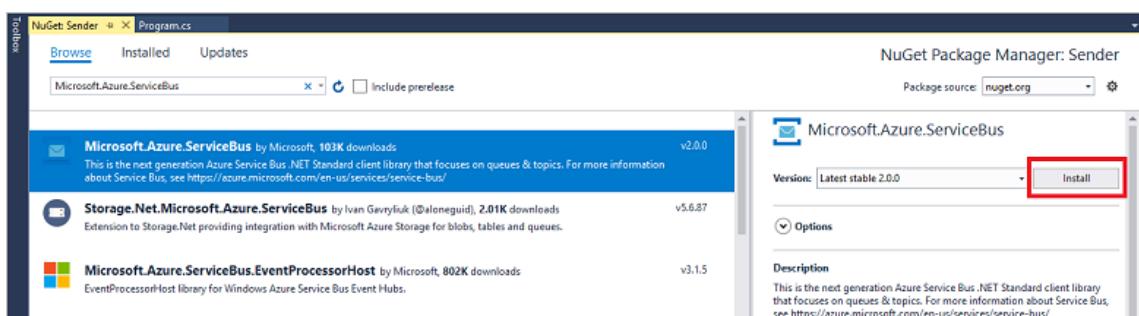
To send messages to the queue, write a C# console application using Visual Studio.

### Create a console application

Launch Visual Studio and create a new **Console App (.NET Core)** project for C#. This example names the app *CoreSenderApp*.

### Add the Service Bus NuGet package

1. Right-click the newly created project and select **Manage NuGet Packages**.
2. Select **Browse**. Search for and select [Microsoft.Azure.ServiceBus](#).
3. Select **Install** to complete the installation, then close the NuGet Package Manager.



### 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. In the `Program` class, declare the following variables:

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

Enter your connection string for the namespace as the `ServiceBusConnectionString` variable. Enter your queue name.

3. Replace the `Main()` method with the following `async Main` method. It calls the `SendMessagesAsync()` method that you will add in the next step to send messages to the queue.

```
public static async Task Main(string[] args)
{
    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();
}
```

4. Directly after the `MainAsync()` method, add the following `SendMessagesAsync()` method that does 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}");
    }
}
```

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;

        public static async Task Main(string[] args)
        {
            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();
        }

        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}");
            }
        }
    }
}
```

Run the program and check the Azure portal.

Select the name of your queue in the namespace **Overview** window to display queue **Essentials**.

The screenshot shows the Azure portal interface for a Service Bus Queue named 'contosoqueue01'. The left sidebar has sections for Overview, Access control (IAM), Diagnose and solve problems, Settings (Shared access policies, Metrics (preview), Properties, Locks, Export template), Support + troubleshooting, and New support request. The main pane displays the queue's namespace as 'messagenamespace01' and its URL as 'https://messagenamespace01.servicebus.windows.net/contosoqueue01'. It shows the following metrics:

- Active message count: 10 MESSAGES
- Scheduled message count: 0 MESSAGES
- Dead-letter message count: 0 MESSAGES
- Transfer message count: 0 MESSAGES
- Transfer dead-letter message count: 0 MESSAGES

A circular gauge indicates 100% FREE SPACE. The current size of the queue is 1.7 KB, and the max size is 1 GB.

The **Active message count** value for the queue is now **10**. Each time you run this sender app without retrieving the messages, this value increases by 10.

The current size of the queue increments the **CURRENT** value in **Essentials** each time the app adds messages to the queue.

The next section describes how to retrieve these messages.

## Receive messages from the queue

To receive the messages you sent, create another **Console App (.NET Core)** application. Install the **Microsoft.Azure.ServiceBus** NuGet package, as you did for the 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. In the `Program` class, declare the following variables:

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

Enter your connection string for the namespace as the `ServiceBusConnectionString` variable. Enter your queue name.

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

```

public static async Task Main(string[] args)
{
    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();
}

```

4. Directly after the `MainAsync()` method, add the following method, which 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);
}

```

5. 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.
}

```

6. 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;
}
```

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.

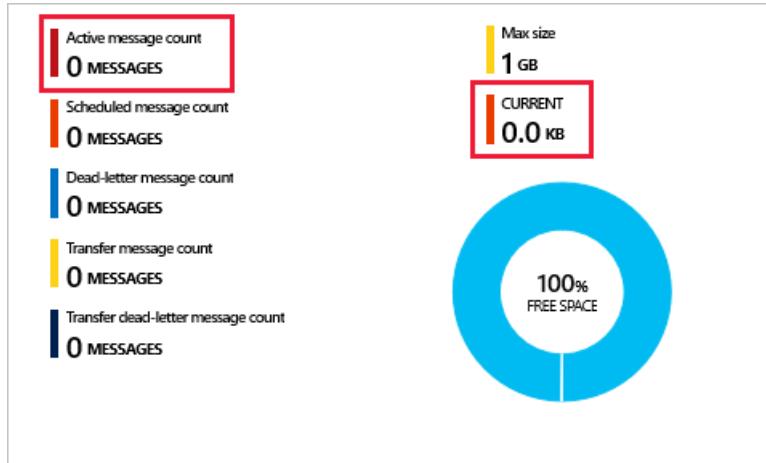
    // If queueClient has already been Closed, you may chose to not call CompleteAsync() or
AbandonAsync() etc. calls
    // to avoid unnecessary exceptions.
}

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. The **Active message count** and **CURRENT** values are now **0**.



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

**NOTE**

You can manage Service Bus resources with [Service Bus Explorer](#). The Service Bus Explorer allows users to easily connect to a Service Bus namespace and administer messaging entities. The tool provides advanced features like import/export functionality or the ability to test topics, queues, subscriptions, relay services, notification hubs, and event hubs.

## Next steps

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

# Quickstart: Use Azure Service Bus queues with Java to send and receive messages

1/24/2020 • 7 minutes to read • [Edit Online](#)

In this tutorial, you learn how to create Java applications to send messages to and receive messages from an Azure Service Bus queue.

## NOTE

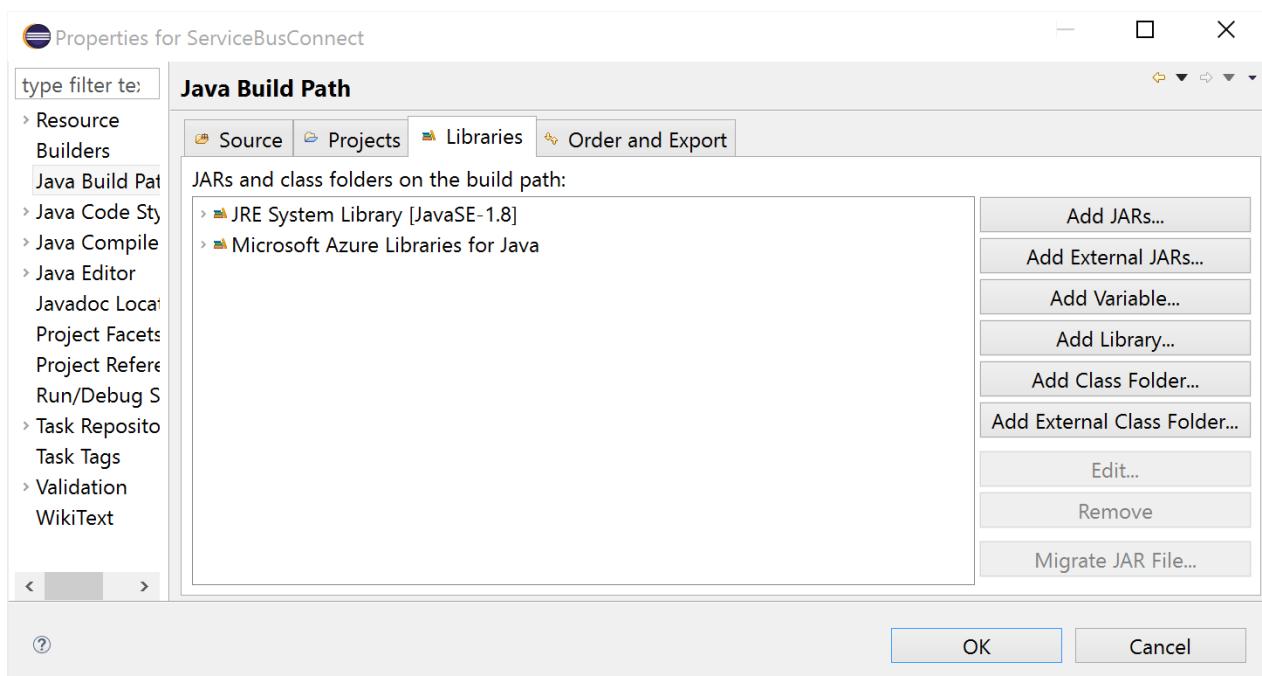
You can find Java samples on GitHub in the [azure-service-bus repository](#).

## Prerequisites

1. An Azure subscription. To complete this tutorial, you need an Azure account. You can activate your [MSDN subscriber benefits](#) or sign up for a [free account](#).
2. If you don't have a queue to work with, follow steps in the [Use Azure portal to create a Service Bus queue](#) article to create a queue.
  - a. Read the quick **overview** of Service Bus **queues**.
  - b. Create a Service Bus **namespace**.
  - c. Get the **connection string**.
  - d. Create a Service Bus **queue**.
3. Install [Azure SDK for Java](#).

## 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.google.gson.reflect.TypeToken;
import com.microsoft.azure.servicebus.*;
import com.microsoft.azure.servicebus.primitives.ConnectionStringBuilder;
import com.google.gson.Gson;

import static java.nio.charset.StandardCharsets.*;

import java.time.Duration;
import java.util.*;
import java.util.concurrent.*;

import org.apache.commons.cli.*;
```

## Send messages to a queue

To send messages to a Service Bus Queue, your application instantiates a **QueueClient** object and sends messages asynchronously. The following code shows how to send a message for a Queue that was created through the portal.

```

public void run() throws Exception {
    // Create a QueueClient instance and then asynchronously send messages.
    // Close the sender once the send operation is complete.
    QueueClient sendClient = new QueueClient(new ConnectionStringBuilder(ConnectionString, QueueName),
    ReceiveMode.PEEKLOCK);
    this.sendMessageAsync(sendClient).thenRunAsync(() -> sendClient.closeAsync());

    sendClient.close();
}

CompletableFuture<Void> sendMessagesAsync(QueueClient sendClient) {
    List<HashMap<String, String>> data =
        GSON.fromJson(
            "[" +
                "{ 'name' = 'Einstein', 'firstName' = 'Albert'}," +
                "{ 'name' = 'Heisenberg', 'firstName' = 'Werner'}," +
                "{ 'name' = 'Curie', 'firstName' = 'Marie'}," +
                "{ 'name' = 'Hawking', 'firstName' = 'Steven'}," +
                "{ 'name' = 'Newton', 'firstName' = 'Isaac'}," +
                "{ 'name' = 'Bohr', 'firstName' = 'Niels'}," +
                "{ 'name' = 'Faraday', 'firstName' = 'Michael'}," +
                "{ 'name' = 'Galilei', 'firstName' = 'Galileo'}," +
                "{ 'name' = 'Kepler', 'firstName' = 'Johannes'}," +
                "{ 'name' = 'Kopernikus', 'firstName' = 'Nikolaus'}" +
            "]",
            new TypeToken<List<HashMap<String, String>>>() {}.getType());
}

List<CompletableFuture> tasks = new ArrayList<>();
for (int i = 0; i < data.size(); i++) {
    final String messageId = Integer.toString(i);
    Message message = new Message(GSON.toJson(data.get(i), Map.class).getBytes(UTF_8));
    message.setContentType("application/json");
    message.setLabel("Scientist");
    message.setMessageId(messageId);
    message.setTimeToLive(Duration.ofMinutes(2));
    System.out.printf("\nMessage sending: Id = %s", message.getMessageId());
    tasks.add(
        sendClient.sendAsync(message).thenRunAsync(() -> {
            System.out.printf("\n\tMessage acknowledged: Id = %s", message.getMessageId());
        }));
}
return CompletableFuture.allOf(tasks.toArray(new CompletableFuture<?>[tasks.size()]));
}

```

Messages sent to, and received from Service Bus queues are instances of the [Message](#) class. Message 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 Message, and the appropriate serializer will then be used to serialize the object. Alternatively, you can provide a [java.io.InputStream](#) object.

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 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 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 :

```

public void run() throws Exception {
    // Create a QueueClient instance for receiving using the connection string builder
    // We set the receive mode to "PeekLock", meaning the message is delivered
    // under a lock and must be acknowledged ("completed") to be removed from the queue
    QueueClient receiveClient = new QueueClient(new ConnectionStringBuilder(ConnectionString, QueueName),
ReceiveMode.PEEKLOCK);
    this.registerReceiver(receiveClient);

    // shut down receiver to close the receive loop
    receiveClient.close();
}

void registerReceiver(QueueClient queueClient) throws Exception {
    // register the RegisterMessageHandler callback
    queueClient.registerMessageHandler(new IMessageHandler() {
// callback invoked when the message handler loop has obtained a message
        public CompletableFuture<Void> onMessageAsync(IMessage message) {
// receives message is passed to callback
            if (message.getLabel() != null &&
                message.getContentType() != null &&
                message.getLabel().contentEquals("Scientist") &&
                message.getContentType().contentEquals("application/json")) {

                byte[] body = message.getBody();
                Map scientist = GSON.fromJson(new String(body, UTF_8), Map.class);

                System.out.printf(
                    "\n\t\t\tMessage received: \n\t\t\t\tMessageId = %s,
\n\t\t\t\tSequenceNumber = %s, \n\t\t\t\tEnqueuedTimeUtc = %s," +
                    "\n\t\t\t\tExpiresAtUtc = %s, \n\t\t\t\tContentType = \"%s\",
\n\t\t\tContent: [ firstName = %s, name = %s ]\n",
                    message.getMessageId(),
                    message.getSequenceNumber(),
                    message.getEnqueuedTimeUtc(),
                    message.getExpiresAtUtc(),
                    message.getContentType(),
                    scientist != null ? scientist.get("firstName") : "",
                    scientist != null ? scientist.get("name") : "");
            }
            return CompletableFuture.completedFuture(null);
        }
    }

    // callback invoked when the message handler has an exception to report
    public void notifyException(Throwable throwable, ExceptionPhase exceptionPhase) {
        System.out.printf(exceptionPhase + "-" + throwable.getMessage());
    }
},
// 1 concurrent call, messages are auto-completed, auto-renew duration
new MessageHandlerOptions(1, true, Duration.ofMinutes(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 **getMessageId** method of the message, which remains constant across delivery attempts.

**NOTE**

You can manage Service Bus resources with [Service Bus Explorer](#). The Service Bus Explorer allows users to connect to a Service Bus namespace and administer messaging entities in an easy manner. The tool provides advanced features like import/export functionality or the ability to test topic, queues, subscriptions, relay services, notification hubs and events hubs.

## 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](#).

# Quickstart: How to use Service Bus queues with Node.js and the azure/service-bus package

1/27/2020 • 4 minutes to read • [Edit Online](#)

In this tutorial, you learn how to write a Nodejs program to send messages to and receive messages from a Service Bus queue using the new [@azure/service-bus](#) package. This package uses the faster [AMQP 1.0 protocol](#) whereas the older [azure-sb](#) package used [Service Bus REST run-time APIs](#). The samples are written in JavaScript.

## Prerequisites

- An Azure subscription. To complete this tutorial, you need an Azure account. You can activate your [MSDN subscriber benefits](#) or sign up for a [free account](#).
- If you don't have a queue to work with, follow steps in the [Use Azure portal to create a Service Bus queue](#) article to create a queue. Note down the connection string for your Service Bus instance and the name of the queue you created. We'll use these values in the samples.

### NOTE

- This tutorial works with samples that you can copy and run using [Nodejs](#). For instructions on how to create a Nodejs application, see [Create and deploy a Node.js application to an Azure Website](#), or [Node.js cloud service using Windows PowerShell](#).
- The new [@azure/service-bus](#) package does not support creation of queues yet. Please use the [@azure/arm-servicebus](#) package if you want to programmatically create them.

## Use Node Package Manager (NPM) to install the package

To install the npm package for Service Bus, open a command prompt that has `npm` in its path, change the directory to the folder where you want to have your samples and then run this command.

```
npm install @azure/service-bus
```

## Send messages to a queue

Interacting with a Service Bus queue starts with instantiating the [ServiceBusClient](#) class and using it to instantiate the [QueueClient](#) class. Once you have the queue client, you can create a sender and use either [send](#) or [sendBatch](#) method on it to send messages.

1. Open your favorite editor, such as [Visual Studio Code](#)
2. Create a file called `send.js` and paste the below code into it. This code will send 10 messages to your queue.

```

const { ServiceBusClient } = require("@azure/service-bus");

// Define connection string and related Service Bus entity names here
const connectionString = "";
const queueName = "";

async function main(){
    const sbClient = ServiceBusClient.createFromConnectionString(connectionString);
    const queueClient = sbClient.createQueueClient(queueName);
    const sender = queueClient.createSender();

    try {
        for (let i = 0; i < 10; i++) {
            const message= {
                body: `Hello world! ${i}`,
                label: `test`,
                userProperties: {
                    myCustomPropertyName: `my custom property value ${i}`
                }
            };
            console.log(`Sending message: ${message.body}`);
            await sender.send(message);
        }

        await queueClient.close();
    } finally {
        await sbClient.close();
    }
}

main().catch((err) => {
    console.log("Error occurred: ", err);
});

```

3. Enter the connection string and name of your queue in the above code.

4. Then run the command `node send.js` in a command prompt to execute this file.

Congratulations! You just sent messages to a Service Bus queue.

Messages have some standard properties like `label` and `messageId` that you can set when sending. If you want to set any custom properties, use the `userProperties`, which is a json object that can hold key-value pairs of your custom data.

Service Bus queues support a maximum message size of 256 KB in the [Standard tier](#) and 1 MB in the [Premium tier](#). There's no limit on the number of messages held in a queue but there's 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

Interacting with a Service Bus queue starts with instantiating the `ServiceBusClient` class and using it to instantiate the `QueueClient` class. Once you have the queue client, you can create a receiver and use either `receiveMessages` or `registerMessageHandler` method on it to receive messages.

1. Open your favorite editor, such as [Visual Studio Code](#)
2. Create a file called `receive.js` and paste the below code into it. This code will attempt to receive 10 messages from your queue. The actual count you receive depends on the number of messages in the queue and network latency.

```

const { ServiceBusClient, ReceiveMode } = require("@azure/service-bus");

// Define connection string and related Service Bus entity names here
const connectionString = "";
const queueName = "";

async function main(){
    const sbClient = ServiceBusClient.createFromConnectionString(connectionString);
    const queueClient = sbClient.createQueueClient(queueName);
    const receiver = queueClient.createReceiver(ReceiveMode.receiveAndDelete);
    try {
        const messages = await receiver.receiveMessages(10)
        console.log("Received messages:");
        console.log(messages.map(message => message.body));

        await queueClient.close();
    } finally {
        await sbClient.close();
    }
}

main().catch((err) => {
    console.log("Error occurred: ", err);
});

```

3. Enter the connection string and name of your queue in the above code.

4. Then run the command `node receiveMessages.js` in a command prompt to execute this file.

Congratulations! You just received messages from a Service Bus queue.

The `createReceiver` method takes in a `ReceiveMode` which is an enum with values `ReceiveAndDelete` and `PeekLock`. Remember to `settle your messages` if you use the `PeekLock` mode by using any of `complete()`, `abandon()`, `defer()`, or `deadletter()` methods on the message.

#### **NOTE**

You can manage Service Bus resources with [Service Bus Explorer](#). The Service Bus Explorer allows users to connect to a Service Bus namespace and administer messaging entities in an easy manner. The tool provides advanced features like import/export functionality or the ability to test topic, queues, subscriptions, relay services, notification hubs and events hubs.

## Next steps

To learn more, see the following resources.

- [Queues, topics, and subscriptions](#)
- Checkout other [Nodejs samples for Service Bus on GitHub](#)
- [Node.js Developer Center](#)

# Quickstart: Use Service Bus queues in Azure with Node.js and the azure-sb package

1/27/2020 • 8 minutes to read • [Edit Online](#)

In this tutorial, you learn how to create Node.js applications to send messages to and receive messages from an Azure Service Bus queue using the [azure-sb](#) package. The samples are written in JavaScript and use the Node.js [Azure module](#) which internally uses the `azure-sb` package.

The `azure-sb` package uses [Service Bus REST run-time APIs](#). You can get a faster experience using the new [@azure/service-bus](#) which uses the faster [AMQP 1.0 protocol](#). To learn more about the new package, see [How to use Service Bus queues with Node.js and @azure/service-bus package](#), otherwise continue reading to see how to use the `azure` package.

## Prerequisites

- An Azure subscription. To complete this tutorial, you need an Azure account. You can activate your [MSDN subscriber benefits](#) or sign up for a [free account](#).
- If you don't have a queue to work with, follow steps in the [Use Azure portal to create a Service Bus queue](#) article to create a queue.
  1. Read the quick [overview](#) of Service Bus **queues**.
  2. Create a Service Bus **namespace**.
  3. Get the **connection string**.

### NOTE

You will create a **queue** in the Service Bus namespace by using Node.js in this tutorial.

## 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 example:

```
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 isn't set, you must specify the account information when calling `createServiceBusService`.

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 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's no limit on the number of messages held in a queue but there's 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 when a failure occurs. 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 before 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 can't 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. It 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's 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` method is called, then the message will be redelivered to the application when it restarts. This approach 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 can't tolerate duplicate processing, then application developers should add additional logic to their application to handle duplicate message delivery. It's often achieved using the **MessageId** property of the message, which will remain constant across delivery attempts.

## NOTE

You can manage Service Bus resources with [Service Bus Explorer](#). The Service Bus Explorer allows users to connect to a Service Bus namespace and administer messaging entities in an easy manner. The tool provides advanced features like import/export functionality or the ability to test topic, queues, subscriptions, relay services, notification hubs and events hubs.

## 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](#)

# Quickstart: How to use Service Bus queues with PHP

1/24/2020 • 7 minutes to read • [Edit Online](#)

In this tutorial, you learn how to create PHP applications to send messages to and receive messages from a Service Bus queue.

## Prerequisites

1. An Azure subscription. To complete this tutorial, you need an Azure account. You can activate your [MSDN subscriber benefits](#) or sign up for a [free account](#).
2. If you don't have a queue to work with, follow steps in the [Use Azure portal to create a Service Bus queue](#) article to create a queue.
  - a. Read the quick **overview** of Service Bus **queues**.
  - b. Create a Service Bus **namespace**.
  - c. Get the **connection string**.

### NOTE

You will create a **queue** in the Service Bus namespace by using PHP in this tutorial.

3. [Azure SDK for PHP](#)

## 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."<br />";
}

```

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

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

## 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.

**NOTE**

You can manage Service Bus resources with [Service Bus Explorer](#). The Service Bus Explorer allows users to connect to a Service Bus namespace and administer messaging entities in an easy manner. The tool provides advanced features like import/export functionality or the ability to test topic, queues, subscriptions, relay services, notification hubs and events hubs.

## 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](#).

# Quickstart: Use Azure Service Bus queues with Python

1/27/2020 • 5 minutes to read • [Edit Online](#)

This article shows you how to use Python to create, send messages to, and receive messages from Azure Service Bus queues.

For more information about the Python Azure Service Bus libraries, see [Service Bus libraries for Python](#).

## Prerequisites

- An Azure subscription. You can activate your [Visual Studio or MSDN subscriber benefits](#) or sign up for a [free account](#).
- A Service Bus namespace, created by following the steps at [Quickstart: Use the Azure portal to create a Service Bus topic and subscriptions](#). Copy the primary connection string from the **Shared access policies** screen to use later in this article.
- Python 3.4x or above, with the [Python Azure Service Bus](#) package installed. For more information, see the [Python Installation Guide](#).

## Create a queue

A **ServiceBusClient** object lets you work with queues. To programmatically access Service Bus, add the following line near the top of your Python file:

```
from azure.servicebus import ServiceBusClient
```

Add the following code to create a **ServiceBusClient** object. Replace `<connectionstring>` with your Service Bus primary connection string value. You can find this value under **Shared access policies** in your Service Bus namespace in the [Azure portal](#).

```
sb_client = ServiceBusClient.from_connection_string('<connectionstring>')
```

The following code uses the `create_queue` method of the **ServiceBusClient** to create a queue named `taskqueue` with default settings:

```
sb_client.create_queue("taskqueue")
```

You can use options to override default queue settings, such as message time to live (TTL) or maximum topic size. The following code creates a queue called `taskqueue` with a maximum queue size of 5 GB and TTL value of 1 minute:

```
sb_client.create_queue("taskqueue", max_size_in_megabytes=5120,
                      default_message_time_to_live=datetime.timedelta(minutes=1))
```

## Send messages to a queue

To send a message to a Service Bus queue, an application calls the `send` method on the **ServiceBusClient** object. The following code example creates a queue client and sends a test message to the `taskqueue` queue. Replace `<connectionstring>` with your Service Bus primary connection string value.

```
from azure.servicebus import QueueClient, Message

# Create the QueueClient
queue_client = QueueClient.from_connection_string("<connectionstring>", "taskqueue")

# Send a test message to the queue
msg = Message(b'Test Message')
queue_client.send(msg)
```

## Message size limits and quotas

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's no limit on the number of messages a queue can hold, but there's a cap on the total size of the messages the queue holds. You can define queue size at creation time, with an upper limit of 5 GB.

For more information about quotas, see [Service Bus quotas](#).

## Receive messages from a queue

The queue client receives messages from a queue by using the `get_receiver` method on the **ServiceBusClient** object. The following code example creates a queue client and receives a message from the `taskqueue` queue. Replace `<connectionstring>` with your Service Bus primary connection string value.

```
from azure.servicebus import QueueClient, Message

# Create the QueueClient
queue_client = QueueClient.from_connection_string("<connectionstring>", "taskqueue")

# Receive the message from the queue
with queue_client.get_receiver() as queue_receiver:
    messages = queue_receiver.fetch_next(timeout=3)
    for message in messages:
        print(message)
        message.complete()
```

## Use the `peek_lock` parameter

The optional `peek_lock` parameter of `get_receiver` determines whether Service Bus deletes messages from the queue as they're read. The default mode for message receiving is *PeekLock*, or `peek_lock` set to **True**, which reads (peeks) and locks messages without deleting them from the queue. Each message must then be explicitly completed to remove it from the queue.

To delete messages from the queue as they're read, you can set the `peek_lock` parameter of `get_receiver` to **False**. Deleting messages as part of the receive operation is the simplest model, but only works if the application can tolerate missing messages if there's a failure. To understand this behavior, consider a scenario in which the consumer issues a receive request and then crashes before processing it. If the message was deleted on being received, when the application restarts and begins consuming messages again, it has missed the message it received before the crash.

If your application can't tolerate missed messages, receive is a two-stage operation. PeekLock finds the next message to be consumed, locks it to prevent other consumers from receiving it, and returns it to the application. After processing or storing the message, the application completes the second stage of the receive process by calling the `complete` method on the **Message** object. The `complete` method marks the message as being

consumed and removes it from the queue.

## 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 can't process a message for some reason, it can call the `unlock` method on the **Message** object. Service Bus unlocks the message within the queue and makes it available to be received again, either by the same or another consuming application.

There's also a timeout for messages locked within the queue. If an application fails to process a message before the lock timeout expires, for example if the application crashes, Service Bus unlocks the message automatically and makes it available to be received again.

If an application crashes after processing a message but before calling the `complete` method, the message is redelivered to the application when it restarts. This behavior is often called *At-least-once Processing*. Each message is processed at least once, but in certain situations the same message may be redelivered. If your scenario can't tolerate duplicate processing, you can use the **MessageId** property of the message, which remains constant across delivery attempts, to handle duplicate message delivery.

### TIP

You can manage Service Bus resources with [Service Bus Explorer](#). Service Bus Explorer lets you connect to a Service Bus namespace and easily administer messaging entities. The tool provides advanced features like import/export functionality and the ability to test topics, queues, subscriptions, relay services, notification hubs, and event hubs.

## Next steps

Now that you've learned the basics of Service Bus queues, see [Queues, topics, and subscriptions](#) to learn more.

# Quickstart: How to use Service Bus queues with Ruby

1/24/2020 • 6 minutes to read • [Edit Online](#)

In this tutorial, you learn how to create Ruby applications to send messages to and receive messages from a Service Bus queue. The samples are written in Ruby and use the Azure gem.

## Prerequisites

1. An Azure subscription. To complete this tutorial, you need an Azure account. You can activate your [MSDN subscriber benefits](#) or sign up for a [free account](#).
2. Follow steps in the [Use Azure portal to create a Service Bus queue](#) article.
  - a. Read the quick **overview** of Service Bus **queues**.
  - b. Create a Service Bus **namespace**.
  - c. Get the **connection string**.

### NOTE

You will create a **queue** in the Service Bus namespace by using Ruby in this tutorial.

## 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.

#### **NOTE**

You can manage Service Bus resources with [Service Bus Explorer](#). The Service Bus Explorer allows users to connect to a Service Bus namespace and administer messaging entities in an easy manner. The tool provides advanced features like import/export functionality or the ability to test topic, queues, subscriptions, relay services, notification hubs and events hubs.

## 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](#)

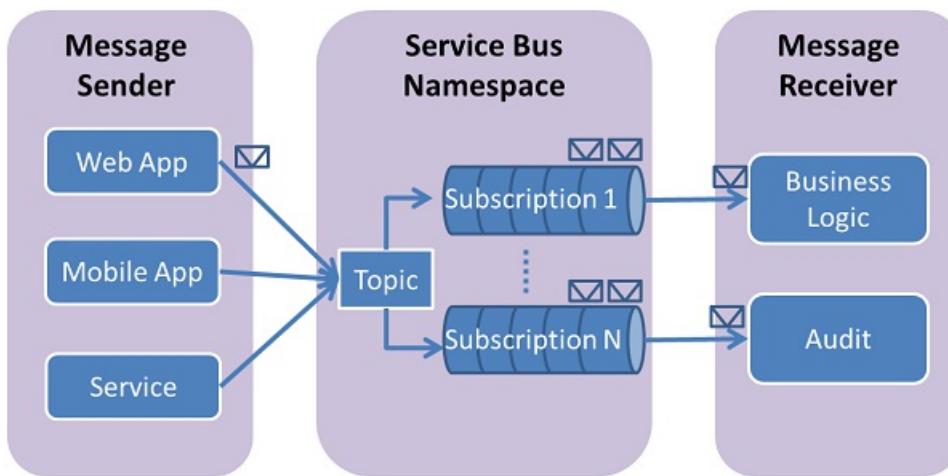
# Quickstart: Use the Azure portal to create a Service Bus topic and subscriptions to the topic

1/17/2020 • 4 minutes to read • [Edit Online](#)

In this quickstart, you use the Azure portal to create a Service Bus topic and then create subscriptions to that topic.

## 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 or restrict which messages to a topic are received by which topic subscriptions.

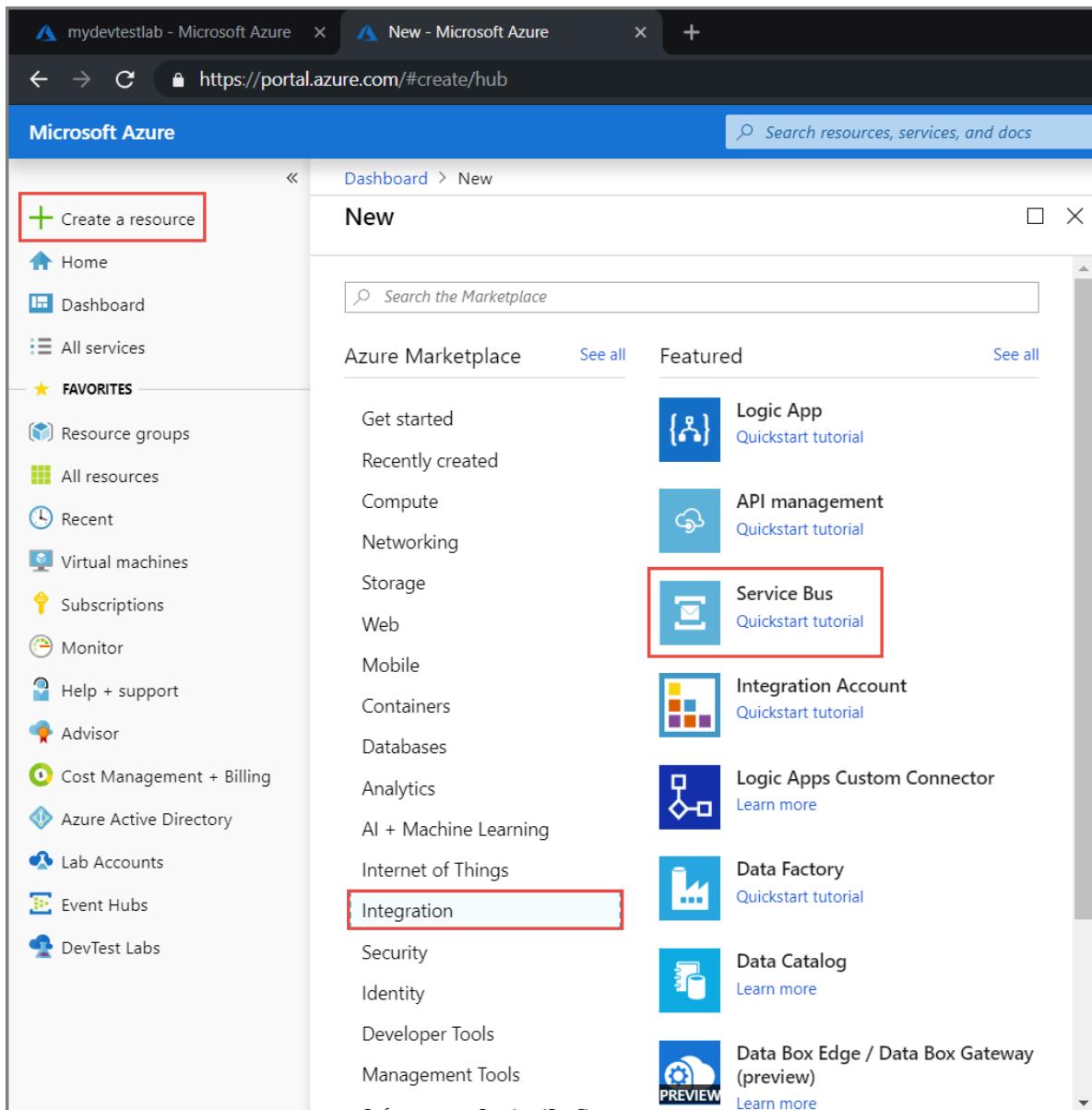
Service Bus topics and subscriptions enable you to scale to process a large number of messages across a large number of users and applications.

## Create a 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. Sign in to the [Azure portal](#)
2. In the left navigation pane of the portal, select **+ Create a resource**, select **Integration**, and then select **Service Bus**.



3. In the **Create namespace** dialog, do the following steps:

- Enter a **name for the namespace**. The system immediately checks to see if the name is available. For a list of rules for naming namespaces, see [Create Namespace REST API](#).
- Select the pricing tier (Basic, Standard, or Premium) for the namespace. If you want to use [topics and subscriptions](#), choose either Standard or Premium. Topics/subscriptions are not supported in the Basic pricing tier.
- If you selected the **Premium** pricing tier, follow these steps:
  - Specify the number of **messaging units**. The premium tier provides resource isolation at the CPU and memory level so that each workload runs in isolation. This resource container is called a messaging unit. A premium namespace has at least one messaging unit. You can select 1, 2, or 4 messaging units for each Service Bus Premium namespace. For more information, see [Service Bus Premium Messaging](#).
  - Specify whether you want to make the namespace **zone redundant**. The zone redundancy provides enhanced availability by spreading replicas across availability zones within one region at no additional cost. For more information, see [Availability zones in Azure](#).
- For **Subscription**, choose an Azure subscription in which to create the namespace.
- For **Resource group**, choose an existing resource group in which the namespace will live, or create a new one.

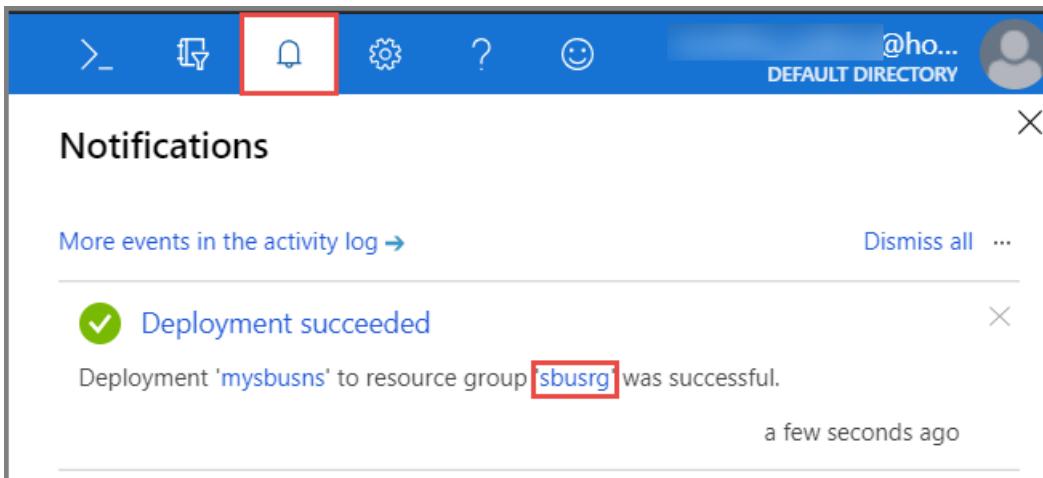
- f. For **Location**, choose the region in which your namespace should be hosted.
- g. Select **Create**. The system now creates your namespace and enables it. You might have to wait several minutes as the system provisions resources for your account.

The screenshot shows the 'Create namespace' dialog box. At the top, there's a 'Service Bus' icon and a title 'Create namespace'. Below the title, there are several input fields with validation messages:

- Name**: 'mysbusns' is entered, followed by a green checkmark and '.servicebus.windows.net'.
- Pricing tier**: 'Standard' is selected.
- Subscription**: 'Visual Studio Ultimate with MSDN' is selected.
- Resource group**: '(New) sbusrg' is selected, with a link to 'Create new' below it.
- Location**: 'West US' is selected.

At the bottom right of the dialog is a large blue 'Create' button.

4. Confirm that the service bus namespace is deployed successfully. To see the notifications, select the **bell icon (Alerts)** on the toolbar. Select the **name of the resource group** in the notification as shown in the image. You see the resource group that contains the service bus namespace.



5. On the **Resource group** page for your resource group, select your **service bus namespace**.

6. You see the home page for your service bus namespace.

## Get the connection string

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 rules with more constrained rights for regular senders and receivers. To copy the primary and secondary keys for your namespace, 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**.

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

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

## Create a topic using the Azure portal

- On the **Service Bus Namespace** page, select **Topics** on the left menu.
- Select **+ Topic** on the toolbar.
- Enter a **name** for the topic. Leave the other options with their default values.
- Select **Create**.

**Create topic**

Service Bus

Name: mytopic

Max topic size: 1 GB

Message time to live: 14 Days, 0 Hours, 0 Minutes, 0 Seconds

Enable duplicate detection:

Enable partitioning:

Create

## Create subscriptions to the topic

1. Select the **topic** that you created in the previous section.

NAME	STATUS	MAX SIZE	SUBSCRIPTION COUNT	ENABLE PARTITIONING
mytopic	Active	16 GB	0	true

2. On the **Service Bus Topic** page, select **Subscriptions** from the left menu, and then select **+ Subscription** on the toolbar.

The screenshot shows the Azure portal interface for managing Service Bus resources. On the left, there's a sidebar with various navigation links like Overview, Diagnose and solve problems, Settings, Metrics (preview), Properties, Locks, Automation script, and Entities. Under Entities, the 'Subscriptions' link is highlighted with a red box. The main content area is titled 'mytopic - Subscriptions' and shows a table with one row: 'no subscriptions yet'. At the top right of this area, there's a red box around the '+ Subscription' button. A search bar at the top says 'Search (Ctrl+I)'. The URL in the address bar is 'Dashboard > spsbrg > spsbusnamespace - Topics > mytopic - Subscriptions'.

3. On the **Create subscription** page, enter **S1** for **name** for the subscription, and then select **Create**.

The screenshot shows the 'Create subscription' dialog box. It has a title 'Create subscription' and a subtitle 'mytopic'. The form contains the following fields:

- Name:** mysubscription (highlighted with a red box)
- Message time to live (default):** 14 days
- Lock duration:** 30 seconds
- max delivery count:** 10
- Optional settings (checkboxes):**
  - Move expired messages to the dead-letter subqueue
  - Move messages that cause filter evaluation exceptions to the dead-letter subqueue
  - Enable sessions

4. Repeat the previous step twice to create subscriptions named **S2** and **S3**.

#### NOTE

You can manage Service Bus resources with [Service Bus Explorer](#). The Service Bus Explorer allows users to connect to a Service Bus namespace and administer messaging entities in an easy manner. The tool provides advanced features like import/export functionality or the ability to test topic, queues, subscriptions, relay services, notification hubs and events hubs.

## Next steps

To learn how to send messages to a topic and receive those messages via a subscription, see the following article:  
select the programming language in the TOC.

[Publish and subscribe for messages](#)

# Quickstart: Create a Service Bus namespace with topic and subscription using an Azure Resource Manager template

1/17/2020 • 2 minutes 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, 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 do 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. Define a parameter for those values that vary based on the project you're deploying or based on the environment you're 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": {}
                    }
                ]
            }
        ]
    }
]

```

For JSON syntax and properties, see [namespaces](#), [topics](#), and [subscriptions](#).

## Commands to run deployment

To deploy the resources to Azure, you must be signed 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:

- [Manage Azure resources by using Azure PowerShell](#)
- [Manage Azure resources by using Azure CLI.](#)

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/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-and-
subscription/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](#)

# Get started with Service Bus topics

11/27/2019 • 8 minutes to read • [Edit Online](#)

This tutorial covers the following steps:

1. Write a .NET Core console application to send a set of messages to the topic.
2. Write a .NET Core console application to receive those messages from the subscription.

## Prerequisites

1. An Azure subscription. To complete this tutorial, you need an Azure account. You can activate your [Visual Studio or MSDN subscriber benefits](#) or sign-up for a [free account](#).
2. Follow steps in the [Quickstart: Use the Azure portal to create a Service Bus topic and subscriptions to the topic](#) to do the following tasks:
  - a. Create a Service Bus **namespace**.
  - b. Get the **connection string**.
  - c. Create a **topic** in the namespace.
  - d. Create **one subscription** to the topic in the namespace.
3. [Visual Studio 2017 Update 3 \(version 15.3, 26730.01\)](#) or later.
4. [.NET Core SDK](#), version 2.0 or later.

## Send messages to the topic

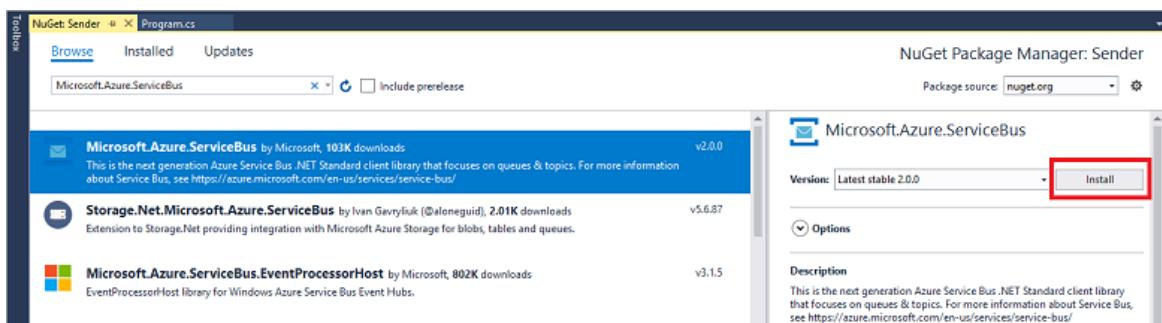
To send messages to the topic, write a C# console application using Visual Studio.

### 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 `Main()` method with the following `async Main` method that sends messages asynchronously using the `SendMessagesAsync` method that you will add in the next step.

```
public static async Task Main(string[] args)
{
    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();
}
```

4. Directly after the `Main` 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}");
    }
}
```

5. 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;

        public static async Task Main(string[] args)
        {
            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();
        }

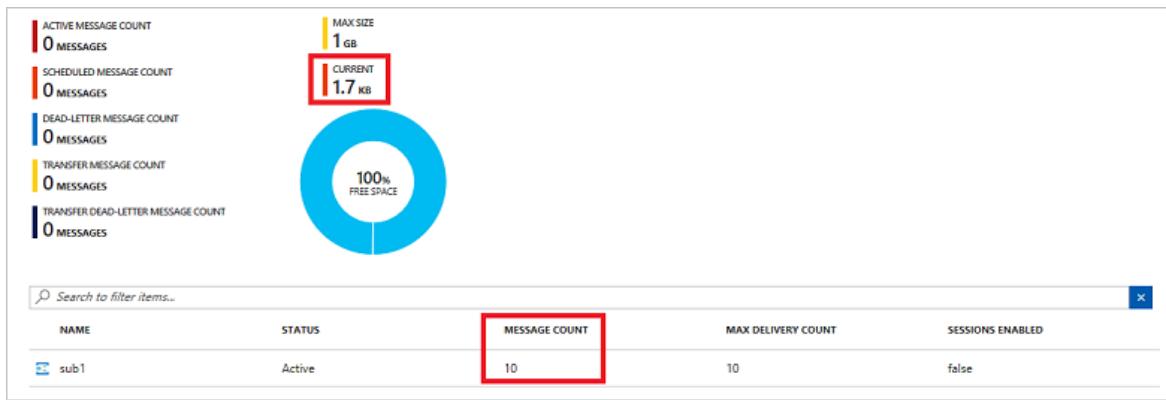
        static async Task SendMessagesAsync(int numberOfMessagesToSend)
        {
            try
            {
                for (var i = 0; i < numberOfMessagesToSend; i++)
                {
                    // Create a new message to send to the topic
                    string messageBody = $"Message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



## Receive messages from the subscription

To receive the messages you 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 `Main()` method with the following `async Main` method. It calls the `RegisterOnMessageHandlerAndReceiveMessages()` method that you will add in the next step.

```
public static async Task Main(string[] args)
{
    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();
}
```

4. Directly after the `Main()` 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);
}
```

5. 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.
}
```

6. 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;
}
```

7. 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;

        public static async Task Main(string[] args)
        {
            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

```

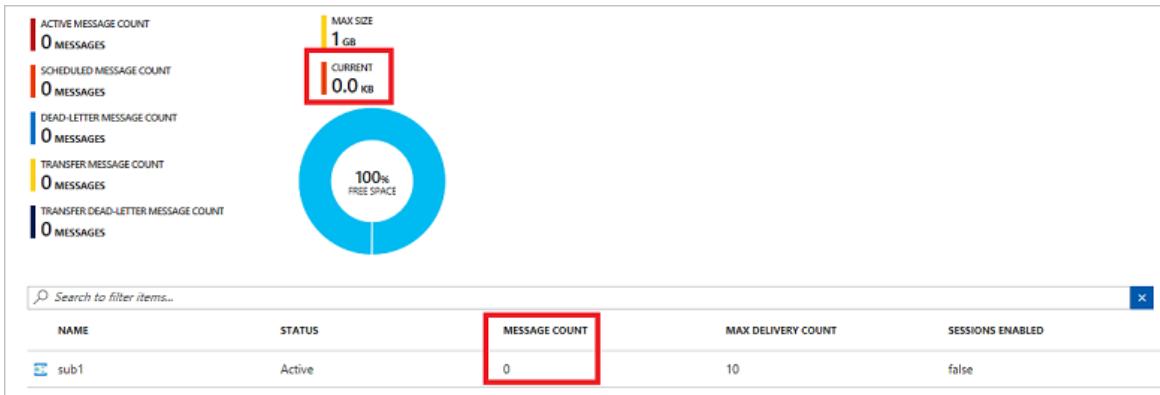
```

        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;
    }
}

```

- 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.

#### NOTE

You can manage Service Bus resources with [Service Bus Explorer](#). The Service Bus Explorer allows users to connect to a Service Bus namespace and administer messaging entities in an easy manner. The tool provides advanced features like import/export functionality or the ability to test topic, queues, subscriptions, relay services, notification hubs and events hubs.

## Next steps

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

# Quickstart: Use Service Bus topics and subscriptions with Java

1/24/2020 • 8 minutes to read • [Edit Online](#)

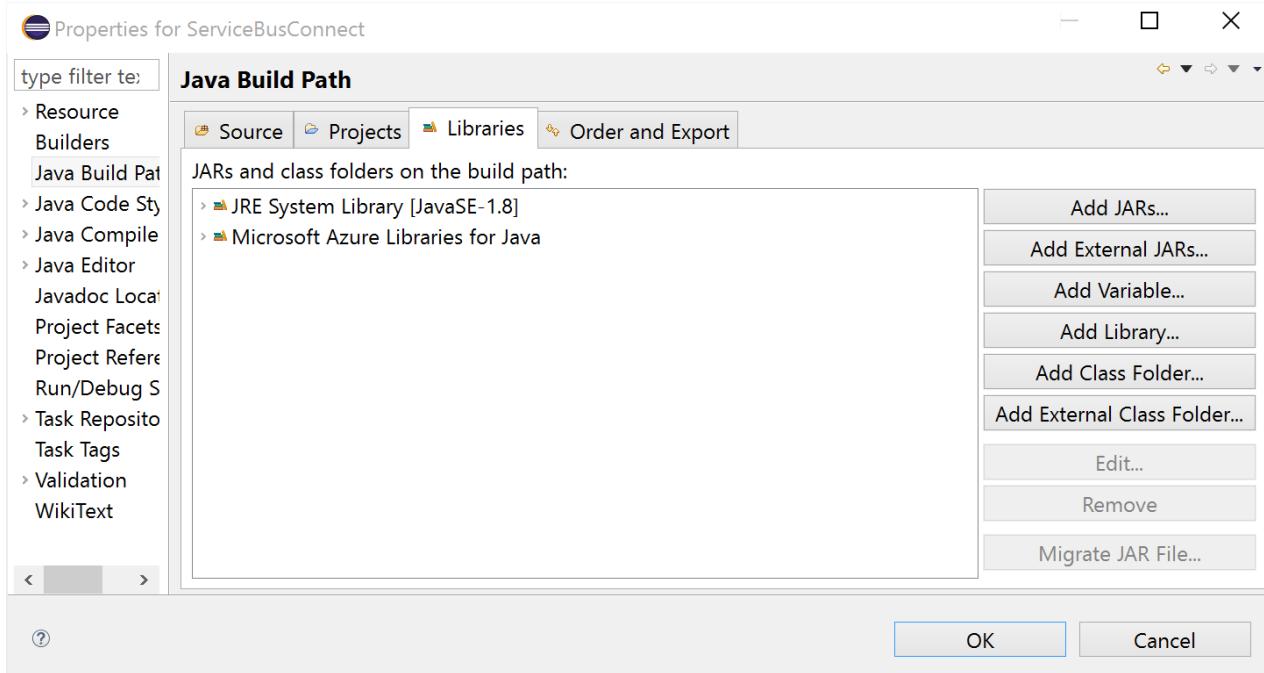
In this quickstart, you write Java code to send messages to an Azure Service Bus topic and then receive messages from subscriptions to that topic.

## Prerequisites

1. An Azure subscription. To complete this tutorial, you need an Azure account. You can activate your [Visual Studio or MSDN subscriber benefits](#) or sign-up for a [free account](#).
2. Follow steps in the [Quickstart: Use the Azure portal to create a Service Bus topic and subscriptions to the topic](#) to do the following tasks:
  - a. Create a Service Bus **namespace**.
  - b. Get the **connection string**.
  - c. Create a **topic** in the namespace.
  - d. Create **three subscriptions** to the topic in the namespace.
3. [Azure SDK for Java](#).

## 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:



You also need to add the following JARs to the Java Build Path:

- gson-2.6.2.jar
- commons-cli-1.4.jar
- proton-j-0.21.0.jar

Add a class with a **Main** method, and then add the following `import` statements at the top of the Java file:

```
import com.google.gson.reflect.TypeToken;
import com.microsoft.azure.servicebus.*;
import com.microsoft.azure.servicebus.primitives.ConnectionStringBuilder;
import com.google.gson.Gson;
import static java.nio.charset.StandardCharsets.*;
import java.time.Duration;
import java.util.*;
import java.util.concurrent.*;
import java.util.function.Function;
import org.apache.commons.cli.*;
import org.apache.commons.cli.DefaultParser;
```

## Send messages to a topic

Update the **main** method to create a **TopicClient** object, and invoke a helper method that asynchronously sends sample messages to the Service Bus topic.

### NOTE

- Replace `<NameOfServiceBusNamespace>` with the name of your Service Bus namespace.
- Replace `<AccessKey>` with the access key for your namespace.

```

public class MyServiceBusTopicClient {

    static final Gson GSON = new Gson();

    public static void main(String[] args) throws Exception, ServiceBusException {
        // TODO Auto-generated method stub

        TopicClient sendClient;
        String connectionString =
"Endpoint=sb://<NameOfServiceBusNamespace>.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=<AccessKey>";
        sendClient = new TopicClient(new ConnectionStringBuilder(connectionString, "BasicTopic"));
        sendMessagesAsync(sendClient).thenRunAsync(() -> sendClient.closeAsync());
    }

    static CompletableFuture<Void> sendMessagesAsync(TopicClient sendClient) {
        List<HashMap<String, String>> data =
            GSON.fromJson(
                "[" +
                    "{"'name' = 'Einstein', 'firstName' = 'Albert'}," +
                    {"'name' = 'Heisenberg', 'firstName' = 'Werner'}," +
                    {"'name' = 'Curie', 'firstName' = 'Marie'}," +
                    {"'name' = 'Hawking', 'firstName' = 'Steven'}," +
                    {"'name' = 'Newton', 'firstName' = 'Isaac'}," +
                    {"'name' = 'Bohr', 'firstName' = 'Niels'}," +
                    {"'name' = 'Faraday', 'firstName' = 'Michael'}," +
                    {"'name' = 'Galilei', 'firstName' = 'Galileo'}," +
                    {"'name' = 'Kepler', 'firstName' = 'Johannes'}," +
                    {"'name' = 'Kopernikus', 'firstName' = 'Nikolaus'}" +
                "]",
                new TypeToken<List<HashMap<String, String>>>() {
                    .getType());
            }

        List<CompletableFuture> tasks = new ArrayList<>();
        for (int i = 0; i < data.size(); i++) {
            final String messageId = Integer.toString(i);
            Message message = new Message(GSON.toJson(data.get(i), Map.class).getBytes(UTF_8));
            message.setContentType("application/json");
            message.setLabel("Scientist");
            message.setMessageId(messageId);
            message.setTimeToLive(Duration.ofMinutes(2));
            System.out.printf("Message sending: Id = %s\n", message.getMessageId());
            tasks.add(
                sendClient.sendAsync(message).thenRunAsync(() -> {
                    System.out.printf("\tMessage acknowledged: Id = %s\n", message.getMessageId());
                }));
        }
        return CompletableFuture.allOf(tasks.toArray(new CompletableFuture<?>[tasks.size()]));
    }
}

```

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

Update the **main** method to create three **SubscriptionClient** objects for three subscriptions, and invoke a helper method that asynchronously receives messages from the Service Bus topic. The sample code assumes that you created a topic named **BasicTopic** and three subscriptions named **Subscription1**, **Subscription2**, and **Subscription3**. If you used different names for them, update the code before testing it.

```

public class MyServiceBusTopicClient {

    static final Gson GSON = new Gson();

    public static void main(String[] args) throws Exception, ServiceBusException {
        SubscriptionClient subscription1Client = new SubscriptionClient(new
ConnectionStringBuilder(connectionString, "BasicTopic/subscriptions/Subscription1"), ReceiveMode.PEEKLOCK);
        SubscriptionClient subscription2Client = new SubscriptionClient(new
ConnectionStringBuilder(connectionString, "BasicTopic/subscriptions/Subscription2"), ReceiveMode.PEEKLOCK);
        SubscriptionClient subscription3Client = new SubscriptionClient(new
ConnectionStringBuilder(connectionString, "BasicTopic/subscriptions/Subscription3"), ReceiveMode.PEEKLOCK);

        registerMessageHandlerOnClient(subscription1Client);
        registerMessageHandlerOnClient(subscription2Client);
        registerMessageHandlerOnClient(subscription3Client);
    }

    static void registerMessageHandlerOnClient(SubscriptionClient receiveClient) throws Exception {

        // register the RegisterMessageHandler callback
        IMessageHandler messageHandler = new IMessageHandler() {
            // callback invoked when the message handler loop has obtained a message
            public CompletableFuture<Void> onMessageAsync(IMessage message) {
                // receives message is passed to callback
                if (message.getLabel() != null &&
                    message.getContentType() != null &&
                    message.getLabel().contentEquals("Scientist") &&
                    message.getContentType().contentEquals("application/json")) {

                    byte[] body = message.getBody();
                    Map scientist = GSON.fromJson(new String(body, UTF_8), Map.class);

                    System.out.printf(
                        "\n\t\t\tMessage received: \n\t\t\t\tMessageId = %s,
\n\t\t\t\tSequenceNumber = %s, \n\t\t\t\tEnqueuedTimeUtc = %s," +
                        "\n\t\t\t\tExpiresAtUtc = %s, \n\t\t\t\tContentType = \"%s\",
\n\t\t\tContent: [ firstName = %s, name = %s ]\n",
                        receiveClient.getEntityPath(),
                        message.getMessageId(),
                        message.getSequenceNumber(),
                        message.getEnqueuedTimeUtc(),
                        message.getExpiresAtUtc(),
                        message.getContentType(),
                        scientist != null ? scientist.get("firstName") : "",
                        scientist != null ? scientist.get("name") : "");
                }
                return receiveClient.completeAsync(message.getLockToken());
            }
        };

        public void notifyException(Throwable throwable, ExceptionPhase exceptionPhase) {
            System.out.printf(exceptionPhase + "-" + throwable.getMessage());
        }
    };

    receiveClient.registerMessageHandler(
        messageHandler,
        // callback invoked when the message handler has an exception to report
        // 1 concurrent call, messages are auto-completed, auto-renew duration
        new MessageHandlerOptions(1, false, Duration.ofMinutes(1)));
    }
}

```

# Run the program

Run the program to see the output similar to the following output:

```
Message sending: Id = 0
Message sending: Id = 1
Message sending: Id = 2
Message sending: Id = 3
Message sending: Id = 4
Message sending: Id = 5
Message sending: Id = 6
Message sending: Id = 7
Message sending: Id = 8
Message sending: Id = 9
Message acknowledged: Id = 0
Message acknowledged: Id = 9
Message acknowledged: Id = 7
Message acknowledged: Id = 8
Message acknowledged: Id = 5
Message acknowledged: Id = 6
Message acknowledged: Id = 3
Message acknowledged: Id = 2
Message acknowledged: Id = 4
Message acknowledged: Id = 1

BasicTopic/subscriptions/Subscription1 Message received:
 messageId = 0,
 sequenceNumber = 11,
 enqueuedTimeUtc = 2018-10-29T18:58:12.442Z,
 expiresAtUtc = 2018-10-29T19:00:12.442Z,
 contentType = "application/json",
 content: [ firstName = Albert, name = Einstein ]

BasicTopic/subscriptions/Subscription2 Message received:
 messageId = 0,
 sequenceNumber = 11,
 enqueuedTimeUtc = 2018-10-29T18:58:12.442Z,
 expiresAtUtc = 2018-10-29T19:00:12.442Z,
 contentType = "application/json",
 content: [ firstName = Albert, name = Einstein ]

BasicTopic/subscriptions/Subscription1 Message received:
 messageId = 9,
 sequenceNumber = 12,
 enqueuedTimeUtc = 2018-10-29T18:58:12.520Z,
 expiresAtUtc = 2018-10-29T19:00:12.520Z,
 contentType = "application/json",
 content: [ firstName = Nikolaus, name = Kopernikus ]

BasicTopic/subscriptions/Subscription1 Message received:
 messageId = 8,
 sequenceNumber = 13,
 enqueuedTimeUtc = 2018-10-29T18:58:12.520Z,
 expiresAtUtc = 2018-10-29T19:00:12.520Z,
 contentType = "application/json",
 content: [ firstName = Johannes, name = Kepler ]

BasicTopic/subscriptions/Subscription3 Message received:
 messageId = 0,
 sequenceNumber = 11,
 enqueuedTimeUtc = 2018-10-29T18:58:12.442Z,
 expiresAtUtc = 2018-10-29T19:00:12.442Z,
 contentType = "application/json",
 content: [ firstName = Albert, name = Einstein ]

BasicTopic/subscriptions/Subscription2 Message received:
 messageId = 9,
```

```
SequenceNumber = 12,
EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,
ExpiresAtUtc = 2018-10-29T19:00:12.520Z,
ContentType = "application/json",
Content: [ firstName = Nikolaus, name = Kopernikus ]

BasicTopic/subscriptions/Subscription1 Message received:
MessageId = 7,
SequenceNumber = 14,
EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,
ExpiresAtUtc = 2018-10-29T19:00:12.520Z,
ContentType = "application/json",
Content: [ firstName = Galileo, name = Galilei ]

BasicTopic/subscriptions/Subscription3 Message received:
MessageId = 9,
SequenceNumber = 12,
EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,
ExpiresAtUtc = 2018-10-29T19:00:12.520Z,
ContentType = "application/json",
Content: [ firstName = Nikolaus, name = Kopernikus ]

BasicTopic/subscriptions/Subscription2 Message received:
MessageId = 8,
SequenceNumber = 13,
EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,
ExpiresAtUtc = 2018-10-29T19:00:12.520Z,
ContentType = "application/json",
Content: [ firstName = Johannes, name = Kepler ]

BasicTopic/subscriptions/Subscription1 Message received:
MessageId = 6,
SequenceNumber = 15,
EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,
ExpiresAtUtc = 2018-10-29T19:00:12.520Z,
ContentType = "application/json",
Content: [ firstName = Michael, name = Faraday ]

BasicTopic/subscriptions/Subscription3 Message received:
MessageId = 8,
SequenceNumber = 13,
EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,
ExpiresAtUtc = 2018-10-29T19:00:12.520Z,
ContentType = "application/json",
Content: [ firstName = Johannes, name = Kepler ]

BasicTopic/subscriptions/Subscription2 Message received:
MessageId = 7,
SequenceNumber = 14,
EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,
ExpiresAtUtc = 2018-10-29T19:00:12.520Z,
ContentType = "application/json",
Content: [ firstName = Galileo, name = Galilei ]

BasicTopic/subscriptions/Subscription1 Message received:
MessageId = 5,
SequenceNumber = 16,
EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,
ExpiresAtUtc = 2018-10-29T19:00:12.520Z,
ContentType = "application/json",
Content: [ firstName = Niels, name = Bohr ]

BasicTopic/subscriptions/Subscription3 Message received:
MessageId = 7,
SequenceNumber = 14,
EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,
ExpiresAtUtc = 2018-10-29T19:00:12.520Z,
ContentType = "application/json",
Content: [ firstName = Galileo, name = Galilei ]
```

```
BasicTopic/subscriptions/Subscription2 Message received:  
  MessageId = 6,  
  SequenceNumber = 15,  
  EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,  
  ExpiresAtUtc = 2018-10-29T19:00:12.520Z,  
  ContentType = "application/json",  
  Content: [ firstName = Michael, name = Faraday ]  
  
BasicTopic/subscriptions/Subscription1 Message received:  
  MessageId = 4,  
  SequenceNumber = 17,  
  EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,  
  ExpiresAtUtc = 2018-10-29T19:00:12.520Z,  
  ContentType = "application/json",  
  Content: [ firstName = Isaac, name = Newton ]  
  
BasicTopic/subscriptions/Subscription3 Message received:  
  MessageId = 6,  
  SequenceNumber = 15,  
  EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,  
  ExpiresAtUtc = 2018-10-29T19:00:12.520Z,  
  ContentType = "application/json",  
  Content: [ firstName = Michael, name = Faraday ]  
  
BasicTopic/subscriptions/Subscription2 Message received:  
  MessageId = 5,  
  SequenceNumber = 16,  
  EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,  
  ExpiresAtUtc = 2018-10-29T19:00:12.520Z,  
  ContentType = "application/json",  
  Content: [ firstName = Niels, name = Bohr ]  
  
BasicTopic/subscriptions/Subscription1 Message received:  
  MessageId = 3,  
  SequenceNumber = 18,  
  EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,  
  ExpiresAtUtc = 2018-10-29T19:00:12.520Z,  
  ContentType = "application/json",  
  Content: [ firstName = Steven, name = Hawking ]  
  
BasicTopic/subscriptions/Subscription3 Message received:  
  MessageId = 5,  
  SequenceNumber = 16,  
  EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,  
  ExpiresAtUtc = 2018-10-29T19:00:12.520Z,  
  ContentType = "application/json",  
  Content: [ firstName = Niels, name = Bohr ]  
  
BasicTopic/subscriptions/Subscription2 Message received:  
  MessageId = 4,  
  SequenceNumber = 17,  
  EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,  
  ExpiresAtUtc = 2018-10-29T19:00:12.520Z,  
  ContentType = "application/json",  
  Content: [ firstName = Isaac, name = Newton ]  
  
BasicTopic/subscriptions/Subscription1 Message received:  
  MessageId = 2,  
  SequenceNumber = 19,  
  EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,  
  ExpiresAtUtc = 2018-10-29T19:00:12.520Z,  
  ContentType = "application/json",  
  Content: [ firstName = Marie, name = Curie ]  
  
BasicTopic/subscriptions/Subscription3 Message received:  
  MessageId = 4,  
  SequenceNumber = 17,  
  EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,
```

```
ExpiresAtUtc = 2018-10-29T19:00:12.520Z,
ContentType = "application/json",
Content: [ firstName = Isaac, name = Newton ]

BasicTopic/subscriptions/Subscription2 Message received:
MessageId = 3,
SequenceNumber = 18,
EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,
ExpiresAtUtc = 2018-10-29T19:00:12.520Z,
ContentType = "application/json",
Content: [ firstName = Steven, name = Hawking ]

BasicTopic/subscriptions/Subscription1 Message received:
MessageId = 1,
SequenceNumber = 20,
EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,
ExpiresAtUtc = 2018-10-29T19:00:12.520Z,
ContentType = "application/json",
Content: [ firstName = Werner, name = Heisenberg ]

BasicTopic/subscriptions/Subscription2 Message received:
MessageId = 2,
SequenceNumber = 19,
EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,
ExpiresAtUtc = 2018-10-29T19:00:12.520Z,
ContentType = "application/json",
Content: [ firstName = Marie, name = Curie ]

BasicTopic/subscriptions/Subscription3 Message received:
MessageId = 3,
SequenceNumber = 18,
EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,
ExpiresAtUtc = 2018-10-29T19:00:12.520Z,
ContentType = "application/json",
Content: [ firstName = Steven, name = Hawking ]

BasicTopic/subscriptions/Subscription3 Message received:
MessageId = 2,
SequenceNumber = 19,
EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,
ExpiresAtUtc = 2018-10-29T19:00:12.520Z,
ContentType = "application/json",
Content: [ firstName = Marie, name = Curie ]

BasicTopic/subscriptions/Subscription2 Message received:
MessageId = 1,
SequenceNumber = 20,
EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,
ExpiresAtUtc = 2018-10-29T19:00:12.520Z,
ContentType = "application/json",
Content: [ firstName = Werner, name = Heisenberg ]

BasicTopic/subscriptions/Subscription3 Message received:
MessageId = 1,
SequenceNumber = 20,
EnqueuedTimeUtc = 2018-10-29T18:58:12.520Z,
ExpiresAtUtc = 2018-10-29T19:00:12.520Z,
ContentType = "application/json",
Content: [ firstName = Werner, name = Heisenberg ]
```

**NOTE**

You can manage Service Bus resources with [Service Bus Explorer](#). The Service Bus Explorer allows users to connect to a Service Bus namespace and administer messaging entities in an easy manner. The tool provides advanced features like import/export functionality or the ability to test topic, queues, subscriptions, relay services, notification hubs and events hubs.

## Next steps

For more information, see [Service Bus queues, topics, and subscriptions](#).

# Quickstart: How to use Service Bus topics and subscriptions with Node.js and the azure/service-bus package

1/17/2020 • 5 minutes to read • [Edit Online](#)

In this tutorial, you learn how to write a Node.js program to send messages to a Service Bus topic and receive messages from a Service Bus subscription using the new [@azure/service-bus](#) package. This package uses the faster [AMQP 1.0 protocol](#) whereas the older [azure-sb](#) package used [Service Bus REST run-time APIs](#). The samples are written in JavaScript.

## Prerequisites

- An Azure subscription. To complete this tutorial, you need an Azure account. You can activate your [MSDN subscriber benefits](#) or sign up for a [free account](#).
- If you don't have a topic and subscription to work with, follow steps in the [Use Azure portal to create a Service Bus topics and subscriptions](#) article to create them. Note down the connection string for your Service Bus instance and the names of the topic and subscription you created. We'll use these values in the samples.

### NOTE

- This tutorial works with samples that you can copy and run using [Nodejs](#). 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](#).
- The new [@azure/service-bus](#) package does not support creation of topics and subscriptions yet. Please use the [@azure/arm-servicebus](#) package if you want to programmatically create them.

## Use Node Package Manager (NPM) to install the package

To install the npm package for Service Bus, open a command prompt that has `npm` in its path, change the directory to the folder where you want to have your samples and then run this command.

```
npm install @azure/service-bus
```

## Send messages to a topic

Interacting with a Service Bus topic starts with instantiating the [ServiceBusClient](#) class and using it to instantiate the [TopicClient](#) class. Once you have the topic client, you can create a sender and use either [send](#) or [sendBatch](#) method on it to send messages.

1. Open your favorite editor, such as [Visual Studio Code](#)
2. Create a file called `send.js` and paste the below code into it. This code will send 10 messages to your topic.

```

const { ServiceBusClient } = require("@azure/service-bus");

// Define connection string and related Service Bus entity names here
const connectionString = "";
const topicName = "";

async function main(){
    const sbClient = ServiceBusClient.createFromConnectionString(connectionString);
    const topicClient = sbClient.createTopicClient(topicName);
    const sender = topicClient.createSender();

    try {
        for (let i = 0; i < 10; i++) {
            const message= {
                body: `Hello world! ${i}`,
                label: `test`,
                userProperties: {
                    myCustomPropertyName: `my custom property value ${i}`
                }
            };
            console.log(`Sending message: ${message.body}`);
            await sender.send(message);
        }

        await topicClient.close();
    } finally {
        await sbClient.close();
    }
}

main().catch((err) => {
    console.log("Error occurred: ", err);
});

```

3. Enter the connection string and name of your topic in the above code.

4. Then run the command `node send.js` in a command prompt to execute this file.

Congratulations! You just sent messages to a Service Bus queue.

Messages have some standard properties like `label` and `messageId` that you can set when sending. If you want to set any custom properties, use the `userProperties`, which is a json object that can hold key-value pairs of your custom data.

Service Bus topics support a maximum message size of 256 KB in the [Standard tier](#) and 1 MB in the [Premium tier](#). There's no limit on the number of messages held in a topic, but there's 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. For more information about quotas, see [Service Bus quotas](#).

## Receive messages from a subscription

Interacting with a Service Bus subscription starts with instantiating the `ServiceBusClient` class and using it to instantiate the `SubscriptionClient` class. Once you have the subscription client, you can create a receiver and use either `receiveMessages` or `registerMessageHandler` method on it to receive messages.

1. Open your favorite editor, such as [Visual Studio Code](#)
2. Create a file called `receive.js` and paste the below code into it. This code will attempt to receive 10 messages from your subscription. The actual count you receive depends on the number of messages in the subscription and network latency.

```

const { ServiceBusClient, ReceiveMode } = require("@azure/service-bus");

// Define connection string and related Service Bus entity names here
const connectionString = "";
const topicName = "";
const subscriptionName = "";

async function main(){
    const sbClient = ServiceBusClient.createFromConnectionString(connectionString);
    const subscriptionClient = sbClient.createSubscriptionClient(topicName, subscriptionName);
    const receiver = subscriptionClient.createReceiver(ReceiveMode.receiveAndDelete);

    try {
        const messages = await receiver.receiveMessages(10);
        console.log("Received messages:");
        console.log(messages.map(message => message.body));

        await subscriptionClient.close();
    } finally {
        await sbClient.close();
    }
}

main().catch((err) => {
    console.log("Error occurred: ", err);
});

```

3. Enter the connection string and names of your topic and subscription in the above code.

4. Then run the command `node receiveMessages.js` in a command prompt to execute this file.

Congratulations! You just received messages from a Service Bus subscription.

The `createReceiver` method takes in a `ReceiveMode` which is an enum with values `ReceiveAndDelete` and `PeekLock`. Remember to [settle your messages](#) if you use the `PeekLock` mode by using any of `complete()`, `abandon()`, `defer()`, or `deadletter()` methods on the message.

## Subscription filters and actions

Service Bus supports [filters and actions on subscriptions](#), which allows you to filter the incoming messages to a subscription and to edit their properties.

Once you have an instance of a `SubscriptionClient` you can use the below methods on it to get, add and remove rules on the subscription to control the filters and actions.

- `getRules`
- `addRule`
- `removeRule`

Every subscription has a default rule that uses the true filter to allow all incoming messages. When you add a new rule, remember to remove the default filter in order for the filter in your new rule to work. If a subscription has no rules, then it will receive no messages.

### NOTE

You can manage Service Bus resources with [Service Bus Explorer](#). The Service Bus Explorer allows users to connect to a Service Bus namespace and administer messaging entities in an easy manner. The tool provides advanced features like import/export functionality or the ability to test topic, queues, subscriptions, relay services, notification hubs and events hubs.

## Next Steps

To learn more, see the following resources.

- [Queues, topics, and subscriptions](#)
- Checkout other [Nodejs samples for Service Bus on GitHub](#)
- [Node.js Developer Center](#)

# Quickstart: How to Use Service Bus topics and subscriptions with Node.js and the azure-sb package

1/17/2020 • 11 minutes to read • [Edit Online](#)

In this tutorial, you learn how to create Node.js applications to send messages to a Service Bus topic and receive messages from a Service Bus subscription using the [azure-sb](#) package. The samples are written in JavaScript and use the Node.js [Azure module](#) which internally uses the [azure-sb](#) package.

The [azure-sb](#) package uses [Service Bus run-time APIs](#). You can get a faster experience using the new [@azure/service-bus](#) package which uses the faster [AMQP 1.0 protocol](#). To learn more about the new package, see [How to use Service Bus topics and subscriptions with Node.js](#) and [@azure/service-bus package](#), otherwise continue reading to see how to use the [azure](#) package.

The scenarios covered here include:

- Creating topics and subscriptions
- Creating subscription filters
- Sending messages to a topic
- Receiving messages from a subscription
- Deleting topics and subscriptions

For more information about topics and subscriptions, see [Next steps](#) section.

## Prerequisites

- An Azure subscription. To complete this tutorial, you need an Azure account. You can activate your [Visual Studio or MSDN subscriber benefits](#) or sign up for a [free account](#).
- Follow steps in the [Quickstart: Use the Azure portal to create a Service Bus topic and subscriptions to the topic](#) to create a Service Bus **namespace** and get the **connection string**.

### NOTE

You will create a **topic** and a **subscription** to the topic by using **Node.js** in this quickstart.

## 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. Open a command-line interface such as **PowerShell** (Windows), **Terminal** (Mac), or **Bash** (Unix).
2. Navigate to the folder where you created your sample application.

3. Type **npm install azure** in the command window, which should result in the following output:

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

4. 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 [Set environment variables](#).

## 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();
```

If you call `createTopicIfNotExists` on the **ServiceBusService** object, the specified topic is returned (if it exists), or a new topic with the specified name is 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 `createTopicIfNotExists` 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 with a time to live of one 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`, and passes 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 invoke `finalCallback` 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 code 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

By default, subscriptions are persistent 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 exists by using the `getSubscription` method.

You can have the subscriptions automatically deleted by setting the [AutoDeleteOnIdle](#) property.

### Create a subscription with the default (**MatchAll**) filter

The **MatchAll** filter is the default filter used when a subscription is created. When you use the **MatchAll** filter, all messages published to the topic are placed in the subscription's virtual queue. The following example creates a subscription named AllMessages and uses the default **MatchAll** filter.

```

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 is 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 are populated by default values.

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 property determines 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 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.

## 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 set the optional parameter `isPeekLock` to **true** to read (peek) and lock the message without deleting it from the subscription.

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 when 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 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.

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 missed messages. When Service Bus receives a request, it finds the next message to consume, locks it to prevent other consumers from receiving it, and returns it to the application. After the application processes the message (or stores it reliably for future processing), it completes the second stage of the receive process by calling **deleteMessage** method, and passes the message to delete as a parameter. The **deleteMessage** method marks the message as consumed and removes 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 method causes Service Bus to unlock the message within the subscription and make it available to be received again. In this instance, either by the same consuming application or by another consuming application.

There is also a timeout associated with a message locked within the subscription. 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 the application crashes after processing the message but before the `deleteMessage` method is called, the message is redelivered to the application when it restarts. This behavior 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 you should add logic to your application to handle duplicate message delivery. You can use the **MessageId** property of the message, which remains constant across delivery attempts.

## Delete topics and subscriptions

Topics and subscriptions are persistent unless the `autoDeleteOnIdle` property is set, 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 also deletes 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);
  }
});
```

#### NOTE

You can manage Service Bus resources with [Service Bus Explorer](#). The Service Bus Explorer allows users to connect to a Service Bus namespace and administer messaging entities in an easy manner. The tool provides advanced features like import/export functionality or the ability to test topic, queues, subscriptions, relay services, notification hubs and events hubs.

## 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.

# Quickstart: How to use Service Bus topics and subscriptions with PHP

1/24/2020 • 10 minutes 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
- Deleting topics and subscriptions

## Prerequisites

1. An Azure subscription. To complete this tutorial, you need an Azure account. You can activate your [Visual Studio or MSDN subscriber benefits](#) or sign up for a [free account](#).
2. Follow steps in the [Quickstart: Use the Azure portal to create a Service Bus topic and subscriptions to the topic](#) to create a Service Bus **namespace** and get the **connection string**.

### NOTE

You will create a **topic** and a **subscription** to the topic by using **PHP** in this quickstart.

## 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/windowsazure": "*"  
    }  
}
```

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
```

## 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

If no filter is specified when a new subscription is created, the **MatchAll** filter (default) 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 `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);

```

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 five test messages to the `mytopic` topic previously created. The `setProperty` method is used to add a custom property (`MessageNumber`) to each message. 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 when a failure occurs. 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, then when the application restarts and begins consuming messages again, it has 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). It 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 type of processing 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. It 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. 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");
```

#### NOTE

You can manage Service Bus resources with [Service Bus Explorer](#). The Service Bus Explorer allows users to connect to a Service Bus namespace and administer messaging entities in an easy manner. The tool provides advanced features like import/export functionality or the ability to test topic, queues, subscriptions, relay services, notification hubs and events hubs.

## Next steps

For more information, see [Queues, topics, and subscriptions](#).

# Quickstart: Use Service Bus topics and subscriptions with Python

1/27/2020 • 7 minutes to read • [Edit Online](#)

This article describes how to use Python with Azure Service Bus topics and subscriptions. The samples use the [Azure Python SDK](#) package to:

- Create topics and subscriptions to topics
- Create subscription filters and rules
- Send messages to topics
- Receive messages from subscriptions
- Delete topics and subscriptions

## Prerequisites

- An Azure subscription. You can activate your [Visual Studio or MSDN subscriber benefits](#) or sign up for a [free account](#).
- A Service Bus namespace, created by following the steps at [Quickstart: Use the Azure portal to create a Service Bus topic and subscriptions](#). Copy the namespace name, shared access key name, and primary key value from the **Shared access policies** screen to use later in this quickstart.
- Python 3.4x or above, with the [Azure Python SDK](#) package installed. For more information, see the [Python Installation Guide](#).

## Create a ServiceBusService object

A **ServiceBusService** object lets you work with topics and subscriptions to topics. To programmatically access Service Bus, add the following line near the top of your Python file:

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

Add the following code to create a **ServiceBusService** object. Replace `<namespace>`, `<sharedaccesskeyname>`, and `<sharedaccesskeyvalue>` with your Service Bus namespace name, Shared Access Signature (SAS) key name, and primary key value. You can find these values under **Shared access policies** in your Service Bus namespace in the [Azure portal](#).

```
bus_service = ServiceBusService(  
    service_namespace='<namespace>',  
    shared_access_key_name='<sharedaccesskeyname>',  
    shared_access_key_value='<sharedaccesskeyvalue>')
```

## Create a topic

The following code uses the `create_topic` method to create a Service Bus topic called `mytopic`, with default settings:

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

You can use topic options to override default topic settings, such as message time to live (TTL) or maximum topic size. The following example creates a topic named `mytopic` with a maximum topic size of 5 GB and default message TTL 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

You also use the **ServiceBusService** object to create subscriptions to topics. A subscription can have a filter to restrict the message set delivered to its virtual queue. If you don't specify a filter, new subscriptions use the default **MatchAll** filter, which places all messages published to the topic into the subscription's virtual queue. The following example creates a subscription to `mytopic` named `AllMessages` that uses the **MatchAll** filter:

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

### Use filters with subscriptions

Use the `create_rule` method of the **ServiceBusService** object to filter the messages that appear in a subscription. You can specify rules when you create the subscription, or add rules to existing subscriptions.

The most flexible type of filter is a **SqlFilter**, which uses a subset of SQL-92. SQL filters operate based on the properties of messages published to the topic. For more information about the expressions you can use with a SQL filter, see the [SqlFilter.SqlExpression](#) syntax.

Because the **MatchAll** default filter applies automatically to all new subscriptions, you must remove it from subscriptions you want to filter, or **MatchAll** will override any other filters you specify. You can remove the default rule by using the `delete_rule` method of the **ServiceBusService** object.

The following example creates a subscription to `mytopic` named `HighMessages`, with a **SqlFilter** rule named `HighMessageFilter`. The `HighMessageFilter` rule selects only messages with a custom `messageposition` property greater than 3:

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

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

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

The following example creates a subscription to `mytopic` named `LowMessages`, with a **SqlFilter** rule named `LowMessageFilter`. The `LowMessageFilter` rule selects only messages with a `messageposition` property less than or equal to 3:

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

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

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

With `AllMessages`, `HighMessages`, and `LowMessages` all in effect, messages sent to `mytopic` are always delivered to receivers of the `AllMessages` subscription. Messages are also selectively delivered to the `HighMessages` or `LowMessages` subscription, depending on the message's `messageposition` property value.

## Send messages to a topic

Applications use the `send_topic_message` method of the **ServiceBusService** object to send messages to a Service Bus topic.

The following example sends five test messages to the `mytopic` topic. The custom `messageposition` property value depends on the iteration of the loop, and determines which subscriptions receive the messages.

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

### Message size limits and quotas

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's no limit on the number of messages a topic can hold, but there's a cap on the total size of the messages the topic holds. You can define topic size at creation time, with an upper limit of 5 GB.

For more information about quotas, see [Service Bus quotas](#).

## Receive messages from a subscription

Applications use the `receive_subscription_message` method on the **ServiceBusService** object to receive messages from a subscription. The following example receives messages from the `LowMessages` subscription and deletes them as they're read:

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

The optional `peek_lock` parameter of `receive_subscription_message` determines whether Service Bus deletes messages from the subscription as they're read. The default mode for message receiving is *PeekLock*, or `peek_lock` set to **True**, which reads (peeks) and locks messages without deleting them from the subscription. Each message must then be explicitly completed to remove it from the subscription.

To delete messages from the subscription as they're read, you can set the `peek_lock` parameter to **False**, as in the preceding example. Deleting messages as part of the receive operation is the simplest model, and works fine if the application can tolerate missing messages if there's a failure. To understand this behavior, consider a scenario in which the application issues a receive request and then crashes before processing it. If the message was deleted on being received, when the application restarts and begins consuming messages again, it has missed the message it received before the crash.

If your application can't tolerate missed messages, the receive becomes a two-stage operation. PeekLock finds the next message to be consumed, locks it to prevent other consumers from receiving it, and returns it to the application. After processing or storing the message, the application completes the second stage of the receive process by calling the `complete` method on the **Message** object. The `complete` method marks the message as being consumed and removes it from the subscription.

The following example demonstrates a peek lock scenario:

```
msg = bus_service.receive_subscription_message('mytopic', 'LowMessages', peek_lock=True)
if msg.body is not None:
    print(msg.body)
    msg.complete()
```

## 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 can't process a message for some reason, it can call the `unlock` method on the **Message** object. Service Bus unlocks the message within the subscription and makes it available to be received again, either by the same or another consuming application.

There's also a timeout for messages locked within the subscription. If an application fails to process a message before the lock timeout expires, for example if the application crashes, Service Bus unlocks the message automatically and makes it available to be received again.

If an application crashes after processing a message but before calling the `complete` method, the message will be redelivered to the application when it restarts. This behavior is often called *At-least-once Processing*. Each message is processed at least once, but in certain situations the same message may be redelivered. If your scenario can't tolerate duplicate processing, you can use the **MessageId** property of the message, which remains constant across delivery attempts, to handle duplicate message delivery.

## Delete topics and subscriptions

To delete topics and subscriptions, use the [Azure portal](#) or the `delete_topic` method. The following code deletes the topic named `mytopic`:

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

Deleting a topic deletes all subscriptions to the topic. You can also delete subscriptions independently. The following code deletes the subscription named `HighMessages` from the `mytopic` topic:

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

By default, topics and subscriptions are persistent, and exist until you delete them. To automatically delete subscriptions after a certain time period elapses, you can set the `auto_delete_on_idle` parameter on the subscription.

### TIP

You can manage Service Bus resources with [Service Bus Explorer](#). Service Bus Explorer lets you connect to a Service Bus namespace and easily administer messaging entities. The tool provides advanced features like import/export functionality and the ability to test topics, queues, subscriptions, relay services, notification hubs, and event hubs.

## Next steps

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

- [Queues, topics, and subscriptions](#)
- [SqlFilter.SqlExpression reference](#)

# Quickstart: How to use Service Bus topics and subscriptions with Ruby

11/6/2019 • 8 minutes 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
- Deleting topics and subscriptions

## Prerequisites

1. An Azure subscription. To complete this tutorial, you need an Azure account. You can activate your [Visual Studio or MSDN subscriber benefits](#) or sign-up for a [free account](#).
2. Follow steps in the [Quickstart: Use the Azure portal to create a Service Bus topic and subscriptions to the topic](#) to create a Service Bus **namespace** and get the **connection string**.

### NOTE

You will create a **topic** and a **subscription** to the topic by using **Ruby** in this quickstart.

## 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 any errors.

```

azure_service_bus_service = Azure::ServiceBus::ServiceBusService.new(sb_host, { signer: signer})
begin
  topic = azure_service_bus_service.create_topic("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.

By default, subscriptions are persistent. They 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.

You can have the subscriptions automatically deleted by setting the [AutoDeleteOnIdle property](#).

### Create a subscription with the default (**MatchAll**) filter

If no filter is specified when a new subscription is created, the **MatchAll** filter (default) 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 "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** overrides 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 are populated by default values.

The following example demonstrates how to send five test messages to `test-topic`. The `message_number` custom property value of each message varies on the iteration of the loop (it 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 marks 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 when a failure occurs. 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.

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. It 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 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. It 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 logic is often achieved using the `message_id` property of the message, which remains constant across delivery attempts.

## Delete topics and subscriptions

Topics and subscriptions are persistent unless the [AutoDeleteOnIdle](#) property is set. They can be deleted either through the [Azure portal](#) or programmatically. The following example 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")
```

### NOTE

You can manage Service Bus resources with [Service Bus Explorer](#). The Service Bus Explorer allows users to connect to a Service Bus namespace and administer messaging entities in an easy manner. The tool provides advanced features like import/export functionality or the ability to test topic, queues, subscriptions, relay services, notification hubs and events hubs.

## 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.

# Tutorial: Update inventory using PowerShell and topics/subscriptions

5/21/2019 • 7 minutes to read • [Edit Online](#)

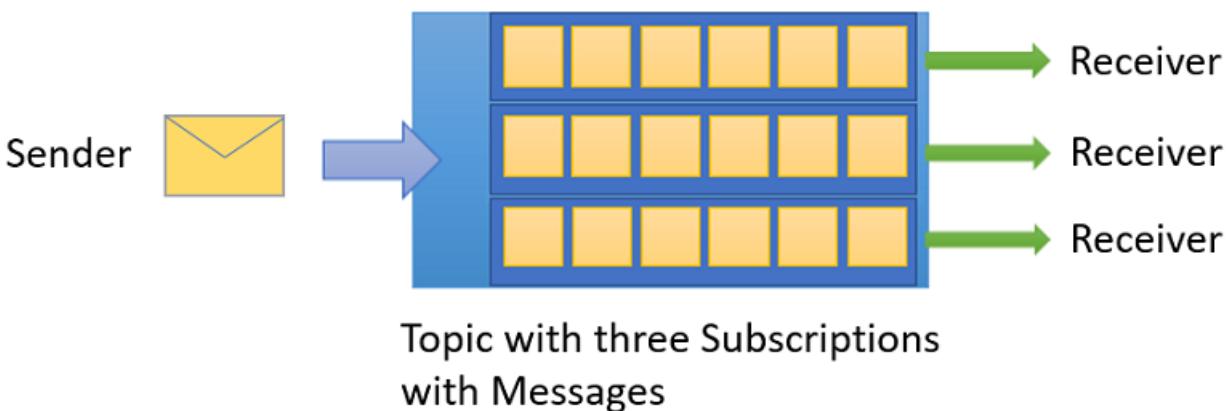
Microsoft Azure Service Bus is a multi-tenant cloud messaging service that sends information between applications and services. Asynchronous operations give you flexible, brokered messaging, along with structured first-in, first-out (FIFO) messaging, and publish/subscribe capabilities.

This tutorial shows how to send and receive messages to and from a Service Bus queue, using PowerShell to create a messaging namespace and a queue within that namespace, and to obtain the authorization credentials on that namespace. The procedure then shows how to send and receive messages from this queue using the [.NET Standard library](#).

In this tutorial, you learn how to:

- Create a Service Bus topic and one or more subscriptions to that topic using Azure PowerShell
- Add topic filters using PowerShell
- Create two messages with different content
- Send the messages and verify they arrived in the expected subscriptions
- Receive messages from the subscriptions

An example of this scenario is an inventory assortment update for multiple retail stores. In this scenario, each store, or set of stores, gets messages intended for them to update their assortments. This tutorial shows how to implement this scenario using subscriptions and filters. First, you create a topic with 3 subscriptions, add some rules and filters, and then send and receive messages from the topic and subscriptions.



If you do not have an Azure subscription, create a [free account](#) before you begin.

## NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

## Prerequisites

To complete this tutorial, make sure you have installed:

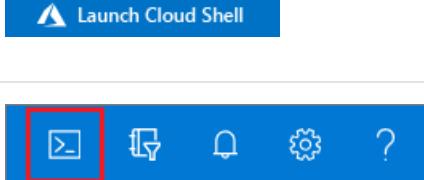
1. [Visual Studio 2017 Update 3 \(version 15.3, 26730.01\)](#) or later.
2. [.NET Core SDK](#), version 2.0 or later.

This tutorial requires that you run the latest version of Azure PowerShell. If you need to install or upgrade, see [Install and Configure Azure PowerShell](#).

## Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

OPTION	EXAMPLE/LINK
Select <b>Try It</b> in the upper-right corner of a code block. Selecting <b>Try It</b> doesn't automatically copy the code to Cloud Shell.	
Go to <a href="https://shell.azure.com">https://shell.azure.com</a> , or select the <b>Launch Cloud Shell</b> button to open Cloud Shell in your browser.	
Select the <b>Cloud Shell</b> button on the menu bar at the upper right in the <a href="#">Azure portal</a> .	

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

## Sign in to Azure

Issue the following commands to sign in to Azure. These steps are not necessary if you're running PowerShell commands in Cloud Shell:

1. Install the Service Bus PowerShell module:

```
Install-Module Az.ServiceBus
```

2. Run the following command to sign in to Azure:

```
Login-AzAccount
```

3. Set the current subscription context, or see the currently active subscription:

```
Select-AzSubscription -SubscriptionName "MyAzureSubName"  
Get-AzContext
```

## Provision resources

After signing in to Azure, issue the following commands to provision Service Bus resources. Be sure to replace all placeholders with the appropriate values:

```
# Create a resource group  
New-AzResourceGroup -Name my-resourcegroup -Location westus2  
  
# Create a Messaging namespace  
New-AzServiceBusNamespace -ResourceGroupName my-resourcegroup -NamespaceName namespace-name -Location westus2  
  
# Create a queue  
New-AzServiceBusQueue -ResourceGroupName my-resourcegroup -NamespaceName namespace-name -Name queue-name -  
EnablePartitioning $False  
  
# Get primary connection string (required in next step)  
Get-AzServiceBusKey -ResourceGroupName my-resourcegroup -Namespace namespace-name -Name  
RootManageSharedAccessKey
```

After the `Get-AzServiceBusKey` cmdlet runs, copy and paste the connection string and the queue name you selected to a temporary location, such as Notepad. You will need it in the next step.

## Send and receive messages

After the namespace and queue are created, and you have the necessary credentials, you are ready to send and receive messages. You can examine the code in [this GitHub sample folder](#).

To run the code, do the following:

1. Clone the [Service Bus GitHub repository](#) by issuing the following command:

```
git clone https://github.com/Azure/azure-service-bus.git
```

2. Open a PowerShell prompt.

3. Navigate to the sample folder

```
azure-service-bus\samples\DotNet\GettingStarted\BasicSendReceiveQuickStart\BasicSendReceiveQuickStart
```

4. If you have not done so already, obtain the connection string using the following PowerShell cmdlet. Be sure to replace `my-resourcegroup` and `namespace-name` with your specific values:

```
Get-AzServiceBusKey -ResourceGroupName my-resourcegroup -Namespace namespace-name -Name  
RootManageSharedAccessKey
```

5. At the PowerShell prompt, type the following command:

```
dotnet build
```

6. Navigate to the `\bin\Debug\netcoreapp2.0` folder.

7. Type the following command to run the program. Be sure to replace `myConnectionString` with the value you previously obtained, and `myQueueName` with the name of the queue you created:

```
dotnet BasicSendReceiveQuickStart.dll -ConnectionString "myConnectionString" -QueueName "myQueueName"
```

8. Observe 10 messages being sent to the queue, and subsequently received from the queue:

```
Administrator: Command Prompt
QueueName: queue1
=====
Press any key to exit after receiving all the messages.
=====
Sending message: Message 0
Sending message: Message 1
Received message: SequenceNumber:33776997221278728 Body:Message 0
Sending message: Message 2
Sending message: Message 3
Sending message: Message 4
Sending message: Message 5
Sending message: Message 6
Received message: SequenceNumber:37436171906517253 Body:Message 1
Sending message: Message 7
Sending message: Message 8
Received message: SequenceNumber:41658296595177092 Body:Message 2
Sending message: Message 9
Received message: SequenceNumber:46161896187547588 Body:Message 3
Received message: SequenceNumber:50946970848628741 Body:Message 4
Received message: SequenceNumber:55169095470288580 Body:Message 5
Received message: SequenceNumber:59954170044369733 Body:Message 6
Received message: SequenceNumber:65020719632161543 Body:Message 7
Received message: SequenceNumber:69524319291532039 Body:Message 8
Received message: SequenceNumber:1970324903974599 Body:Message 9
```

## Clean up resources

Run the following command to remove the resource group, namespace, and all related resources:

```
Remove-AzResourceGroup -Name my-resourcegroup
```

## Understand the sample code

This section contains more details about what the sample code does.

### Get connection string and queue

The connection string and queue name are passed to the `Main()` method as command line arguments. `Main()` declares two string variables to hold these values:

```

static void Main(string[] args)
{
    string ServiceBusConnectionString = "";
    string QueueName = "";

    for (int i = 0; i < args.Length; i++)
    {
        var p = new Program();
        if (args[i] == "-ConnectionString")
        {
            Console.WriteLine($"ConnectionString: {args[i+1]}");
            ServiceBusConnectionString = args[i + 1];
        }
        else if(args[i] == "-QueueName")
        {
            Console.WriteLine($"QueueName: {args[i+1]}");
            QueueName = args[i + 1];
        }
    }

    if (ServiceBusConnectionString != "" && QueueName != "")
        MainAsync(ServiceBusConnectionString, QueueName).GetAwaiter().GetResult();
    else
    {
        Console.WriteLine("Specify -ConnectionString and -QueueName to execute the example.");
        Console.ReadKey();
    }
}

```

The `Main()` method then starts the asynchronous message loop, `MainAsync()`.

## Message loop

The `MainAsync()` method creates a queue client with the command line arguments, calls a receiving message handler named `RegisterOnMessageHandlerAndReceiveMessages()`, and sends the set of messages:

```

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

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

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

    // Send Messages
    await SendMessagesAsync(numberOfMessages);

    Console.ReadKey();

    await queueClient.CloseAsync();
}

```

The `RegisterOnMessageHandlerAndReceiveMessages()` method sets a few message handler options, then calls the queue client's `RegisterMessageHandler()` method, which starts the receiving:

```

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);
}

```

## Send messages

The message creation and send operations occur in the `SendMessagesAsync()` method:

```

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}");
    }
}

```

## Process messages

The `ProcessMessagesAsync()` method acknowledges, processes, and completes the receipt of the 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.
    await queueClient.CompleteAsync(message.SystemProperties.LockToken);
}

```

**NOTE**

You can manage Service Bus resources with [Service Bus Explorer](#). The Service Bus Explorer allows users to connect to a Service Bus namespace and administer messaging entities in an easy manner. The tool provides advanced features like import/export functionality or the ability to test topic, queues, subscriptions, relay services, notification hubs and events hubs.

## Next steps

In this tutorial, you provisioned resources using Azure PowerShell, then sent and received messages from a Service Bus topic and its subscriptions. You learned how to:

- Create a Service Bus topic and one or more subscriptions to that topic using the Azure portal
- Add topic filters using .NET code
- Create two messages with different content
- Send the messages and verify they arrived in the expected subscriptions
- Receive messages from the subscriptions

For more examples of sending and receiving messages, get started with the [Service Bus samples on GitHub](#).

Advance to the next tutorial to learn more about using the publish/subscribe capabilities of Service Bus.

[Update inventory using PowerShell and topics/subscriptions](#)

# Tutorial: Update inventory using Azure portal and topics/subscriptions

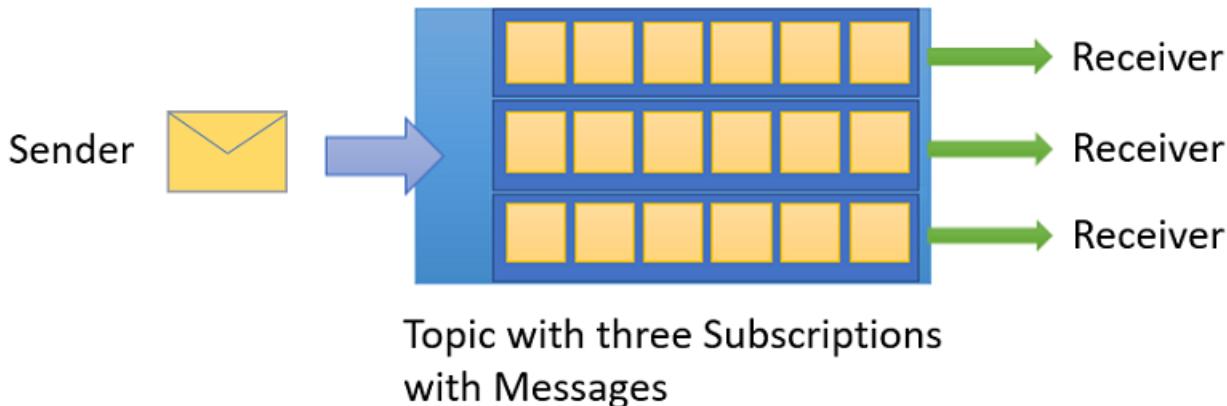
1/21/2020 • 12 minutes to read • [Edit Online](#)

Microsoft Azure Service Bus is a multi-tenant cloud messaging service that sends information between applications and services. Asynchronous operations give you flexible, brokered messaging, along with structured first-in, first-out (FIFO) messaging, and publish/subscribe capabilities. This tutorial shows how to use Service Bus topics and subscriptions in a retail inventory scenario, with publish/subscribe channels using the Azure portal and .NET.

In this tutorial, you learn how to:

- Create a Service Bus topic and one or more subscriptions to that topic using the Azure portal
- Add topic filters using .NET code
- Create two messages with different content
- Send the messages and verify they arrived in the expected subscriptions
- Receive messages from the subscriptions

An example of this scenario is an inventory assortment update for multiple retail stores. In this scenario, each store, or set of stores, gets messages intended for them to update their assortments. This tutorial shows how to implement this scenario using subscriptions and filters. First, you create a topic with 3 subscriptions, add some rules and filters, and then send and receive messages from the topic and subscriptions.



If you don't have an Azure subscription, you can create a [free account](#) before you begin.

## Prerequisites

To complete this tutorial, make sure you have installed:

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

## Service Bus topics and subscriptions

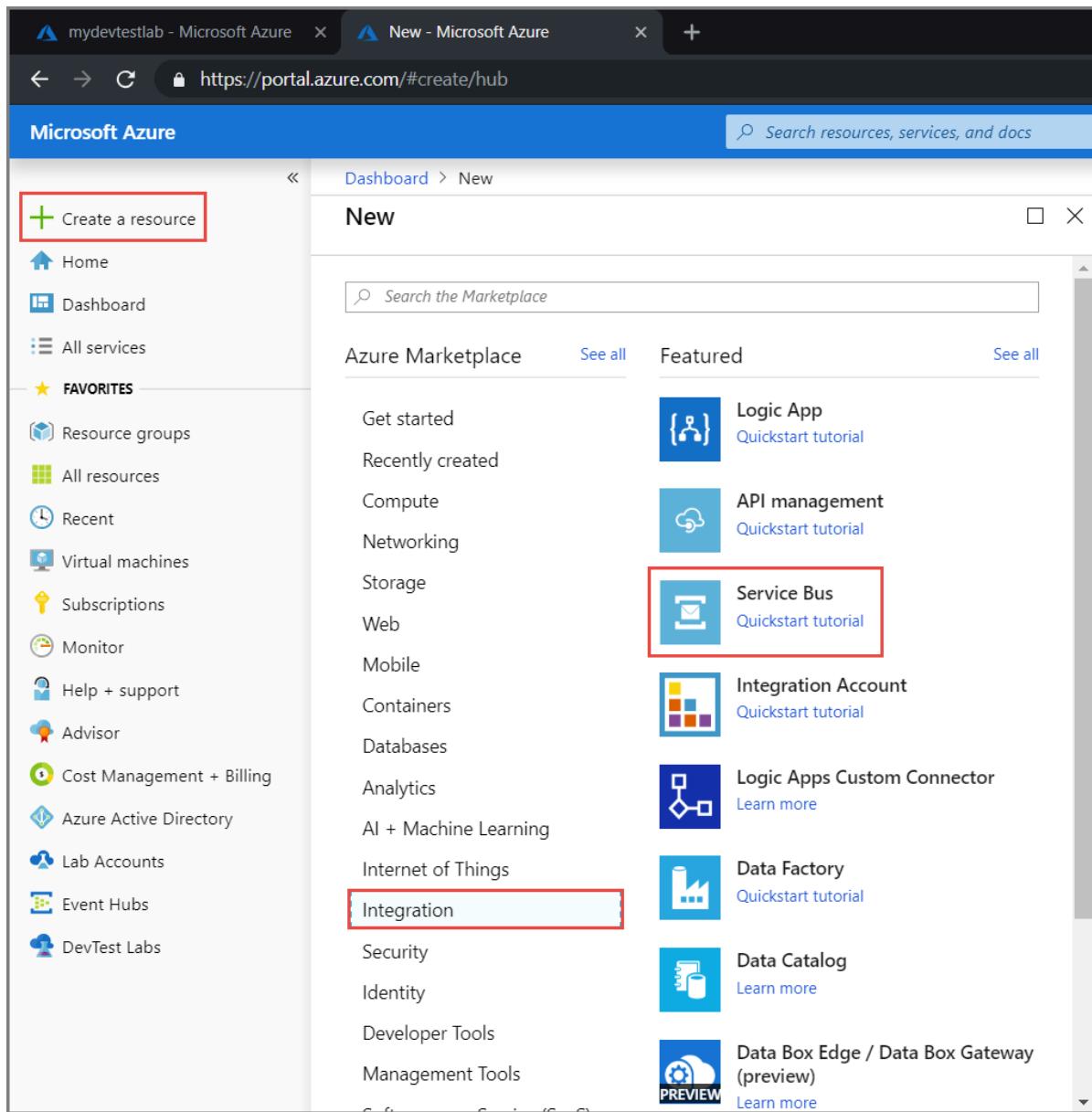
Each [subscription to a topic](#) can receive a copy of each message. Topics are fully protocol and semantically compatible with Service Bus queues. Service Bus topics support a wide array of selection rules with filter conditions, with optional actions that set or modify message properties. Each time a rule matches, it produces a message. To learn more about rules, filters, and actions, follow this [link](#).

# 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. Sign in to the [Azure portal](#)
2. In the left navigation pane of the portal, select **+ Create a resource**, select **Integration**, and then select **Service Bus**.



3. In the **Create namespace** dialog, do the following steps:
  - a. Enter a **name for the namespace**. The system immediately checks to see if the name is available.  
For a list of rules for naming namespaces, see [Create Namespace REST API](#).
  - b. Select the pricing tier (Basic, Standard, or Premium) for the namespace. If you want to use [topics and subscriptions](#), choose either Standard or Premium. Topics/subscriptions are not supported in the Basic pricing tier.
  - c. If you selected the **Premium** pricing tier, follow these steps:
    - a. Specify the number of **messaging units**. The premium tier provides resource isolation at the

CPU and memory level so that each workload runs in isolation. This resource container is called a messaging unit. A premium namespace has at least one messaging unit. You can select 1, 2, or 4 messaging units for each Service Bus Premium namespace. For more information, see [Service Bus Premium Messaging](#).

- b. Specify whether you want to make the namespace **zone redundant**. The zone redundancy provides enhanced availability by spreading replicas across availability zones within one region at no additional cost. For more information, see [Availability zones in Azure](#).
- d. For **Subscription**, choose an Azure subscription in which to create the namespace.
- e. For **Resource group**, choose an existing resource group in which the namespace will live, or create a new one.
- f. For **Location**, choose the region in which your namespace should be hosted.
- g. Select **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.

The screenshot shows the 'Create namespace' wizard in the Azure portal. The top navigation bar includes 'Dashboard > New > Create namespace'. The main title is 'Create namespace' with a 'Service Bus' icon. The form fields are as follows:

- Name:** mybusns (highlighted with a green checkmark)
- Pricing tier:** Standard
- Subscription:** Visual Studio Ultimate with MSDN
- Resource group:** (New) sbusrg (with a 'Create new' link)
- Location:** West US

A large blue 'Create' button is at the bottom of the form.

4. Confirm that the service bus namespace is deployed successfully. To see the notifications, select the **bell icon (Alerts)** on the toolbar. Select the **name of the resource group** in the notification as shown in the image. You see the resource group that contains the service bus namespace.

The screenshot shows the Azure Notifications page. At the top, there are several icons: a left arrow, a refresh, a bell (which is highlighted with a red box), a gear, a question mark, and a smiley face. To the right, it says '@ho...' and 'DEFAULT DIRECTORY' with a user profile icon. Below the header, the word 'Notifications' is displayed. On the left, there's a link 'More events in the activity log →'. On the right, there are links 'Dismiss all' and '...'. A prominent message box contains a green checkmark icon and the text 'Deployment succeeded'. Below that, it says 'Deployment 'mysbusns' to resource group 'sbusrsg'' was successful. A timestamp 'a few seconds ago' is at the bottom of the message box.

5. On the **Resource group** page for your resource group, select your **service bus namespace**.

The screenshot shows the Azure Resource Group 'sbusrsg' dashboard. The left sidebar has sections for Overview, Activity log, Access control (IAM), Tags, Events, Settings (Quickstart, Resource costs, Deployments, Policies, Properties, Locks, Automation script), Monitoring (Insights (preview), Alerts, Metrics, Diagnostic settings), and a search bar. The main area shows deployment details: 'Subscription (change) : Visual Studio Ultimate with MSDN', 'Deployment ID : [redacted]', 'Tags (change) : Click here to add tags', and 'Deployments : 1 Succeeded'. Below this is a table with one item: 'NAME' (mysbusns), 'TYPE' (Service Bus Namespace), and 'LOCATION' (West US). The entire row for 'mysbusns' is highlighted with a red box.

6. You see the home page for your service bus namespace.

The screenshot shows the Azure Service Bus Namespace 'mysbusns' dashboard. The left sidebar includes sections for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Shared access policies, Scale, Geo-Recovery, Properties, Locks, Automation script), Entities (Queues, Topics), Monitoring (Alerts, Metrics, Diagnostic settings), Support + troubleshooting, Resource health, and New support request. The main area displays monitoring data: 'Incoming Requests (P. mybusns)' at 0, 'Successful Requests (P. mybusns)' at 0, 'Server Errors (P. mybusns)' at 0, 'User Errors (P. mybusns)' at 0, 'Throttled Requests (P. mybusns)' at 0, 'Incoming Messages (P. mybusns)' at 0, and 'Outgoing Messages (P. mybusns)' at 0. Below this are two tabs: 'Queues' (selected) and 'Topics'. A table lists queues: NAME (no queues yet), STATUS, MAX SIZE, and ENABLE PARTITIONING. The entire table row is highlighted with a red box.

Get the connection string

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 rules with more constrained rights for regular senders and receivers. To copy the primary and secondary keys for your namespace, 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 managing a Service Bus namespace named 'mysbusns'. The left sidebar lists various management options: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Shared access policies (which is selected and highlighted with a red box), Scale, Geo-Recovery, Properties, Locks, and Automation script. The main content area displays the 'Shared access policies' for the 'mysbusns - Shared access policies' screen. It includes a search bar, a 'POLICY' table with one row ('RootManageSharedAccessKey'), and a 'CLAIMS' table with one row ('Manage, Send, Listen'). A red box highlights the 'RootManageSharedAccessKey' policy row.

4. In the **Policy: RootManageSharedAccessKey** window, click the copy button next to **Primary Connection String**, 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, there are save, discard, delete, and more options. Below that, there are checkboxes for 'Manage', 'Send', and 'Listen', all of which are checked. Under 'Primary Key', there is a text input field containing '<Primary key>' with a copy icon. Under 'Secondary Key', there is a text input field containing '<Secondary key>' with a copy icon. Under 'Primary Connection String', there is a text input field containing 'Endpoint=sb://mysbusns.servicebus.windows.net/\$...' with a copy icon. Under 'Secondary Connection String', there is a text input field containing 'Endpoint=sb://mysbusns.servicebus.windows.net/\$...' with a copy icon. The 'copy' icons for the connection strings are highlighted with red boxes.

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

# Create a topic using the Azure portal

1. On the **Service Bus Namespace** page, select **Topics** on the left menu.
2. Select **+ Topic** on the toolbar.
3. Enter a **name** for the topic. Leave the other options with their default values.
4. Select **Create**.

The screenshot shows the Azure portal interface for creating a new topic in a Service Bus namespace. The left sidebar navigation bar is visible, with the 'Topics' item selected and highlighted with a red box. The main content area displays a 'Create topic' dialog box. The 'Name' field is populated with 'mytopic'. Other settings include a 'Max topic size' of 1 GB, a 'Message time to live' of 14 days, and checkboxes for 'Enable duplicate detection' (unchecked) and 'Enable partitioning' (checked). A large red box highlights the 'Create' button at the bottom right of the dialog.

## Create subscriptions to the topic

1. Select the **topic** that you created in the previous section.

The screenshot shows the Azure portal interface for a Service Bus Namespace. The left sidebar has a tree view with nodes like 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Settings' (with 'Shared access policies', 'Scale', 'Geo-Recovery', 'Properties', 'Locks', 'Automation script'), 'Entities' (with 'Queues' and 'Topics' selected), 'Monitoring' (with 'Alerts', 'Metrics', 'Diagnostic settings'), and 'Support + troubleshooting' (with 'Resource health'). The main content area is titled 'spbusnamespace - Topics' and shows a table with one row for 'mytopic'. The table columns are NAME, STATUS, MAX SIZE, SUBSCRIPTION COUNT, and ENABLE PARTITIONING. The row for 'mytopic' has a red border around it.

2. On the **Service Bus Topic** page, select **Subscriptions** from the left menu, and then select **+ Subscription** on the toolbar.

The screenshot shows the 'mytopic - Subscriptions' page. The left sidebar is identical to the previous screenshot. The main content area shows a table with a single row labeled 'no subscriptions yet'. The table has columns: NAME, STATUS, MAX DELIVERY COUNT, SESSIONS ENABLED, and MESSAGE COUNT. A red box highlights the '+ Subscription' button in the toolbar at the top of the main area.

3. On the **Create subscription** page, enter **S1** for **name** for the subscription, and then select **Create**.

**Create subscription**

mytopic

\* Name  
mysubscription ✓

\* Message time to live (default)  
14 days

\* Lock duration  
30 seconds

\* max delivery count  
10

Move expired messages to the dead-letter subqueue

Move messages that cause filter evaluation exceptions to the dead-letter subqueue

Enable sessions

**Create**

This screenshot shows the 'Create subscription' dialog in the Azure portal. It's a form-based interface with several input fields and checkboxes. The 'Name' field is filled with 'mysubscription'. The 'Message time to live' field has '14' in the input and 'days' in the dropdown. The 'Lock duration' field has '30' in the input and 'seconds' in the dropdown. The 'max delivery count' field has '10' in it. At the bottom, there are three checkboxes: 'Move expired messages to the dead-letter subqueue', 'Move messages that cause filter evaluation exceptions to the dead-letter subqueue', and 'Enable sessions'. A large blue 'Create' button is at the bottom right.

4. Repeat the previous step twice to create subscriptions named **S2** and **S3**.

## Create filter rules on subscriptions

After the namespace and topic/subscriptions are provisioned, and you have the necessary credentials, you are ready to create filter rules on the subscriptions, then send and receive messages. You can examine the code in [this GitHub sample folder](#).

### Send and receive messages

To run the code, do the following:

1. In a command prompt or PowerShell prompt, clone the [Service Bus GitHub repository](#) by issuing the following command:

```
git clone https://github.com/Azure/azure-service-bus.git
```

2. Navigate to the sample folder

```
azure-service-bus\samples\DotNet\GettingStarted\BasicSendReceiveTutorialwithFilters .
```

3. Obtain the connection string you copied to Notepad in the Obtain the management credentials section of this tutorial. You also need the name of the topic you created in the previous section.

4. At the command prompt, type the following command:

```
dotnet build
```

5. Navigate to the `BasicSendReceiveTutorialwithFilters\bin\Debug\netcoreapp2.0` folder.

6. Type the following command to run the program. Be sure to replace `myConnectionString` with the value you previously obtained, and `myTopicName` with the name of the topic you created:

```
dotnet BasicSendReceiveTutorialwithFilters.dll -ConnectionString "myConnectionString" -TopicName  
"myTopicName"
```

7. Follow the instructions in the console to select filter creation first. Part of creating filters is to remove the default filters. When you use PowerShell or CLI you don't need to remove the default filter, but if you do this in code, you must remove them. The console commands 1 and 3 help you manage the filters on the subscriptions you previously created:

- Execute 1: to remove the default filters.
- Execute 2: to add your own filters.
- Execute 3: to optionally remove your own filters. Note that this will not recreate the default filters.

```
Choose an action:  
[1] Remove the default filters which accept all messages. DO THIS ALWAYS FIRST.  
[2] Create your own filters. DO THIS SECOND.  
[3] Remove your own filters. OPTIONAL  
[4] Send messages.  
[5] Receive messages.  
  
Filters and actions for S1 have been created.  
Filters and actions for S2 have been created.  
Filters and actions for S3 have been created.  
All filters and actions have been created.
```

8. After filter creation, you can send messages. Press 4 and observe 10 messages being sent to the topic:

```
Administrator: Windows PowerShell  
  
Choose an action:  
[1] Remove the default filters which accept all messages. DO THIS ALWAYS FIRST.  
[2] Create your own filters. DO THIS SECOND.  
[3] Remove your own filters. OPTIONAL  
[4] Send messages.  
[5] Receive messages.  
  
Sent item to Store Store1. Price=1.4, Color=Yellow, Category=Vegetables  
Sent item to Store Store5. Price=2.3, Color=Red, Category=Vegetables  
Sent item to Store Store4. Price=3.2, Color=Orange, Category=Other  
Sent item to Store Store6. Price=5.1, Color=Red, Category=Other  
Sent item to Store Store8. Price=5.1, Color=Red, Category=Vegetables  
Sent item to Store Store9. Price=2.3, Color=Blue, Category=Meat  
Sent item to Store Store10. Price=4.1, Color=Red, Category=Beverage  
Sent item to Store Store3. Price=3.2, Color=Blue, Category=Vegetables  
Sent item to Store Store2. Price=3.2, Color=Yellow, Category=Other  
Sent item to Store Store7. Price=2.3, Color=Orange, Category=Other
```

9. Press 5 and observe the messages being received. If you did not get 10 messages back, press "m" to display the menu, then press 5 again.

```
Choose an action:  
[1] Remove the default filters which accept all messages. DO THIS ALWAYS FIRST.  
[2] Create your own filters. DO THIS SECOND.  
[3] Remove your own filters. OPTIONAL  
[4] Send messages.  
[5] Receive messages.
```

Receiving messages. Press any key to exit once all messages have been received.  
Alternatively press "M" to get to the menu

```
StoreId=Store5  
Item data: Price=2.3, Color=Red, Category=Vegetables  
StoreId=Store9  
Item data: Price=2.3, Color=Blue, Category=Meat  
StoreId=Store1  
Item data: Price=1.4, Color=Yellow, Category=Vegetables  
StoreId=Store7  
Item data: Price=2.3, Color=Orange, Category=Other  
StoreId=Store3  
Item data: Price=3.2, Color=Blue, Category=Vegetables  
StoreId=Store10  
Item data: Price=4.1, Color=Red, Category=Beverage  
StoreId=Store6  
Item data: Price=5.1, Color=Red, Category=Other  
StoreId=Store2  
Item data: Price=3.2, Color=Yellow, Category=Other  
StoreId=Store8  
Item data: Price=5.1, Color=Red, Category=Vegetables  
StoreId=Store4  
Item data: Price=3.2, Color=Orange, Category=Other
```

## Clean up resources

When no longer needed, delete the namespace and queue. To do so, select these resources on the portal and click **Delete**.

## Understand the sample code

This section contains more details about what the sample code does.

### Get connection string and topic

First, the code declares a set of variables, which drive the remaining execution of the program.

```

string ServiceBusConnectionString;
string TopicName;

static string[] Subscriptions = { "S1", "S2", "S3" };
static IDictionary<string, string[]> SubscriptionFilters = new Dictionary<string, string[]> {
    { "S1", new[] { "StoreId IN('Store1', 'Store2', 'Store3')", "StoreId = 'Store4'" } },
    { "S2", new[] { "sys.To IN ('Store5','Store6','Store7') OR StoreId = 'Store8'" } },
    { "S3", new[] { "sys.To NOT IN ('Store1','Store2','Store3','Store4','Store5','Store6','Store7','Store8')" } }
    OR StoreId NOT IN ('Store1','Store2','Store3','Store4','Store5','Store6','Store7','Store8') } }
};

// You can have only have one action per rule and this sample code supports only one action for the first
filter, which is used to create the first rule.
static IDictionary<string, string> SubscriptionAction = new Dictionary<string, string> {
    { "S1", "" },
    { "S2", "" },
    { "S3", "SET sys.Label = 'SalesEvent'" }
};

static string[] Store = { "Store1", "Store2", "Store3", "Store4", "Store5", "Store6", "Store7", "Store8",
"Store9", "Store10" };
static string SysField = "sys.To";
static string CustomField = "StoreId";
static int NrOfMessagesPerStore = 1; // Send at least 1.

```

The connection string and topic name are passed in via command line parameters as shown, and then are read in the `Main()` method:

```

static void Main(string[] args)
{
    string ServiceBusConnectionString = "";
    string TopicName = "";

    for (int i = 0; i < args.Length; i++)
    {
        if (args[i] == "-ConnectionString")
        {
            Console.WriteLine($"ConnectionString: {args[i + 1]}");
            ServiceBusConnectionString = args[i + 1]; // Alternatively enter your connection string here.
        }
        else if (args[i] == "-TopicName")
        {
            Console.WriteLine($"TopicName: {args[i + 1]}");
            TopicName = args[i + 1]; // Alternatively enter your queue name here.
        }
    }

    if (ServiceBusConnectionString != "" && TopicName != "")
    {
        Program P = StartProgram(ServiceBusConnectionString, TopicName);
        P.PresentMenu().GetAwaiter().GetResult();
    }
    else
    {
        Console.WriteLine("Specify -ConnectionString and -TopicName to execute the example.");
        Console.ReadKey();
    }
}

```

## Remove default filters

When you create a subscription, Service Bus creates a default filter per subscription. This filter enables receiving every message sent to the topic. If you want to use custom filters, you can remove the default filter, as shown in the following code:

```
private async Task RemoveDefaultFilters()
{
    Console.WriteLine($"Starting to remove default filters.");

    try
    {
        foreach (var subscription in Subscriptions)
        {
            SubscriptionClient s = new SubscriptionClient(ServiceBusConnectionString, TopicName,
subscription);
            await s.RemoveRuleAsync(RuleDescription.DefaultRuleName);
            Console.WriteLine($"Default filter for {subscription} has been removed.");
            await s.CloseAsync();
        }

        Console.WriteLine("All default Rules have been removed.\n");
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }

    await PresentMenu();
}
```

## Create filters

The following code adds the custom filters defined in this tutorial:

```

private async Task CreateCustomFilters()
{
    try
    {
        for (int i = 0; i < Subscriptions.Length; i++)
        {
            SubscriptionClient s = new SubscriptionClient(ServiceBusConnectionString, TopicName,
Subscriptions[i]);
            string[] filters = SubscriptionFilters[Subscriptions[i]];
            if (filters[0] != "")
            {
                int count = 0;
                foreach (var myFilter in filters)
                {
                    count++;

                    string action = SubscriptionAction[Subscriptions[i]];
                    if (action != "")
                    {
                        await s.AddRuleAsync(new RuleDescription
                        {
                            Filter = new SqlFilter(myFilter),
                            Action = new SqlRuleAction(action),
                            Name = $"MyRule{count}"
                        });
                    }
                    else
                    {
                        await s.AddRuleAsync(new RuleDescription
                        {
                            Filter = new SqlFilter(myFilter),
                            Name = $"MyRule{count}"
                        });
                    }
                }
            }

            Console.WriteLine($"Filters and actions for {Subscriptions[i]} have been created.");
        }

        Console.WriteLine("All filters and actions have been created.\n");
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }

    await PresentMenu();
}

```

### Remove your custom created filters

If you want to remove all filters on your subscription, the following code shows how to do that:

```
private async Task CleanUpCustomFilters()
{
    foreach (var subscription in Subscriptions)
    {
        try
        {
            SubscriptionClient s = new SubscriptionClient(ServiceBusConnectionString, TopicName,
subscription);
            IEnumerable<RuleDescription> rules = await s.GetRulesAsync();
            foreach (RuleDescription r in rules)
            {
                await s.RemoveRuleAsync(r.Name);
                Console.WriteLine($"Rule {r.Name} has been removed.");
            }
            await s.CloseAsync();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.ToString());
        }
    }
    Console.WriteLine("All default filters have been removed.\n");

    await PresentMenu();
}
```

## Send messages

Sending messages to a topic is similar to sending messages to a queue. This example shows how to send messages, using a task list and asynchronous processing:

```

public async Task SendMessages()
{
    try
    {
        TopicClient tc = new TopicClient(ServiceBusConnectionString, TopicName);

        var taskList = new List<Task>();
        for (int i = 0; i < Store.Length; i++)
        {
            taskList.Add(SendItems(tc, Store[i]));
        }

        await Task.WhenAll(taskList);
        await tc.CloseAsync();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }
    Console.WriteLine("\nAll messages sent.\n");
}

private async Task SendItems(TopicClient tc, string store)
{
    for (int i = 0; i < NrOfMessagesPerStore; i++)
    {
        Random r = new Random();
        Item item = new Item(r.Next(5), r.Next(5), r.Next(5));

        // Note the extension class which is serializing an deserializing messages
        Message message = item.AsMessage();
        message.To = store;
        message.UserProperties.Add("StoreId", store);
        message.UserProperties.Add("Price", item.getPrice().ToString());
        message.UserProperties.Add("Color", item.getColor());
        message.UserProperties.Add("Category", item.getItemCategory());

        await tc.SendAsync(message);
        Console.WriteLine($"Sent item to Store {store}. Price={item.getPrice()}, Color={item.getColor()},
Category={item.getItemCategory()}"); ;
    }
}

```

## Receive messages

Messages are again received via a task list, and the code uses batching. You can send and receive using batching, but this example only shows how to batch receive. In reality, you would not break out of the loop, but keep looping and set a higher timespan, such as one minute. The receive call to the broker is kept open for this amount of time and if messages arrive, they are returned immediately and a new receive call is issued. This concept is called *long polling*. Using the receive pump which you can see in the [quickstart](#), and in several other samples in the repository, is a more typical option.

```

public async Task Receive()
{
    var taskList = new List<Task>();
    for (var i = 0; i < Subscriptions.Length; i++)
    {
        taskList.Add(this.ReceiveMessages(Subscriptions[i]));
    }

    await Task.WhenAll(taskList);
}

private async Task ReceiveMessages(string subscription)
{
    var entityPath = EntityNameHelper.FormatSubscriptionPath(TopicName, subscription);
    var receiver = new MessageReceiver(ServiceBusConnectionString, entityPath, ReceiveMode.PeekLock,
    RetryPolicy.Default, 100);

    while (true)
    {
        try
        {
            IList<Message> messages = await receiver.ReceiveAsync(10, TimeSpan.FromSeconds(2));
            if (messages.Any())
            {
                foreach (var message in messages)
                {
                    lock (Console.Out)
                    {
                        Item item = message.As<Item>();
                        IDictionary<string, object> myUserProperties = message.UserProperties;
                        Console.WriteLine($"StoreId={myUserProperties["StoreId"]}");

                        if (message.Label != null)
                        {
                            Console.WriteLine($"Label={message.Label}");
                        }

                        Console.WriteLine(
                            $"Item data: Price={item.getPrice()}, Color={item.getColor()}, Category=
{item.getItemCategory()}");
                    }
                }

                await receiver.CompleteAsync(message.SystemProperties.LockToken);
            }
        }
        else
        {
            break;
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
    }
}

await receiver.CloseAsync();
}

```

#### **NOTE**

You can manage Service Bus resources with [Service Bus Explorer](#). The Service Bus Explorer allows users to connect to a Service Bus namespace and administer messaging entities in an easy manner. The tool provides advanced features like import/export functionality or the ability to test topic, queues, subscriptions, relay services, notification hubs and events hubs.

## Next steps

In this tutorial, you provisioned resources using the Azure portal, then sent and received messages from a Service Bus topic and its subscriptions. You learned how to:

- Create a Service Bus topic and one or more subscriptions to that topic using the Azure portal
- Add topic filters using .NET code
- Create two messages with different content
- Send the messages and verify they arrived in the expected subscriptions
- Receive messages from the subscriptions

For more examples of sending and receiving messages, get started with the [Service Bus samples on GitHub](#).

Advance to the next tutorial to learn more about using the publish/subscribe capabilities of Service Bus.

[Update inventory using PowerShell and topics/subscriptions](#)

# Tutorial: Update inventory using CLI and topics/subscriptions

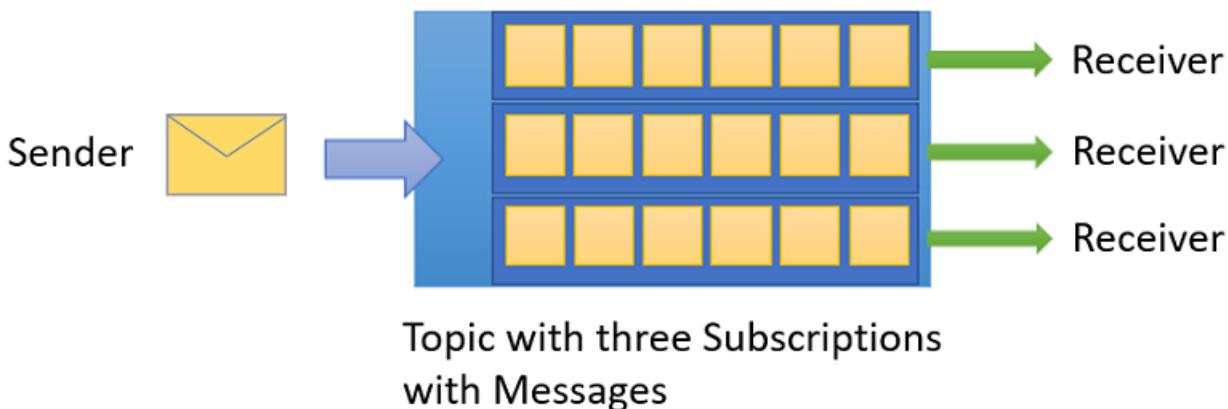
11/6/2019 • 9 minutes to read • [Edit Online](#)

Microsoft Azure Service Bus is a multi-tenant cloud messaging service that sends information between applications and services. Asynchronous operations give you flexible, brokered messaging, along with structured first-in, first-out (FIFO) messaging, and publish/subscribe capabilities. This tutorial shows how to use Service Bus topics and subscriptions in a retail inventory scenario, with publish/subscribe channels using Azure CLI and Java.

In this tutorial, you learn how to:

- Create a Service Bus topic and one or more subscriptions to that topic using Azure CLI
- Add topic filters using Azure CLI
- Create two messages with different content
- Send the messages and verify they arrived in the expected subscriptions
- Receive messages from the subscriptions

An example of this scenario is an inventory assortment update for multiple retail stores. In this scenario, each store, or set of stores, gets messages intended for them to update their assortments. This tutorial shows how to implement this scenario using subscriptions and filters. First, you create a topic with 3 subscriptions, add some rules and filters, and then send and receive messages from the topic and subscriptions.



If you don't have an Azure subscription, you can create a [free account](#) before you begin.

## Prerequisites

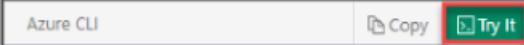
To develop a Service Bus app with Java, you must have the following installed:

- [Java Development Kit](#), latest version.
- [Azure CLI](#)
- [Apache Maven](#), version 3.0 or above.

## Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

OPTION	EXAMPLE/LINK
Select <b>Try It</b> in the upper-right corner of a code block. Selecting <b>Try It</b> doesn't automatically copy the code to Cloud Shell.	
Go to <a href="https://shell.azure.com">https://shell.azure.com</a> , or select the <b>Launch Cloud Shell</b> button to open Cloud Shell in your browser.	
Select the <b>Cloud Shell</b> button on the menu bar at the upper right in the <a href="#">Azure portal</a> .	

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

If you choose to install and use the CLI locally, this tutorial requires that you are running the Azure CLI version 2.0.4 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install the Azure CLI](#).

## Service Bus topics and subscriptions

Each [subscription to a topic](#) can receive a copy of each message. Topics are fully protocol and semantically compatible with Service Bus queues. Service Bus topics support a wide array of selection rules with filter conditions, with optional actions that set or modify message properties. Each time a rule matches, it produces a message. To learn more about rules, filters, and actions, follow this [link](#).

## Sign in to Azure

Once CLI is installed, open a command prompt and issue the following commands to sign in to Azure. These steps are not necessary if you're using Cloud Shell:

1. If you are using Azure CLI locally, run the following command to sign in to Azure. This sign on step is not necessary if you're running these commands in Cloud Shell:

```
az login
```

2. Set the current subscription context to the Azure subscription you want to use:

```
az account set --subscription Azure_subscription_name
```

## Use CLI to provision resources

Issue the following commands to provision Service Bus resources. Be sure to replace all placeholders with the appropriate values:

```

# Create a resource group
az group create --name myResourcegroup --location eastus

# Create a Service Bus messaging namespace with a unique name
namespaceName=myNameSpace$RANDOM
az servicebus namespace create \
--resource-group myResourceGroup \
--name $namespaceName \
--location eastus

# Create a Service Bus topic
az servicebus topic create --resource-group myResourceGroup \
--namespace-name $namespaceName \
--name myTopic

# Create subscription 1 to the topic
az servicebus subscription create --resource-group myResourceGroup --namespace-name $namespaceName --topic-name myTopic --name S1

# Create filter 1 - use custom properties
az servicebus rule create --resource-group myResourceGroup --namespace-name $namespaceName --topic-name myTopic --subscription-name S1 --name MyFilter --filter-sql-expression "StoreId IN ('Store1','Store2','Store3')"

# Create filter 2 - use custom properties
az servicebus rule create --resource-group myResourceGroup --namespace-name $namespaceName --topic-name myTopic --subscription-name S1 --name MySecondFilter --filter-sql-expression "StoreId = 'Store4'" 

# Create subscription 2
az servicebus subscription create --resource-group myResourceGroup --namespace-name $namespaceName --topic-name myTopic --name S2

# Create filter 3 - use message header properties via IN list and
# combine with custom properties.
az servicebus rule create --resource-group myResourceGroup --namespace-name $namespaceName --topic-name myTopic --subscription-name S2 --name MyFilter --filter-sql-expression "sys.To IN ('Store5','Store6','Store7') OR StoreId = 'Store8'"

# Create subscription 3
az servicebus subscription create --resource-group myResourceGroup --namespace-name $namespaceName --topic-name myTopic --name S3

# Create filter 4 - Get everything except messages for subscription 1 and 2.
# Also modify and add an action; in this case set the label to a specified value.
# Assume those stores might not be part of your main store, so you only add
# specific items to them. For that, you flag them specifically.
az servicebus rule create --resource-group DemoGroup --namespace-name DemoNamespaceSB --topic-name tutorialtest1
--subscription-name S3 --name MyFilter --filter-sql-expression "sys.To NOT IN ('Store1','Store2','Store3','Store4','Store5','Store6','Store7','Store8') OR StoreId NOT IN ('Store1','Store2','Store3','Store4','Store5','Store6','Store7','Store8') --action-sql-expression "SET sys.Label = 'SalesEvent'" 

# Get the connection string
connectionString=$(az servicebus namespace authorization-rule keys list \
--resource-group myResourceGroup \
--namespace-name $namespaceName \
--name RootManageSharedAccessKey \
--query primaryConnectionString --output tsv)

```

After the last command runs, copy and paste the connection string, and the queue name you selected, to a temporary location such as Notepad. You will need it in the next step.

## Create filter rules on subscriptions

After the namespace and topic/subscriptions are provisioned, and you have the necessary credentials, you are ready to create filter rules on the subscriptions, then send and receive messages. You can examine the code in [this GitHub sample folder](#).

## Send and receive messages

1. Make sure that Cloud Shell is open and displaying the Bash prompt.
2. Clone the [Service Bus GitHub repository](#) by issuing the following command:

```
git clone https://github.com/Azure/azure-service-bus.git
```

3. Navigate to the sample folder

```
azure-service-bus/samples/Java/quickstarts-and-tutorials/quickstart-java/tutorial-topics-subscriptions-filters-java
```

. Note that in the Bash shell, the commands are case-sensitive and path separators must be forward slashes.

4. Issue the following command to build the application:

```
mvn clean package -DskipTests
```

5. To run the program, issue the following command. Make sure to replace the placeholders with the connection string and topic name you obtained in the previous step:

```
java -jar .\target\tutorial-topics-subscriptions-filters-1.0.0-jar-with-dependencies.jar -c "myConnectionString" -t "myTopicName"
```

Observe 10 messages being sent to the topic, and subsequently received from the individual subscriptions:

```

Sending orders to topic.
Sent order to Store Store1. Price=4.100000, Color=Yellow, Category=Beverage
Sent order to Store Store2. Price=2.300000, Color=Blue, Category=Meat
Sent order to Store Store3. Price=3.200000, Color=Yellow, Category=Vegetables
Sent order to Store Store4. Price=4.100000, Color=Green, Category=Bread
Sent order to Store Store5. Price=1.400000, Color=Red, Category=Vegetables
Sent order to Store Store6. Price=5.100000, Color=Blue, Category=Bread
Sent order to Store Store7. Price=2.300000, Color=Orange, Category=Vegetables
Sent order to Store Store8. Price=4.100000, Color=Orange, Category=Meat
Sent order to Store Store9. Price=1.400000, Color=Red, Category=Meat
Sent order to Store Store10. Price=3.200000, Color=Green, Category=Bread

All messages sent.

Start Receiving Messages.

Receiving messages from subscription S1.

StoreId=Store2
Item data. Price=2.300000, Color=Blue, Category=Meat
StoreId=Store1
Item data. Price=4.100000, Color=Yellow, Category=Beverage
StoreId=Store3
Item data. Price=3.200000, Color=Yellow, Category=Vegetables
StoreId=Store4
Item data. Price=4.100000, Color=Green, Category=Bread

Received 4 messages from subscription S1.

Receiving messages from subscription S2.

StoreId=Store6
Item data. Price=5.100000, Color=Blue, Category=Bread
StoreId=Store5
Item data. Price=1.400000, Color=Red, Category=Vegetables
StoreId=Store7
Item data. Price=2.300000, Color=Orange, Category=Vegetables
StoreId=Store8
Item data. Price=4.100000, Color=Orange, Category=Meat

Received 4 messages from subscription S2.

Receiving messages from subscription S3.

StoreId=Store9
Item data. Price=1.400000, Color=Red, Category=Meat
StoreId=Store10
Item data. Price=3.200000, Color=Green, Category=Bread

Received 2 messages from subscription S3.

```

## Clean up resources

Run the following command to remove the resource group, namespace, and all related resources:

```
az group delete --resource-group my-resourcegroup
```

## Understand the sample code

This section contains more details about what the sample code does.

### Get connection string and queue

First, the code declares a set of variables, which drive the remaining execution of the program:

```

public String ConnectionString = null;
public String TopicName = null;
static final String[] Subscriptions = {"S1", "S2", "S3"};
static final String[] Store =
{"Store1", "Store2", "Store3", "Store4", "Store5", "Store6", "Store7", "Store8", "Store9", "Store10"};
static final String SysField = "sys.To";
static final String CustomField = "StoreId";
int NrOfMessagesPerStore = 1; // Send at least 1.

```

The connection string and the topic name are the only values added via command-line parameters and passed to

`main()`. The actual code execution is triggered in the `run()` method and sends, then receives messages from the topic:

```
public static void main(String[] args) {
    TutorialTopicsSubscriptionsFilters app = new TutorialTopicsSubscriptionsFilters();
    try {
        app.runApp(args);
        app.run();
    } catch (Exception e) {
        System.out.printf("%s", e.toString());
    }
    System.exit(0);
}

public void run() throws Exception {
    // Send sample messages.
    this.sendMessagesToTopic();

    // Receive messages from subscriptions.
    this.receiveAllMessages();
}
```

### Create topic client to send messages

To send and receive messages, the `sendMessagesToTopic()` method creates a topic client instance, which uses the connection string and the topic name, then calls another method that sends the messages:

```

public void sendMessagesToTopic() throws Exception, ServiceBusException {
    // Create client for the topic.
    TopicClient topicClient = new TopicClient(new ConnectionStringBuilder(ConnectionString, TopicName));

    // Create a message sender from the topic client.

    System.out.printf("\nSending orders to topic.\n");

    // Now we can start sending orders.
    CompletableFuture.allOf(
        SendOrders(topicClient,Store[0]),
        SendOrders(topicClient,Store[1]),
        SendOrders(topicClient,Store[2]),
        SendOrders(topicClient,Store[3]),
        SendOrders(topicClient,Store[4]),
        SendOrders(topicClient,Store[5]),
        SendOrders(topicClient,Store[6]),
        SendOrders(topicClient,Store[7]),
        SendOrders(topicClient,Store[8]),
        SendOrders(topicClient,Store[9])
    ).join();

    System.out.printf("\nAll messages sent.\n");
}

public CompletableFuture<Void> SendOrders(TopicClient topicClient, String store) throws Exception {

    for(int i = 0;i<NrOfMessagesPerStore;i++) {
        Random r = new Random();
        final Item item = new Item(r.nextInt(5),r.nextInt(5),r.nextInt(5));
        IMessage message = new Message(GSON.toJson(item,Item.class).getBytes(UTF_8));
        // We always set the Sent to field
        message.setTo(store);
        final String StoreId = store;
        Double priceToString = item.getPrice();
        final String priceForPut = priceToString.toString();
        message.setProperties(new HashMap<String, String>() {{
            // Additionally we add a customer store field. In reality you would use sys.To or a customer
            property but not both.
            // This is just for demo purposes.
            put("StoreId", StoreId);
            // Adding more potential filter / rule and action able fields
            put("Price", priceForPut);
            put("Color", item.getColor());
            put("Category", item.getItemCategory());
        }});
        System.out.printf("Sent order to Store %s. Price=%f, Color=%s, Category=%s\n", StoreId,
        item.getPrice(), item.getColor(), item.getItemCategory());
        topicClient.sendAsync(message);
    }

    return new CompletableFuture().completedFuture(null);
}

```

## Receive messages from the individual Subscriptions

The `receiveAllMessages()` method calls the `receiveAllMessageFromSubscription()` method, which then creates a subscription client per call and receives messages from the individual subscriptions:

```

public void receiveAllMessages() throws Exception {
    System.out.printf("\nStart Receiving Messages.\n");

    CompletableFuture.allOf(
        receiveAllMessageFromSubscription(Subscriptions[0]),
        receiveAllMessageFromSubscription(Subscriptions[1]),
        receiveAllMessageFromSubscription(Subscriptions[2])
    ).join();
}

public CompletableFuture<Void> receiveAllMessageFromSubscription(String subscription) throws Exception {

    int receivedMessages = 0;

    // Create subscription client.
    IMessageReceiver subscriptionClient = ClientFactory.createMessageReceiverFromConnectionStringBuilder(new
ConnectionStringBuilder(ConnectionString, TopicName+"/"+subscriptions+"/"+subscription), ReceiveMode.PEEKLOCK);

    // Create a receiver from the subscription client and receive all messages.
    System.out.printf("\nReceiving messages from subscription %s.\n\n", subscription);

    while (true)
    {
        // This will make the connection wait for N seconds if new messages are available.
        // If no additional messages come we close the connection. This can also be used to realize long
polling.
        // In case of long polling you would obviously set it more to e.g. 60 seconds.
        IMessage receivedMessage = subscriptionClient.receive(Duration.ofSeconds(1));
        if (receivedMessage != null)
        {
            if ( receivedMessage.getProperties() != null )
                System.out.printf("StoreId=%s\n", receivedMessage.getProperties().get("StoreId"));

            // Show the label modified by the rule action
            if(receivedMessage.getLabel() != null)
                System.out.printf("Label=%s\n", receivedMessage.getLabel());
        }

        byte[] body = receivedMessage.getBody();
        Item theItem = JSON.fromJson(new String(body, UTF_8), Item.class);
        System.out.printf("Item data. Price=%f, Color=%s, Category=%s\n", theItem.getPrice(),
theItem.getColor(), theItem.getItemCategory());

        subscriptionClient.complete(receivedMessage.getLockToken());
        receivedMessages++;
    }
    else
    {
        // No more messages to receive.
        subscriptionClient.close();
        break;
    }
}
System.out.printf("\nReceived %s messages from subscription %s.\n", receivedMessages, subscription);

return new CompletableFuture().completedFuture(null);
}

```

#### NOTE

You can manage Service Bus resources with [Service Bus Explorer](#). The Service Bus Explorer allows users to connect to a Service Bus namespace and administer messaging entities in an easy manner. The tool provides advanced features like import/export functionality or the ability to test topic, queues, subscriptions, relay services, notification hubs and events hubs.

## Next steps

In this tutorial, you provisioned resources using Azure CLI, then sent and received messages from a Service Bus topic and its subscriptions. You learned how to:

- Create a Service Bus topic and one or more subscriptions to that topic using the Azure portal
- Add topic filters using .NET code
- Create two messages with different content
- Send the messages and verify they arrived in the expected subscriptions
- Receive messages from the subscriptions

For more examples of sending and receiving messages, get started with the [Service Bus samples on GitHub](#).

Advance to the next tutorial to learn more about using the publish/subscribe capabilities of Service Bus.

[Update inventory using PowerShell and topics/subscriptions](#)

# Service Bus messaging samples

1/27/2020 • 2 minutes to read • [Edit Online](#)

The Service Bus messaging samples demonstrate key features in [Service Bus messaging](#). Currently, you can find the samples in the following places:

PROGRAMMING LANGUAGE	SDK OR SAMPLES LOCATION
.NET, Java, and Management	<a href="https://github.com/Azure/azure-service-bus/">https://github.com/Azure/azure-service-bus/</a>
Node.js	<a href="https://github.com/Azure/azure-sdk-for-js/tree/master/sdk/servicebus/service-bus/samples">https://github.com/Azure/azure-sdk-for-js/tree/master/sdk/servicebus/service-bus/samples</a>
Python	<a href="https://github.com/Azure/azure-sdk-for-python/tree/master/sdk/servicebus/azure-servicebus">https://github.com/Azure/azure-sdk-for-python/tree/master/sdk/servicebus/azure-servicebus</a>

## 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

See the following topics for conceptual overviews of Service Bus.

- [Service Bus messaging overview](#)
- [Service Bus architecture](#)

# Service Bus Premium and Standard messaging tiers

1/27/2020 • 4 minutes 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, 4 or 8 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. 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).

## Premium Messaging resource usage

In general, any operation on an entity may cause CPU and memory usage. Here are some of these operations:

- Management operations such as CRUD (Create, Retrieve, Update, and Delete) operations on queues, topics, and subscriptions.
- Runtime operations (send and receive messages)
- Monitoring operations and alerts

The additional CPU And memory usage is not priced additionally though. For the Premium Messaging tier, there is a single price for the message unit.

The CPU and memory usage are tracked and displayed to the you for the following reasons:

- Provide transparency into the system internals
- Understand the capacity of resources purchased.
- Capacity planning that helps you decide to scale up/down.

## Messaging unit - How many are needed?

When provisioning an Azure Service Bus Premium namespace, the number of messaging units allocated must be specified. These messaging units are dedicated resources that are allocated to the namespace.

The number of messaging units allocated to the Service Bus Premium namespace can be **dynamically adjusted** to factor in the change (increase or decrease) in workloads.

There are a number of factors to take into consideration when deciding the number of messaging units for your architecture:

- Start with **1 or 2 messaging units** allocated to your namespace.
- Study the CPU usage metrics within the [Resource usage metrics](#) for your namespace.
  - If CPU usage is **below 20%**, you might be able to **scale down** the number of messaging units allocated to your namespace.
  - If CPU usage is **above 70%**, your application will benefit from **scaling up** the number of messaging units allocated to your namespace.

The process of scaling the resources allocated to a Service Bus namespaces can be automated by using [Azure Automation Runbooks](#).

### NOTE

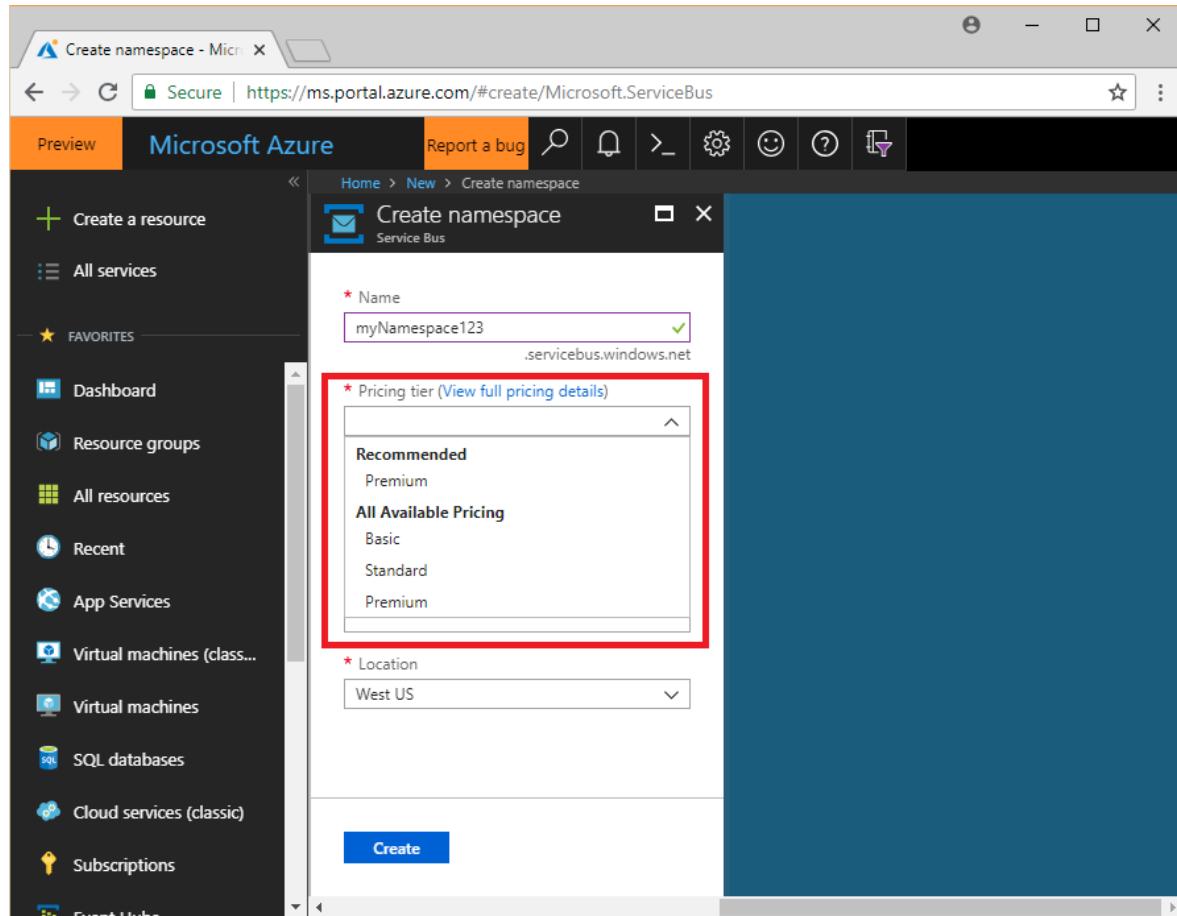
**Scaling** of the resources allocated to the namespace can be either preemptive or reactive.

- **Preemptive:** If additional workload is expected (due to seasonality or trends), you can proceed to allocate more messaging units to the namespace before the workloads hit.
- **Reactive:** If additional workloads are identified by studying the resource usage metrics, then additional resources can be allocated to the namespace to incorporate increasing demand.

The billing meters for Service Bus are hourly. In the case of scaling up, you only pay for the additional resources for the hours that these were used.

# 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 **Pricing tier**. Click **View full pricing details** to see more information about each tier.



You can also create [Premium namespaces](#) using Azure Resource Manager templates.

## Next steps

To learn more about Service Bus Messaging, see the following links:

- [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

12/31/2019 • 15 minutes 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.
- 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. For more information, see the following articles:
  - [Authenticate with managed identities](#)
  - [Authenticate from an application](#)
- 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	<b>No</b>  For more information, see the first note in the "Additional Information" section.	<b>Yes - First-In-First-Out (FIFO)</b>  (through the use of messaging sessions)
Delivery guarantee	<b>At-Least-Once</b>	<b>At-Least-Once</b> (using PeekLock receive mode - this is the default)  <b>At-Most-Once</b> (using ReceiveAndDelete receive mode)  Learn more about various <a href="#">Receive modes</a>
Atomic operation support	<b>No</b>	<b>Yes</b>

COMPARISON CRITERIA	STORAGE QUEUES	SERVICE BUS QUEUES
Receive behavior	<b>Non-blocking</b>  (completes immediately if no new message is found)	<b>Blocking with/without timeout</b>  (offers long polling, or the " <a href="#">Comet technique</a> ")
		<b>Non-blocking</b>  (through the use of .NET managed API only)
Push-style API	<b>No</b>	<b>Yes</b>  <a href="#">OnMessage</a> and <b>OnMessage</b> sessions .NET API.
Receive mode	<b>Peek &amp; Lease</b>	<b>Peek &amp; Lock</b>  <b>Receive &amp; Delete</b>
Exclusive access mode	<b>Lease-based</b>	<b>Lock-based</b>
Lease/Lock duration	<b>30 seconds (default)</b>  <b>7 days (maximum)</b> (You can renew or release a message lease using the <a href="#">UpdateMessage</a> API.)	<b>60 seconds (default)</b>  You can renew a message lock using the <a href="#">RenewLock</a> API.
Lease/Lock precision	<b>Message level</b>  (each message can have a different timeout value, which you can then update as needed while processing the message, by using the <a href="#">UpdateMessage</a> API)	<b>Queue level</b>  (each queue has a lock precision applied to all of its messages, but you can renew the lock using the <a href="#">RenewLock</a> API.)
Batched receive	<b>Yes</b>  (explicitly specifying message count when retrieving messages, up to a maximum of 32 messages)	<b>Yes</b>  (implicitly enabling a pre-fetch property or explicitly through the use of transactions)
Batched send	<b>No</b>	<b>Yes</b>  (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.
- Inconsistencies with regard to message handling in the context of Service Bus sessions can be avoided by using session state to store the application's state relative to the progress of handling the session's message sequence, and by using transactions around settling received messages and updating the session state. This kind of consistency feature is sometimes labeled *Exactly-Once Processing* in other vendor's products, but transaction failures will obviously cause messages to be redelivered and therefore the term is not exactly adequate.
- 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	<b>Yes</b>	<b>Yes</b>
Automatic dead lettering	<b>No</b>	<b>Yes</b>
Increasing queue time-to-live value	<b>Yes</b> (via in-place update of visibility timeout)	<b>Yes</b> (provided via a dedicated API function)
Poison message support	<b>Yes</b>	<b>Yes</b>
In-place update	<b>Yes</b>	<b>Yes</b>
Server-side transaction log	<b>Yes</b>	<b>No</b>

COMPARISON CRITERIA	STORAGE QUEUES	SERVICE BUS QUEUES
Storage metrics	<b>Yes</b>  <b>Minute Metrics:</b> 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 <a href="#">About Storage Analytics Metrics</a> .	<b>Yes</b>  (bulk queries by calling <a href="#">GetQueues</a> )
State management	<b>No</b>	<b>Yes</b>  <a href="#">Microsoft.ServiceBus.Messaging.EntityStatus.Active</a> , <a href="#">Microsoft.ServiceBus.Messaging.EntityStatus.Disabled</a> , <a href="#">Microsoft.ServiceBus.Messaging.EntityStatus.SendDisabled</a> , <a href="#">Microsoft.ServiceBus.Messaging.EntityStatus.ReceiveDisabled</a>
Message auto-forwarding	<b>No</b>	<b>Yes</b>
Purge queue function	<b>Yes</b>	<b>No</b>
Message groups	<b>No</b>	<b>Yes</b>  (through the use of messaging sessions)
Application state per message group	<b>No</b>	<b>Yes</b>
Duplicate detection	<b>No</b>	<b>Yes</b>  (configurable on the sender side)
Browsing message groups	<b>No</b>	<b>Yes</b>
Fetching message sessions by ID	<b>No</b>	<b>Yes</b>

## 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	<b>500 TB</b>  (limited to a <a href="#">single storage account capacity</a> )	<b>1 GB to 80 GB</b>  (defined upon creation of a queue and <a href="#">enabling partitioning</a> – see the "Additional Information" section)
Maximum message size	<b>64 KB</b>  (48 KB when using <b>Base64</b> encoding)  Azure supports large messages by combining queues and blobs – at which point you can enqueue up to 200 GB for a single item.	<b>256 KB or 1 MB</b>  (including both header and body, maximum header size: 64 KB).  Depends on the <a href="#">service tier</a> .
Maximum message TTL	<b>Infinite</b> (as of api-version 2017-07-27)	<b>TimeSpan.Max</b>
Maximum number of queues	<b>Unlimited</b>	<b>10,000</b>  (per service namespace)
Maximum number of concurrent clients	<b>Unlimited</b>	<b>Unlimited</b>  (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](#).

- Partitioning is not supported in the [Premium tier](#). 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 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](#).
- 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	<b>REST over HTTP/HTTPS</b>	<b>REST over HTTPS</b>
Runtime protocol	<b>REST over HTTP/HTTPS</b>	<b>REST over HTTPS</b> <b>AMQP 1.0 Standard (TCP with TLS)</b>
.NET API	<b>Yes</b> (.NET Storage Client API)	<b>Yes</b> (.NET Service Bus API)
Native C++	<b>Yes</b>	<b>Yes</b>
Java API	<b>Yes</b>	<b>Yes</b>
PHP API	<b>Yes</b>	<b>Yes</b>
Node.js API	<b>Yes</b>	<b>Yes</b>
Arbitrary metadata support	<b>Yes</b>	<b>No</b>
Queue naming rules	<b>Up to 63 characters long</b> (Letters in a queue name must be lowercase.)	<b>Up to 260 characters long</b> (Queue paths and names are case-insensitive.)
Get queue length function	<b>Yes</b> (Approximate value if messages expire beyond the TTL without being deleted.)	<b>Yes</b> (Exact, point-in-time value.)

COMPARISON CRITERIA	STORAGE QUEUES	SERVICE BUS QUEUES
Peek function	Yes	Yes

#### Additional information

- Storage queues provide support for arbitrary attributes that can be applied to the queue description, in the form of name/value pairs.
- Both queue technologies offer the ability to peek a message without having to lock it, which can be useful when implementing a queue explorer/browser tool.
- The Service Bus .NET brokered messaging APIs leverage full-duplex TCP connections for improved performance when compared to REST over HTTP, and they support the AMQP 1.0 standard protocol.
- Names of Storage queues can be 3-63 characters long, can contain lowercase letters, numbers, and hyphens. For more information, see [Naming Queues and Metadata](#).
- Service Bus queue names can be up to 260 characters long and have less restrictive naming rules. Service Bus queue names can contain letters, numbers, periods, hyphens, and underscores.

## Authentication and authorization

This section discusses the authentication and authorization features supported by Storage queues and Service Bus queues.

COMPARISON CRITERIA	STORAGE QUEUES	SERVICE BUS QUEUES
Authentication	Symmetric key	Symmetric key
Security model	Delegated access via SAS tokens.	SAS
Identity provider federation	No	Yes

#### Additional information

- Every request to either of the queuing technologies must be authenticated. Public queues with anonymous access are not supported. Using [SAS](#), you can address this scenario by publishing a write-only SAS, read-only SAS, or even a full-access SAS.
- The authentication scheme provided by Storage queues involves the use of a symmetric key, which is a hash-based Message Authentication Code (HMAC), computed with the SHA-256 algorithm and encoded as a **Base64** string. For more information about the respective protocol, see [Authentication for the Azure Storage Services](#). Service Bus queues support a similar model using symmetric keys. For more information, see [Shared Access Signature Authentication with Service Bus](#).

## Conclusion

By gaining a deeper understanding of the two technologies, you will be able to make a more informed decision on which queue technology to use, and when. The decision on when to use Storage queues or Service Bus queues clearly depends on a number of factors. These factors may depend heavily on the individual needs of your application and its architecture. If your application already uses the core capabilities of Microsoft Azure, you may prefer to choose Storage queues, especially if you require basic communication and messaging between services or need queues that can be larger than 80 GB in size.

Because Service Bus queues provide a number of advanced features, such as sessions, transactions, duplicate detection, automatic dead-lettering, and durable publish/subscribe capabilities, they may be a preferred choice if you are building a hybrid application or if your application otherwise requires these features.

## Next steps

The following articles provide more guidance and information about using Storage queues or Service Bus queues.

- [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

1/17/2020 • 15 minutes 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 article, 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 article 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. It is recommended that you do 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 reusing the same factory and client objects for multiple operations. You can safely use these client objects for concurrent asynchronous operations and from 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.

The client schedules concurrent operations by performing asynchronous operations. The next request is started before the previous request is completed. The following code snippet is an example of an asynchronous send operation:

```

Message m1 = new BrokeredMessage(body);
Message 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");

```

The following code is an example of an asynchronous receive operation. See the full program [here](#):

```

var receiver = new MessageReceiver(connectionString, queueName, ReceiveMode.PeekLock);
var doneReceiving = new TaskCompletionSource<bool>();

receiver.RegisterMessageHandler(...);

```

## 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. These steps reduce 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 20 ms. 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 using the [Microsoft.ServiceBus.Messaging](#) library. The HTTP protocol does not support batching.

#### NOTE

Setting BatchFlushInterval ensures that the batching is implicit from the application's perspective. i.e. The application makes SendAsync() and CompleteAsync() calls and does not make specific Batch calls.

Explicit client side batching can be implemented by utilizing the below method call -

```
Task SendBatchAsync (IEnumerable<BrokeredMessage> messages);
```

Here the combined size of the messages must be less than the maximum size supported by the pricing tier.

## 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 20 ms.

#### 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. With the lock, 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 [PrefetchCount](#) is 20 times the maximum processing rates of all receivers of the factory. For example, a factory creates three 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, [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.

## Prefetching and ReceiveBatch

While the concepts of prefetching multiple messages together have similar semantics to processing messages in a batch (`ReceiveBatch`), there are some minor differences that must be kept in mind when leveraging these together.

Prefetch is a configuration (or mode) on the client (`QueueClient` and `SubscriptionClient`) and `ReceiveBatch` is an operation (that has request-response semantics).

While using these together, consider the following cases -

- Prefetch should be greater than or equal to the number of messages you are expecting to receive from `ReceiveBatch`.
- Prefetch can be up to  $n/3$  times the number of messages processed per second, where  $n$  is the default lock duration.

There are some challenges with having a greedy approach(i.e. keeping the prefetch count very high), because it implies that the message is locked to a particular receiver. The recommendation is to try out prefetch values between the thresholds mentioned above and empirically identify what fits.

## Multiple queues

If the expected load cannot be handled by a single 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, 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.

- 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 50 ms to reduce the number of Service Bus client protocol transmissions. If multiple senders are used, increase the batching interval to 100 ms.
- Leave batched store access enabled. This access 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 count 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.

- 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 setting the count, 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, 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, perform the following steps:

- 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 20 ms to reduce the number of Service Bus client protocol transmissions.
- Leave batched store access enabled. This access 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 count 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, 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:

- 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 access 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 count 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:

- 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 20 ms to reduce the number of Service Bus client protocol transmissions.
- Leave batched store access enabled. This access 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 count 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 low throughput 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, try the following steps:

- Use asynchronous operations to take advantage of client-side batching.
- Use the default batching interval of 20 ms to reduce the number of Service Bus client protocol transmissions.

- Leave batched store access enabled. This access 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 count reduces the number of Service Bus client protocol transmissions.

# Azure Service Bus Geo-disaster recovery

6/18/2019 • 6 minutes 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* is an important feature for any enterprise. Azure Service Bus supports geo-disaster recovery at the namespace level.

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

## NOTE

Geo-Disaster recovery currently only ensures that the metadata (Queues, Topics, Subscriptions, Filters) are copied over from the primary namespace to secondary namespace when paired.

## 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:

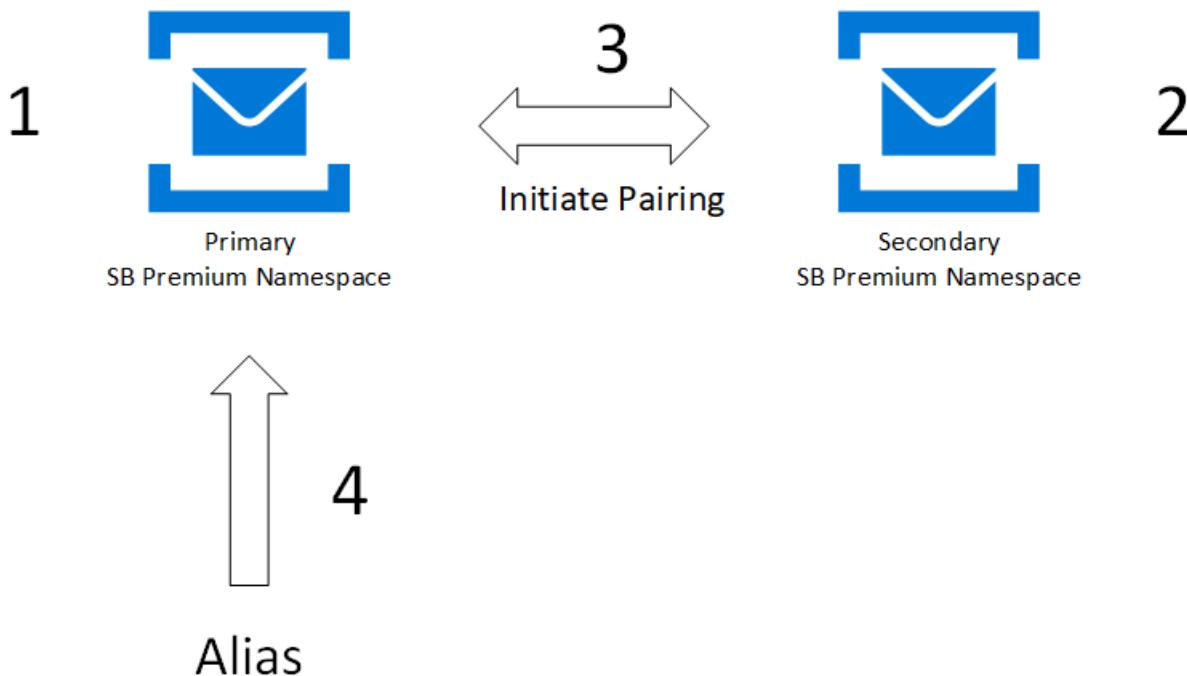
- **Alias:** The name for a disaster recovery configuration that you set up. The alias provides a single stable Fully Qualified Domain Name (FQDN) connection string. Applications use this alias connection string to connect to a namespace. Using an alias ensures that the connection string is unchanged when the failover is triggered.
- **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

The following section is an overview to setup pairing between the namespaces.



The setup process is as follows -

1. Provision a **Primary** Service Bus Premium Namespace.
2. Provision a **Secondary** Service Bus Premium Namespace in a region *different from where the primary namespace is provisioned*. This will help allow fault isolation across different datacenter regions.
3. Create pairing between the Primary namespace and Secondary namespace to obtain the **alias**.

### NOTE

If you have [migrated your Azure Service Bus Standard namespace to Azure Service Bus Premium](#), then you must use the pre-existing alias (i.e. your Service Bus Standard namespace connection string) to create the disaster recovery configuration through the **PS/CLI or REST API**.

This is because, during migration, your Azure Service Bus Standard namespace connection string/DNS name itself becomes an alias to your Azure Service Bus Premium namespace.

Your client applications must utilize this alias (i.e. the Azure Service Bus Standard namespace connection string) to connect to the Premium namespace where the disaster recovery pairing has been setup.

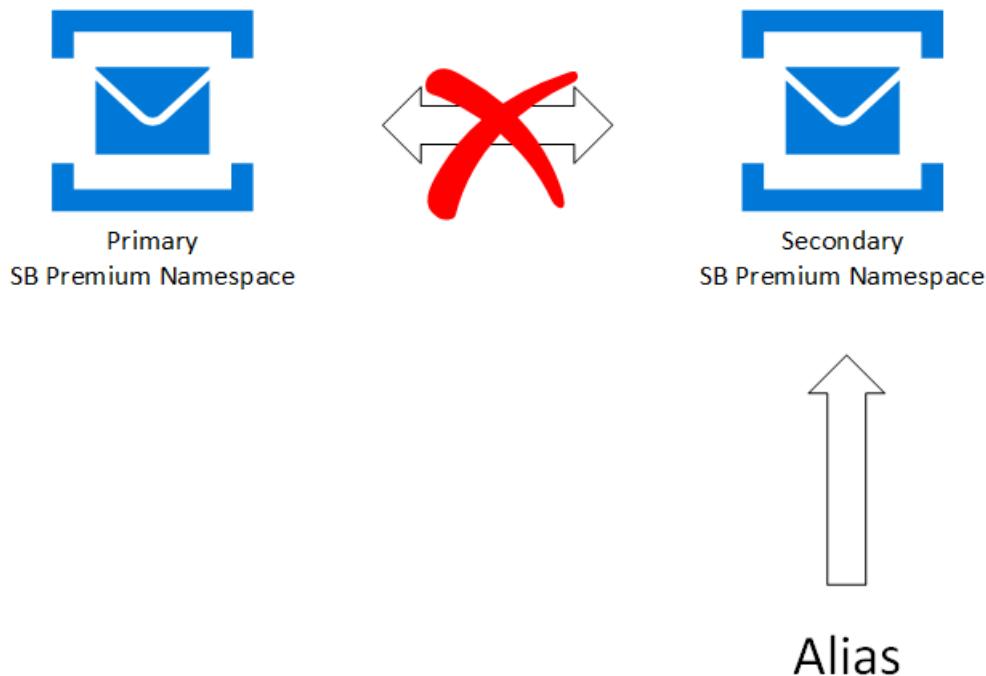
If you use the Portal to setup the Disaster recovery configuration, then the portal will abstract this caveat from you.

4. Use the **alias** obtained in step 3 to connect your client applications to the Geo-DR enabled primary namespace. Initially, the alias points to the primary namespace.

5. [Optional] Add some monitoring to detect if a failover is necessary.

## Failover flow

A failover is triggered manually by the customer (either explicitly through a command, or through client owned business logic that triggers the command) and never by Azure. This gives the customer full ownership and visibility for outage resolution on Azure's backbone.



After the failover is triggered -

1. The **alias** connection string is updated to point to the Secondary Premium namespace.
2. Clients(senders and receivers) automatically connect to the Secondary namespace.
3. The existing pairing between Primary and Secondary premium namespace is broken.

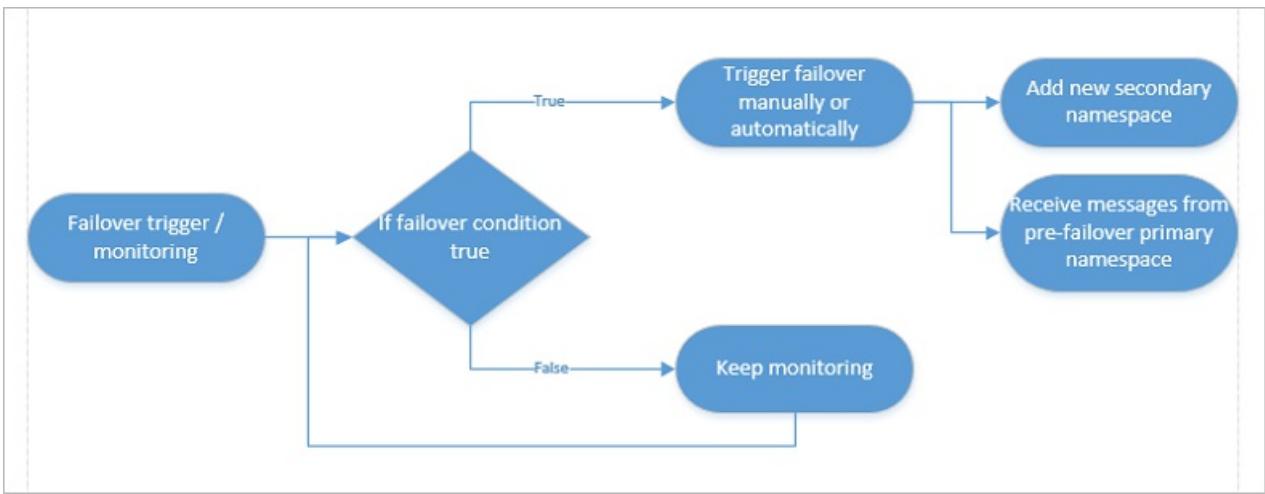
Once the failover is initiated -

1. If another outage occurs, you want to be able to fail over 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.

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.



## 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 that are required in Azure Active Directory to use Azure Resource Manager with Service Bus, to set up and enable Geo-disaster recovery.
- Steps required to execute the sample code.
- How to use an existing namespace as an 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.

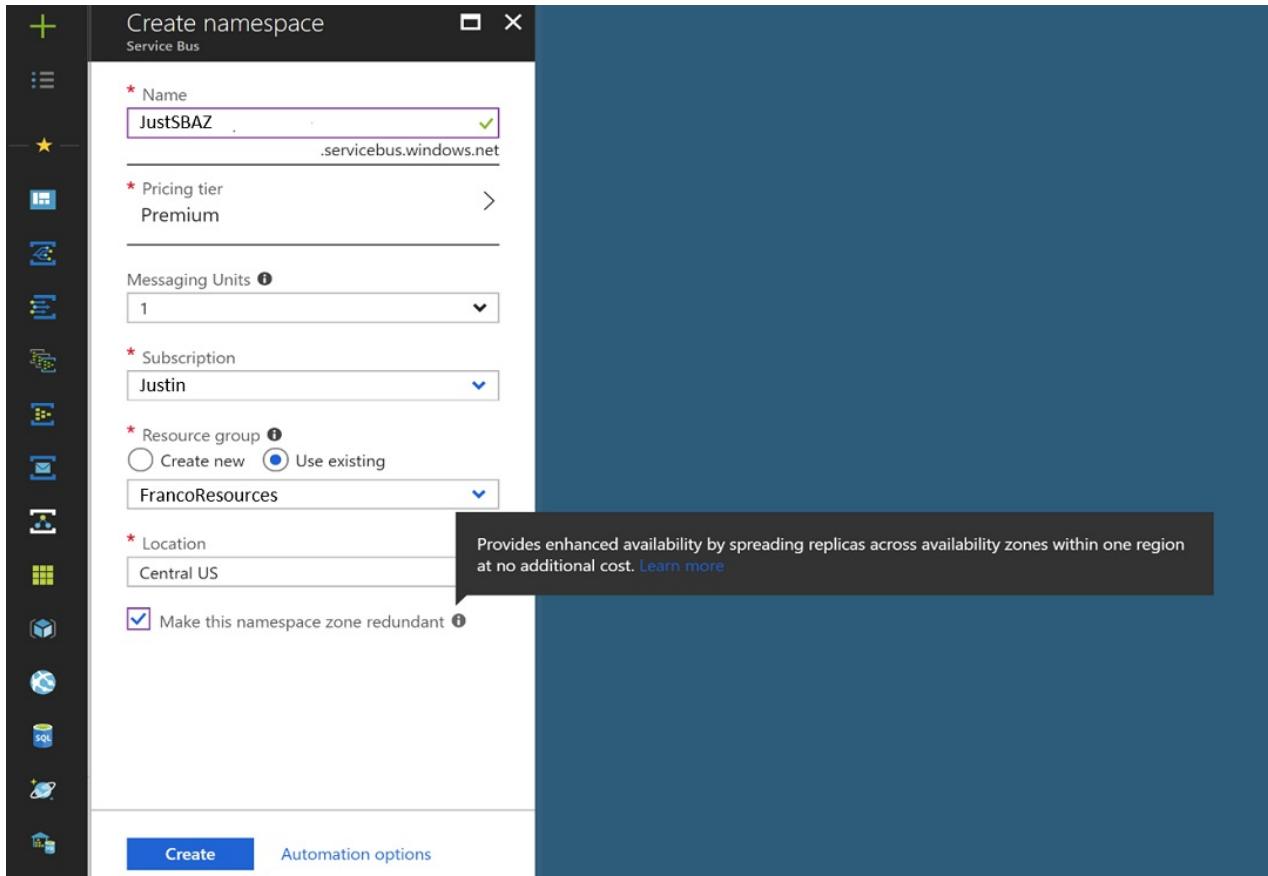
## Availability Zones

The Service Bus Premium SKU also supports [Availability Zones](#), providing fault-isolated locations within an Azure region.

**NOTE**

The Availability Zones support for Azure Service Bus Premium is only available in [Azure regions](#) where availability zones are present.

You can enable Availability Zones on new namespaces only, using the Azure portal. Service Bus does not support migration of existing namespaces. You cannot disable zone redundancy after enabling it on your namespace.



## 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 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

1/24/2020 • 4 minutes 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 mentioned in the article on [handling outages and disasters](#).

## **Next steps**

Now that you've learned the basics of asynchronous messaging in Service Bus, read more details about [handling outages and disasters](#).

# Best practices for insulating applications against Service Bus outages and disasters

1/27/2020 • 6 minutes to read • [Edit Online](#)

Mission-critical applications must operate continuously, even in the presence of unplanned outages or disasters. This article 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.

## Protecting against Outages and Disasters - Service Bus Premium

High Availability and Disaster Recovery concepts are built right into the Azure Service Bus Premium tier, both within the same region (via Availability Zones) and across different regions (via Geo-Disaster Recovery).

### Geo-Disaster Recovery

Service Bus Premium supports Geo-disaster recovery, 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.

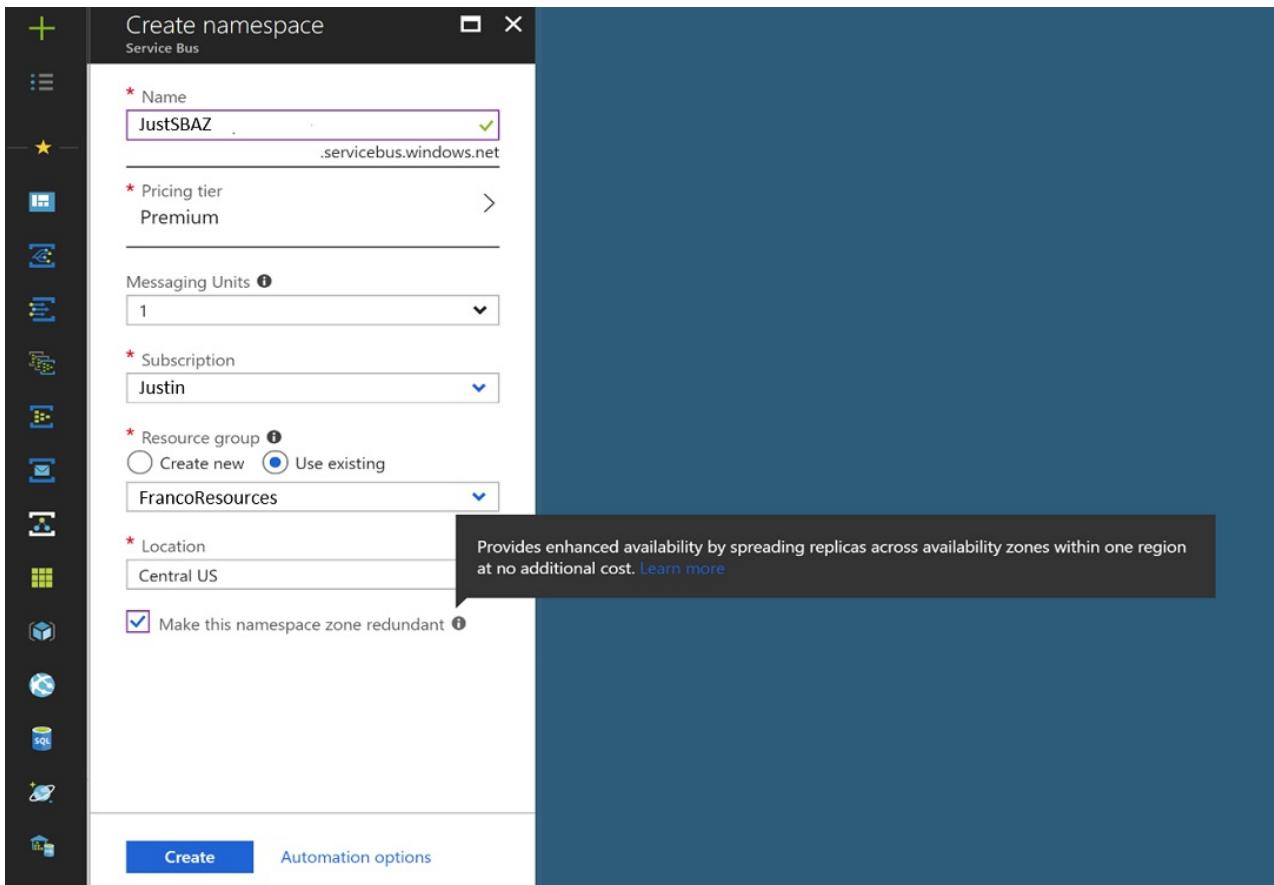
### Availability Zones

The Service Bus Premium SKU supports [Availability Zones](#), providing fault-isolated locations within the same Azure region.

#### NOTE

The Availability Zones support for Azure Service Bus Premium is only available in [Azure regions](#) where availability zones are present.

You can enable Availability Zones on new namespaces only, using the Azure portal. Service Bus does not support migration of existing namespaces. You cannot disable zone redundancy after enabling it on your namespace.



## Protecting against Outages and Disasters - Service Bus Standard

To achieve resilience against datacenter outages when using the standard messaging pricing tier, 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**.

### NOTE

The **Active Replication** and **Passive Replication** setup are general purpose solutions and not specific features of Service Bus. The replication logic (sending to 2 different namespaces) lives on the sender applications and the receiver has to have custom logic for duplicate detection.

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 Standard Tier](#) 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 Standard Tier](#) sample demonstrates passive replication of messaging entities.

## Protecting relay endpoints against datacenter outages or disasters

Geo-replication of [Azure 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.

## 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](#)

# Throttling operations on Azure Service Bus

10/11/2019 • 5 minutes to read • [Edit Online](#)

Cloud native solutions give a notion of unlimited resources that can scale with your workload. While this notion is more true in the cloud than it is with on-premises systems, there are still limitations that exist in the cloud.

These limitations may cause throttling of client application requests in both Standard and Premium tiers as discussed in this article.

## Throttling in Azure Service Bus Standard tier

The Azure Service Bus Standard tier operates as a multi-tenant setup with a pay-as-you-go pricing model. Here multiple namespaces in the same cluster share the allocated resources. Standard tier is the recommended choice for developer, testing, and QA environments along with low throughput production systems.

In the past, Azure Service Bus had coarse throttling limits strictly dependent on resource utilization. However, there is an opportunity to refine the throttling logic and provide predictable throttling behavior to all namespaces that are sharing these resources.

In an attempt to ensure fair usage and distribution of resources across all the Azure Service Bus Standard namespaces that use the same resources, the throttling logic has been modified to be credit-based.

### NOTE

It is important to note that throttling is **not new** to Azure Service Bus, or any cloud native service.

Credit based throttling is simply refining the way various namespaces share resources in a multi-tenant Standard tier environment and thus enabling fair usage by all namespaces sharing the resources.

### What is credit-based throttling?

Credit-based throttling limits the number of operations that can be performed on a given namespace in a specific time period.

Below is the workflow for credit-based throttling -

- At the start of each time period, we provide a certain number of credits to each namespace.
- Any operations performed by the sender or receiver client applications will be counted against these credits (and subtracted from the available credits).
- If the credits are depleted, subsequent operations will be throttled until the start of the next time period.
- Credits are replenished at the start of the next time period.

### What are the credit limits?

The credit limits are currently set to '1000' credits every second (per namespace).

Not all operations are created equal. Here are the credit costs of each of the operations -

OPERATION	CREDIT COST
Data operations (Send, SendAsync, Receive, ReceiveAsync, Peek)	1 credit per message

OPERATION	CREDIT COST
Management operations (Create, Read, Update, Delete on Queues, Topics, Subscriptions, Filters)	10 credits

### How will I know that I'm being throttled?

When the client application requests are being throttled, the below server response will be received by your application and logged.

```
The request was terminated because the entity is being throttled. Error code: 50009. Please wait 2 seconds and try again.
```

### How can I avoid being throttled?

With shared resources, it is important to enforce some sort of fair usage across various Service Bus namespaces that share those resources. Throttling ensures that any spike in a single workload does not cause other workloads on the same resources to be throttled.

As mentioned later in the article, there is no risk in being throttled because the client SDKs (and other Azure PaaS offerings) have the default retry policy built into them. Any throttled requests will be retried with exponential backoff and will eventually go through when the credits are replenished.

Understandably, some applications may be sensitive to being throttled. In that case, it is recommended to [migrate your current Service Bus Standard namespace to Premium](#).

On migration, you can allocate dedicated resources to your Service Bus namespace and appropriately scale up the resources if there is a spike in your workload and reduce the likelihood of being throttled. Additionally, when your workload reduces to normal levels, you can scale down the resources allocated to your namespace.

## Throttling in Azure Service Bus Premium tier

The [Azure Service Bus Premium](#) tier allocates dedicated resources, in terms of messaging units, to each namespace setup by the customer. These dedicated resources provide predictable throughput and latency and are recommended for high throughput or sensitive production grade systems.

Additionally, the Premium tier also enables customers to scale up their throughput capacity when they experience spikes in the workload.

### How does throttling work in Service Bus Premium?

With exclusive resource allocation for Service Bus Premium, throttling is purely driven by the limitations of the resources allocated to the namespace.

If the number of requests are more than the current resources can service, then the requests will be throttled.

### How will I know that I'm being throttled?

There are various ways to identifying throttling in Azure Service Bus Premium -

- **Throttled Requests** show up on the [Azure Monitor Request metrics](#) to identify how many requests were throttled.
- High **CPU Usage** indicates that current resource allocation is high and requests may get throttled if the current workload doesn't reduce.
- High **Memory Usage** indicates that current resource allocation is high and requests may get throttled if the current workload doesn't reduce.

### How can I avoid being throttled?

Since the Service Bus Premium namespace already has dedicated resources, you can reduce the possibility of

getting throttled by scaling up the number of Messaging Units allocated to your namespace in the event (or anticipation) of a spike in the workload.

Scaling up/down can be achieved by creating [runbooks](#) that can be triggered by changes in the above metrics.

## FAQs

### **How does throttling affect my application?**

When a request is throttled, it implies that the service is busy because it is facing more requests than the resources allow. If the same operation is tried again after a few moments, once the service has worked through its current workload, then the request can be honored.

Since throttling is the expected behavior of any cloud native service, we have built the retry logic into the Azure Service Bus SDK itself. The default is set to auto retry with an exponential back-off to ensure that we don't have the same request being throttled each time.

The default retry logic will apply to every operation.

### **Does throttling result in data loss?**

Azure Service Bus is optimized for persistence, we ensure that all the data sent to Service Bus is committed to storage before the service acknowledges the success of the request.

Once the request is successfully 'ACK' (acknowledged) by Service Bus, it implies that Service Bus has successfully processed the request. If Service Bus returns a 'NACK' (failure), then it implies that Service Bus has not been able to process the request and the client application must retry the request.

However, when a request is throttled, the service is implying that it cannot accept and process the request right now because of resource limitations. This **does not** imply any sort of data loss because Service Bus simply hasn't looked at the request. In this case, relying on the default retry policy of the Service Bus SDK ensures that the request is eventually processed.

## Next steps

For more information and examples of using Service Bus messaging, see the following advanced topics:

- [Service Bus messaging overview](#)
- [Quickstart: Send and receive messages using the Azure portal and .NET](#)
- [Tutorial: Update inventory using Azure portal and topics/subscriptions](#)

# Service Bus authentication and authorization

8/23/2019 • 3 minutes 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](#).

## Azure Active Directory

Azure Active Directory (Azure AD) integration for Service Bus resources provides role-based access control (RBAC) for fine-grained control over a client's access to resources. You can use role-based access control (RBAC) to grant permissions to security principal, which may be a user, a group, or an application service principal. The security principal is authenticated by Azure AD to return an OAuth 2.0 token. The token can be used to authorize a request to access a Service Bus resource (queue, topic, etc.).

For more information about authenticating with Azure AD, see the following articles:

- [Authenticate with managed identities](#)
- [Authenticate from an application](#)

## IMPORTANT

Authorizing users or applications using OAuth 2.0 token returned by Azure AD provides superior security and ease of use over shared access signatures (SAS). With Azure AD, there is no need to store the tokens in your code and risk potential security vulnerabilities. We recommend that you use using Azure AD with your Azure Service Bus applications when possible.

## Shared access signature

[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](#).

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

1/27/2020 • 4 minutes 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 access control with Shared Access Signatures

1/16/2020 • 15 minutes 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.

## NOTE

Azure Service Bus supports authorizing access to a Service Bus namespace and its entities using Azure Active Directory (Azure AD). Authorizing users or applications using OAuth 2.0 token returned by Azure AD provides superior security and ease of use over shared access signatures (SAS). With Azure AD, there is no need to store the tokens in your code and risk potential security vulnerabilities.

Microsoft recommends using Azure AD with your Azure Service Bus applications when possible. For more information, see the following articles:

- [Authenticate and authorize an application with Azure Active Directory to access Azure Service Bus entities.](#)
- [Authenticate a managed identity with Azure Active Directory to access Azure Service Bus resources](#)

## 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.

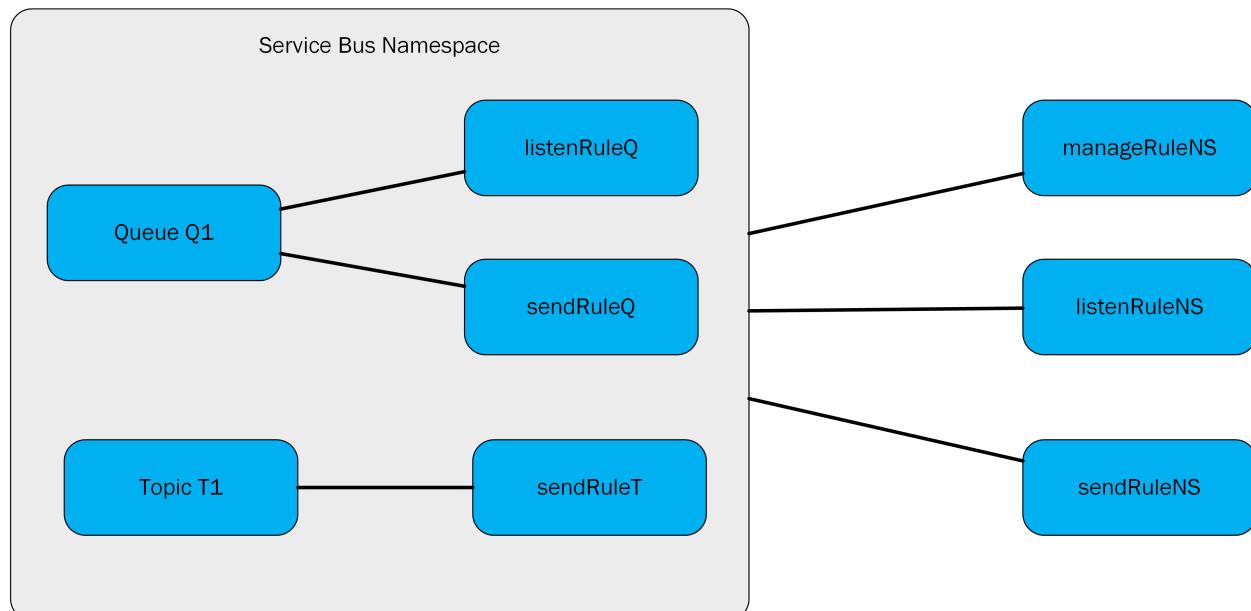
A namespace or entity policy can hold up to 12 Shared Access Authorization rules, providing room for three sets of rules, each covering the basic rights and the combination of Send and Listen. This limit underlines that the SAS policy store is not intended to be a user or service account store. If your 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 LF.

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 [AMQP.NET Lite](#) library. This is useful if you can't use the official Service Bus SDK (for example on WinRT, .NET Compact Framework, .NET Micro Framework and Mono) developing in C#. Of course, this library is useful to help understand how claims-based security works at the AMQP level, as you saw how it works at the HTTP level (with an HTTP POST request and the SAS token sent inside the "Authorization" header). If you don't need such deep knowledge about AMQP, you can use the official Service Bus SDK with .NET Framework applications, which will do it for you.

C#

```

/// <summary>
/// Send claim-based security (CBS) token
/// </summary>
/// <param name="shareAccessSignature">Shared access signature (token) to send</param>
private bool PutCbsToken(Connection connection, string sasToken)
{
    bool result = true;
    Session session = new Session(connection);

    string cbsClientAddress = "cbs-client-reply-to";
    var cbsSender = new SenderLink(session, "cbs-sender", "$cbs");
    var cbsReceiver = new ReceiverLink(session, cbsClientAddress, "$cbs");

    // construct the put-token message
    var request = new Message(sasToken);
    request.Properties = new Properties();
    request.Properties.MessageId = Guid.NewGuid().ToString();
    request.Properties.ReplyTo = cbsClientAddress;
    request.ApplicationProperties = new ApplicationProperties();
    request.ApplicationProperties["operation"] = "put-token";
    request.ApplicationProperties["type"] = "servicebus.windows.net:sastoken";
    request.ApplicationProperties["name"] = Fx.Format("amqp:///{0}/{1}", sbNamespace, entity);
    cbsSender.Send(request);

    // receive the response
    var response = cbsReceiver.Receive();
    if (response == null || response.Properties == null || response.ApplicationProperties == null)
    {
        result = false;
    }
    else
    {
        int statusCode = (int)response.ApplicationProperties["status-code"];
        if (statusCode != (int)HttpStatusCode.Accepted && statusCode != (int)HttpStatusCode.OK)
        {
            result = false;
        }
    }
}

// the sender/receiver may be kept open for refreshing tokens
cbsSender.Close();
cbsReceiver.Close();
session.Close();

return result;
}

```

The `PutCbsToken()` method receives the `connection` (AMQP connection class instance as provided by the [AMQP .NET Lite library](#)) that represents the TCP connection to the service and the `sasToken` parameter that is the SAS token to send.

#### NOTE

It's important that the connection is created with **SASL authentication mechanism set to 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 queues, topics, and subscriptions](#)
- [How to use Service Bus queues](#)
- [How to use Service Bus topics and subscriptions](#)

# Authenticate a managed identity with Azure Active Directory to access Azure Service Bus resources

1/24/2020 • 8 minutes to read • [Edit Online](#)

[Managed identities for Azure resources](#) 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 managed identities, 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 managed entities for Azure resources 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 managed entity's context to the client in an operation that is shown in an example later in this article. Once it is associated with a managed identity, your Service Bus client can do all authorized operations. Authorization is granted by associating a managed entity with Service Bus roles.

## Overview

When a security principal (a user, group, or application) attempts to access a Service Bus entity, the request must be authorized. With Azure AD, access to a resource is a two-step process.

1. First, the security principal's identity is authenticated, and an OAuth 2.0 token is returned. The resource name to request a token is <https://servicebus.azure.net>.
2. Next, the token is passed as part of a request to the Service Bus service to authorize access to the specified resource.

The authentication step requires that an application request contains an OAuth 2.0 access token at runtime. If an application is running within an Azure entity such as an Azure VM, a virtual machine scale set, or an Azure Function app, it can use a managed identity to access the resources.

The authorization step requires that one or more RBAC roles be assigned to the security principal. Azure Service Bus provides RBAC roles that encompass sets of permissions for Service Bus resources. The roles that are assigned to a security principal determine the permissions that the principal will have. To learn more about assigning RBAC roles to Azure Service Bus, see [Built-in RBAC roles for Azure Service Bus](#).

Native applications and web applications that make requests to Service Bus can also authorize with Azure AD. This article shows you how to request an access token and use it to authorize requests for Service Bus resources.

## Assigning RBAC roles for access rights

Azure Active Directory (Azure AD) authorizes access rights to secured resources through [role-based access control \(RBAC\)](#). Azure Service Bus defines a set of built-in RBAC roles that encompass common sets of permissions used to access Service Bus entities and you can also define custom roles for accessing the data.

When an RBAC role is assigned to an Azure AD security principal, Azure grants access to those resources for that security principal. Access can be scoped to the level of subscription, the resource group, or the Service Bus namespace. An Azure AD security principal may be a user, a group, an application service principal, or a managed identity for Azure resources.

# Built-in RBAC roles for Azure Service Bus

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. Azure provides the below built-in RBAC roles for authorizing access to a Service Bus namespace:

- [Azure Service Bus Data Owner](#): Enables data access to Service Bus namespace and its entities (queues, topics, subscriptions, and filters)
- [Azure Service Bus Data Sender](#): Use this role to give send access to Service Bus namespace and its entities.
- [Azure Service Bus Data Receiver](#): Use this role to give receiving access to Service Bus namespace and its entities.

## Resource scope

Before you assign an RBAC role to a security principal, determine the scope of access that the security principal should have. Best practices dictate that it's always best to grant only the narrowest possible scope.

The following list describes the levels at which you can scope access to Service Bus resources, starting with the narrowest scope:

- **Queue, topic, or subscription**: Role assignment applies to the specific Service Bus entity. Currently, the Azure portal doesn't support assigning users/groups/managed identities to Service Bus RBAC roles at the subscription level. Here's an example of using the Azure CLI command: [az-role-assignment-create](#) to assign an identity to a Service Bus RBAC role:

```
az role assignment create \
    --role $service_bus_role \
    --assignee $assignee_id \
    --scope
    /subscriptions/$subscription_id/resourceGroups/$resource_group/providers/Microsoft.ServiceBus/namespaces/
    /$service_bus_namespace/topics/$service_bus_topic/subscriptions/$service_bus_subscription
```

- **Service Bus namespace**: Role assignment spans the entire topology of Service Bus under the namespace and to the consumer group associated with it.
- **Resource group**: Role assignment applies to all the Service Bus resources under the resource group.
- **Subscription**: Role assignment applies to all the Service Bus resources in all of the resource groups in the subscription.

### NOTE

Keep in mind that RBAC role assignments may take up to five minutes to propagate.

For more information about how built-in roles are defined, see [Understand role definitions](#). For information about creating custom RBAC roles, see [Create custom roles for Azure Role-Based Access Control](#).

## Enable managed identities on a VM

Before you can use managed identities for Azure Resources to authorize Service Bus resources from your VM, you must first enable managed identities for Azure Resources on the VM. To learn how to enable managed identities for Azure Resources, see one of these articles:

- [Azure portal](#)
- [Azure PowerShell](#)

- Azure CLI
- Azure Resource Manager template
- Azure Resource Manager client libraries

## Grant permissions to a managed identity in Azure AD

To authorize a request to the Service Bus service from a managed identity in your application, first configure role-based access control (RBAC) settings for that managed identity. Azure Service Bus defines RBAC roles that encompass permissions for sending and reading from Service Bus. When the RBAC role is assigned to a managed identity, the managed identity is granted access to Service Bus entities at the appropriate scope.

For more information about assigning RBAC roles, see [Authenticate and authorize with Azure Active Directory for access to Service Bus resources](#).

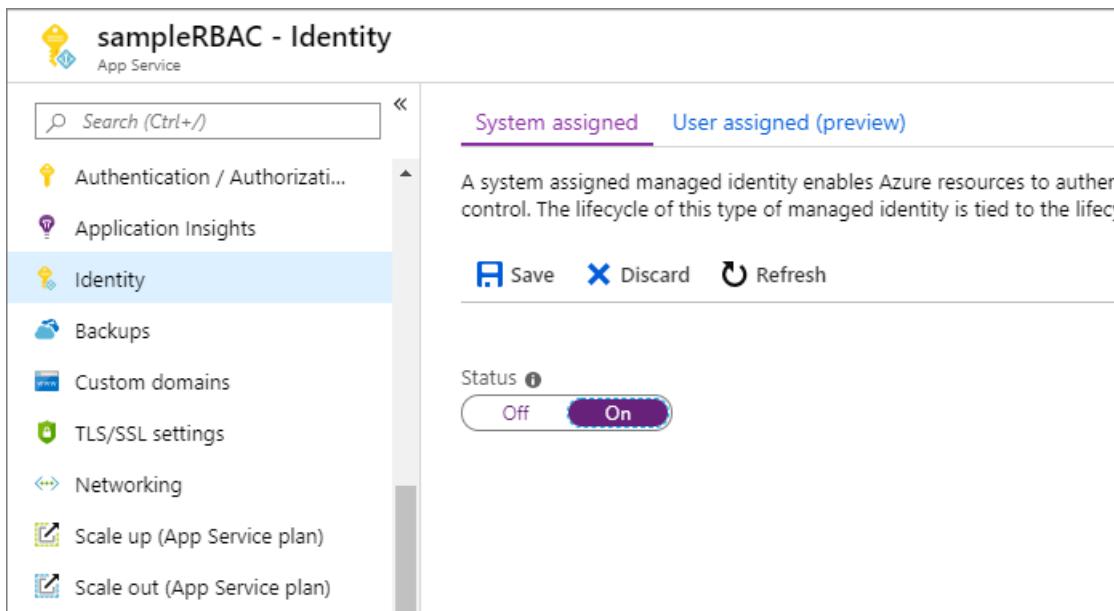
## Use Service Bus with managed identities for Azure resources

To use Service Bus with managed identities, you need to assign the identity the role and the appropriate scope. The procedure in this section uses a simple application that runs under a managed identity and accesses Service Bus resources.

Here we're using a sample web application hosted in [Azure App Service](#). For step-by-step instructions for creating a web application, see [Create an ASP.NET Core web app in Azure](#)

Once the application is created, follow these steps:

1. Go to **Settings** and select **Identity**.
2. Select the **Status** to be **On**.
3. Select **Save** to save the setting.



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

Now, assign this service identity to a role in the required scope in your Service Bus resources.

### To Assign RBAC roles using the Azure portal

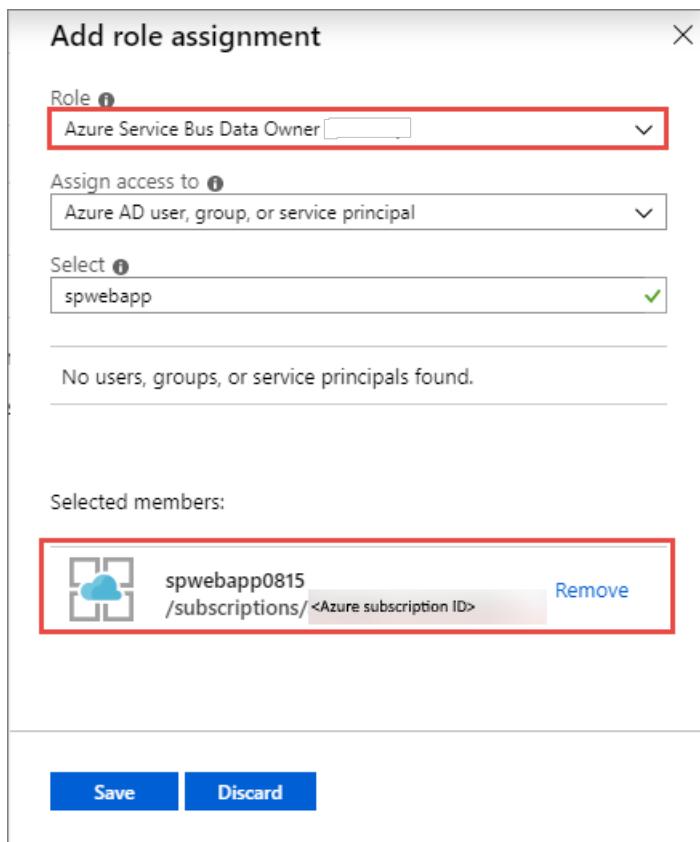
To assign a role to a Service Bus namespace, navigate to the namespace in the Azure portal. Display the Access Control (IAM) settings for the resource, and follow these instructions to manage role assignments:

## NOTE

The following steps assigns a service identity role to your Service Bus namespaces. You can follow the same steps to assign a role at other supported scopes (resource group and subscription).

[Create a Service Bus Messaging namespace](#) if you don't have one.

1. In the Azure portal, navigate to your Service Bus namespace and display the **Overview** for the namespace.
2. Select **Access Control (IAM)** on the left menu to display access control settings for the Service Bus namespace.
3. Select the **Role assignments** tab to see the list of role assignments.
4. Select **Add** to add a new role.
5. On the **Add role assignment** page, select the Azure Service Bus roles that you want to assign. Then search to locate the service identity you had registered to assign the role.



6. Select **Save**. The identity to whom you assigned the role appears listed under that role. For example, the following image shows that service identity has Azure Service Bus Data owner.

The screenshot shows the 'Access control (IAM)' blade for the 'spsbusns0815' namespace. The 'Role assignments' tab is selected. A table lists role assignments with columns for NAME, TYPE, ROLE, and SCOPE. One entry is highlighted: 'spwebapp0815 /subscriptions/<Your Azure subscription ID>' with 'Azure Service Bus Data Owner' in the ROLE column and 'This resource' in the SCOPE column.

Once you've assigned the role, the web application will have access to the Service Bus entities under the defined

scope.

## Run the app

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

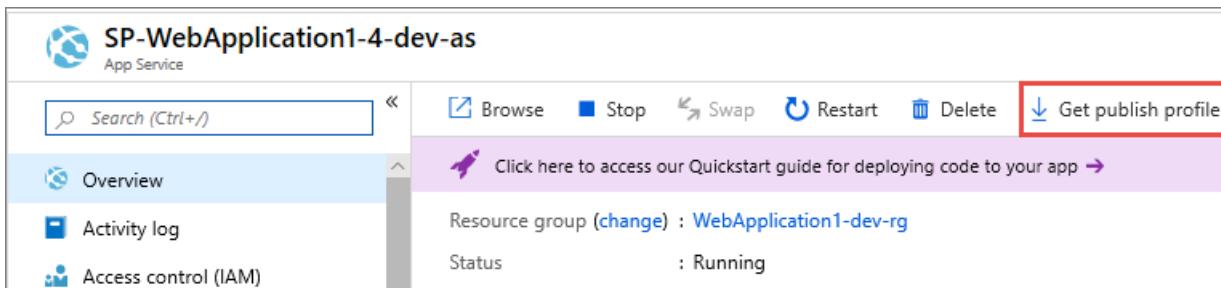
The Default.aspx page is your landing page. The code can be found in the Default.aspx.cs file. The result is a minimal web application with a few entry fields, and with **send** and **receive** buttons that connect to Service Bus to either send or receive messages.

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 identity with the

```
var msiTokenProvider = TokenProvider.CreateManagedIdentityTokenProvider();
```

 call. As such, there are no secrets to retain and use. The flow of the managed identity context to Service Bus and the authorization handshake are automatically handled by the token provider. It is a simpler model than using SAS.

After you make these changes, publish and run the application. You can obtain the correct publishing data easily by downloading and then importing 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

- The managed identity works only inside the Azure environment, on App services, Azure VMs, and scale sets. For .NET applications, the Microsoft.Azure.Services.AppAuthentication library, which is used by the Service Bus NuGet package, provides an abstraction over this protocol and supports a local development experience. This library also allows you to test your code locally on your development machine, using your user account from Visual Studio, Azure CLI 2.0 or Active Directory Integrated Authentication. For more on local development options with this library, see [Service-to-service authentication to Azure Key Vault using .NET](#).
- Currently, managed identities do not work with App Service deployment slots.

## Next steps

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

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

# Authenticate and authorize an application with Azure Active Directory to access Azure Service Bus entities

9/13/2019 • 9 minutes to read • [Edit Online](#)

Azure Service Bus supports using Azure Active Directory (Azure AD) to authorize requests to Service Bus entities (queues, topics, subscriptions, or filters). With Azure AD, you can use role-based access control (RBAC) to grant permissions to a security principal, which may be a user, group, or application service principal. To learn more about roles and role assignments, see [Understanding the different roles](#).

## Overview

When a security principal (a user, group, or application) attempts to access a Service Bus entity, the request must be authorized. With Azure AD, access to a resource is a two-step process.

1. First, the security principal's identity is authenticated, and an OAuth 2.0 token is returned. The resource name to request a token is `https://servicebus.azure.net`.
2. Next, the token is passed as part of a request to the Service Bus service to authorize access to the specified resource.

The authentication step requires that an application request contains an OAuth 2.0 access token at runtime. If an application is running within an Azure entity such as an Azure VM, a virtual machine scale set, or an Azure Function app, it can use a managed identity to access the resources. To learn how to authenticate requests made by a managed identity to Service Bus service, see [Authenticate access to Azure Service Bus resources with Azure Active Directory and managed identities for Azure Resources](#).

The authorization step requires that one or more RBAC roles be assigned to the security principal. Azure Service Bus provides RBAC roles that encompass sets of permissions for Service Bus resources. The roles that are assigned to a security principal determine the permissions that the principal will have. To learn more about assigning RBAC roles to Azure Service Bus, see [Built-in RBAC roles for Azure Service Bus](#).

Native applications and web applications that make requests to Service Bus can also authorize with Azure AD. This article shows you how to request an access token and use it to authorize requests for Service Bus resources.

## Assigning RBAC roles for access rights

Azure Active Directory (Azure AD) authorizes access rights to secured resources through [role-based access control \(RBAC\)](#). Azure Service Bus defines a set of built-in RBAC roles that encompass common sets of permissions used to access Service Bus entities and you can also define custom roles for accessing the data.

When an RBAC role is assigned to an Azure AD security principal, Azure grants access to those resources for that security principal. Access can be scoped to the level of subscription, the resource group, or the Service Bus namespace. An Azure AD security principal may be a user, a group, an application service principal, or a [managed identity for Azure resources](#).

## Built-in RBAC roles for Azure Service Bus

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.

Azure provides the below built-in RBAC roles for authorizing access to a Service Bus namespace:

- [Azure Service Bus Data Owner](#): Enables data access to Service Bus namespace and its entities (queues, topics, subscriptions, and filters)
- [Azure Service Bus Data Sender](#): Use this role to give send access to Service Bus namespace and its entities.
- [Azure Service Bus Data Receiver](#): Use this role to give receiving access to Service Bus namespace and its entities.

## Resource scope

Before you assign an RBAC role to a security principal, determine the scope of access that the security principal should have. Best practices dictate that it's always best to grant only the narrowest possible scope.

The following list describes the levels at which you can scope access to Service Bus resources, starting with the narrowest scope:

- **Queue, topic, or subscription**: Role assignment applies to the specific Service Bus entity. Currently, the Azure portal doesn't support assigning users/groups/managed identities to Service Bus RBAC roles at the subscription level.
- **Service Bus namespace**: Role assignment spans the entire topology of Service Bus under the namespace and to the consumer group associated with it.
- **Resource group**: Role assignment applies to all the Service Bus resources under the resource group.
- **Subscription**: Role assignment applies to all the Service Bus resources in all of the resource groups in the subscription.

### NOTE

Keep in mind that RBAC role assignments may take up to five minutes to propagate.

For more information about how built-in roles are defined, see [Understand role definitions](#). For information about creating custom RBAC roles, see [Create custom roles for Azure Role-Based Access Control](#).

## Assign RBAC roles using the Azure portal

To learn more on managing access to Azure resources using RBAC and the Azure portal, see [this article](#).

After you've determined the appropriate scope for a role assignment, navigate to that resource in the Azure portal. Display the access control (IAM) settings for the resource, and follow these instructions to manage role assignments:

### NOTE

The steps described below assigns a role to your Service Bus namespace. You can follow the same steps to assign a role to other supported scopes (resource group, subscription, etc.).

1. In the [Azure portal](#), navigate to your Service Bus namespace. Select **Access Control (IAM)** on the left menu to display access control settings for the namespace. If you need to create a Service Bus namespace, follow instructions from this article: [Create a Service Bus Messaging namespace](#).

The screenshot shows the Azure portal's Access control (IAM) blade for a Service Bus Namespace named 'spsbusns'. The left sidebar contains navigation links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Shared access policies, Scale, Geo-Recovery, Properties, Locks, Export template), Entities (Queues), Monitoring (Alerts, Metrics, Diagnostic settings, Logs), and Help & support.

The main content area has tabs for Check access, Role assignments, Deny assignments, Classic administrators, and Roles. The 'Role assignments' tab is selected. A callout box highlights the 'Add a role assignment' section, which includes a description: 'Grant access to resources at this scope by assigning a role to a user, group, service principal, or managed identity.' It features an 'Add' button and a 'Learn more' link.

2. Select the **Role assignments** tab to see the list of role assignments. Select the **Add** button on the toolbar and then select **Add role assignment**.

The screenshot shows the 'Add role assignment' page. The toolbar includes a red box around the '+ Add' button. The main area displays instructions: 'Manage access to Azure resources for users, groups, service principals and managed identities at this scope by creating role assignments.' It includes fields for Name (Search by name or email), Type (dropdown), Scope (dropdown set to All scopes), Group by (dropdown), and Role (dropdown set to 0 selected).

3. On the **Add role assignment** page, do the following steps:

- Select the **Service Bus role** that you want to assign.
- Search to locate the **security principal** (user, group, service principal) to which you want to assign the role.
- Select **Save** to save the role assignment.

**Add role assignment**

Role <i>i</i>	Azure Service Bus Data Owner
Assign access to <i>i</i>	Azure AD user, group, or service principal
Select <i>i</i>	Search by name or email address
 Jane Doe janedoe@<domain name>.onmicrosoft.c...	
Selected members:	
 John Doe john doe@<domain name>.on...	
<a href="#">Save</a> <a href="#">Discard</a>	

- d. The identity to whom you assigned the role appears listed under that role. For example, the following image shows that Azure-users is in the Azure Service Bus Data Owner role.

<a href="#">+ Add</a> <a href="#">Edit columns</a> <a href="#">Refresh</a>   <a href="#">Remove</a> <a href="#">Got feedback?</a>				
<a href="#">Check access</a> <a href="#">Role assignments</a> <a href="#">Deny assignments</a> <a href="#">Classic administrators</a> <a href="#">Roles</a>				
Manage access to Azure resources for users, groups, service principals and managed identities at this scope by creating role assignments. <a href="#">Learn more</a>				
Name <i>i</i>	Type <i>i</i>	Role <i>i</i>	Scope <i>i</i>	Group by <i>i</i>
<input type="text"/> Search by name or email	All	3 selected	All scopes	Role
4 items (3 Users, 1 Service Principals)	NAME	TYPE	ROLE	SCOPE
<b>AZURE SERVICE BUS DATA OWNER (PREVIEW)</b>				
 John Doe john doe@<domain name>.onmicrosoft.... User				
Azure Service Bus Data Owner <a href="#">This resource</a>				

You can follow similar steps to assign a role scoped to a resource group, or a subscription. Once you define the role and its scope, you can test this behavior with the [samples on GitHub](#).

## Authenticate from an application

A key advantage of using Azure AD with Service Bus is that your credentials no longer need to be stored in your code. Instead, you can request an OAuth 2.0 access token from Microsoft identity platform. Azure AD authenticates the security principal (a user, a group, or service principal) running the application. If authentication succeeds, Azure AD returns the access token to the application, and the application can then use the access token to authorize requests to Azure Service Bus.

Following sections shows you how to configure your native application or web application for authentication with Microsoft identity platform 2.0. For more information about Microsoft identity platform 2.0, see [Microsoft identity platform \(v2.0\) overview](#).

For an overview of the OAuth 2.0 code grant flow, see [Authorize access to Azure Active Directory web applications using the OAuth 2.0 code grant flow](#).

### Register your application with an Azure AD tenant

The first step in using Azure AD to authorize Service Bus entities is registering your client application with an Azure AD tenant from the [Azure portal](#). When you register your client application, you supply information about the application to AD. Azure AD then provides a client ID (also called an application ID) that you can use to

associate your application with Azure AD runtime. To learn more about the client ID, see [Application and service principal objects in Azure Active Directory](#).

The following images show steps for registering a web application:

The screenshot shows the 'Register an application' wizard. Step 1: 'Name'. It asks for the application name and says it's the user-facing display name. A text input field is provided. Step 2: 'Supported account types'. It asks who can use the application or access the API. Three options are shown: 'Accounts in this organizational directory only (Default Directory only - Single tenant)' (selected), 'Accounts in any organizational directory (Any Azure AD directory - Multitenant)', and 'Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)'. A 'Help me choose...' link is available. Step 3: 'Redirect URI (optional)'. It asks for the authentication response URL. A dropdown menu shows 'Web' selected, and a text input field shows 'e.g. https://myapp.com/auth'. Step 4: 'Agreement'. It states that by proceeding, you agree to the Microsoft Platform Policies. A blue 'Register' button is at the bottom.

Dashboard > App registrations > Register an application

## Register an application

\* Name

The user-facing display name for this application (this can be changed later).

Supported account types

Who can use this application or access this API?

Accounts in this organizational directory only (Default Directory only - Single tenant)

Accounts in any organizational directory (Any Azure AD directory - Multitenant)

Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)

[Help me choose...](#)

Redirect URI (optional)

We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

Web  e.g. <https://myapp.com/auth>

By proceeding, you agree to the Microsoft Platform Policies [\[link\]](#)

### NOTE

If you register your application as a native application, you can specify any valid URI for the Redirect URI. For native applications, this value does not have to be a real URL. For web applications, the redirect URI must be a valid URI, because it specifies the URL to which tokens are provided.

After you've registered your application, you'll see the **Application (client) ID** under **Settings**:

Dashboard > App registrations > mywebappregistration

**mywebappregistration**

Search (Ctrl+ /) Delete Endpoints

Welcome to the new and improved App registrations. Looking to learn how it's changed from App registrations (Legacy)? →

Display name: mywebappregistration  
Application (client) ID: <Application ID>  
Supported account types: My organization only  
Directory (tenant) ID: <Directory ID>  
Object ID: <Object ID>  
Redirect URIs: 1 web, 0 public client  
Managed application in local directory: mywebappregistration

Quickstart Overview Manage API permissions  
Branding Authentication Certificates & secrets Expose an API Owners Roles and administrators (Previous)  
Manifest Support + Troubleshooting Troubleshooting New support request

Call APIs Documentation

Microsoft identity platform  
Authentication scenarios  
Authentication libraries  
Code samples  
Microsoft Graph  
Glossary  
Help and Support

Build more powerful apps with rich user and business data from Microsoft services and your own company's data sources.

[View API Permissions](#)

For more information about registering an application with Azure AD, see [Integrating applications with Azure Active Directory](#).

### IMPORTANT

Make note of the **TenantId** and the **ApplicationId**. You will need these values to run the application.

### Create a client secret

The application needs a client secret to prove its identity when requesting a token. To add the client secret, follow these steps.

1. Navigate to your app registration in the Azure portal if you aren't already on the page.
2. Select **Certificates & secrets** on the left menu.
3. Under **Client secrets**, select **New client secret** to create a new secret.

Dashboard > App registrations > mywebappregistration - Certificates & secrets

**mywebappregistration - Certificates & secrets**

Search (Ctrl+ /)

Credentials enable applications to identify themselves to the authentication service when receiving tokens at a web addressable location (using an HTTPS scheme). For a higher level of assurance, we recommend using a certificate (instead of a client secret) as a credential.

Certificates

Certificates can be used as secrets to prove the application's identity when requesting a token. Also can be referred to as public keys.

[Upload certificate](#)

No certificates have been added for this application.

THUMPRINT	START DATE	EXPIRES

Client secrets

A secret string that the application uses to prove its identity when requesting a token. Also can be referred to as application password.

[+ New client secret](#)

DESCRIPTION	EXPIRES	VALUE

No client secrets have been created for this application.

4. Provide a description for the secret, and choose the wanted expiration interval, and then select **Add**.

Add a client secret

Description	ServiceBusClientSecret
Expires	<input type="radio"/> In 1 year <input type="radio"/> In 2 years <input checked="" type="radio"/> Never
<input type="button" value="Add"/> <input type="button" value="Cancel"/>	

- Immediately copy the value of the new secret to a secure location. The fill value is displayed to you only once.

Client secrets

A secret string that the application uses to prove its identity when requesting a token. Also can be referred to as application password.

New client secret			
DESCRIPTION	EXPIRES	VALUE	
ServiceBusClientSecret	12/31/2299	<Secret>	 

## Permissions for the Service Bus API

If your 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 <https://servicebus.microsoft.com> for this example, because the sample code already uses that URI.

## Client libraries for token acquisition

Once you've registered your application and granted it permissions to send/receive data in Azure Service Bus, you can add code to your application to authenticate a security principal and acquire OAuth 2.0 token. To authenticate and acquire the token, you can use either one of the [Microsoft identity platform authentication libraries](#) or another open-source library that supports OpenID or Connect 1.0. Your application can then use the access token to authorize a request against Azure Service Bus.

For a list of scenarios for which acquiring tokens is supported, see the [Scenarios](#) section of the [Microsoft Authentication Library \(MSAL\) for .NET](#) GitHub repository.

## Sample on GitHub

See the following sample on GitHub: [Role-base access control for Service Bus](#).

Use the **Client Secret Login** option, not the **Interactive User Login** option. When you use the client secret option, you don't see a pop-up window. The application utilizes the tenant ID and app ID for authentication.

## Run the sample

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 sign in 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. Select **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 RBAC, see [What is role-based access control \(RBAC\)?](#)
- To learn how to assign and manage RBAC role assignments with Azure PowerShell, Azure CLI, or the REST API, see these articles:
  - [Manage role-based access control \(RBAC\) with Azure PowerShell](#)
  - [Manage role-based access control \(RBAC\) with Azure CLI](#)
  - [Manage role-based access control \(RBAC\) with the REST API](#)
  - [Manage role-based access control \(RBAC\) with Azure Resource Manager Templates](#)

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

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

# Security controls for Azure Service Bus Messaging

1/11/2020 • 2 minutes to read • [Edit Online](#)

This article documents the security controls built into Azure Service Bus Messaging.

A security control is a quality or feature of an Azure service that contributes to the service's ability to prevent, detect, and respond to security vulnerabilities.

For each control, we use "Yes" or "No" to indicate whether it is currently in place for the service, "N/A" for a control that is not applicable to the service. We might also provide a note or links to more information about an attribute.

## Network

SECURITY CONTROL	YES/NO	NOTES	DOCUMENTATION
Service endpoint support	Yes (Premium tier only)	VNet service endpoints are supported for <a href="#">Service Bus Premium tier</a> only.	
VNet injection support	No		
Network isolation and firewalling support	Yes (Premium tier only)		
Forced tunneling support	No		

## Monitoring & logging

SECURITY CONTROL	YES/NO	NOTES	DOCUMENTATION
Azure monitoring support (Log analytics, App insights, etc.)	Yes	Supported via <a href="#">Azure Monitor and Alerts</a> .	
Control and management plane logging and audit	Yes	Operations logs are available.	<a href="#">Service Bus diagnostic logs</a>
Data plane logging and audit	No		

## Identity

SECURITY CONTROL	YES/NO	NOTES	DOCUMENTATION
Authentication	Yes	Managed through Azure <a href="#">Active Directory Managed Service Identity</a> .	<a href="#">Service Bus authentication and authorization</a> .
Authorization	Yes	Supports authorization via <a href="#">RBAC</a> and SAS token.	<a href="#">Service Bus authentication and authorization</a> .

## Data protection

SECURITY CONTROL	YES/NO	NOTES	DOCUMENTATION
Server-side encryption at rest: Microsoft-managed keys	Yes for server-side encryption-at-rest by default.		
Server-side encryption at rest: customer-managed keys (BYOK)	Yes.	A customer managed key in Azure KeyVault can be used to encrypt the data on the Service Bus Namespace at rest.	<a href="#">Configure customer-managed keys for encrypting Azure Service Bus data at rest by using the Azure portal</a>
Column level encryption (Azure Data Services)	N/A		
Encryption in transit (such as ExpressRoute encryption, in VNet encryption, and VNet-VNet encryption)	Yes	Supports standard HTTPS/TLS mechanism.	
API calls encrypted	Yes	API calls are made through <a href="#">Azure Resource Manager</a> and HTTPS.	

## Configuration management

SECURITY CONTROL	YES/NO	NOTES	DOCUMENTATION
Configuration management support (versioning of configuration, etc.)	Yes	Supports resource provider versioning through the <a href="#">Azure Resource Manager API</a> .	

## Next steps

- Learn more about the [built-in security controls across Azure services](#).

# Create a Service Bus namespace using the Azure portal

1/24/2020 • 3 minutes to read • [Edit Online](#)

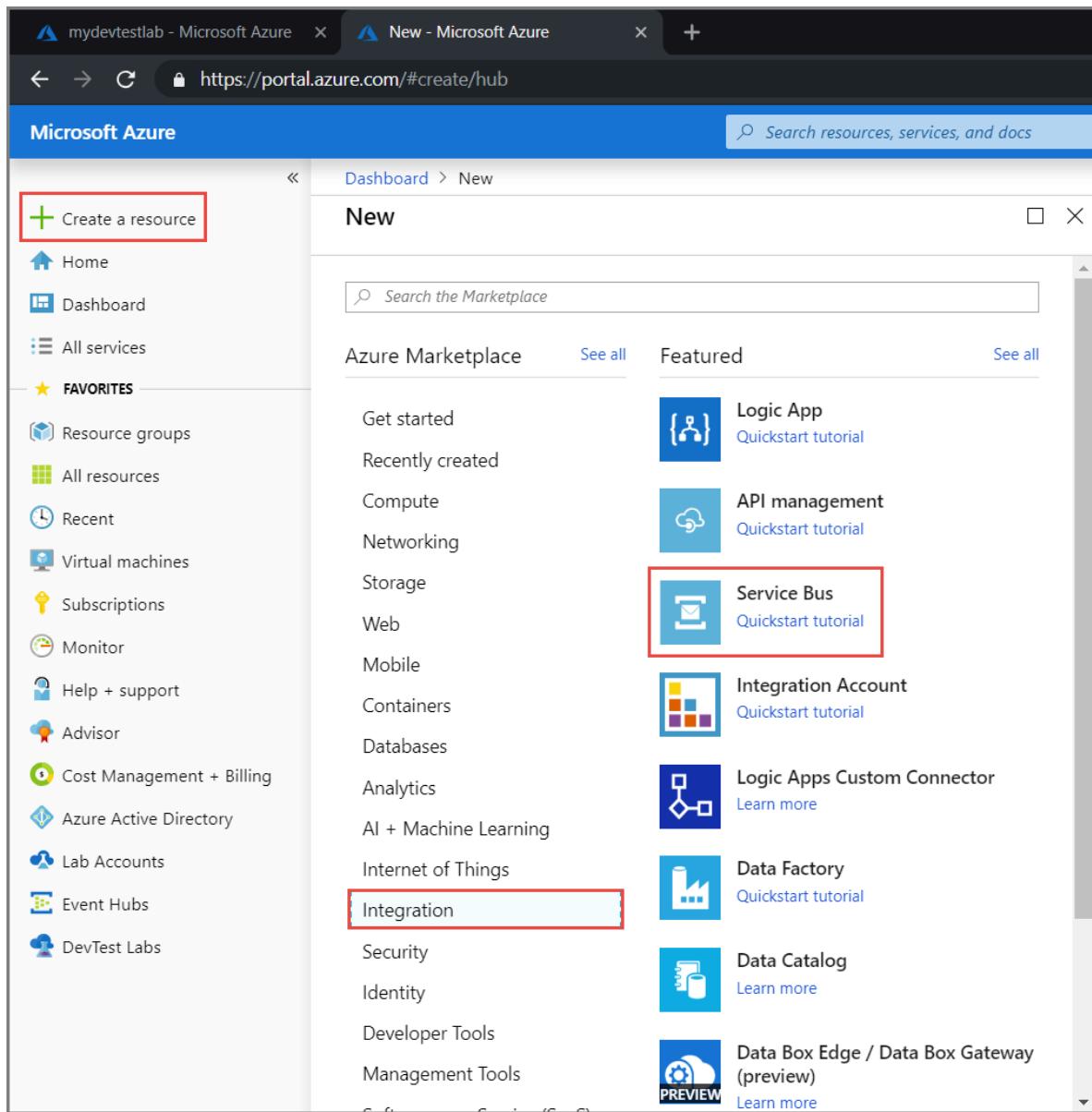
A namespace is a scoping container for all messaging components. Multiple queues and topics can reside within a single namespace, and namespaces often serve as application containers. This article provides instructions for creating a namespace in the Azure portal.

## 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. Sign in to the [Azure portal](#)
2. In the left navigation pane of the portal, select **+ Create a resource**, select **Integration**, and then select **Service Bus**.



3. In the **Create namespace** dialog, do the following steps:

- Enter a **name for the namespace**. The system immediately checks to see if the name is available. For a list of rules for naming namespaces, see [Create Namespace REST API](#).
- Select the pricing tier (Basic, Standard, or Premium) for the namespace. If you want to use [topics and subscriptions](#), choose either Standard or Premium. Topics/subscriptions are not supported in the Basic pricing tier.
- If you selected the **Premium** pricing tier, follow these steps:
  - Specify the number of **messaging units**. The premium tier provides resource isolation at the CPU and memory level so that each workload runs in isolation. This resource container is called a messaging unit. A premium namespace has at least one messaging unit. You can select 1, 2, or 4 messaging units for each Service Bus Premium namespace. For more information, see [Service Bus Premium Messaging](#).
  - Specify whether you want to make the namespace **zone redundant**. The zone redundancy provides enhanced availability by spreading replicas across availability zones within one region at no additional cost. For more information, see [Availability zones in Azure](#).
- For **Subscription**, choose an Azure subscription in which to create the namespace.
- For **Resource group**, choose an existing resource group in which the namespace will live, or create a new one.

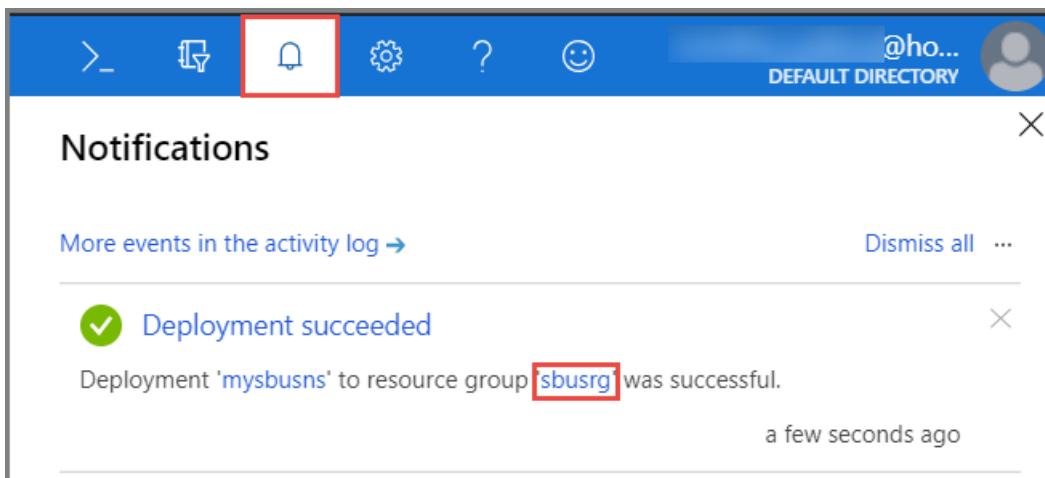
- f. For **Location**, choose the region in which your namespace should be hosted.
- g. Select **Create**. The system now creates your namespace and enables it. You might have to wait several minutes as the system provisions resources for your account.

The screenshot shows the 'Create namespace' dialog box. At the top, there's a 'Service Bus' icon and a close button (X). The form contains the following fields:

- Name:** mysbusns (with a green checkmark)
- Pricing tier:** Standard
- Subscription:** Visual Studio Ultimate with MSDN
- Resource group:** (New) sbusrg (with a dropdown arrow)
- Location:** West US

At the bottom is a large blue 'Create' button.

4. Confirm that the service bus namespace is deployed successfully. To see the notifications, select the **bell icon (Alerts)** on the toolbar. Select the **name of the resource group** in the notification as shown in the image. You see the resource group that contains the service bus namespace.



5. On the **Resource group** page for your resource group, select your **service bus namespace**.

6. You see the home page for your service bus namespace.

## Get the connection string

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 rules with more constrained rights for regular senders and receivers. To copy the primary and secondary keys for your namespace, 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 managing a Service Bus namespace. On the left, there's a navigation sidebar with options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Shared access policies (which is selected and highlighted with a red box), Scale, Geo-Recovery, Properties, Locks, and Automation script. The main content area is titled 'mysbusns - Shared access policies' and shows a table with two columns: 'POLICY' and 'CLAIMS'. A single row is selected, showing 'RootManageSharedAccessKey' in the POLICY column and 'Manage, Send, Listen' in the CLAIMS column. There are search bars at the top of the main content area.

4. In the **Policy: RootManageSharedAccessKey** window, click the copy button next to **Primary Connection String**, 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 configuration dialog for a SAS policy named 'RootManageSharedAccessKey'. At the top, there are buttons for Save, Discard, Delete, and More. Below are three checked checkboxes: 'Manage', 'Send', and 'Listen'. Underneath these are four input fields: 'Primary Key' containing '<Primary key>', 'Secondary Key' containing '<Secondary key>', 'Primary Connection String' containing 'Endpoint=sb://mysbusns.servicebus.windows.net/S...', and 'Secondary Connection String' containing 'Endpoint=sb://mysbusns.servicebus.windows.net/S...'. Each string field has a small blue copy icon to its right. The 'Primary Connection String' field and its copy icon are 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 the Service Bus [GitHub samples](#), which show some of the more advanced features of Service Bus messaging.

# Service Bus queues, topics, and subscriptions

1/24/2020 • 7 minutes 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, receivers typically receive and process messages in the order in which they were added to the queue, and only one message consumer receives and processes each message. 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 capability 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.

### Create queues

You create queues using the [Azure portal](#), [PowerShell](#), [CLI](#), or [Resource Manager templates](#). You then send and receive messages using a [QueueClient](#) object.

To quickly learn how to create a queue, then send and receive messages to and from the queue, see the [quickstarts](#) for each method. For a more in-depth tutorial on how to use queues, see [Get started with Service Bus queues](#).

For a working sample, see the [BasicSendReceiveUsingQueueClient sample](#) on GitHub.

### Receive modes

You can specify two different modes in which Service Bus receives messages: *ReceiveAndDelete* or *PeekLock*. In the [ReceiveAndDelete](#) mode, the receive operation is single-shot; that is, when Service Bus receives the

request from the consumer, it marks the message as being consumed and returns it to the consumer application. **ReceiveAndDelete** mode is the simplest model and works best for scenarios in which the application can tolerate not processing a message if a failure occurs. To understand this scenario, 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 [CompleteAsync](#) on the received message. When Service Bus sees the [CompleteAsync](#) call, it marks the message as being consumed.

If the application is unable to process the message for some reason, it can call the [AbandonAsync](#) method on the received message (instead of [CompleteAsync](#)). This method 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 [AbandonAsync](#) operation by default).

In the event that the application crashes after processing the message, but before the [CompleteAsync](#) request is issued, 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. 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 feature 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 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 feature 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.

### Create topics and subscriptions

Creating a topic is similar to creating a queue, as described in the previous section. You then send messages using the [TopicClient](#) class. To receive messages, you create one or more subscriptions to the topic. 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 a full working example, see the [BasicSendReceiveUsingTopicSubscriptionClient sample](#) on GitHub.

## Rules and actions

In many scenarios, messages that have specific characteristics must be processed in different ways. To enable this processing, 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 filtering 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.

For a full working example, see the [TopicSubscriptionWithRuleOperationsSample sample](#) on GitHub.

For more information about possible filter values, see the documentation for the [SqlFilter](#) and [SqlRuleAction](#) classes.

## Next steps

For more information and examples of using Service Bus messaging, see the following advanced topics:

- [Service Bus messaging overview](#)
- [Quickstart: Send and receive messages using the Azure portal and .NET](#)
- [Tutorial: Update inventory using Azure portal and topics/subscriptions](#)

# Messages, payloads, and serialization

1/24/2020 • 9 minutes 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
<a href="#">ContentType</a> (content-type)	Optionally describes the payload of the message, with a descriptor following the format of RFC2045, Section 5; for example, <code>application/json</code> .
<a href="#">CorrelationId</a> (correlation-id)	Enables an application to specify a context for the message for the purposes of correlation; for example, reflecting the <b>MessageId</b> of a message that is being replied to.
<a href="#">DeadLetterSource</a>	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.
<a href="#">DeliveryCount</a>	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.
<a href="#">EnqueuedSequenceNumber</a>	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.
<a href="#">EnqueuedTimeUtc</a>	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
<a href="#">ExpiresAtUtc</a> (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 <b>TimeToLive</b> property and this property is computed from <code>EnqueuedTimeUtc+TimeToLive</code> . This property is read-only.
<a href="#">ForcePersistence</a>	For queues or topics that have the <a href="#">EnableExpress</a> 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.
<a href="#">Label</a> (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.
<a href="#">LockedUntilUtc</a>	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 <a href="#">DeliveryCount</a> is incremented and the message is again available for retrieval. This property is read-only.
<a href="#">LockToken</a>	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 <a href="#">Deferral</a> API and, with that, take the message out of the regular delivery state flow. This property is read-only.
<a href="#">MessageId</a> (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 <a href="#">duplicate detection</a> feature identifies and removes second and further submissions of messages with the same <b>MessageId</b> .
<a href="#">PartitionKey</a>	For <a href="#">partitioned entities</a> , 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 <b>SessionId</b> property overrides this value.
<a href="#">ReplyTo</a> (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.
<a href="#">ReplyToSessionId</a> (reply-to-group-id)	This value augments the <b>ReplyTo</b> information and specifies which <b>SessionId</b> should be set for the reply when sent to the reply entity.
<a href="#">ScheduledEnqueueTimeUtc</a>	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
<a href="#">SequenceNumber</a>	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.
<a href="#">SessionId</a> (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.
<a href="#">Size</a>	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.
<a href="#">State</a>	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.
<a href="#">TimeToLive</a>	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 <b>EnqueueTimeUtc</b> . When not set explicitly, the assumed value is the <b>DefaultTimeToLive</b> for the respective queue or topic. A message-level <b>TimeToLive</b> value cannot be longer than the entity's <b>DefaultTimeToLive</b> setting. If it is longer, it is silently adjusted.
<a href="#">To (to)</a>	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.
<a href="#">ViaPartitionKey</a>	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<T>()** 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 queues, topics, and subscriptions](#)
- [Get started with Service Bus queues](#)

- How to use Service Bus topics and subscriptions

# Message transfers, locks, and settlement

1/24/2020 • 8 minutes 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);

```

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);

```

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*.

### ReceiveAndDelete

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.

### PeekLock

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.

#### IMPORTANT

It is important to note that the lock that PeekLock acquires on the message is volatile and may be lost in the following conditions

- Service Update
- OS update
- Changing properties on the entity (Queue, Topic, Subscription) while holding the lock.

When the lock is lost, Azure Service Bus will generate a `LockLostException` which will be surfaced on the client application code. In this case, the client's default retry logic should automatically kick in and retry the operation.

## Next steps

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

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

# Message sequencing and timestamps

1/24/2020 • 2 minutes 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 queues, topics, and subscriptions](#)

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

# Message expiration (Time to Live)

1/24/2020 • 4 minutes to read • [Edit Online](#)

The payload in 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](#)). The time-to-live (TTL) setting on a brokered message is not enforced when there are no clients actively listening.

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.

### NOTE

The default [TimeToLive](#) value for a brokered message is [TimeSpan.MaxValue](#) if not otherwise specified.

For messaging entities (queues and topics), the default expiration time is also [TimeSpan.MaxValue](#) for Service Bus standard and premium tiers. For the basic tier, the default expiration time is 14 days.

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. This property is set to the duration for which an entity must be idle (unused) before it is automatically deleted. The minimum value for this property is 5.

The `autoDeleteOnIdle` property must be set through an Azure Resource Manager operation or via the .NET Framework client [NamespaceManager](#) APIs. You can't set it in the portal.

## Idleness

Here's what considered idleness of entities (queues, topics, and subscriptions):

- Queues
  - No sends
  - No receives
  - No updates to the queue
  - No scheduled messages
  - No browse/peek
- Topics
  - No sends
  - No updates to the topic
  - No scheduled messages
- Subscriptions
  - No receives
  - No updates to the subscription
  - No new rules added to the subscription
  - No browse/peek

## Next steps

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

- [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

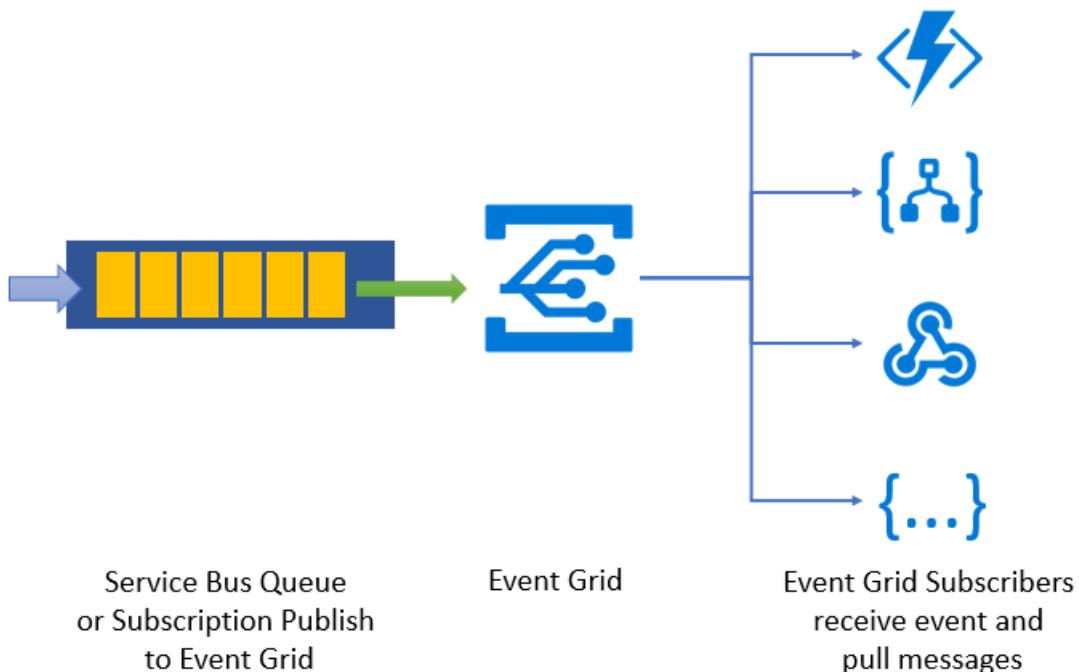
1/27/2020 • 5 minutes to read • [Edit Online](#)

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.



## NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

## Verify that you have contributor access

Go to your Service Bus namespace, and then select **Access control (IAM)**, and select **Role assignments** tab.

Verify that you have the contributor access to the namespace.

## 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 an Event Grid subscription:

The screenshot shows the Azure Service Bus Namespace Events page. On the left, there's a navigation menu with options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events (selected), Settings, Properties, Locks, Automation script, Entities, and Queues. The main area has a search bar and a 'Event Subscription' button with a red box around it. Below that are tabs for 'Get Started' and 'Event Subscriptions' (selected). It shows metrics for General and Errors over the last 30 days. A chart displays event counts at specific times: PUBLISHED EVENTS (0 at 16:45), UNMATCHED EVENTS (0 at 17:00), and FAILED EVENTS (SUM) (-- at 17:15 and 17:30). A table lists one subscription: SBSsubscription, which is a StorageQueue endpoint type with no filters applied.

NAME	ENDPOINT TYPE	PREFIX FILTER	SUFFIX FILTER	EVENT TYPES
SBSsubscription	StorageQueue			All

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

\* Name

 ✓

Subscribe to all event types

Subscriber Type

Web Hook ▼

\* Subscriber Endpoint

 ✓

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 "<Azure subscription name>

namespaceid=$(az resource show --namespace Microsoft.ServiceBus --resource-type namespaces --name "<service bus namespace>" --resource-group "<resource group that contains the service bus namespace>" --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>"
```

If you are using BASH

## 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-AzAccount

Select-AzSubscription -SubscriptionName "<YOUR SUBSCRIPTION NAME>

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

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

New-AzEventGridSubscription -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](#).

# Tutorial: Respond to Azure Service Bus events received via Azure Event Grid by using Azure Functions and Azure Logic Apps

12/11/2019 • 7 minutes to read • [Edit Online](#)

In this tutorial, you learn how to respond to Azure Service Bus events that are received via Azure Event Grid by using Azure Functions and Azure Logic Apps. You'll do the following steps:

- Create a test Azure function for debugging and viewing the initial flow of events from the Event Grid.
- Create an Azure function to receive and process Azure Service Bus messages based on Event Grid events.
- Create a logic app to respond to Event Grid events

After you create the Service Bus, Event Grid, Azure Functions, and Logic Apps artifacts, you do the following actions:

1. Send messages to a Service Bus topic.
2. Verify that the subscriptions to the topic received those messages
3. Verify that the function or logic app that subscribed for the event has received the event.

## Create a Service Bus namespace

Follow instructions in this tutorial: [Quickstart: Use the Azure portal to create a Service Bus topic and subscriptions to the topic](#) to do the following tasks:

- Create a **premium** Service Bus namespace.
- Get the connection string.
- Create a Service Bus topic.
- Create two subscriptions to the topic.

## Prepare a sample application to send messages

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're 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 Service Bus topic name and the connection string you got from the previous step:

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

5. Build and run the program to send test messages to the Service Bus topic.

## Set up a test function on Azure

Before you work through the entire scenario, set up at least a small test function, which you can use to debug and observe the events that are flowing. Follow instructions in the [Create your first function in the Azure portal](#) article to do the following tasks:

1. Create a function app.
2. Create an HTTP triggered function.

Then, do the following steps:

- [Azure Functions V2](#)
- [Azure Functions V1](#)

1. Expand **Functions** in the tree view, and select your function. Replace the code for the function with the following code:

```
#r "Newtonsoft.Json"

using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;

public static async Task<IActionResult> Run(HttpContext req, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");
    var content = req.Body;
    string jsonContent = await new StreamReader(content).ReadToEndAsync();
    log.LogInformation($"Received Event with payload: {jsonContent}");

    IEnumerable<string> headerValues;
    headerValues = req.Headers.GetCommaSeparatedValues("Aeg-Event-Type");

    if (headerValues.Count() != 0)
    {
        var validationHeaderValue = headerValues.FirstOrDefault();
        if(validationHeaderValue == "SubscriptionValidation")
        {
            var events = JsonConvert.DeserializeObject<GridEvent[]>(jsonContent);
            var code = events[0].Data["validationCode"];
            log.LogInformation("Validation code: {code}");
            return (ActionResult) new OkObjectResult(new { validationResponse = code });
        }
    }

    return jsonContent == null
        ? new BadRequestObjectResult("Please pass a name on the query string or in the request body")
        : (ActionResult) new OkObjectResult($"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; }
}
```

2. Select **Save and run**.

```

myfunctionapp0514 - HttpTriggerCSharp1
Function Apps

All subscriptions
Function Apps
myfunctionapp0514
Functions
HttpTriggerCSharp1
Integrate
Manage
Monitor
Proxies
Slots (preview)

run.csx Save Run </> Get function URL

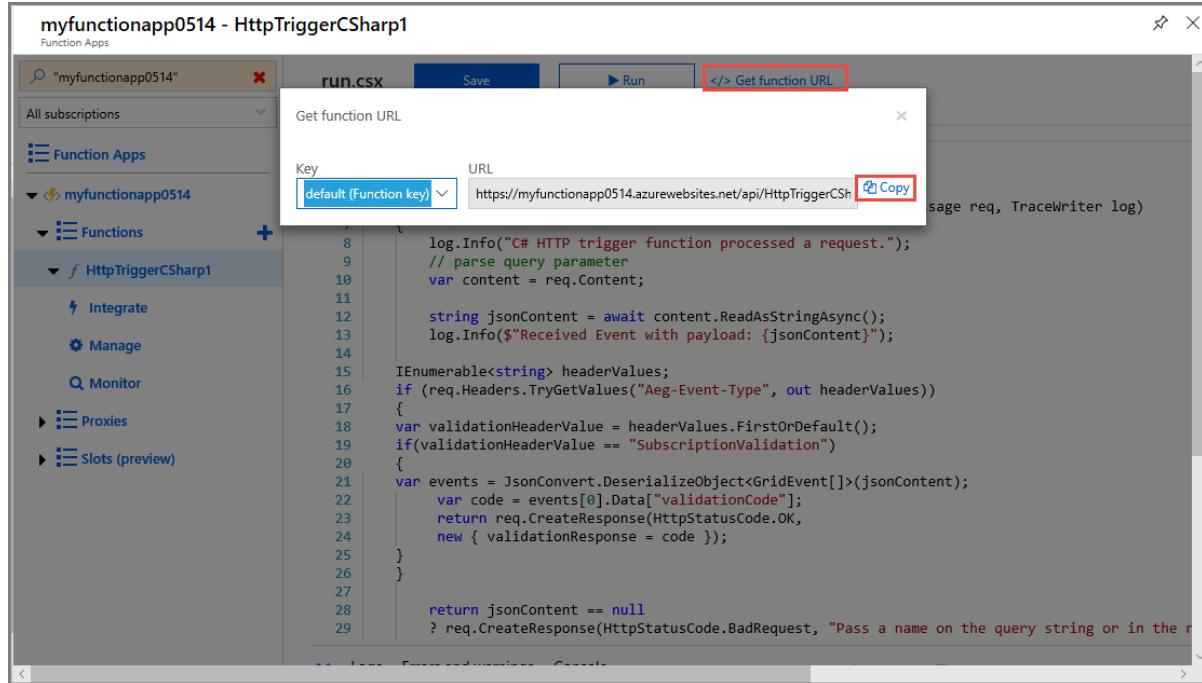
1 #r "Newtonsoft.Json"
2 using System.Net;
3 using Newtonsoft.Json;
4 using Newtonsoft.Json.Linq;
5
6 public static async Task<HttpResponseMessage> Run(HttpRequestMessage req, TraceWriter log)
7 {
8     log.Info("C# HTTP trigger function processed a request.");
9     // parse query parameter
10    var content = req.Content;
11
12    string jsonContent = await content.ReadAsStringAsync();
13    log.Info($"Received Event with payload: {jsonContent}");
14
15    IEnumerable<string> headerValues;
16    if (req.Headers.TryGetValues("Aeg-Event-Type", out headerValues))
17    {
18        var validationHeaderValue = headerValues.FirstOrDefault();
19        if(validationHeaderValue == "SubscriptionValidation")
20        {
21            var events = JsonConvert.DeserializeObject<GridEvent[]>(jsonContent);
22            var code = events[0].Data["ValidationCode"];
23            return req.CreateResponse(HttpStatusCode.OK,
24                new { validationResponse = code });
25        }
26    }
27
28
29    return jsonContent == null
? req.CreateResponse(HttpStatusCode.BadRequest, "Pass a name on the query string or in the request body")

```

Logs Errors and warnings Console

2019-05-14T17:41:50 Welcome, you are now connected to log-streaming service.  
2019-05-14T17:42:07.922 [Info] Script for function 'HttpTriggerCSharp1' changed. Reloading.  
2019-05-14T17:42:09.109 [Info] Compilation succeeded.  
2019-05-14T17:42:09.234 [Info] Function started (Id=e6c50c0f-0341-40d2-8b9b-79d86c164062)  
2019-05-14T17:42:09.984 [Info] C# HTTP trigger function processed a request.  
2019-05-14T17:42:09.984 [Info] Received Event with payload:  
{"name": "Azure"}  
2019-05-14T17:42:09.984 [Info] Function completed (Success, Id=e6c50c0f-0341-40d2-8b9b-79d86c164062, Duration=755ms)

### 3. Select **Get function URL** and note down the URL.

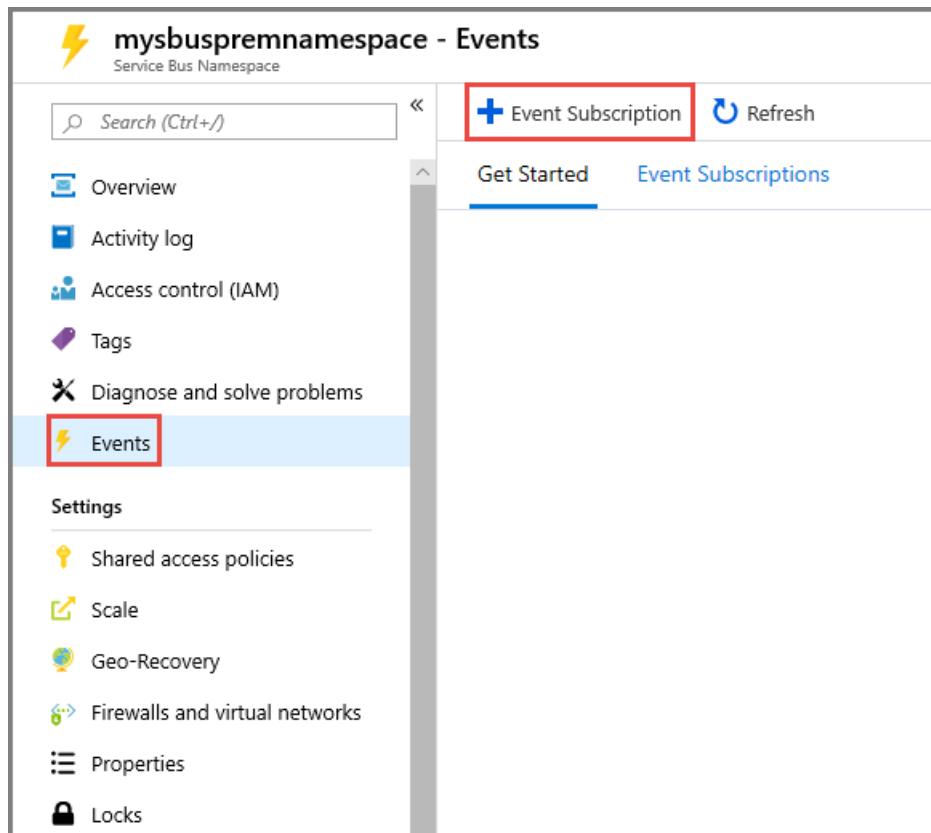


## Connect the function and namespace via Event Grid

In this section, you tie together the function and the Service Bus namespace by using the Azure portal.

To create an Azure Event Grid subscription, follow these steps:

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



2. Select **+ Event Subscription** on the toolbar.
3. On the **Create Event Subscription** page, do the following steps:
  - a. Enter a **name** for the subscription.
  - b. Select **Web Hook** for **Endpoint Type**.

**Create Event Subscription**  
Event Grid

Basic    **Filters**    Additional Features

Event Subscriptions listen for events emitted by the topic resource and send them to the endpoint resource. [Learn more](#)

**EVENT SUBSCRIPTION DETAILS**

Name	spsbusgridsubscription
Event Schema	Event Grid Schema

**TOPIC DETAILS**

Pick a topic resource for which events should be pushed to your destination. [Learn more](#)

Topic Type	Service Bus Namespace
Topic Resource	mysbuspremnamespace

**EVENT TYPES**

Pick which event types get pushed to your destination. [Learn more](#)

Subscribe to all event types

**ENDPOINT DETAILS**

Pick an event handler to receive your events. [Learn more](#)

Endpoint Type	Web Hook (change)
Endpoint	Select an endpoint

**Create**

c. Choose **Select an endpoint**, paste the function URL, and then select **Confirm selection**.

**Create Event Subscription**  
Event Grid

Basic    **Filters**    Additional Features

Event Subscriptions listen for events emitted by the topic resource and send them to the endpoint resource. [Learn more](#)

**EVENT SUBSCRIPTION DETAILS**

Name	spsbusgridsubscription
Event Schema	Event Grid Schema

**TOPIC DETAILS**

Pick a topic resource for which events should be pushed to your destination. [Learn more](#)

Topic Type	Service Bus Namespace
Topic Resource	mysbuspremnamespace

**EVENT TYPES**

Pick which event types get pushed to your destination. [Learn more](#)

Subscribe to all event types

**ENDPOINT DETAILS**

Pick an event handler to receive your events. [Learn more](#)

Endpoint Type	Web Hook (change)
Endpoint	Select an endpoint

**Confirm Selection**

d. Switch to the **Filters** tab, enter the name of the **first subscription** to the Service Bus topic you created earlier, and then select the **Create** button.

**Create Event Subscription**

Event Grid

**Basic** **Filters** **Additional Features**

**SUBJECT FILTERS**

Apply filters to the subject of each event. Only events with matching subjects get delivered. [Learn more](#)

Enable subject filtering

Subject Begins With

Subject Ends With

Case-sensitive subject matching

**ADVANCED FILTERS**

Filter on attributes of each event. Only events that match all filters get delivered. Up to 5 filters can be specified. All string comparisons are case-insensitive. [Learn more](#)

Valid keys for currently selected event schema:

- id, topic, subject, eventtype, dataversion
- Custom properties at most one level inside the data payload, using ":" as the nesting separator. (e.g. data, data.key are valid, data.key.key is not)

KEY	OPERATOR	VALUE
No results		
<a href="#">Add new filter</a>		

**Create**

4. Confirm that you see the event subscription in the list.

**mysbuspremn namespace - Events**

Service Bus Namespace

**Get Started** **Event Subscriptions**

Show metrics: **General** Errors For the last: 1 hour 6 hours 12 hours 1 day 7 days 30 days

Published Events (Sum) mybuspremn namespace/- 1:15 PM  
Unmatched Events (Sum) mybuspremn namespace/- 0 May 14 1:28 PM 1:45 PM 2 PM

**NAME** **ENDPOINT** **PREFIX FILTER** **SUFFIX FILTER** **EVENT TYPES**

spsbusegridsubscription	WebHook			Microsoft.ServiceBus.ActiveMessagesAvailableWithN...
-------------------------	---------	--	--	--

## Send messages to the Service Bus topic

1. Run the .NET C# application, which sends messages to the Service Bus topic.

```
C:\WINDOWS\system32\cmd.exe
=====
Press any key to exit after sending the message.
=====
Sending message: Message 1
Sending message: Message 2
Sending message: Message 3
Sending message: Message 4
Sending message: Message 5
```

2. On the page for your Azure function app, expand **Functions**, expand your **function**, and select **Monitor**.

myfunctionapp0514 - HttpTriggerCSharp1

Function Apps

Visual Studio Ultimate with MSDN

Refresh

Results may be delayed for up to 5 minutes.

Application Insights Instance: myfunctionapp0514insights

Success count in last 30 days: 5

Error count in last 30 days: 0

Query returned 5 items

Run in Application Insights

DATE (UTC)	SUCCESS	RESULT CODE	DURATION (MS)
2019-05-14 18:19:04.006	✓	200	20.423
2019-05-14 18:12:30.932	✓	200	20.2697
2019-05-14 18:05:20.174	✓	200	20.6084
2019-05-14 18:01:20.426	✓	200	78.1282
2019-05-14 17:42:09.234	✓	200	767.9553

## 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.

### Publish a function from Visual Studio

1. In the same Visual Studio solution (**SBEventGridIntegration**) that you opened, select **ReceiveMessagesOnEvent.cs** in the **SBEventGridIntegration** project.
2. Enter your Service Bus connection string in the following code:

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

3. Download the **publish profile** for the function:

- Select your function app.
- Select the **Overview** tab if it isn't already selected.
- Select **Get publish profile** on the toolbar.

The screenshot shows the Azure portal's 'Overview' page for a function app named 'myfunctionapp0514'. The 'Get publish profile' button is highlighted with a red box and the number 3. Other buttons like 'Stop', 'Swap', 'Restart', 'Reset publish profile', 'Download app content', and 'Delete' are also visible.

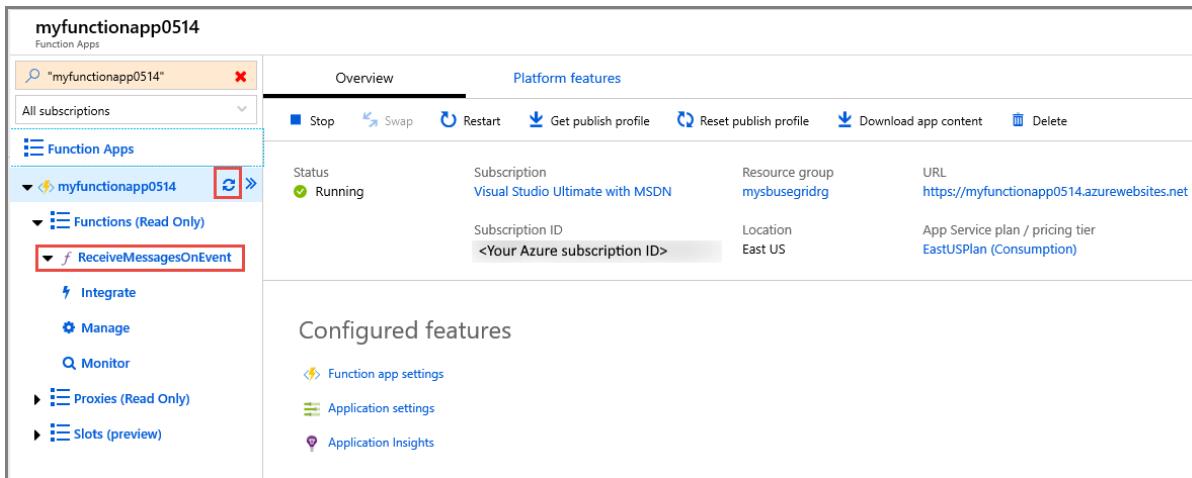
- d. Save the file to your project's folder.
4. In Visual Studio, right-click **SBEventGridIntegration**, and then select **Publish**.
5. Select **Start** on the **Publish** page.
6. On the **Pick a publish target** page, do the following steps, select **Import Profile**.

The screenshot shows the Visual Studio 'Publish' dialog. Under the 'Connected Services' tab, the 'Publish' button is highlighted with a red box and the number 1. In the main area, the 'Start' button is highlighted with a red box and the number 2. A modal window titled 'Pick a publish target' is open, showing options for 'Azure Function App' and 'Folder'. The 'Create New' radio button is selected. At the bottom of the modal, the 'Import Profile...' button is highlighted with a red box and the number 3.

7. Select the **publish profile file** you downloaded earlier.
8. Select **Publish** on the **Publish** page.

The screenshot shows the Visual Studio 'Publish' dialog with the 'Connected Services' tab selected. The 'Publish' button is highlighted with a red box and the number 1. The main area shows a dropdown menu set to 'myfunctionapp0514 - Web Deploy', which is highlighted with a red box and the number 2. The 'Publish' button at the bottom right is highlighted with a red box and the number 3. Below the dropdown, there are fields for Site URL, Configuration, Delete existing files, Username, and Password.

9. Confirm that you see the new Azure function **ReceiveMessagesOnEvent**. Refresh the page if needed.



The screenshot shows the Azure portal's Function Apps blade for the app 'myfunctionapp0514'. In the left sidebar, under 'Functions (Read Only)', the 'ReceiveMessagesOnEvent' function is selected and highlighted with a red border. The main pane displays the 'Overview' tab with the following details:

Status	Subscription	Resource group	URL
Running	Visual Studio Ultimate with MSDN	mysbusegridrg	<a href="https://myfunctionapp0514.azurewebsites.net">https://myfunctionapp0514.azurewebsites.net</a>
Subscription ID <Your Azure subscription ID>		Location	App Service plan / pricing tier EastUSPlan (Consumption)

Below the overview, there is a section titled 'Configured features' which includes links to 'Function app settings', 'Application settings', and 'Application Insights'.

10. Get the URL to the new function and note it down.

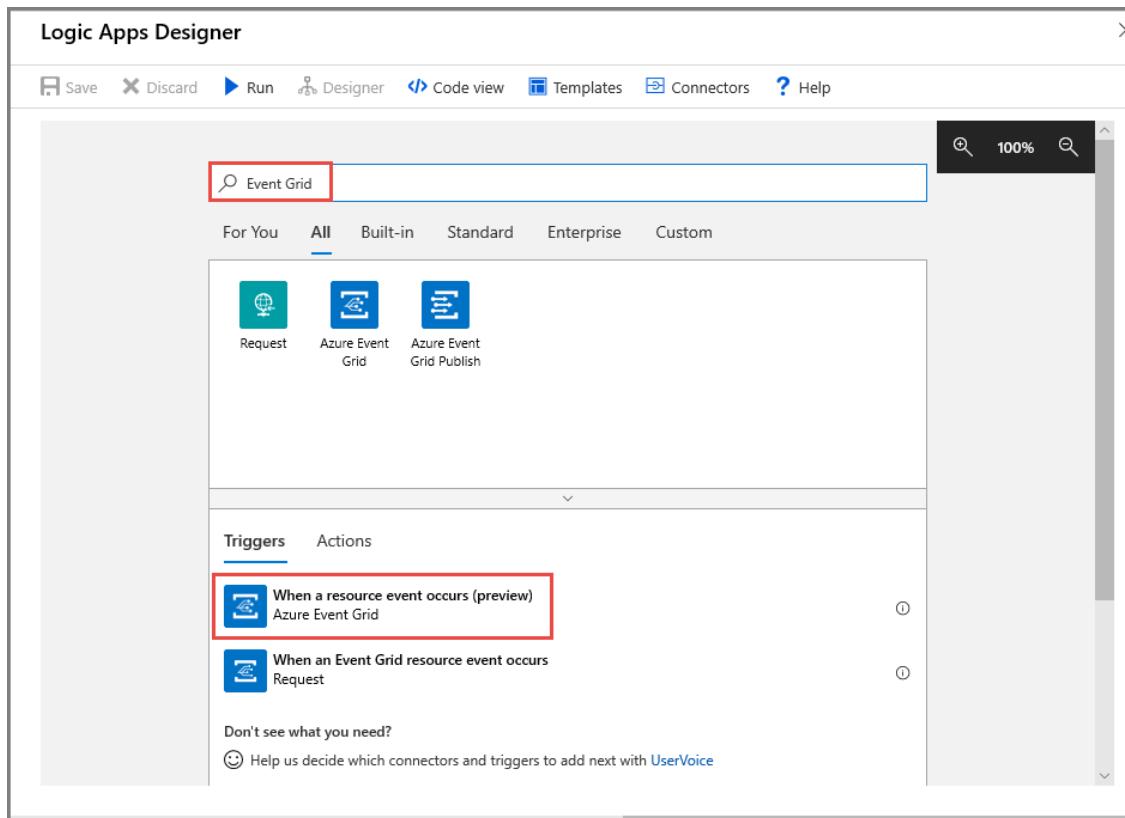
### Event Grid subscription

1. Delete the existing Event Grid subscription:
  - a. On the **Service Bus Namespace** page, select **Events** on the left menu.
  - b. Select the existing event subscription.
  - c. On the **Event Subscription** page, select **Delete**.
2. Follow instructions in the [Connect the function and namespace via Event Grid](#) section to create an Event Grid subscription using the new function URL.
3. Follow instruction in the [Send messages to the Service Bus topic](#) section to send messages to the topic and monitor the function.

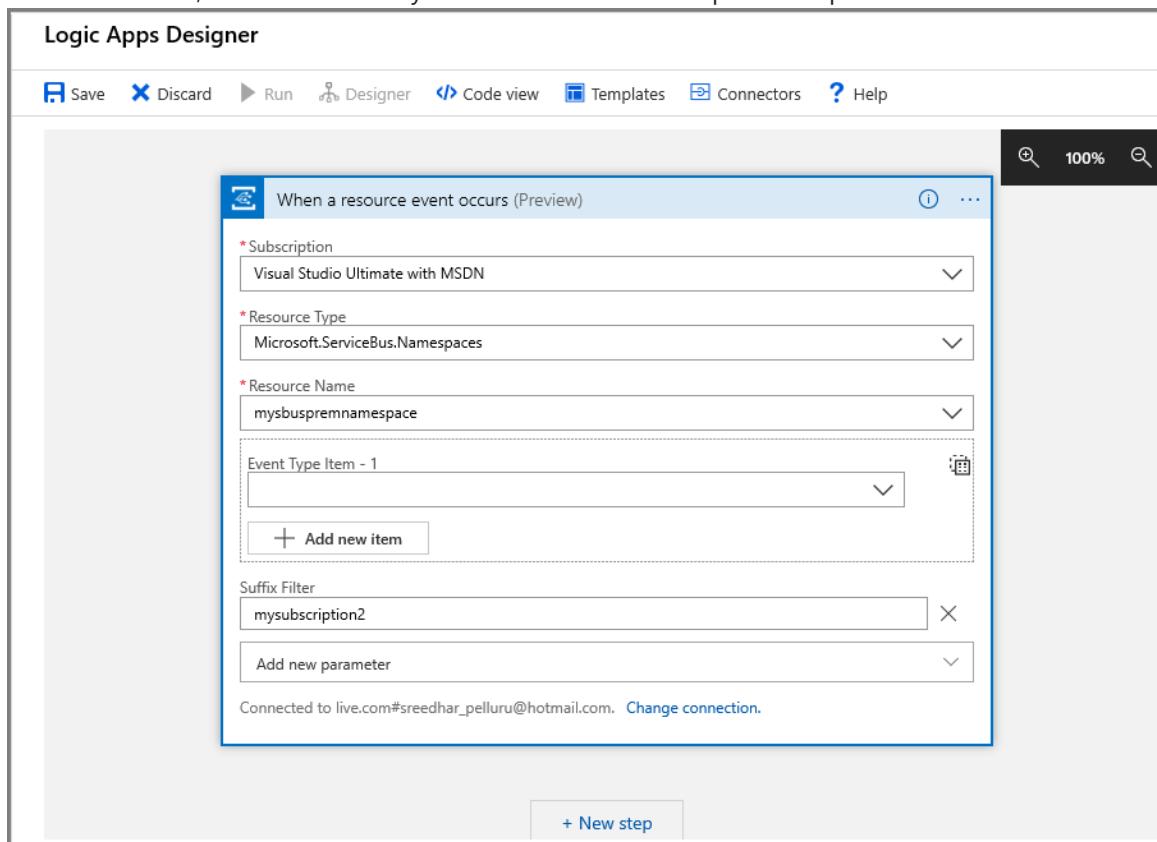
## Receive messages by using Logic Apps

Connect a logic app with Azure Service Bus and Azure Event Grid by following these steps:

1. Create a logic app in the Azure portal.
  - a. Select **+ Create a resource**, select **Integration**, and then select **Logic App**.
  - b. On the **Logic App - Create** page, enter a **name** for the logic app.
  - c. Select your Azure **subscription**.
  - d. Select **Use existing** for the **Resource group**, and select the resource group that you used for other resources (like Azure function, Service Bus namespace) that you created earlier.
  - e. Select the **Location** for the logic app.
  - f. Select **Create** to create the logic app.
2. On the **Logic Apps Designer** page, select **Blank Logic App** under **Templates**.
3. On the designer, do the following steps:
  - a. Search for **Event Grid**.
  - b. Select **When a resource event occurs (preview) - Azure Event Grid**.

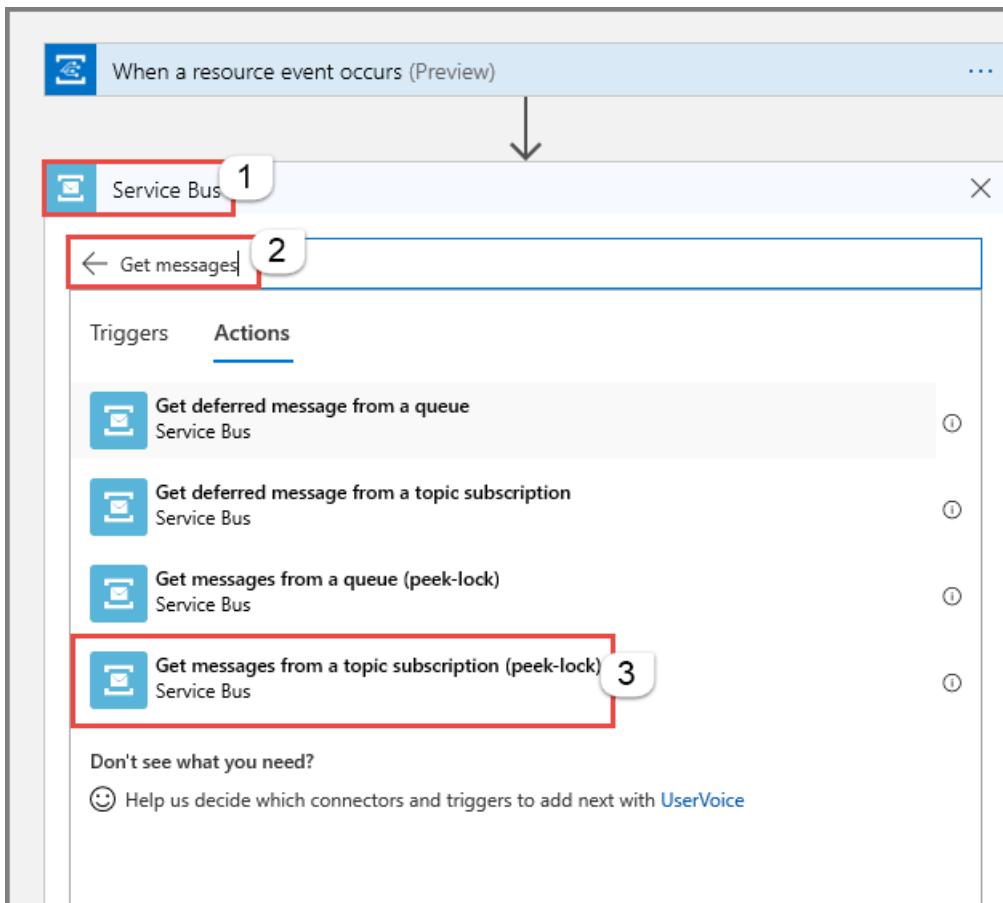


4. Select **Sign in**, enter your Azure credentials, and select **Allow Access**.
5. On the **When a resource event occurs** page, do the following steps:
  - a. Select your Azure subscription.
  - b. For **Resource Type**, select **Microsoft.ServiceBus.Namespaces**.
  - c. For **Resource Name**, select your Service Bus namespace.
  - d. Select **Add new parameter**, and select **Suffix Filter**.
  - e. For **Suffix Filter**, enter the name of your second Service Bus topic subscription.

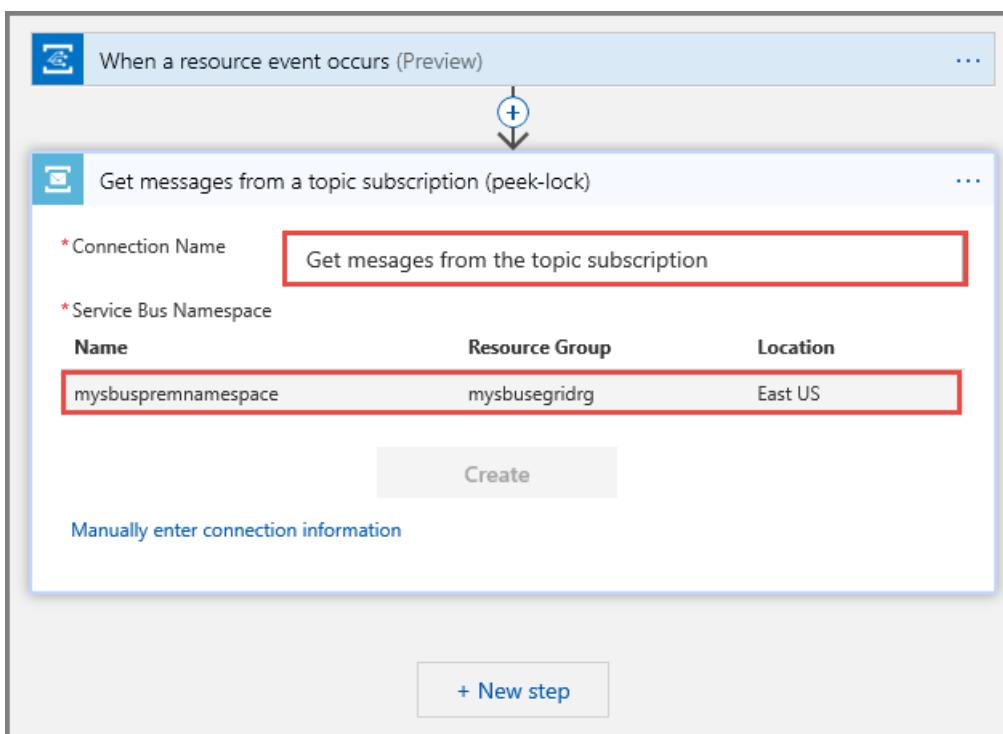


6. Select **+ New Step** in the designer, and do the following steps:

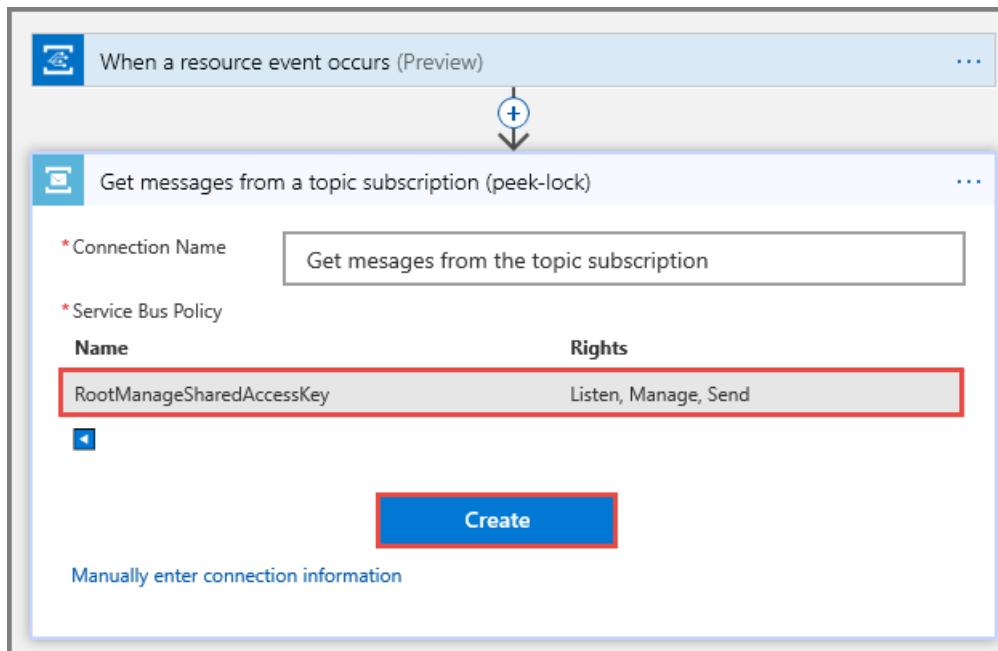
- a. Search for **Service Bus**.
- b. Select **Service Bus** in the list.
- c. Select for **Get messages** in the **Actions** list.
- d. Select **Get messages from a topic subscription (peek-lock)**.



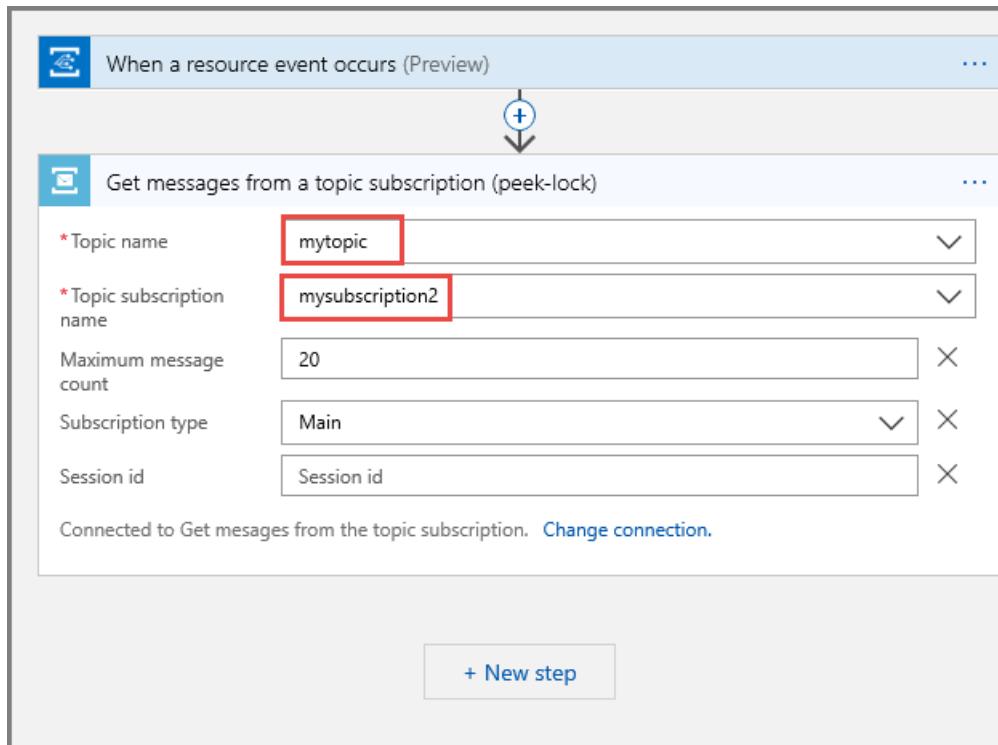
- e. Enter a **name for the connection**. For example: **Get messages from the topic subscription**, and select the Service Bus namespace.



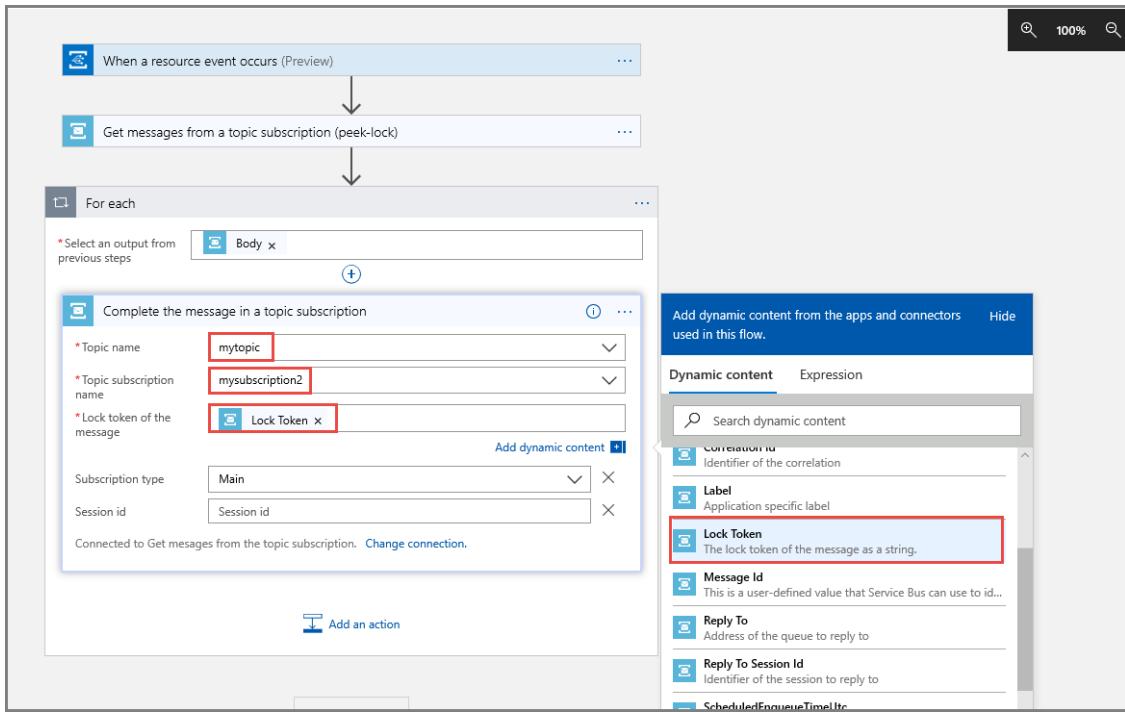
- f. Select **RootManageSharedAccessKey**.



- g. Select **Create**.
- h. Select your topic and subscription.



7. Select **+ New step**, and do the following steps:
  - a. Select **Service Bus**.
  - b. Select **Complete the message in a topic subscription** from the list of actions.
  - c. Select your Service Bus **topic**.
  - d. Select the second **subscription** to the topic.
  - e. For **Lock token of the message**, select **Lock Token** from the **Dynamic content**.



8. Select **Save** on the toolbar on the Logic Apps Designer to save the logic app.
9. Follow instruction in the [Send messages to the Service Bus topic](#) section to send messages to the topic.
10. Switch to the **Overview** page of your logic app. You see the logic app runs in the **Runs history** for the messages sent.

STATUS	START TIME	IDENTIFIER	DURATION	STATIC RESULTS
Succeeded	5/14/2019, 3:54 PM	08586437432031071895898265107CU39	908 Milliseconds	
Succeeded	5/14/2019, 3:51 PM	08586437433758372958676669338CU06	854 Milliseconds	
Succeeded	5/14/2019, 3:50 PM	08586437434431289810897384215CU02	1.36 Seconds	

## 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](#).

# Topic filters and actions

1/27/2020 • 3 minutes 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, 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 subscription. 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 queues, topics, and subscriptions](#)
- [SQL Filter syntax](#)
- [How to use Service Bus topics and subscriptions](#)

# Migrate existing Azure Service Bus standard namespaces to the premium tier

10/22/2019 • 8 minutes to read • [Edit Online](#)

Previously, Azure Service Bus offered namespaces only on the standard tier. Namespaces are multi-tenant setups that are optimized for low throughput and developer environments. The premium tier offers dedicated resources per namespace for predictable latency and increased throughput at a fixed price. The premium tier is optimized for high throughput and production environments that require additional enterprise features.

This article describes how to migrate existing standard tier namespaces to the premium tier.

## WARNING

Migration is intended for Service Bus standard namespaces to be upgraded to the premium tier. The migration tool does not support downgrading.

Some of the points to note:

- This migration is meant to happen in place, meaning that existing sender and receiver applications **don't require any changes to code or configuration**. The existing connection string will automatically point to the new premium namespace.
- The **premium** namespace should have **no entities** in it for the migration to succeed.
- All **entities** in the standard namespace are **copied** to the premium namespace during the migration process.
- Migration supports **1,000 entities per messaging unit** on the premium tier. To identify how many messaging units you need, start with the number of entities that you have on your current standard namespace.
- You can't directly migrate from **basic tier** to **premier tier**, but you can do so indirectly by migrating from basic to standard first and then from the standard to premium in the next step.

## Migration steps

Some conditions are associated with the migration process. Familiarize yourself with the following steps to reduce the possibility of errors. These steps outline the migration process, and the step-by-step details are listed in the sections that follow.

1. Create a new premium namespace.
2. Pair the standard and premium namespaces to each other.
3. Sync (copy-over) entities from the standard to the premium namespace.
4. Commit the migration.
5. Drain entities in the standard namespace by using the post-migration name of the namespace.
6. Delete the standard namespace.

## IMPORTANT

After the migration has been committed, access the old standard namespace and drain the queues and subscriptions. After the messages have been drained, they may be sent to the new premium namespace to be processed by the receiver applications. After the queues and subscriptions have been drained, we recommend that you delete the old standard namespace.

## Migrate by using the Azure CLI or PowerShell

To migrate your Service Bus standard namespace to premium by using the Azure CLI or PowerShell tool, follow these steps.

1. Create a new Service Bus premium namespace. You can reference the [Azure Resource Manager templates](#) or [use the Azure portal](#). Be sure to select **premium** for the **serviceBusSku** parameter.
2. Set the following environment variables to simplify the migration commands.

```
resourceGroup = <resource group for the standard namespace>
standardNamespace = <standard namespace to migrate>
premiumNamespaceArmId = <Azure Resource Manager ID of the premium namespace to migrate to>
postMigrationDnsName = <post migration DNS name entry to access the standard namespace>
```

### IMPORTANT

The Post-migration alias/name (`post_migration_dns_name`) will be used to access the old standard namespace post migration. Use this to drain the queues and the subscriptions, and then delete the namespace.

3. Pair the standard and premium namespaces and start the sync by using the following command:

```
az servicebus migration start --resource-group $resourceGroup --name $standardNamespace --target-namespace $premiumNamespaceArmId --post-migration-name $postMigrationDnsName
```

4. Check the status of the migration by using the following command:

```
az servicebus migration show --resource-group $resourceGroup --name $standardNamespace
```

The migration is considered complete when you see the following values:

- `MigrationState = "Active"`
- `pendingReplicationsOperationsCount = 0`
- `provisioningState = "Succeeded"`

This command also displays the migration configuration. Check to ensure the values are set correctly. Also check the premium namespace in the portal to ensure all the queues and topics have been created, and that they match what existed in the standard namespace.

5. Commit the migration by executing the following complete command:

```
az servicebus migration complete --resource-group $resourceGroup --name $standardNamespace
```

## Migrate by using the Azure portal

Migration by using the Azure portal has the same logical flow as migrating by using the commands. Follow these steps to migrate by using the Azure portal.

1. On the **Navigation** menu in the left pane, select **Migrate to premium**. Click the **Get Started** button to continue to the next page.

**Migrate to Premium**

Migrate to Premium to get all the benefits of Standard along with ...

- Exclusive Resource Allocation**: Predictable processing latency and throughput.
- Fixed Pricing**: Premium is priced per **Messaging Unit** instead of Pay-As-You-Go.
- Access to New Features**: Get access to **NET Availability Zones**, **Geo-DR** and more, first on Premium.

[Find out more about the migration process](#)

[Get Started!](#)

## 2. Complete Setup.

**Setup Namespaces**

**Premium Namespace**

Create a new Premium namespace or select an existing empty Premium namespace to migrate to.

[Create a new premium namespace](#) or [Select an existing empty premium namespace](#)

**Post-Migration Standard Namespace DNS**

This will be the new DNS name for your standard namespace, which you can continue using post-migration. [Learn more](#).

Post-migration name:

[Previous](#) [Next](#)

### a. Create and assign the premium namespace to migrate the existing standard namespace to.

**Create namespace**

**Name**: premium-namespace

**Messaging Units**: 1

Make this namespace zone redundant [?](#)

**Subscription**: ServiceBusTestSubscription-1

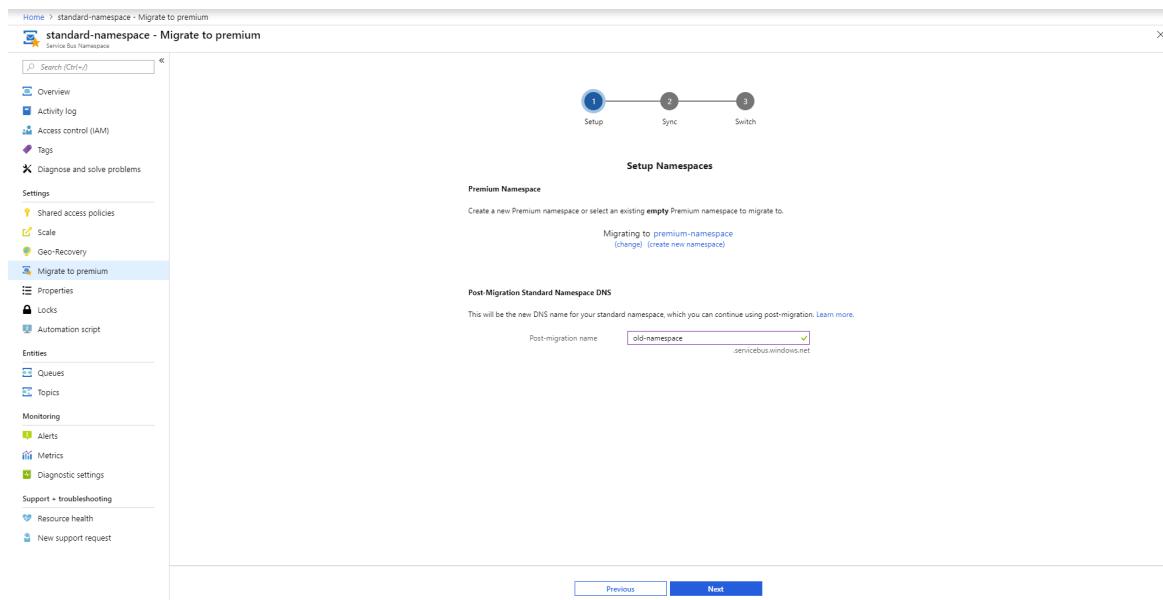
**Resource group**: rg

**Create new**

**Location**: West US

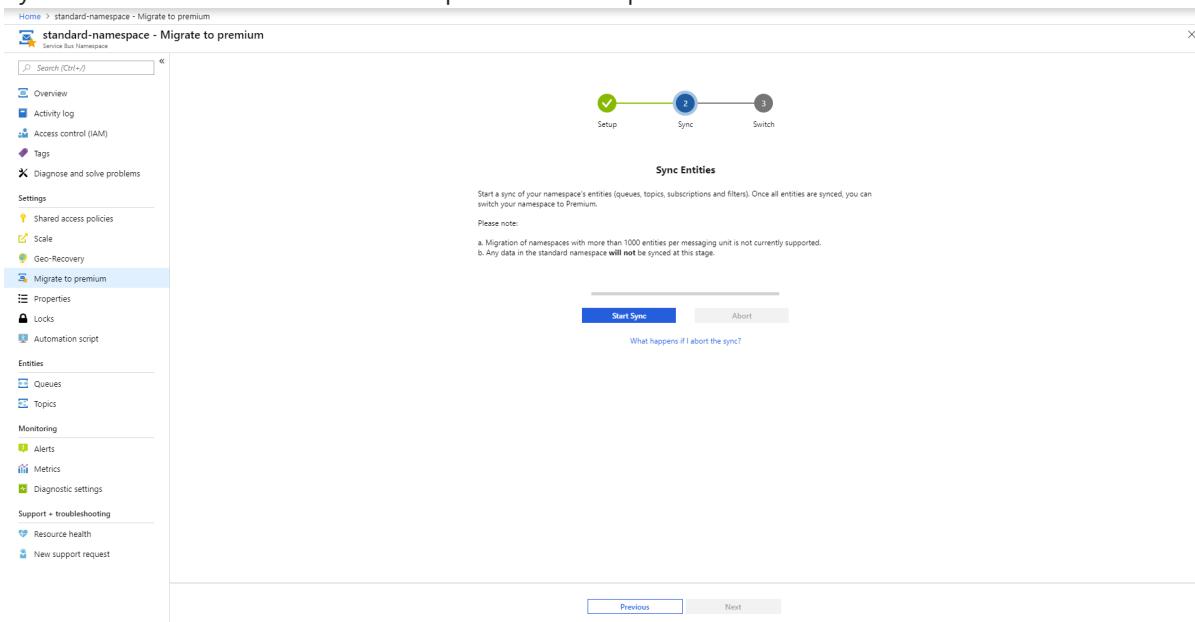
[Create](#)

### b. Choose a **Post Migration name**. You'll use this name to access the standard namespace after the migration is complete.



c. Select '**Next**' to continue.

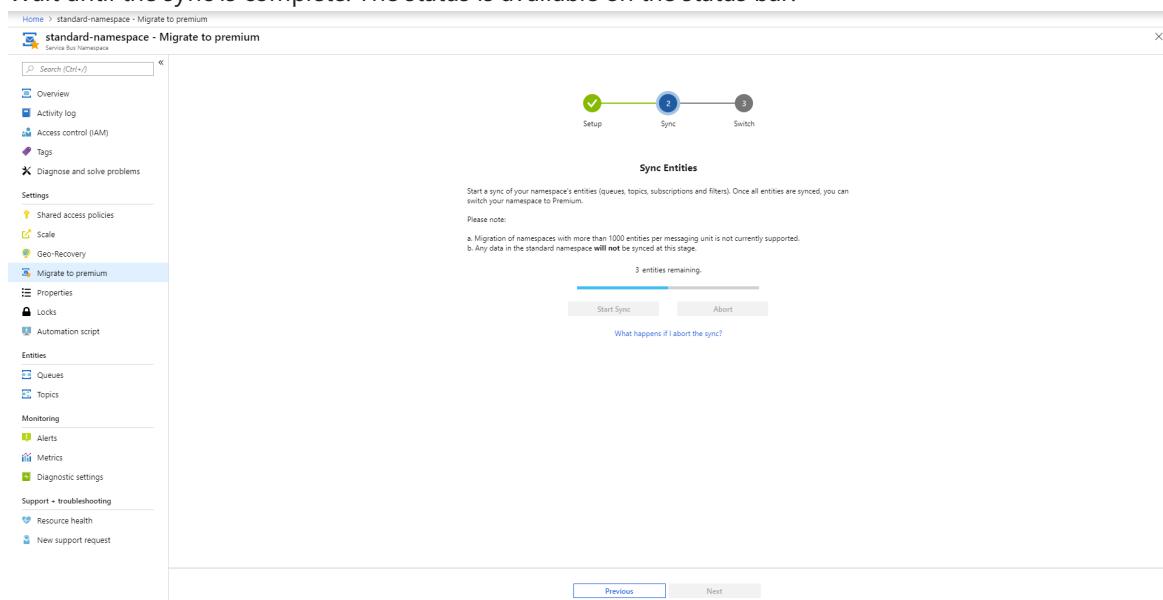
### 3. Sync entities between the standard and premium namespaces.



a. Select **Start Sync** to begin syncing the entities.

b. Select **Yes** in the dialog box to confirm and start the sync.

c. Wait until the sync is complete. The status is available on the status bar.



## IMPORTANT

If you need to abort the migration for any reason, please review the abort flow in the FAQ section of this document.

- d. After the sync is complete, select **Next** at the bottom of the page.
4. Review changes on the summary page. Select **Complete Migration** to switch namespaces and to complete the migration.

The screenshot shows the 'standard-namespace - Migrate to premium' blade in the Azure portal. On the left, there's a sidebar with various navigation links like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Shared access policies, Scale, Geo-Recovery), Entities (Queues, Topics), Monitoring (Alerts, Metrics, Diagnostic settings), Support + troubleshooting, Resource health, and New support request. The main area has a title 'standard-namespace - Migrate to premium' and a sub-section 'Service Bus Namespace'. A progress bar at the top indicates three steps: Setup (green checkmark), Sync (green checkmark), and Switch (blue circle). Below the progress bar, a message says 'Almost there!'. Under 'Review your changes', it lists 'Premium Namespace: premium-namespace.servicebus.windows.net' and 'Standard Namespace DNS (post-migration): old-namespace.servicebus.windows.net'. It also includes a note: 'This final step will complete the migration and switch traffic to your Premium namespace. Once you have switched, your standard namespace will still exist and can be accessed through your post-migration name. This may still have messages that you must drain.' At the bottom, there are 'Previous' and 'Complete Migration' buttons.

The confirmation page appears when the migration is complete.

The screenshot shows the same 'standard-namespace - Migrate to premium' blade after the migration is completed. The progress bar now shows all three steps (Setup, Sync, Switch) with green checkmarks. A large 'Success!' message is centered. Below it, a note says: 'Congrats! You have successfully migrated all entities to your Premium Namespace. Your standard namespace will still exist and can be accessed through your post-migration name. Please make sure that you drain any messages left in this namespace.' At the bottom, there is a 'Go to Premium Namespace' button.

## Caveats

Some of the features provided by Azure Service Bus Standard tier are not supported by Azure Service Bus Premium tier. These are by design since the premium tier offers dedicated resources for predictable throughput and latency.

Here is a list of features not supported by Premium and their mitigation -

### Express entities

Express entities that don't commit any message data to storage are not supported in Premium. Dedicated resources provided significant throughput improvement while ensuring that data is persisted, as is expected from any enterprise messaging system.

During migration, any of your express entities in your Standard namespace will be created on the Premium namespace as a non-express entity.

If you utilize Azure Resource Manager (ARM) templates, please ensure that you remove the 'enableExpress' flag from the deployment configuration so that your automated workflows execute without errors.

### **Partitioned entities**

Partitioned entities were supported in the Standard tier to provide better availability in a multi-tenant setup. With the provision of dedicated resources available per namespace in the Premium tier, this is no longer needed.

During migration, any partitioned entity in the Standard namespace is created on the Premium namespace as a non-partitioned entity.

If your ARM template sets 'enablePartitioning' to 'true' for a specific Queue or Topic, then it will be ignored by the broker.

## **FAQs**

### **What happens when the migration is committed?**

After the migration is committed, the connection string that pointed to the standard namespace will point to the premium namespace.

The sender and receiver applications will disconnect from the standard Namespace and reconnect to the premium namespace automatically.

### **What do I do after the standard to premium migration is complete?**

The standard to premium migration ensures that the entity metadata such as topics, subscriptions, and filters are copied from the standard namespace to the premium namespace. The message data that was committed to the standard namespace isn't copied from the standard namespace to the premium namespace.

The standard namespace may have some messages that were sent and committed while the migration was underway. Manually drain these messages from the standard Namespace and manually send them to the premium Namespace. To manually drain the messages, use a console app or a script that drains the standard namespace entities by using the Post Migration DNS name that you specified in the migration commands. Send these messages to the premium namespace so that they can be processed by the receivers.

After the messages have been drained, delete the standard namespace.

#### **IMPORTANT**

After the messages from the standard namespace have been drained, delete the standard namespace. This is important because the connection string that initially referred to the standard namespace now refers to the premium namespace. You won't need the standard Namespace anymore. Deleting the standard namespace that you migrated helps reduce later confusion.

### **How much downtime do I expect?**

The migration process is meant to reduce the expected downtime for the applications. Downtime is reduced by using the connection string that the sender and receiver applications use to point to the new premium namespace.

The downtime that is experienced by the application is limited to the time it takes to update the DNS entry to point to the premium namespace. Downtime is approximately 5 minutes.

### **Do I have to make any configuration changes while doing the migration?**

No, there are no code or configuration changes needed to do the migration. The connection string that sender and receiver applications use to access the standard Namespace is automatically mapped to act as an alias for the premium namespace.

## What happens when I abort the migration?

The migration can be aborted either by using the `Abort` command or by using the Azure portal.

### Azure CLI

```
az servicebus migration abort --resource-group $resourceGroup --name $standardNamespace
```

### Azure portal

The screenshot shows the Azure portal interface for a Service Bus Namespace named 'standard-namespace'. In the left sidebar, under 'Migrate to premium', the 'Migrate to premium' option is selected. On the main page, there is a prominent 'Abort' button with a warning message: 'Are you sure you would like to abort the migration? This operation will not delete any synced entities from your premium namespace.' Below this, there is a 'Sync Entities' section with a progress bar showing '0 entities remaining'. At the bottom of the sync section, there are 'Start Sync' and 'Abort' buttons. A tooltip 'What happens if I abort the sync?' is visible over the 'Abort' button. Navigation buttons 'Previous' and 'Next' are at the bottom of the page.

This screenshot shows the same Azure portal interface as the previous one, but the migration process has been completed. The 'Sync Entities' section now displays a green checkmark icon and the text 'Successfully aborted migration' with a timestamp '10:56 AM'. The progress bar shows '0 entities remaining'. The 'Start Sync' and 'Abort' buttons are still present. The navigation buttons 'Previous' and 'Next' are at the bottom.

When the migration process is aborted, it aborts the process of copying the entities (topics, subscriptions, and filters) from the standard to the premium namespace and breaks the pairing.

The connection string isn't updated to point to the premium namespace. Your existing applications continue to work as they did before you started the migration.

However, it doesn't delete the entities on the premium namespace or delete the premium namespace. Delete the entities manually if you decided not to move forward with the migration.

## **IMPORTANT**

If you decide to abort the migration, delete the premium Namespace that you had provisioned for the migration so that you are not charged for the resources.

### **I don't want to have to drain the messages. What do I do?**

There may be messages that are sent by the sender applications and committed to the storage on the standard Namespace while the migration is taking place and just before the migration is committed.

During migration, the actual message data/payload isn't copied from the standard to the premium namespace. The messages have to be manually drained and then sent to the premium namespace.

However, if you can migrate during a planned maintenance/housekeeping window, and you don't want to manually drain and send the messages, follow these steps:

1. Stop the sender applications. The receiver applications will process the messages that are currently in the standard namespace and will drain the queue.
2. After the queues and subscriptions in the standard Namespace are empty, follow the procedure that is described earlier to execute the migration from the standard to the premium namespace.
3. After the migration is complete, you can restart the sender applications.
4. The senders and receivers will now automatically connect with the premium namespace.

## **NOTE**

You do not have to stop the receiver applications for the migration.

After the migration is complete, the receiver applications will disconnect from the standard namespace and automatically connect to the premium namespace.

## **Next steps**

- Learn more about the [differences between standard and premium Messaging](#).
- Learn about the [High-Availability and Geo-Disaster recovery aspects for Service Bus premium](#).

# Partitioned queues and topics

3/12/2019 • 11 minutes 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. Partitioning 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.

## NOTE

Partitioning is available at entity creation for all queues and topics in Basic or Standard SKUs. It is not available for the Premium messaging SKU, but any previously existing partitioned entities in Premium namespaces continue to work as expected.

It is not possible to change the partitioning option on any existing queue or topic; you can only set the option when you create the entity.

## How it works

Each partitioned queue or topic consists of multiple partitions. Each partition 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 partitions. 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 partitions 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, each of the same size specified. 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, partitioning entities are not supported. However, you can still create Service Bus

queues and topics in 1, 2, 3, 4, 5, 10, 20, 40, or 80-GB sizes (the default is 1 GB). You can see the size of your queue or topic by looking at its entry on the [Azure portal](#), in the **Overview** blade for that entity.

## 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, and are available only in the older `WindowsAzure.ServiceBus` library. 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](#). 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 not supported, and the checkbox has no effect.

## 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 partition based on that key. If it does not find a partition key, it selects the partition based on an internal algorithm.

### Using a partition key

Some scenarios, such as sessions or transactions, require messages to be stored in a specific partition. All these scenarios require the use of a partition key. All messages that use the same partition key are assigned to the same partition. If the partition 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 `SessionId` property set, then Service Bus uses **SessionID** as the partition key. This way, all messages that belong to the same session are handled by the same message broker. Sessions enable Service Bus to guarantee message ordering as well as the consistency of session states.

**PartitionKey:** If a message has the `PartitionKey` property but not the `SessionId` property set, then Service Bus uses the `PartitionKey` property value 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 `RequiresDuplicateDetection` property set to `true` and the `SessionId` or `PartitionKey` properties are not set, then the `MessageId` property value serves as the partition key. (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 ID enables Service Bus to detect and eliminate duplicate messages. If the `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 partitions of

the partitioned queue or topic. If the chosen partition is not available, Service Bus assigns the message to a different partition. 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 partition, the [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.

A partition key "pins" a message to a specific partition. If the messaging store that holds this partition is unavailable, Service Bus returns an error. In the absence of a partition key, Service Bus can choose a different partition 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. The key can be one of the following properties: [SessionId](#), [PartitionKey](#), or [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))
{
    Message msg = new Message("This is a message");
    msg.PartitionKey = "myPartitionKey";
    messageSender.SendAsync(msg);
    ts.CompleteAsync();
}
committableTransaction.Commit();
```

If any of the properties that serve as a partition key are set, Service Bus pins the message to a specific partition. 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 [SessionId](#) property set. If the [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))
{
    Message msg = new Message("This is a message");
    msg.SessionId = "mySession";
    messageSender.SendAsync(msg);
    ts.CompleteAsync();
}
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 partition. If any of the partitions 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 partitions of the entity. If any partition is unhealthy, it could result in failures for these operations. For the Get operation, information such as message counts must be aggregated from all partitions. If any partition 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 partitions 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:** Available only in the older [WindowsAzure.ServiceBus](#) library. [PeekBatch](#) does not always return the number of messages specified in the [MessageCount](#) property. There are two common reasons for this behavior. 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 are not supported in the Premium messaging tier. Sessions are supported in the premier tier by using [SessionId](#).
- 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)

1/24/2020 • 6 minutes 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.

## NOTE

The basic tier of Service Bus does not support sessions. The standard and premium tiers support sessions. For more information, see [Service Bus pricing](#).

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**

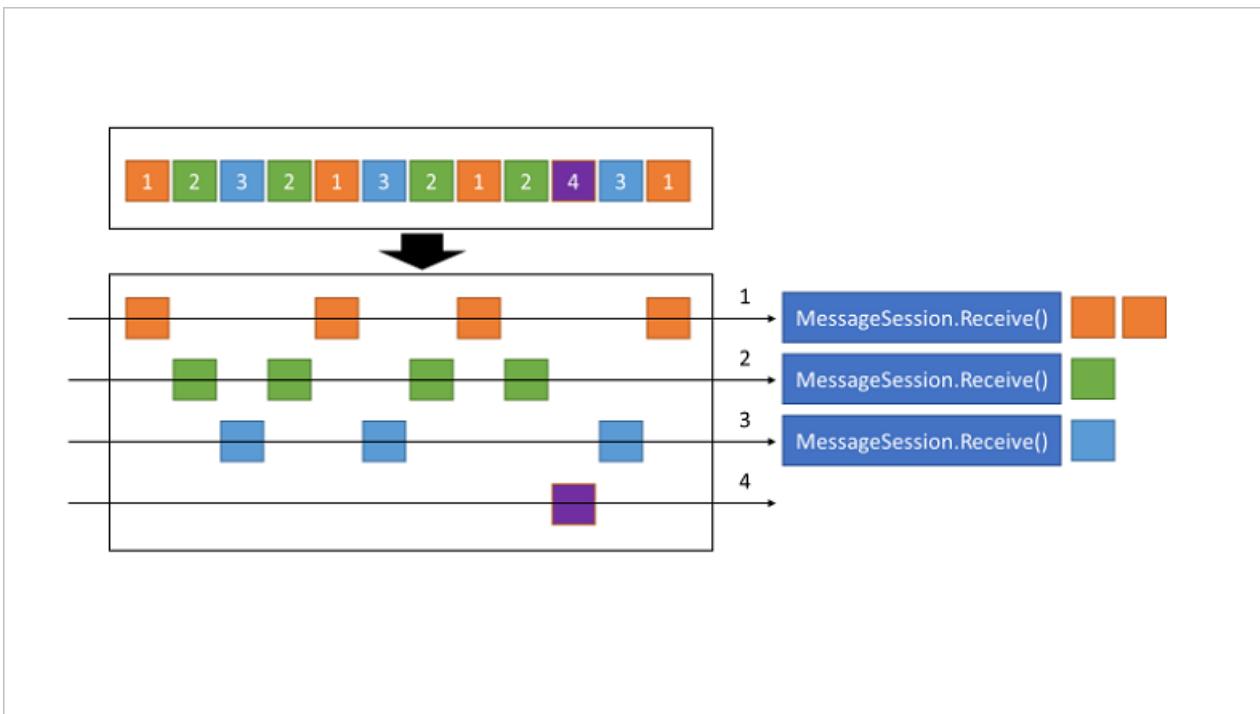
**NOTE**

When Sessions are enabled on a queue or a subscription, the client applications can **no longer** send/receive regular messages. All messages must be sent as part of a session (by setting the session id) and received by receiving the session.

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 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.

## Impact of delivery count

The definition of delivery count per message in the context of sessions varies slightly from the definition in the absence of sessions. Here is a table summarizing when the delivery count is incremented.

SCENARIO	IS THE MESSAGE'S DELIVERY COUNT INCREMENTED
Session is accepted, but the session lock expires (due to timeout)	Yes
Session is accepted, the messages within the session are not completed (even if they are locked), and the session is closed	No
Session is accepted, messages are completed, and then the session is explicitly closed	N/A (this is the standard flow. Here messages are removed from the session)

## Next steps

- See either the [Microsoft.Azure.ServiceBus samples](#) or [Microsoft.ServiceBus.Messaging samples](#) for an example that uses the .NET Framework client to handle session-aware messages.

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

- [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

1/24/2020 • 5 minutes 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.

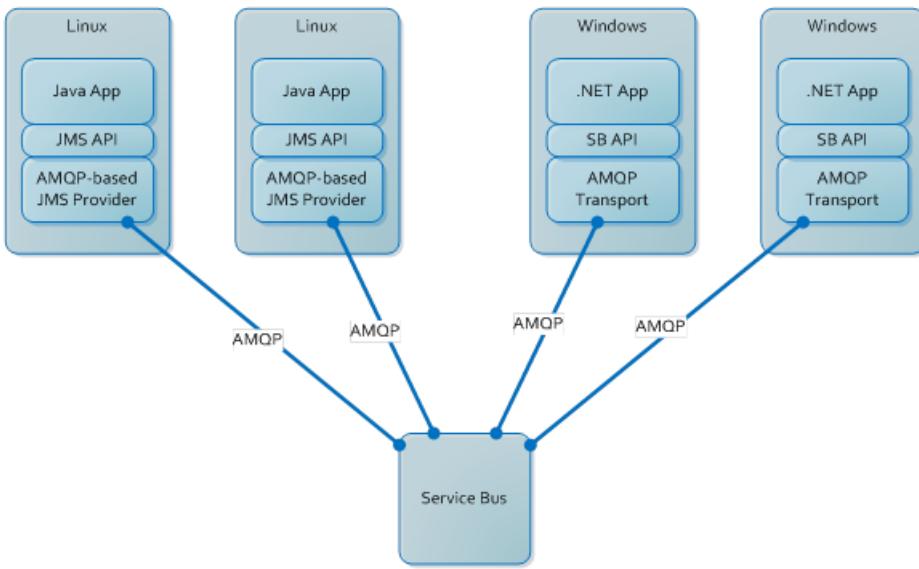
Some of the commonly cited benefits of open standards include:

- 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 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

1/24/2020 • 4 minutes 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 in the section [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](#)

# Use the Java Message Service (JMS) with Azure Service Bus and AMQP 1.0

10/28/2019 • 10 minutes to read • [Edit Online](#)

This article explains how to use Azure 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 Azure Service Bus .NET API. You can use these two guides together to learn about cross-platform messaging using AMQP 1.0.

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 Azure 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.

## Get started with Service Bus

This guide assumes that you already have a Service Bus namespace containing a queue named **basicqueue**. 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-jms-client-[version].jar

### NOTE

JMS JAR names and versions may have changed. For details, see [Qpid JMS - AMQP 1.0](#).

## 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
```

#### Setup JNDI context and Configure the ConnectionFactory

The **ConnectionString** referenced in the one available in the 'Shared Access Policies' in the [Azure Portal](#) under **Primary Connection String**

```
// The connection string builder is the only part of the azure-servicebus SDK library
// we use in this JMS sample and for the purpose of robustly parsing the Service Bus
// connection string.
ConnectionStringBuilder csb = new ConnectionStringBuilder(connectionString);

// set up JNDI context
Hashtable<String, String> hashtable = new Hashtable<>();
hashtable.put("connectionfactory.SBCF", "amqps://" + csb.getEndpoint().getHost() + "?"
amqp.idleTimeout=120000&amqp.traceFrames=true");
hashtable.put("queue.QUEUE", "BasicQueue");
hashtable.put(Context.INITIAL_CONTEXT_FACTORY, "org.apache.qpid.jms.jndi.JmsInitialContextFactory");
Context context = new InitialContext(hashtable);

ConnectionFactory cf = (ConnectionFactory) context.lookup("SBCF");

// Look up queue
Destination queue = (Destination) context.lookup("QUEUE");
```

#### Configure Producer and Consumer Destination Queues

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

To create the destination queue for the Producer -

```
String queueName = "queueName";
Destination queue = (Destination) queueName;

ConnectionFactory cf = (ConnectionFactory) context.lookup("SBCF");
Connection connection = cf.createConnection(csb.getSasKeyName(), csb.getSasKey());

Session session = connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);

// Create Producer
MessageProducer producer = session.createProducer(queue);
```

To create a destination queue for the Consumer -

```

String queueName = "queueName";
Destination queue = (Destination) queueName;

ConnectionFactory cf = (ConnectionFactory) context.lookup("SBCF");
Connection connection = cf.createConnection(csb.getSasKeyName(), csb.getSasKey());

Session session = connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);

// Create Consumer
MessageConsumer consumer = session.createConsumer(queue);

```

## 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**.

```

// set up JNDI context
Hashtable<String, String> hashtable = new Hashtable<>();
hashtable.put("connectionfactory.SBCF", "amqps://" + csb.getEndpoint().getHost() + \
"?amqp.idleTimeout=120000&amqp.traceFrames=true");
hashtable.put("queue.QUEUE", "BasicQueue");
hashtable.put(Context.INITIAL_CONTEXT_FACTORY, "org.apache.qpid.jms.jndi.JmsInitialContextFactory");
Context context = new InitialContext(hashtable);

```

## 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.

You can all access all the source code and configuration information from the [Azure Service Bus Samples JMS Queue Quick Start](#)

```

// Copyright (c) Microsoft. All rights reserved.
// Licensed under the MIT license. See LICENSE file in the project root for full license information.

package com.microsoft.azure.servicebus.samples.jmsqueuequickstart;

import com.microsoft.azure.servicebus.primitives.ConnectionStringBuilder;
import org.apache.commons.cli.*;
import org.apache.log4j.*;

import javax.jms.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Hashtable;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.function.Function;

/**
 * This sample demonstrates how to send messages from a JMS Queue producer into
 * an Azure Service Bus Queue, and receive them with a JMS message consumer.
 * JMS Queue.
 */
public class JmsQueueQuickstart {

```

```

// Number of messages to send
private static int totalSend = 10;
//Tracking counter for how many messages have been received; used as termination condition
private static AtomicInteger totalReceived = new AtomicInteger(0);
// log4j logger
private static Logger logger = Logger.getRootLogger();

public void run(String connectionString) throws Exception {

    // The connection string builder is the only part of the azure-servicebus SDK library
    // we use in this JMS sample and for the purpose of robustly parsing the Service Bus
    // connection string.
    ConnectionStringBuilder csb = new ConnectionStringBuilder(connectionString);

    // set up JNDI context
    Hashtable<String, String> hashtable = new Hashtable<>();
    hashtable.put("connectionfactory.SBCF", "amqps://" + csb.getEndpoint().getHost() + "?"
amqp.idleTimeout=120000&amqp.traceFrames=true");
    hashtable.put("queue.QUEUE", "BasicQueue");
    hashtable.put(Context.INITIAL_CONTEXT_FACTORY, "org.apache.qpid.jms.jndi.JmsInitialContextFactory");
    Context context = new InitialContext(hashtable);
    ConnectionFactory cf = (ConnectionFactory) context.lookup("SBCF");

    // Look up queue
    Destination queue = (Destination) context.lookup("QUEUE");

    // we create a scope here so we can use the same set of local variables cleanly
    // again to show the receive side separately with minimal clutter
    {
        // Create Connection
        Connection connection = cf.createConnection(csb.getSasKeyName(), csb.getSasKey());
        // Create Session, no transaction, client ack
        Session session = connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);

        // Create producer
        MessageProducer producer = session.createProducer(queue);

        // Send messages
        for (int i = 0; i < totalSend; i++) {
            BytesMessage message = session.createBytesMessage();
            message.writeBytes(String.valueOf(i).getBytes());
            producer.send(message);
            System.out.printf("Sent message %d.\n", i + 1);
        }

        producer.close();
        session.close();
        connection.stop();
        connection.close();
    }

    {
        // Create Connection
        Connection connection = cf.createConnection(csb.getSasKeyName(), csb.getSasKey());
        connection.start();
        // Create Session, no transaction, client ack
        Session session = connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);
        // Create consumer
        MessageConsumer consumer = session.createConsumer(queue);
        // create a listener callback to receive the messages
        consumer.setMessageListener(message -> {
            try {
                // receives message is passed to callback
                System.out.printf("Received message %d with sq#: %s\n",
                    totalReceived.incrementAndGet(), // increments the tracking counter
                    message.getJMSMessageID());
                message.acknowledge();
            } catch (Exception e) {
                logger.error(e);
            }
        });
    }
}

```

```

        }

    });

    // wait on the main thread until all sent messages have been received
    while (totalReceived.get() < totalSend) {
        Thread.sleep(1000);
    }
    consumer.close();
    session.close();
    connection.stop();
    connection.close();
}

System.out.printf("Received all messages, exiting the sample.\n");
System.out.printf("Closing queue client.\n");
}

public static void main(String[] args) {

    System.exit(runApp(args, (connectionString) -> {
        JmsQueueQuickstart app = new JmsQueueQuickstart();
        try {
            app.run(connectionString);
            return 0;
        } catch (Exception e) {
            System.out.printf("%s", e.toString());
            return 1;
        }
    }));
}
}

static final String SB_SAMPLES_CONNECTIONSTRING = "SB_SAMPLES_CONNECTIONSTRING";

public static int runApp(String[] args, Function<String, Integer> run) {
    try {

        String connectionString = null;

        // parse connection string from command line
        Options options = new Options();
        options.addOption(new Option("c", true, "Connection string"));
        CommandLineParser clp = new DefaultParser();
        CommandLine cl = clp.parse(options, args);
        if (cl.getOptionValue("c") != null) {
            connectionString = cl.getOptionValue("c");
        }

        // get overrides from the environment
        String env = System.getenv(SB_SAMPLES_CONNECTIONSTRING);
        if (env != null) {
            connectionString = env;
        }

        if (connectionString == null) {
            HelpFormatter formatter = new HelpFormatter();
            formatter.printHelp("run jar with", "", options, "", true);
            return 2;
        }
        return run.apply(connectionString);
    } catch (Exception e) {
        System.out.printf("%s", e.toString());
        return 3;
    }
}
}
}

```

## Run the application

Pass the **Connection String** from the Shared Access Policies to run the application. Below is the output of the form by running the Application:

```
> mvn clean package  
>java -jar ./target/jmsqueuequickstart-1.0.0-jar-with-dependencies.jar -c "<CONNECTION_STRING>"  
  
Sent message 1.  
Sent message 2.  
Sent message 3.  
Sent message 4.  
Sent message 5.  
Sent message 6.  
Sent message 7.  
Sent message 8.  
Sent message 9.  
Sent message 10.  
Received message 1 with sq#: ID:7f6a7659-bcdf-4af6-afc1-4011e2ddcb3c:1:1:1-1  
Received message 2 with sq#: ID:7f6a7659-bcdf-4af6-afc1-4011e2ddcb3c:1:1:1-2  
Received message 3 with sq#: ID:7f6a7659-bcdf-4af6-afc1-4011e2ddcb3c:1:1:1-3  
Received message 4 with sq#: ID:7f6a7659-bcdf-4af6-afc1-4011e2ddcb3c:1:1:1-4  
Received message 5 with sq#: ID:7f6a7659-bcdf-4af6-afc1-4011e2ddcb3c:1:1:1-5  
Received message 6 with sq#: ID:7f6a7659-bcdf-4af6-afc1-4011e2ddcb3c:1:1:1-6  
Received message 7 with sq#: ID:7f6a7659-bcdf-4af6-afc1-4011e2ddcb3c:1:1:1-7  
Received message 8 with sq#: ID:7f6a7659-bcdf-4af6-afc1-4011e2ddcb3c:1:1:1-8  
Received message 9 with sq#: ID:7f6a7659-bcdf-4af6-afc1-4011e2ddcb3c:1:1:1-9  
Received message 10 with sq#: ID:7f6a7659-bcdf-4af6-afc1-4011e2ddcb3c:1:1:1-10  
Received all messages, exiting the sample.  
Closing queue client.
```

## AMQP disposition and Service Bus operation mapping

Here is how an AMQP disposition translates to a Service Bus operation:

```
ACCEPTED = 1; -> Complete()  
REJECTED = 2; -> DeadLetter()  
RELEASED = 3; (just unlock the message in service bus, will then get redelivered)  
MODIFIED_FAILED = 4; -> Abandon() which increases delivery count  
MODIFIED_FAILED_UNDELIVERABLE = 5; -> Defer()
```

## JMS Topics vs. Service Bus Topics

Using Azure Service Bus topics and subscriptions through the Java Message Service (JMS) API provides basic send and receive capabilities. It's a convenient choice when porting applications from other message brokers with JMS compliant APIs, even though Service Bus topics differ from JMS Topics and require a few adjustments.

Azure Service Bus topics route messages into named, shared, durable subscriptions that are managed through the Azure Resource Management interface, the Azure command line tools, or through the Azure portal. Each subscription allows for up to 2000 selection rules, each of which may have a filter condition and, for SQL filters, also a metadata transformation action. Each filter condition match selects the input message to be copied into the subscription.

Receiving messages from subscriptions is identical receiving messages from queues. Each subscription has an associated dead-letter queue as well as the ability to automatically forward messages to another queue or topics.

JMS Topics allow clients to dynamically create nondurable and durable subscribers that optionally allow filtering messages with message selectors. These unshared entities are not supported by Service Bus. The SQL filter rule syntax for Service Bus is, however, very similar to the message selector syntax supported by JMS.

The JMS Topic publisher side is compatible with Service Bus, as shown in this sample, but dynamic subscribers are not. The following topology-related JMS APIs are not supported with Service Bus.

## 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.
- Transacted sessions and distributed transactions are not supported.

Additionally, Azure Service Bus splits the control plane from the data plane and therefore does not support several of JMS's dynamic topology functions:

UNSUPPORTED METHOD	REPLACE WITH
createDurableSubscriber	create a Topic subscription porting the message selector
createDurableConsumer	create a Topic subscription porting the message selector
createSharedConsumer	Service Bus topics are always shareable, see above
createSharedDurableConsumer	Service Bus topics are always shareable, see above
createTemporaryTopic	create a topic via management API/tools/portal with <i>AutoDeleteOnIdle</i> set to an expiration period
createTopic	create a topic via management API/tools/portal
unsubscribe	delete the topic management API/tools/portal
createBrowser	unsupported. Use the Peek() functionality of the Service Bus API
createQueue	create a queue via management API/tools/portal
createTemporaryQueue	create a queue via management API/tools/portal with <i>AutoDeleteOnIdle</i> set to an expiration period
receiveNoWait	utilize the receive() method provided by the Service Bus SDK and specify a very low or zero timeout

## 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

2/1/2019 • 27 minutes 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, 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).

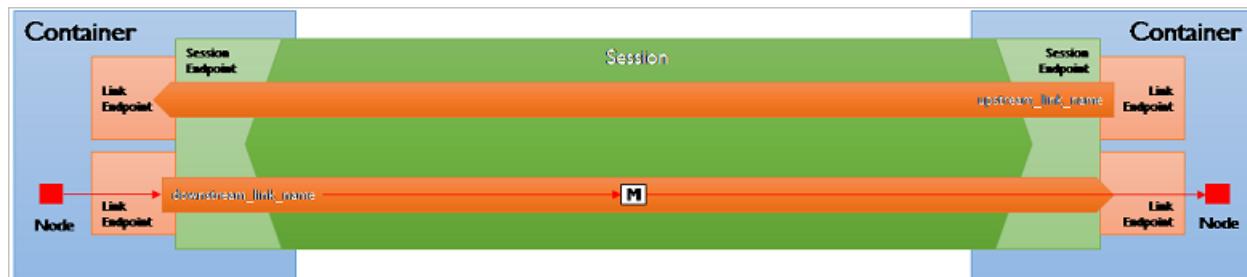
This window-based model is roughly analogous to the TCP concept of window-based flow control, but at the session level inside the socket. The protocol's concept of allowing for multiple concurrent sessions exists so that high priority traffic could be rushed past throttled normal traffic, like on a highway 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.

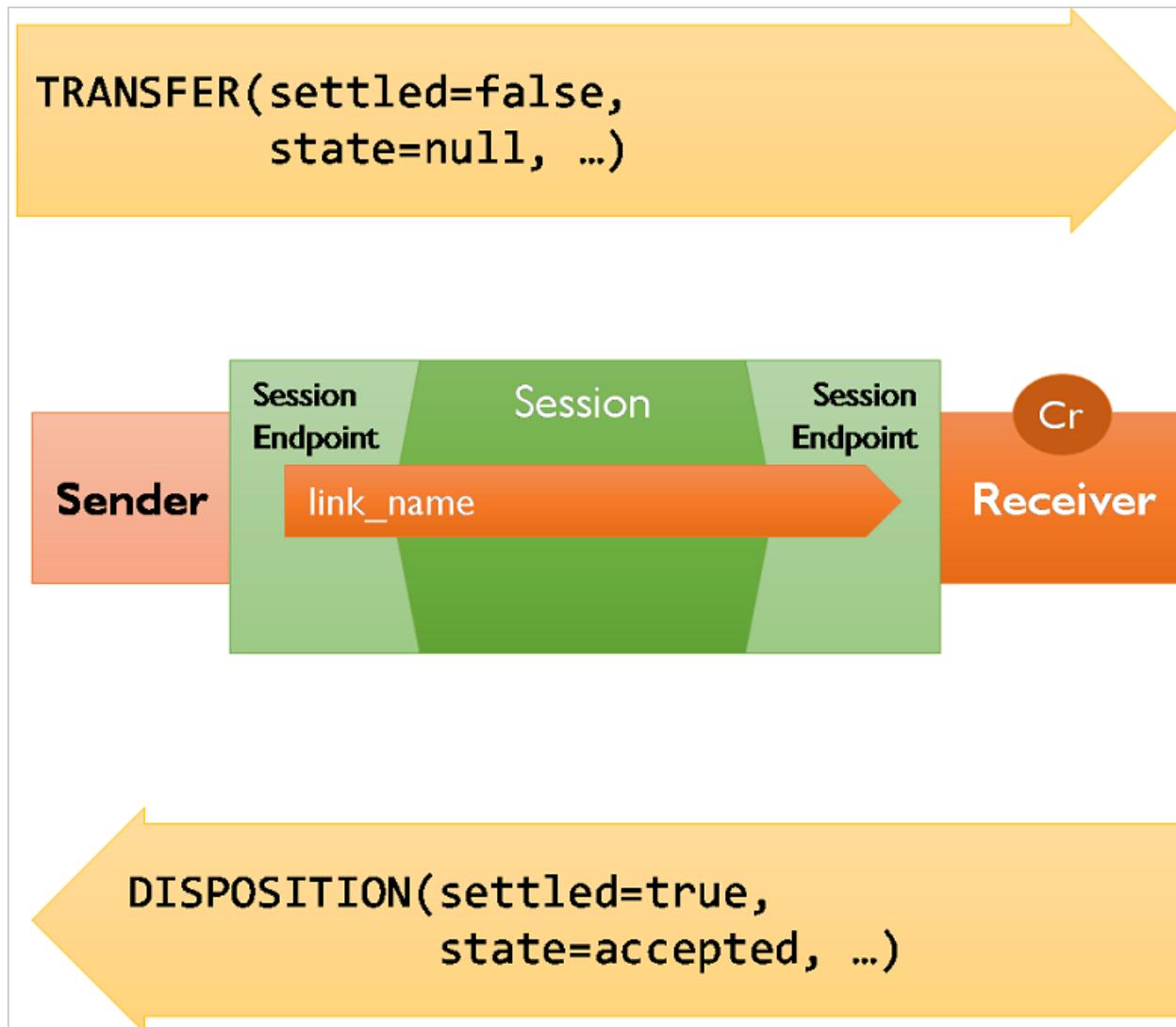
In Service Bus, a node is directly equivalent to a queue, a topic, a subscription, or a deadletter subqueue of a

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` on 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 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.

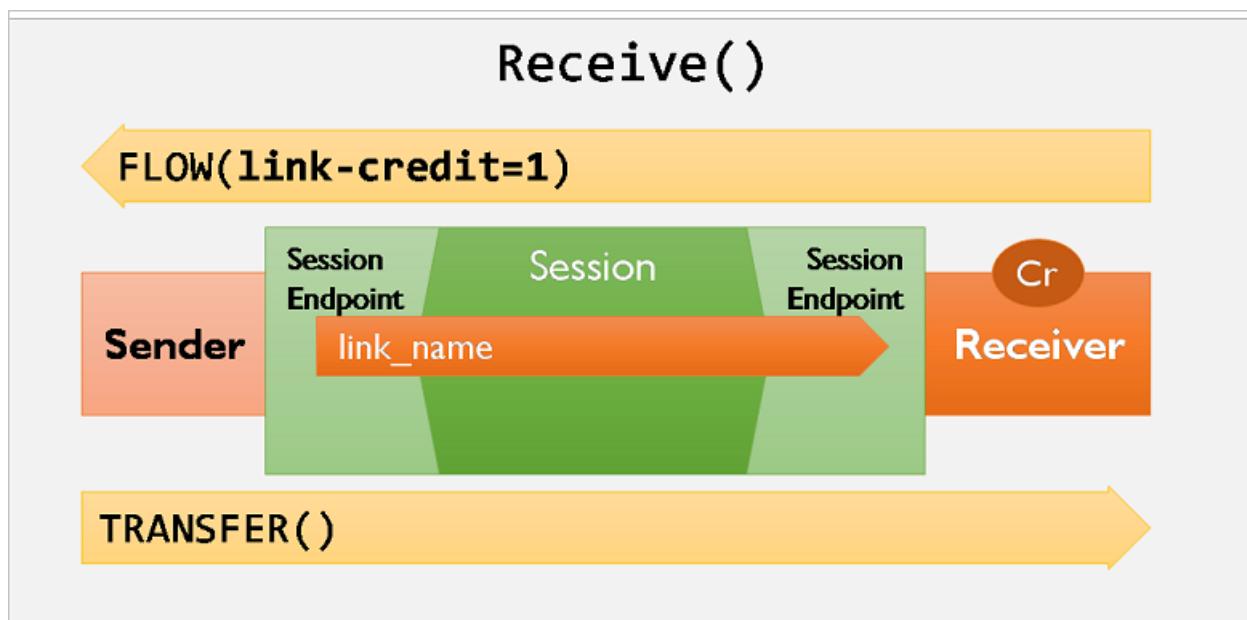
Service Bus does not support link recovery; if the client loses the connection to Service Bus with an unsettled message transfer pending, that message transfer is lost, and the client must reconnect, reestablish the link, and retry the transfer.

As such, Service Bus and Event Hubs support "at least once" transfer 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}, role=sender, source={client link ID}, target={entity name} )	No action
No action	<-- attach( name={link name}, handle={numeric handle}, role=receiver, source={client link ID}, target={entity name} )

#### Create message sender (error)

CLIENT	SERVICE BUS
--> attach( name={link name}, handle={numeric handle}, role=sender, source={client link ID}, target={entity name} )	No action
No action	<-- attach( name={link name}, handle={numeric handle}, role=receiver, source=null, target=null )  <-- detach( handle={numeric handle}, closed=true, error={error info} )

#### Close message receiver/sender

CLIENT	SERVICE BUS
--> detach( handle={numeric handle}, closed=true )	No action
No action	<-- detach( handle={numeric handle}, closed=true )

#### Send (success)

CLIENT	SERVICE BUS
--> transfer( delivery-id={numeric handle}, delivery-tag={binary handle}, settled=false,,more=false, state=null, resume=false )	No action
No action	<-- disposition( role=receiver, first={delivery ID}, last={delivery ID}, settled=true, state=accepted )

#### Send (error)

CLIENT	SERVICE BUS
--> transfer( delivery-id={numeric handle}, delivery-tag={binary handle}, settled=false,,more=false, state=null, resume=false )	No action
No action	<-- disposition( role=receiver, first={delivery ID}, last={delivery ID}, settled=true, state=rejected( error={error info} ) )

#### Receive

CLIENT	SERVICE BUS
--> flow( link-credit=1 )	No action
No action	< transfer( delivery-id={numeric handle}, delivery-tag={binary handle}, settled=false, more=false, state=null, resume=false )

CLIENT	SERVICE BUS
--> disposition( role= <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 `txn-id`.

For transactional interaction, the client acts as 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, that is, 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, 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 (Declare())}</pre>	----->	
	<-----	<pre>disposition(   first=0, last=0,   state=Declared(     txn-id={transaction ID}   )) )</pre>

#### Discharging a transaction

The controller concludes the transactional work by sending a `discharge` message to the coordinator. The controller indicates that it wishes to commit or roll back 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 (Declare())}</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>txn-id=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
amqp:jwt	JSON Web Token (JWT)	AMQP Value (string)	Not yet available.
amqp:swt	Simple Web Token (SWT)	AMQP Value (string)	Only supported for SWT tokens issued by AAD/ACS
servicebus.windows.net:sastoken	Service Bus SAS Token	AMQP Value (string)	-

Tokens confer rights. Service Bus knows 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 problem occurring, 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 feature is 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 are 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>	----->	

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

1/24/2020 • 14 minutes 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_name>	“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.

#### NOTE

Lock tokens are the `DeliveryTag` property on received messages. See the following example in the [.NET SDK](#) which retrieves these. The token may also appear in the 'DeliveryAnnotations' as 'x-opt-lock-token' however, this is not guaranteed and the `DeliveryTag` should be preferred.

#### Response

The response message must include the following application properties:

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
statusCode	int	Yes	HTTP response code [RFC2616]  200: OK – success, otherwise failed.
statusDescription	string	No	Description of the status.

The response message body must consist of an **amqp-value** section containing a map with the following entries:

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
expirations	array of timestamp	Yes	Message lock token new expiration corresponding to the request lock tokens.

#### Peek Message

Peeks messages without locking.

#### Request

The request message must include the following application properties:

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
operation	string	Yes	com.microsoft:peek-message
com.microsoft:server-timeout	uint	No	Operation server timeout in milliseconds.

The request message body must consist of an **amqp-value** section containing a **map** with the following entries:

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
from-sequence-number	long	Yes	Sequence number from which to start peek.
message-count	int	Yes	Maximum number of messages to peek.

#### Response

The response message must include the following application properties:

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
statusCode	int	Yes	HTTP response code [RFC2616]  200: OK – has more messages  204: 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	<code>amqpMessage.Properties.MessageId</code> as string
session-id	string	No	<code>amqpMessage.Properties.GroupId</code> as string
partition-key	string	No	<code>amqpMessage.MessageAnnotations."x-opt-partition-key"</code>
via-partition-key	string	No	<code>amqpMessage.MessageAnnotations."x-opt-via-partition-key"</code>
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	<code>com.microsoft:cancel-scheduled-message</code>

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
com.microsoft:server-timeout	uint	No	Operation server timeout in milliseconds.

The request message body must consist of an **amqp-value** section containing a **map** with the following entries:

KEY	VALUE TYPE	REQUIRED	VALUE CONTENTS
sequence-numbers	array of long	Yes	Sequence numbers 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]
statusDescription	string	No	Description of the status.

## 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>204: 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>204: 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:set-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.
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  204: 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>	0x000001370000006	SQL filter
<code>com.microsoft:correlation-filter:list</code>	0x000001370000009	Correlation filter
<code>com.microsoft:true-filter:list</code>	0x000001370000007	True filter representing 1=1
<code>com.microsoft:false-filter:list</code>	0x000001370000008	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](#)

# AMQP errors in Azure Service Bus

4/3/2019 • 2 minutes to read • [Edit Online](#)

This article provides some of the errors you receive when using AMQP with Azure Service Bus. They are all standard behaviors of the service. You can avoid them by making send/receive calls on the connection/link, which automatically recreates the connection/link.

## Link is closed

You see the following error when the AMQP connection and link are active but no calls (for example, send or receive) are made using the link for 10 minutes. So, the link is closed. The connection is still open.

```
amqp:linkdetach-forced:The link 'G2:7223832:user.tenant0.cud_0000000000-0000-0000-0000-000000000000' is  
force detached by the broker due to errors occurred in publisher(link164614). Detach origin:  
AmqpMessagePublisher.IdleTimerExpired: Idle timeout: 00:10:00.  
TrackingId:000000000000000000000000000000_G2_B3, SystemTracker:mynamespace:Topic:MyTopic,  
Timestamp:2/16/2018 11:10:40 PM
```

## Connection is closed

You see the following error on the AMQP connection when all links in the connection have been closed because there was no activity (idle) and a new link has not been created in 5 minutes.

```
Error{condition=amqp:connection:forced, description='The connection was inactive for more than the allowed  
30000 milliseconds and is closed by container 'LinkTracker'.  
TrackingId:000000000000000000000000000000_G21, SystemTracker:gateway5, Timestamp:2019-03-06T17:32:00',  
info=null}
```

## Link is not created

You see this error when a new AMQP connection is created but a link is not created within 1 minute of the creation of the AMQP Connection.

```
Error{condition=amqp:connection:forced, description='The connection was inactive for more than the allowed  
60000 milliseconds and is closed by container 'LinkTracker'.  
TrackingId:000000000000000000000000000000_G21, SystemTracker:gateway5, Timestamp:2019-03-06T18:41:51',  
info=null}
```

## 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](#)

# Azure Service Bus - use firewall rules

1/14/2020 • 2 minutes to read • [Edit Online](#)

For scenarios in which Azure Service Bus is only accessible from certain well-known sites, Firewall rules enable you to configure rules for accepting traffic originating from specific IPv4 addresses. For example, these addresses may be those of a corporate NAT gateway.

## When to use

If you are looking to setup Service Bus such that it should receive traffic only from a specified range of IP addresses and reject everything else, then you can leverage a *Firewall* to block Service Bus endpoints from other IP addresses. For example, you are using Service Bus with [Azure Express Route](#) to create private connections to your on-premises infrastructure.

## How filter rules are applied

The IP filter rules are applied at the Service Bus namespace level. Therefore, the rules apply to all connections from clients using any supported protocol.

Any connection attempt from an IP address that does not match an allowed IP rule on the Service Bus namespace is rejected as unauthorized. The response does not mention the IP rule.

## Default setting

By default, the **IP Filter** grid in the portal for Service Bus is empty. This default setting means that your namespace accepts connections any IP address. This default setting is equivalent to a rule that accepts the 0.0.0.0/0 IP address range.

## IP filter rule evaluation

IP filter rules are applied in order and the first rule that matches the IP address determines the accept or reject action.

### WARNING

Implementing Firewall rules can prevent other Azure services from interacting with Service Bus.

Trusted Microsoft services are not supported when IP Filtering (Firewall rules) are implemented, and will be made available soon.

Common Azure scenarios that don't work with IP Filtering (note that the list is **NOT** exhaustive) -

- Azure Stream Analytics
- Integration with Azure Event Grid
- Azure IoT Hub Routes
- Azure IoT Device Explorer

The below Microsoft services are required to be on a virtual network

- Azure App Service
- Azure Functions

## Creating a virtual network and firewall rule with Azure Resource Manager templates

### IMPORTANT

Firewalls and Virtual Networks are supported only in the **premium** tier of Service Bus.

The following Resource Manager template enables adding a virtual network rule to an existing Service Bus namespace.

Template parameters:

- **ipMask** is a single IPv4 address or a block of IP addresses in CIDR notation. For example, in CIDR notation 70.37.104.0/24 represents the 256 IPv4 addresses from 70.37.104.0 to 70.37.104.255, with 24 indicating the number of significant prefix bits for the range.

### NOTE

While there are no deny rules possible, the Azure Resource Manager template has the default action set to "**Allow**" which doesn't restrict connections. When making Virtual Network or Firewalls rules, we must change the "**defaultAction**"

from

```
"defaultAction": "Allow"
```

to

```
"defaultAction": "Deny"
```

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "servicebusNamespaceName": {
            "type": "string",
            "metadata": {
                "description": "Name of the Service Bus namespace"
            }
        },
        "location": {
            "type": "string",
            "metadata": {
                "description": "Location for Namespace"
            }
        }
    },
    "variables": {
        "namespaceNetworkRuleSetName": "[concat(parameters('servicebusNamespaceName'), concat('/', 'default'))]"
    },
    "resources": [
        {
            "apiVersion": "2018-01-01-preview",
            "name": "[parameters('servicebusNamespaceName')]",
            "type": "Microsoft.ServiceBus/namespaces",
            "location": "[parameters('location')]",
            "sku": {
                "name": "Premium",
                "tier": "Premium"
            },
            "properties": { }
        },
        {
            "apiVersion": "2018-01-01-preview",
            "name": "[variables('namespaceNetworkRuleSetName')]",
            "type": "Microsoft.ServiceBus/namespaces/networkruleset",
            "dependsOn": [
                "[concat('Microsoft.ServiceBus/namespaces/', parameters('servicebusNamespaceName'))]"
            ],
            "properties": {
                "virtualNetworkRules": [<YOUR EXISTING VIRTUAL NETWORK RULES>],
                "ipRules": [
                    {
                        "ipMask": "10.1.1.1",
                        "action": "Allow"
                    },
                    {
                        "ipMask": "11.0.0.0/24",
                        "action": "Allow"
                    }
                ],
                "defaultAction": "Deny"
            }
        }
    ],
    "outputs": { }
}
```

To deploy the template, follow the instructions for [Azure Resource Manager](#).

## Next steps

For constraining access to Service Bus to Azure virtual networks, see the following link:

- Virtual Network Service Endpoints for Service Bus

# Use Virtual Network service endpoints with Azure Service Bus

1/14/2020 • 4 minutes to read • [Edit Online](#)

The integration of Service Bus with [Virtual Network \(VNet\) service endpoints](#) enables secure access to messaging capabilities from workloads like virtual machines that are bound to virtual networks, with the network traffic path being secured on both ends.

Once configured to be bound to at least one virtual network subnet service endpoint, the respective Service Bus namespace will no longer accept traffic from anywhere but authorized virtual network(s). From the virtual network perspective, binding a Service Bus namespace to a service endpoint configures an isolated networking tunnel from the virtual network subnet to the messaging service.

The result is a private and isolated relationship between the workloads bound to the subnet and the respective Service Bus namespace, in spite of the observable network address of the messaging service endpoint being in a public IP range.

## WARNING

Implementing Virtual Networks integration can prevent other Azure services from interacting with Service Bus.

Trusted Microsoft services are not supported when Virtual Networks are implemented.

Common Azure scenarios that don't work with Virtual Networks (note that the list is **NOT** exhaustive) -

- Azure Stream Analytics
- Integration with Azure Event Grid
- Azure IoT Hub Routes
- Azure IoT Device Explorer

The below Microsoft services are required to be on a virtual network

- Azure App Service
- Azure Functions

## IMPORTANT

Virtual Networks are supported only in [Premium tier](#) Service Bus namespaces.

## Enable service endpoints with Service Bus

An important consideration when using VNet service endpoints with Service Bus is that you should not enable these endpoints in applications that mix Standard and Premium tier Service Bus namespaces. Because Standard tier does not support VNets, the endpoint is restricted to Premium tier namespaces only.

## Advanced security scenarios enabled by VNet integration

Solutions that require tight and compartmentalized security, and where virtual network subnets provide the segmentation between the compartmentalized services, generally still need communication paths between services residing in those compartments.

Any immediate IP route between the compartments, including those carrying HTTPS over TCP/IP, carries the risk of exploitation of vulnerabilities from the network layer on up. Messaging services provide completely insulated communication paths, where messages are even written to disk as they transition between parties. Workloads in two distinct virtual networks that are both bound to the same Service Bus instance can communicate efficiently and reliably via messages, while the respective network isolation boundary integrity is preserved.

That means your security sensitive cloud solutions not only gain access to Azure industry-leading reliable and scalable asynchronous messaging capabilities, but they can now use messaging to create communication paths between secure solution compartments that are inherently more secure than what is achievable with any peer-to-peer communication mode, including HTTPS and other TLS-secured socket protocols.

## Binding Service Bus to Virtual Networks

*Virtual network rules* are the firewall security feature that controls whether your Azure Service Bus server accepts connections from a particular virtual network subnet.

Binding a Service Bus namespace to a virtual network is a two-step process. You first need to create a **Virtual Network service endpoint** on a Virtual Network subnet and enable it for "Microsoft.ServiceBus" as explained in the [service endpoint overview](#). Once you have added the service endpoint, you bind the Service Bus namespace to it with a *virtual network rule*.

The virtual network rule is an association of the Service Bus namespace with a virtual network subnet. While the rule exists, all workloads bound to the subnet are granted access to the Service Bus namespace. Service Bus itself never establishes outbound connections, does not need to gain access, and is therefore never granted access to your subnet by enabling this rule.

### Creating a virtual network rule with Azure Resource Manager templates

The following Resource Manager template enables adding a virtual network rule to an existing Service Bus namespace.

Template parameters:

- **namespaceName**: Service Bus namespace.
- **virtualNetworkingSubnetId**: Fully qualified Resource Manager path for the virtual network subnet; for example,

```
/subscriptions/{id}/resourceGroups/{rg}/providers/Microsoft.Network/virtualNetworks/{vnet}/subnets/default
```

for the default subnet of a virtual network.

#### NOTE

While there are no deny rules possible, the Azure Resource Manager template has the default action set to "**Allow**" which doesn't restrict connections. When making Virtual Network or Firewalls rules, we must change the "**defaultAction**" from

```
"defaultAction": "Allow"
```

to

```
"defaultAction": "Deny"
```

Template:

```
{
```

```
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
```

```

"contentVersion": "1.0.0.0",
"parameters": {
    "servicebusNamespaceName": {
        "type": "string",
        "metadata": {
            "description": "Name of the Service Bus namespace"
        }
    },
    "virtualNetworkName": {
        "type": "string",
        "metadata": {
            "description": "Name of the Virtual Network Rule"
        }
    },
    "subnetName": {
        "type": "string",
        "metadata": {
            "description": "Name of the Virtual Network Sub Net"
        }
    },
    "location": {
        "type": "string",
        "metadata": {
            "description": "Location for Namespace"
        }
    }
},
"variables": {
    "namespaceNetworkRuleSetName": "[concat(parameters('servicebusNamespaceName'), concat('/', 'default'))]",
    "subNetId": "[resourceId('Microsoft.Network/virtualNetworks/subnets/', parameters('virtualNetworkName'), parameters('subnetName'))]"
},
"resources": [
{
    "apiVersion": "2018-01-01-preview",
    "name": "[parameters('servicebusNamespaceName')]",
    "type": "Microsoft.ServiceBus/namespaces",
    "location": "[parameters('location')]",
    "sku": {
        "name": "Premium",
        "tier": "Premium"
    },
    "properties": { }
},
{
    "apiVersion": "2017-09-01",
    "name": "[parameters('virtualNetworkName')]",
    "location": "[parameters('location')]",
    "type": "Microsoft.Network/virtualNetworks",
    "properties": {
        "addressSpace": {
            "addressPrefixes": [
                "10.0.0.0/23"
            ]
        },
        "subnets": [
            {
                "name": "[parameters('subnetName')]",
                "properties": {
                    "addressPrefix": "10.0.0.0/23",
                    "serviceEndpoints": [
                        {
                            "service": "Microsoft.ServiceBus"
                        }
                    ]
                }
            }
        ]
    }
}
]

```

```
        },
      },
      {
        "apiVersion": "2018-01-01-preview",
        "name": "[variables('namespaceNetworkRuleSetName')]",
        "type": "Microsoft.ServiceBus/namespaces/networkruleset",
        "dependsOn": [
          "[concat('Microsoft.ServiceBus/namespaces/', parameters('servicebusNamespaceName'))]"
        ],
        "properties": {
          "virtualNetworkRules": [
            [
              {
                "subnet": {
                  "id": "[variables('subNetId')]"
                },
                "ignoreMissingVnetServiceEndpoint": false
              }
            ],
            "ipRules": [<YOUR EXISTING IP RULES>],
            "defaultAction": "Deny"
          }
        }
      ],
      "outputs": { }
    }
  }
```

To deploy the template, follow the instructions for [Azure Resource Manager](#).

## Next steps

For more information about virtual networks, see the following links:

- [Azure virtual network service endpoints](#)
- [Azure Service Bus IP filtering](#)

# Overview of Service Bus dead-letter queues

1/24/2020 • 4 minutes 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	The maximum number of allowed hops when forwarding between queues. Value is set to 4.
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 4 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;
    }
}
```

## Path to the dead-letter queue

You can access the dead-letter queue by using the following syntax:

```
<queue path>/$deadletterqueue
<topic path>/Subscriptions/<subscription path>/$deadletterqueue
```

If you are using the .NET SDK, you can get the path to the dead-letter queue by using the `SubscriptionClient.FormatDeadLetterPath()` method. This method takes the topic name/subscription name and suffixes with **/\$DeadLetterQueue**.

## 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

1/24/2020 • 3 minutes 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 throughput 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 queues, topics, and subscriptions](#)
- [Get started with Service Bus queues](#)
- [How to use Service Bus topics and subscriptions](#)

# Duplicate detection

1/24/2020 • 3 minutes to read • [Edit Online](#)

If an application fails due to 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 resend 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 with *MessageId* that was 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 when a failure occurs.

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.

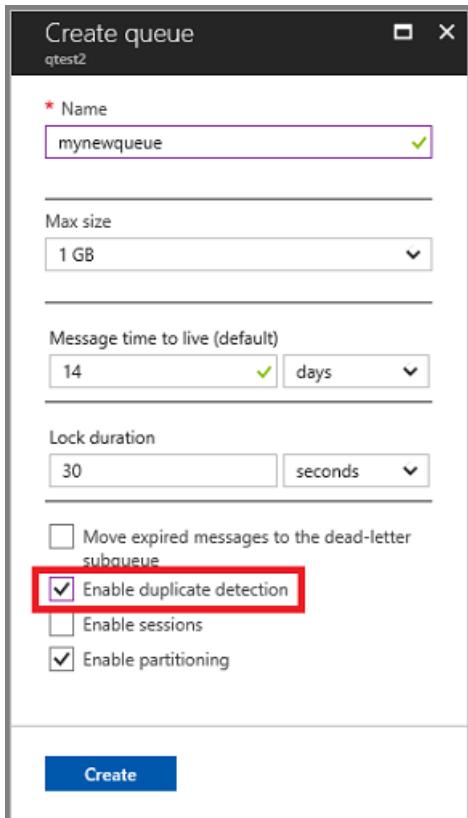
## NOTE

If the duplicate detection is enabled and session ID or partition key are not set, the message ID is used as the partition key.

If the message ID is also not set, .NET and AMQP libraries automatically generate a message ID for the message. For more information, see [Use of partition keys](#).

## 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.



#### IMPORTANT

You can't enable/disable duplicate detection after the queue is created. You can only do so at the time of creating the queue.

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 seven 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 managing a Service Bus queue named 'mynewqueue'. The left sidebar has a 'Search (Ctrl+/' input field and links for Overview, Diagnose and solve problems, Shared access policies, Properties (selected), Locks, and Automation script. The main area has sections for SETTINGS (Message time to live, Lock duration, Duplicate detection history, Maximum Delivery Count, Maximum size, Queue state), SUPPORT + TROUBLESHOOTING (New support request), and a bottom section for Queues and Topics.

**Duplicate detection history**

10	minutes
----	---------

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.

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

In scenarios where client code is unable to resubmit a message with the same *MessageId* as before, it is important to design messages which can be safely re-processed. This [blog post about idempotence](#) describes various techniques for how to do that.

# Message counters

1/24/2020 • 2 minutes 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.

## NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

With PowerShell, you can obtain the count as follows:

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

# Message deferral

1/24/2020 • 2 minutes 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 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 reordering 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 [IMessageReceiver.defer](#) or [IMessageReceiver.deferAsync](#) 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 subqueue), 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)`. For queues, you can use the [QueueClient](#), Topic subscriptions use the [SubscriptionClient](#).

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. 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.

## Next steps

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

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

# Message browsing

1/24/2020 • 2 minutes to read • [Edit Online](#)

Message browsing, or peeking, enables a Service Bus client to enumerate all messages that reside 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's 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 haven't yet been completed. However, because Peek returns a disconnected snapshot, the lock state of a message can't 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 and isn't 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 queues, topics, and subscriptions](#)
- [Get started with Service Bus queues](#)
- [How to use Service Bus topics and subscriptions](#)

# Chaining Service Bus entities with autoforwarding

1/24/2020 • 3 minutes to read • [Edit Online](#)

The Service Bus *autoforwarding* feature enables you to chain a queue or subscription to another queue or topic that is part of the same namespace. When autoforwarding 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). It is still possible to send a message to the destination entity directly.

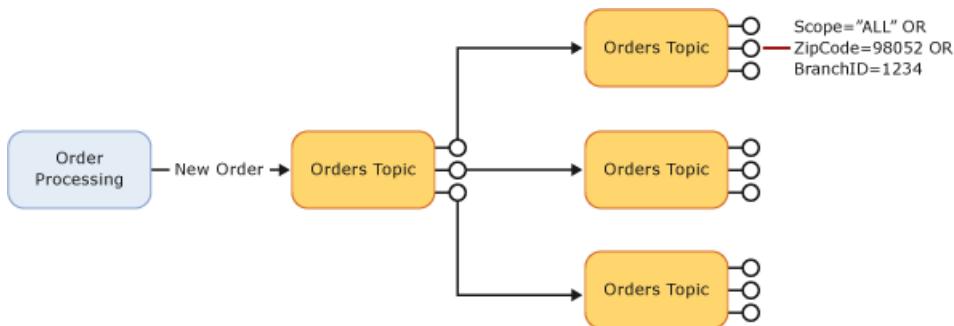
## Using autoforwarding

You can enable autoforwarding by setting the [QueueDescription.ForwardTo](#) or [SubscriptionDescription.ForwardTo](#) properties on the [QueueDescription](#) or [SubscriptionDescription](#) objects for the source, as in the following example:

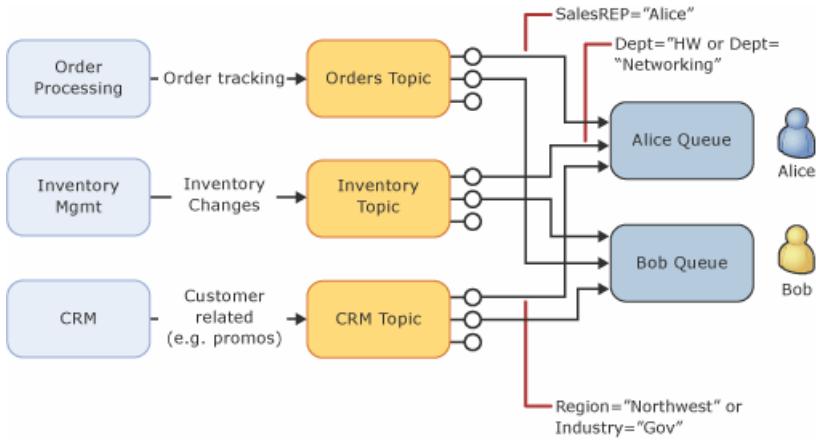
```
SubscriptionDescription srcSubscription = new SubscriptionDescription (srcTopic, srcSubscriptionName);
srcSubscription.ForwardTo = destTopic;
namespaceManager.CreateSubscription(srcSubscription));
```

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 autoforwarding 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 autoforwarding 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.

#### NOTE

When autoforwarding is setup, the value for AutoDeleteOnIdle on **both the Source and the Destination** is automatically set to the maximum value of the data type.

- On the Source side, autoforwarding acts as a receive operation. So the source which has autoforwarding setup is never really "idle".
- On the destination side, this is done to ensure that there is always a destination to forward the message to.

## Autoforwarding 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 autoforward 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 autoforwarding, 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

1/27/2020 • 3 minutes to read • [Edit Online](#)

This article discusses the transaction capabilities of Microsoft Azure Service Bus. Much of the discussion is illustrated by the [AMQP 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 connection = new ServiceBusConnection(connectionString);

var sender = new MessageSender(connection, QueueName);
var receiver = new MessageReceiver(connection, QueueName);
```

A simple transaction then uses these elements, as in the following example. To refer the full example, refer the [source code on GitHub](#):

```
var receivedMessage = await receiver.ReceiveAsync();

using (var ts = new TransactionScope(TransactionScopeAsyncFlowOption.Enabled))
{
    try
    {
        // do some processing
        if (receivedMessage != null)
            await receiver.CompleteAsync(receivedMessage.SystemProperties.LockToken);

        var myMsgBody = new MyMessage
        {
            Name = "Some name",
            Address = "Some street address",
            ZipCode = "Some zip code"
        };

        // send message
        var message = myMsgBody.AsMessage();
        await sender.SendAsync(message).ConfigureAwait(false);
        Console.WriteLine("Message has been sent");

        // complete the transaction
        ts.Complete();
    }
    catch (Exception ex)
    {
        // This rolls back send and complete in case an exception happens
        ts.Dispose();
        Console.WriteLine(ex.ToString());
    }
}
```

## 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](#)

# Distributed tracing and correlation through Service Bus messaging

1/24/2020 • 9 minutes 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](#) - install version 2.5-beta2 or higher
- [ASP.NET Core](#) - install 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 automatic 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>	<code>IList&lt;Message&gt; Messages</code> - List of messages being sent

OPERATION NAME	TRACKED API	SPECIFIC PAYLOAD PROPERTIES
Microsoft.Azure.ServiceBus.ScheduleMessage	<a href="#">MessageSender.ScheduleMessageAsync</a>	<p><code>Message Message</code> - Message being processed</p> <p><code>DateTimeOffset ScheduleEnqueueTimeUtc</code></p> <p>- Scheduled message offset</p> <p><code>long SequenceNumber</code> - Sequence number of scheduled message ('Stop' event payload)</p>
Microsoft.Azure.ServiceBus.Cancel	<a href="#">MessageSender.CancelScheduledMessageAsync</a>	<code>long SequenceNumber</code> - Sequence number of the message to be canceled
Microsoft.Azure.ServiceBus.Receive	<a href="#">MessageReceiver.ReceiveAsync</a>	<p><code>int RequestedMessageCount</code> - The maximum number of messages that could be received.</p> <p><code>IList&lt;Message&gt; Messages</code> - List of received messages ('Stop' event payload)</p>
Microsoft.Azure.ServiceBus.Peek	<a href="#">MessageReceiver.PeekAsync</a>	<p><code>int FromSequenceNumber</code> - The starting point from which to browse a batch of messages.</p> <p><code>int RequestedMessageCount</code> - The number of messages to retrieve.</p> <p><code>IList&lt;Message&gt; Messages</code> - List of received messages ('Stop' event payload)</p>
Microsoft.Azure.ServiceBus.ReceiveDeferred	<a href="#">MessageReceiver.ReceiveDeferredMessageAsync</a>	<p><code>IEnumerable&lt;long&gt; SequenceNumbers</code> - The list containing the sequence numbers to receive.</p> <p><code>IList&lt;Message&gt; Messages</code> - List of received messages ('Stop' event payload)</p>
Microsoft.Azure.ServiceBus.Complete	<a href="#">MessageReceiver.CompleteAsync</a>	<code>IList&lt;string&gt; LockTokens</code> - The list containing the lock tokens of the corresponding messages to complete.
Microsoft.Azure.ServiceBus.Abandon	<a href="#">MessageReceiver.AbandonAsync</a>	<code>string LockToken</code> - The lock token of the corresponding message to abandon.
Microsoft.Azure.ServiceBus.Defer	<a href="#">MessageReceiver.DeferAsync</a>	<code>string LockToken</code> - The lock token of the corresponding message to defer.
Microsoft.Azure.ServiceBus.DeadLetter	<a href="#">MessageReceiver.DeadLetterAsync</a>	<code>string LockToken</code> - The lock token of the corresponding message to dead letter.
Microsoft.Azure.ServiceBus.RenewLock	<a href="#">MessageReceiver.RenewLockAsync</a>	<p><code>string LockToken</code> - The lock token of the corresponding message to renew lock on.</p> <p><code>DateTime LockedUntilUtc</code> - New lock token expiry date and time in UTC format. ('Stop' event payload)</p>

OPERATION NAME	TRACKED API	SPECIFIC PAYLOAD PROPERTIES
Microsoft.Azure.ServiceBus.Process	Message Handler lambda function provided in <a href="#">IReceiverClient.RegisterMessageHandler</a>	<code>Message</code> <code>Message</code> - Message being processed.
Microsoft.Azure.ServiceBus.ProcessSession	Message Session Handler lambda function provided in <a href="#">IQueueClient.RegisterSessionHandler</a>	<code>Message</code> <code>Message</code> - Message being processed. <code>IMessageSession</code> <code>Session</code> - Session being processed
Microsoft.Azure.ServiceBus.AddRule	<a href="#">SubscriptionClient.AddRuleAsync</a>	<code>RuleDescription</code> <code>Rule</code> - The rule description that provides the rule to add.
Microsoft.Azure.ServiceBus.RemoveRule	<a href="#">SubscriptionClient.RemoveRuleAsync</a>	<code>string</code> <code>RuleName</code> - Name of the rule to remove.
Microsoft.Azure.ServiceBus.GetRules	<a href="#">SubscriptionClient.GetRulesAsync</a>	<code>IEnumerable&lt;RuleDescription&gt;</code> <code>Rules</code> - All rules associated with the subscription. ('Stop' payload only)
Microsoft.Azure.ServiceBus.AcceptMessageSession	<a href="#">ISessionClient.AcceptMessageSessionAsycn</a>	<code>string</code> <code>SessionId</code> - The sessionId present in the messages.
Microsoft.Azure.ServiceBus.GetSessionState	<a href="#">IMessageSession.GetStateAsync</a>	<code>string</code> <code>SessionId</code> - The sessionId present in the messages. <code>byte []</code> <code>State</code> - Session state ('Stop' event payload)
Microsoft.Azure.ServiceBus.SetSessionState	<a href="#">IMessageSession.SetStateAsync</a>	<code>string</code> <code>SessionId</code> - The sessionId present in the messages. <code>byte []</code> <code>State</code> - Session state
Microsoft.Azure.ServiceBus.RenewSessionLock	<a href="#">IMessageSession.RenewSessionLockAsync</a>	<code>string</code> <code>SessionId</code> - The sessionId present in the messages.
Microsoft.Azure.ServiceBus.Exception	any instrumented API	<code>Exception</code> <code>Exception</code> - 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)evt.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

- [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](#)

# .NET multi-tier application using Azure Service Bus queues

3/5/2019 • 14 minutes 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:

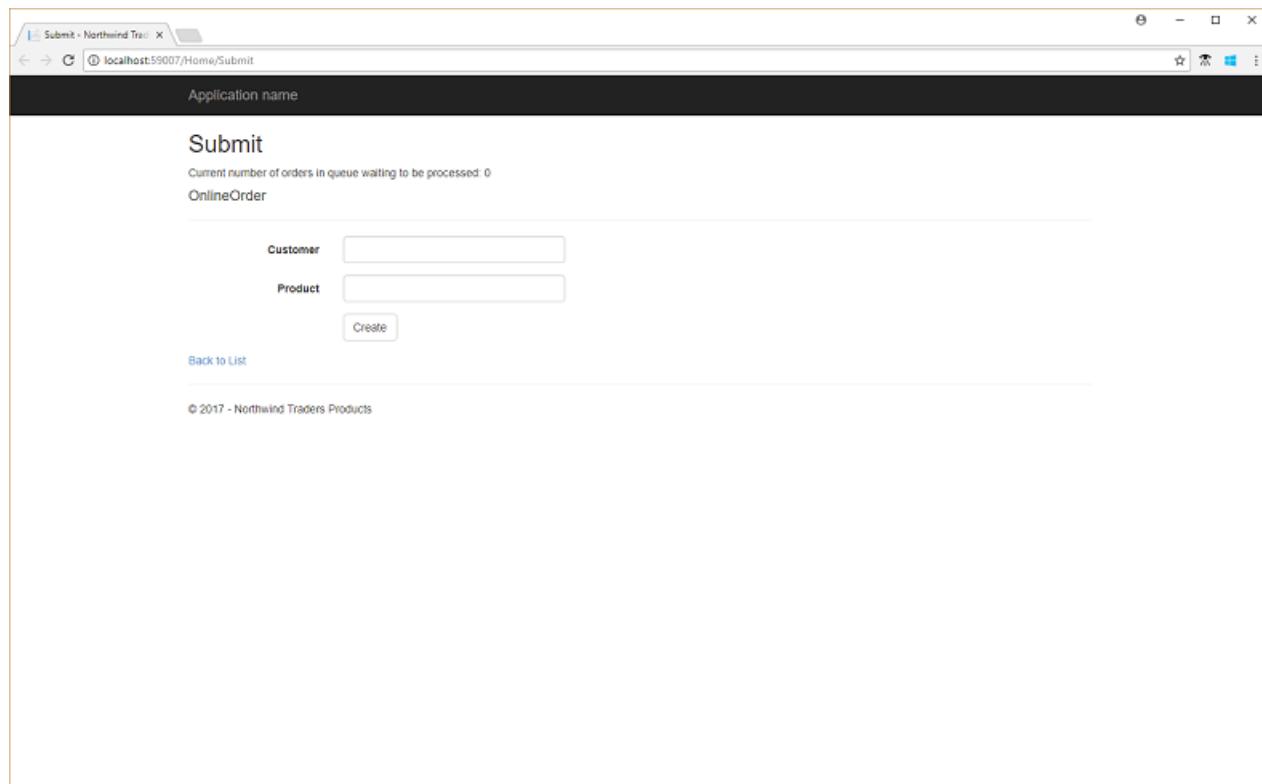
- How to enable your computer for Azure development with a single download and install.
- How to use Visual Studio to develop for Azure.
- How to create a multi-tier application in Azure using web and worker roles.
- How to communicate between tiers using Service Bus queues.

## 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 screenshot shows the completed application.

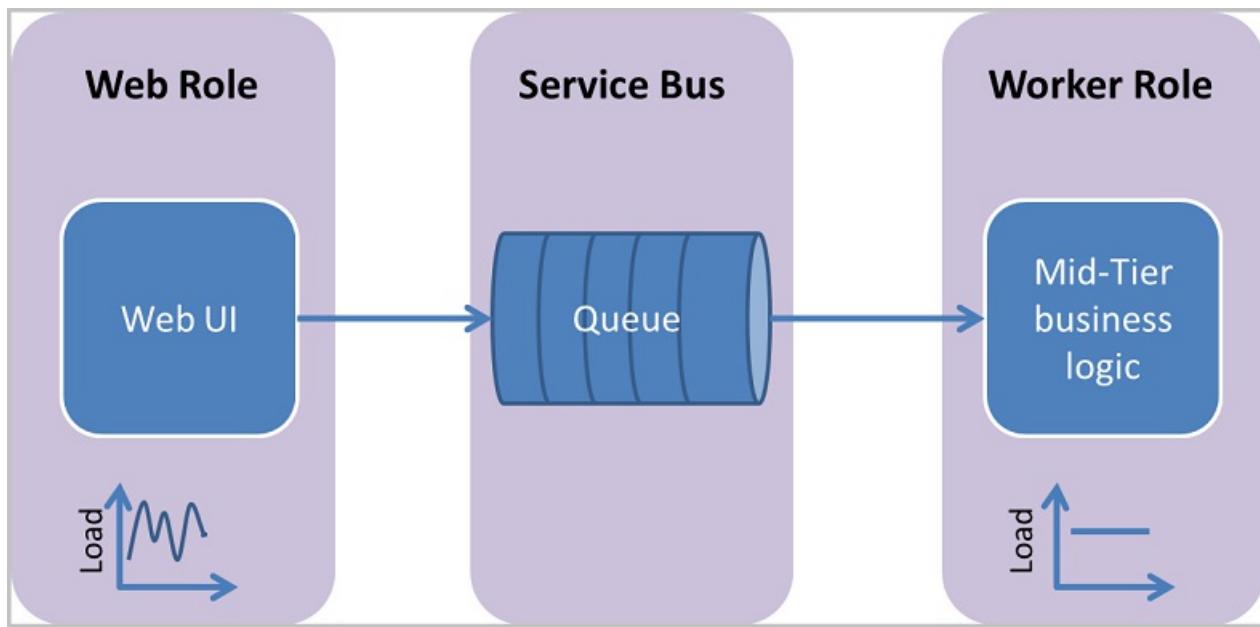


## Scenario overview: inter-role communication

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.

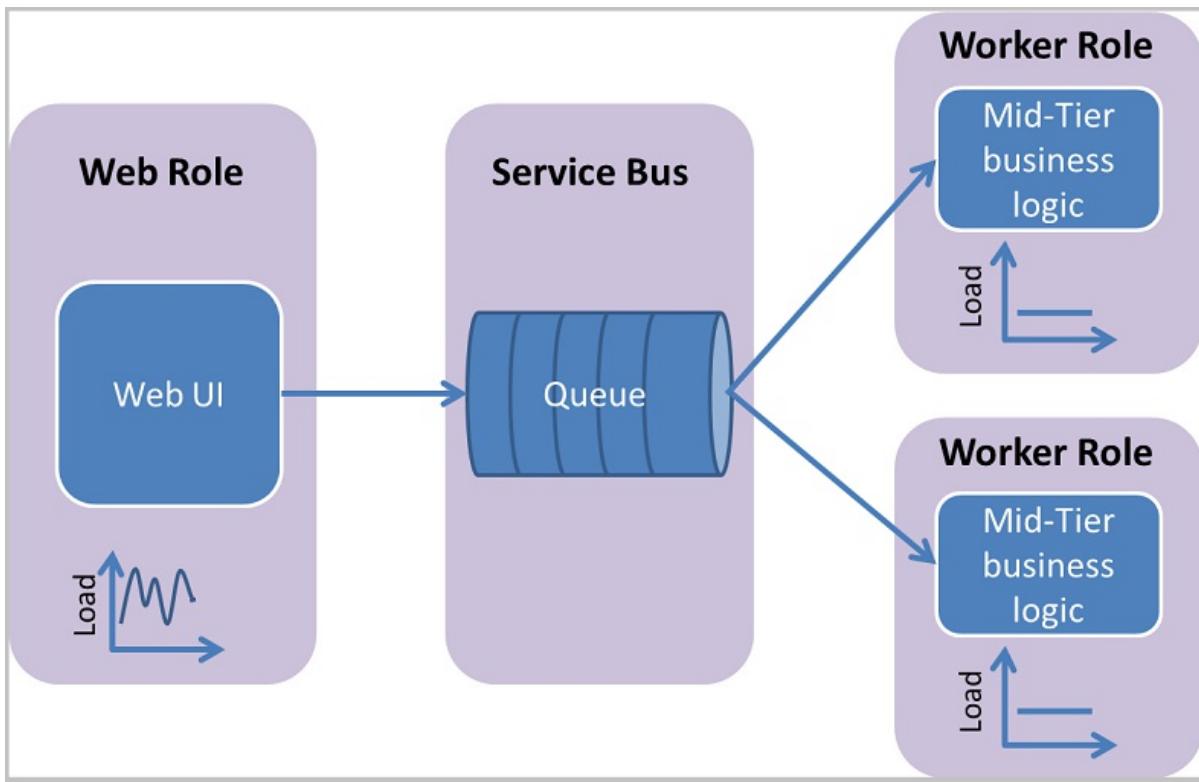
Using Service Bus messaging between the web and middle tiers decouples the two components. In contrast to direct messaging (that is, TCP or HTTP), the web tier does not connect to the middle tier directly; instead it pushes units of work, as messages, into Service Bus, which reliably retains them until 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.

## Create a namespace

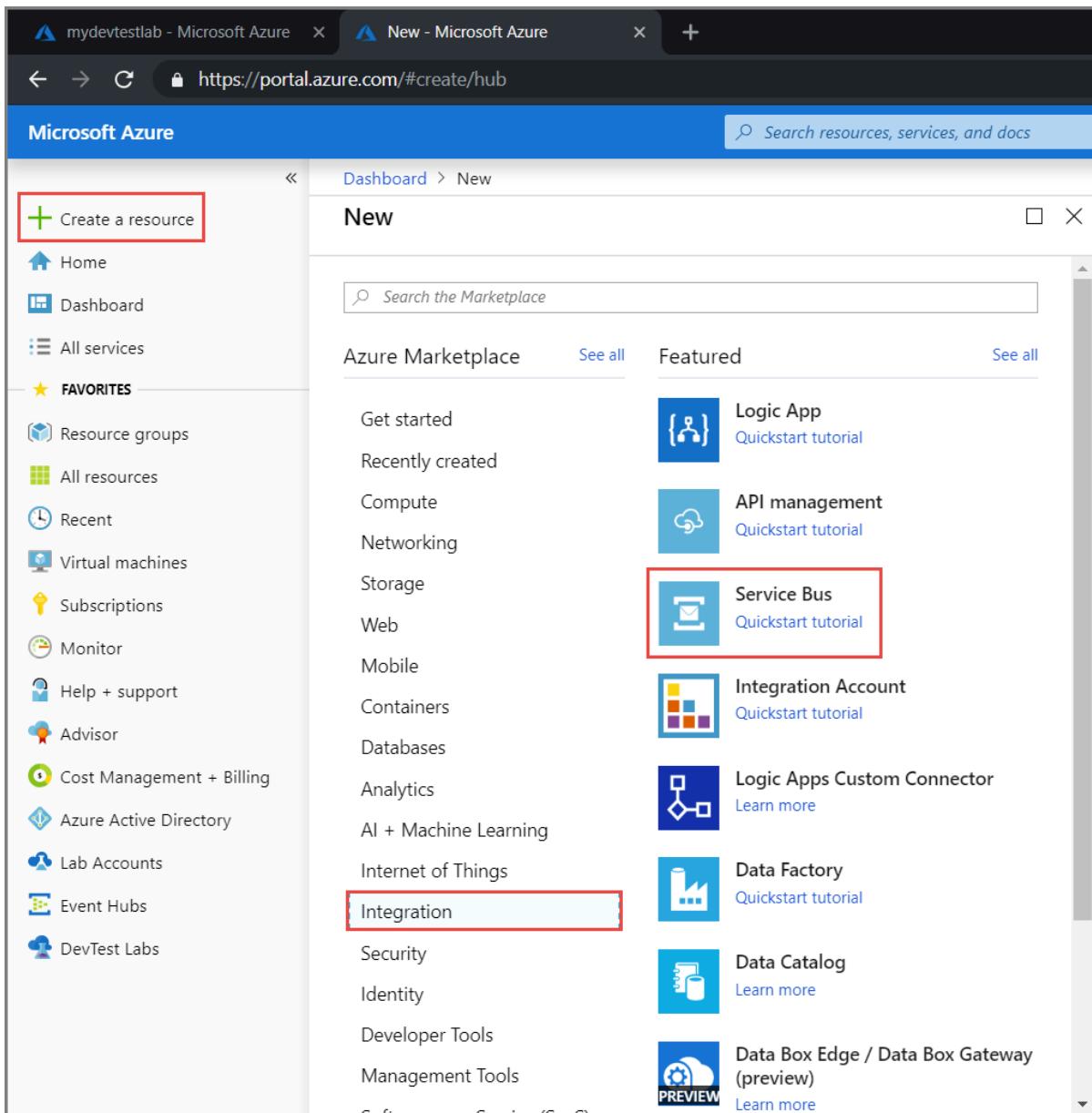
The first step is to create a *namespace*, and obtain a [Shared Access Signature \(SAS\)](#) key for that namespace. A namespace provides an application boundary for each application exposed through Service Bus. A SAS key is generated by the system when a namespace is created. The combination of namespace name and SAS key provides the credentials for Service Bus to authenticate access to an application.

## 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:

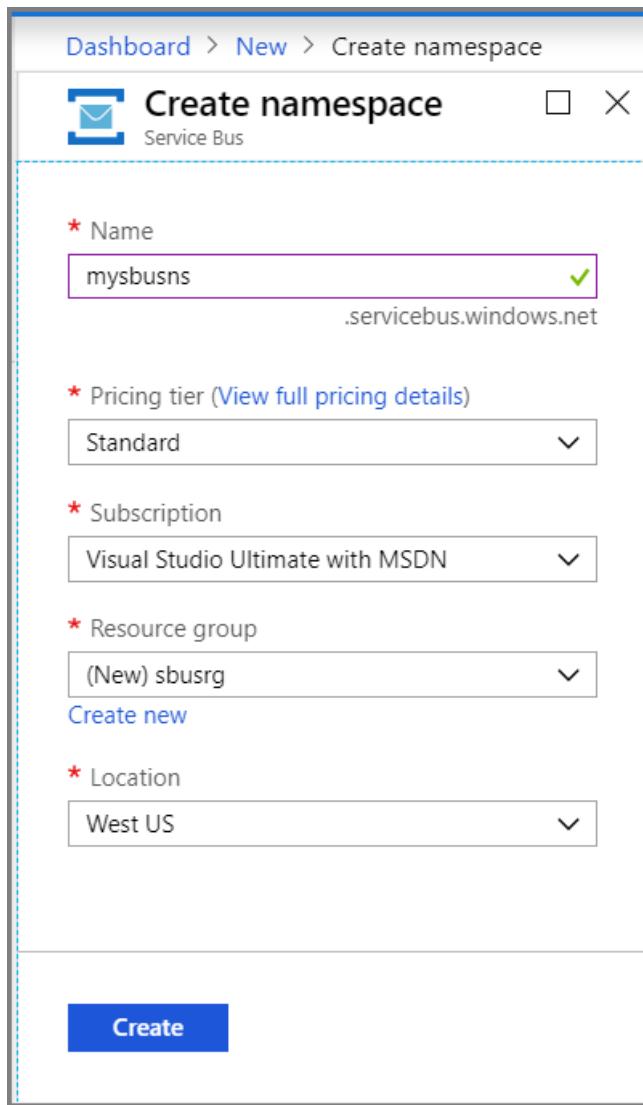
1. Sign in to the [Azure portal](#)
2. In the left navigation pane of the portal, select + **Create a resource**, select **Integration**, and then select **Service Bus**.



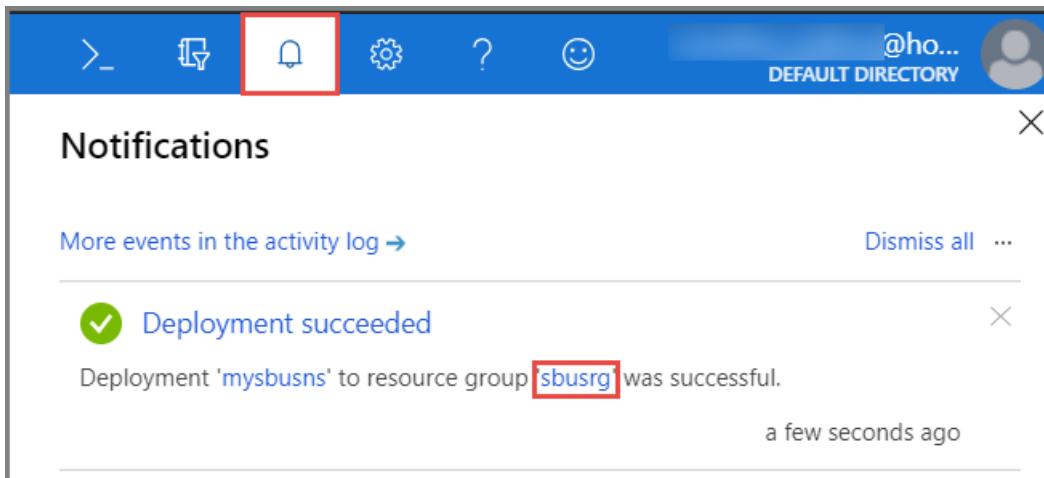
3. In the **Create namespace** dialog, do the following steps:

- Enter a **name for the namespace**. The system immediately checks to see if the name is available. For a list of rules for naming namespaces, see [Create Namespace REST API](#).
- Select the pricing tier (Basic, Standard, or Premium) for the namespace. If you want to use [topics and subscriptions](#), choose either Standard or Premium. Topics/subscriptions are not supported in the Basic pricing tier.
- If you selected the **Premium** pricing tier, follow these steps:
  - Specify the number of **messaging units**. The premium tier provides resource isolation at the CPU and memory level so that each workload runs in isolation. This resource container is called a messaging unit. A premium namespace has at least one messaging unit. You can select 1, 2, or 4 messaging units for each Service Bus Premium namespace. For more information, see [Service Bus Premium Messaging](#).
  - Specify whether you want to make the namespace **zone redundant**. The zone redundancy provides enhanced availability by spreading replicas across availability zones within one region at no additional cost. For more information, see [Availability zones in Azure](#).
  - For **Subscription**, choose an Azure subscription in which to create the namespace.
  - For **Resource group**, choose an existing resource group in which the namespace will live, or create a new one.

- f. For **Location**, choose the region in which your namespace should be hosted.
- g. Select **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.



4. Confirm that the service bus namespace is deployed successfully. To see the notifications, select the **bell icon (Alerts)** on the toolbar. Select the **name of the resource group** in the notification as shown in the image. You see the resource group that contains the service bus namespace.



5. On the **Resource group** page for your resource group, select your **service bus namespace**.

Subscription (change) : Visual Studio Ultimate with MSDN  
Subscription ID :  
Tags (change) : Click here to add tags

Filter by name... All types All locations No grouping

1 items Show hidden types

NAME	TYPE	LOCATION
mysbusns	Service Bus Namespace	West US

6. You see the home page for your service bus namespace.

Resource group (change) sbusrsg Status Active Location West US Subscription (change) Visual Studio Ultimate with MSDN

Subscription ID Created Tuesday, February 12, 2019 Updated Tuesday, February 12, 2019

Tags (change) Click here to add tags

Show data for the last: 1 hour 6 hours 12 hours 1 day 7 days 30 days

Requests

Messages

Queues Topics

NAME	STATUS	MAX SIZE	ENABLE PARTITIONING
no queues yet.			

## Get the connection string

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 rules with more constrained rights for regular senders and receivers. To copy the primary and secondary keys for your namespace, 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 **Primary Connection String**, 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.

## Create a web role

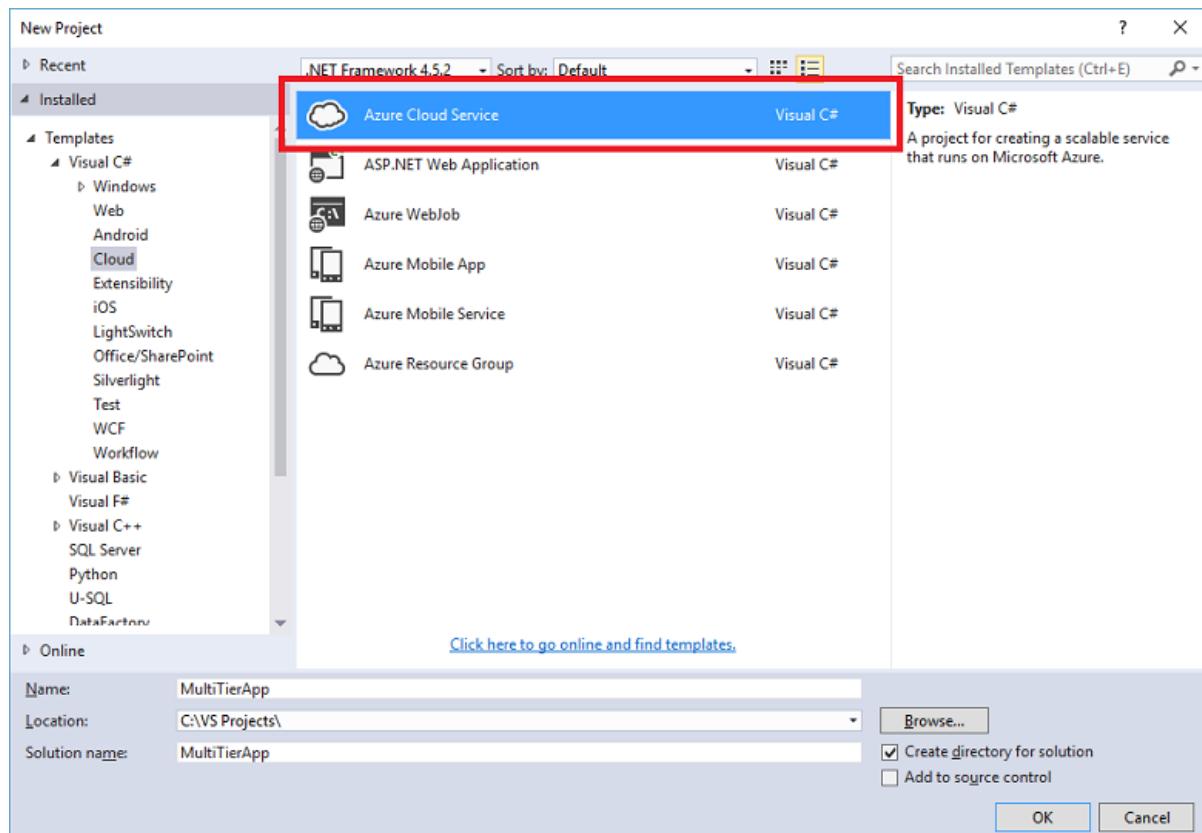
In this section, you build the front end of your application. First, you create the pages that your application displays. After that, add code that submits items to a Service Bus queue and displays status information about the queue.

### Create the project

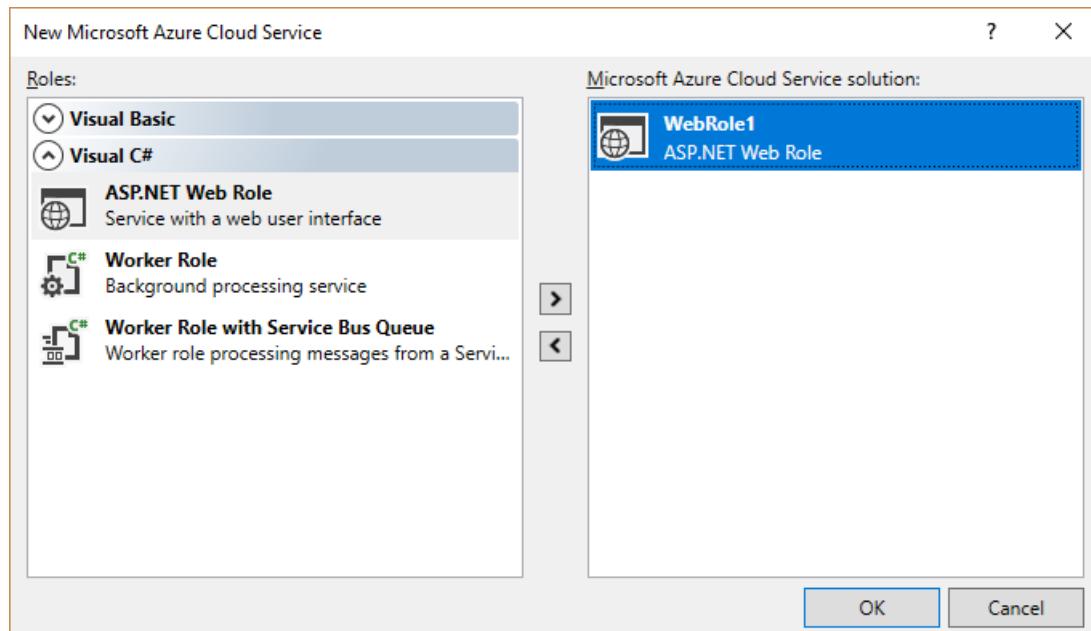
1. Using administrator privileges, start Visual Studio: right-click the **Visual Studio** program icon, and then click **Run as administrator**. The Azure compute emulator, discussed later in this article, requires that Visual Studio be started with administrator privileges.

In Visual Studio, on the **File** menu, click **New**, and then click **Project**.

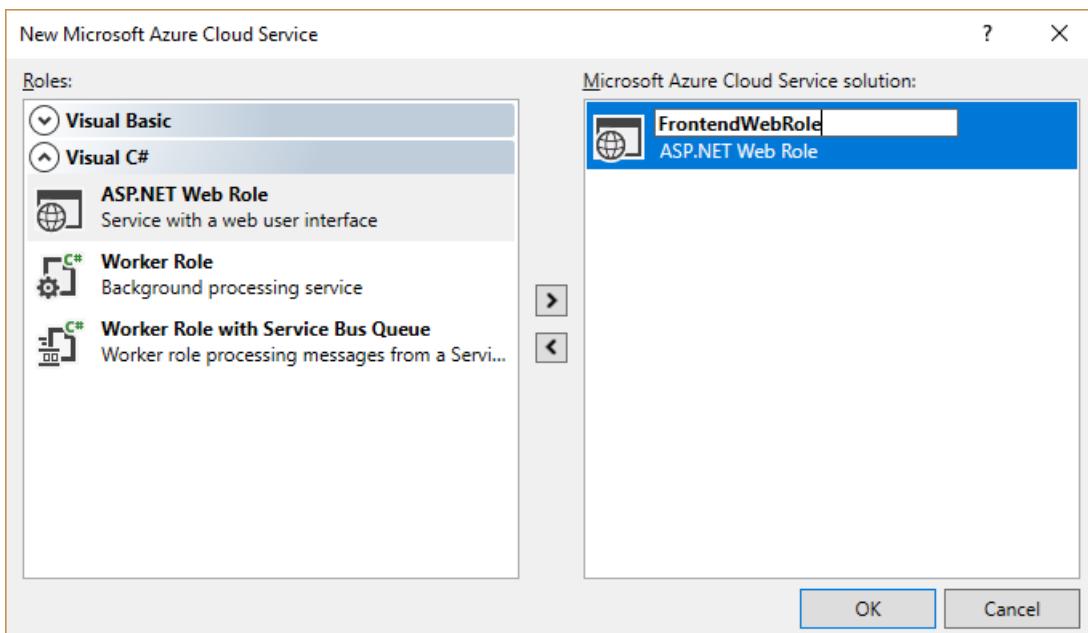
2. From **Installed Templates**, under **Visual C#**, click **Cloud** and then click **Azure Cloud Service**. Name the project **MultiTierApp**. Then click **OK**.



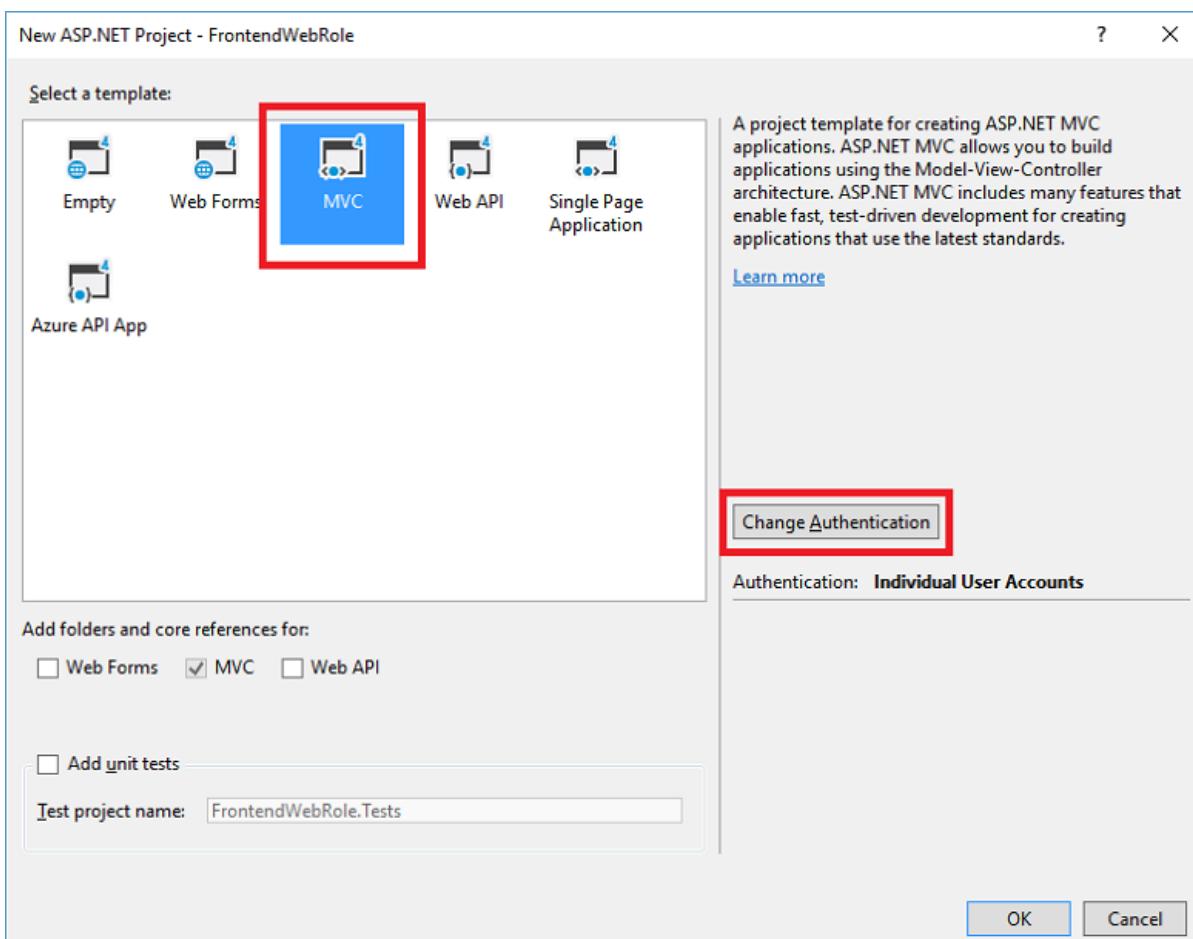
3. From the **Roles** pane, double-click **ASP.NET Web Role**.



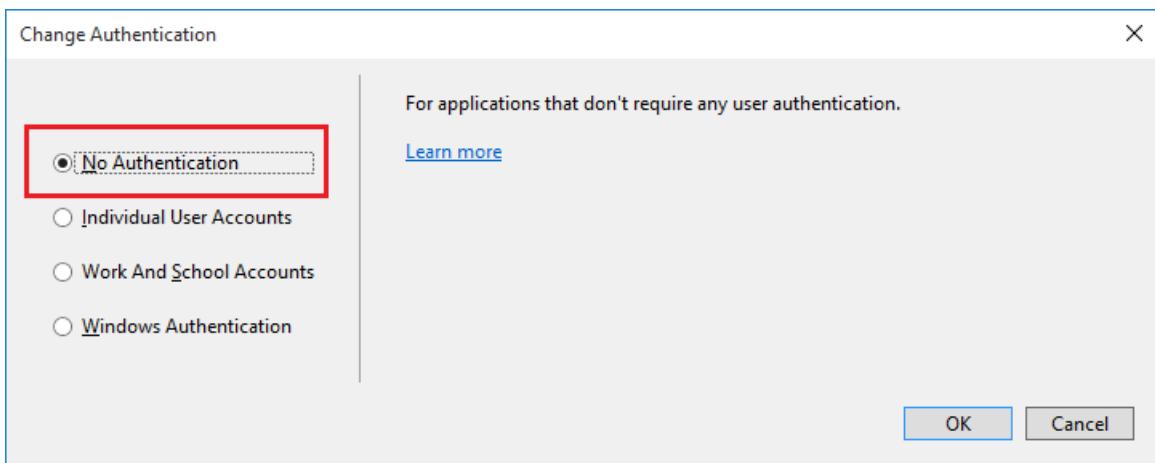
4. Hover over **WebRole1** under **Azure Cloud Service solution**, click the pencil icon, and rename the web role to **FrontendWebRole**. Then click **OK**. (Make sure you enter "Frontend" with a lower-case 'e,' not "FrontEnd".)



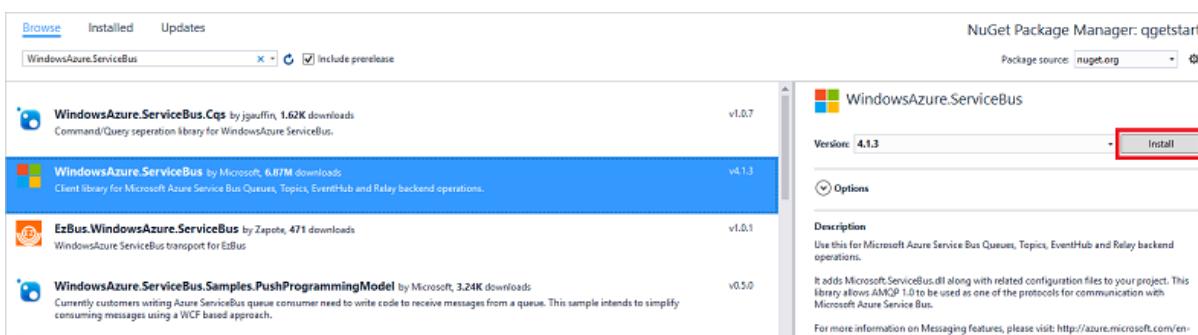
5. From the **New ASP.NET Project** dialog box, in the **Select a template** list, click **MVC**.



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();
        }

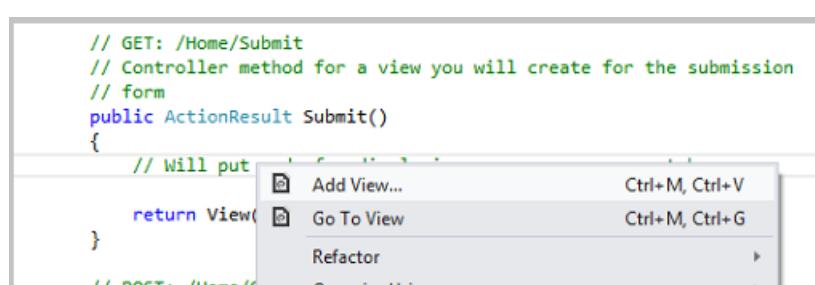
        // GET: /Home/Submit.
        // Controller method for a view you will create for the submission
        // form.
        public ActionResult Submit()
        {
            // Will put code for displaying queue message count here.

            return View();
        }

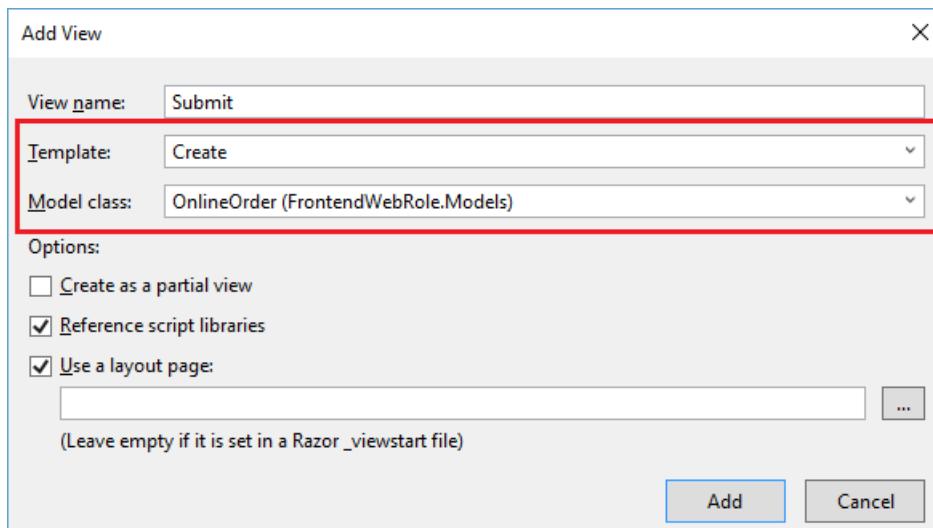
        // POST: /Home/Submit.
        // Controller method for handling submissions from the submission
        // form.
        [HttpPost]
        // Attribute to help prevent cross-site scripting attacks and
        // cross-site request forgery.
        [ValidateAntiForgeryToken]
        public ActionResult 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.



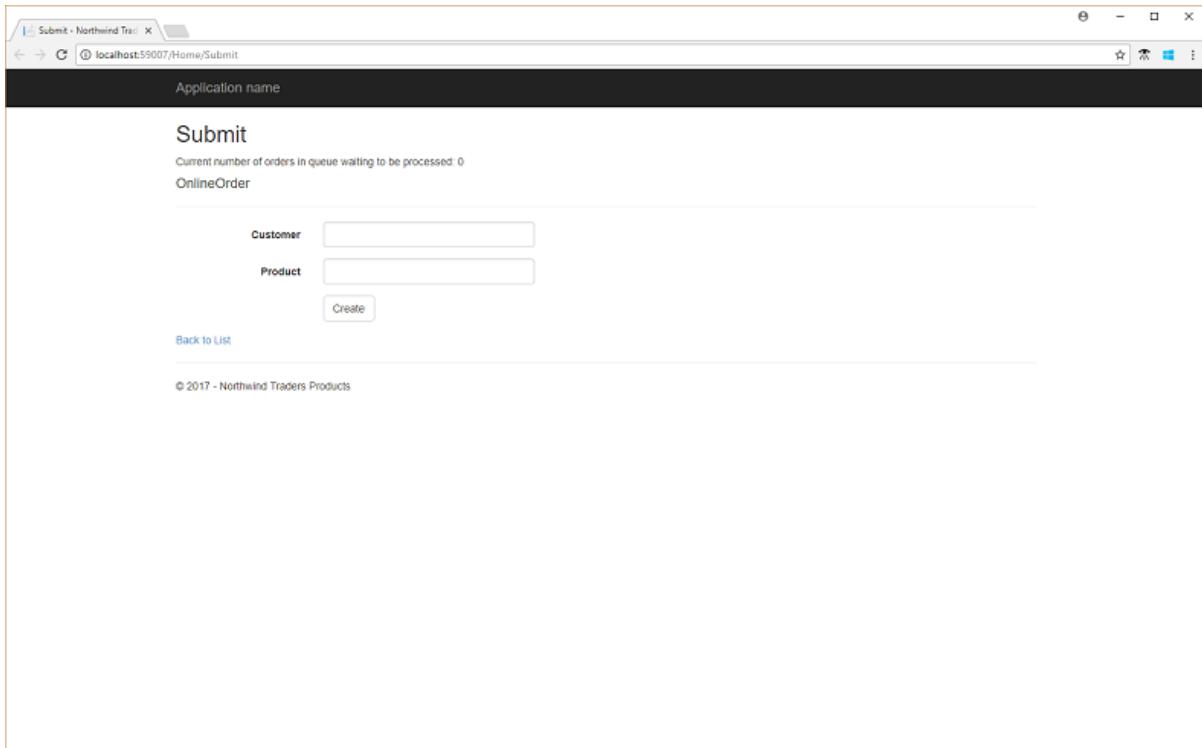
7. Click **Add**.
8. Now, change the displayed name of your application. In **Solution Explorer**, double-click the **Views\Shared\Layout.cshtml** file to open it in the Visual Studio editor.
9. Replace all occurrences of **My ASP.NET Application** with **Northwind Traders Products**.
10. Remove the **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.

```

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);
    }
}
```

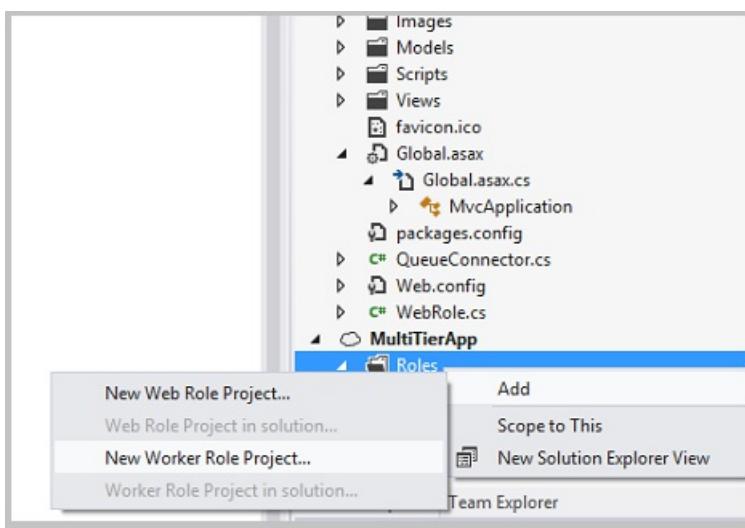
9. You can now run the application again. Each time you submit an order, the message count increases.

The screenshot shows a web application titled "Submit - Northwind Traders". The URL is "localhost:59007/Home/Submit". The page has a header "Application name" and a title "Submit". It displays a message: "Current number of orders in queue waiting to be processed: 1" followed by "OnlineOrder". Below this are two input fields: "Customer" and "Product", each with a corresponding text input box. A "Create" button is located below the product field. At the bottom left is a "Back to List" link, and at the bottom right is a copyright notice: "© 2017 - Northwind Traders Products".

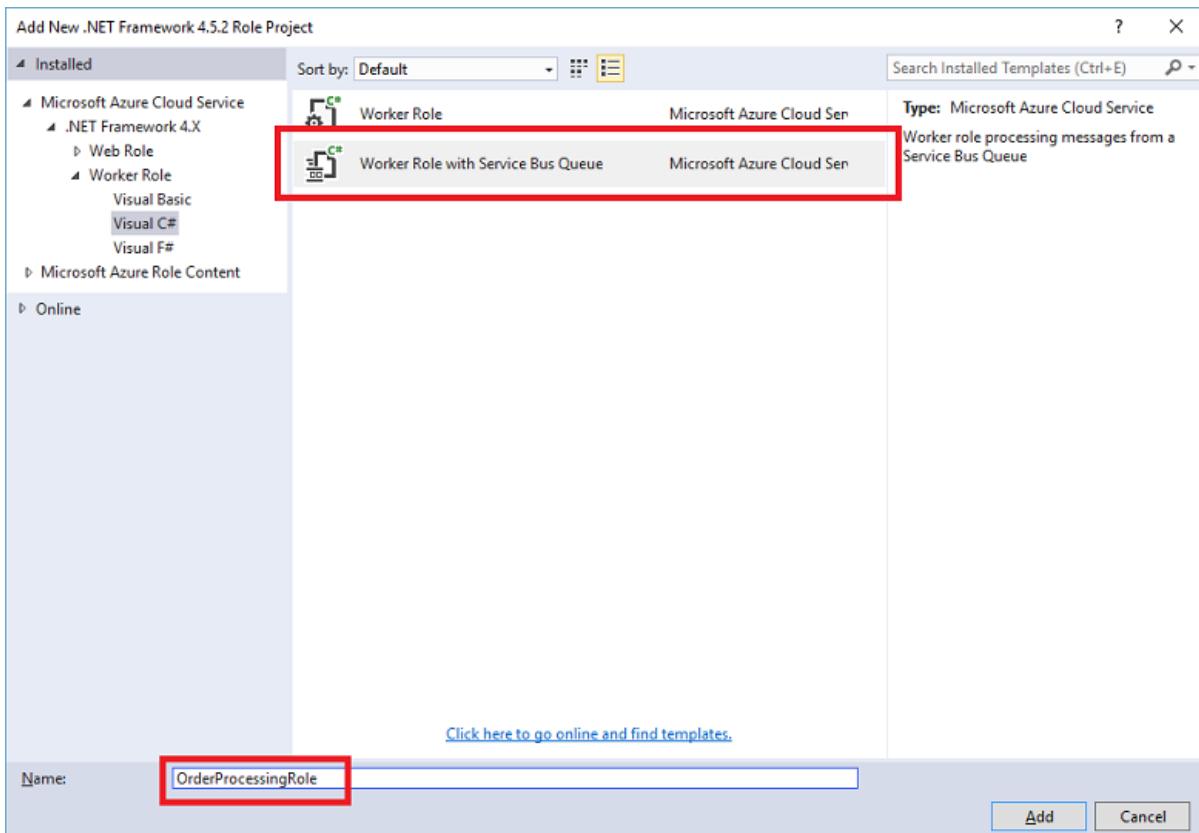
## Create the worker role

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.

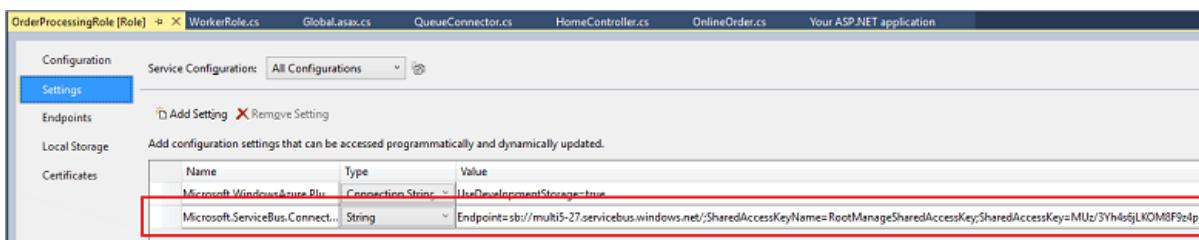
1. Make sure you have connected Visual Studio to your Azure account.
2. In Visual Studio, in **Solution Explorer** right-click the **Roles** folder under the **MultiTierApp** project.
3. Click **Add**, and then click **New Worker Role Project**. The **Add New Role Project** dialog box appears.



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";
```

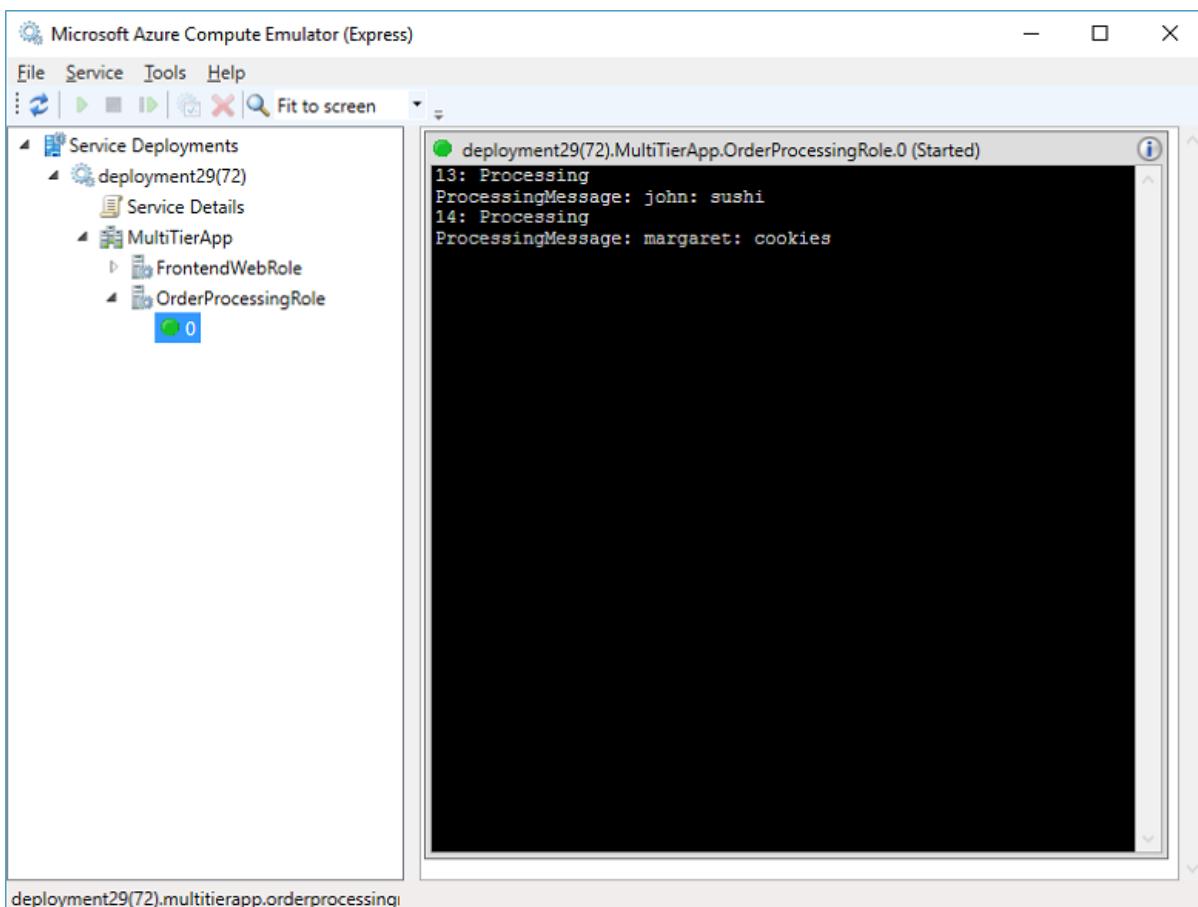
12. Add the following using statement at the top of the **WorkerRole.cs** file.

```
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:

- [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](#)

# Use PowerShell to manage Service Bus resources

1/24/2020 • 4 minutes 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](#).

## NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

## 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-AzServiceBusNamespace](#), [New-AzServiceBusNamespace](#), [Remove-AzServiceBusNamespace](#), and [Set-AzServiceBusNamespace](#) 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-AzServiceBusNamespace -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-AzServiceBusNamespace -ResourceGroup $ResGrpName -NamespaceName $Namespace
}
else
{
    Write-Host "The $Namespace namespace does not exist."
    Write-Host "Creating the $Namespace namespace in the $Location region..."
    New-AzServiceBusNamespace -ResourceGroup $ResGrpName -NamespaceName $Namespace -Location $Location
    $CurrentNamespace = Get-AzServiceBusNamespace -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-AzServiceBusAuthorizationRule](#), [Get-AzServiceBusAuthorizationRule](#), [Set-AzServiceBusAuthorizationRule](#), and [Remove-AzServiceBusAuthorizationRule](#) cmdlets.

```

# Query to see if rule exists
$CurrentRule = Get-AzServiceBusAuthorizationRule -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-AzServiceBusAuthorizationRule -ResourceGroup $ResGrpName -NamespaceName $Namespace -
AuthorizationRuleName $AuthRule -Rights @("Listen","Send")
    $CurrentRule = Get-AzServiceBusAuthorizationRule -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-AzServiceBusAuthorizationRule -ResourceGroup $ResGrpName -NamespaceName $Namespace -
AuthorizationRuleName $AuthRule

    Write-Host "Remove Send rights"
    $authRuleObj.Rights.Remove("Send")
    Set-AzServiceBusAuthorizationRule -ResourceGroup $ResGrpName -NamespaceName $Namespace -AuthRuleObj
$authRuleObj

    Write-Host "Add Send and Manage rights to the namespace"
    $authRuleObj.Rights.Add("Send")
    Set-AzServiceBusAuthorizationRule -ResourceGroup $ResGrpName -NamespaceName $Namespace -AuthRuleObj
$authRuleObj
    $authRuleObj.Rights.Add("Manage")
    Set-AzServiceBusAuthorizationRule -ResourceGroup $ResGrpName -NamespaceName $Namespace -AuthRuleObj
$authRuleObj

    Write-Host "Show value of primary key"
    $CurrentKey = Get-AzServiceBusKey -ResourceGroup $ResGrpName -NamespaceName $Namespace -Name $AuthRule

    Write-Host "Remove this authorization rule"
    Remove-AzServiceBusAuthorizationRule -ResourceGroup $ResGrpName -NamespaceName $Namespace -Name $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-AzServiceBusQueue -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-AzServiceBusQueue -ResourceGroup $ResGrpName -NamespaceName $Namespace -QueueName $QueueName -
EnablePartitioning $True
    $CurrentQ = Get-AzServiceBusQueue -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-AzServiceBusQueue](#) 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-AzServiceBusQueue -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](#)

# Azure Service Bus metrics in Azure Monitor

1/27/2020 • 5 minutes to read • [Edit Online](#)

Service Bus metrics give 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.

## IMPORTANT

When there has not been any interaction with an entity for 2 hours, the metrics will start showing "0" as a value until the entity is no longer idle.

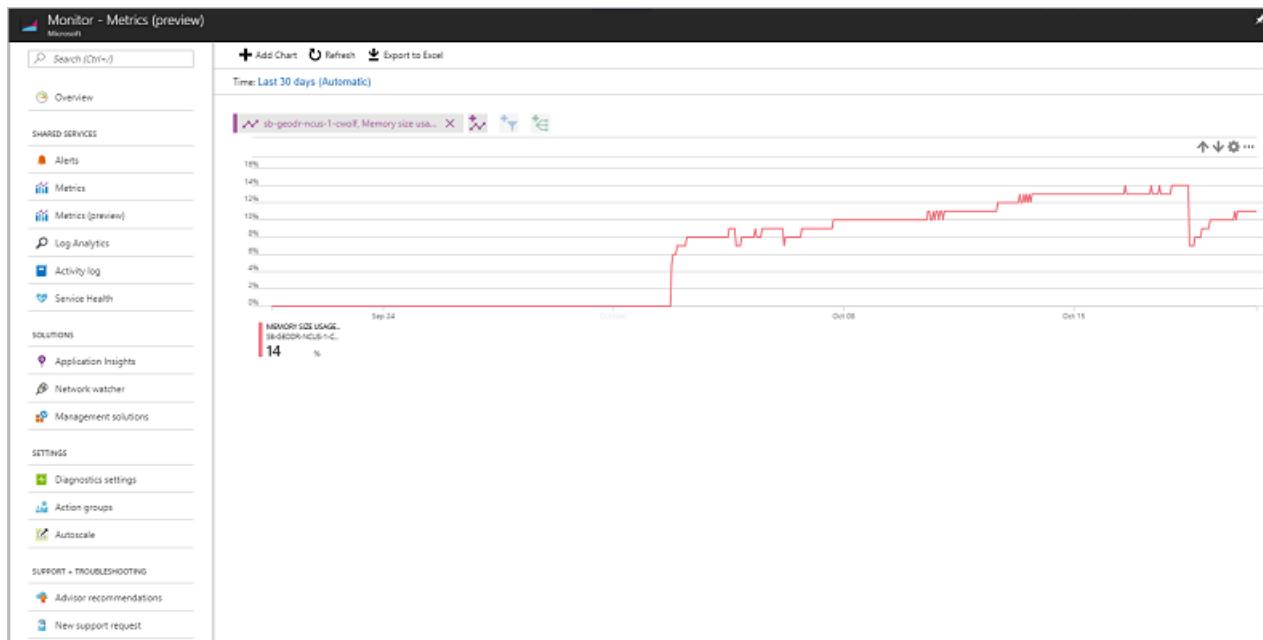
## Access metrics

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 Azure Monitor logs and Event Hubs. For more information, see [Metrics in Azure Monitor](#).

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 value is configured in [diagnostic settings](#) in Azure Monitor.

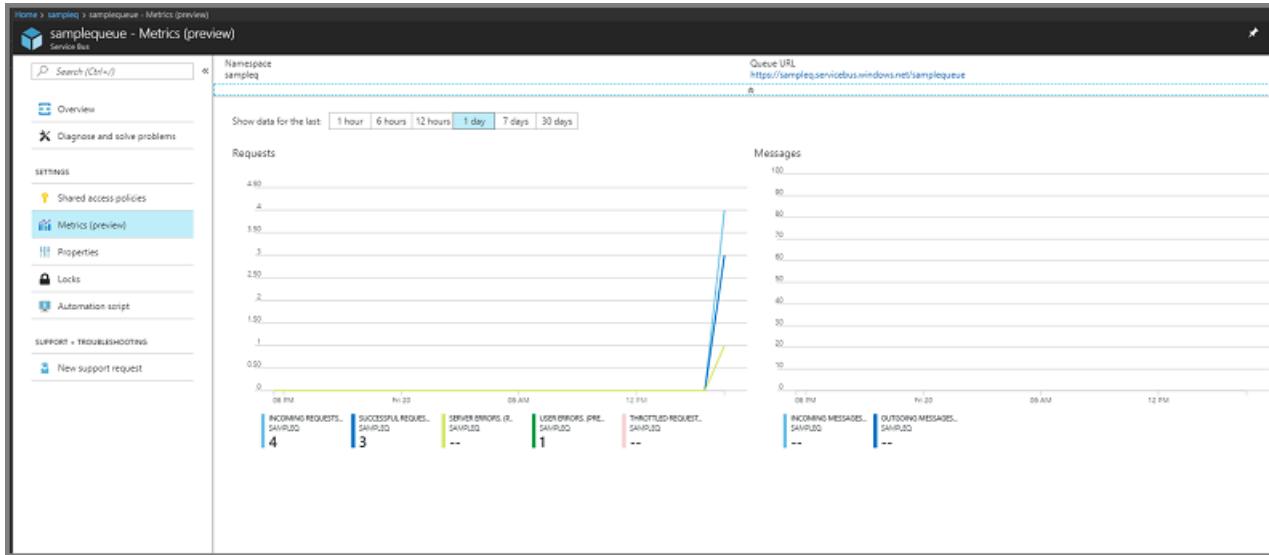
## Access metrics in the portal

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**. To

display metrics filtered to the scope of the entity, select the entity and then click **Metrics**.



For metrics supporting dimensions, you must filter with the desired dimension value.

## Billing

Metrics and Alerts on Azure Monitor are charged on a per alert basis. These charges should be available on the portal when the alert is setup and before it is saved.

Additional solutions that ingest metrics data are billed directly by those 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	<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>

METRIC NAME	DESCRIPTION
Successful Requests	<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>
Server Errors	<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 (see the following subsection)	<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	<p>The number of requests that were throttled because the usage was exceeded.</p> <p>Unit: Count Aggregation Type: Total Dimension: EntityName</p>

### User errors

The following two types of errors are classified as user errors:

1. Client-side errors (In HTTP that would be 400 errors).
2. Errors that occur while processing messages, such as [MessageLockLostException](#).

## Message metrics

METRIC NAME	DESCRIPTION
Incoming Messages	<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	<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>

METRIC NAME	DESCRIPTION
Messages	Count of messages in a queue/topic.  Unit: Count Aggregation Type: Average Dimension: EntityName
ActiveMessages	Count of active messages in a queue/topic.  Unit: Count Aggregation Type: Average Dimension: EntityName
Dead-lettered messages	Count of dead-lettered messages in a queue/topic.  Unit: Count Aggregation Type: Average Dimension: EntityName
Scheduled messages	Count of scheduled messages in a queue/topic.  Unit: Count Aggregation Type: Average Dimension: EntityName

## Connection metrics

METRIC NAME	DESCRIPTION
ActiveConnections	The number of active connections on a namespace as well as on an entity.  Unit: Count Aggregation Type: Total Dimension: EntityName

## Resource usage metrics

### NOTE

The following metrics are available only with the **premium** tier.

METRIC NAME	DESCRIPTION
CPU usage per namespace	The percentage CPU usage of the namespace.  Unit: Percent Aggregation Type: Maximum Dimension: EntityName

METRIC NAME	DESCRIPTION
Memory size usage per namespace	The percentage memory usage of the namespace.  Unit: Percent Aggregation Type: Maximum Dimension: EntityName

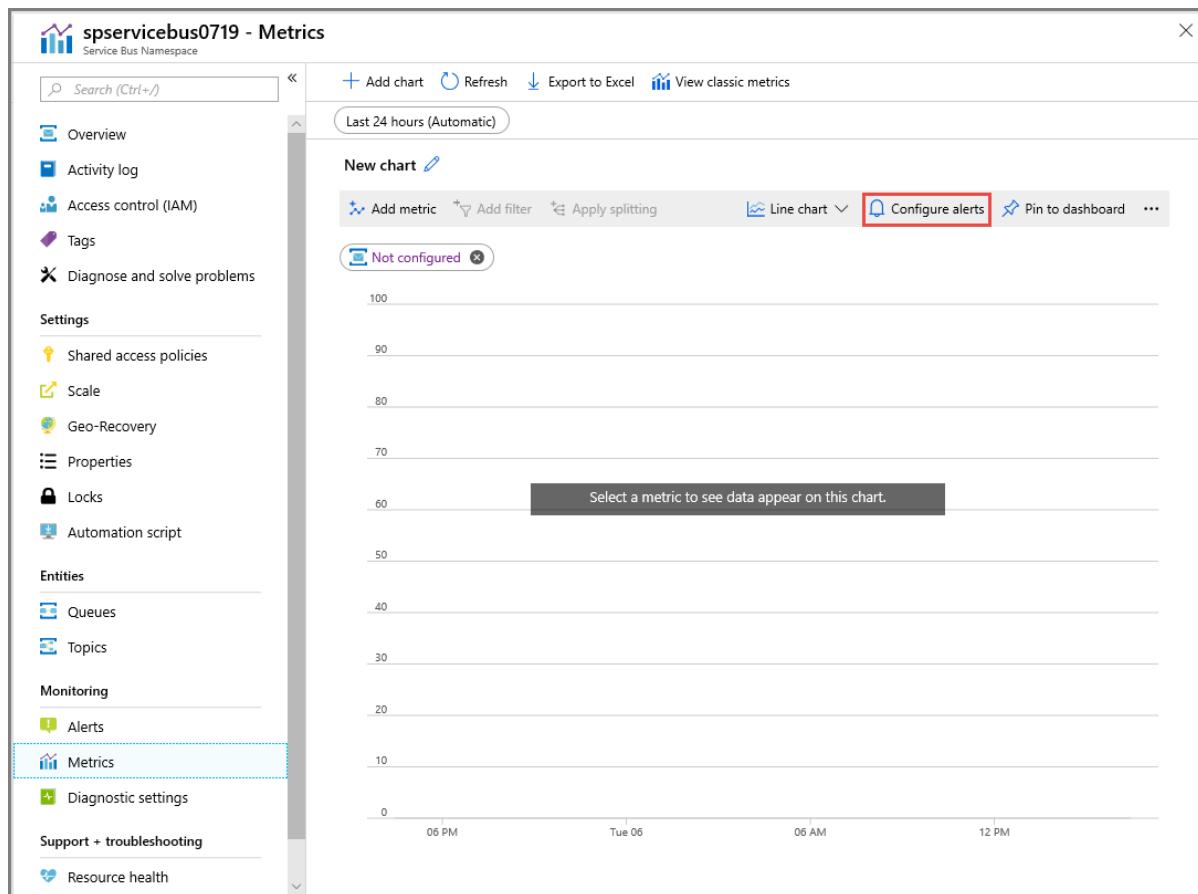
## Metrics dimensions

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.

## Set up alerts on metrics

1. On the **Metrics** tab of the **Service Bus Namespace** page, select **Configure alerts**.



2. Select the **Select target** option, and do the following actions on the **Select a resource** page:

- a. Select **Service Bus Namespaces** for the **Filter by resource type** field.
- b. Select your subscription for the **Filter by subscription** field.
- c. Select the **service bus namespace** from the list.
- d. Select **Done**.

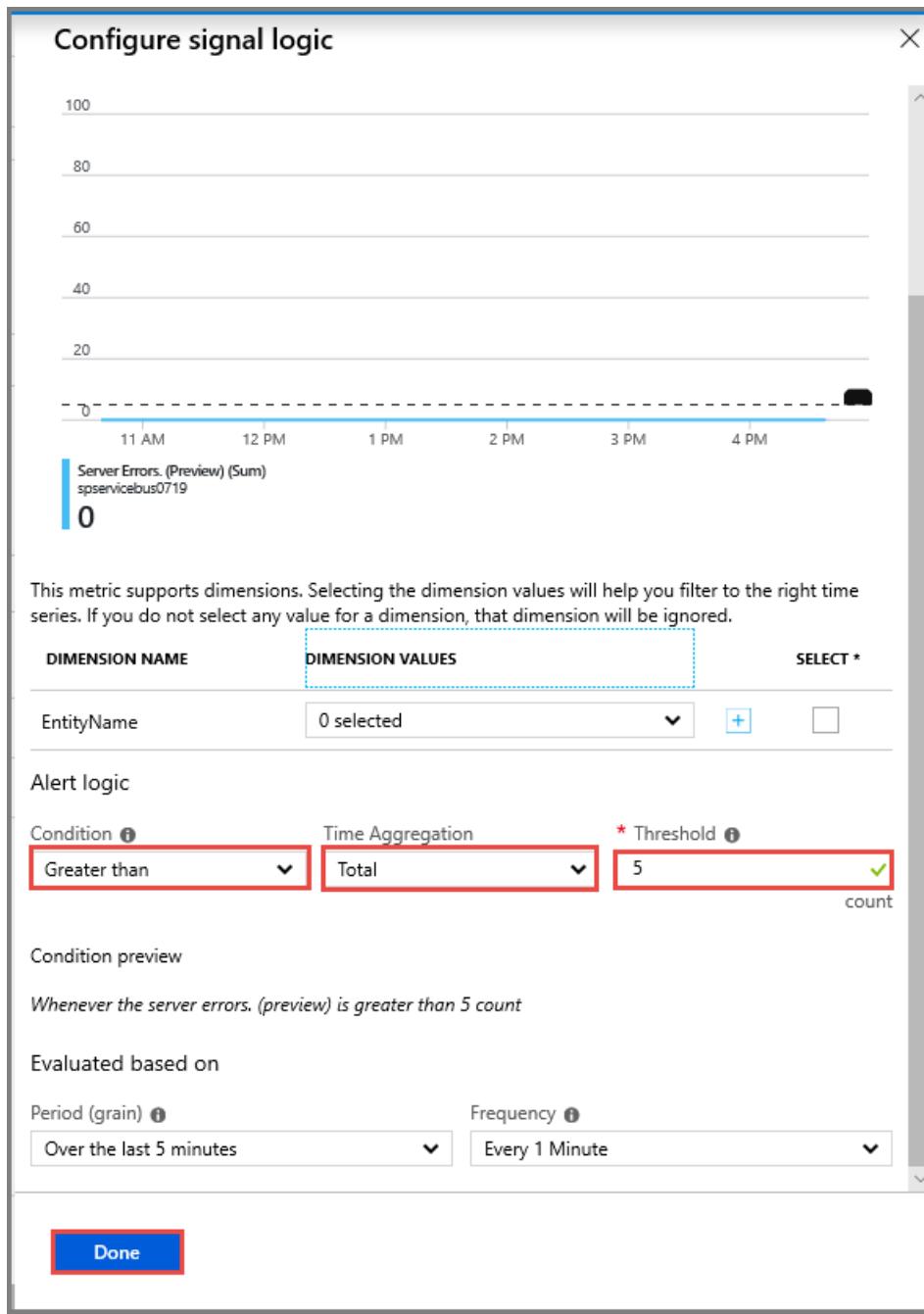
The screenshot shows the 'Create rule' wizard in the Azure portal. The current step is 'Select a resource'. The 'Subscription name' dropdown is set to 'spservicebus0719'. The list of resources shows 'spservicebus0719' selected. A red box highlights the 'Done' button at the bottom right.

### 3. Select **Add criteria**, and do the following actions on the **Configure signal logic** page:

- Select **Metrics for Signal type**.
- Select a signal. For example: **Service errors**.

The screenshot shows the 'Create rule' wizard in the Azure portal. The current step is 'Configure signal logic'. The 'Signal type' dropdown is set to 'Metrics'. The 'Server Errors. (Preview)' signal is selected. A red box highlights the 'Done' button at the bottom right.

- Select **Greater than** for **Condition**.
- Select **Total** for **Time Aggregation**.
- Enter **5** for **Threshold**.
- Select **Done**.



4. On the **Create rule** page, expand **Define alert details**, and do the following actions:
  - a. Enter a **name** for the alert.
  - b. Enter a **description** for the alert.
  - c. Select **severity** for the alert.

Create rule

Rules management

1. Define alert condition

2. Define alert details

\* Alert rule name ⓘ  
Too many server errors

\* Description  
There has been too many servers (greater than 5).

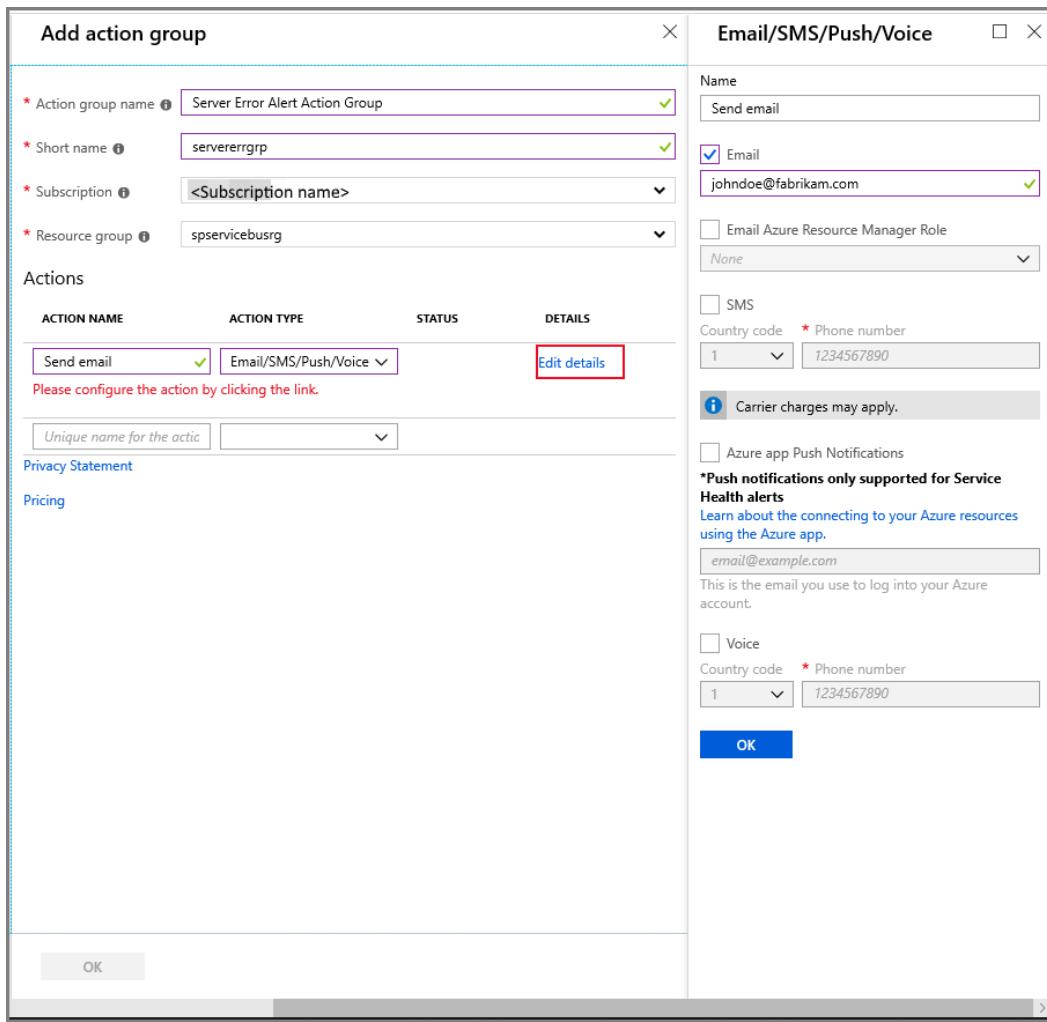
\* Severity ⓘ  
Sev 2

Enable rule upon creation  
Yes No

 It can take up to 10 minutes for a metric alert rule to become active.

3. Define action group

5. On the **Create rule** page, expand **Define action group**, select **New action group**, and do the following actions on the **Add action group** page.
  - a. Enter a name for the action group.
  - b. Enter a short name for the action group.
  - c. Select your subscription.
  - d. Select a resource group.
  - e. For this walkthrough, enter **Send email** for **ACTION NAME**.
  - f. Select **Email/SMS/Push/Voice** for **ACTION TYPE**.
  - g. Select **Edit details**.
  - h. On the **Email/SMS/Push/Voice** page, do the following actions:
    - a. Select **Email**.
    - b. Type the **email address**.
    - c. Select **OK**.



- d. On the **Add action group** page, select **OK**.
6. On the **Create rule** page, select **Create alert rule**.

**Create rule**  
Rules management

✓ 1. Define alert condition

✗ 2. Define alert details

\* Alert rule name ⓘ  
Too many server errors ✓

\* Description  
There has been too many servers (greater than 5). ✓

\* Severity ⓘ  
Sev 2

Enable rule upon creation  
 Yes  No

 It can take up to 10 minutes for a metric alert rule to become active.

✗ 3. Define action group

Notify your team via email and text messages or automate actions using webhooks, runbooks, functions, logic apps or integrating with external ITSM solutions. Learn more [here](#).

ACTION GROUP NAME	SUBSCRIPTION	ACTION GROUP TYPE	REMOVE
Server Error Alert Action Group	<Subscription name>	1 Email	

[Select action group](#) [+ New action group](#)

[Create alert rule](#)

## Next steps

See the [Azure Monitor overview](#).

# Service Bus management libraries

1/24/2020 • 2 minutes 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.azure.com/", 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);

```

## Complete code to create a queue

Here is the complete code to create a Service Bus queue:

```

using System;
using System.Threading.Tasks;

using Microsoft.Azure.Management.ServiceBus;
using Microsoft.Azure.Management.ServiceBus.Models;
using Microsoft.IdentityModel.Clients.ActiveDirectory;
using Microsoft.Rest;

namespace SBusADApp
{
    class Program
    {
        static void Main(string[] args)
        {
            CreateQueue().GetAwaiter().GetResult();
        }

        private static async Task CreateQueue()
        {
            try
            {
                var subscriptionID = "<SUBSCRIPTION ID>";
                var resourceGroupName = "<RESOURCE GROUP NAME>";
                var namespaceName = "<SERVICE BUS NAMESPACE NAME>";
                var queueName = "<NAME OF QUEUE YOU WANT TO CREATE>";

                var token = await GetToken();

                var creds = new TokenCredentials(token);
                var sbClient = new ServiceBusManagementClient(creds)
                {
                    SubscriptionId = subscriptionID,
                };

                var queueParams = new SBQueue();

                Console.WriteLine("Creating queue...");
                await sbClient.Queues.CreateOrUpdateAsync(resourceGroupName, namespaceName, queueName,
queueParams);
                Console.WriteLine("Created queue successfully.");
            }
            catch (Exception e)
            {
                Console.WriteLine("Could not create a queue...");
                Console.WriteLine(e.Message);
                throw e;
            }
        }
    }
}

```

```
private static async Task<string> GetToken()
{
    try
    {
        var tenantId = "<AZURE AD TENANT ID>";
        var clientId = "<APPLICATION/CLIENT ID>";
        var clientSecret = "<CLIENT SECRET>";

        var context = new AuthenticationContext($"https://login.microsoftonline.com/{tenantId}");

        var result = await context.AcquireTokenAsync(
            "https://management.azure.com/",
            new ClientCredential(clientId, clientSecret)
        );

        // If the token isn't a valid string, throw an error.
        if (string.IsNullOrEmpty(result.AccessToken))
        {
            throw new Exception("Token result is empty!");
        }

        return result.AccessToken;
    }
    catch (Exception e)
    {
        Console.WriteLine("Could not get a token...");
        Console.WriteLine(e.Message);
        throw e;
    }
}

}

}
```

#### IMPORTANT

For a complete example, see the [.NET management sample on GitHub](#).

## Next steps

[Microsoft.Azure.Management.ServiceBus API reference](#)

# Enable diagnostics logs for Service Bus

1/24/2020 • 2 minutes to read • [Edit Online](#)

When you start using your Azure Service Bus namespace, you might want to monitor how and when your namespace is created, deleted, or accessed. This article provides an overview of all the operational and diagnostics logs that are available.

Azure Service Bus currently supports activity and operational logs, which capture *management operations* that are performed on the Azure Service Bus namespace. Specifically, these logs capture the operation type, including queue creation, resources used, and the status of the operation.

## Operational logs schema

All logs are stored in JavaScript Object Notation (JSON) format in the following two locations:

- **AzureActivity**: Displays logs from operations and actions that are conducted against your namespace in the Azure portal or through Azure Resource Manager template deployments.
- **AzureDiagnostics**: Displays logs from operations and actions that are conducted against your namespace by using the API, or through management clients on the language SDK.

Operational log JSON strings include the elements listed in the following table:

NAME	DESCRIPTION
ActivityId	Internal ID, used to identify the specified activity
EventName	Operation name
ResourceId	Azure Resource Manager resource ID
SubscriptionId	Subscription ID
EventTimeString	Operation time
EventProperties	Operation properties
Status	Operation status
Caller	Caller of operation (the 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"
}
```

## Events and operations captured in operational logs

Operational logs capture all management operations that are performed on the Azure Service Bus namespace. Data operations are not captured, because of the high volume of data operations that are conducted on Azure Service Bus.

### NOTE

To help you better track data operations, we recommend using client-side tracing.

The following management operations are captured in operational logs:

SCOPE	OPERATION
Namespace	<ul style="list-style-type: none"> <li>Create Namespace</li> <li>Update Namespace</li> <li>Delete Namespace</li> </ul>
Queue	<ul style="list-style-type: none"> <li>Create Queue</li> <li>Update Queue</li> <li>Delete Queue</li> </ul>
Topic	<ul style="list-style-type: none"> <li>Create Topic</li> <li>Update Topic</li> <li>Delete Topic</li> </ul>
Subscription	<ul style="list-style-type: none"> <li>Create Subscription</li> <li>Update Subscription</li> <li>Delete Subscription</li> </ul>

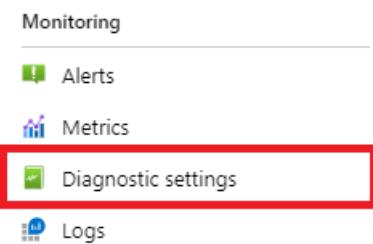
### NOTE

Currently, *Read* operations are not tracked in the operational logs.

## Enable operational logs

Operational logs are disabled by default. To enable diagnostics logs, do the following:

1. In the [Azure portal](#), go to your Azure Service Bus namespace and then, under **Monitoring**, select **Diagnostic settings**.



2. In the **Diagnostics settings** pane, select **Add diagnostic setting**.

A screenshot of the 'Diagnostics settings' pane. It has columns for 'Name', 'Storage account', 'Event hub', 'Log analytic', and 'Edit setting'. A message at the top says 'No diagnostic settings defined'. Below it is a red box around the '+ Add diagnostic setting' button. Further down, instructions say 'Click 'Add Diagnostic setting' above to configure the collection of the following data:' followed by a list: '• OperationalLogs' and '• AllMetrics'.

3. Configure the diagnostics settings by doing the following:

- a. In the **Name** box, enter a name for the diagnostics settings.
- b. Select one of the following three destinations for your diagnostics logs:
  - If you select **Archive to a storage account**, you need to configure the storage account where the diagnostics logs will be stored.
  - If you select **Stream to an event hub**, you need to configure the event hub that you want to stream the diagnostics logs to.
  - If you select **Send to Log Analytics**, you need to specify which instance of Log Analytics the diagnostics will be sent to.
- c. Select the **OperationalLogs** check box.

## Diagnostics settings

X

 Save  Discard  Delete

Name \*

test-operational-logs 

Archive to a storage account

Stream to an event hub

Send to Log Analytics

LOG

OperationalLogs

METRIC

AllMetrics

#### 4. Select **Save**.

The new settings take effect in about 10 minutes. The logs are displayed in the configured archival target, in the **Diagnostics logs** pane.

For more information about configuring diagnostics settings, see the [overview of Azure diagnostics logs](#).

## Next steps

To learn more about Service Bus, see:

- [Introduction to Service Bus](#)
- [Get started with Service Bus](#)

# Suspend and reactivate messaging entities (disable)

1/24/2020 • 2 minutes 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 screenshot 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 contains settings for the queue, including 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 (a dropdown menu). At the bottom, there is a 'Queue state' section with two buttons: 'Active' (highlighted in blue) and 'Disabled'. A red box surrounds this section. Below it is a checkbox for 'Move expired messages to the dead-letter subqueue'. At the top right, there are 'Save changes' and 'Discard changes' buttons.

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.

#### NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install 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-AzServiceBusQueue -ResourceGroup mygrp -NamespaceName myns -QueueName myqueue  
$q.Status = "Disabled"  
  
Set-AzServiceBusQueue -ResourceGroup mygrp -NamespaceName myns -QueueName myqueue -QueueObj $q
```

## Next steps

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

- [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

1/17/2020 • 5 minutes 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**.

## NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

## 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": "https://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/Namespaces",
      "location": "[resourceGroup().location]",
      "sku": {
        "name": "Standard"
      },
      "properties": {}
    },
    {
      "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')).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": "https://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-AzAccount
```

You are prompted to log on to your Azure account. After logging on, run the following command to view your available subscriptions:

```
Get-AzSubscription
```

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-AzContext -SubscriptionID <YourSubscriptionId>
```

### Set the resource group

If you do not have an existing resource group, create a new resource group with the **New-AzResourceGroup** command. Provide the name of the resource group and location you want to use. For example:

```
New-AzResourceGroup -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-AzResourceGroupDeployment** cmdlet. When testing the deployment, provide parameters exactly as you would when executing the deployment.

```
Test-AzResourceGroupDeployment -ResourceGroupName MyDemoRG -TemplateFile <path to template file>\azuredeploy.json
```

## Create the deployment

To create the new deployment, run the **New-AzResourceGroupDeployment** 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-AzResourceGroupDeployment -Name MyDemoDeployment -ResourceGroupName MyDemoRG -TemplateFile <path to template file>\azuredeploy.json
```

To specify a parameters file instead, use the following command:

```
New-AzResourceGroupDeployment -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-AzResourceGroupDeployment -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-AzResourceGroupDeployment -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](#)
- [Microsoft.ServiceBus resource types](#)

# Create a Service Bus namespace by using an Azure Resource Manager template

1/17/2020 • 3 minutes to read • [Edit Online](#)

Learn how to deploy an Azure Resource Manager template to create a Service Bus namespace. You can use this template for your own deployments, or customize it to meet your requirements. For more information about creating templates, see [Azure Resource Manager documentation](#).

The following templates are also available for creating Service Bus namespaces:

- [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](#)

## NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

If you don't have an Azure subscription, [create a free account](#) before you begin.

## Create a service bus namespace

In this quickstart, you use an [existing Resource Manager template](#) from [Azure Quickstart Templates](#):

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "serviceBusNamespaceName": {
            "type": "string",
            "metadata": {
                "description": "Name of the Service Bus namespace"
            }
        },
        "serviceBusSku": {
            "type": "string",
            "allowedValues": [
                "Basic",
                "Standard",
                "Premium"
            ],
            "defaultValue": "Standard",
            "metadata": {
                "description": "The messaging tier for service Bus namespace"
            }
        },
        "location": {
            "type": "string",
            "defaultValue": "[resourceGroup().location]",
            "metadata": {
                "description": "Location for all resources."
            }
        }
    },
    "variables": {
        "defaultSASKeyName": "RootManageSharedAccessKey",
        "defaultAuthRuleResourceId": "[resourceId('Microsoft.ServiceBus/namespaces/authorizationRules', parameters('serviceBusNamespaceName'), variables('defaultSASKeyName'))]",
        "sbVersion": "2017-04-01"
    },
    "resources": [
        {
            "apiVersion": "2017-04-01",
            "name": "[parameters('serviceBusNamespaceName')]",
            "type": "Microsoft.ServiceBus/namespaces",
            "location": "[parameters('location')]",
            "sku": {
                "name": "[parameters('serviceBusSku')]"
            }
        }
    ],
    "outputs": {
        "NamespaceDefaultConnectionString": {
            "type": "string",
            "value": "[listkeys(variables('defaultAuthRuleResourceId'), variables('sbVersion')).primaryConnectionString]"
        },
        "DefaultSharedAccessPolicyPrimaryKey": {
            "type": "string",
            "value": "[listkeys(variables('defaultAuthRuleResourceId'), variables('sbVersion')).primaryKey]"
        }
    }
}
```

To find more template samples, see [Azure Quickstart Templates](#).

To create a service bus namespace by deploying a template:

1. Select **Try it** from the following code block, and then follow the instructions to sign in to the Azure Cloud shell.

```

$serviceBusNamespaceName = Read-Host -Prompt "Enter a name for the service bus namespace to be created"
.setLocation = Read-Host -Prompt "Enter the location (i.e. centralus)"
$resourceGroupName = "${serviceBusNamespaceName}rg"
$templateUri = "https://raw.githubusercontent.com/Azure/azure-quickstart-templates/master/101-
servicebus-create-namespace/azuredeploy.json"

New-AzResourceGroup -Name $resourceGroupName -Location $location
New-AzResourceGroupDeployment -ResourceGroupName $resourceGroupName -TemplateUri $templateUri -
serviceBusNamespaceName $serviceBusNamespaceName

Write-Host "Press [ENTER] to continue ..."

```

The resource group name is the service bus namespace name with **rg** appended.

2. Select **Copy** to copy the PowerShell script.
3. Right-click the shell console, and then select **Paste**.

It takes a few moments to create an event hub.

## Verify the deployment

To see the deployed service bus namespace, you can either open the resource group from the Azure portal, or use the following Azure PowerShell script. If the Cloud shell is still open, you don't need to copy/run the first and second lines of the following script.

```

$serviceBusNamespaceName = Read-Host -Prompt "Enter the same service bus namespace name used earlier"
$resourceGroupName = "${serviceBusNamespaceName}rg"

Get-AzServiceBusNamespace -ResourceGroupName $resourceGroupName -Name $serviceBusNamespaceName

Write-Host "Press [ENTER] to continue ..."

```

Azure PowerShell is used to deploy the template in this tutorial. For other template deployment methods, see:

- [By using the Azure portal](#).
- [By using Azure CLI](#).
- [By using REST API](#).

## Clean up resources

When the Azure resources are no longer needed, clean up the resources you deployed by deleting the resource group. If the Cloud shell is still open, you don't need to copy/run the first and second lines of the following script.

```

$serviceBusNamespaceName = Read-Host -Prompt "Enter the same service bus namespace name used earlier"
$resourceGroupName = "${serviceBusNamespaceName}rg"

Remove-AzResourceGroup -ResourceGroupName $resourceGroupName

Write-Host "Press [ENTER] to continue ..."

```

## Next steps

In this article, you created a Service Bus namespace. See the other quickstarts to learn how to create queues, topics/subscriptions, and use them:

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

- Get started with Service Bus topics

# Create a Service Bus authorization rule for namespace and queue using an Azure Resource Manager template

12/23/2019 • 3 minutes 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**.

## NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

## 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"]
                        }
                    }
                ]
            }
        ],
        "dependsOn": [
            "[concat('Microsoft.ServiceBus/namespaces/',
parameters('serviceBusNamespaceName'))]"
        ],
        "location": "[resourceGroup().location]",
        "properties": {
            "Rights": ["Send"]
        }
    }
]

```

For JSON syntax and properties, see [namespaces](#), [queues](#), and [AuthorizationRules](#).

## Commands to run deployment

To deploy the resources to Azure, you must be signed 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:

- [Manage Azure resources by using Azure PowerShell](#)
- [Manage Azure resources by using Azure CLI](#).

The following examples assume you already have a resource group in your account with the specified name.

### PowerShell

```
New-AzResourceGroupDeployment -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

12/23/2019 • 3 minutes 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 do 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. It 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, 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. 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": {
                                    "filterType": "SqlFilter",
                                    "sqlFilter": {
                                        "sqlExpression": "StoreName = 'Store1'",
                                        "requiresPreprocessing": "false"
                                    },
                                    "action": {
                                        "sqlExpression": "set FilterTag = 'true'"
                                    }
                                }
                            }
                        ]
                    }
                ]
            }
        ]
    }
]
}

```

For JSON syntax and properties, see [namespaces](#), [topics](#), [subscriptions](#), and [rules](#).

## Commands to run deployment

To deploy the resources to Azure, you must be signed 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:

- [Manage Azure resources by using Azure PowerShell](#)
- [Manage Azure resources by using Azure CLI](#).

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

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](#)

# Configure customer-managed keys for encrypting Azure Service Bus data at rest by using the Azure portal

1/11/2020 • 4 minutes to read • [Edit Online](#)

Azure Service Bus Premium provides encryption of data at rest with Azure Storage Service Encryption (Azure SSE). Service Bus Premium relies on Azure Storage to store the data and by default, all the data that is stored with Azure Storage is encrypted using Microsoft-managed keys.

## Overview

Azure Service Bus now supports the option of encrypting data at rest with either Microsoft-managed keys or customer-managed keys (Bring Your Own Key - BYOK). This feature enables you to create, rotate, disable, and revoke access to the customer-managed keys that are used for encrypting Azure Service Bus at rest.

Enabling the BYOK feature is a one time setup process on your namespace.

### NOTE

There are some caveats to the customer managed key for service side encryption.

- This feature is supported by [Azure Service Bus Premium](#) tier. It cannot be enabled for standard tier Service Bus namespaces.
- The encryption can only be enabled for new or empty namespaces. If the namespace contains data, then the encryption operation will fail.

You can use Azure Key Vault to manage your keys and audit your key usage. You can either create your own keys and store them in a key vault, or you can use the Azure Key Vault APIs to generate keys. For more information about Azure Key Vault, see [What is Azure Key Vault?](#)

This article shows how to configure a key vault with customer-managed keys by using the Azure portal. To learn how to create a key vault using the Azure portal, see [Quickstart: Set and retrieve a secret from Azure Key Vault using the Azure portal](#).

### IMPORTANT

Using customer-managed keys with Azure Service Bus requires that the key vault have two required properties configured. They are: **Soft Delete** and **Do Not Purge**. These properties are enabled by default when you create a new key vault in the Azure portal. However, if you need to enable these properties on an existing key vault, you must use either PowerShell or Azure CLI.

## Enable customer-managed keys

To enable customer-managed keys in the Azure portal, follow these steps:

1. Navigate to your Service Bus Premium namespace.
2. On the **Settings** page of your Service Bus namespace, select **Encryption**.
3. Select the **Customer-managed key encryption at rest** as shown in the following image.

The screenshot shows the Azure portal interface for managing a Service Bus namespace named 'contoso-service-bus'. The left sidebar lists various management options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Events, Settings (Shared access policies, Geo-Recovery, Firewalls and virtual networks, Encryption, Properties, Locks, Export template), Entities (Queues, Topics), and Metrics. The 'Encryption' option under Settings is highlighted with a red box. The main content area displays a warning about enabling customer-managed key encryption at rest, information about Azure Service Bus encryption, and a note about enabling encryption on empty namespaces. It also includes a table for managing encryption keys and an 'Add Key' button.

## Set up a key vault with keys

After you enable customer-managed keys, you need to associate the customer managed key with your Azure Service Bus namespace. Service Bus supports only Azure Key Vault. If you enable the **Encryption with customer-managed key** option in the previous section, you need to have the key imported into Azure Key Vault. Also, the keys must have **Soft Delete** and **Do Not Purge** configured for the key. These settings can be configured using [PowerShell](#) or [CLI](#).

1. To create a new key vault, follow the Azure Key Vault [Quickstart](#). For more information about importing existing keys, see [About keys, secrets, and certificates](#).
2. To turn on both soft delete and purge protection when creating a vault, use the [az keyvault create](#) command.

```
az keyvault create --name contoso-SB-BYOK-keyvault --resource-group ContosoRG --location westus --enable-soft-delete true --enable-purge-protection true
```

3. To add purge protection to an existing vault (that already has soft delete enabled), use the [az keyvault update](#) command.

```
az keyvault update --name contoso-SB-BYOK-keyvault --resource-group ContosoRG --enable-purge-protection true
```

4. Create keys by following these steps:

- a. To create a new key, select **Generate/Import** from the **Keys** menu under **Settings**.

The screenshot shows the 'contoso-SB-BYOK-keyvault - Keys' page in the Azure portal. The left sidebar has a 'Keys' section highlighted with a red box. At the top, there's a search bar and a 'Generate/Import' button also highlighted with a red box. A success message at the top right says 'The key 'test-key1' has been successfully created.' The main table lists one key:

Name	Status	Expiration Date
test-key1	✓ Enabled	

b. Set **Options** to **Generate** and give the key a name.

### Create a key

The screenshot shows the 'Create a key' form. The 'Options' section is highlighted with a red box, showing 'Generate' selected in a dropdown. The 'Name' field is also highlighted with a red box, containing 'test-key 1'. Other visible fields include 'Key Type' (RSA selected), 'RSA Key Size' (2048 selected), and 'Enabled?' (Yes selected). A 'Create' button is at the bottom.

Options  
Generate

Name \* ⓘ  
test-key 1

Key Type ⓘ  
RSA EC

RSA Key Size  
2048 3072 4096

Set activation date? ⓘ

Set expiration date? ⓘ

Enabled?  Yes  No

**Create**

c. You can now select this key to associate with the Service Bus namespace for encrypting from the drop-down list.

## Select key from Azure Key Vault

Subscription *	Azure Messaging PM Playground
Key vault *	contoso-SB-BYOK-keyvault
	<a href="#">Create new</a>
Key *	<input type="text" value="Select the key."/> test-key1

[Select](#)

### NOTE

For redundancy, you can add up to 3 keys. In the event that one of the keys has expired, or is not accessible, the other keys will be used for encryption.

- d. Fill in the details for the key and click **Select**. This will enable the encryption of data at rest on the namespace with a customer managed key.

### IMPORTANT

If you are looking to use Customer managed key along with Geo disaster recovery, please review the below -

To enable encryption at rest with customer managed key, an [access policy](#) is set up for the Service Bus' managed identity on the specified Azure KeyVault. This ensures controlled access to the Azure KeyVault from the Azure Service Bus namespace.

Due to this:

- If [Geo disaster recovery](#) is already enabled for the Service Bus namespace and you are looking to enable customer managed key, then
  - Break the pairing
  - [Set up the access policy](#) for the managed identity for both the primary and secondary namespaces to the key vault.
  - Set up encryption on the primary namespace.
  - Re-pair the primary and secondary namespaces.
- If you are looking to enable Geo-DR on a Service Bus namespace where customer managed key is already set up, then -
  - [Set up the access policy](#) for the managed identity for the secondary namespace to the key vault.
  - Pair the primary and secondary namespaces.

## Rotate your encryption keys

You can rotate your key in the key vault by using the Azure Key Vaults rotation mechanism. For more information, see [Set up key rotation and auditing](#). Activation and expiration dates can also be set to automate key rotation. The Service Bus service will detect new key versions and start using them automatically.

## Revoke access to keys

Revoking access to the encryption keys won't purge the data from Service Bus. However, the data can't be accessed from the Service Bus namespace. You can revoke the encryption key through access policy or by deleting the key. Learn more about access policies and securing your key vault from [Secure access to a key vault](#).

Once the encryption key is revoked, the Service Bus service on the encrypted namespace will become inoperable. If the access to the key is enabled or the deleted key is restored, Service Bus service will pick the key so you can access the data from the encrypted Service Bus namespace.

## Next steps

See the following articles:

- [Service Bus overview](#)
- [Key Vault overview](#)

# Troubleshooting guide for Azure Service Bus

1/24/2020 • 9 minutes to read • [Edit Online](#)

This article provides some of the .NET exceptions generated by Service Bus .NET Framework APIs and also other tips for troubleshooting issues.

## Service Bus messaging exceptions

This section lists the .NET exceptions generated by .NET Framework APIs.

### 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. 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. The `RetryPolicy` class in the client SDK can be configured to handle retries automatically. For more information, see [Retry guidance](#).
4. Other exceptions ([System.Transactions.TransactionException](#), [System.TimeoutException](#), [Microsoft.ServiceBus.Messaging.MessageLockLostException](#), [Microsoft.ServiceBus.Messaging.SessionLockLostException](#)). General action: specific to the exception type; 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 didn't 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. It can happen because of 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/Example S	Suggested Action	Note on Automatic/Immediate Retry
<a href="#">InvalidOperationException</a>	<p>The requested user operation isn't 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.</p>	<p>Check the code and the documentation. Make sure the requested operation is valid.</p>	<p>Retry doesn't help.</p>
<a href="#">OperationCanceledException</a>	<p>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.</p>	<p>Check the code and make sure it doesn't invoke operations on a disposed object.</p>	<p>Retry doesn't help.</p>
<a href="#">UnauthorizedAccessException</a>	<p>The <a href="#">TokenProvider</a> object couldn't acquire a token, the token is invalid, or the token doesn't contain the claims required to do the operation.</p>	<p>Make sure the token provider is created with the correct values. Check the configuration of the Access Control Service.</p>	<p>Retry might help in some cases; add retry logic to code.</p>
<a href="#">ArgumentException</a> <a href="#">ArgumentNullException</a> <a href="#">ArgumentOutOfRangeException</a>	<p>One or more arguments supplied to the method are invalid.            The URI supplied to <a href="#">NamespaceManager</a> or <a href="#">Create</a> contains path segment(s).            The URI scheme supplied to <a href="#">NamespaceManager</a> or <a href="#">Create</a> is invalid.            The property value is larger than 32 KB.</p>	<p>Check the calling code and make sure the arguments are correct.</p>	<p>Retry doesn't help.</p>
<a href="#">MessagingEntityNotFoundException</a>	<p>Entity associated with the operation doesn't exist or it has been deleted.</p>	<p>Make sure the entity exists.</p>	<p>Retry doesn't help.</p>
<a href="#">MessageNotFoundException</a>	<p>Attempt to receive a message with a particular sequence number. This message isn't found.</p>	<p>Make sure the message hasn't been received already. Check the deadletter queue to see if the message has been deadlettered.</p>	<p>Retry doesn't help.</p>
<a href="#">MessagingCommunicationException</a>	<p>Client isn't able to establish a connection to Service Bus.</p>	<p>Make sure the supplied host name is correct and the host is reachable.</p>	<p>Retry might help if there are intermittent connectivity issues.</p>
<a href="#">ServerBusyException</a>	<p>Service isn't able to process the request at this time.</p>	<p>Client can wait for a period of time, then retry the operation.</p>	<p>Client may retry after certain interval. If a retry results in a different exception, check retry behavior of that exception.</p>

Exception Type	Description/Cause/Example S	Suggested Action	Note on Automatic/Immediate Retry
<a href="#">MessageLockLostException</a>	Lock token associated with the message has expired, or the lock token isn't found.	Dispose the message.	Retry doesn't help.
<a href="#">SessionLockLostException</a>	Lock associated with this session is lost.	Abort the <a href="#">MessageSession</a> object.	Retry doesn't help.
<a href="#">MessagingException</a>	<p>Generic messaging exception that may be thrown in the following cases:</p> <ul style="list-style-type: none"> <li>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).</li> <li>An attempt is made to send a message larger than 256 KB. The server or service encountered an error during processing of the request. See the exception message for details. It's usually a transient exception.</li> </ul>	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's <b>true</b> , you can retry the operation.	Retry behavior is undefined and might not help.
<a href="#">MessagingEntityAlreadyExist sException</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 doesn't 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 subqueues. 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 doesn't 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 doesn't 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 doesn't 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 doesn't help.
MessageSizeExceededException	A message payload exceeds the 256-KB limit. 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 doesn't 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 doesn't 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, it's 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. It 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 aren't 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.

## Connectivity, certificate, or timeout issues

The following steps may help you with troubleshooting connectivity/certificate/timeout issues for all services under \*.servicebus.windows.net.

- Browse to or [wget https://<yournamespace>.servicebus.windows.net/](#). It helps with checking whether you have IP filtering or virtual network or certificate chain issues (most common when using java SDK).

An example of successful message:

```
<feed xmlns="http://www.w3.org/2005/Atom"><title type="text">Publicly Listed Services</title><subtitle type="text">This is the list of publicly-listed services currently available.</subtitle><id>uuid:27fcde1e2-3a99-44b1-8f1e-3e92b52f0171;id=30</id><updated>2019-12-27T13:11:47Z</updated><generator>Service Bus 1.1</generator></feed>
```

An example of failure error message:

```
<Error>
  <Code>400</Code>
  <Detail>
    Bad Request. To know more visit https://aka.ms/sbResourceMgrExceptions. . TrackingId:b786d4d1-cbaf-47a8-a3d1-be689cda2a98_G22, SystemTracker:NoSystemTracker, Timestamp:2019-12-27T13:12:40
  </Detail>
</Error>
```

- Run the following command to check if any port is blocked on the firewall. Ports used are 443 (HTTPS), 5671 (AMQP) and 9354 (Net Messaging/SBMP). Depending on the library you use, other ports are also used. Here is the sample command that check whether the 5671 port is blocked.

```
tnc <yournamespacename>.servicebus.windows.net -port 5671
```

On Linux:

```
telnet <yournamespacename>.servicebus.windows.net 5671
```

- When there are intermittent connectivity issues, run the following command to check if there are any dropped packets. This command will try to establish 25 different TCP connections every 1 second with the service. Then, you can check how many of them succeeded/failed and also see TCP connection latency. You can download the `psping` tool from [here](#).

```
.\psping.exe -n 25 -i 1 -q <yournamespace>.servicebus.windows.net:5671 -nobanner
```

You can use equivalent commands if you're using other tools such as `tnc`, `ping`, and so on.

- Obtain a network trace if the previous steps don't help and analyze it using tools such as [Wireshark](#). Contact [Microsoft Support](#) if needed.

## 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 architecture](#)

# Service Bus Resource Manager exceptions

1/14/2020 • 5 minutes to read • [Edit Online](#)

This article lists exceptions generated when interacting with Azure Service Bus using Azure Resource Manager - via templates or direct calls.

## IMPORTANT

This document is frequently updated. Please check back for updates.

Below are the various exceptions/errors that are surfaced through the Azure Resource Manager.

## Error: Bad Request

"Bad Request" implies that the request received by the Resource Manager failed validation.

ERROR CODE	ERROR SUBCODE	ERROR MESSAGE	DESCRIPTION	RECOMMENDATION
Bad Request	40000	SubCode=40000. The property ' <i>property name</i> ' cannot be set when creating a Queue because the namespace ' <i>namespace name</i> ' is using the 'Basic' Tier. This operation is only supported in 'Standard' or 'Premium' tier.	On Azure Service Bus Basic Tier, the below properties cannot be set or updated - <ul style="list-style-type: none"><li>• RequiresDuplicateDetection</li><li>• AutoDeleteOnIdle</li><li>• RequiresSession</li><li>• DefaultMessageTimeToLive</li><li>• DuplicateDetectionHistoryTimeWindow</li><li>• EnableExpress</li><li>• ForwardTo</li><li>• Topics</li></ul>	Consider upgrading from Basic to Standard or Premium tier to use this functionality.
Bad Request	40000	SubCode=40000. The value for the 'requiresDuplicateDetection' property of an existing Queue(or Topic) cannot be changed.	Duplicate detection must be enabled/disabled at the time of entity creation. The duplicate detection configuration parameter cannot be changed after creation.	To enable duplicate detection on a previously created queue/topic, you can create a new queue/topic with duplicate detection and then forward from the original queue to the new queue/topic.

Error Code	Error Subcode	Error Message	Description	Recommendation
Bad Request	40000	SubCode=40000. The specified value 16384 is invalid. The property 'MaxSizeInMegabytes' , must be one of the following values: 1024;2048;3072;4096;5120.	The MaxSizeInMegabytes value is invalid.	Ensure that the MaxSizeInMegabytes is one of the following - 1024, 2048, 3072, 4096, 5120.
Bad Request	40000	SubCode=40000. Partitioning cannot be changed for entity Queue/Topic.	Partitioning cannot be changed for entity.	Create a new entity (queue or topic) and enable partitions.
Bad Request	none	The namespace 'namespace name' does not exist.	The namespace does not exist within your Azure subscription.	To resolve this error, please try the below <ul style="list-style-type: none"> <li>• Ensure that the Azure Subscription is correct.</li> <li>• Ensure the namespace exists.</li> <li>• Verify the namespace name is correct (no spelling errors or null strings).</li> </ul>
Bad Request	40400	SubCode=40400. The auto forwarding destination entity does not exist.	The destination for the autoforwarding destination entity doesn't exist.	The destination entity (queue or topic), must exist before the source is created. Retry after creating the destination entity.
Bad Request	40000	SubCode=40000. The supplied lock time exceeds the allowed maximum of '5' minutes.	The time for which a message can be locked must be between 1 minute (minimum) and 5 minutes (maximum).	Ensure that the supplied lock time is between 1 min and 5 mins.
Bad Request	40000	SubCode=40000. Both DelayedPersistence and RequiresDuplicateDetection property cannot be enabled together.	Entities with Duplicate detection enabled on them must be persistent, so persistence cannot be delayed.	Learn more about <a href="#">Duplicate Detection</a>

Error Code	Error Subcode	Error Message	Description	Recommendation
Bad Request	40000	SubCode=40000. The value for RequiresSession property of an existing Queue cannot be changed.	Support for sessions should be enabled at the time of entity creation. Once created, you cannot enable/disable sessions on an existing entity (queue or subscription)	Delete and recreate a new queue (or subscription) with the "RequiresSession" property enabled.
Bad Request	40000	SubCode=40000. 'URI_PATH' contains character(s) that is not allowed by Service Bus. Entity segments can contain only letters, numbers, periods(.), hyphens(-), and underscores(_). Any other characters cause the request to fail.	Entity segments can contain only letters, numbers, periods(.), hyphens(-), and underscores(_). Any other characters cause the request to fail.	Ensure that there are no invalid characters in the URI Path.

## Error code: 429

Just like in HTTP, "Error code 429" indicates "too many requests". It implies that the specific resource (namespace) is being throttled because of too many requests (or due to conflicting operations) on that resource.

Error Code	Error Subcode	Error Message	Description	Recommendation
429	50004	SubCode=50004. The request was terminated because the namespace <i>your namespace</i> is being throttled.	This error condition is hit when the number of incoming requests exceed the limitation of the resource.	Wait for a few seconds and try again.  Learn more about the <a href="#">quotas</a> and <a href="#">Azure Resource Manager request limits</a>
429	40901	SubCode=40901. Another conflicting operation is in progress.	Another conflicting operation is in progress on the same resource/entity	Wait for the current in-progress operation to complete before trying again.
429	40900	SubCode=40900. Conflict. You're requesting an operation that isn't allowed in the resource's current state.	This condition may be hit when multiple requests are made to perform the operations on the same entity (queue, topic, subscription, or rule) at the same time.	Wait for a few seconds and try again
429	40901	Request on entity ' <i>entity name</i> ' conflicted with another request	Another conflicting operation is in progress on the same resource/entity	Wait for the previous operation to complete before trying again

Error Code	Error Subcode	Error Message	Description	Recommendation
429	40901	Another update request is in progress for the entity ' <i>entity name</i> '.	Another conflicting operation is in progress on the same resource/entity	Wait for the previous operation to complete before trying again
429	none	Resource Conflict Occurred. Another conflicting operation may be in progress. If this is a retry for failed operation, background cleanup is still pending. Try again later.	This condition may be hit when there is a pending operation against the same entity.	Wait for the previous operation to complete before trying again.

## Error code: Not Found

This class of errors indicates that the resource was not found.

Error Code	Error Subcode	Error Message	Description	Recommendation
Not found	none	Entity ' <i>entity name</i> ' was not found.	The entity against which the operation was not found.	Check if the entity exists and try the operation again.
Not found	none	Not Found. The Operation doesn't exist.	The operation you are trying to perform does not exist.	Check the operation and try again.
Not found	none	The incoming request is not recognized as a namespace policy put request.	The incoming request body is null and hence cannot be executed as a put request.	Please check the request body to ensure that it is not null.
Not found	none	The messaging entity ' <i>entity name</i> ' could not be found.	The entity that you are trying to execute the operation against could not be found.	Please check whether the entity exists and try the operation again.

## Error code: Internal Server Error

This class of errors indicates that there was a internal server error

Error Code	Error Subcode	Error Message	Description	Recommendation

Error Code	Error Subcode	Error Message	Description	Recommendation
Internal Server Error	50000	SubCode=50000. Internal Server Error	Can happen for various reasons. Some of the symptoms are - <ul style="list-style-type: none"><li>● Client request/body is corrupt and leads to an error.</li><li>● The client request timed out due to processing issues on the service.</li></ul>	To resolve this <ul style="list-style-type: none"><li>● Ensure that the requests parameters are not null or malformed.</li><li>● Retry the request.</li></ul>

## Error code: Unauthorized

This class of errors indicates the absence of authorization to run the command.

Error Code	Error Subcode	Error Message	Description	Recommendation
Unauthorized	none	Invalid operation on the Secondary namespace. Secondary namespace is read-only.	The operation was performed against the secondary namespace, which is setup as a readonly namespace.	Retry the command against the primary namespace. Learn more about <a href="#">secondary namespace</a>
Unauthorized	none	MissingToken: The authorization header was not found.	This error occurs when the authorization has null or incorrect values.	Ensure that the token value mentioned in the authorization header is correct and not null.

# Service Bus quotas

1/24/2020 • 4 minutes 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 Azure Service Bus messaging. For information about pricing and other quotas for Service Bus, see [Service Bus pricing](#).

Quota name	Scope	Notes	Value
Maximum number of Basic or Standard namespaces per Azure subscription	Namespace	Subsequent requests for additional Basic or Standard namespaces are rejected by the Azure portal.	100
Maximum number of Premium namespaces per Azure subscription	Namespace	Subsequent requests for additional Premium namespaces are rejected by the portal.	100
Queue or topic size	Entity	Defined upon creation of the queue or topic.  Subsequent incoming messages are rejected, and an exception is received by the calling code.	1, 2, 3, 4 GB or 5 GB.  In the Premium SKU, and the Standard SKU with <a href="#">partitioning</a> enabled, the maximum queue or 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 don't count toward concurrent TCP connections.	NetMessaging: 1,000.  AMQP: 5,000.
Number of concurrent receive requests on a queue, topic, or 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 or queues per namespace	Namespace	Subsequent requests for creation of a new topic or queue on the 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 for the Basic or Standard tier. The total number of topics and queues in a namespace must be less than or equal to 10,000.  For the Premium tier, 1,000 per messaging unit (MU). Maximum limit is 4,000.
Number of <a href="#">partitioned topics or queues</a> per namespace	Namespace	Subsequent requests for creation of a new partitioned topic or queue on the 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, the exception <b>QuotaExceededException</b> is received by the calling code.	Basic and Standard tiers: 100.  Partitioned entities aren't supported in the Premium tier.  Each partitioned queue or topic counts toward the quota of 1,000 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.
Maximum size of a <a href="#">message ID</a>	Entity	-	128
Maximum size of a message <a href="#">session ID</a>	Entity	-	128

Quota Name	Scope	Notes	Value
Message size for a queue, topic, or subscription entity	Entity	<p>Incoming messages that exceed these quotas are rejected, and an exception is received by the calling code.</p> <p>Due to system overhead, this limit is less than these values.</p> <p>Maximum header size: 64 KB.</p> <p>Maximum number of header properties in property bag: <b>byte/int.MaxValue</b>.</p> <p>Maximum size of property in property bag: No explicit limit. Limited by maximum header size.</p>	<p>Maximum message size: 256 KB for <a href="#">Standard tier</a>, 1 MB for <a href="#">Premium tier</a>.</p>
Message property size for a queue, topic, or subscription entity	Entity	The exception <b>SerializationException</b> is generated.	<p>Maximum message property size for each property is 32,000.</p> <p>Cumulative size of all properties can't exceed 64,000. This limit applies to the entire header of the <a href="#">BrokeredMessage</a>, which has both user properties and system properties, such as <a href="#">SequenceNumber</a>, <a href="#">Label</a>, and <a href="#">MessageId</a>.</p>
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 per-topic for the Standard tier.
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

Quota Name	Scope	Notes	Value
Size of SQL filters or actions	Namespace	Subsequent requests for creation of additional filters are rejected, and an exception is received by the calling code.	<p>Maximum length of filter condition string: 1,024 (1 K).</p> <p>Maximum length of rule action string: 1,024 (1 K).</p> <p>Maximum number of expressions per rule action: 32.</p>
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.	<p>Maximum number of rules: 12.</p> <p>Rules that are configured on a Service Bus namespace apply to all queues and topics in that namespace.</p>
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.	<p>100</p> <p>For both <b>Send()</b> and <b>SendAsync()</b> operations.</p>
Number of virtual network and IP filter rules	Namespace		128

# SQLFilter syntax

1/24/2020 • 5 minutes 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]
[HHR-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 reference for Azure Service Bus

1/24/2020 • 4 minutes 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](#)

# Azure Service Bus - Frequently asked questions (FAQ)

1/24/2020 • 7 minutes 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.

## NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

## 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 your own hardware 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. Supported only in the Basic and Standard messaging tiers, a [partitioned queue or topic](#) is handled by multiple message brokers and stored in multiple messaging stores. This feature 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.

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.

Partitioned entities are no longer supported in the [Premium SKU](#).

### What ports do I need to open on the firewall?

You can use the following protocols with Azure Service Bus to send and receive messages:

- Advanced Message Queuing Protocol (AMQP)

- Service Bus Messaging Protocol (SBMP)
- HTTP

See the following table for the outbound ports you need to open to use these protocols to communicate with Azure Event Hubs.

PROTOCOL	PORTS	DETAILS
AMQP	5671 and 5672	See <a href="#">AMQP protocol guide</a>
SBMP	9350 to 9354	See <a href="#">Connectivity mode</a>
HTTP, HTTPS	80, 443	

### What IP addresses do I need to whitelist?

To find the right IP addresses to white list for your connections, follow these steps:

1. Run the following command from a command prompt:

```
nslookup <YourNamespaceName>.servicebus.windows.net
```

2. Note down the IP address returned in **Non-authoritative answer**. This IP address is static. The only point in time it would change is if you restore the namespace on to a different cluster.

If you use the zone redundancy for your namespace, you need to do a few additional steps:

1. First, you run nslookup on the namespace.

```
nslookup <yournamespace>.servicebus.windows.net
```

2. Note down the name in the **non-authoritative answer** section, which is in one of the following formats:

```
<name>-s1.servicebus.windows.net
<name>-s2.servicebus.windows.net
<name>-s3.servicebus.windows.net
```

3. Run nslookup for each one with suffixes s1, s2, and s3 to get the IP addresses of all three instances running in three availability zones,

## 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.

## I have a Service Bus Standard namespace. Why do I see charges under resource group '\$system'?

Azure Service Bus recently upgraded the billing components. Due to this, if you have a Service Bus Standard namespace, you may see line items for the resource

'/subscriptions/<azure\_subscription\_id>/resourceGroups/\$system/providers/Microsoft.ServiceBus/namespaces/\$system' under resource group '\$system'.

These charges represent the base charge per Azure subscription that has provisioned a Service Bus Standard namespace.

It is important to note that these are not new charges, i.e. they existed in the previous billing model too. The only change is that they are now listed under '\$system'. This is done due to constraints in the new billing system which groups subscription level charges, not tied to a specific resource, under the '\$system' resource id.

# 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. If you need more than these limits, you can contact customer service at any time to understand your needs and adjust these limits appropriately. For Service Bus, the aggregate usage quota is 5 billion messages per month.

While Microsoft reserves the right to disable a customer account that has exceeded its usage quotas in a given month, e-mail notifications are sent and multiple attempts are made 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](#).

## How to handle messages of size > 1 MB?

Service Bus messaging services (queues and topics/subscriptions) allow application to send messages of size up to 256 KB (standard tier) or 1 MB (premium tier). If you are dealing with messages of size greater than 1 MB, use the claim check pattern described in [this blog post](#).

# Troubleshooting

## Why am I not able to create a namespace after deleting it from another subscription?

When you delete a namespace from a subscription, wait for 4 hours before recreating it with the same name in another subscription. Otherwise, you may receive the following error message: `Namespace already exists`.

## 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.js, PHP, Java, Python, 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-AzSubscription -SubscriptionId 'ffffffff-ffff-ffff-ffff-ffffffffffff'
New-AzResourceGroup -Name 'targetRG' -Location 'East US'

# Move namespace from source subscription to target subscription
Select-AzSubscription -SubscriptionId 'aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaa'
$res = Find-AzResource -ResourceNameContains mynamespace -ResourceType 'Microsoft.ServiceBus/namespaces'
Move-AzResource -DestinationResourceGroupName 'targetRG' -DestinationSubscriptionId 'ffffffff-ffff-ffff-ffff-ffffffffffff' -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](#)
- [Get started with Service Bus queues](#)