**What is service bus?**

Microsoft Azure Service Bus is a fully managed enterprise [integration](https://azure.com/integration) message broker. Service Bus is most commonly used to decouple applications and services from each other, and is a reliable and secure platform for asynchronous data and state transfer. Data is transferred between different applications and services using messages. A message is in binary format, which can contain JSON, XML, or just text.

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 do not 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.

Launched as a CTP in 2007 under the name of BizTalk Services, the Service Bus has evolved and matured into an enterprise-class messaging platform. The Service Bus provides synchronous and asynchronous communication

The Microsoft Azure Service Bus provides a number of messaging focused services that can be leveraged by on-premise and cloud-based applications to handle synchronous and asynchronous messaging.

The Service Bus currently consists of the following services.

Relayed Messaging

The Service Bus relay service allows services to expose their endpoints “in the cloud” using a public URL within the Azure service bus namespace. Clients make calls to this public endpoint, and the calls are relayed to the service, which can be hosted on-premise, or hosted by any public cloud provider. Authentication, authorization and transport level encryption are provided by the service bus, and REST and Soap services are supported. The relay service is typically used for synchronous, request-response communication.

Brokered Messaging

Service Bus brokered messaging service provides asynchronous messaging capabilities. Point-to-point messaging is implemented using queues, and publish-subscribe messaging with topics and subscriptions. Brokered messaging has some similarities with queues in the Azure Storage service, but provides much more sophisticated functionality, making it a good choice for many scenarios.

Event Hubs

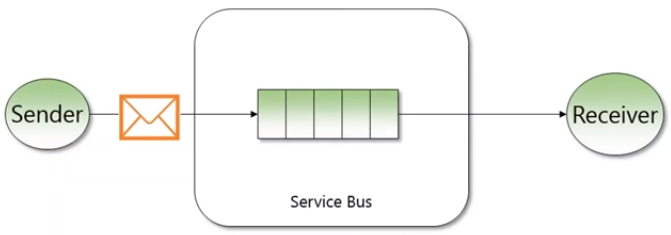
Service Bus event hubs provide messaging capabilities on a massive scale. Event hubs are built to handle the massive load generated by telemetry data from applications and devices. The emergence of the “Internet of Things” (IoT) and the on-line gaming industry using connected consoles and devices is generating a demand for systems that can handle hundreds of thousands and even millions of messages per second. Service Bus event hubs are able to provide this capacity as an on-demand service, allowing companies to leverage these capabilities and scale them when required.

Notification Hubs

Service Bus Notification Hubs provide application to device messaging capabilities for sending push notifications to mobile devices. Notification hubs can integrate with the messaging services provided by Microsoft, Apple and the Android operating systems to make the transmission of push notifications to multiple device platforms simple to implement.

Queue

A messaging protocol based on the first-in-first-out pattern. Here a sender would send a message to a queue from where the receiver would collect it in initial or later stage. Besides that, queues enable us to store the message until the receiver is available to receive and also process them. The user will receive the message upon request.



# Managing Queues

Queues, topics and subscriptions are managed programmatically by calling methods on the NamespaceManager class. As the techniques for managing queues are very similar to that of topics and subscriptions queue management will be covered in detail, then additional coverage for topics and subscriptions provided where appropriate.

**Creating Queues**

The CreateQueue method of the NamespaceManager is overloaded and provides two options for creating queues.

|  |
| --- |
| public QueueDescription CreateQueue(string path)    public QueueDescription CreateQueue(QueueDescription description) |

The first method will create a queue with the address of the NamespaceManager and path specified. The queue created will have the default property values. If the default property values are to be overridden, a QueueDescription can be created and the appropriate values set. This can then be used to create the queue using the second method.

Attempting to create queue with a path that already exists in the service bus namespace will result in a MessagingEntityAlreadyExistsException exception being thrown.

**QueueDescription Class**

The QueueDescription class is a data contract class that provides properties which can be set when creating a queue. Setting any of these properties will override their default values in the new queue.

**Useful QueueDescription Properties**

DefaultMessageTimeToLive

The DefaultMessageTimeToLive is a TimeSpan property that sets the default expiration timeout for messages that are added to the queue. The default value is TimeSpan.MaxValue, which is almost 30,000 years, meaning that messages will never expire during the expected lifetime of the application. If this value is set for a queue, any messages placed on that queue that have not been assigned a TimeToLive value will default to the value set for the queue. If the sending application has set the TimeToLive value for the message the value will not be changed.

DuplicateDetectionHistoryTimeWindow

If duplicate detection is enabled for a queue the DuplicateDetectionHistoryTimeWindow value will specify how long the MessageId values for the received messages will be retained in order to check for duplicate messages. The property is a TimeSpan with a default value of 10 minutes and a maximum allowed value of 7 days.

EnableDeadLetteringOnMessageExpiration

By default any messages that have been on the queue for longer than the assigned time to live will expire and be removed from the queue. Setting EnableDeadLetteringOnMessageExpiration will cause expired messages to be moved to the dead-letter queue instead of being removed from the queue. This will allow the messages to be processed at a later point in time.

Be aware that expired messages on the dead letter queue will increase the size of the queue, which could result in the queue failing to receive new messages if expired messages are not de-queued from the dead letter queue.

IsReadOnly

When an instance of the QueueDescription class is created using the default constructor the properties can be modified. When a QueueDescription for a queue that is present in a service bus namespace is returned the properties can’t be modified. The IsReadOnly property indicates weather the queue description properties can be modified.

LockDuration

LockDuration is a TimeSpan property that determines the maximum time a message will remain in the locked state when received using the peek-lock receive mode. The default value is 30 seconds; the maximum allowed value is 5 minutes. If the receiver fails to complete, abandon, defer or dead-letter the message within this time the message will become visible on the queue for other receivers to receive.

If it is expected that the receiving application may take longer than 30 seconds to determine if a message can be successfully processed or not the LockDuration of the queue should be set to an appropriate value. Be aware that it is no possible for the receiving application to specify the lock duration when dequeuing a message, and once a queue is created the lock duration cannot be changed. In production systems care should be taken to ensure a suitable lock duration has been set.

MaxDeliveryCount

The MaxDeliveryCount property specifies the maximum number of times a message can be received by a receiving application before the message is automatically place on the dead letter queue. This is used to prevent the repeated processing of “poison messages” causing bottle necks in a message channel.

The default setting is 10, the minimum allowed value is 1 and the maximum allowed value is int.MaxValue, (2,147,483,647). Setting the maximum value effectively turns off the dead lettering of poison messages.

MaxSizeInMegabytes

The MaxSizeInMegabytes property defines the maximum size a queue can reach before new messages are rejected from the queue. The default setting is 1,024, which is 1 GB, and the current maximum allowable value is 5120, which is 5 GB. The maximum size of a queue in GB is used as a multiplier when calculating the billing for entity hours in the service bus, a queue with a maximum size of 4 GB will be billed at 4 times the price of one with a maximum size of 1 GB.

MessageCount

The MessageCount property returns a count of the number of messages on the queue.

RequiresDuplicateDetection

When set to true, the RequiresDuplicateDetection property will ensure that all enqueued messages with a duplicate value of the MessageId property within the DuplicateDetectionHistoryTimeWindow will be ignored.

RequiresSession

The RequiresSession property specifies that messages sent to the queue most have a value set for their SessionId property. Sessions are used for correlating related messages into transactions.

SizeInBytes

The SizeInBytes returns the current size of the queue.

**Send Message**

private void SendMessage()

{

try

{

QueueClient queueClient = new QueueClient(serviceBusConnString, queueName);

string messageBody = string.Empty;

messageBody = "Message From application";

Console.WriteLine("Hello World!");

Message message;

for (int i = 0; i <= 5; i++)

{

message = new Message(Encoding.UTF8.GetBytes(messageBody + i.ToString()));

//In Session Enabled case add the session id to message

// message.SessionId = Guid.NewGuid().ToString();

queueClient.SendAsync(message);

}

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

finally

{

queueClient.CloseAsync();

Console.ReadKey();

}

}

**Read Message**

**private void ReadMessage()**

{

QueueClient queueClient = new QueueClient(serviceBusConnString, queueName);

MessageHandlerOptions messageHandlerOptions = new MessageHandlerOptions(ExceptionReceivedHandler)

{

MaxConcurrentCalls = 1,

AutoComplete = false

};

queueClient.RegisterMessageHandler(ReceiveMessagesAsync, messageHandlerOptions);

}

**private async Task ReceiveMessagesAsync(Message message, CancellationToken token)**

{

Console.WriteLine(Encoding.UTF8.GetString(message.Body));

await queueClient.CompleteAsync(message.SystemProperties.LockToken);

}

**private Task ExceptionReceivedHandler(ExceptionReceivedEventArgs exceptionReceivedEventArgs)**

{

Console.WriteLine(exceptionReceivedEventArgs.Exception);

return Task.CompletedTask;

}

//Session Enabled Section

private void ReadSessionEnabledMessage()

{

SessionHandlerOptions sessionHandlerOptions = new SessionHandlerOptions(ExceptionReceivedHandler)

{

MaxConcurrentSessions = 100,

AutoComplete = true

};

queueClient.RegisterSessionHandler(ProcessMessagesInSessionAsync, sessionHandlerOptions);

}

**private static async Task ProcessMessagesInSessionAsync(IMessageSession messageSession, Message message, CancellationToken token)**

{

Console.WriteLine($"Received message: SequenceNumber:{message.SystemProperties.SequenceNumber} Body:{Encoding.UTF8.GetString(message.Body)}");

await Task.CompletedTask;

//// We don't use the CompleteAsync() method.

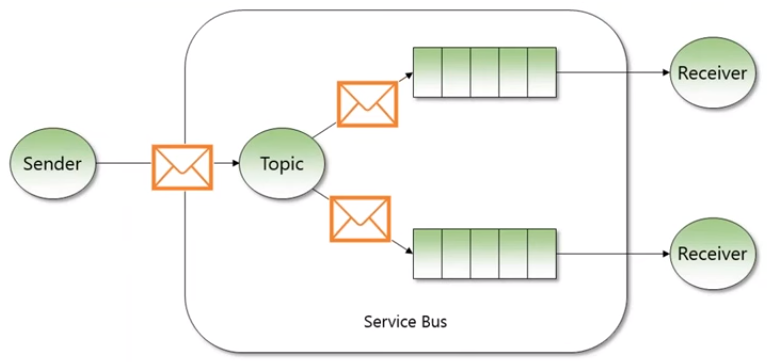
//// await subscriptionClient.CompleteAsync(message.SystemProperties.LockToken);

}

Topics

Topics are similar to queues. Most importantly, topics make multiple subscriptions available. A sender will send a message to a topic then a routing rule can determine which subscription may receive the slotted copy of the message.

The receiver can receive the message from the subscription. Multiple receivers could be allocated to a subscription.



Other messaging patterns include,

* RPC
* Scatter Gather
* Pub/Sub

**private async Task SendMessage()**

{

string messageBody = string.Empty;

Message message = new Message();

for (int i = 0; i < 5; i++)

{

messageBody = $"Message {i.ToString()}";

message.Body = Encoding.UTF8.GetBytes(messageBody);

//In Session Enabled case add the session id to message

// message.SessionId = Guid.NewGuid().ToString();

await topicClient.SendAsync(message);

}

await topicClient.CloseAsync();

}

**private void ReadMessage()**

{

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);

}

**private Task ExceptionReceivedHandler(ExceptionReceivedEventArgs exceptionReceivedEventArgs)**

{

Console.WriteLine(exceptionReceivedEventArgs.Exception.Message);

return Task.CompletedTask;

}

**private async Task ProcessMessagesAsync(Message message, CancellationToken token)**

{

Console.WriteLine(Encoding.UTF8.GetString(message.Body));

await subscriptionClient.CompleteAsync(message.SystemProperties.LockToken);

}

//Session Enabled Section

**private void ReadSessionEnabledMessage()**

{

SessionHandlerOptions sessionHandlerOptions = new SessionHandlerOptions(ExceptionReceivedHandler)

{

MaxConcurrentSessions = 100,

AutoComplete = true

//// We don't use the CompleteAsync() method.

//// await subscriptionClient.CompleteAsync(message.SystemProperties.LockToken);

};

subscriptionClient.RegisterSessionHandler(ProcessMessagesInSessionAsync, sessionHandlerOptions);

}

**private static async Task ProcessMessagesInSessionAsync(IMessageSession messageSession, Message message, CancellationToken token)**

{

Console.WriteLine($"Received message: SequenceNumber:{message.SystemProperties.SequenceNumber} Body:{Encoding.UTF8.GetString(message.Body)}");

await Task.CompletedTask;

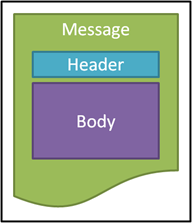
//// We don't use the CompleteAsync() method.

//// await subscriptionClient.CompleteAsync(message.SystemProperties.LockToken);

}

# Messages

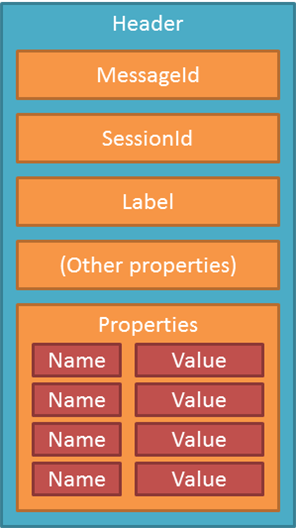
All messages used in the Service Bus brokered messaging services are instances of the BrokeredMessage class. Messages are made up of a header and a body. The message header contains properties and contextual information about the message. The message body contains the payload or data of the message.



**Message Header**

The message header contains a number of fixed properties that are used by the service bus messaging entities and by sending and receiving applications to handle the processing of messages. Some of these properties are used by the messaging entities, others are application specific.

The header also contains a string, object dictionary, named Properties, which can be used to store a number of name-value pairs that can be used for routing messages between topics and subscriptions and also by sending and receiving applications for application specific logic.



The maximum allowed size for a message header is 64 kb.

Be aware that “message properties” could possibly refer to the fixed properties of the message, such as MessageId, and Label, and also to the name-value pairs in the Properties dictionary. I will endeavor to use “message properties collection” when referring to the name-value properties collection.

**Message Body**

The message body contains the serialized data that is transmitted in the message. This is typically the serialized state of an object, or a data stream. In some cases the message body can be empty.

**Message Immutability**

BizTalk developers are familiar with the concept of immutable messages; the same goes in Service Bus brokered messaging. Once an instance of the BrokeredMessage class has been created it is not possible to set or modify the message body content.

**Message Size Limitations**

The Service Bus brokered messaging services have a limit on the size of message that can be placed on a queue or a topic. The current limit on the message size is 256 KB, which includes the message body and header. There is, however, no limit in the size of message that can be created using the BrokeredMessage class. The handling of large messages in the Service Bus brokered messaging services will be covered later in this chapter.

# Queue, Topic and Subscription Properties Summarized

The following table provides a summary of the properties available on queues, topics and subscriptions. In general enqueue related properties are available on queues and topics, and dequeue related properties on queues and subscriptions.

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Default** | **Max Value** | **Applicable to** |
| DefaultMessageTimeToLive | TimeSpan.MaxValue | TimeSpan.MaxValue | Queue, Topic, Subscription |
| DuplicateDetectionHistoryTimeWindow | 00:10:00 | 7 days | Queue, Topic |
| EnableBatchedOperations | True |  | Queue, Topic, Subscription |
| EnableDeadLetteringOnFilterEvaluationExceptions | False |  | Subscription |
| EnableDeadLetteringOnMessageExpiration | False |  | Queue, Subscription |
| ExtensionData |  |  | Queue, Topic, Subscription |
| IsReadOnly | False |  | Queue, Topic, Subscription |
| LockDuration | 00:01:00 | 00:05:00 | Queue, Subscription |
| MaxDeliveryCount | 10 | Int.MaxValue | Queue, Subscription |
| MaxSizeInMegabytes | 1024 | 5120 | Queue, Topic |
| MessageCount |  |  | Queue, Subscription |
| Name |  | 50 characters | Subscription |
| Path |  | 260 characters | Queue, Topic |
| RequiresDuplicateDetection | False |  | Queue, Topic |
| RequiresSession | False |  | Queue, Subscription |
| SizeInBytes |  |  | Queue, Topic |
| TopicPath |  | 260 characters | Subscription |

# Filters

Azure Service Bus Topics and Subscriptions offer you a powerful way to configure multiple subscribers for a single message. So you could post an OrderCreated message to a topic, and one subscriber might initiate payment processing, another might send a confirmation email, and another might write to an audit log. So using topics and subscriptions instead of queues offers you a lot of flexibility.

But sometimes in a topics and subscriptions setup you might find your subscriber is only interested in receiving a subset of the messages posted to that topic. One way to handle this is for your subscribers to simply ignore the messages they don’t care about, but that is wasteful since you pay for every receive operation.

A better approach is to create a filtered subscription which will only receive messages you are interested in. A filter can be based on any properties of the BrokeredMessage, except the body, since that would require every message to be deserialized in order to be handed on to the correct subscriptions.

There are three types of filters supported by the broker:

1. Boolean filters
2. SQL filters
3. Correlation filters

## Boolean filters

These filters (TrueFilter and FalseFilter) are not the most sophisticated. They are literally "catch-all" or "catch nothing" options. The TrueFilter is the default when nothing else is defined. It’s handy when implementing a wiretap to analyze all messages flowing through a topic.

## SQL filters

Just as the name indicates, SQL filters allow SQL language-based expressions to define criteria used to filter to identify messages that will be copied over to subscription.

One thing I’ll mention is the idea of scope, which denotes the type of property; user-defined properties prefixed with user and system defined properties prefixed with sys.

With SQL filters it’s possible to create very complex rules for filtering messages out. Keep in mind that the more complex these rules are, the higher performance tall will be on the broker will have to apply these rules to every message. An example of a SQL rule:

sys.Label LIKE '%bus%'` OR `user.tag IN ('queue', 'topic', 'subscription')

## Correlation filters

Unlike Boolean and SQL filters, this group is used to perform matching against one or more user and system properties in a very efficient way.

The CorrelationFilter provides an efficient filter that deal with equality only. As such, the cost of evaluating filter expression is minimal and almost immediate w/o extra compute required

The only challenge with this filter, it’s not quite clear how to use it. There are two constructors

1. Default (empty) constructor
2. Constructor taking a single argument

var filter = new CorrelationFilter();

filter.Label = "blah";

filter.ReplyTo = "x";

filter.Properties["prop1"] = "abc";

filter.Properties["prop2"] = "xyz";

Created filter will have the following criteria:

sys.ReplyTo = 'x' AND sys.Label = 'blah' AND prop1 = 'abc' AND prop2 = 'xyz'

And you’ve guessed it right. When correlating on multiple properties, logical AND will be used so that all properties have to have the expected values for the filter to be evaluated as truthy.

In case you wondered how user-defined properties are populated, here's an example:

message.Properties["prop1"] = "abc";

static void CreateTopicsAndSubscriptions(NamespaceManager namespaceManager)

{

Console.WriteLine("\nCreating a topic and 3 subscriptions.");

// Create a topic and 3 subscriptions.

TopicDescription topicDescription = namespaceManager.CreateTopic(Program.TopicName);

Console.WriteLine("Topic created.");

// Create a subscription for all messages sent to topic.

namespaceManager.CreateSubscription(topicDescription.Path, SubsNameAllMessages, new TrueFilter());

Console.WriteLine("Subscription {0} added with filter definition set to TrueFilter.", Program.SubsNameAllMessages);

// Create a subscription that'll receive all orders which have color "blue" and quantity 10.

namespaceManager.CreateSubscription(topicDescription.Path, SubsNameColorBlueSize10Orders, new SqlFilter("color = 'blue' AND quantity = 10"));

Console.WriteLine("Subscription {0} added with filter definition \"color = 'blue' AND quantity = 10\".", Program.SubsNameColorBlueSize10Orders);

//var ruleDesc = new RuleDescription();

//ruleDesc.Filter = new CorrelationFilter("high");

//ruleDesc.Action = new SbAction();

var filter = new CorrelationFilter();

filter.Label = "blah";

filter.ReplyTo = "x";

filter.Properties["prop1"] = "abc";

filter.Properties["prop2"] = "xyz";

// Create a subscription that'll receive all high priority orders.

namespaceManager.CreateSubscription(topicDescription.Path, SubsNameHighPriorityOrders, new CorrelationFilter("high"));

// filter = new CorrelationFilter("high")

Console.WriteLine("Subscription {0} added with correlation filter definition \"high\".", Program.SubsNameHighPriorityOrders);

Console.WriteLine("Create completed.");

}

### Step 1 - Create Your Topic

Subscriptions are based off of topics, so we need to ensure we have a topic. Here’s some simple code to create a topic if it doesn’t already exist:

string connectionString = // your servicebus connection string here;

// the names of topics and subscriptions we'll be working with

const string topicName = "MyTestTopic";

const string allMessagesSubName = "AllMessages";

const string filteredSubName1 = "Filtered1";

const string filteredSubName2 = "Filtered2";

// let's create the topic if it doesn't already exist...

var namespaceManager = NamespaceManager.CreateFromConnectionString(connectionString);

if (!namespaceManager.TopicExists(topicName))

{

var td = new TopicDescription(topicName);

namespaceManager.CreateTopic(td);

}

### Step 2 – Create the Filtered Subscriptions

You can create subscriptions at any time, but one important thing to remember with topics is that unlike queues, if no one is listening, messages sent to the topic will be lost. So make sure you create your subscriptions before sending messages to that topic.

Let’s create three subscriptions on our topic. The first one will have no filter, and so will receive all messages.

The second is going to filter on a user defined message property called “From” that we will add to each message we send. It uses a SqlFilter which lets us use a SQL like syntax to say that we want all messages whose ‘From’ property ends in ‘Smith’.

Our third subscription will use a built-in property of the BrokeredMessage. We want to only receive messages whose Label property is set to “important”. We have to prefix Label with sys. in order to indicate this is a built-in property rather than one of the custom user defined properties.

if (!namespaceManager.SubscriptionExists(topicName, allMessagesSubName))

{

namespaceManager.CreateSubscription(topicName, allMessagesSubName);

}

if (!namespaceManager.SubscriptionExists(topicName, filteredSubName1))

{

namespaceManager.CreateSubscription(topicName, filteredSubName1, new SqlFilter("From LIKE '%Smith'"));

}

if (!namespaceManager.SubscriptionExists(topicName, filteredSubName2))

{

namespaceManager.CreateSubscription(topicName, filteredSubName2, new SqlFilter("sys.Label='important'"));

}

### Step 3 – Send Some Messages

You don’t send your messages directly to a subscription, you send them to the topic, and that will forward them to all the relevant subscriptions based on their filters.

Here we send three messages, setting up the “From” user defined property for each one, and also setting the built-in Label property for two of them.

var message1 = new BrokeredMessage("Hello World");

message1.Properties["From"] = "Ian Wright";

var message2 = new BrokeredMessage("Second message");

message2.Properties["From"] = "Alan Smith";

message2.Label = "important";

var message3 = new BrokeredMessage("Third message");

message3.Properties["From"] = "Kelly Smith";

message3.Label = "information";

var client = TopicClient.CreateFromConnectionString(connectionString, topicName);

client.Send(message1);

client.Send(message2);

client.Send(message3);

client.Close();

### Step 4 – Receive Messages

Now we need to listen on each of those three subscriptions. I’m just going to use the SubscriptionClient’s ReceiveBatch method to pull off a batch of messages from each subscription. Here’s how to perform ReceiveBatch for a single subscription:

var subClient = SubscriptionClient.CreateFromConnectionString(connectionString, topicName, subscriptionName);

var received = subClient.ReceiveBatch(10, TimeSpan.FromSeconds(5));

foreach (var message in received)

{

Console.WriteLine("{0} '{1}' Label: '{2}' From: '{3}'",

subscriptionName,

message.GetBody<string>(),

message.Label,

message.Properties["From"]);

}

subClient.Close();

If this works correctly, our unfiltered subscription will receive all three messages, the "From LIKE '%Smith'" filter will get two, and the "sys.Label='important'" filter will get one:

AllMessages 'Hello World' Label: '' From: 'Ian Wright'

AllMessages 'Second message' Label: 'important' From: 'Alan Smith'

AllMessages 'Third message' Label: 'information' From: 'Kelly Smith'

Filtered1 'Second message' Label: 'important' From: 'Alan Smith'

Filtered1 'Third message' Label: 'information' From: 'Kelly Smith'

Filtered2 'Second message' Label: 'important' From: 'Alan Smith'

Perfect!

### Bonus Step – Modifying Your Filters

You might get into a situation where you’ve already created a subscription but now you want it to be filtered. It is possible to change the filter, by deleting the default “rule” that was created when you initially created the subscription (which will be called “$Default”), and creating your own new rule with the new subscription. For safety, let’s add the new rule before we delete the old one, to eliminate the chance that we miss a message we wanted during the window when there are no rules.

var subClient = SubscriptionClient.CreateFromConnectionString(connectionString, topicName, subscriptionName);

var newRule = new RuleDescription("FilteredRule", new SqlFilter("From LIKE '%Smith'"));

subClient.AddRule(newRule);

subClient.RemoveRule("$Default");

And now your subscription has the new filter applied. Note that any messages already forwarded to this subscription before this filter was applied will still be in the subscription even if they don’t match the new filter.

# What is Azure Relay?

The Azure Relay service enables you to securely expose services that run in your corporate network to the public cloud. You can do the same without opening a port on your firewall, or making any changes to your corporate network infrastructure.

The relay service supports the following scenarios between on-premises services and applications running in the cloud or in another on-premises environment.

* Traditional one-way, request/response, and peer-to-peer communication
* Event distribution at internet-scope to enable publish/subscribe scenarios
* Bi-directional and unbuffered socket communication across network boundaries.

In the relayed data transfer pattern, the basic steps involved are:

1. An on-premises service connects to the relay service through an outbound port.
2. It creates a bi-directional socket for communication tied to a particular address.
3. The client can then communicate with the on-premises service by sending traffic to the relay service targeting that address.
4. The relay service then *relays* data to the on-premises service through the bi-directional socket dedicated to the client. The client doesn't need a direct connection to the on-premises service. It doesn't need to know the location of the service. And, the on-premises service doesn't need any inbound ports open on the firewall.

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

* [Event Grid](https://docs.microsoft.com/en-us/azure/event-grid/)
* [Event Hubs](https://docs.microsoft.com/en-us/azure/event-hubs/)
* [Service Bus](https://docs.microsoft.com/en-us/azure/service-bus-messaging/)

**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](https://azure.com/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.

### 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

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

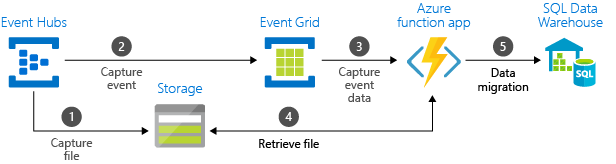
### Service Bus

Service Bus is intended for traditional enterprise applications. These enterprise applications require transactions, ordering, duplicate detection, and instantaneous consistency. Service Bus enables [cloud-native](https://azure.microsoft.com/overview/cloudnative/) 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



<https://dzone.com/articles/windows-azure-service-bus-1>

public class HomeController : Controller

{

public ActionResult Index()

{

return RedirectToAction("Submit");

}

public ActionResult About()

{

ViewBag.Message = "Your application description page.";

return View();

}

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;

//Process Message

QueueConnector.queueClient.OnMessage(ProcessMessage);

var topic = namespaceManager.GetTopic(QueueConnector.topicName);

ViewBag.TopicMessageCount = topic.MessageCountDetails;

QueueConnector.topicSubscriptionClient.OnMessage(ProcessTopicMessage,new OnMessageOptions { AutoComplete = false,AutoRenewTimeout = TimeSpan.FromMinutes(1)});

return View();

}

private void ProcessMessage(BrokeredMessage brokeredMessage)

{

var fata = brokeredMessage;

var data = brokeredMessage.GetBody<OnlineOrder>();

}

private void ProcessTopicMessage(BrokeredMessage brokeredMessage)

{

var fata = brokeredMessage;

var data = brokeredMessage.GetBody<OnlineOrder>();

}

[HttpPost]

// Attribute to help prevent cross-site scripting attacks and

// cross-site request forgery.

[ValidateAntiForgeryToken]

public ActionResult Submit(OnlineOrder order)

{

if (ModelState.IsValid)

{

// Create a message from the order.

var message = new BrokeredMessage(order);

// Submit the order.

QueueConnector.queueClient.Send(message);

return RedirectToAction("Submit");

}

else

{

return View(order);

}

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

using Microsoft.ServiceBus;

using Microsoft.ServiceBus.Messaging;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace FrontEndRole

{

public static class QueueConnector

{

public static QueueClient queueClient;

//public static TopicClient topicClient ;

public static SubscriptionClient topicSubscriptionClient;

public const string serviceBusNamespace = "trialservicebusss";

public const string queueName = "msgorderqueue";

public const string topicName = "msgordertopic";

public const string subcription = "msgordertopicsubcription";

public static NamespaceManager CreateNamespaceManager()

{

var uri = ServiceBusEnvironment.CreateServiceUri("sb", serviceBusNamespace, string.Empty);

var token = TokenProvider.CreateSharedAccessSignatureTokenProvider("RootManageSharedAccessKey", "70JDjrWoYUPsJ6bCj1qoluypw3zXN0qInSYZrsAOY3I=");

return new NamespaceManager(uri, token);

}

public static void Initialize()

{

ServiceBusEnvironment.SystemConnectivity.Mode = ConnectivityMode.Http;

var namespaceManager = CreateNamespaceManager();

if (!namespaceManager.QueueExists(queueName))

{

namespaceManager.CreateQueue(queueName);

}

var messagingFactory = MessagingFactory.Create(

namespaceManager.Address,

namespaceManager.Settings.TokenProvider);

queueClient = messagingFactory.CreateQueueClient(queueName);

TopicDescription topicDescription = new TopicDescription(topicName)

{

MaxSizeInMegabytes = 5120,

DefaultMessageTimeToLive = new TimeSpan(0, 1, 0)

};

//checking the topic is exist or not and if not creating the same topic

if (!namespaceManager.TopicExists(topicName))

namespaceManager.CreateTopic(topicDescription);

//checking the subscription is exist or not and if not creating the same subscription

if (namespaceManager.SubscriptionExists(topicName,subcription))

namespaceManager.CreateSubscription(topicName, subcription);

//Adding subscription with filters

if (namespaceManager.SubscriptionExists(topicName, "highincome" + subcription))

{

SqlFilter highIncomeSqlFilter = new SqlFilter("income > 5000");

namespaceManager.CreateSubscription(topicName, "highincome" + subcription, highIncomeSqlFilter);

}

//Adding subscription with filters

if(namespaceManager.SubscriptionExists(topicName, "lowincome" + subcription))

{

SqlFilter lowhIncomeSqlFilter = new SqlFilter("income < 5000");

namespaceManager.CreateSubscription(topicName, "lowincome" + subcription, lowhIncomeSqlFilter);

}

messagingFactory = MessagingFactory.Create(

namespaceManager.Address,

namespaceManager.Settings.TokenProvider);

topicSubscriptionClient = messagingFactory.CreateSubscriptionClient(topicName, subcription);

}

}

}

protected void Application\_Start()

{

AreaRegistration.RegisterAllAreas();

FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);

RouteConfig.RegisterRoutes(RouteTable.Routes);

BundleConfig.RegisterBundles(BundleTable.Bundles);

FrontEndRole.QueueConnector.Initialize();

}

Shared Access Signatures are a claims-based authorization mechanism using simple tokens. Using SAS, keys are never passed on the wire.

SAS authentication in Service Bus is configured with named [Shared Access Authorization Rules](https://docs.microsoft.com/en-us/dotnet/api/microsoft.servicebus.messaging.sharedaccessauthorizationrule) 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](https://docs.microsoft.com/en-us/azure/service-bus-relay/relay-what-is-it), [queues](https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview#queues), and [topics](https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview#topics).

The [Shared Access Signature](https://docs.microsoft.com/en-us/dotnet/api/microsoft.servicebus.sharedaccesssignaturetokenprovider) 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/.

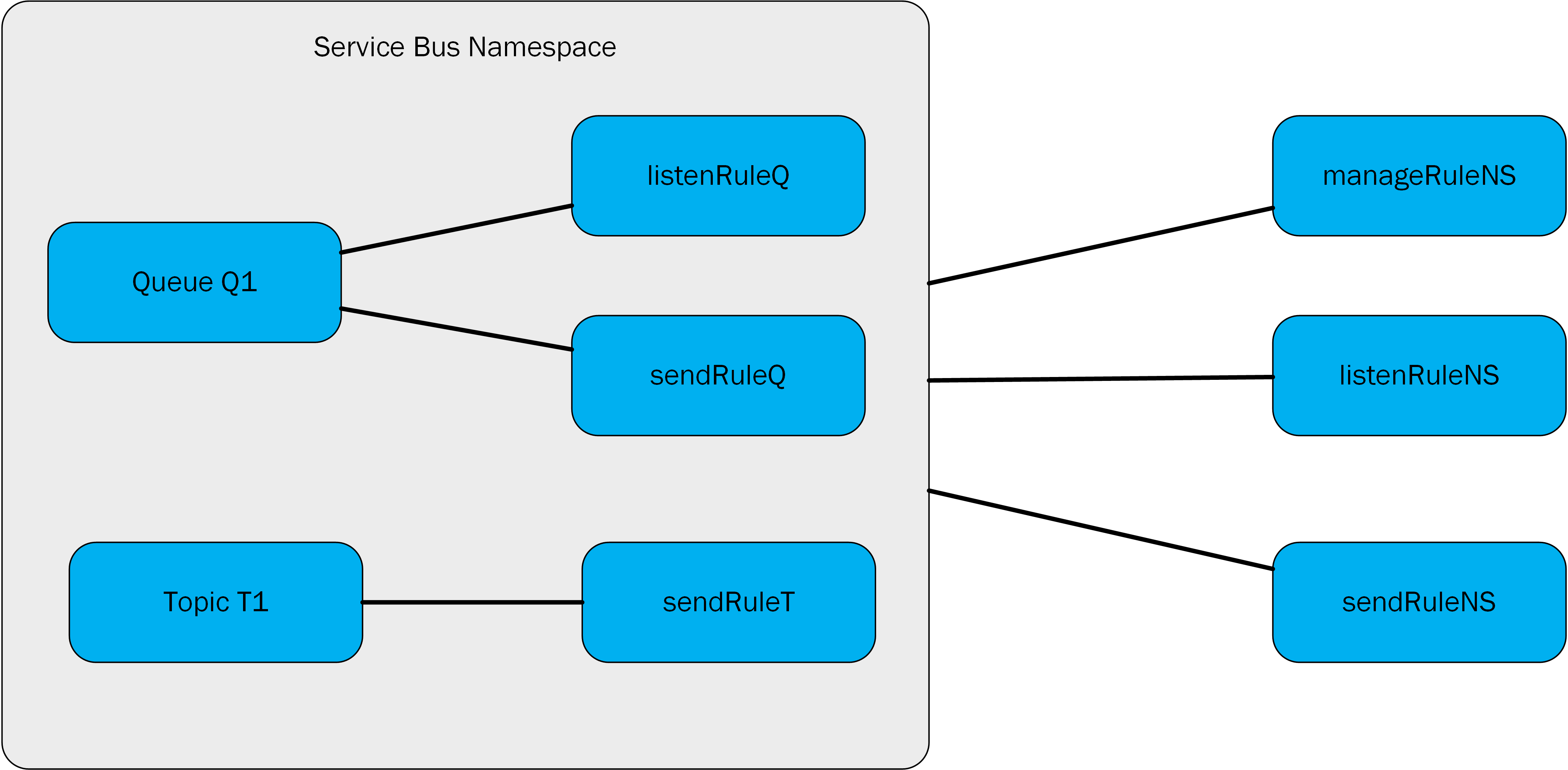
The rights conferred by the policy rule can be a combination of:

* '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.

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.



**namespace** Microsoft.Samples.SharedAccessSignatureAuthentication

{

**using** Microsoft.ServiceBus;

**using** Microsoft.ServiceBus.Messaging;

**using** System;

**using** System.Collections.Generic;

**using** System.Collections.Specialized;

**using** System.Linq;

**using** System.Net;

**using** System.Net.Http;

**using** System.Text;

**using** System.Threading.Tasks;

**class** EntityAuthRulesCRUD

    {

        // This method will create a queue "sampleQueues/contosoQ" with 2 shared access authorization rules

        // for the Listen & Send rights. It uses SharedAccessSignature auth to create the queue and manage the

        // authorization rules for the queue.

**public** **static** **void** CreateSASRuleOnEntity(**string** serviceNamespace, **string** qPath, **string** keyName, **string** key)

        {

            // Create an instance of NamespaceManager for the operation

            Uri managementUri = ServiceBusEnvironment.CreateServiceUri("https", serviceNamespace, **string**.Empty);

            TokenProvider sasTP = TokenProvider.CreateSharedAccessSignatureTokenProvider(keyName, key);

            NamespaceManager nsm = **new** NamespaceManager(managementUri, sasTP);

            QueueDescription qd = **new** QueueDescription(qPath);

            // Setup a rule with send rights with keyName as "contosoQSendKey"

            // and add it to the queue description.

            Program.contosoQSendRule = **new** SharedAccessAuthorizationRule("contosoQSendKey",

                SharedAccessAuthorizationRule.GenerateRandomKey(),

**new**[] { AccessRights.Send });

            qd.Authorization.Add(Program.contosoQSendRule);

            // Setup a rule with listen rights with keyName as "contosoQListenKey"

            // and add it to the queue description.

            Program.contosoQListenRule = **new** SharedAccessAuthorizationRule("contosoQListenKey",

                SharedAccessAuthorizationRule.GenerateRandomKey(),

**new**[] { AccessRights.Listen });

            qd.Authorization.Add(Program.contosoQListenRule);

            // Setup a rule with manage rights with keyName as "contosoQManageKey"

            // and add it to the queue description.

            // A rule with the Manage right MUST also have the Send & Receive rights.

            Program.contosoQManageRule = **new** SharedAccessAuthorizationRule("contosoQManageKey",

                SharedAccessAuthorizationRule.GenerateRandomKey(),

**new**[] { AccessRights.Manage, AccessRights.Listen, AccessRights.Send });

            qd.Authorization.Add(Program.contosoQManageRule);

            // Create the queue.

            nsm.CreateQueue(qd);

        }

        // This method rolls the primary key on a auth rule.

**public** **static** **void** RollSharedAccessKeysOnEntity(**string** serviceNamespace, **string** qPath, **string** keyName, **string** key)

        {

            // Create an instance of NamespaceManager for the operation.

            Uri managementUri = ServiceBusEnvironment.CreateServiceUri("https", serviceNamespace, **string**.Empty);

            TokenProvider sasTP = TokenProvider.CreateSharedAccessSignatureTokenProvider(keyName, key);

            NamespaceManager nsm = **new** NamespaceManager(managementUri, sasTP);

            // Get the queue description.

            QueueDescription qd = nsm.GetQueue(qPath);

            IEnumerator<AuthorizationRule> rulesEnumerator = qd.Authorization.GetEnumerator();

**while** (rulesEnumerator.MoveNext())

            {

                SharedAccessAuthorizationRule typedRule = rulesEnumerator.Current **as** SharedAccessAuthorizationRule;

**if** (typedRule != **null**) // Confirm that this is a 'SharedAccessAuthorizationRule'

                {

                    // Roll the keys.

                    // Note that this will also roll the keys on the 'contosoQManageRule' which is being used to

                    // authenticate this request, but since it won't take effect until we call NamespaceManager.UpdateQueue()

                    // that will still work.

                    typedRule.SecondaryKey = typedRule.PrimaryKey;

                    typedRule.PrimaryKey = SharedAccessAuthorizationRule.GenerateRandomKey();

                }

            }

            // Apply the updated rules

            nsm.UpdateQueue(qd);

        }

        // This method removes the auth rules with KeyName "contosoQSendKey" and "contosoQListenKey"

        // from the queue.

**public** **static** **void** RemoveSASRulesFromEntity(**string** serviceNamespace, **string** qPath, **string** keyName, **string** key)

        {

            // Create an instance of NamespaceManager for the operation.

            Uri managementUri = ServiceBusEnvironment.CreateServiceUri("https", serviceNamespace, **string**.Empty);

            TokenProvider sasTP = TokenProvider.CreateSharedAccessSignatureTokenProvider(keyName, key);

            NamespaceManager nsm = **new** NamespaceManager(managementUri, sasTP);

            // Get the queue description.

            QueueDescription qd = nsm.GetQueue(qPath);

            List<AuthorizationRule> rulesToRemove = **new** List<AuthorizationRule>();

            rulesToRemove = qd.Authorization.GetRules(rule =>

                {

                    SharedAccessAuthorizationRule typedRule = rule **as** SharedAccessAuthorizationRule;

**if** (typedRule != **null** &&

                        (String.Equals(typedRule.KeyName, "contosoQSendKey", StringComparison.Ordinal) ||

                        String.Equals(typedRule.KeyName, "contosoQListenKey", StringComparison.Ordinal)))

                    {

**return** **true**;

                    }

**return** **false**;

                });

            rulesToRemove.ForEach(**delegate**(AuthorizationRule rule)

            {

                qd.Authorization.Remove(rule);

            });

            nsm.UpdateQueue(qd);

        }

**public** **static** **void** DeleteQueue(**string** serviceNamespace, **string** qPath, **string** keyName, **string** key)

        {

            // Create an instance of NamespaceManager for the operation.

            Uri managementUri = ServiceBusEnvironment.CreateServiceUri("https", serviceNamespace, **string**.Empty);

            TokenProvider sasTP = TokenProvider.CreateSharedAccessSignatureTokenProvider(keyName, key);

            NamespaceManager nsm = **new** NamespaceManager(managementUri, sasTP);

            // Delete the queue.

            nsm.DeleteQueue(qPath);

        }

    }

}

**namespace** Microsoft.Samples.SharedAccessSignatureAuthentication

{

**using** Microsoft.ServiceBus;

**using** Microsoft.ServiceBus.Messaging;

**using** System;

**using** System.Collections.Generic;

**using** System.Linq;

**using** System.Text;

**using** System.Threading.Tasks;

**class** SASAuthTokenProvider

    {

**public** **static** **void** SendMessageToQ(**string** serviceNamespace, **string** qPath, **string** keyName, **string** key)

        {

            Uri runtimeUri = ServiceBusEnvironment.CreateServiceUri("sb", serviceNamespace, **string**.Empty);

            MessagingFactory mf = MessagingFactory.Create(runtimeUri,

                TokenProvider.CreateSharedAccessSignatureTokenProvider(keyName, key));

            QueueClient sendClient = mf.CreateQueueClient(qPath);

            //Sending message to queue.

            Console.WriteLine("Sending Hello message to queue.");

            BrokeredMessage sentMessage = CreateHelloMessage();

            sendClient.Send(sentMessage);

        }

**public** **static** **void** ReceiveMessageFromQ(**string** serviceNamespace, **string** qPath, **string** keyName, **string** key)

        {

            Uri runtimeUri = ServiceBusEnvironment.CreateServiceUri("sb", serviceNamespace, **string**.Empty);

            MessagingFactory mf = MessagingFactory.Create(runtimeUri,

                    TokenProvider.CreateSharedAccessSignatureTokenProvider(keyName, key));

            QueueClient receiveClient = mf.CreateQueueClient(qPath);

            BrokeredMessage receivedMessage = receiveClient.Receive(TimeSpan.FromSeconds(10));

            Console.WriteLine("Received message from queue: ID = {0}, Body = {1}.", receivedMessage.MessageId, receivedMessage.GetBody<**string**>());

        }

**private** **static** BrokeredMessage CreateHelloMessage()

        {

            BrokeredMessage helloMessage = **new** BrokeredMessage("Hello, Service Bus!");

            helloMessage.MessageId = "SAS-Sample-Message";

**return** helloMessage;

        }

    }

}