Brian Dekker

[E-mailadres]

Service Busses

Summary of azure service bus

Index

[What is a service bus. 2](#_Toc102831132)

[Queues 2](#_Toc102831133)

[Topics 2](#_Toc102831134)

[Setting up a service bus. 3](#_Toc102831135)

[Creating a service bus in the portal. 3](#_Toc102831136)

[Navigating service bus and creating queues/topics. 4](#_Toc102831137)

[Creating Queues. 5](#_Toc102831138)

[Python code to send and receive messages from queue. 7](#_Toc102831139)

[Connection the the service bus. 7](#_Toc102831140)

[Sending a message with python. 8](#_Toc102831141)

[The full code with list and batch messages. 10](#_Toc102831142)

[Creating Topics. 12](#_Toc102831143)

[Subscribers. 12](#_Toc102831144)

[Mircosoft course. 16](#_Toc102831145)

[Sender 16](#_Toc102831146)

[Receiver. 17](#_Toc102831147)

[Using a service bus queue with function apps. 20](#_Toc102831148)

[Connection between function app and service bus. 20](#_Toc102831149)

[The function.json 20](#_Toc102831150)

[Init.py 21](#_Toc102831151)

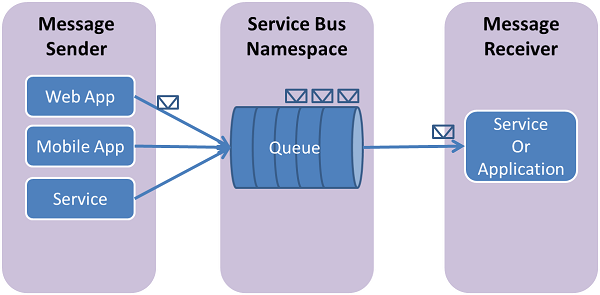
# What is a service bus.

A service bus is a fully managed message service in azure with queues and topics. Service bus is used to decouple applications from each other it kind of takes the roll as a middle man between services that you want to connect with each other.

* Load balancing
* Safely routing and transferring of data
* Coordinating transactional work

## Queues

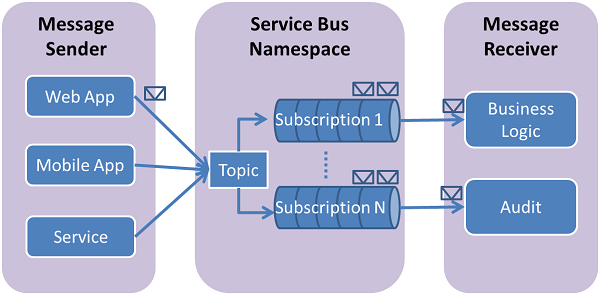
Messages are send and received in queues. Queues store messages for a set amount of time or until the receiving application is available to receive and process them.



Messages in queues are ordered and timestamped on arrival. Messages are delivered in pull mode, only delivering messages when it is requested by a receiver.

## Topics

Topics are the second option you have when it comes to handling messages in a service bus. While queues are used for point to point communication (one to one) topics are used to send the same message to multiple receivers (one to many) called subscribers.

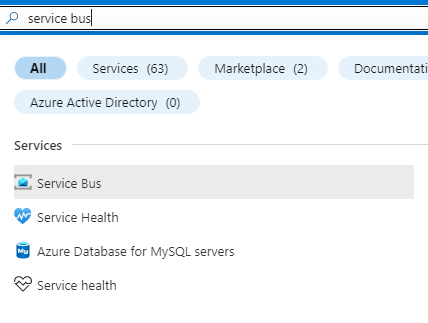


# Setting up a service bus.

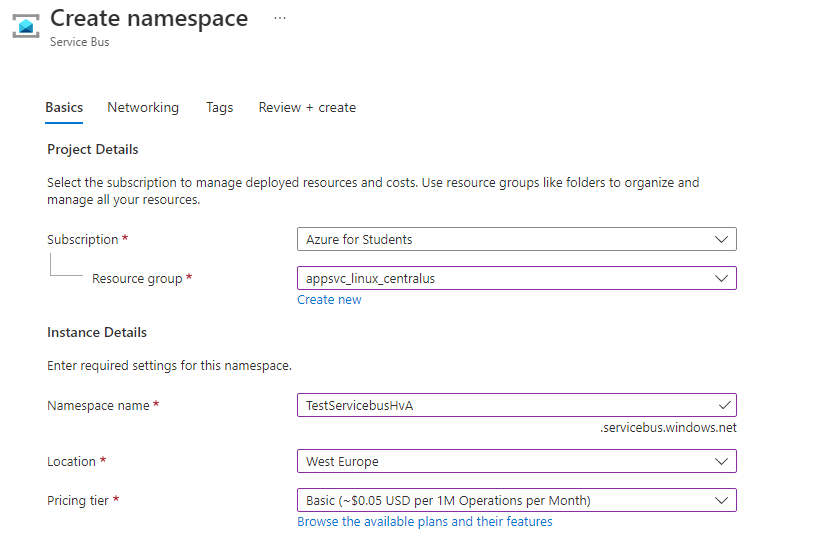
In this chapter I will go over how to setup a service bus and use it in function apps and raw code.

## Creating a service bus in the portal.

Like with pretty much every other azure service just look for service bus in the search bar and select it.

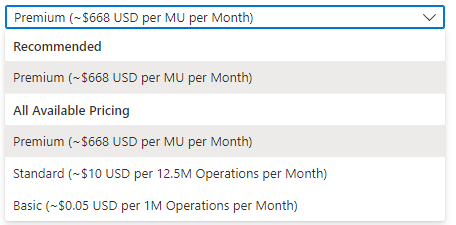


Now click create on the top right and fill in the information on the form



The only option that has a big impact in this form is the pricing tiers.

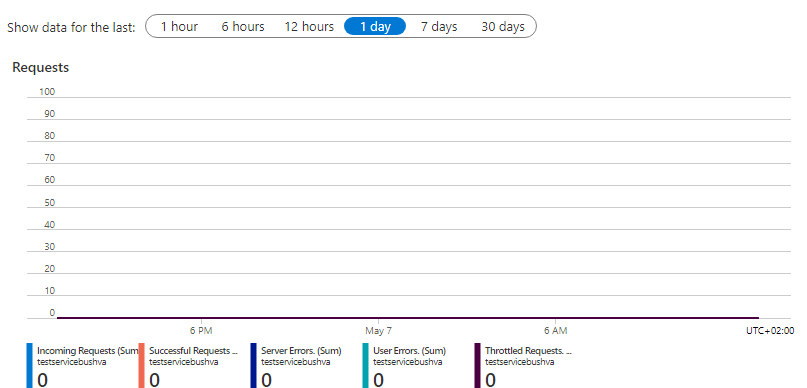
* Basic: Can only create queues and can not do all of the advanced settings
* Standard: Can do queues and topic and most if not all the advanced settings
* Premium: Isolates all operations very expensive

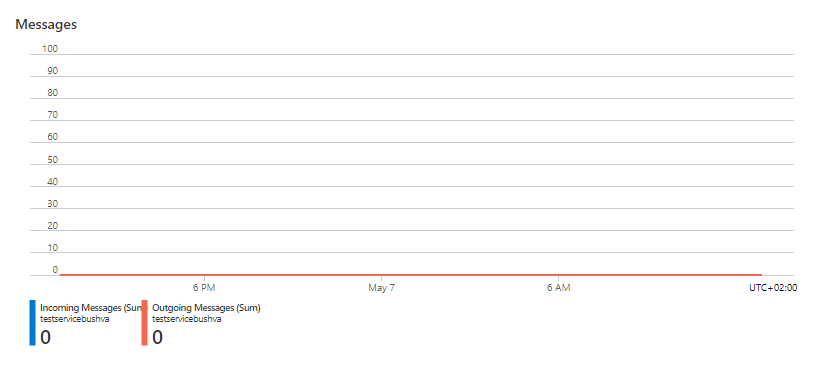


In order to show topics and queues I will choose the standard pricing tier.

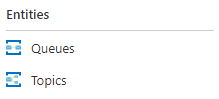
## Navigating service bus and creating queues/topics.

In the overview page of a service bus you can see how many request and messages are processed.



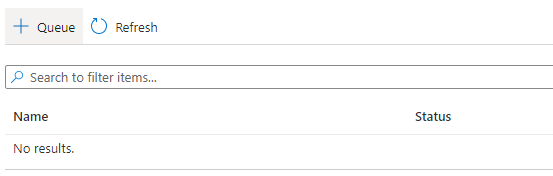


Now in order to make queues and topics we need to navigate the left menu to Entities usually it is visible without needing to scroll down.

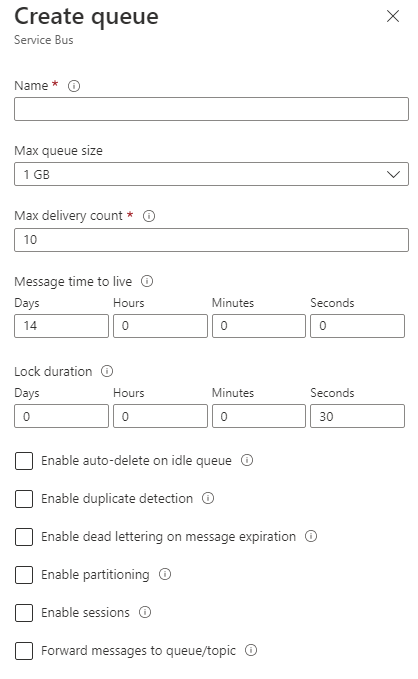


## Creating Queues.

When we select queues it will open a list view on the right side of all the queues in this service bus of course it will be empty right now.



To create a queue simply select +queue at the top a window will open on the far right of the screen with options for the queue.



Because we choose the standard pricing option we get to see more options then if we would choose basic most notably all the bottom option apart from partitioning and dead lettering are because we are on a standard tier.

**Max delivery count** means the amount of times a messages can be delivered to a receiver the default value of this is 10.

**Message time to live** is the time a message will stay in the queue before it is removed. When you deliver a message you can specify a custom time to live if you don’t it will take the default time.

**Lock duration** is the time a message is locked when ever it is requested by one receiver before another receiver can get the message. Default time is 30 seconds.

**Auto-delete queue** when you enable this you can specify a time when the queue should auto delete itself when its been idle for too long.

**Duplicate detection** this option will keep track of all messages send to the queue if a duplicate message is send it will ignore this message and not add it to the queue.

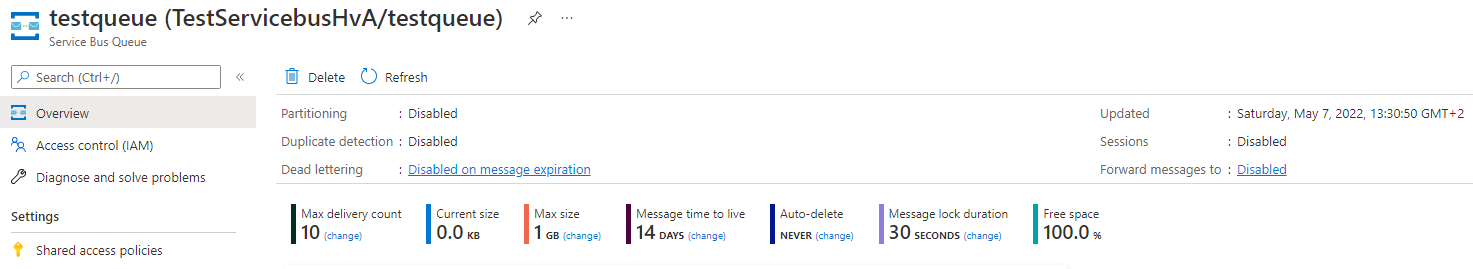
**Dead lettering** this means that if a message has failed to be used by any receiver if will be stored in a dead lettering queue messages are never deleted in this queue.

**Partitioning** is used to split a queue across multiple message brokers and message stores.

**Sessions** are used to allow ordered handling of unbounded sequences of related messages. If you want to guarantee FIFO delivery enable this.

**Forward messaging** is to forward messages in the queue to a different queue or topic automatically.

For now I leave everything on default and only fill in the name of the queue for this I chose “testqueue”.



Now that we have our queue we can start sending messages to it for this I used python.

## Python code to send and receive messages from queue.

Here I will share the python code I used to test the queue.

First we need to import 2 things.

from azure.servicebus import ServiceBusClient, ServiceBusMessage

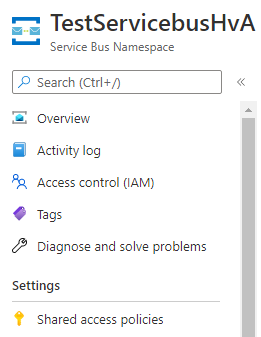
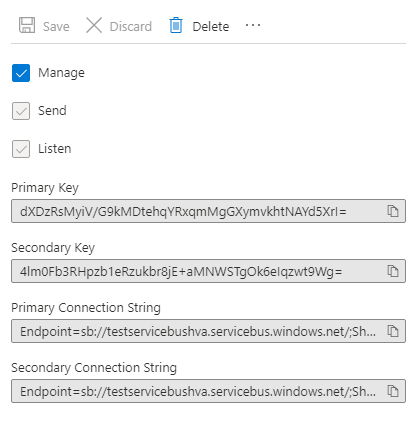
### Connection the the service bus.

CONNECTION\_STR = "Endpoint=sb://testservicebushva.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=dXDzRsMyiV/G9kMDtehqYRxqmMgGXymvkhtNAYd5XrI="

QUEUE\_NAME = "testqueue"

These 2 lines of code are important because this will allow our code to connect to our service bus and choose the correct queue to communicate with.

The connection string that you need to use in CONNECTION\_STR can be found in the shared access policy of our service bus.



When we go to our SAP if the service bus we can see one policy named RootManageSharedAccessPolicy select then one and a window on the right side will open with 4 keys copy the Primary Connection String key and paste it in the CONNECTION\_STR in python.

### Sending a message with python.

In order to send a message with python we first need to create a service bus client and sender.

# create a Service Bus client using the connection string

servicebus\_client = ServiceBusClient.from\_connection\_string(conn\_str=CONNECTION\_STR, logging\_enable=True)

This line of code will create the service bus client with our connection string.

sender = servicebus\_client.get\_queue\_sender(queue\_name=QUEUE\_NAME)

This line of code will create a sender that can send messages to the given queue in the servicebus.

Now we need to create a method that will send the message we want to the queue.

def send\_single\_message(sender):

    # create a Service Bus message

    singleMessage = "My name is Bob"

    message = ServiceBusMessage(singleMessage)

    # send the message to the queue

    sender.send\_messages(message)

    print("Message: " + singleMessage + " deliverd.")

In order to execute this method we call it with this bit on code:

with sender:

    # send one message

    send\_single\_message(sender)

Now in order to receive the message in the queue we need to create a receiver this is done by this line of code:

# get the Queue Receiver object for the queue

receiver = servicebus\_client.get\_queue\_receiver(queue\_name=QUEUE\_NAME, max\_wait\_time=5)

And in order to receive the message we need to use this bit of code:

with receiver:

    for msg in receiver:

        print("Received: " + str(msg))

        # complete the message so that the message is removed from the queue

        receiver.complete\_message(msg)

The last line in this code completes the message so that it is removed from the queue.

The full code should look something like this:

from azure.servicebus import ServiceBusClient, ServiceBusMessage

CONNECTION\_STR = "Endpoint=sb://testservicebushva.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=dXDzRsMyiV/G9kMDtehqYRxqmMgGXymvkhtNAYd5XrI="

QUEUE\_NAME = "testqueue"

# create a Service Bus client using the connection string

servicebus\_client = ServiceBusClient.from\_connection\_string(conn\_str=CONNECTION\_STR, logging\_enable=True)

sender = servicebus\_client.get\_queue\_sender(queue\_name=QUEUE\_NAME)

# get the Queue Receiver object for the queue

receiver = servicebus\_client.get\_queue\_receiver(queue\_name=QUEUE\_NAME, max\_wait\_time=5)

def send\_single\_message(sender):

    # create a Service Bus message

    singleMessage = "My name is Bob"

    message = ServiceBusMessage(singleMessage)

    # send the message to the queue

    sender.send\_messages(message)

    print("Message: " + singleMessage + " deliverd.")

with sender:

    # send one message

    send\_single\_message(sender)

print("Done sending messages")

print("-----------------------")

with receiver:

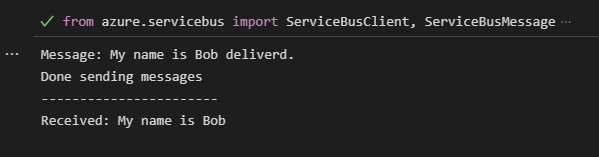
    for msg in receiver:

        print("Received: " + str(msg))

        # complete the message so that the message is removed from the queue

        receiver.complete\_message(msg)

Now if we execute this code the output should look like this:



### The full code with list and batch messages.

Here I will share the full code with 2 more methods of list and batch messages for a total of 16 message that are send and retrieved from the queue.

from azure.servicebus import ServiceBusClient, ServiceBusMessage

CONNECTION\_STR = "Endpoint=sb://testservicebushva.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=dXDzRsMyiV/G9kMDtehqYRxqmMgGXymvkhtNAYd5XrI="

QUEUE\_NAME = "testqueue"

# create a Service Bus client using the connection string

servicebus\_client = ServiceBusClient.from\_connection\_string(conn\_str=CONNECTION\_STR, logging\_enable=True)

sender = servicebus\_client.get\_queue\_sender(queue\_name=QUEUE\_NAME)

# get the Queue Receiver object for the queue

receiver = servicebus\_client.get\_queue\_receiver(queue\_name=QUEUE\_NAME, max\_wait\_time=5)

def send\_single\_message(sender):

    # create a Service Bus message

    singleMessage = "My name is Bob"

    message = ServiceBusMessage(singleMessage)

    # send the message to the queue

    sender.send\_messages(message)

    print("Message: " + singleMessage + " deliverd.")

def send\_a\_list\_of\_messages(sender):

    # create a list of messages

    listMessage = "This is a list of messages"

    messages = [ServiceBusMessage(listMessage) for \_ in range(5)]

    # send the list of messages to the queue

    sender.send\_messages(messages)

    print("Message: " + listMessage + " deliverd.")

def send\_batch\_message(sender):

    # create a batch of messages

    batchMessage = "This is a batch of messages"

    batch\_message = sender.create\_message\_batch()

    for \_ in range(10):

        try:

            # add a message to the batch

            batch\_message.add\_message(ServiceBusMessage(batchMessage))

        except ValueError:

            # ServiceBusMessageBatch object reaches max\_size.

            # New ServiceBusMessageBatch object can be created here to send more data.

            break

    # send the batch of messages to the queue

    sender.send\_messages(batch\_message)

    print("Message: " + batchMessage + " deliverd.")

with sender:

    # send one message

    send\_single\_message(sender)

    # send a list of messages

    send\_a\_list\_of\_messages(sender)

    # send a batch of messages

    send\_batch\_message(sender)

print("Done sending messages")

print("-----------------------")

with receiver:

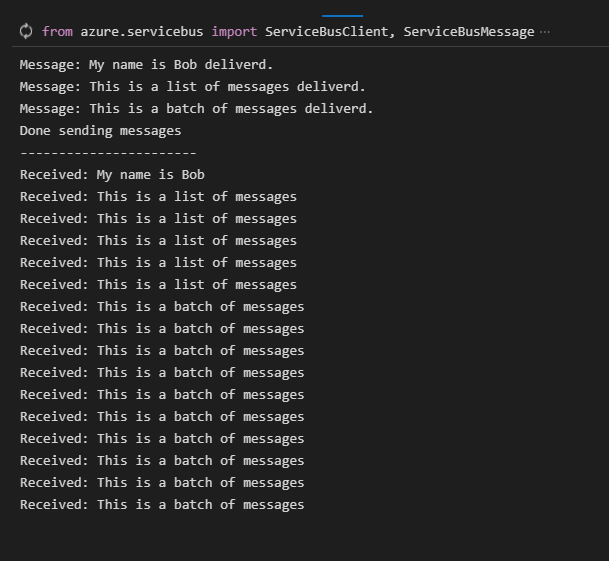
    for msg in receiver:

        print("Received: " + str(msg))

        # complete the message so that the message is removed from the queue

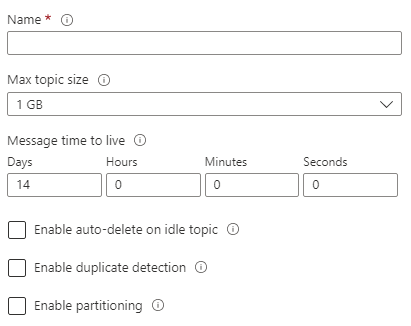
        receiver.complete\_message(msg)

And the output will look like this:



## Creating Topics.

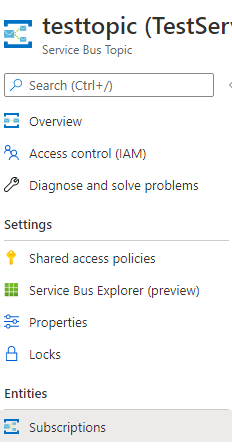
When creating a topic you have less option then when creating a queue. This is because the other option come into play when creating subscribers for this topic.



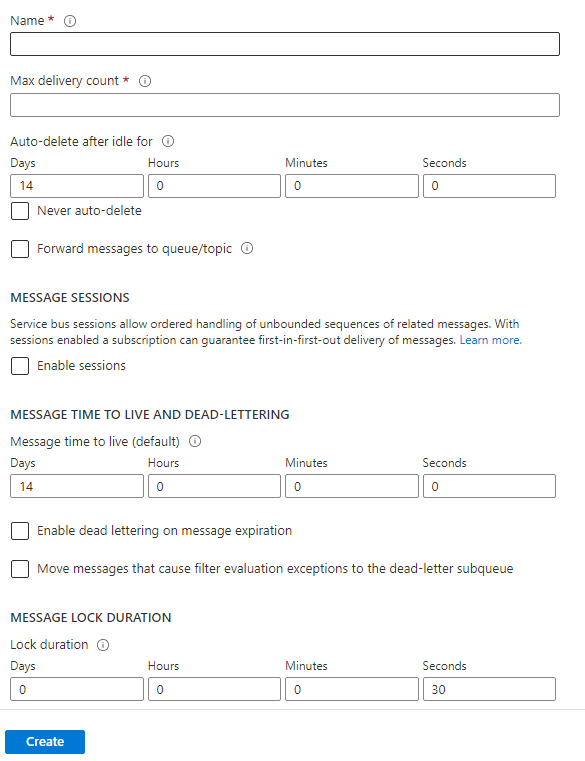
Just like with the queue I keep everything default and just fill in the name.

### Subscribers.

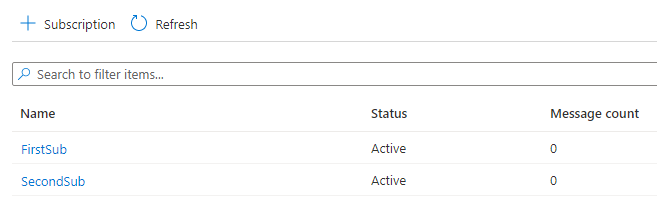
When you have created you topic you need to create subscribers that can receive messages from this topic.



When creating a subscriber you can see all the options that were missing from the create topic form come back in the subscribers creation form.

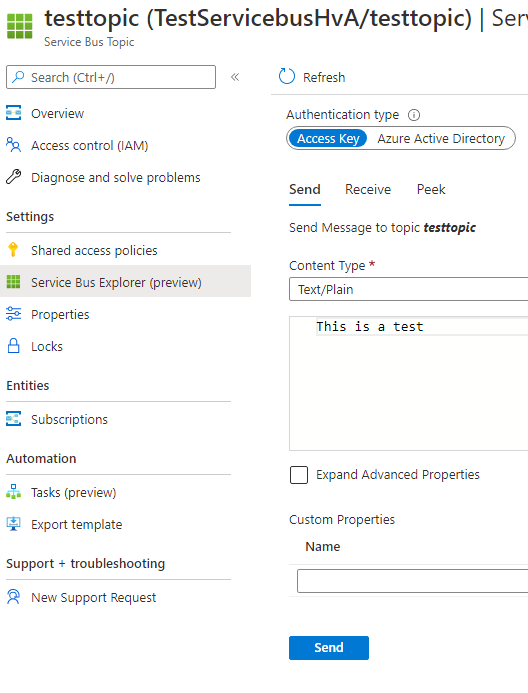


Again here I keep everything default and just fill in what is needed.

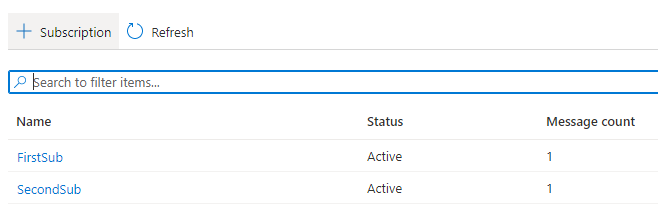


I have now created my first two subs.

Now lets send a message to this topic. In order to do this you can simply select servicebus explorer when you are on the topic layer and send a plain test message.

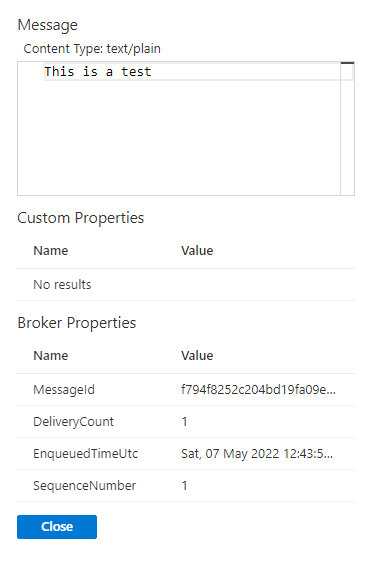
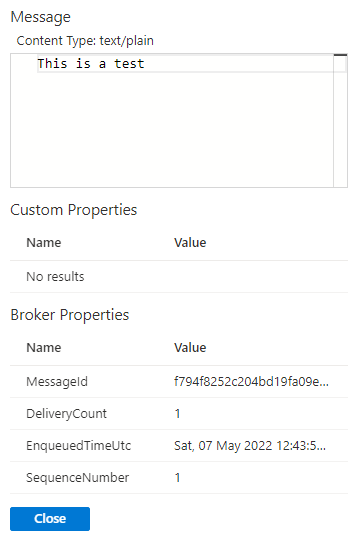


When I now click send the message will be send to both subscribers.



As you can see they both have 1 message.

If you now go to the topic layer and go to service bus explorer we can peek to see that the messages are indeed identical.



Now I will demonstrate with the help of a Microsoft course how to use topics using code.

## Mircosoft course.

This Microsoft course I did uses a sandbox in order to demo code that uses queues and topics. In this case I will only go over the topic code since I have not yet used that in the project. The code used in in C#.

### Sender

The sender code is as followed:

using System;

using System.Threading.Tasks;

using Azure.Messaging.ServiceBus;

namespace performancemessagesender

{

    class Program

    {

        const string ServiceBusConnectionString = "Endpoint=sb://learnservicebushva.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=R3yTHOVg8MNxc3UXOP2QJuztkUgyDZPCoiNwtde9U78=";

        const string TopicName = "salesperformancemessages";

        static void Main(string[] args)

        {

            Console.WriteLine("Sending a message to the Sales Performance topic...");

            SendPerformanceMessageAsync().GetAwaiter().GetResult();

            Console.WriteLine("Message was sent successfully.");

        }

        static async Task SendPerformanceMessageAsync()

        {

            // By leveraging "await using", the DisposeAsync method will be called automatically once the client variable goes out of scope.

            // In more realistic scenarios, you would store off a class reference to the client (rather than to a local variable) so that it can be used throughout your program.

            await using var client = new ServiceBusClient(ServiceBusConnectionString);

            await using ServiceBusSender sender = client.CreateSender(TopicName);

            try

            {

                string messageBody = "Total sales for Brazil in August: $13m.";

                var message = new ServiceBusMessage(messageBody);

                Console.WriteLine($"Sending message: {messageBody}");

                await sender.SendMessageAsync(message);

            }

            catch (Exception exception)

            {

                Console.WriteLine($"{DateTime.Now} :: Exception: {exception.Message}");

            }

        }

    }

}

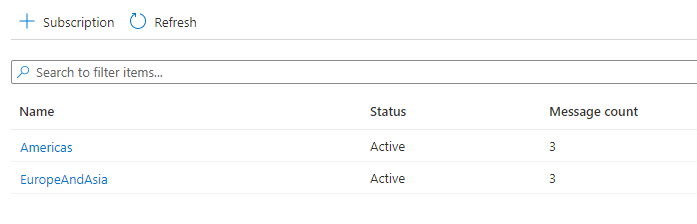
We see that is very similar to the python queue code we start with a connection string and the name of the topic we want to send our message to.

The piece of code that is responsible for sending the message is under the SendPreformanceMessageAsync method.

Here we can see just like in the python code we create a client and a sender once we have those we can create a message and use the sender to send the message to the topic. The message in this case is Total sales for Brazil in August: $13M.

If we execute this code and check our topic subscribers we can see that they now contain messages.

Executed code 3 times:



### Receiver.

The receiver code is as followed:

using System;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

using Azure.Messaging.ServiceBus;

namespace performancemessagereceiver

{

    class Program

    {

        const string ServiceBusConnectionString = "Endpoint=sb://alexgeddyneil.servicebus.windows.net/;SharedAccessKeyName=RootManageSharedAccessKey;SharedAccessKey=LIWIyxs8baqQ0bRf5zJLef6OTfrv0kBEDxFM/ML37Zs=";

        const string TopicName = "salesperformancemessages";

        const string SubscriptionName = "Americas";

        static void Main(string[] args)

        {

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

        }

        static async Task MainAsync()

        {

            var client = new ServiceBusClient(ServiceBusConnectionString);

            Console.WriteLine("======================================================");

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

            Console.WriteLine("======================================================");

            var processorOptions = new ServiceBusProcessorOptions

            {

                MaxConcurrentCalls = 1,

                AutoCompleteMessages = false

            };

            ServiceBusProcessor processor = client.CreateProcessor(TopicName, SubscriptionName, processorOptions);

            processor.ProcessMessageAsync += MessageHandler;

            processor.ProcessErrorAsync += ErrorHandler;

            await processor.StartProcessingAsync();

            Console.Read();

            await processor.DisposeAsync();

            await client.DisposeAsync();

        }

        static async Task MessageHandler(ProcessMessageEventArgs args)

        {

            Console.WriteLine($"Received message: SequenceNumber:{args.Message.SequenceNumber} Body:{args.Message.Body}");

            await args.CompleteMessageAsync(args.Message);

        }

        static Task ErrorHandler(ProcessErrorEventArgs args)

        {

            Console.WriteLine($"Message handler encountered an exception {args.Exception}.");

            Console.WriteLine("Exception context for troubleshooting:");

            Console.WriteLine($"- Endpoint: {args.FullyQualifiedNamespace}");

            Console.WriteLine($"- Entity Path: {args.EntityPath}");

            Console.WriteLine($"- Executing Action: {args.ErrorSource}");

            return Task.CompletedTask;

        }

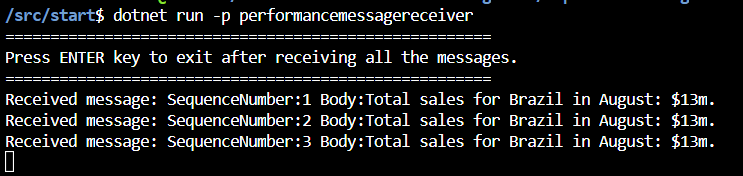
    }

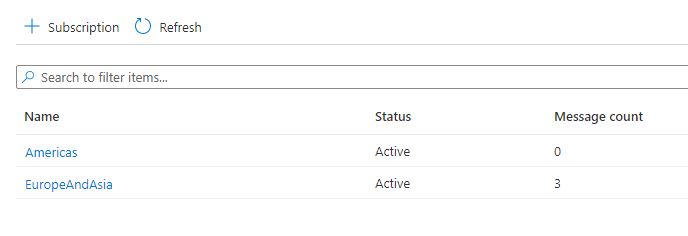
}

In this code we can see once again we need to define the connection string and the topic name but this time we also need a subscriber name so that we know where to get our message from in this case we choose Americas.

When we execute this code and check on our subscribers again we can see that Americas is now empty while the other one still has 3 messages because we didn’t use that subscriber.

Executed code:





## Using a service bus queue with function apps.

So for our project we decided to use a service bus in combination with a function app so I will show how to make that work.

### Connection between function app and service bus.

When you create a function app you can choose what type of trigger you want to use in our case it is the service bus queue trigger so when ever a message comes into the queue that the function app is listing to it will triggers its code.

When you select this option the default code for this action will be automatically generated for you. Now I did this in python which means you need to use visual studio code to edit and deploy your function app I will not go over this in the service bus summary but will go into detail on this topic in the function app summary.

Short version is when you have a function app you have 2 main files that work together in order to make the code work.

* Function.json this file is to define input and outputs that will be used in the function app code.
* A python file (or another language) with the actual code that will be executed.

### The function.json

The default will look like this

"scriptFile": "\_\_init\_\_.py",

"bindings": [

{

"name": "msg",

"type": "serviceBusTrigger",

"direction": "in",

"queueName": "textdatafromwebapp",

"connection": "MixitAppServiceBus\_SERVICEBUS"

}

This is the json that will say what scriptfile the function app should use and bindings are inputs and outputs we only have 1 binding so far and that is the trigger binding we chose when creating the function app.

In order to send a message to a queue with a function app we need to create a service bus output. Seen below:

{

"name": "msg3",

"direction": "out",

"type": "serviceBus",

"queueName": "quetowebapp",

"connection": "MixitAppServiceBus\_SERVICEBUS"

}

Here we declare a name of your choosing the type of binding in this case output the type of output so serviceBus the name of the queue we want to send it to and the connection string needed to connect to the service bus the queue is located in.

Once this is set up we can now use these input and output in our python code.

### Init.py

The init.py file is the file where we write our main function code that will be executed once the function is triggered.

The default when creating the function app with the trigger is this:

import logging

import azure.functions as func

def main(msg: func.ServiceBusMessage):

logging.info('Python ServiceBus queue trigger processed message: %s',

msg.get\_body().decode('utf-8'))

Here we see come imports and then the code its very simple it’s the main method with something as its parameter this parameter is important because this is basically the trigger that will receive the message from the first queue. So as of now once a message gets send to the queue named “testdatafromwebapp” it will trigger this function and print the message.

Now in order to use our declared output we need to add a parameter to the main function like so:

def main(msg: func.ServiceBusMessage,

msg3: func.Out[str]):

logging.info('Python ServiceBus queue trigger processed message: %s',

msg.get\_body().decode('utf-8'))

We have now coded that it will send a message back to the second queue once this function is triggered only we have not yet giving it a message we can do this by calling the set function.

def main(msg: func.ServiceBusMessage,

msg2: func.Out[str],

msg3: func.Out[str]):

logging.info('Python ServiceBus queue trigger processed message: %s',

msg.get\_body().decode('utf-8'))

msg2.set(msg.get\_body().decode('utf-8') + ' toevoeging StorageQueue')

msg3.set(msg.get\_body().decode('utf-8') + ' toevoeging ServiceBus')

This is the final version of my test code you can ignore the msg2 part is basically the same thing but for a storage queue but as you can see I set a message to msg3 and that message will be send back to the second queue declared in the function.json