



SERVICE BUSSES

Summary of azure service bus

Brian Dekker
[E-mailadres]

Index

What is a service bus.....	2
Queues	2
Topics.....	2
Setting up a service bus.....	3
Creating a service bus in the portal.....	3
Navigating service bus and creating queues/topics.....	4
Creating Queues.....	5
Python code to send and receive messages from queue.....	7
Connection the the service bus.....	7
Sending a message with python.....	8
The full code with list and batch messages.....	10
Creating Topics.....	12
Subscribers.....	12
Microsoft course.....	16
Sender.....	16
Receiver.....	17
Using a service bus queue with function apps.....	20
Connection between function app and service bus.....	20
The function.json.....	20
Init.py.....	21

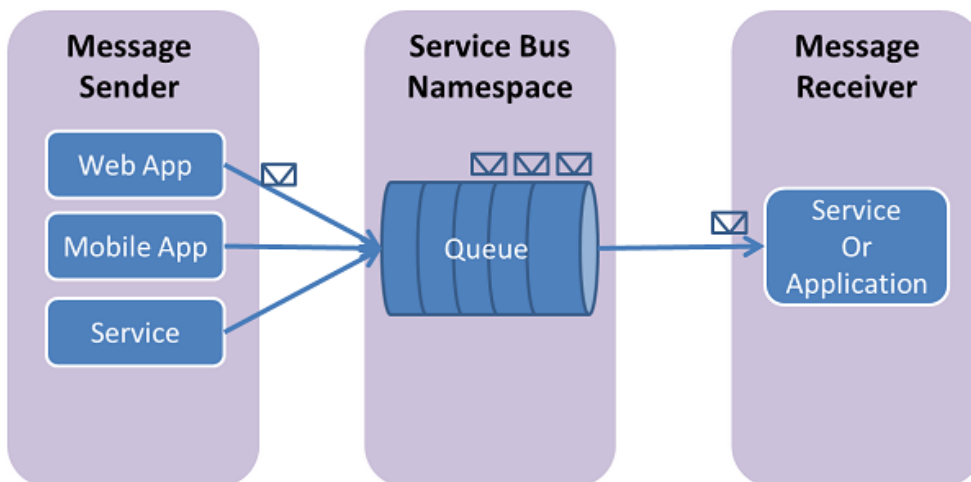
What is a service bus.

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

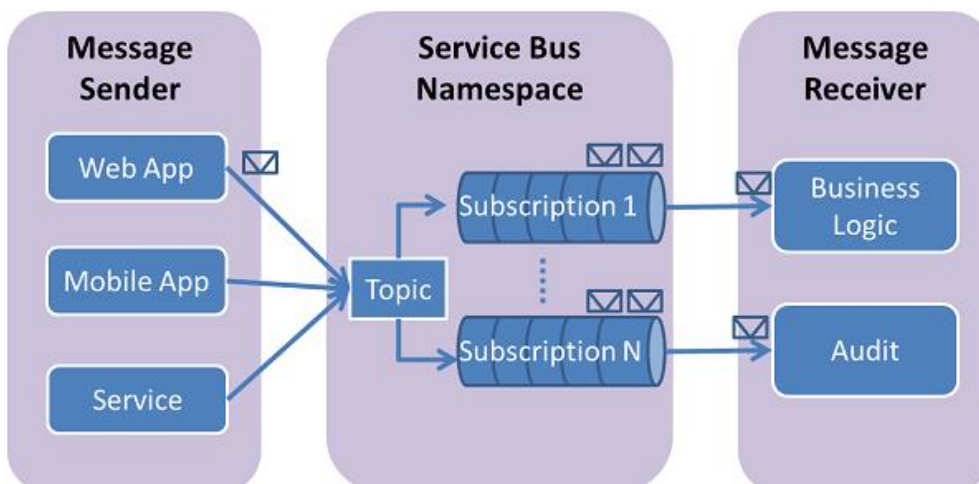
Messages are send and received in queues. Queues store messages for a set amount of time or until the receiving application is available to receive and process them.



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.

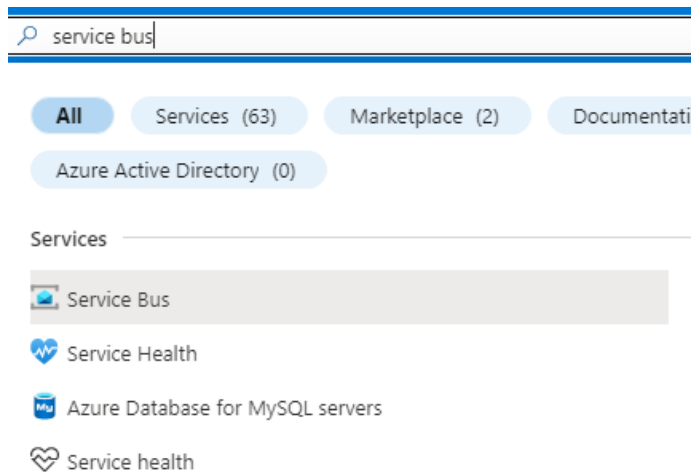


Setting up a service bus.

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

A screenshot of the 'Create namespace' form in the Azure portal. The form is titled 'Create namespace' with a 'Service Bus' icon and a three-dot menu. Below the title, there are tabs for 'Basics', 'Networking', 'Tags', and 'Review + create'. The 'Basics' tab is selected. Under 'Project Details', there is a description: 'Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.' Below this, there are two dropdown menus: 'Subscription *' (set to 'Azure for Students') and 'Resource group *' (set to 'appsvc_linux_centralus'). A 'Create new' link is visible below the resource group dropdown. Under 'Instance Details', there is a description: 'Enter required settings for this namespace.' Below this, there are three dropdown menus: 'Namespace name *' (set to 'TestServicebusHvA', with '.servicebus.windows.net' as a suffix), 'Location *' (set to 'West Europe'), and 'Pricing tier *' (set to 'Basic (~\$0.05 USD per 1M Operations per Month)'). A link 'Browse the available plans and their features' is visible below the pricing tier dropdown.

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

Premium (~\$668 USD per MU per Month) ▼

Recommended

Premium (~\$668 USD per MU per Month)

All Available Pricing

Premium (~\$668 USD per MU per Month)

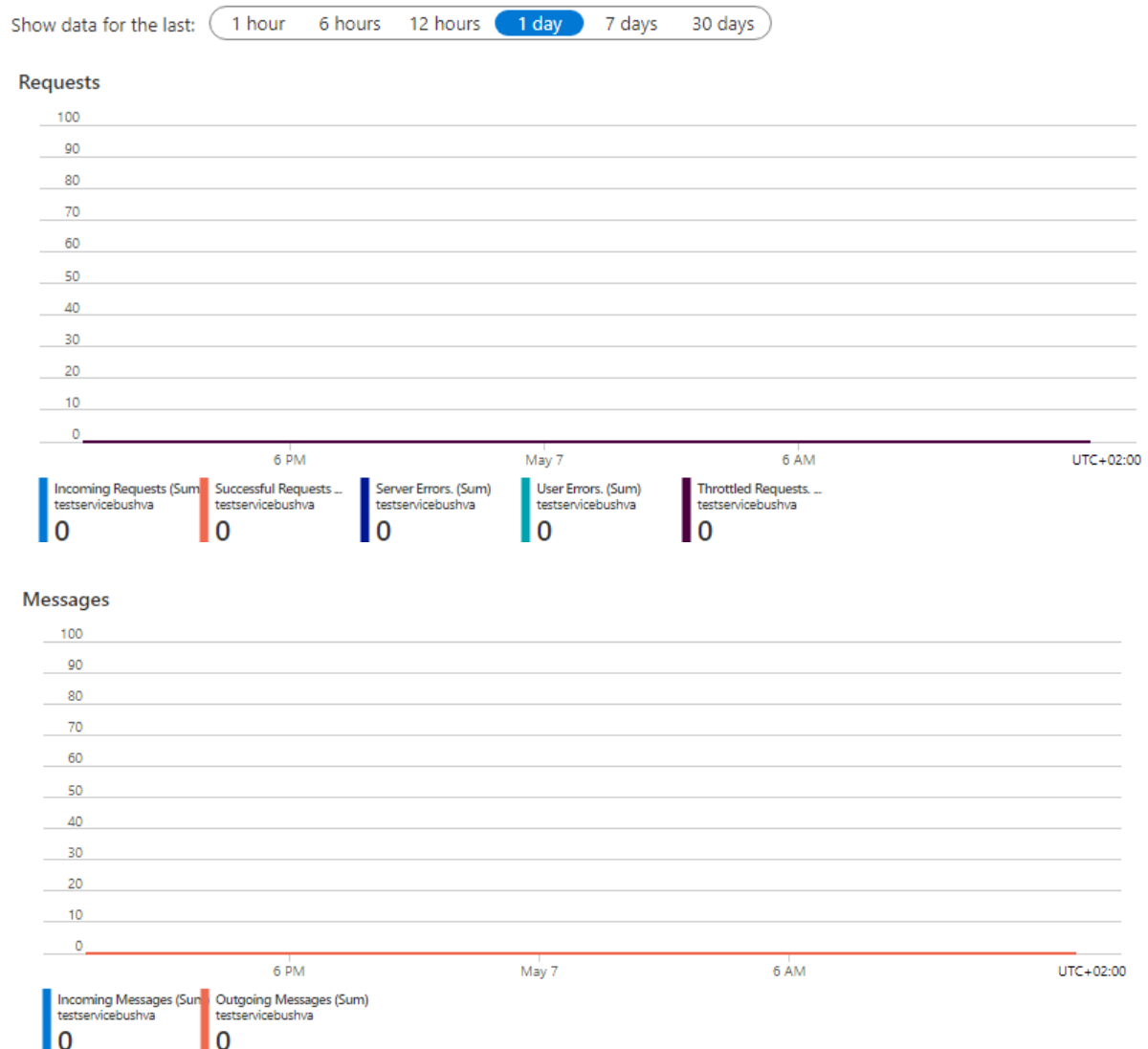
Standard (~\$10 USD per 12.5M Operations per Month)

Basic (~\$0.05 USD per 1M Operations per Month)

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.

Entities

 Queues

 Topics

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.

+ Queue Refresh

Name	Status
No results.	

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.

Create queue

Service Bus



Name * ⓘ

Max queue size

1 GB

▼

Max delivery count * ⓘ

Message time to live ⓘ

Days	Hours	Minutes	Seconds
<input type="text" value="14"/>	<input type="text" value="0"/>	<input type="text" value="0"/>	<input type="text" value="0"/>

Lock duration ⓘ

Days	Hours	Minutes	Seconds
<input type="text" value="0"/>	<input type="text" value="0"/>	<input type="text" value="0"/>	<input type="text" value="30"/>

☐ Enable auto-delete on idle queue ⓘ

☐ Enable duplicate detection ⓘ

☐ Enable dead lettering on message expiration ⓘ

☐ Enable partitioning ⓘ

☐ Enable sessions ⓘ

☐ Forward messages to queue/topic ⓘ

Because we choose the standard pricing option we get to see more options than 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 message 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 it 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".

The screenshot shows the Azure portal interface for a Service Bus Queue named 'testqueue' (TestServicebusHvA/testqueue). The left sidebar contains navigation links: Overview (selected), Access control (IAM), and Diagnose and solve problems. The main content area displays the queue's configuration. At the top, there is a search bar and buttons for Delete and Refresh. Below this, a table lists various settings: Partitioning (Disabled), Duplicate detection (Disabled), Dead lettering (Disabled on message expiration), Updated (Saturday, May 7, 2022, 13:30:50 GMT+2), Sessions (Disabled), and Forward messages to (Disabled). A summary bar at the bottom provides a quick overview of key settings: Max delivery count (10), Current size (0.0 KB), Max size (1 GB), Message time to live (14 DAYS), Auto-delete (NEVER), Message lock duration (30 SECONDS), and Free space (100.0 %).

Setting	Value
Max delivery count	10
Current size	0.0 KB
Max size	1 GB
Message time to live	14 DAYS
Auto-delete	NEVER
Message lock duration	30 SECONDS
Free space	100.0 %

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/G9kMDtehqYRxxmMgGXymvkhtNAYd5XrI="
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.

The screenshot shows the Azure Portal interface for a Service Bus Namespace named 'TestServicebusHvA'. The left sidebar contains navigation links: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, and Settings. The 'Settings' section is expanded, showing 'Shared access policies'. The right pane displays the configuration for the 'RootManageSharedAccessKey' policy. It includes checkboxes for 'Manage', 'Send', and 'Listen', all of which are checked. Below these are text boxes for the Primary Key, Secondary Key, Primary Connection String, and Secondary Connection String. The Primary Connection String is highlighted, showing the endpoint 'sb://testservicebushva.servicebus.windows.net/' and the shared access key 'dXDzRsMyiV/G9kMDtehqYRxxmMgGXymvkhtNAYd5XrI='.

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 + " delivered.")
```

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/G9kMDtehQYRxqmMgGXymvkhtNA
Yd5XrI="
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:

```

✓ from azure.servicebus import ServiceBusClient, ServiceBusMessage ...

... Message: My name is Bob deliverd.
    Done sending messages
    -----
    Received: My name is Bob

```

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 + " delivered.")

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:

```
from azure.servicebus import ServiceBusClient, ServiceBusMessage ...
```

[illegible]

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.

Name * ⓘ

Max topic size ⓘ

1 GB ▾

Message time to live ⓘ

Days	Hours	Minutes	Seconds
14	0	0	0

☐ Enable auto-delete on idle topic ⓘ


☐ Enable duplicate detection ⓘ


☐ Enable partitioning ⓘ


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.


 **testtopic (TestSer**
Service Bus Topic


 Overview


 Access control (IAM)


 Diagnose and solve problems

Settings


 Shared access policies

 Service Bus Explorer (preview)

 Properties

 Locks

Entities

 Subscriptions

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.

Name * ⓘ

Max delivery count * ⓘ

Auto-delete after idle for ⓘ

Days

Hours

Minutes

Seconds

☐ Never auto-delete

☐ Forward messages to queue/topic ⓘ

MESSAGE SESSIONS

Service bus sessions allow ordered handling of unbounded sequences of related messages. With sessions enabled a subscription can guarantee first-in-first-out delivery of messages. [Learn more.](#)

☐ Enable sessions

MESSAGE TIME TO LIVE AND DEAD-LETTERING

Message time to live (default) ⓘ

Days

Hours

Minutes

Seconds

☐ Enable dead lettering on message expiration

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

MESSAGE LOCK DURATION

Lock duration ⓘ

Days

Hours

Minutes

Seconds

Create

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


+ Subscription ↻ Refresh

🔍 Search to filter items...

Name	Status	Message count
FirstSub	Active	0
SecondSub	Active	0

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.

 **testtopic (TestServicebusHvA/testtopic)** | Send

Service Bus Topic

🔍 Search (Ctrl+/) ⏪ ↻ Refresh

Overview

Access control (IAM)

Diagnose and solve problems

Settings

Shared access policies

Service Bus Explorer (preview)

Properties

Locks

Entities

Subscriptions

Automation

Tasks (preview)

Export template

Support + troubleshooting

New Support Request

Authentication type ⓘ

Access Key Azure Active Directory

Send Receive Peek

Send Message to topic **testtopic**

Content Type *
Text/Plain

This is a test

☐ Expand Advanced Properties

Custom Properties

Name

Send

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

<div> <div>+</div> <div>Subscription</div> <div>↺ Refresh</div> </div>		
<div> <div>🔍</div> <div>Search to filter items...</div> </div>		
Name	Status	Message count
FirstSub	Active	1
SecondSub	Active	1

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.

Message

Content Type: text/plain

This is a test

Custom Properties

Name	Value
No results	

Broker Properties

Name	Value
MessageId	f794f8252c204bd19fa09e...
DeliveryCount	1
EnqueuedTimeUtc	Sat, 07 May 2022 12:43:5...
SequenceNumber	1

Close

Message

Content Type: text/plain

This is a test

Custom Properties

Name	Value
No results	

Broker Properties

Name	Value
MessageId	f794f8252c204bd19fa09e...
DeliveryCount	1
EnqueuedTimeUtc	Sat, 07 May 2022 12:43:5...
SequenceNumber	1

Close

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

Microsoft 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 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=R3yTHOVg8MNxc3UXOP2QJuztkUgyDZPCoiNw
tde9U78=";
        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:

+ Subscription
🔄 Refresh

Name	Status	Message count
Americas	Active	3
EuropeAndAsia	Active	3

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=RootM
anageSharedAccessKey;SharedAccessKey=LIWIyxs8baqQ0bRf5zJLef60Tfrv0kBEDxFM/ML37
Zs=";

    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:

```

/src/start$ dotnet run -p performancemessagereceiver
=====
Press ENTER key to exit after receiving all the messages.
=====
Received message: SequenceNumber:1 Body:Total sales for Brazil in August: $13m.
Received message: SequenceNumber:2 Body:Total sales for Brazil in August: $13m.
Received message: SequenceNumber:3 Body:Total sales for Brazil in August: $13m.

```

[+ Subscription](#) [Refresh](#)

Name	Status	Message count
Americas	Active	0
EuropeAndAsia	Active	3

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 listening to it will trigger 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 some imports and then the code is 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 sent 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 given 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 sent back to the second queue declared in the function.json