FUNCTION APPS

Summary azure function apps

Brian Dekker
Brian.dekker@hva.nl

Index

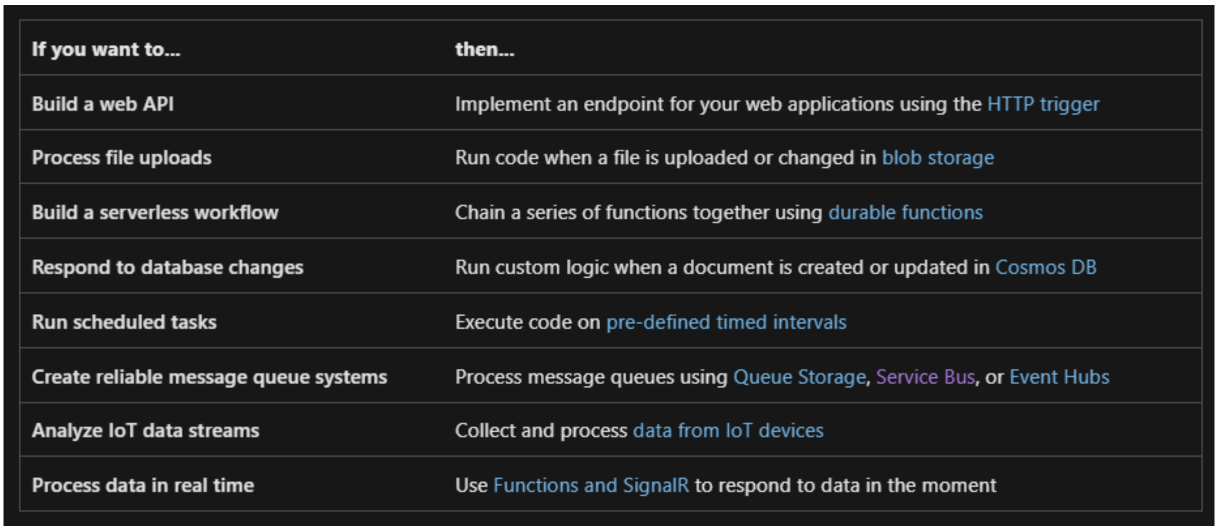
Function apps.

Azure functions is a serverless service that allows you to write less code and maintain less infrastructure. With function apps all you need to worry about is the code everything else is handled by azure functions.

Often we build systems that react to certain events. This can be something in a web app, database, queues and so on. Almost always in every application code needs to trigger whenever a certain event occurs.

Azure function can do this for you by using "Compute on demand".

Example Scenario's



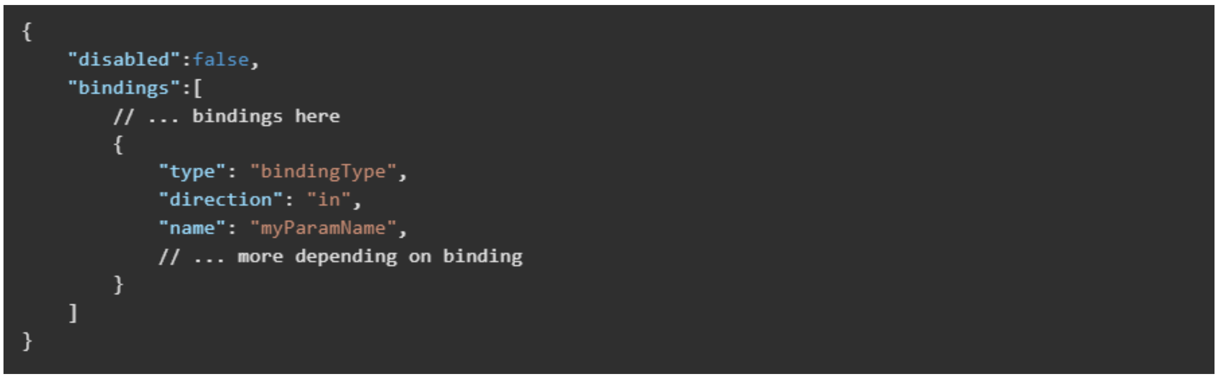
If you want to...	then...
Build a web API	Implement an endpoint for your web applications using the HTTP trigger
Process file uploads	Run code when a file is uploaded or changed in blob storage
Build a serverless workflow	Chain a series of functions together using durable functions
Respond to database changes	Run custom logic when a document is created or updated in Cosmos DB
Run scheduled tasks	Execute code on pre-defined timed intervals
Create reliable message queue systems	Process message queues using Queue Storage, Service Bus, or Event Hubs
Analyze IoT data streams	Collect and process data from IoT devices
Process data in real time	Use Functions and SignalR to respond to data in the moment

Function Code.

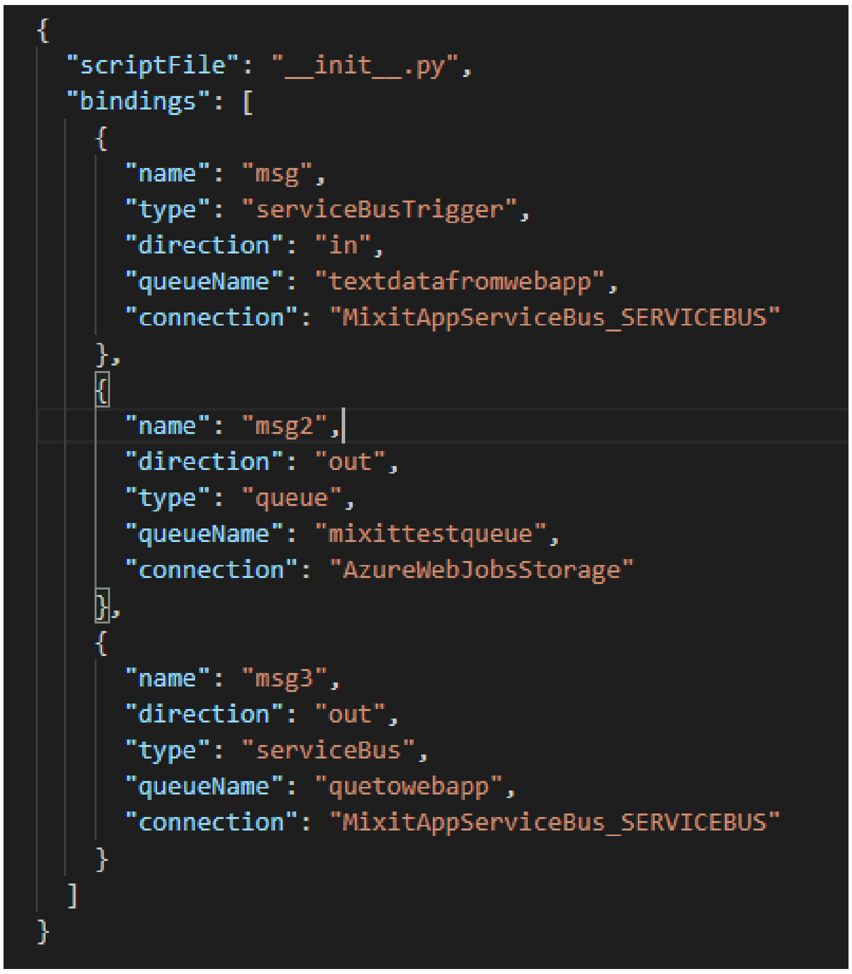
A *function* is the primary concept in Azure Functions. A function contains two files:

- File containing the code
- Function.json a config file

The function.json defines the function's trigger, bindings and other configuration settings. Every function has only one trigger but can have multiple in and out bindings. The runtime uses this config file to determine the events to monitor and how to pass and return data from a function execution. Here is are two example of a function.json file one is a concept the other one I use:



```
{
    "disabled":false,
    "bindings":[
        // ... bindings here
        {
            "type": "bindingType",
            "direction": "in",
            "name": "myParamName",
            // ... more depending on binding
        }
    ]
}
```



```json
{
  "scriptFile": "__init__.py",
  "bindings": [
    {
      "name": "msg",
      "type": "serviceBusTrigger",
      "direction": "in",
      "queueName": "textdatafromwebapp",
      "connection": "MixitAppServiceBus_SERVICEBUS"
    },
    {
      "name": "msg2",
      "direction": "out",
      "type": "queue",
      "queueName": "mixittestqueue",
      "connection": "AzureWebJobsStorage"
    },
    {
      "name": "msg3",
      "direction": "out",
      "type": "serviceBus",
      "queueName": "quetowebapp",
      "connection": "MixitAppServiceBus_SERVICEBUS"
    }
  ]
}
```

The bindings property is where you configure both triggers and bindings. Each individual binding has a few propertys in common these are:

- Type: Type of binding for example serviceBus or serviceBusTrigger
- Direction: Indicates whether the binding is receiving or sending data.
- Name : The name that is used in the code file when calling the binding.

The code file for this function.json looks like this:

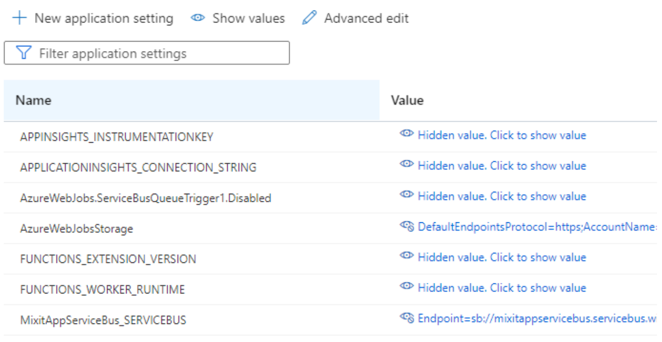
```python
import logging

import azure.functions as func

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')
```

Connections

As you can see in the function.json image all 3 bindings have a connection property this is a reference to the connection string it needs to use to connect with the service bus or storage account. The function.json will not accept raw connection strings so you have to create references in order to use them you can do this in the configuration settings of the function app see below:

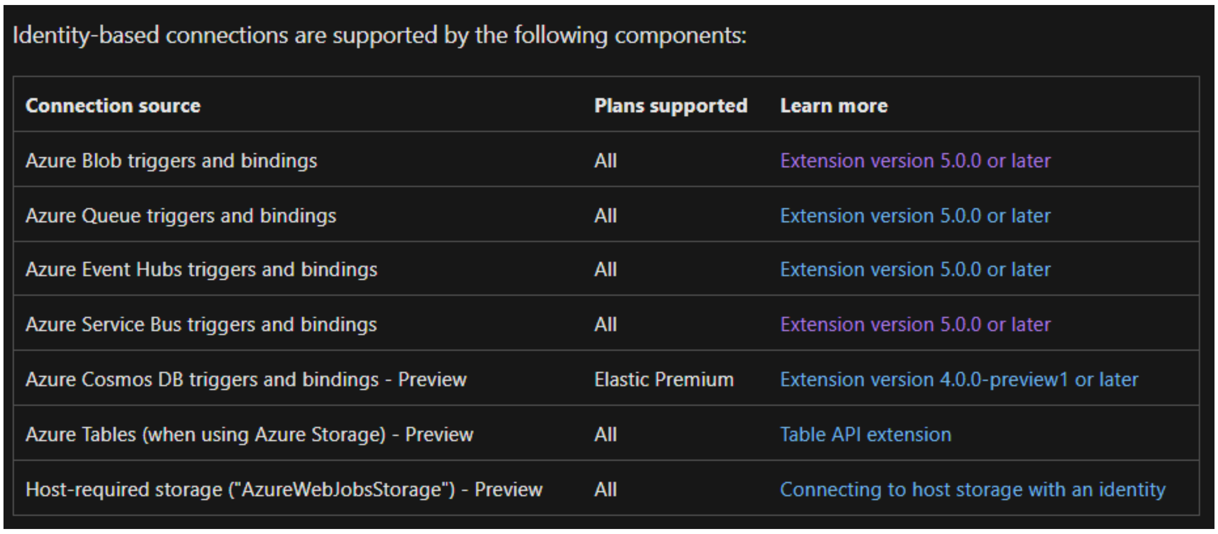


Name	Value
APPINSIGHTS_INSTRUMENTATIONKEY	Hidden value. Click to show value
APPLICATIONINSIGHTS_CONNECTION_STRING	Hidden value. Click to show value
AzureWebJobs.ServiceBusQueueTrigger1.Disabled	Hidden value. Click to show value
AzureWebJobsStorage	DefaultEndpointsProtocol=https;AccountName=
FUNCTIONS_EXTENSION_VERSION	Hidden value. Click to show value
FUNCTIONS_WORKER_RUNTIME	Hidden value. Click to show value
MixitAppServiceBus_SERVICEBUS	Endpoint=sb://mixitappservicebus.servicebus.w

Here you can see AzurewebjobsStorage and MixitAppServiceBus_SERVICEBUS connection that are used in the function.json and the start of the connection string in the value column. You can add more app settings here when needed.

Identity based connections.

Some connections in Azure functions can also be configured using an identity opposed to a secret. In some cases a connection string may still be required in Functions even though the service you are trying to connect to supports identity based connections.



Identity-based connections are supported by the following components:

Connection source	Plans supported	Learn more
Azure Blob triggers and bindings	All	Extension version 5.0.0 or later
Azure Queue triggers and bindings	All	Extension version 5.0.0 or later
Azure Event Hubs triggers and bindings	All	Extension version 5.0.0 or later
Azure Service Bus triggers and bindings	All	Extension version 5.0.0 or later
Azure Cosmos DB triggers and bindings - Preview	Elastic Premium	Extension version 4.0.0-preview1 or later
Azure Tables (when using Azure Storage) - Preview	All	Table API extension
Host-required storage ("AzureWebJobsStorage") - Preview	All	Connecting to host storage with an identity

When hosted in the Azure Function service identity based connections use a managed identity. The system-assigned identity is used by default however a user assigned identity can be specified with the credential and ClientID properties. When run as local development your developer identity will be used instead.

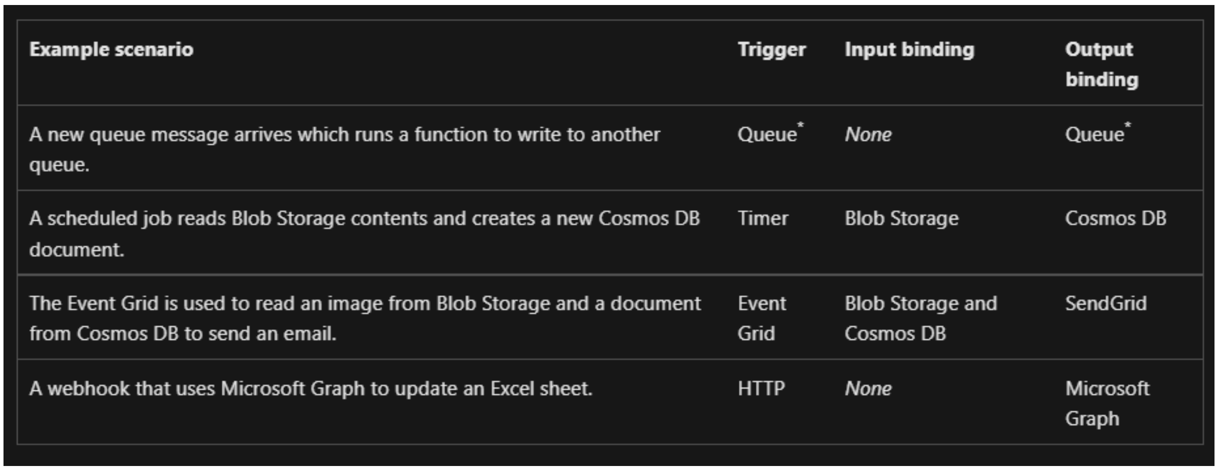
Triggers and Bindings.

Triggers are what causes a function to run. A trigger determines when a function will execute its code. A function can and must have only one trigger. Triggers have associated data, which is often provided as the payload of the function.

Bindings are a way to connect other resources to the function. Bindings may be connected as an input or an output or even in some rare cases as a inoutput. Data from bindings is provided to the function as parameters.

You can mix and match different bindings from all kinds of azure services to suit your needs. Unlike a trigger bindings are completely optional a function can have one or many inputs and outputs.

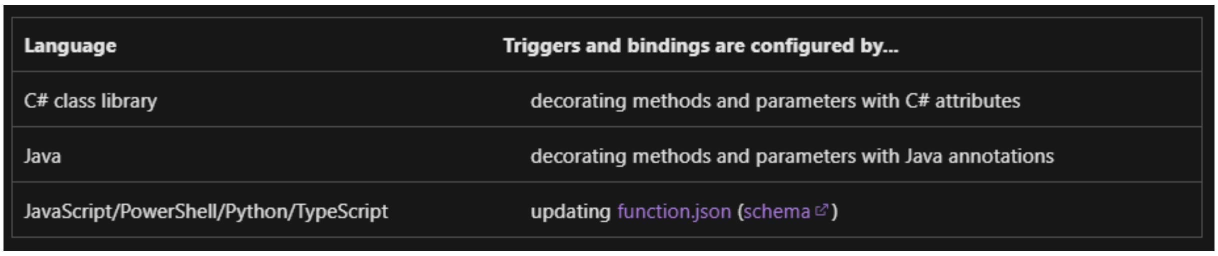
Triggers and bindings make it easy for you to connect to other services without needing to hardcode it. For example your function receives data from a queue in the function parameter and then send data back to a different queue via a output binding. See some examples below:



Example scenario	Trigger	Input binding	Output binding
A new queue message arrives which runs a function to write to another queue.	Queue*	*None*	Queue*
A scheduled job reads Blob Storage contents and creates a new Cosmos DB document.	Timer	Blob Storage	Cosmos DB
The Event Grid is used to read an image from Blob Storage and a document from Cosmos DB to send an email.	Event Grid	Blob Storage and Cosmos DB	SendGrid
A webhook that uses Microsoft Graph to update an Excel sheet.	HTTP	*None*	Microsoft Graph

Trigger and binding definitions.

Triggers and bindings are defined differently depending on what development language you use for your functions apps. See below:



Language	Triggers and bindings are configured by...
C# class library	decorating methods and parameters with C# attributes
Java	decorating methods and parameters with Java annotations
JavaScript/PowerShell/Python/TypeScript	updating function.json (schema)

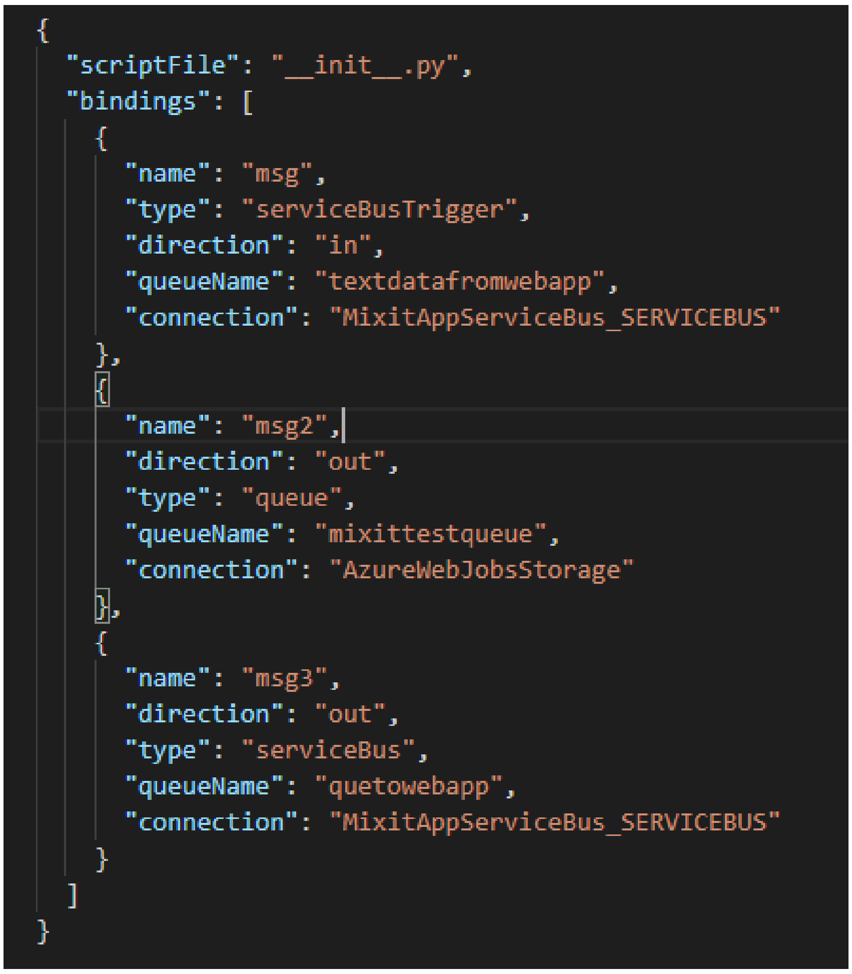
In .NET and Java, the parameter type defines the data type for input data. For instance, use string to bind to the text of a queue trigger, a byte array to read as binary, and a custom type to de-serialize to an object. Since .NET class library functions and Java functions don't rely on function.json for binding definitions, they can't be created and edited in the portal. C# portal editing is based on C# script, which uses function.json instead of attributes.

For example function app code for java will look like this

```
    @FunctionName("HttpTrigger-Java")
     public HttpResponseMessage run(
        @HttpTrigger(name = "req", methods = {HttpMethod.GET,
HttpMethod.POST}, authLevel = AuthorizationLevel.FUNCTION)
        HttpRequestMessage<Optional<String>> request,
        @QueueOutput(name = "msg", queueName = "outqueue", connection =
"AzureWebJobsStorage")
        OutputBinding<String> msg, final ExecutionContext context) {
    ...
}
```

Se here we can see that the @functionname (Trigger) and @QueueOutput (Output binding) are defined in the code instead of a json file.

An example of a function.json will be something like this:



```
{
  "scriptFile": "__init__.py",
  "bindings": [
    {
      "name": "msg",
      "type": "serviceBusTrigger",
      "direction": "in",
      "queueName": "textdatafromwebapp",
      "connection": "MixitAppServiceBus_SERVICEBUS"
    },
    {
      "name": "msg2",
      "direction": "out",
      "type": "queue",
      "queueName": "mixittestqueue",
      "connection": "AzureWebJobsStorage"
    },
    {
      "name": "msg3",
      "direction": "out",
      "type": "serviceBus",
      "queueName": "quetowebapp",
      "connection": "MixitAppServiceBus_SERVICEBUS"
    }
  ]
}
```

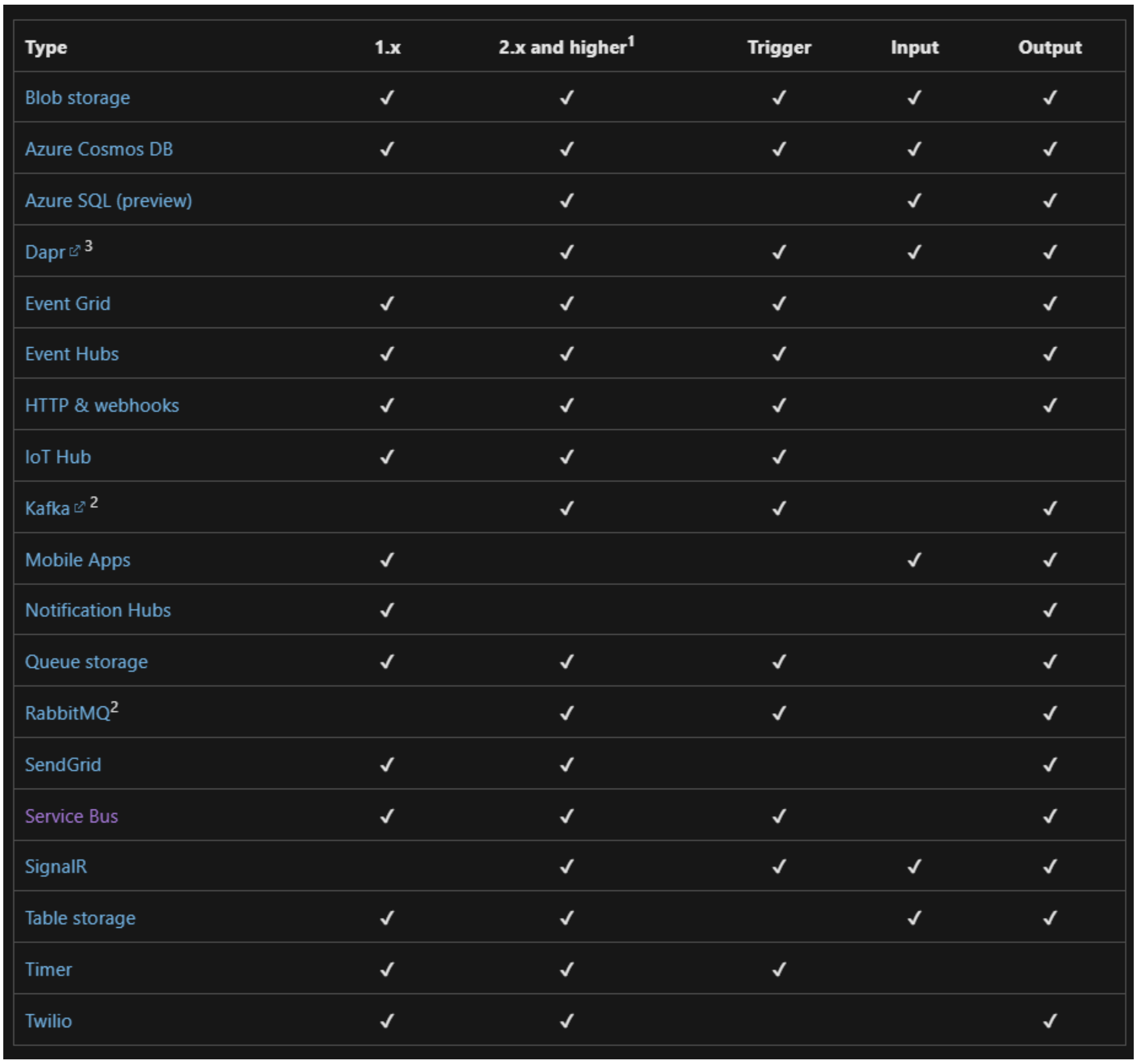
Here the tigger and bindigns are defined in the function.json that can then be called in the code by using the name of the trigger or binding.

Binding direction.

All triggers and bindings have a direction value when using the function.json file to define them.

- For triggers the direction is always **in**
- For In and output bindings it's **in** or **out**
- Some bindings support a third direction called **inout**.

List of supported binding from azure services.

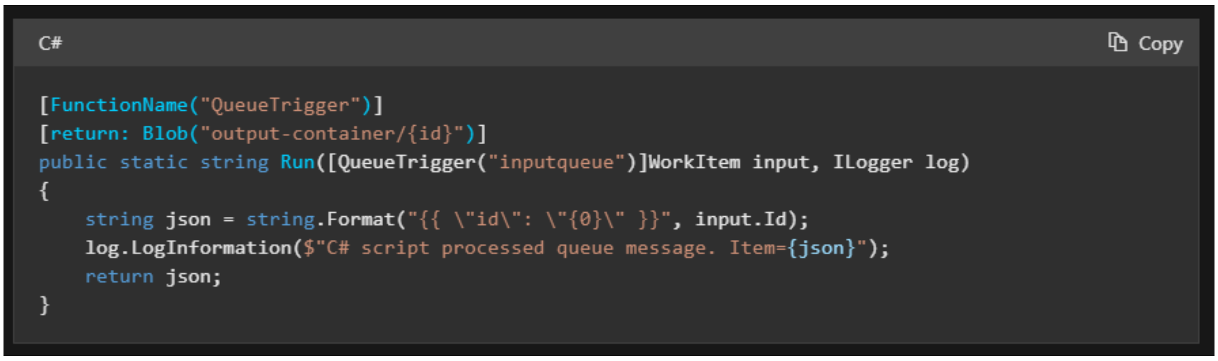


Type	1.x	2.x and higher[1]	Trigger	Input	Output
Blob storage	✓	✓	✓	✓	✓
Azure Cosmos DB	✓	✓	✓	✓	✓
Azure SQL (preview)		✓		✓	✓
Dapr[3]		✓	✓	✓	✓
Event Grid	✓	✓	✓		✓
Event Hubs	✓	✓	✓		✓
HTTP & webhooks	✓	✓	✓		✓
IoT Hub	✓	✓	✓		
Kafka[2]		✓	✓		✓
Mobile Apps	✓			✓	✓
Notification Hubs	✓				✓
Queue storage	✓	✓	✓		✓
RabbitMQ[2]		✓	✓		✓
SendGrid	✓	✓			✓
Service Bus	✓	✓	✓		✓
SignalR		✓	✓	✓	✓
Table storage	✓	✓		✓	✓
Timer	✓	✓	✓		
Twilio	✓	✓			✓

Azure function Return value.

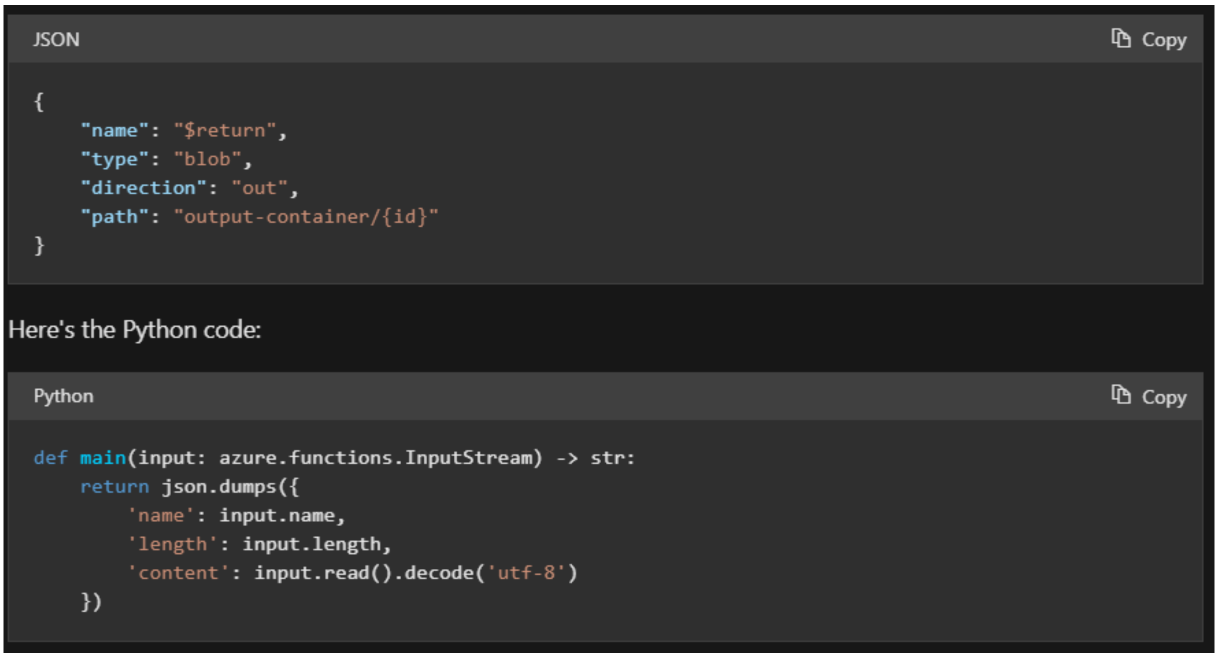
In languages that have a return value you can bind a function output binding to the return value.

C# example code:



```
[FunctionName("QueueTrigger")]
[return: Blob("output-container/{id}")]
public static string Run([QueueTrigger("inputqueue")]WorkItem input, ILogger log)
{
    string json = string.Format("{{ \"id\": \"{0}\" }}", input.Id);
    log.LogInformation($"C# script processed queue message. Item={json}");
    return json;
}
```

Python example:

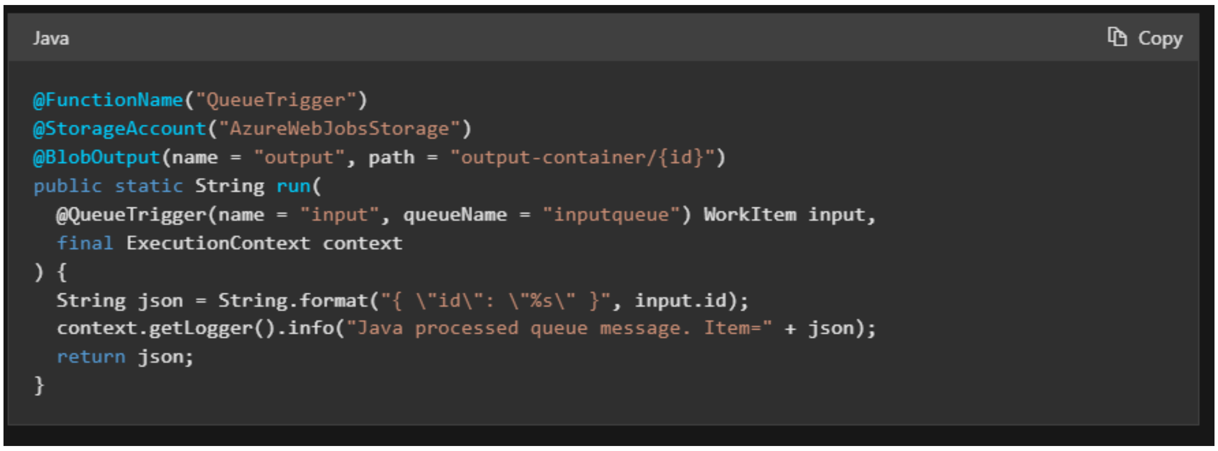


```
{
    "name": "$return",
    "type": "blob",
    "direction": "out",
    "path": "output-container/{id}"
}
```

Here's the Python code:

```
def main(input: azure.functions.InputStream) -> str:
    return json.dumps({
        'name': input.name,
        'length': input.length,
        'content': input.read().decode('utf-8')
    })
```

Java example code:



```
@FunctionName("QueueTrigger")
@StorageAccount("AzureWebJobsStorage")
@BlobOutput(name = "output", path = "output-container/{id}")
public static String run(
  @QueueTrigger(name = "input", queueName = "inputqueue") WorkItem input,
  final ExecutionContext context
) {
  String json = String.format("{ \"id\": \"%s\" }", input.id);
  context.getLogger().info("Java processed queue message. Item=" + json);
  return json;
}
```

Important knows for creating Azure functions.

In a hosted cloud environment, it's expected that VMs can occasionally restart or move, and system upgrades will occur. Your function app also likely depends on APIs, azure services and other databases which are also prone to periodic unreliability. So lets detail some best practices for designing and deploying efficient function apps that remain healthy.

Hosting plans.

When you create a azure function app you must choose a hosting plan for your app. The plan you choose has a direct effect on performance, reliability and of course costs. There are three basic hosting plans:

- Consumption plan
- Premium plan
- Dedicated (App service) plan

All plan are available when running either Linux or Windows.

The hosting plan you choose determines the following behaiviors:

- How your app function is scaled based on demand and how instances are allocated
- The resource available to each function app
- Support for advanced functionality

Its important that you choose the correct plan when you create your function app. Functions have very limited options when it comes to switching hosting plans.

Storage

Functions require a storage account be associated with your function app. The storage account connection is used by the functions host for operations such as managing triggers and logging function executions. It's also used for dynamically scaling function apps.

A misconfigured storage account in you function app can affect the performance and availability of your functions.

How we used azure functions in Sprint 3.

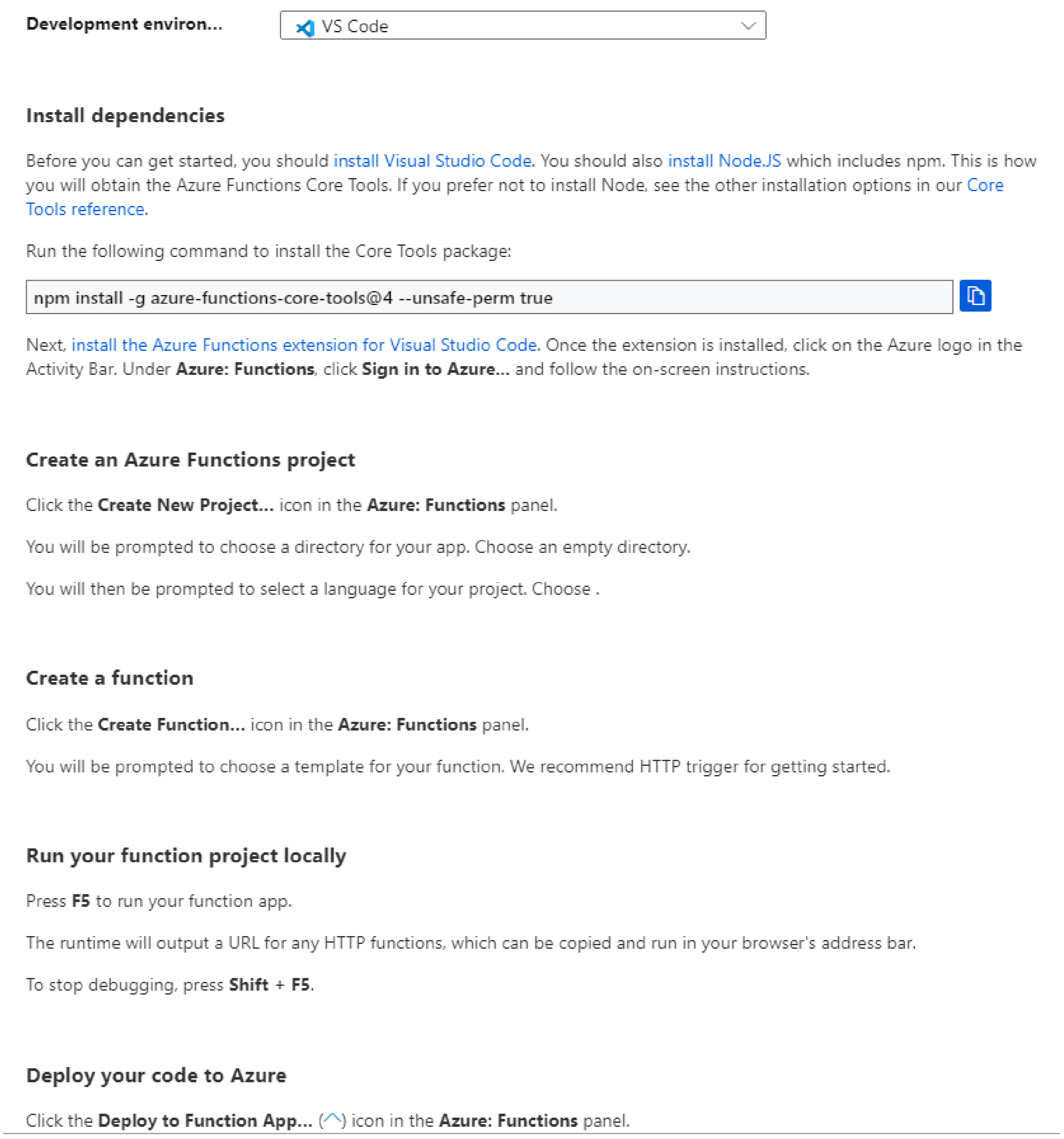
So for our project we are trying out using function apps to read and write informatie to service bus queues. Here I will show how we create the apps and what we ended up with in sprint 3.

Creating a function in Python and Visual studio code.

So we decided that we want to use python as our code language but with that comes some restrictions like only being able to use visual studio code to create and edit our function apps we cannot use the portal like with the other languages.

Creating a function.

So to create a function in VS code u need to install a couple of requirements you can see which ones when creating a function in the portal.



Development environ... VS Code

Install dependencies

Before you can get started, you should install Visual Studio Code. You should also install Node.JS which includes npm. This is how you will obtain the Azure Functions Core Tools. If you prefer not to install Node, see the other installation options in our Core Tools reference.

Run the following command to install the Core Tools package:

```
npm install -g azure-functions-core-tools@4 --unsafe-perm true
```

Next, install the Azure Functions extension for Visual Studio Code. Once the extension is installed, click on the Azure logo in the Activity Bar. Under **Azure: Functions**, click **Sign in to Azure...** and follow the on-screen instructions.

Create an Azure Functions project

Click the **Create New Project...** icon in the **Azure: Functions** panel.

You will be prompted to choose a directory for your app. Choose an empty directory.

You will then be prompted to select a language for your project. Choose .

Create a function

Click the **Create Function...** icon in the **Azure: Functions** panel.

You will be prompted to choose a template for your function. We recommend HTTP trigger for getting started.

Run your function project locally

Press **F5** to run your function app.

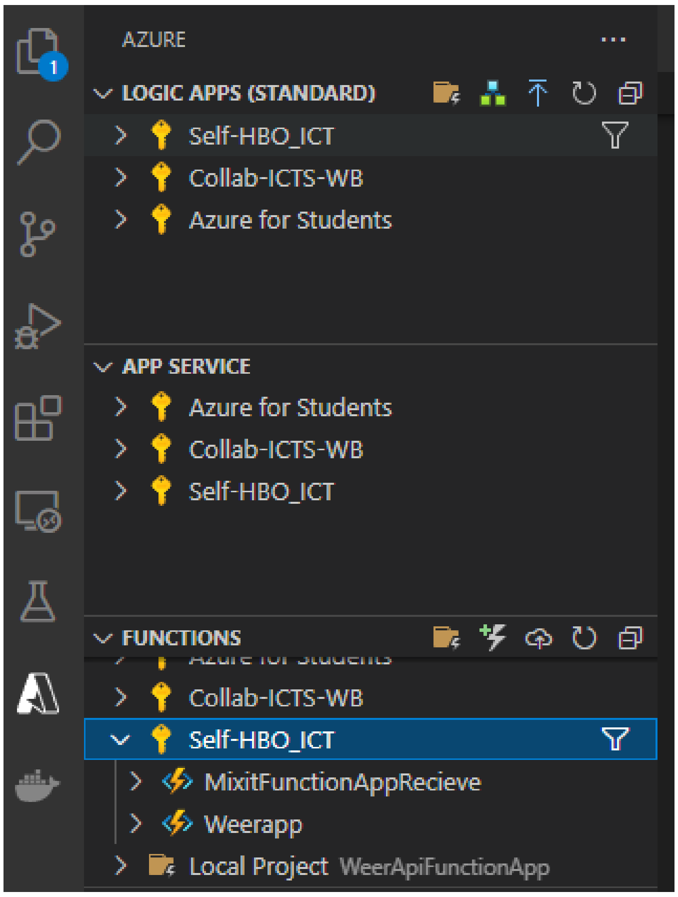
The runtime will output a URL for any HTTP functions, which can be copied and run in your browser's address bar.

To stop debugging, press **Shift** + **F5**.

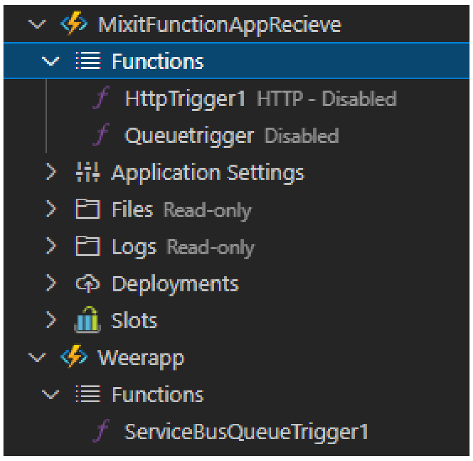
Deploy your code to Azure

Click the **Deploy to Function App...** (^) icon in the **Azure: Functions** panel.

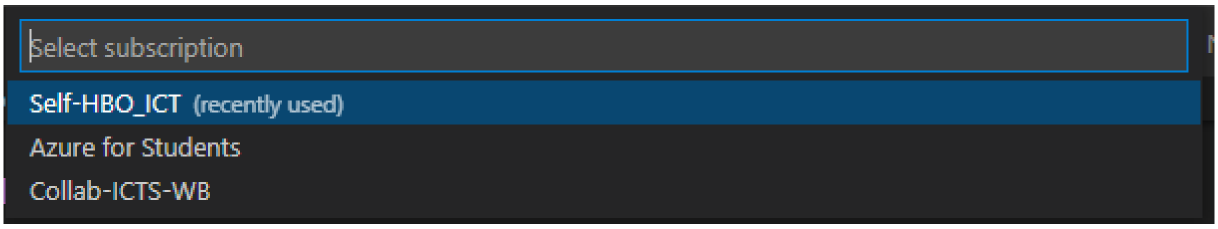
So when you have done all this u can now create a Function app project in VS code and create functions in that project that you can then deploy to a Function app on azure.



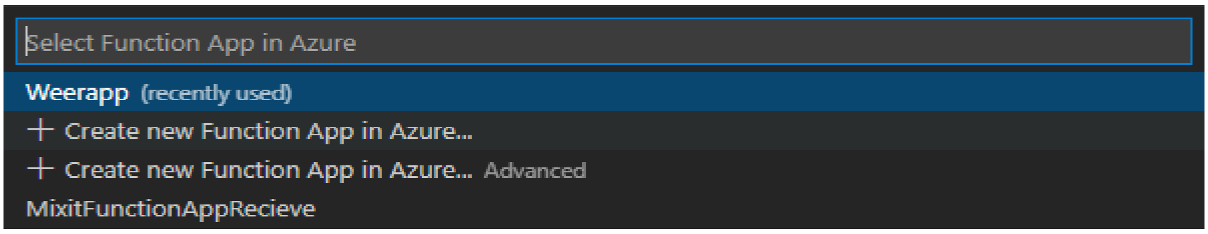
So when u have the azure extension in VS code you can now see you functions in VS code as you can see we have 2 functions apps in our subscription. Those Function apps can have multiple functions in them. As you can see we have 2 functions in the first one and another function in the second one.



Now we also have a local project and that is a function app that is located on your own pc in our case we save it on our git repository so that everyone has access to it. We edit our code in that local project and then deploy it to the cloud like so:



When we click on the middle cloud icon in the functions menu this will pop up at the top now it will ask you to what function app in the cloud you want to deploy your local project to.


Select Function App in Azure
Weerapp (recently used)
Create new Function App in Azure...
Create new Function App in Azure... Advanced
MixitFunctionAppRecieve

When we select our subscription we can now see the 2 function apps in the azure cloud that we can choose. These are the only 2 options you need to choose from and then it will deploy your local app to the cloud.

First attempt.

So our first attempt at making a fuction app that will read a service bus queue and send a edited message back to another queue was a failure because of a number of error that came with the raw python code we wanted to use. Our idea was simple we already have working python code that makes a connection with a service bus and send a message to a specified queue so lets just copy and paste that and see if it works. Example below:

Python code we use in webapp:

```
def send_single_message_to_weatherfunction_que(UserText):
    with
ServiceBusClient.from_connection_string(CONNECTION_STR_WeatherAPISendQue) as
client:
        with client.get_queue_sender(QUEUE_NAME_WeatherAPISendQue) as sender:
            single_message = ServiceBusMessage(UserText)
            sender.send_messages(single_message)
```

So here we can see a method we use in our webapp to send a message to a queue and we tought since that works we just copy paste that to the function app and done.

Our first function:

```
import logging
import azure.functions as func

CONNECTION_STR =
"Endpoint=sb://mixitappservicebus.servicebus.windows.net/;SharedAccessKeyName=
WebappSendToQue;SharedAccessKey=JJK3zWqFvc1PAaPKcrjyefEVLGrgSHz7ZEIBt4gLwVM=;E
ntityPath=fromwebappwheatherdata"
QUEUE_NAME = "someQueue"

def main(msg: func.ServiceBusMessage):
    logging.info('Python ServiceBus queue trigger processed message: %s',
                 msg.get_body().decode('utf-8'))

    with func.ServiceBusClient.from
         _connection_string(CONNECTION_STR) as client:
        with client.get_queue_sender(QUEUE_NAME) as sender:
            sender.send_messages("Dit is een toevoeging" + msg)
```

This is what our first function app kind of looked like as you can see we adapted the webapp code into the function app but unfortunately it didn’t and we got some errors that I cant remember but I think it had to do with we were missing an servicebus import and just in general this is not the correct way of doing it or at least if not efficient.

Second try.

So we went looking for solution and how to send messages to other services from a function app and at some point we ran into bindings. Now I already went over binding in a previous topic so I wont explain here but in this attempt we decided to try out bindings to send messages to a queue (storage and servicebus)

Function.json:

```json
{
  "scriptFile": "__init__.py",
  "bindings": [
    {
      "name": "msg",
      "type": "serviceBusTrigger",
      "direction": "in",
      "queueName": "textdatafromwebapp",
      "connection": "MixitAppServiceBus_SERVICEBUS"
    },
    {
      "name": "msg2",
      "direction": "out",
      "type": "queue",
      "queueName": "mixittestqueue",
      "connection": "AzureWebJobsStorage"
    },
    {
      "name": "msg3",
      "direction": "out",
      "type": "serviceBus",
      "queueName": "quetowebapp",
      "connection": "MixitAppServiceBus_SERVICEBUS"
    }
  ]
}
```

So here is the function.json we created for this attempt as you can see we have 1 trigger and 2 output bindings for a type queue (storage account) and a type serviceBus with the necessary information to connect and send it to the right queue.

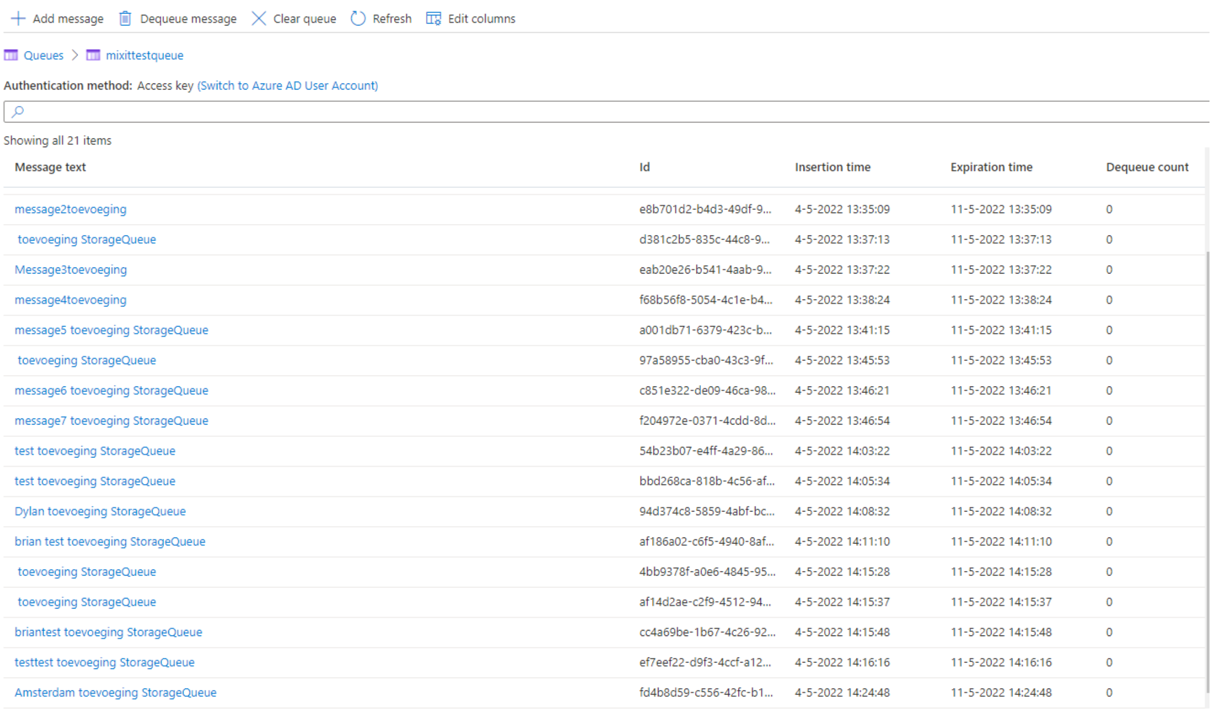
Init.py:

```python
import logging
import azure.functions as func

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')
```

So in our init.py is our code and as explained in the bindings topic you declare your bindings as parameters in the code like we have done here after that we take our input trigger called **msg** and first do the logging.info just to see what message we have as our input after that we create **msg2** and **msg3** by taking **msg** and adding some stuff to it at the end. Now this worked first try we didn't run into any errors and we knew it worked because we now had messages in a servicebus queue and in the storage account queue.

Storage account queue:



Authentication method: Access key (Switch to Azure AD User Account)

Showing all 21 items

Message text	Id	Insertion time	Expiration time	Dequeue count
message2toevoeging	e8b701d2-b4d3-49df-9...	4-5-2022 13:35:09	11-5-2022 13:35:09	0
toevoeging StorageQueue	d381c2b5-835c-44c8-9...	4-5-2022 13:37:13	11-5-2022 13:37:13	0
Message3toevoeging	eab20e26-b541-4aab-9...	4-5-2022 13:37:22	11-5-2022 13:37:22	0
message4toevoeging	f68b56f8-5054-4c1e-b4...	4-5-2022 13:38:24	11-5-2022 13:38:24	0
message5 toevoeging StorageQueue	a001db71-6379-423c-b...	4-5-2022 13:41:15	11-5-2022 13:41:15	0
toevoeging StorageQueue	97a58955-cba0-43c3-9f...	4-5-2022 13:45:53	11-5-2022 13:45:53	0
message6 toevoeging StorageQueue	c851e322-de09-46ca-98...	4-5-2022 13:46:21	11-5-2022 13:46:21	0
message7 toevoeging StorageQueue	f204972e-0371-4cdd-8d...	4-5-2022 13:46:54	11-5-2022 13:46:54	0
test toevoeging StorageQueue	54b23b07-e4ff-4a29-86...	4-5-2022 14:03:22	11-5-2022 14:03:22	0
test toevoeging StorageQueue	bbd268ca-818b-4c56-af...	4-5-2022 14:05:34	11-5-2022 14:05:34	0
Dylan toevoeging StorageQueue	94d374c8-5859-4abf-bc...	4-5-2022 14:08:32	11-5-2022 14:08:32	0
brian test toevoeging StorageQueue	af186a02-c6f5-4940-8af...	4-5-2022 14:11:10	11-5-2022 14:11:10	0
toevoeging StorageQueue	4bb9378f-a0e6-4845-95...	4-5-2022 14:15:28	11-5-2022 14:15:28	0
toevoeging StorageQueue	af14d2ae-c2f9-4512-94...	4-5-2022 14:15:37	11-5-2022 14:15:37	0
briantest toevoeging StorageQueue	cc4a69be-1b67-4c26-92...	4-5-2022 14:15:48	11-5-2022 14:15:48	0
testtest toevoeging StorageQueue	ef7eef22-d9f3-4ccf-a12...	4-5-2022 14:16:16	11-5-2022 14:16:16	0
Amsterdam toevoeging StorageQueue	fd4b8d59-c556-42fc-b1...	4-5-2022 14:24:48	11-5-2022 14:24:48	0

So now that we have a working prototype if you will it was time to try and get some information form a API and send it back to a queue that the webapp can receive messages from.

Third attempt with weather API.

So for our third attempt we decided to integrate a weather API call to our function app instead of using Outlook because it was just easier to implement for testing purposes and checking if it would work in the first place.

Weather API.

So to call this weather API you need to send a request by using a URL with the name of the location you want a weather update from. Example URL:

```
https://api.openweathermap.org/data/2.5/weather?q=Amsterdam&appid=a6c60d139555
4bbac92f34b7ebf8122d
```

So here we can see the URL with the location Amsterdam.

Function.json

```
{
  "scriptFile": "__init__.py",
  "bindings": [
    {
      "name": "inputRequest",
      "type": "serviceBusTrigger",
      "direction": "in",
      "queueName": "fromwebappwheatherdata",
      "connection": "MixitAppServiceBus_SERVICEBUS"
    },
    {
      "name": "outputWeatherAPI",
      "direction": "out",
      "type": "serviceBus",
      "queueName": "fromweerapptowebapp",
      "connection": "MixitAppServiceBus_SERVICEBUS"
    }
  ]
}
```

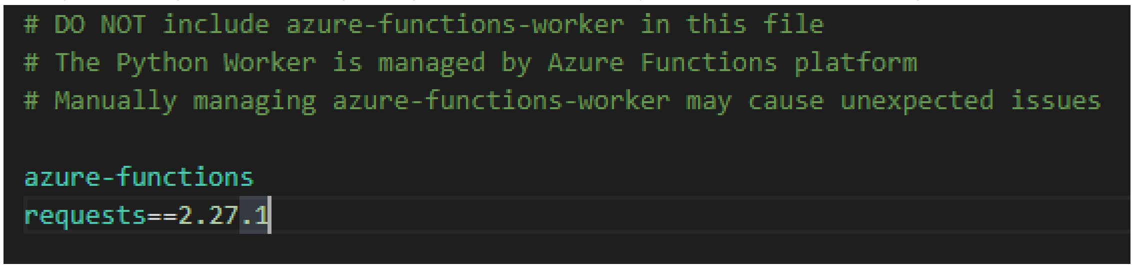
So the function.json is very simple since we only need 1 trigger that gives us the staring input from the first queue this input is the location name we want a weather update for. And the output binding is used to send the response we get from the API back to a second queue that will be received by the webapp.

Init.py

```python
from http.client import responses
import logging
import requests
import json
import azure.functions as func

#The main method that runs when app is triggerd
#The inputRequest is the trigger input
#The outputWeatherAPI is the output these are defined in the function.json
def main(inputRequest: func.ServiceBusMessage,
         outputWeatherAPI: func.Out[str]):
    # Log the input so we can see what it gets from the queue
    logging.info('Python ServiceBus queue trigger processed message: %s',
                 inputRequest.get_body().decode('utf-8'))
    #Create the request URL for the weather API with the input from the queue
    message =  inputRequest.get_body().decode('utf-8')
    api_url = "https://api.openweathermap.org/data/2.5/weather?q=" + message +
"&appid=a6c60d1395554bbac92f34b7ebf8122d"
    #Request data from weather API with URL
    response = requests.get(api_url)
    #Log the response of the Weather API so we can see what it sends back
    logging.info(response.json())
    #Convert json to a string so that we can send it back to the second queue
    data = json.loads(response.text)
    jsonToString = json.dumps(data)
    #Set the value of output to the json to send back
    outputWeatherAPI.set(jsonToString)
```

Now for this a couple of interesting things we had to deal with were the imports when we first deployed it to the Cloud we got error of azure not knowing what a request was because it didn't know what to do with the imports and we found out that every function app has a requirments.txt that you can update with imports you want to use in your function code. Requirement.txt:



```
# DO NOT include azure-functions-worker in this file
# The Python Worker is managed by Azure Functions platform
# Manually managing azure-functions-worker may cause unexpected issues

azure-functions
requests==2.27.1
```

So as you can see here the default would be azure-functions since that one is always used when creating a azure function but we had to add requests so that azure knew it had to import that one aswell.

After doing this the request worked and we got something back from the weather API but we ran into another problem because what we got back is a JSON with all the information the API can provide but our output expected a string to send back. In order to do this we use json.dumps this metod will transform a provide json into a string and we then set out output to that string to send back and this worked.