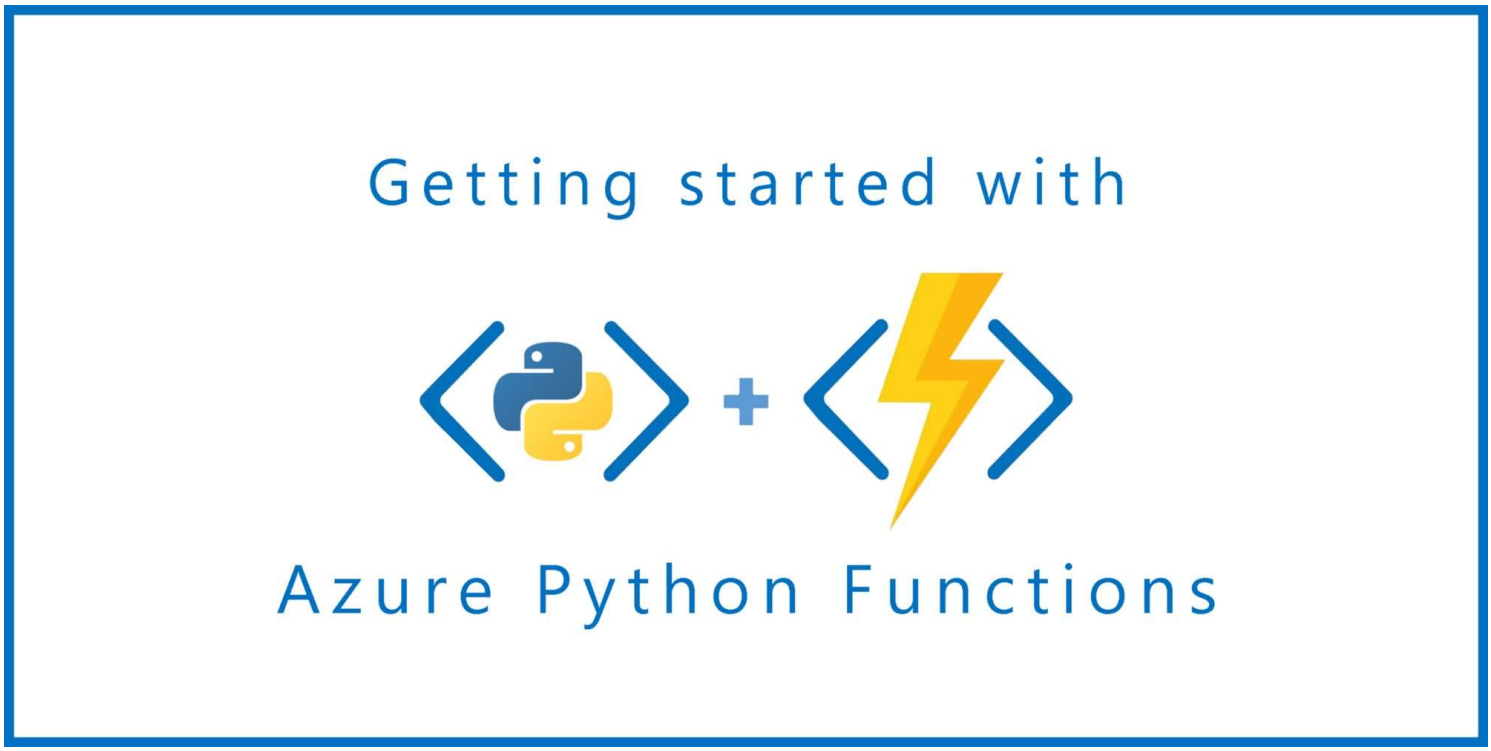
Getting started with Azure Python functions

Katia Gil Guzman Follow
Aug 7, 2019 · 10 min read ★



What are Azure functions?

Functions allow to run **serverless code**, code that you run only when you need it and that doesn't need a whole infrastructure to live in. You can decide that a function is triggered when hitting an HTTP endpoint, or when new files are added into a blob storage for instance. It might always perform the same task, or it might do something different depending on your inputs. Basically, you can write code that does (almost) anything you can think of and host it as an Azure Function.

For more information on Azure functions, take a look at the documentation.

. . .

> *In this tutorial, we'll learn how to create an HTTP triggered function and receive and return data. To illustrate the use of external libraries and show how the code can be segmented, we'll create a function that performs text processing, and we'll finish by publishing it to Azure.*

1. Create a new function app

The documentation explains well how to get started quickly with Azure Python Functions, but if you prefer to stay here with me (I'm honored), **here are the steps to setting up an Azure Python Functions project:**

- Install Python 3.6
- Install Azure Functions Core Tools:

```
npm install -g azure-functions-core-tools
```

- Install the Azure CLI version 2.x or a later version.

If you don't already have an Azure subscription, you can sign up for a free trial here.

Note that only Python 3.6 is supported for now, so make sure you have this version installed (it won't work with Python 3.7).

. . .

Next step is creating a new virtual environment. Even if it's not mandatory, you

To create a virtual environment, navigate to where you want to set up your project and use this Bash command:

```
python3.6 -m venv .venv
source .venv/bin/activate
```

Or this Powershell command on Windows:

```
py -3.6 -m venv .venv
.venv\scripts\activate
```

If you're on Windows and you can't activate your virtual environment (you get the error "The file is not digitally signed. You cannot run this script on the current system"), set your Execution Policy to Remote Signed to be able to run the script:

Open a Powershell command line in admin mode and run

```
Set-ExecutionPolicy RemoteSigned
```

You should now be able to activate your virtual environment, and you'll see that (.env) precedes every line in your command prompt.

Inside your virtual environment, **create a new Functions project by running**:

```
func init FunctionProjTest
```

Then select Python as worker runtime.

Functions app can contain several functions (up to 200 for the regular consumption plan, 20 for premium and app service plans). But remember all of them share the same environment, so if you want to have several functions that don't share anything and use completely different libraries, it might make more sense to create several Functions projects.

To **create a new function**, navigate inside the Functions project folder and run

```
func new
```

Select the HTTP trigger template as it's the type of functions that we will cover in this tutorial, and follow the prompts to name your function. If you have no particular interest in being original you can use the same name as me: *func_test*

And… Congrats! You have now created a new Azure function, and you can already test what the HTTP trigger template does.

Run it locally using the command

```
func host start
```

and make a GET request by either navigating to the corresponding URL in your browser or using Postman.

The URL is provided in the command line when you start your function, and it looks like this: http://localhost:7071/api/func_test

The template is designed to say 'Hello' to a given name, so to test it properly, append a query parameter, like this: http://localhost:7071/api/func_test?name=Katia

2. Receive and return data

Let's take a look at *__init__.py* (inside your function folder), which was generated automatically as part of the HTTP Trigger template. This file contains the main method of our function, which is executed when the function is triggered.

It looks like this:

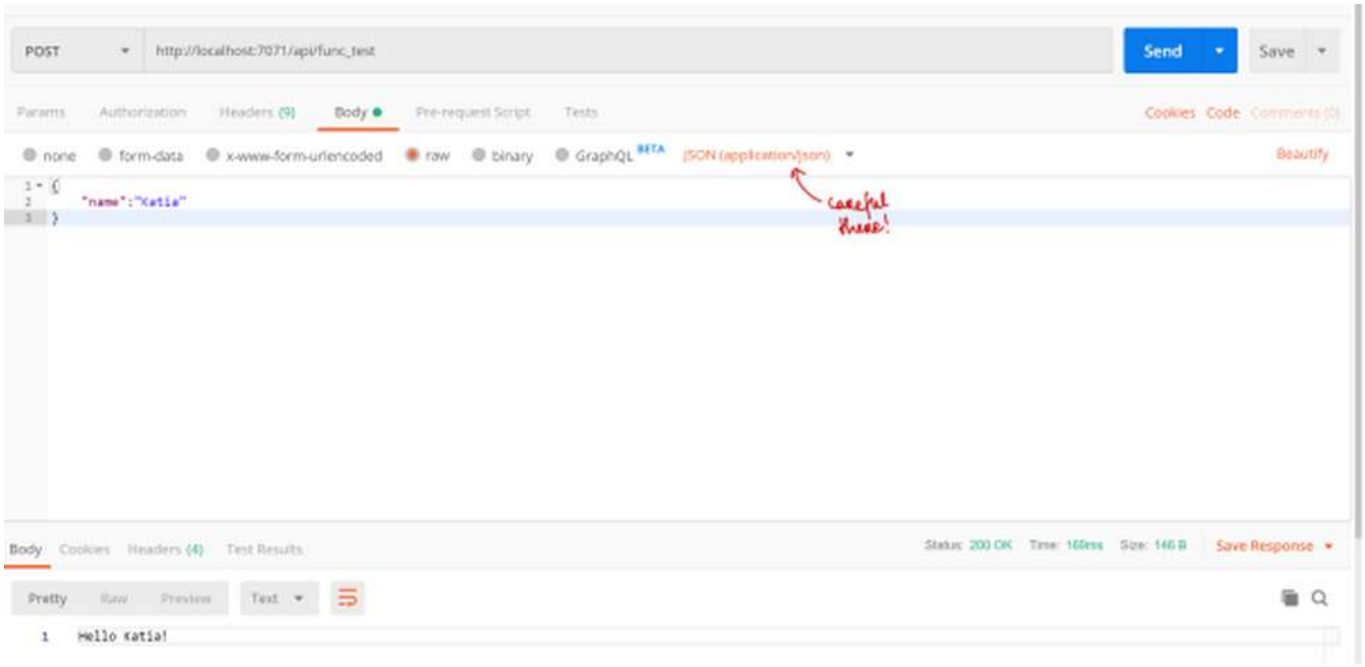
```python
import logging

import azure.functions as func


def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    name = req.params.get('name')
    if not name:
        try:
            req_body = req.get_json()
        except ValueError:
            pass
        else:
            name = req_body.get('name')

    if name:
        return func.HttpResponse(f"Hello {name}!")
    else:
        return func.HttpResponse(
             "Please pass a name on the query string or in the request body",
             status_code=400
        )
```

__init__.py hosted with ♡ by GitHub view raw

As you can see, it tries to get the name from the query parameters, and if it doesn't find it, tries to get it from the request body. Note that it's possible to use req_body['name'] instead of req_body.get('name').

This means that we can also test sending the function a POST request

```
{
"name": "You can use a fake name if you don't like yours"
}
```



As expected, we get back the same response as what we got with the GET request.

. . .

So with all that **we've seen how to get a request parameters and body**, which is already pretty cool.

But let's go a bit further:

What happens if you want to **send images or other files to your function**?

Well instead of using *req.get_json()* you can use *req.get_body()*.

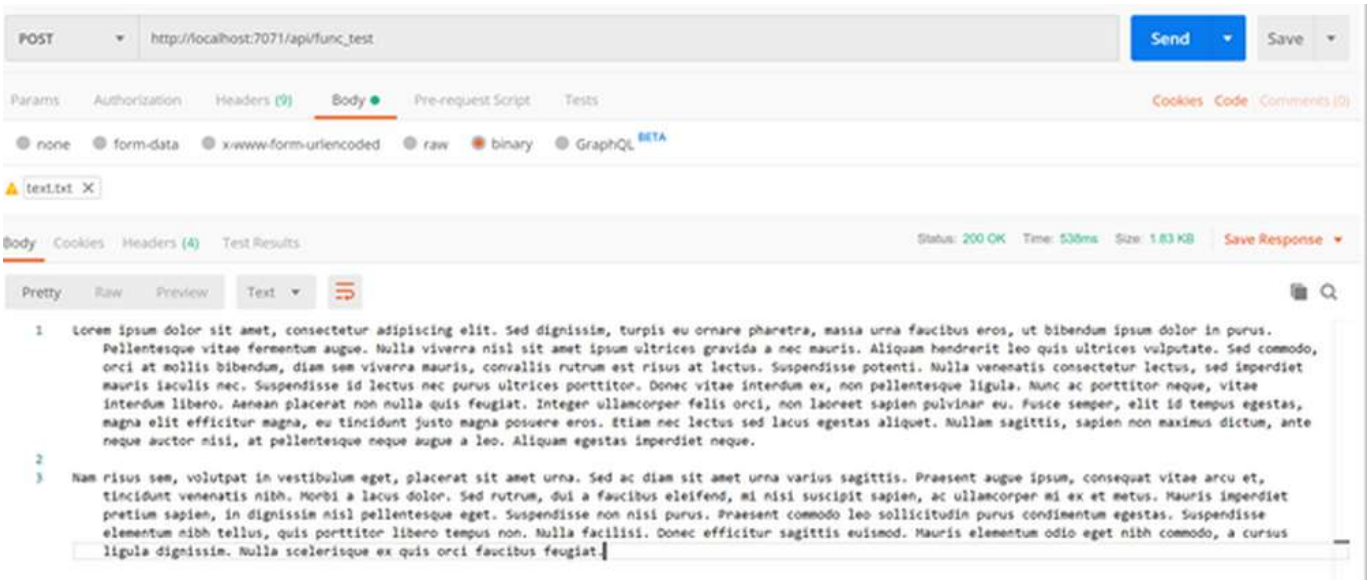
Let's try:

```
3
4       file_sent = None
5
6       try:
7           file_sent = req.get_body()
8       except ValueError:
9           pass
10
11      if file_sent:
12          return func.HttpResponse(file_sent)
13      else:
14          return func.HttpResponse(
15               "Please pass a file in the request body",
16               status_code=400
17          )
```

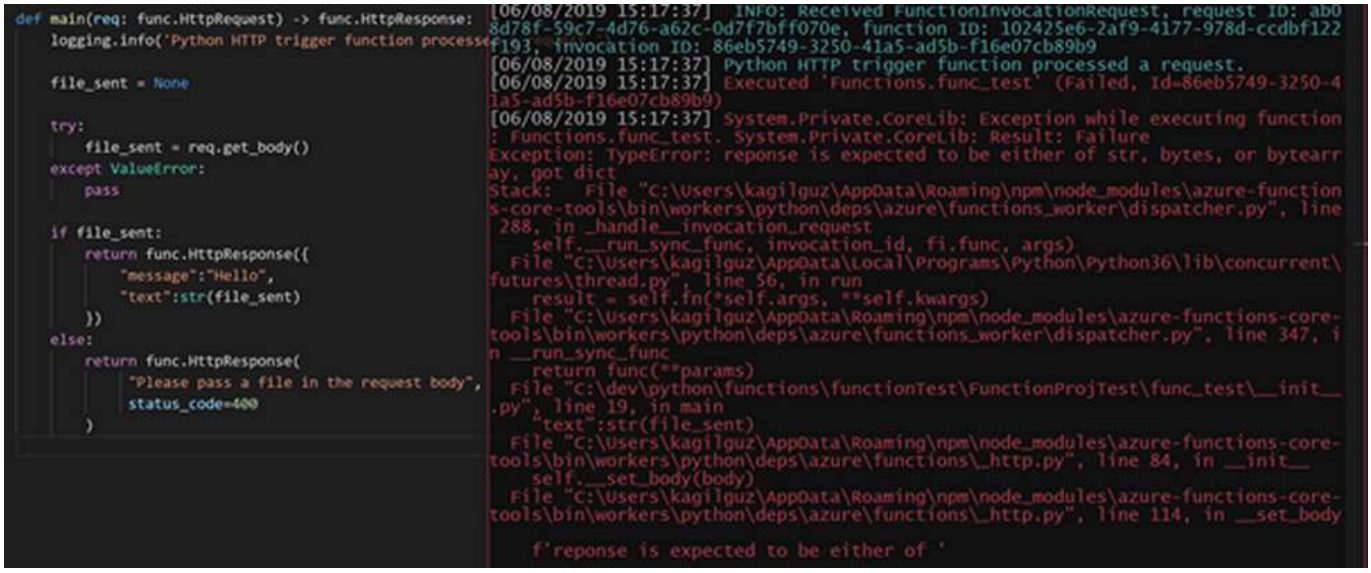
__init__.py hosted with ♡ by GitHub view raw

Note that when you save the code after changes have been made, the function restarts automatically.

Now create a new txt file with whatever you want in it (if you're not inspired click here) and make a new POST request to your function with the txt file as request body. What you should get back is the text contained in your file:



If you try and just pass json as the HTTP response, you'll find yourself with an annoying error:



The right way to do it is using *json.dumps*, so go ahead and impo ng of your file before updating your code:

```
1  import logging
2  import json
3  import azure.functions as func
4
5  def main(req: func.HttpRequest) -> func.HttpResponse:
6      logging.info('Python HTTP trigger function processed a request.')
7
8      file_sent = None
9
10     try:
11         file_sent = req.get_body()
12     except ValueError:
13         pass
14
15     if file_sent:
16         return func.HttpResponse(
17             json.dumps([ {
18             "text": str(file_sent),
```

```
    else:
        return func.HttpResponse(
             "Please pass a file in the request body",
             status_code=400
        )
```

__init__.py hosted with ♡ by GitHub view raw

Now that we know how to receive and return data with Python functions, let's dive into the most interesting part: **develop anything you want!**

3. Segment your code & Use external libraries

If your code requires using external libraries, you can **install them the same way you would with a regular project**: just run

```
pip install NAME_OF_LIBRARY
```

at the root of your Functions project (not in the folder of the function itself).

For the sake of this tutorial, we're going to install two text/natural language processing libraries: nltk and spaCy, along with a spaCy module.

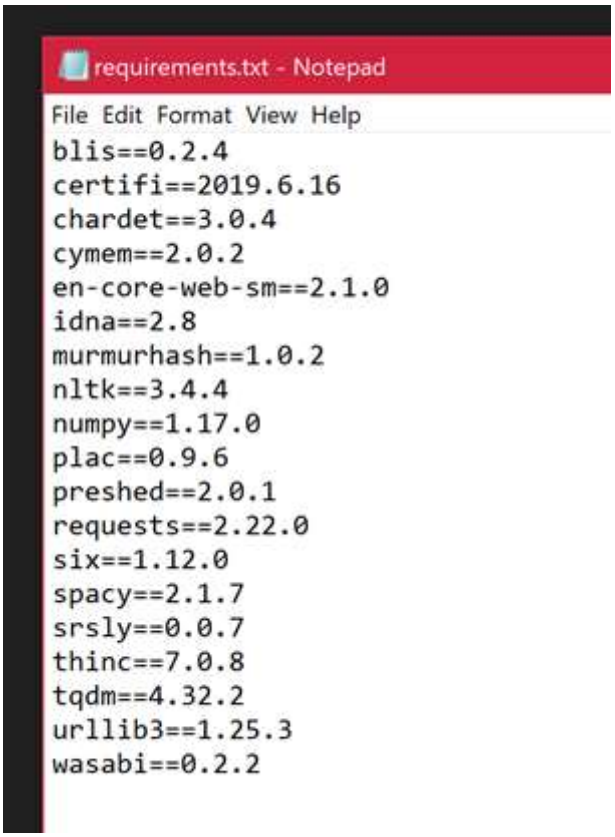
Go ahead and run (after stopping the execution of your function project):

```
pip install nltk
pip install spacy
python -m spacy download en_core_web_sm
```

Once you've done that, run

```
pip freeze > requirements.txt
```

If you open it, you'll see that something's wrong:



Guessed it? The spaCy module wasn't recorded properly (there's a *en-core-web-sm* something in there but that's not what we want) — you'll need to manually add the module to your requirements:

Just delete the *en-core-web-sm==2.1.0* line and put this URL at the end of *requirements.txt*:

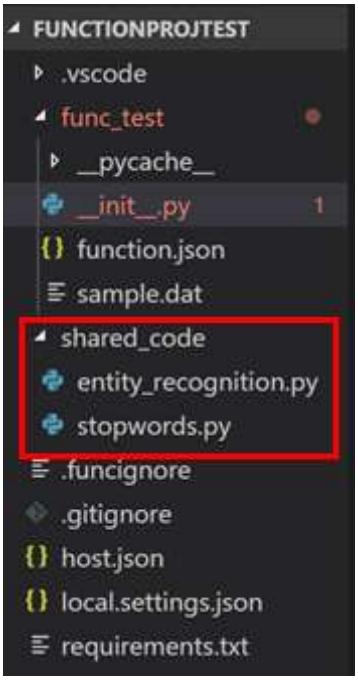
https://github.com/explosion/spacy-models/releases/download/en_core_web_sm-2.1.0/en_core_web_sm-2.1.0.tar.gz

The same goes for other libraries you may want to use that do not have a "regular" installation.

• • •

use SpaCy to find named entities in a text.

Now your project tree should look like this:



I created a new folder at the root of the Functions project so that it could be used by several functions inside the project (even though here I only have one), but you can also create the files and folders you want inside a function folder, it's up to you really.

In *__init.py__*, import the files you just created like this:

```
from ..shared_code import entity_recognition
from ..shared_code import stopwords
```

If your files are in the same folder, use this instead:

```
from . import entity_recognition
from . import stopwords
```

```
nltk.download("stopwords")
from nltk.corpus import stopwords

def remove_stop_words(text):
    STOPLIST = stopwords.words('english')
    text = ' '.join([word for word in text.split() if word not in STOPLIST])
    return text
```

stopwords.py hosted with ♡ by GitHub view raw

And this in *entity_recognition.py* to return a text containing named entities recognized in an input text:

```
import spacy
import en_core_web_sm

def get_entities(text):

    nlp = en_core_web_sm.load()
    doc = nlp(text)
    entities = ""

    for ent in doc.ents:
        entity = "/ %s - %s /" % (ent.text,ent.label_)
        entities = entities + entity

    return entities
```

entity_recognition.py hosted with ♡ by GitHub view raw

Now all we need to do is update *__init.py__* to call our new modules:

```
import logging
import json
import azure.functions as func
from ..shared_code import entity_recognition
from ..shared_code import stopwords

```

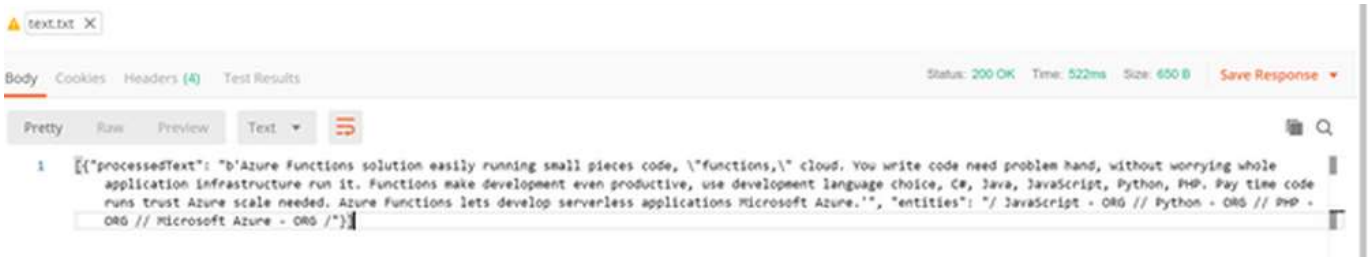
```
12
13      try:
14          file_sent = req.get_body()
15      except ValueError:
16          pass
17      else:
18          text = str(file_sent)
19
20      processed_text = stopwords.remove_stop_words(text)
21      entities = entity_recognition.get_entities(text)
22
23      if file_sent:
24          return func.HttpResponse(
25              json.dumps([ {
26              "processedText": processed_text,
27              "entities": entities
28              } ]),
29              status_code=200
30          )
31      else:
32          return func.HttpResponse(
33               "Please pass a file in the request body",
34               status_code=400
35          )
```

__init__.py hosted with ♡ by GitHub view raw

Let's test it (remember to run *func host start* first) — this is the text I used:

Azure Functions is a solution for easily running small pieces of code, or "functions," in the cloud. You can write just the code you need for the problem at hand, without worrying about a whole application or the infrastructure to run it. Functions can make development even more productive, and you can use your development language of choice, such as C#, Java, JavaScript, Python, or PHP. Pay only for the time your code runs and trust Azure to scale as needed. Azure Functions lets you develop serverless applications on Microsoft Azure.

And here's the result :



As you can see, the stopwords have been removed and our entity recognition code picked up a few named entities.

. . .

Now that you know how to do this — **you can do pretty much anything you would normally do in Python** 😉

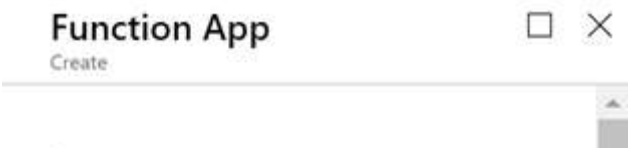
Perfect, so **your function is now ready and running locally.** That's great if you want to keep it to yourself. **But if you want others to be able to access it, you'll want to publish it.** Which is — as you might expect — what we'll learn below.

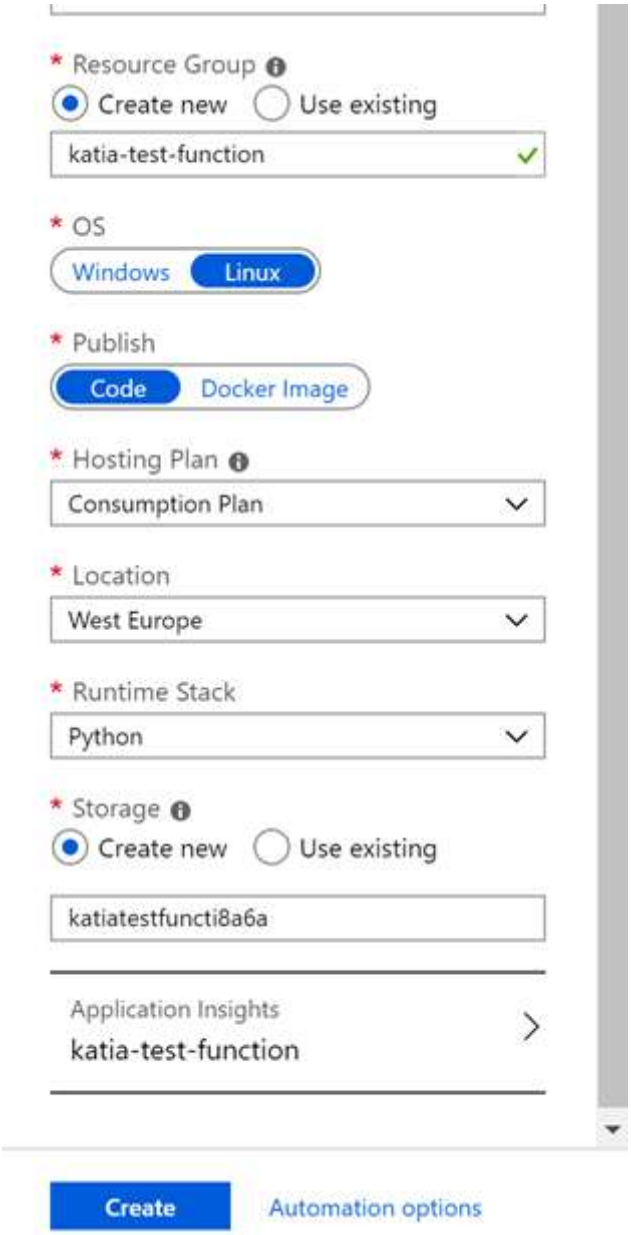
4. Publish

Publishing your Functions project means an image of your virtual environment will be created and deployed to a Function App in Azure, and since recently the build is done on the server side, so you don't need to install anything.

There are 2 ways to create a Function App: you can either use the Azure CLI and do it from your command line, or do it from the portal. I'll show how to do it from the portal since the documentation covers the former.

From the Azure portal, click on *Create a Resource* and create a new Function App:





Choose a unique name, select Linux as OS, choose to publish Code, and select Python as the Runtime Stack. For the Hosting Plan and Location it's up to you but if you want your Function App to be cheap, then go for the Consumption Plan.

• • •

Once it's deployed, go back to your command prompt (still in the Functions project folder, and stop the execution of your Functions project). Make sure you're logged into

```
az login
```

and set your subscription with

```
az account set --subscription "SUBSCRIPTION_ID"
```

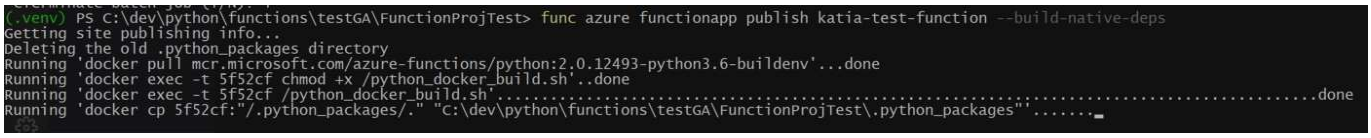
Now you can either do a local or remote build, but depending on binary dependencies needed in your project, the remote build might fail. In this tutorial, we use a spaCy module which has a different behavior so we'll use the local build approach.

Install Docker and make sure it is running, then run

```
func azure functionapp publish APP_NAME --build-native-deps
```

(APP_NAME being the name of the Function App you just created on Azure)

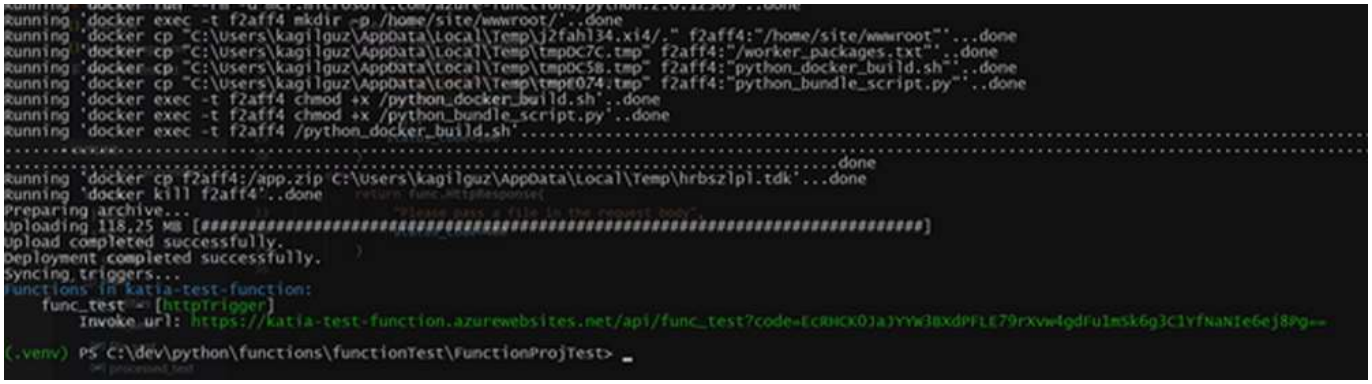
You should see something like this (don't be surprised if it takes a while):



If it fails for no apparent reason, increase the memory allocated to Docker (Right click on Docker icon > Settings > Advanced).

If for other projects you only use standard libraries, you can build remotely by running:

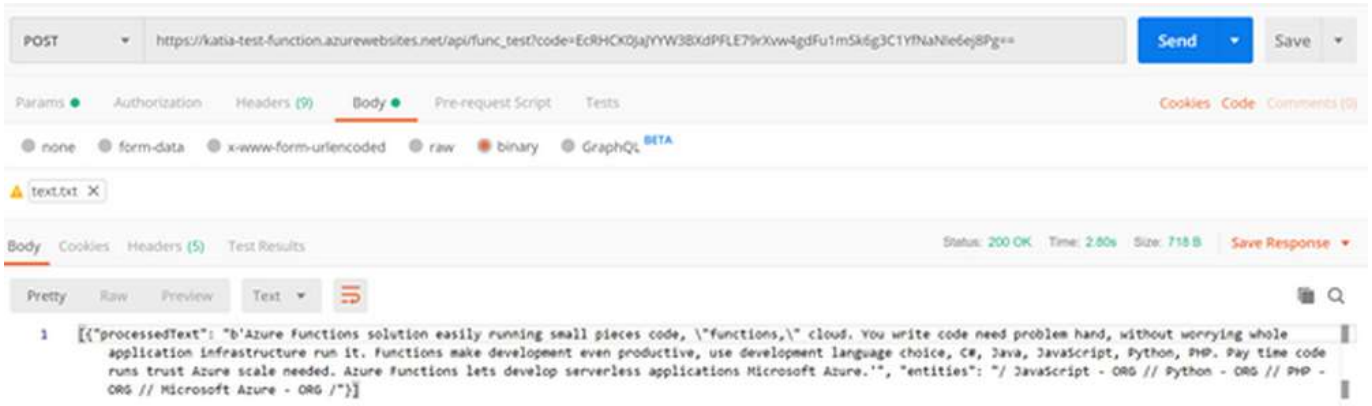
```
func azure functionapp publish APP_NAME --build remote
```



We are (at least I am — if it didn't work for you, read through the steps again and make sure you followed them properly) almost done!

. . .

To make sure everything works as expected, open Postman again and just replace the endpoint with the invoke URL provided after deployment (with the code parameter, otherwise you'll get an *Unauthorized Access* error).



Yay! We're now ready to use this Python function anywhere!

Here's an example of how to use this in a JS web app:

```
1       fetch(functionURL, {
2         method: 'POST',
```

```
          "filename":filename
        })
      })
      .then(res => res.json())
      .then((data) => {
         console.log(data);
      })
      .catch((error) => {
          console.error(error);
      });
```

fetch_function.js hosted with ♡ by GitHub view raw

Hope you find Azure functions as useful as I do 😉

Resources

Azure Functions Introduction

Create an HTTP Triggered Azure Python Function Quickstart

Finished project on GitHub

Python Serverless Functions As A Service Azure Functions Python Functions