Docker Container on Azure Functions with Python

#azure #python #docker #serverless

Omer Farooq Ahmed 13 Oct 2020 · 6 min read

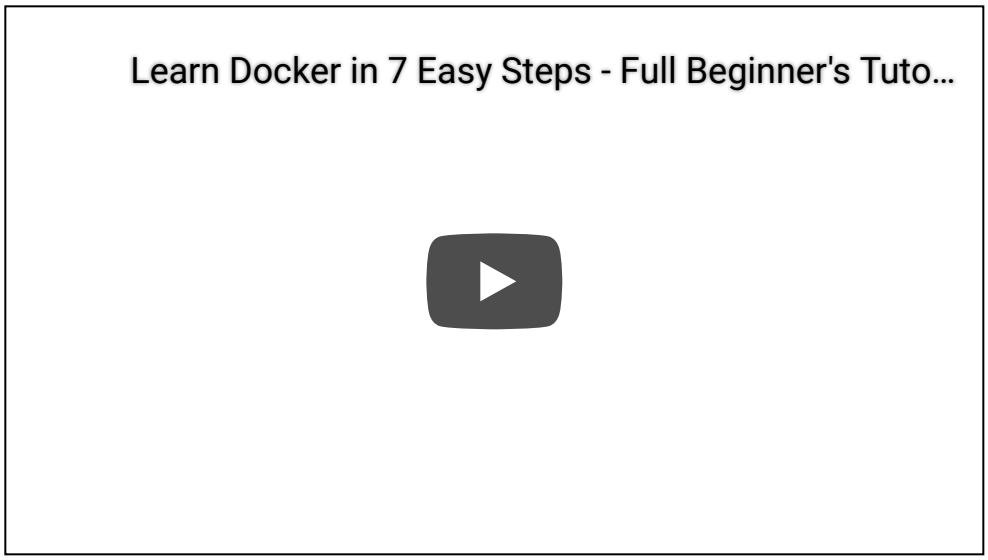
Introduction

Serverless Computing, also known as Serverless Architecture, Functions as a Service (FaaS) or just Serverless, is all the rage these days. For any developer looking to quickly deploy their code to the cloud without having to worry about managing server resources or getting charged insane amounts for running a hello world application, services like AWS Lambda, Google Cloud Functions or Azure Functions is the solution. With a host of event triggers, and rich CLI tooling to create boilerplate code and deploy straight to the cloud, all three cloud powerhouses provide a great service to use. However, serverless can be a double edged sword. While providing convenience, it also restricts setting up custom environments on the machine the functions are running on. This is where Docker shines. Cloud Functions like Azure Functions provide the ability to run custom containers from a repository like Docker Hub. In this tutorial, I will show how to create a custom container with an Azure Function that performs optical character recognition (OCR) in python, and

to perform OCR on the image and return the extracted text.

What you need

1. A Microsoft Azure Account. Get a free account with 1,000,000 free requests per month for Azure Functions: https://azure.microsoft.com/en-gb/free/
2. Azure Functions core tools. Installation steps here: https://github.com/Azure/azure-functions-core-tools
3. Azure CLI. Installation steps here: https://docs.microsoft.com/en-us/cli/azure/install-azure-cli-apt
4. Docker. Get Docker here: https://docs.docker.com/get-docker/
   If you're new to Docker, check out this excellent introduction from my favorite dev, Jeff Delaney:



5. An account on Docker Hub: https://hub.docker.com/
6. Python 3.*
7. Tesseract OCR: Install both the python module and core libraries:

```
sudo apt install tesseract-ocr
pip install pytesseract
```

```
pip install pillow
```

Creating an Azure Function

1. Run the following command to create a local function app project called `OcrFunctionsProject`. The `--docker` option will generate a `Dockerfile` that we can edit to install custom libraries and dependencies in the Azure Functions app where the container will be deployed:

   ```
   func init OcrFunctionsProject --worker-runtime python --docker
   ```

2. Navigate into the `OcrFunctionsProject` folder and edit the `Dockerfile` to look like this:

   ```
   FROM mcr.microsoft.com/azure-functions/python:3.0-python3.7
   ENV AzureWebJobsScriptRoot=/home/site/wwwroot \
   AzureFunctionsJobHost__Logging__Console__IsEnabled=true
   COPY requirements.txt /
   RUN pip install -r /requirements.txt
   RUN apt-get update && apt-get install -y \
   tesseract-ocr
   COPY . /home/site/wwwroot
   ```

   This will allow us to install Tesseract OCR in the base Debian machine where the Azure Functions will be hosted.

3. Add a function to the project using the following command. The `--name` option specifies a unique name for the function and `--template` specifies the trigger. In our case we want our function to run in response to an HTTP trigger.

   ```
   func new --name HttpOcrFunc --template "HTTP trigger"
   ```

4. Add `pytesseract` and `pillow` as a new line to the `requirements.txt` file so that modules are automatically installed once our container is deployed on the Azure Functions app in the cloud.

5. Test the new function locally by running the following command in the project root folder:

   ```
   func start
   ```

   Navigate to the `HttpOcrFunc` endpoint URL in the terminal output and if there is a response with 'This HTTP triggered function executed successfully.', then we're good

6. Edit the `OcrFunctionsProject/HttpOcrFunc/__init__.py` file and add the following code:

```
import logging
import pytesseract
from PIL import Image
import azure.functions as func
import os
def main(req: func.HttpRequest) -> func.HttpResponse:

    logging.info('Python HTTP trigger function processed a request.')

    # test code for OCR
    try:
        file = req.files.get('file')
        file.save('/tmp/1.jpg')
    except ValueError:
        pass

    text = ''
    if file:
        text = str(pytesseract.image_to_string(Image.open('/tmp/1.jpg')))

    return func.HttpResponse('text Extracted from Image: {}'.format(text))
```

Since the instance on which our Function will be deployed has a read only filesystem, we will persist our data to the /tmp/ directory.

Build and push Docker container

1. Build the docker image for the container described by our `Dockerfile` Remember to replace `<YOUR_DOCKER_HUBB_ID>` with your own Docker ID:

```
docker build --tag <YOUR_DOCKER_HUBB_ID>/ocrfunctionsimage:v1.0.0 .
```

2. Test the build by running the following command:

```
docker run -p 8080:80 -it <YOUR_DOCKER_HUB_ID>/ocrfunctionsimage:v1.0.0
```

Navigate to `http://localhost:8080` and you should see a placeholder image that says: Your Functions 3.0 app is up and running. We can't test the function running in this container because we need an access key that hasn't been generated yet as we haven't

```
docker login
```

and pushing:

```
docker push <YOUR_DOCKER_HUB_ID>/ocrfunctionsimage:v1.0.0
```

Create Azure Functions App

1. Go to Azure portal: https://portal.azure.com/ and create a new Resource Group by clicking on Create a resource and searching for Resource group. Give it a unique name, select your preferred location and click Review + Create. A resource group is a container of related resources for a specific Azure cloud solution. In our case, we will group an Azure Functions app and a Storage account in our resource group.
2. On the portal home, click Create a resource and search for and select Function App. In the Basics tab, select your Subscription, the resource group you just created, a unique Function App name and then for the publish field select Docker Container and finally region. Select Next: Hosting. In the Hosting tab, for storage, select a new storage account, then select a plan and select Review + Create. Finally, select create and then on the new page, select Go to resource.

3. On the Function App page, go to Container settings on the sidebar, select Docker Hub in image source and enter YOUR_DOCKER_HUB_ID/ocrfunctionsimage:v1.0.0 in the Full Image Name and Tag field and then click save.
4. Go to Functions on the sidebar, select the HttpOcrFunc function and click on Get Function URL.
5. Download Postman https://www.postman.com/ and create a new GET request. Paste the Function URL in the GET request endpoint field and in the body tab, select form-data, then add a key called file, change the type from Text to File, and in the value field upload any image with text in it. Click submit!

If you followed all steps in this tutorial, you should get the extracted text as a response on Postman.

Logs!

errors in the code that might prevent getting the expected output. While you can view the logs on the Azure Functions app portal by clicking on Functions in the sidebar, then clicking the relevant function and then clicking Monitoring in the sidebar, these logs are often very slow to appear after the initial request is made.

A better way to view logs is to test your function locally. This can be done as follows:

1. First, navigate to `OcrFunctionsProject/HttpOcrFunc/function.json` and change the value of the `authLevel` key in httpTrigger bindings to "anonymous".
2. Build the image using `docker build` as described in a previous step.
3. Run the container image using `docker run` as described in a previous step.
4. Use Postman to send a request to http://localhost:8080/api/HttpOcrFunc with an image file attached as form-data as described in a previous step and you can get immediate results from the local container as well as logs directly in your terminal. It's a good idea to test the function locally, and once you're done, change authLevel back to "function", rebuild the image, push to docker hub and save the new image on Azure Functions app portal container settings.

Conclusion

We now have a fully functioning Python OCR Docker container deployed to an Azure Function. We can trigger the function using an HTTP GET request to its public endpoint URL and attach an image file that will be parsed by tesseract OCR in the cloud function to extract text and return it in a response. We can create more functions and play around with triggers, for example initiating the function in response to an entry in a relational database. https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings?tabs=csharp
We can also persist an output file to blob storage using the Azure Blob Storage Python SDK. https://docs.microsoft.com/en-us/azure/storage/blobs/storage-quickstart-blobs-python
The sky is the limit and once the infrastructure is set up, you can easily edit the function python code, build the container, test is locally, push to Docker hub and then deploy it to Azure Functions. Hope this tutorial was useful for anyone looking to spin up a custom container on Azure Functions.

Add to the discussion

Johan Eriksson • Oct 16 '20

That was a nice article. But why even use docker ? I dont really see the purpose

2

Omer Farooq Ahmed • Oct 19 '20

So in this specific case, I needed to install tesseract-ocr on the Linux machine hosting my functions. One great way to do this is through a Dockerfile where I can define the commands I want to run on my base machine.

1

Code of Conduct • Report abuse

Omer Farooq Ahmed

Tech Associate at GSK & Machine Learning Engineer. Captivated by the applications of AI in Healthcare.

Follow

WORK
Tech Associate at GlaxoSmithKline

LOCATION
London

EDUCATION
Masters in Artificial Intelligence

JOINED
25 Apr 2019

2 1 3

Hardening Docker and Kubernetes with seccomp

#docker #kubernetes #devops #security

What is the nightmare for programmers?

#discuss #watercooler #beginners #codenewbie

5 Convincing reasons to choose Linux for programming

#linux #opensource #productivity #beginners