[Tutorial]: Serve a Containerised ML Model using FastAPI and Docker

Ashmi Banerjee · Follow

5 min read · Sep 3, 2022

22 1

> *A step-by-step tutorial to serve a containerised Machine Learning (ML) model using FastAPI and docker.*

Our tech stack for the tutorial

In my ***previous*** tutorial, we journeyed through building end-points to serve a machine learning (**ML**) model for an image classifier using

Python and *FastAPI*.

In this follow-up tutorial, we will focus on containerising the model using docker when serving through FastAPI.

> If you have followed my **last tutorial** on serving a pre-trained image classifier model from TensorFlow Hub using FastAPI, then you can **directly jump** to **Step 3** ⬇️ of this tutorial.😉

[Tutorial]: Serve an ML Model in Production using FastAPI

A step-by-step tutorial to serve a (pre-trained) image classifier model from TensorFlow Hub using FastAPI.

medium.com

Advantages of Containerisation

It works on my machine — a popular docker meme on the internet

Containerisation offers the following advantages:

1. **Performance consistency**
   DevOps teams know applications in containers will run the same, regardless of where they are deployed.
2. **Greater efficiency**
   Containers allow applications to be more rapidly deployed, patched,

or scaled.

3. **Less overhead**
   Containers require fewer system resources than traditional or hardware virtual machine environments because they don't include operating system images.

Methodology

We can achieve this using 2 approaches here.

1. Implements a `Dockerfile` which needs to be `built` manually every time there is a change in the code and then separately executed.
2. Automating the aforementioned process using `docker-compose`

Step 0: Prerequisites

1. Python 3.6+
2. You have ***docker*** installed.
3. Create the project structure as follows

```
fastapi-backend
├── src
│   ├── app
│   │   ├── app.py
│   └── pred
│   │   ├── models
│   │   │   ├── tf_pred.py
│   │   └── image_classifier.py
│   └── utils
│   │   ├── utilities.py
│   └── main.py
├── Dockerfile
├── docker-compose.yml
└── requirements.txt
```

Step 1: Setup

In the `app.py` file, implement the `/predict/tf/` end-point using `FastAPI`.

```python
from fastapi import FastAPI, HTTPException
from src.pred.image_classifier import *
from pydantic import BaseModel

app = FastAPI(title="Image Classifier API")

class Img(BaseModel):
    img_url: str

@app.post("/predict/tf/", status_code=200)
async def predict_tf(request: Img):
    prediction = tf_run_classifier(request.img_url)
    if not prediction:
        # the exception is raised, not returned - you will get a validation
        # error otherwise.
        raise HTTPException(
            status_code=404, detail="Image could not be downloaded"
        )
    return prediction
```

app.py hosted with ❤ by GitHub view raw

In the `main.py` file, we use the `uvicorn` server, which is an ASGI web server implementation for Python.

```python
import uvicorn

if __name__ == "__main__":
    uvicorn.run("app:app", host="0.0.0.0", port=8000, log_level="debug",
                proxy_headers=True, reload=True)
```

[fastapi_docker]main.py hosted with ❤ by **GitHub** **view raw**

Note: *Since the goal of the tutorial is to containerise the application, the detailed explanation of the above snippet has been explained in the* ***Step 2*** *of the* ***previous*** *tutorial.*

Step 2: Create a Dockerfile

The first step to containerise an application is to create a Dockerfile in your project directory *(see project structure above)*.

`Dockerfile` is a text document containing all instructions required to build a docker image.

We define our `Dockerfile` as follows:

```
FROM python:3-buster

RUN pip install --upgrade pip
WORKDIR /code

COPY ./requirements.txt /code/requirements.txt
RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt

COPY ./src ./src/
COPY ./src/main.py ./main.py
COPY ./src/app/app.py ./app.py

CMD ["python", "main.py"]
```

[fastapi_docker]Dockerfile hosted with ❤ by **GitHub** **view raw**

Explanations:

- `Line 1` : Downloads the specified Docker image (`python:3-buster`) from the ***docker hub registry***. You can check out the available images ***here***.

- `Line 3` : Upgrades `pip` so that later we can install the `requirements.txt`

- `Line 6` : Copies `requirements.txt` file and its content from the host to container

- `Line 7` : Installs the contents of `requirements.txt` inside the image

- `Lines 9-12` : Copies the respective files from the host to the image

- `Line 13` : CMD (executable) instruction is used to set a command to be executed when running a container. A `Dockerfile` must have only one CMD instruction and in case of multiple ones only the last one takes effect. This statement also sets the default command for the container to `python main.py` .

Step 4: Build and Run your App

Once we have implemented our Dockerfile, the next step is to `build` the docker image and then `run` it.

Step 4.1. Build the image

We build the image (here `myfastapiimage`) as follows.
You can call it by any name of your choice.

```
docker build -t myfastapiimage .
```

Important points to note:

- Do not forget the `.` after the image
- In case your docker image fails to rebuild after changing the files, you can initiate a forced build as follows:

```
docker build --no-cache -t myfastapiimage .
```

`docker build` usually uses cache to speed things up. `--no-cache` makes sure that it forcefully rebuilds it.

Step 4.2. Run the container

```
docker run --name mycontainer -p 80:8000 myfastapiimage
```

Explanations

- `docker run --name <myContainerName>` : creates a container named `mycontainer` (you can use any name here). The name of the container is later used for identification purposes.
- `-p <host_port>:<container_port>` maps the port of the host (`80`) to a port of the container (`8000`).

- `myfastapiimage` is the name of the image from which the container is derived.

[Alternative to Step 4] Use Docker-compose

`docker-compose` is a tool ideal for defining and running multi-container Docker applications. With Compose, we use a `YAML file` (`.yml`)to configure the services of the application.

The biggest advantage of using `docker-compose` is that, with a single command, we create and start all the services from our configuration. You can read more about `docker-compose` ***here***.

Now let's see how we define the `docker-compose` in our case. Since ours is a single service application, defining the `docker-compose` should be fairly simple.

```yaml
version: "3.9"  # optional since v1.27.0
services:
  app:
    container_name: mycontainer
    build: .
    ports:
      - "8000:8000"
    volumes:
      - .:/src/

```

[fastapi_docker]docker-compose.yml hosted with ❤ by **GitHub** **view raw**

Explanations:

- This docker-compose has a single service called `app` defined.
- The `app` service uses an image that is built from the `Dockerfile` in the current directory.
- It then binds the container and the host machine to the exposed port, `8000`
- It also binds the `/src/` from the host to the container. The volume binding is necessary to reflect any change in the files on the host in the container.

Running `docker-compose` :

From the project directory, start up the application by running the following command.

```
docker-compose up --build
```

Visit [***http://127.0.0.1:8000/***](http://127.0.0.1:8000/) from your browser to have the application up and running.

Some Handy Debugging Tips

- In case you’re facing problems with your image, you can try to get inside the container’s shell and debug it. To bash into the running container, type the following:

```
docker exec -t -i mycontainer /bin/bash
```

- In case your docker image fails to rebuild after changing the files, you can initiate a forced build as follows:

```
docker build --no-cache -t myfastapiimage .
```

- Sometimes, if you try re-running the docker container, you could get `container already in use` error. In that case, terminate the container, remove it by `docker rm <containerName>` and re-run your container. Or if you are using docker-compose, you can also type the following from the terminal (inside the project directory)

```
docker-compose down
```

- In case of `path errors`, be careful of the `container roots` you're using and adjust the paths accordingly.
- Also another handy docker command is the following:

```
docker ps -a
```

It provides you with all the list of all containers (running and stopped).

Conclusion

In this tutorial, we learnt how to containerise our application using `Docker`.

The next step after containerisation is deployment.
Once we are happy with the behaviour of our containerised application, we can deploy it on any managed/hybrid/on-premise cloud service with a minimised risk of failure.

Note: *this tutorial can be extended to any type of app containerisation and not only to FasAPI application.*
You just have to tweak the `Dockerfile` *and* `docker-compose` *accordingly. The rest remains the same* 😉.

✨ *The source code on GitHub can be accessed* ***here*****.**
The references and further readings on this topic have been summarised ***here****.*

✨ *If you like the article, please* ***subscribe*** *to get my latest ones.*
To get in touch, either reach out to me on ***LinkedIn*** *or via* ***ashmibanerjee.com****.*

Fastapi Docker Docker Compose Mlops Machine Learning

Written by Ashmi Banerjee

Follow

133 Followers

👩‍💻 Woman in tech, excited about new technical challenges. You can read more about me at: https://ashmibanerjee.com/

More from Ashmi Banerjee

Ashmi Banerjee

[Tutorial]: Performance Test ML Serving APIs using Locust and...

A step-by-step tutorial to use Locust to load test a (pre-trained) image classifier...

6 min read · Jul 11, 2022

75

Ashmi Banerjee

[Tutorial]: Serve an ML Model in Production using FastAPI

A step-by-step tutorial to serve a (pre-trained) image classifier model from...

5 min read · Jun 28, 2022

115 2

Ashmi Banerjee

[Tutorial]: Introduction to ML Model Serving using TensorFlo...

A step-by-step tutorial to serve a (pre-trained) image classifier model from...

6 min read · Sep 10, 2022

8 2

Ashmi Banerjee

[Tutorial]: Serve an ML Model as REST API using TensorFlow...

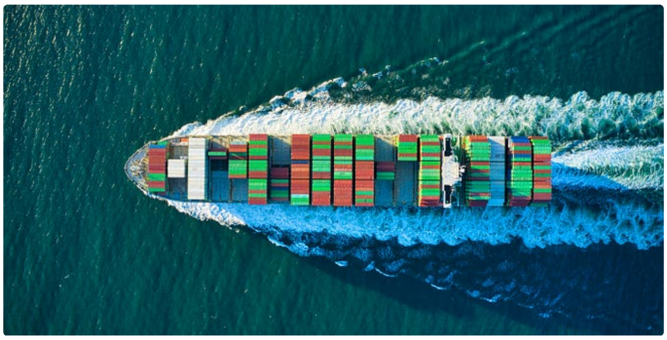
A step-by-step tutorial to serve a (pre-trained) image classifier model from...

6 min read · Aug 2, 2022

65 1

See all from Ashmi Banerjee

Recommended from Medium



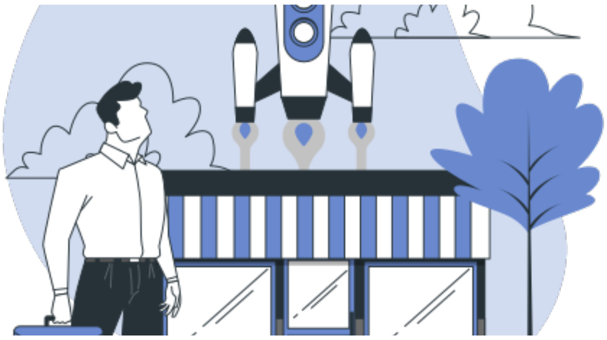
Naman Gupta

How to deploy ML Models on AWS ECS using Docker and...

Highly secure, reliable, and scalable way to run ML Models in containers!!

6 min read · Jan 14, 2024

53



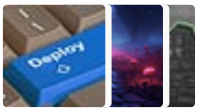
Ido Ali

Building Deep Learning API with FastAPI & Docker

Introduction

8 min read · Feb 5, 2024

Lists

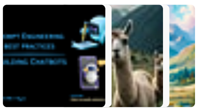


Predictive Modeling w/ Python

20 stories · 1124 saves

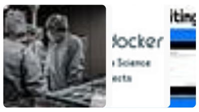
Practical Guides to Machine Learning

10 stories · 1349 saves



Natural Language Processing

1399 stories · 898 saves



Coding & Development

11 stories · 577 saves

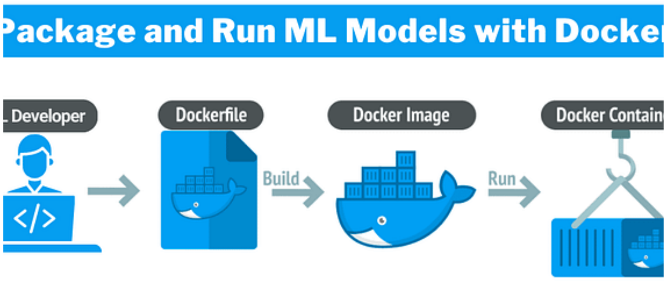
Reza Shokrzad

FastAPI: The Modern Toolkit for Machine Learning Deployment

In the constantly changing world of web development, we are always looking for...

9 min read · Nov 26, 2023

👏 2 💬



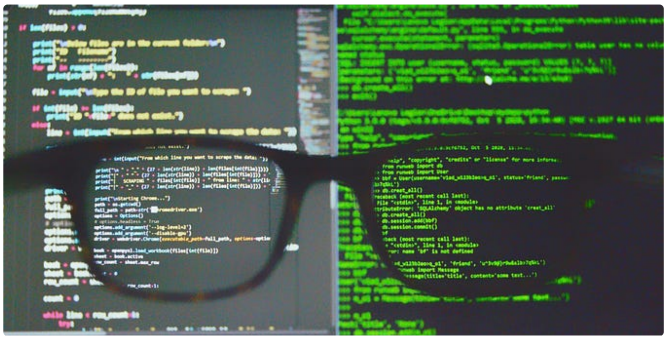
Sushant Kapare

Containerizing and Deploying Machine Learning Models with...

Introduction:-

5 min read · Dec 5, 2023

👏 21 💬 1



Charu Makhijani

Machine Learning Model Deployment as a Web App usin...

Complete Guide to deploy ML model using Streamlit

5 min read · Nov 7, 2023

👏 388 💬 4

Hai Rozencwajg in Towards Data Science

Deploy a Custom ML Model as a SageMaker Endpoint

A quick and easy guide for creating an AWS SageMaker endpoint for your model

10 min read · Dec 8, 2023

👏 218 💬 3

See more recommendations