Rihab Feki Follow
Sep 6, 2021 · 9 min read · Listen

Save

Deploying Machine Learning models with Streamlit, FastAPI and Docker

A step by step guide to put your ML model into production

Photo by Matt Hardy from Pexels

Have your Machine Learning experiments been floating around Jupyter Notebooks and your performant trained models never left your local machine? If YES, this tutorial is for you to get to learn how to serve your model in production.

If you also want to find out more about Streamlit, FastAPI and Docker and why to use them, keep on reading.

In this article I will cover:

- **Building a web app using Streamlit**
- **Deploying a Machine Learning Model using fastAPI**
- **Serving model via REST API with FastAPI**
- **Setting up the environment with Docker and Docker-compose**

Before getting into the implementation I will share with you some important concepts which will help you get through the whole tutorial and have a better understanding of the coming steps.

What means deploying a Machine Learning model?

Deploying a machine learning model simply means making your model available to other IT systems within your organisation or the web. So that it can be consumed by receiving data and returning its predictions.

> Here are the steps I followed to deploy my ML model:

1. Training a machine learning model on a local system.
2. Creating a frontend to make the model accessible via the web using a web framework e.g Streamlit.
3. Wrapping the inference logic with a backend framework e.g FastAPI.
4. Using docker to containerise your application.
5. Hosting the docker container on the cloud and consuming the web-service.

What means an API and what is it used for?

An **API** stands for **A**pplication **P**rogramming **I**nterface. It's simply an intermediary between two independent applications that communicate with each other.

Creating an API opens up certain user-defined URL endpoints, which can be used to send requests or receive responses with data via HTTP methods.

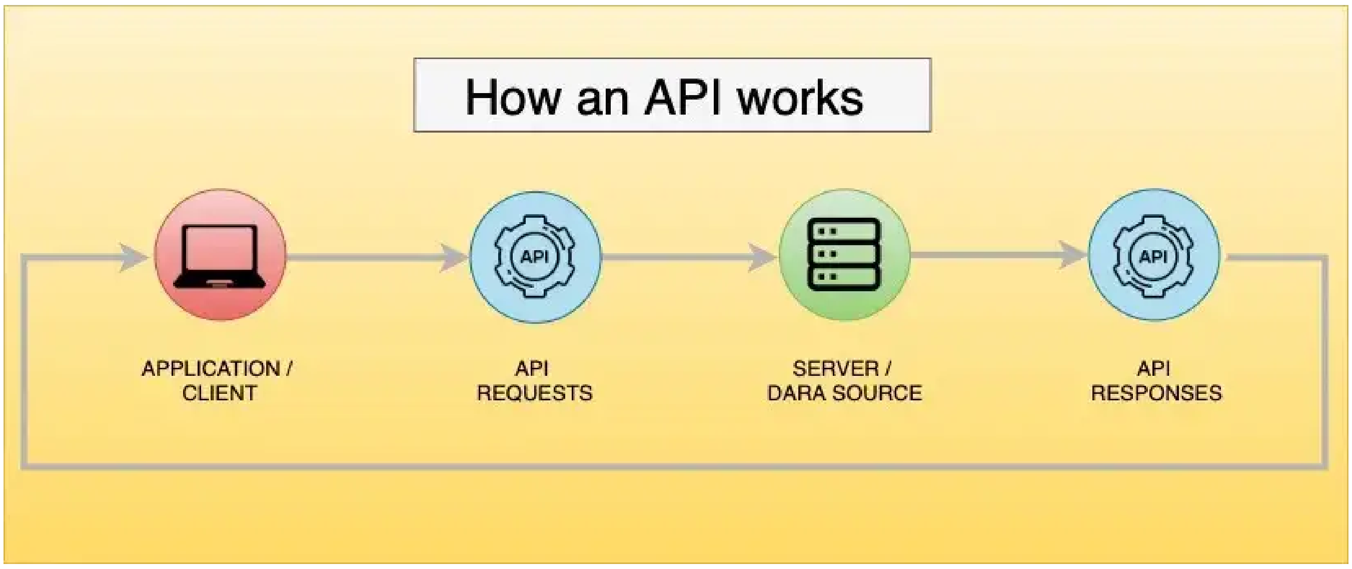


Image by the author

Think of APIs as an abstraction of your application (the users don't see your code and don't install anything, they simply call the API) and a simplification of the integration with third parties (developers, apps, other APIs, etc.)

Now that you understood these basic concepts, let's dive into the technical details of the solution implementation.

What is Streamlit and why it is useful?

Streamlit is an **open-source python framework for building web apps for Machine Learning and Data Science.** This framework makes it easy for data scientists and ML engineers to create powerful user interfaces that interact with machine learning models.

Until the arrival of Streamlit, Flask and Django were the goto libraries which developers chose to use in order to **develop** and **deploy** their **application over the web**; however both these frameworks required the user to write HTML/CSS code to render their work as a web-app. Streamlit abstracts all this and provides an easy pythonic interface for adding custom components like sliders, drop-downs, forms, etc...

Check these links to find out some awesome content about Streamlit 👇

> *This article showcases all **Streamlit functionalities with examples.***
>
> *This is **Streamlit gallery**: you can find here a selection of applications developed by the community with its source code.*
>
> *This is **Streamlit cheat sheet***

What is FastAPI and why it is special?

FastAPI is a web framework that accelerates the **backend development** of a website using **Python**. This framework is new, adaptive, and easy to learn. It allows users to quickly set up the API, generates automatic docs for all the endpoints, offers authentication, data validation, allows **asynchronous** code, and much more.

Since the majority of machine learning models are developed in Python, the web frameworks that serve them up are usually Python-based as well. For a long time, Flask, a micro-framework, was the goto framework. But that's changing. A new framework, designed to compensate for almost everything Flask lacks is becoming more and more popular. It's called FastAPI.

Fast API, works perfectly if your concern is **speed**. It also **scales perfectly in deploying production-ready machine learning models** because ML models work best in production when they are wrapped around a **REST API** and deployed in a microservice.

> *Check this article to find out more which framework to use between Django, Flask, and FastAPI.*
>
> *FastAPI supports data validation via pydantic and automatic API documentation as well.*

What is Docker?

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and deploy it as one package.

What is Docker-compose?

Docker compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

Check this article to know more about Docker

All you need to know about Docker, Docker-Compose and Dockerfile.

A detailed explanation of dockerization

rihab-feki.medium.com

In this coming part, I will explain the implementation steps. You can find the link to the source code on Github here.

Let's get started!

This project consists of two parts, a frontend based on Streamlit and a backend based on FastAPI and the whole application is packaged using Docker.

Setting up a Streamlit app with Docker

I used Streamlit to create the frontend for my web application and expose my Machine Learning model via a web page to be used in production.

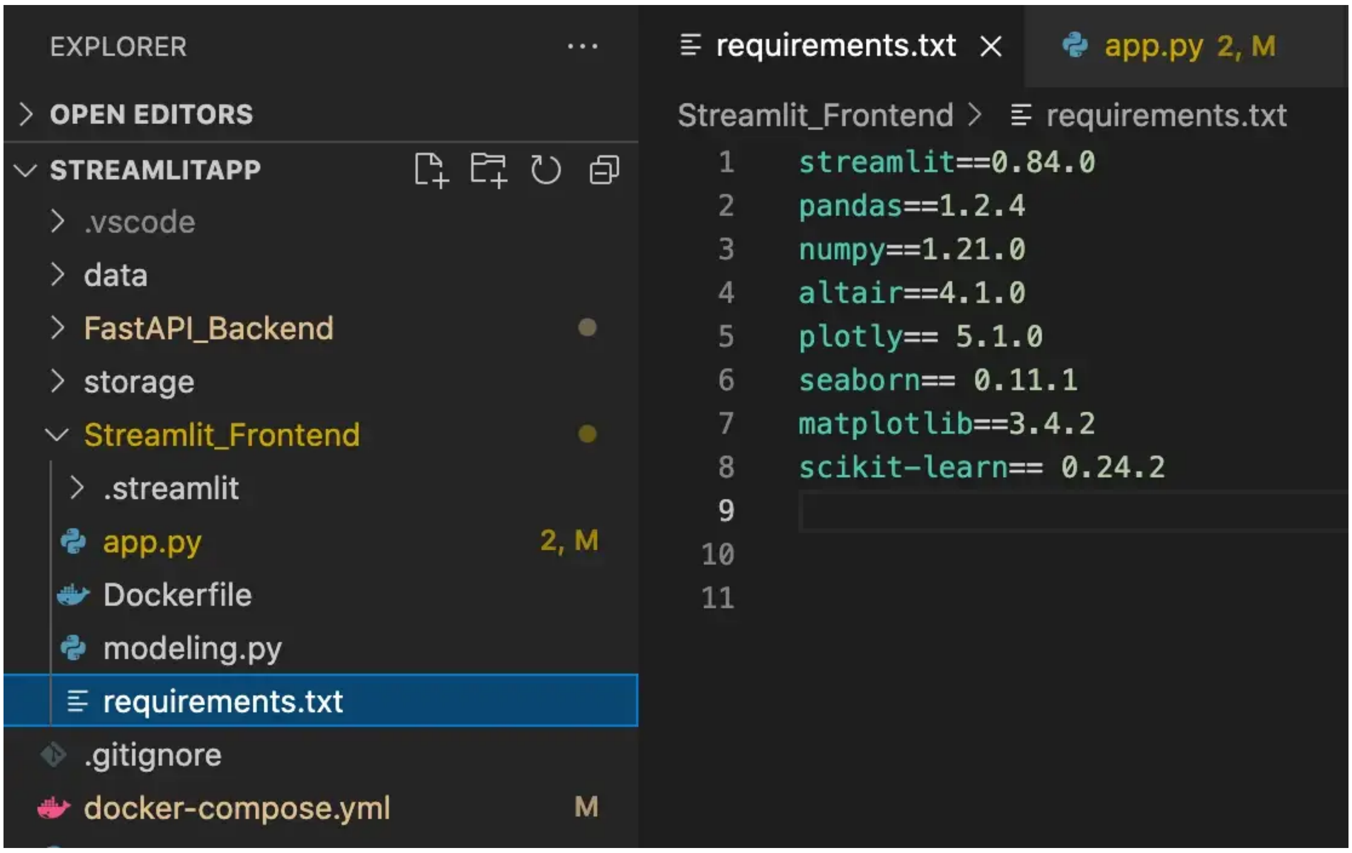
This is my frontend repository structure:

```
📦Streamlit_Frontend
 ┣ 📂.streamlit
 ┃ ┗ 📜config.toml
 ┣ 📜Dockerfile
 ┣ 📜app.py
 ┣ 📜modeling.py
 ┗ 📜requirements.txt
```

In the app.py you create your web page and use Streamlit widgets to do so.

For Custom themes for your Stremalit app, they can be defined in the config file: `./.streamlit/config.toml` . Check this link for more details.

The image below shows the dependencies I used:



```
streamlit==0.84.0
pandas==1.2.4
numpy==1.21.0
altair==4.1.0
plotly== 5.1.0
seaborn== 0.11.1
matplotlib==3.4.2
scikit-learn== 0.24.2
```

Screenshot by the author

The following is the Dockerfile I used to build my Streamlit application. It consists of the dependencies which I listed in the requirements.txt file. Once I run the build command, these dependencies and will be installed with its corresponding versions. I am using Python to develop my app and I can access it on port 8501.

```
# frontent/Dockerfile

FROM python:3.9

COPY requirements.txt app/requirements.txt

WORKDIR /app

RUN pip install -r requirements.txt

COPY . /app

EXPOSE 8501

ENTRYPOINT ["streamlit","run"]
CMD ["app.py"]
```

Dockerfile hosted with ❤ by GitHub view raw

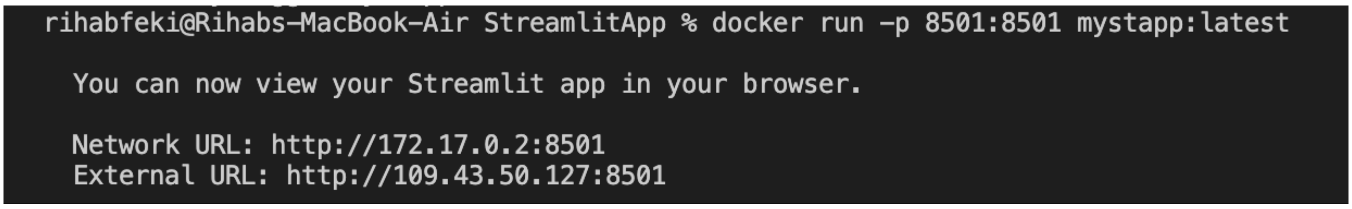
The command to build my Dockerfile is:

```
docker build -t mystapp:latest .
```

To run the application use this command in the terminal:

```
docker run -p 8501:8501 mystapp:latest
```

You should get this as a result:



```
rihabfeki@Rihabs-MacBook-Air StreamlitApp % docker run -p 8501:8501 mystapp:latest

  You can now view your Streamlit app in your browser.

  Network URL: http://172.17.0.2:8501
  External URL: http://109.43.50.127:8501
```

Then you can view your app in the browser using the Network URL: http://localhost:8501/

Setting up the backend with FastAPI and Docker

As previously introduced, I used FastAPI as the framework for the backend development of my web app. I used it to easily define my **APIs** and to deploy my Machine Learning model.

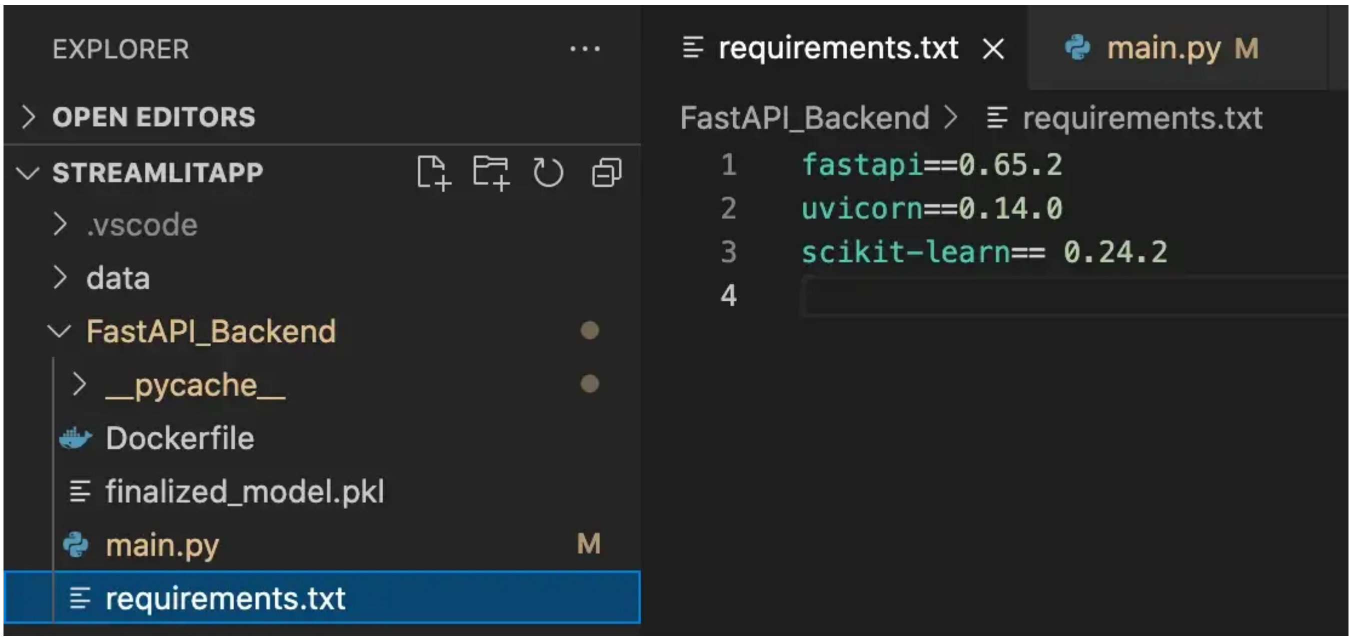
I have already trained my model and I saved it as a pickle file. This pre-trained model will be triggered via the web interface, to get predictions from a test data sample which the end user will provide.

This is my backend repository structure:

```
📦FastAPI_Backend
 ┣ 📜Dockerfile
 ┣ 📜finalized_model.pkl
 ┣ 📜main.py
 ┗ 📜requirements.txt
```

In the main.py file, I load my pre-trianed model and create APIs for getting attributes from the test data and generate the prediction associated to it.

These are the dependencies I used in this project for the backend:



```
fastapi==0.65.2
uvicorn==0.14.0
scikit-learn== 0.24.2
```

Screenshot by the author

And to build the backend, I also used a Docker file:

```
# backend/Dockerfile

FROM python:3.9

COPY requirements.txt app/requirements.txt

WORKDIR /app

RUN pip install -r requirements.txt

COPY . /app

EXPOSE 8000

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000" , "--reload"]
```

Dockerfile hosted with ❤ by GitHub view raw

To elaborate on the last command in the backend Dockerfile, the following are the defined **settings** for Uvicorn:

— **host 0.0.0.0** defines the address to host the server on.

— **port 8008** defines the port to host the server on.

main:app tells Uvicorn where it can find the FastAPI ASGI application — e.g., "within the the 'main.py' file, you'll find the ASGI app, app = FastAPI().

— **reload** enables auto-reload so the server will restart after changes are made to the code base.

FastAPI provides an **API document** engine too. If you visit http://localhost:8000/docs which is using the Swagger UI interface.

This is a simple Hello World API with FastAPI

```python
import uvicorn
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def home():
    return {"Hello": "World"}

if __name__ == "__main__":
    uvicorn.run("hello_world_fastapi:app")
```

FastAPI-hello-world.py hosted with ❤ by GitHub view raw

Serving the API is uvicorn's responsibility which is a good choice given that uvicorn is a lightning-fast ASGI server implementation, using uvloop and httptools.

After setting up our app it's time to integrate the model into the FastAPI code structure of making prediction requests. I created a **"/prediction"** route which will take the data sent by the client request body and the API will return the response as a JSON object containing the result.

This is my main.py file in which I created my prediction API

```python
from fastapi import FastAPI
from pydantic import BaseModel
import pickle

app = FastAPI()

#Creating a class for the attributes input to the ML model.
class water_metrics(BaseModel):
        ph : float
        Hardness :float
        Solids : float
        Chloramines : float
        Sulfate : float
        Conductivity : float
        Organic_carbon : float
        Trihalomethanes : float
        Turbidity : float

#Loading the trained model
with open("./finalized_model.pkl", "rb") as f:
```

```python
def get_potability(data: water_metrics):
    received = data.dict()
    ph = received['ph']
    Hardness = received['Hardness']
    Solids = received['Solids']
    Chloramines = received['Chloramines']
    Sulfate = received['Sulfate']
    Conductivity = received['Conductivity']
    Organic_carbon = received['Organic_carbon']
    Trihalomethanes = received['Trihalomethanes']
    Turbidity = received['Turbidity']
    pred_name = loaded_model.predict([[ph, Hardness, Solids,
                               Chloramines, Sulfate, Conductivity, Organic_carbon,
                               Trihalomethanes,Turbidity]]).tolist()[0]
    return {'Prediction':  pred_name}
```

main.py hosted with ❤️ by GitHub view raw

These are the steps which explain the code below:

1. I have created a "water_metrics" Model class that defines all the parameters of our ML model. All the values are float type.

2. Next, I am loading the model by unpickling it and saving the model as "loaded_model". This model object will be used to get the predictions.

3. The "/prediction" route function declares a parameter called "data" of the "water_metrics" Model type. This parameter can be accessed as a dictionary. The dictionary object will allow you to access the values of the parameters as key-value pairs.

4. Now, you are saving all the parameter values sent by the client. These values are now fed to the model predict function and you have your prediction for the data provided

From the "backend" folder in your terminal, build the image:

```
$ docker build -t backe
```

307 | 1 | •••

Run the container:

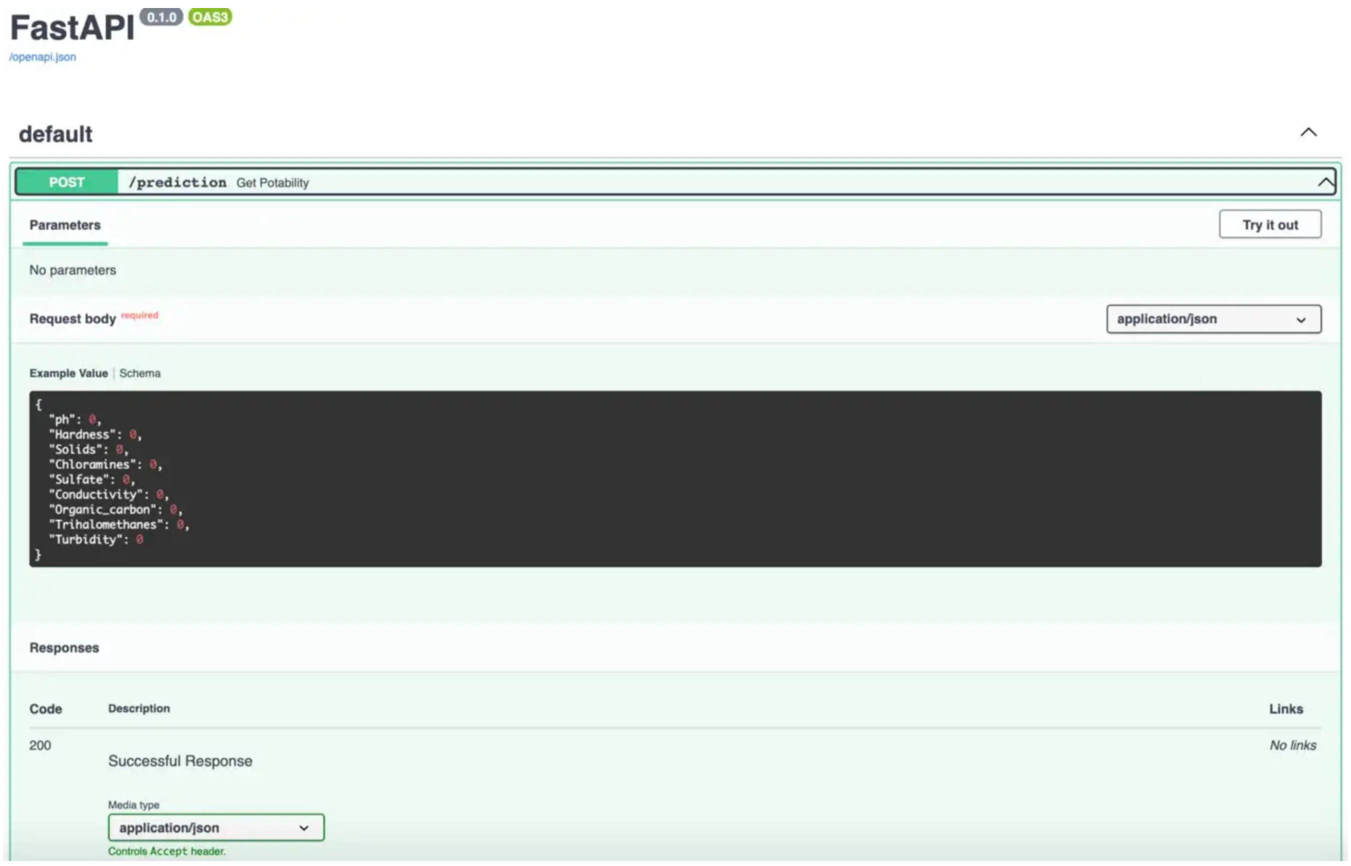
```
$ docker run -p 8080:8080 backend
```

In your browser, navigate to http://localhost:8080/. And you should see:

```
{
  {"message":"This is the homepage of the API "}
}
```

You could access you API documentation if you navigate to http://localhost:8080/docs

You'll see the documentation for every route you created as well as an interactive interface where you can test each endpoint directly from the browser.



```
{
  "ph": 0,
  "Hardness": 0,
  "Solids": 0,
  "Chloramines": 0,
  "Sulfate": 0,
  "Conductivity": 0,
  "Organic_carbon": 0,
  "Trihalomethanes": 0,
  "Turbidity": 0
}
```

Image by the author

Now you will define a docker-compose that will create a service for our API and a service for the frontend Streamlit app.
Docker-compose is great because it is a tool for defining multi-container applications and it enables you to configure them and also it gives you the ability to set up the communication between you services.

This is the docker-compose file which I used

```yaml
version: "3.7"

services:
  frontend:
    build: Streamlit_Frontend
    ports:
      - 8501:8501
    networks:
      AIservice:
        aliases:
          - frontend.docker
    depends_on:
      - backend
    volumes:
        - ./Streamlit_Frontend:/app
        - ./storage:/storage

  backend:
    build: FastAPI_Backend
    ports:
      - 8000:8000
    networks:
      AIservice:
        aliases:
          - backend.docker
    volumes:
      - ./FastAPI_Backend:/app
      - ./storage:/storage
```

docker-compose.yml hosted with ❤ by GitHub view raw

This docker-compose file is configured that we can no longer need to build each Dockerfile of the different images at a time. we could do that in one shot by running the docker- compose command:

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

The Image for service Streamlit and FastAPI_Backend were built because they did not already exist.

To enable the communication between the docker containers of the frontend and the backend, creating a network and giving each of the containers an alias was necessary.

To see changes it is useful to use the following commands:

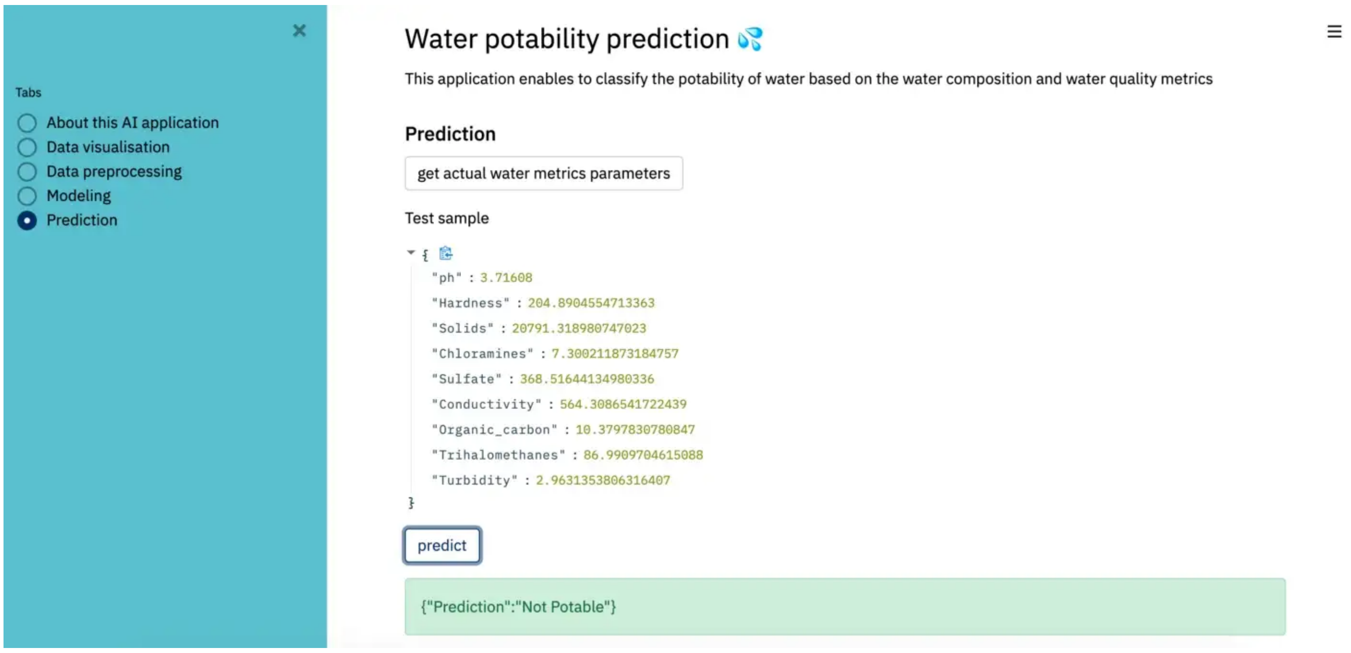
```
docker-compose stop
```

And then, if you have not added any new dependency, you do not need to rebuild again your images of Streamlit and FastAPI and you could just use this command to see changes:

```
docker-compose up -d
```

You have learned now how you could configure your Streamlit application and FastAPI backend with Docker and you are now able to serve you Machine Learning models into production in this way.

And this is the final result:



Water potability prediction 💦

This application enables to classify the potability of water based on the water composition and water quality metrics

Prediction

Test sample

```
{
  "ph" : 3.71608
  "Hardness" : 204.8904554713363
  "Solids" : 20791.318980747023
  "Chloramines" : 7.300211873184757
  "Sulfate" : 368.51644134980336
  "Conductivity" : 564.3086541722439
  "Organic_carbon" : 10.3797830780847
  "Trihalomethanes" : 86.9909704615088
  "Turbidity" : 2.9631353806316407
}
```

```
{"Prediction":"Not Potable"}
```

Image by the author

There are many ways to do that and this was one of them.

I hope this article was useful and I see you in a new article!

> *Author: Rihab Feki*

Docker Streamlit Fastapi Deployment Machine Learning