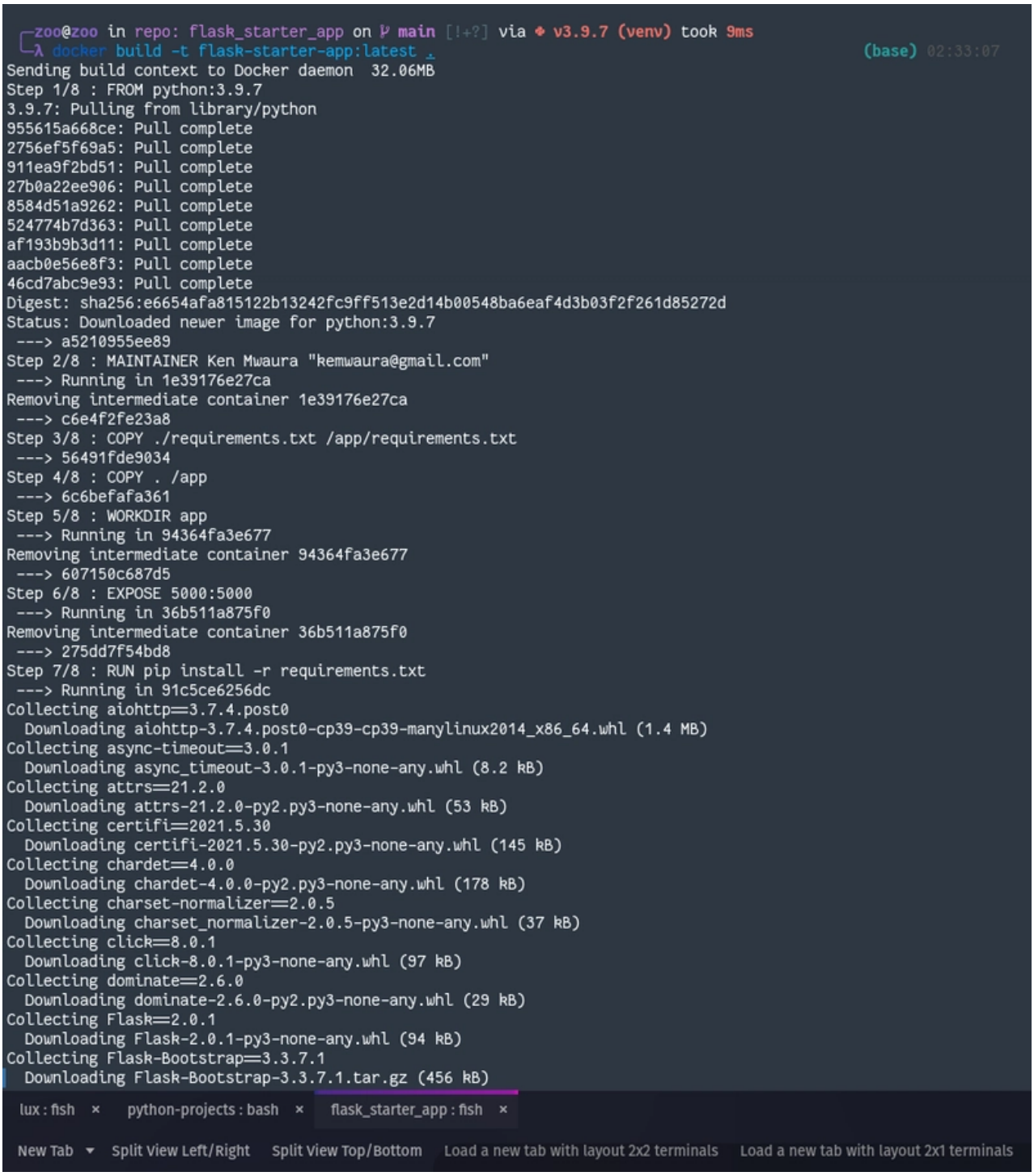
Getting Started with Flask and Docker 🐳🚀

dev.to/ken_mwaura1/getting-started-with-flask-and-docker-3ie8

Zoo Codes September 16, 2021



Over the past few weeks, I've worked on a few flask apps across a variety of use cases. The aim was brush up my knowledge of flask as well proper structure for a production application. When I got challenged to use docker and flask app for a starter project and write about it. It was a perfect opportunity to really cement my knowledge as well provide my version of a quick-start guide.

Audience and Objectives

This article is aimed at beginner developers who are looking for a guide using docker and flask. However, intermediate developers can also glean some knowledge. I will also endeavor to point out issues I faced while working on this project.

This article aims at developing a simple flask app and dockerizing the app and pushing the code to GitHub.

Prerequisites to Getting Started

To effectively follow along with this post and subsequent code, you will need the following prerequisites.

- Python and pip (I am currently using 3.9.7) Any version above 3.7 should work.
- Git installed in your system. Check appropriate instructions for your system.
- Docker on your system. Installation instructions
- Terminal.

Initial Setup

These instructions are verified to work on most Unix systems. **Note:** Windows implementation may vary.

Create a new directory and change into it.

```
mkdir flask_starter_app && cd flask_starter_app
```

- Create a new virtual environment for the project. Alternatively activate your preferred virtual environment.
- Proceed to use pip to install our required modules using pip. we'll be using flask, flask-bootstrap and jikanpy
- Save the installed packages in a requirements file.

```
python3 -m venv venv
source venv/bin/activate
pip install flask flask-bootstrap jikanpy
pip freeze > requirements.txt
```

We are installing main flask module for our project. Flask-Bootstrap will help us integrate bootstrap in our app for styling.
We also install Jikanpy is a python wrapper for Jikan Api, which is the unofficial MyAnimeList Api.

Hopefully, everything is installed successfully. Alternatively check the code on

Simple flask starter app utilizing docker to showcase seasonal anime using jikanpy (myanimelist unofficial api).

Link to write-up here

Using Docker is recommended, as it guarantees the application is run using compatible versions of Python and Node.

Inside the app there a Dockerfile to help you get started.

To build the development version of the app

```
docker build -t flask-starter-app .
```

To run the app

```
 docker run --name=flask-app -p 5001:5000 -t -i flask-starter-app
```

If everything went well, the app should be running on localhost:5001

View on GitHub

It's All Containers

Docker refers to open source containerization platform.
Containers are standardized executable components that combine application source code with OS-level dependencies and libraries. We can create containers without docker, however it provides a consistent, simpler and safer way to build containers. One of the major reasons for the meteoric growth of the use of containers from software development to software delivery and even testing, is the ease of use and reproducibility of the entire workflow.

Previously developers used Virtual Machines in the cloud or self-hosted servers to run their applications and workloads.
However, going from development to production was sometimes plagued with failures due differences in Operating systems or at times dependencies. Containers allow us to essentially take the code, file structure, dependencies etc. and package them and deploy them to a server and have them run as expected with minimal changes.

Terminology

Here we'll run through some tools and terminology in reference to Docker:

DockerFile

Docker containers start out as single text file containing all the relevant instructions on how build an *image*.
A *Dockerfile* automates the process of creating an image, contains a series of CLI

instructions for the Docker engine to assemble the image.

Docker Images

Docker images hold all application source code, libraries and dependencies to run an application. It is very possible to build a docker image from scratch but developers leverage common repositories to pull down pre-built images for common software and tools.
Docker images are made up of layers and each time a container is built from an image. a new layer is added becoming the latest version of the image. You can use a single to run multiple live containers.

Docker Hub

This is a public repository of Docker images, containing over 100,000 container images. It holds containers of software from commercial vendors, open-source projects and even individual developers.

Docker daemon

Refers the service that runs in your system powering the creation of Docker images and containers.The daemon receives commands from client and executes them.

Docker registry

This is an open-source scalable storage and distribution system for docker images. Using git(a version control system) the registry track image versions in repositories using tags for identification.

Let's Build!

Flask

Flask prides itself in being micro framework therefore it only comes with simple configuration out of the box. However. it allows for a wide range of custom configuration options. This gives you
the freedom to start simple, add extensions for variety utilities as you grow.

What we're Building

Today we'll be building a simple web app to display the current seasonal anime from MyAnimeList. If you follow me on Twitter you'll know am a massive manga and anime fan. MyAnimeList is defacto platform for information, reviews and rankings thus it was the optimal choice. However, it doesn't have an API or sdk to access their content. Normally we would have to scrape the site, luckily the awesome community created Jikan Api as well jikanpy which is python wrapper for the API.

Where's the code?

Now hopefully you carried out the steps above in Prerequisites section. Ensure your virtual environment is activated. Inside our `flask-starter-app` directory create `run.py` file.

```
# run.py
from app import app


if __name__ == '__main__':
    app.run()
```

This file will serve our app. First we import our app instance from app directory, which doesn't exist yet. Let's create it.
Create the app directory, inside it create:

- templates folder
- `__init__.py` file
- `models.py` file
- `views.py` file
- `anime_requests.py` file

Your folder structure should now look like:

```
python-projects $ tree flask_starter_app
flask_starter_app
├── app
│   ├── __init__.py
│   ├── anime_request.py
│   ├── models.py
│   ├── views.py
│   └── templates
│       └── index.html
└── requirements.txt
└── run.py
└── venv


2 directories, 7 files
```

Inside the `__init__.py` file add the following code:

```
# app/__init__.py

from flask import Flask
from flask_bootstrap import Bootstrap

bootstrap = Bootstrap()
app = Flask(__name__)
bootstrap.init_app(app)
from . import views
```

The first two lines import the Flask and Bootstrap classes from their respective modules. We then instantiate the Bootstrap class and assign it bootstrap variable.The app variable contains an instance of the Flask class and pass along the **name** as the first parameter to refer to our app. We then initialize bootstrap by calling the `init_app()` method as passing pur app as an argument.
Finally, we import views file from our current directory.

Class is in Session

Inside the `models.py` file ad the following code:

```
# app/models.py
from dataclasses import dataclass

@dataclass
class Anime:
    """
    class to model anime data
    """
    mal_id: int
    url: str
    title: str
    image_url: str
    synopsis: str
    type: str
    airing_start: str
    episodes: int
    members: int
```

This file will hold all of our models, here we create an Anime class to hold data from the Api. We import dataclass decorator from the dataclasses module. This will give us access to a variety of special methods, allowing us to keep our code simple and succinct. We attach the decorator to our class and then proceed to define the structure of the data from the Api. Check the docs to understand more.

Requests, Requests ...

Add the following to the `anime_request.py` file:

```
# app/anime_request.py
from jikanpy import Jikan
from .models import Anime

jikan = Jikan()
# function to get seasonal anime
def get_season_anime():
    """
    function to get the top anime from My anime list
    :return: list of anime
    """
    season_anime_request = jikan.season()
    season_anime = []
    if season_anime_request['anime']:
        response = season_anime_request['anime']

        for anime in response:
            mal_id = anime.get('mal_id')
            url = anime.get('url')
            title = anime.get('title')
            image_url = anime.get('image_url')
            synopsis = anime.get('synopsis')
            type = anime.get('type')
            airing_start = anime.get('airing_start')
            episodes = anime.get('episodes')
            members = anime.get('members')

            new_anime = Anime(mal_id, url, title, image_url, synopsis, type,
airing_start, episodes, members)
            season_anime.append(new_anime)

    return season_anime
```

In the code above we import Jikan class from the jikanpy module, this will give us access to variety of methods to make requests to the Jikan Api. We also import our Anime class from the `models` file. we create a variable jikan and assign it an instance of the Jikan class.
We now define `get_season_anime` function to make requests to the Jikan Api and append it to a list. We create a variable `season_anime_request` that calls `season` method from Jikan class. It accepts the two parameters: year and season, this is handy when you want retrieve specific data from year and even season. In our case we don't specify in order to get the current season anime. We then define an empty list to hold our data.

The season method returns a dictionary of various key value pairs. The data we need is values of the `anime` key. which is a list of dictionaries. We add an if statement to check if key we want exists, then loop through the values. We create appropriate variables to reference the data in the response.
We create a `new_anime` variable that is an instance of Anime class. We append our class to our empty list, finally we return the list of classes.

Views For Days

Add the following code your `views.py` file.

```
from flask import render_template
from .anime_request import get_season_anime

from . import app


@app.route('/', methods=['GET', 'POST'])
def index():
    """
    root page view that returns the index page and its data
    :return: index template
    """
    season_anime = get_season_anime()
    return render_template('index.html', season_anime=season_anime)
```

This file is holds routes for our flask application. Currently, we'll only have one, feel free to add more. We begin by importing `render_template` this will pass render our html pages in the browser and pass along any required parameters. We also import `get_season_anime` function from anime_request file. We also import our app from `__init__.py` file, this allows use the `@app` decorator that exposes the route method. This registers routes passed as arguments as well the methods allowed for the route.

We define the `index` function that will be called once the user opens root route. Inside the function, we define season_anime variable that holds list of instance of the Anime classes. We finally call `render_template` function and pass along our index.html file inside the templates folder, along with season_anime variable to our template.

Add the following to your index.html file inside the templates folder:

```
<!--app/templates/index.html -->
{% extends 'bootstrap/base.html' %}

{% block navbar %}
    <div class="navbar navbar-inverse" role="navigation">
        <div class="container-fluid">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse"
data-target=".navbar-collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a class="navbar-brand" href="/"> Anime Watchlist </a>
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a href="/">Home</a></li>
                </ul>
            </div>
        </div>
    </div>
{% endblock %}

{% block content %}
    {% for anime in season_anime %}
        <div class="card-group col-xs-12 col-sm-4 col-md-2 col-lg-2">
            <div class="card">
                <img src="{{ anime.image_url }}" alt="{{ anime.title }} poster"
class="img-responsive"
                     style="max-height: 30rem">
                <li class="text-left ">
                    <a href="/anime/{{anime.mal_id}}">
                        {{ anime.title|truncate(30)}}</a>
                    <p> Episodes : {{ anime.episodes }}</p>
                    <p> date started airing: {{ anime.airing_start | truncate(13)
}}</p>
                </li>
            </div>

        </div>
    {% endfor %}
{% endblock %}
```

Flask uses the jinja templating engine. This allows us to use a slew of advanced features. In our case we extend the base html file containing all bootstrap styling, this keeps allows to have a basic structure that applies all of our pages. We also use special `{% %}` to define a special navbar block.
Normally this is set in its own file and imported but here we'll just have it here. We define a content block inside we loop through season_anime argument passed in our views file. For each value we render a card with title, image, number of episodes and the date it started airing.

Open a terminal and run `python run.py` Your app should look similar below:

Dockerize Everything

Now we have a fully functional flask app, lets dockerize it. Inside the root of our app(flask_starter_app), create a `Dockerfile`.
Add the following configuration:

```
#Dockerfile

FROM python:3.9.7

MAINTAINER Ken Mwaura "kemwaura@gmail.com"

COPY ./requirements.txt /app/requirements.txt

RUN pip install -r requirements.txt

COPY . /app

WORKDIR app

ENV FLASK_APP=run.py

ENV FLASK_ENV=development

EXPOSE 5001:5000

CMD ["flask", "run", "--host", "0.0.0.0"]
```

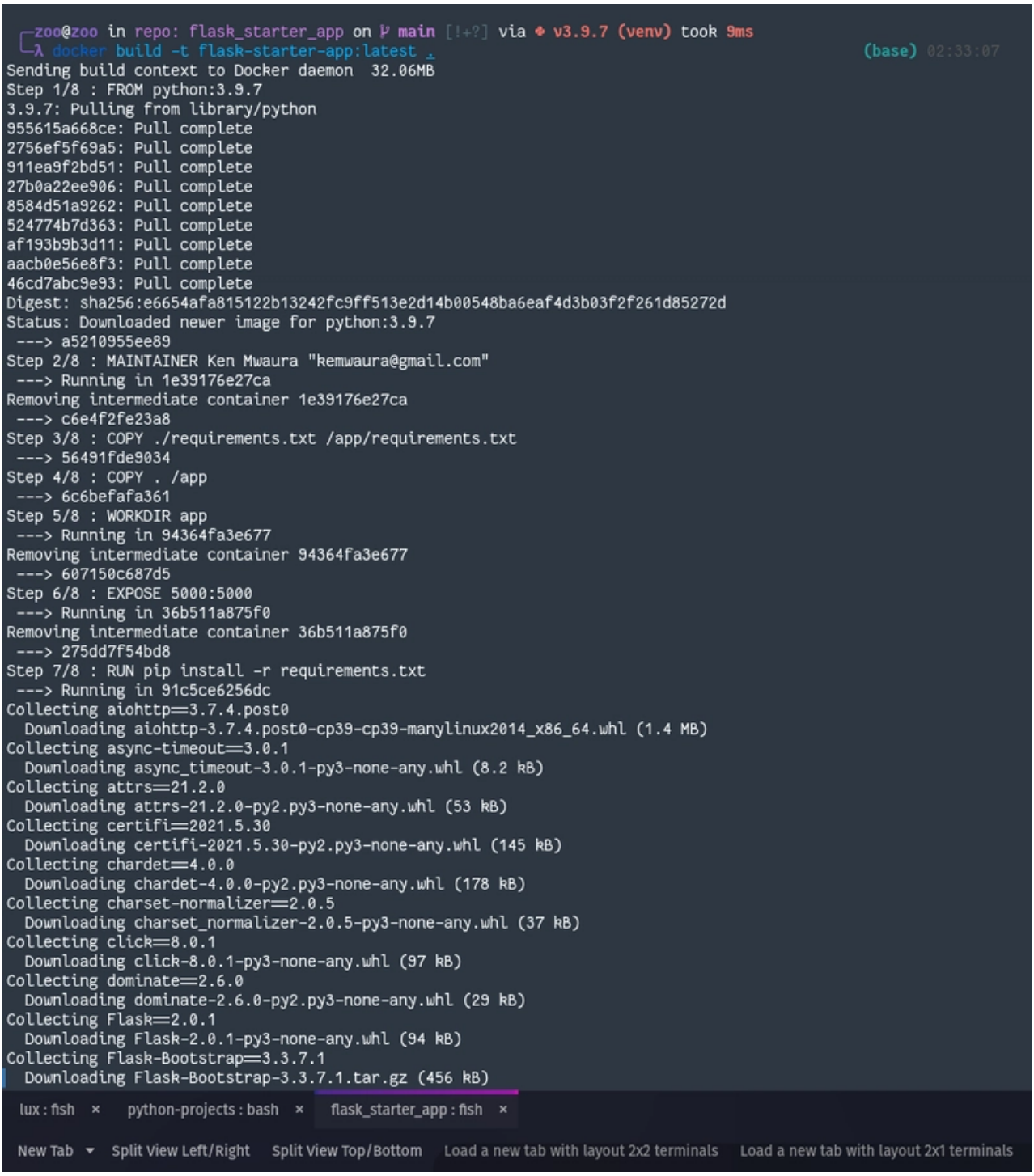
The first line sets the base image to build from, in our case we're using the python 3.9.7 image to mirror the development
environment. Let’s go over some of these Docker instructions:

1. MAINTAINER sets the Author field of the image (useful when pushing to Docker Hub).
2. COPY copies files from the first parameter (the source .) to the destination parameter (in this case, /app).
3. RUN uses pip to install the required dependencies from the requirements file.
4. WORKDIR sets the working directory (all following instructions operate within this directory); you may use WORKDIR as often as you like.
5. ENV sets environment variable `FLASK_APP` to the run.py file. This flask cli the file to run.
6. ENV sets environment variable `FLASK_ENV` to development. This tells flask to run the app in a development mode.
7. EXPOSE tells docker to map port 5001 to port 5000 where our app is running.
8. CMD tells docker what should be executed to run our app. In our case it's flask run command and the specified host.

Build the Image

Now that we gave a Dockerfile, let's check it builds correctly

```
docker build -t flask-starter-app .
```

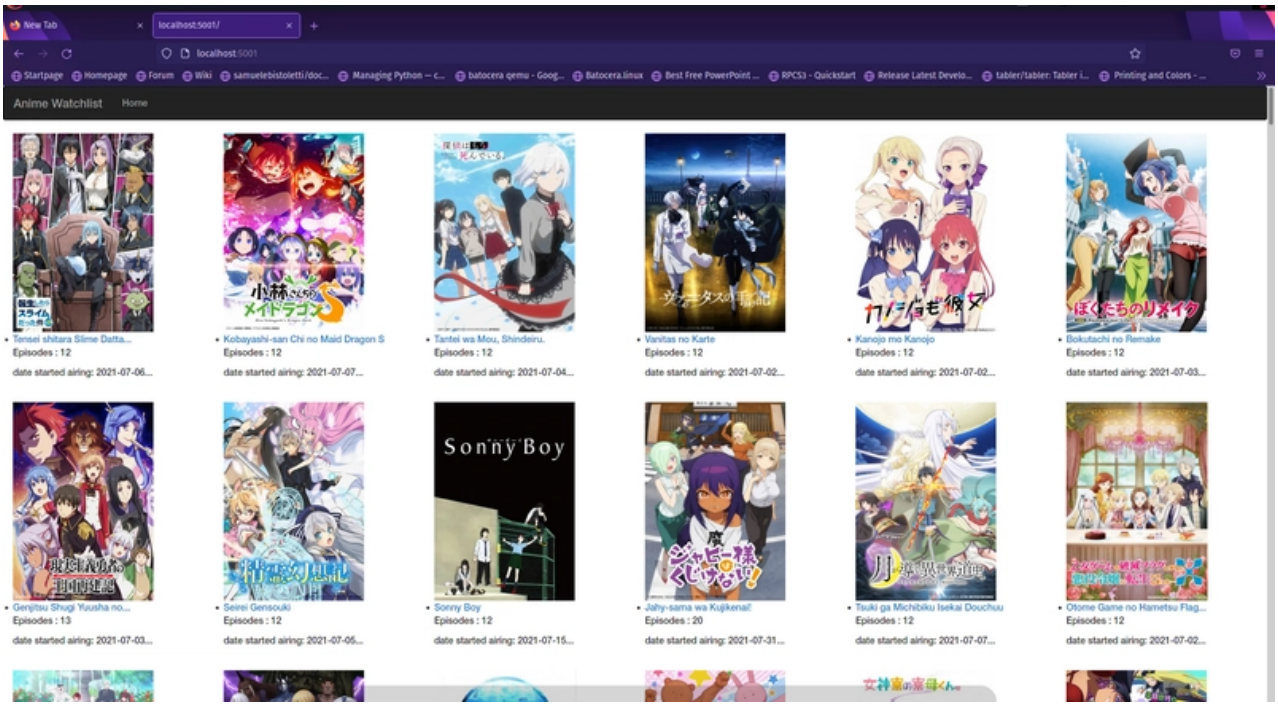


Run the Container

After the build completes, run the container

```
docker run --name=flask-app -p 5001:5000 -t -i flask-starter-app
```

Go to localhost:5001 and you should see your app running as below:



Further information

Ensure you are using the right ports. Flask by default runs on port 5000 (not 8000 like Django or 80 like Apache).
Check out Binding Docker Ports for more information.

I hope you liked this write-up and get inspired to extend it further. Keep coding! Feel free to leave comments below or
reach out on Twitter: Ken_Mwaura1.

Buy me a coffee

Subscribe