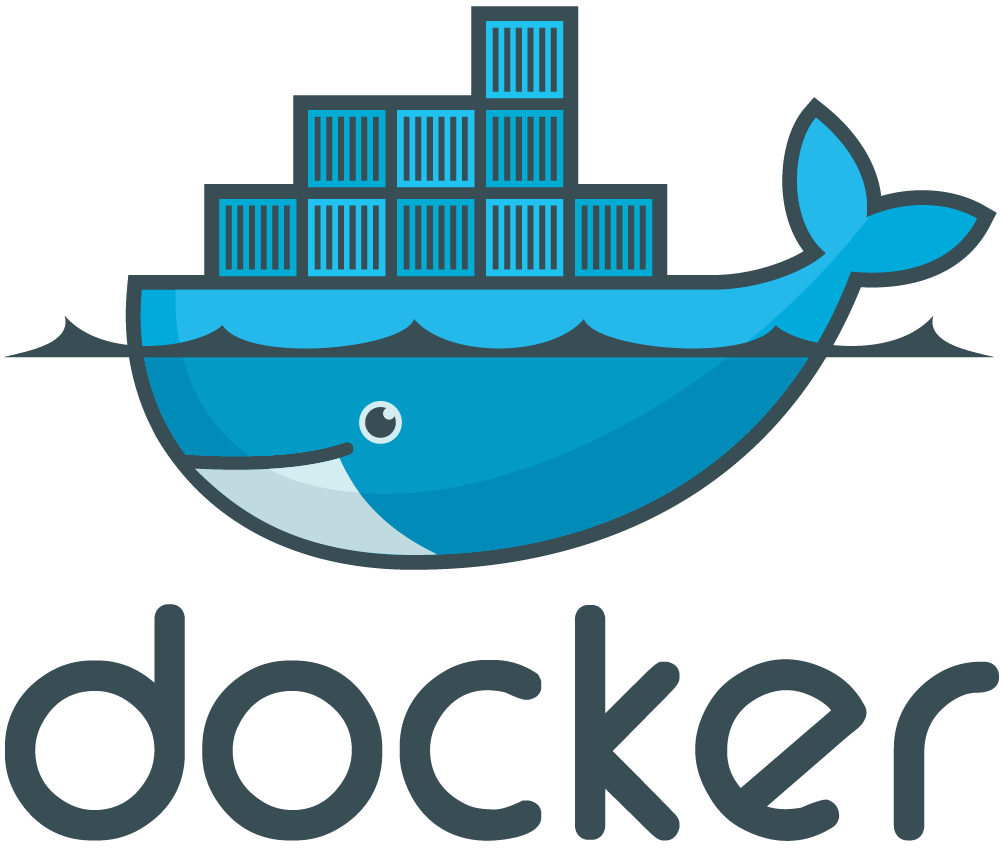
Dockerizing a Python 3 Flask App Line-by-Line

Zach Bloomquist Follow
Jul 20, 2018 · 5 min read

If you're like me, you end up writing a lot of Flask apps for random web applications and APIs. When you're done building these Flask apps, typically, you need to deploy them to production somehow. Unless you're using a container service like Heroku, deployment is synonymous with SSHing to a live server, installing your dependencies, and starting your application from source.



Docker logo. Because, containers, get it?

Docker makes the process of deployment easier by containerizing your app and automatically installing dependencies from a simple declarative config called the Dockerfile. In your Dockerfile, you can specify the operating system to run on, the `apt-get` or `yum` commands to run before your application starts, or any other logic you can think of. Docker will generate an image from this and handle running your application using that image inside a container.

In this guide, we'll walk through the process of writing a Dockerfile for a modern Flask app. By the time we're done, we'll have a nice Ubuntu + nginx + uwsgi + Flask stack all working.

. . .

Requirements

Before we begin, ensure that you have Docker Community Edition installed. Instructions vary by operating system but it is available for Windows, OS X, and Linux.

Also, you'll need a Python 3 Flask application to deploy. Make sure that in your `app.py`, any `app.run()` calls are wrapped in an `if __name__ == '__main__'` check so that uwsgi does not accidentally spawn 2 copies of your server.

. . .

The Dockerfile

First, create a file named `Dockerfile` (no extension) in the root of your Flask application and open it in your favorite text editor.

Docker uses a very simple, declarative language to define the build process. Let's start off by identifying the operating system we'd like to use:

```
FROM ubuntu:18.10
```

This tells Docker to fetch the Ubuntu 18.10 Cosmic Cuttlefish disk image from the Docker official repository and use it as the base OS for this container. Next, let's add some information about the maintainer of this package:

```
LABEL maintainer="Zach Bloomquist <zach@bloomqu.ist>"
```

Docker doesn't use this information for anything except for setting the author field of the created image. It's also nice to leave a breadcrumb for developers who may come after you. Next:

```
RUN apt-get update
RUN apt-get install -y python3 python3-dev python3-pip nginx
RUN pip3 install uwsgi
```

`RUN` commands are executed while building the Docker image. These will update apt's package index and then fetch our dependencies. The package `python3-dev` may stand out to you — this package is required for `uwsgi` to build when we install it with `pip` .

```
COPY ./ ./app
WORKDIR ./app
```

The `COPY` command copies files from the source's filesystem to the container's filesystem. These commands copy over the application's source code to a new folder and `cd` into it for the rest of the build. Next:

```
RUN pip3 install -r requirements.txt
```

This installs the requirements for your Python 3 app to execute, assuming you list your dependencies in `requirements.txt`.

```
COPY ./nginx.conf /etc/nginx/sites-enabled/default
```

This command sets up the configuration for nginx inside the container by overwriting Ubuntu's default. It assumes you have a proper nginx configuration at `./nginx.conf` containing something like this:

```
server {
  location @flask {
    include uwsgi_params;
    uwsgi_pass unix://tmp/uwsgi.sock;
  }
  location / {
    try_files @flask;
  }
}
```

This example config will pass all requests to the container's port 80 to the `uwsgi` application listening on that socket.

Now, back to the Dockerfile:

```
CMD service nginx start && uwsgi -s /tmp/uwsgi.sock --chmod-
socket=666 --manage-script-name --mount /=app:app
```

The `CMD` command tells Docker what command to execute when someone runs the image our Dockerfile creates. In this case, we want to start nginx, then start up uwsgi to back it.

. . .

Building & Running

Now that our Dockerfile is created, let's build the image from the current directory:

```
docker build -t my-image-name .
```

You can watch Docker chew through installing Ubuntu, installing the system packages, and installing our pip requirements. Once that's done, you're ready to boot up a new container based off your image:

```
docker run -d -p 1337:80 my-image-name
```

The `-d` option tells Docker to run the container in the background and print out the container ID. `-p 1337:80` maps port 1337 on the host machine to port 80 in the container.

Now that your container is running and port 1337 is mapped, you should be able to visit `http://localhost:1337/` and see your Flask application running. Woohoo!

You can use the container ID printed out by `docker run -d` to manage the life cycle of your app. Here are some good CLI commands to know:

- `docker image ls` — list available images
- `docker container ls` — list all containers
- `docker logs <partial container ID>` — tail logs from a container
- `docker kill <partial container ID>` — kill execution of a container
- `docker restart <partial container ID>` — restart container
- `docker start <partial container ID>` — start stopped container
- `docker stop <partial container ID>` — gracefully end container
- `docker container prune` — delete all non-running containers

I hope this guide was helpful to you in Dockerizing your Flask application. If you have any comments or questions, please leave them below.

. . .

Appendix: Next Steps

If you really want to turbo-charge your Dockerization, try these tips:

- Make use of named containers so you don't have to keep track of container IDs. Pass the `--name my-awesome-name` to `docker run` to name your new containers, then you can do things like `docker restart my-awesome-name`.
- Create shell script helpers to help your users interact with the Dockerized app. For example, you might create a shell script called `install.sh` that ensures Docker is installed, then builds the Docker image and ensures it runs on startup.
- Use Docker Compose to further abstract your deployment. Docker Compose allows you to orchestrate multiple containers for one deployment. In this example, we might've used a container just running `nginx` to proxy requests to a separate container running our Flask app.

Read the second part of this article, ***"Docker-composing" a Python 3 Flask App Line-by-Line***, next!

. . .

How'd you like this article? If you liked it or learned something, please leave a clap! BitCraft is a software development group and we're always taking on new clients. Reach out to us at hello@bitcraft.io or visit our website at bitcraft.io!

Thanks to Luis Ferrer-Labarca.

Docker Flask Python Python Programming DevOps