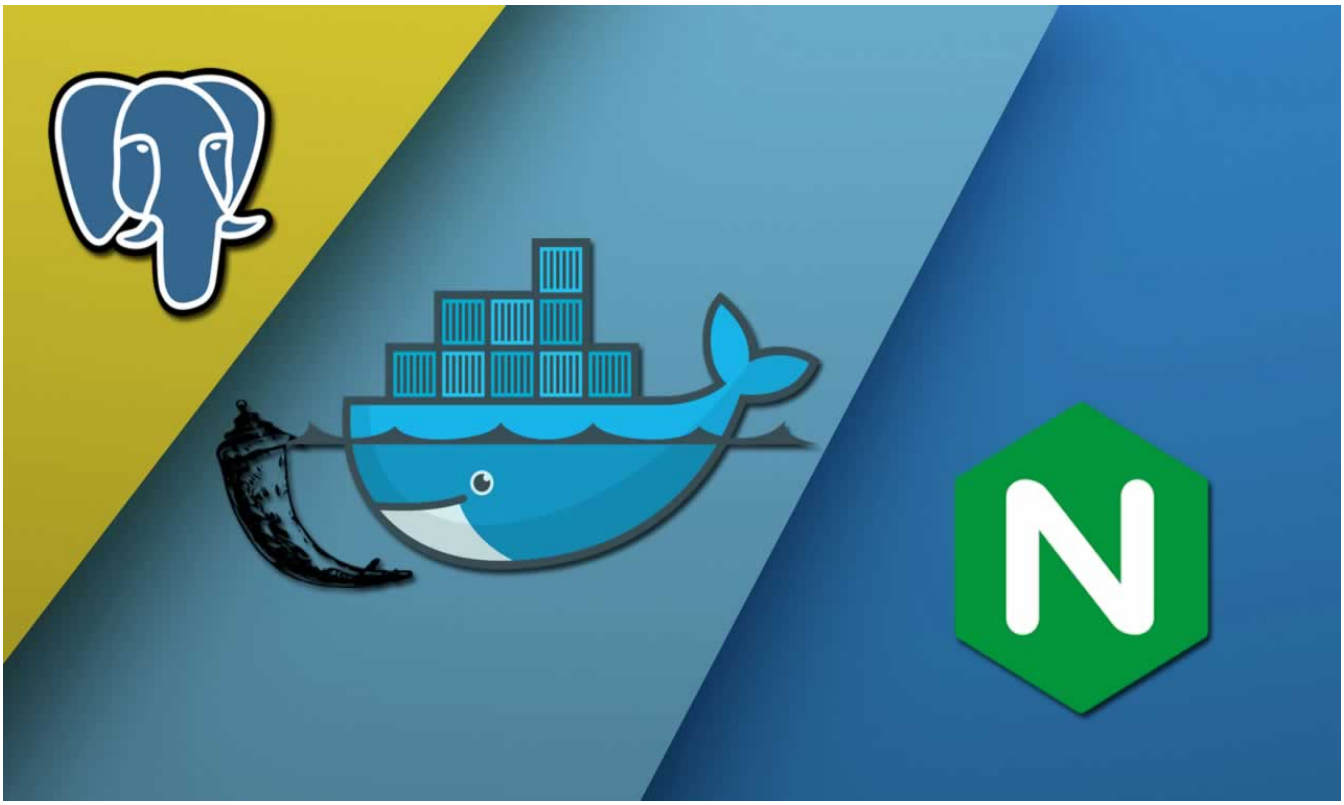

How to configure Flask to run on Docker with Postgres, Gunicorn, Nginx

This is a step-by-step tutorial that details how to configure Flask to run on Docker with Postgres. For production environments, we'll add on Nginx and Gunicorn. We'll also take a look at how to serve static and user-uploaded media files via Nginx.

Dependencies:

1. Flask v1.1.1
2. Docker v19.03.2
3. Python v3.8.0

Contents

Project Setup

Create a new project directory and install Flask:

```
$ mkdir flask-on-docker && cd flask-on-docker
$ mkdir services && cd services
$ mkdir web && cd web
$ mkdir project
$ python3.8 -m venv env
$ source env/bin/activate
(env)$ pip install flask==1.1.1
```

Next, let's create a new Flask app.

Add an **init**.py file to the "project" directory and configure the first route:

```
from flask import Flask, jsonify



app = Flask(__name__)



@app.route("/")
def hello_world():
    return jsonify(hello="world")
```

Then, to configure the Flask CLI tool to run and manage the app from the command line, add a manage.py file to the "web" directory:

```
from flask.cli import FlaskGroup

from project import app
```

```
cli = FlaskGroup(app)



if __name__ == "__main__":
    cli()
```

Here, we created a new `FlaskGroup` instance to extend the normal CLI with commands related to the Flask app.

Run the server from the "web" directory:

```
(env)$ export FLASK_APP=project/__init__.py
(env)$ python manage.py run
```

Navigate to http://localhost:5000/. You should see:

```
{
  "hello": "world"
}
```

Kill the server and exit from the virtual environment once done.

Create a requirements.txt file in the "web" directory and add Flask as a dependency:

```
Flask==1.1.1
```

Your project structure should look like:

```
└── services
    └── web
        ├── manage.py
        ├── project
        │   └── __init__.py
        └── requirements.txt
```

Docker

Install Docker, if you don't already have it, then add a Dockerfile to the "web" directory:

```
# pull official base image
FROM python:3.8.0-alpine
```

```
# set work directory
WORKDIR /usr/src/app

# set environment variables
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# install dependencies
RUN pip install --upgrade pip
COPY ./requirements.txt /usr/src/app/requirements.txt
RUN pip install -r requirements.txt

# copy project
COPY . /usr/src/app/
```

So, we started with an Alpine-based Docker image for Python 3.8.0. We then set a working directory along with two environment variables:

1. PYTHONDONTWRITEBYTECODE: Prevents Python from writing pyc files to disc (equivalent to python -B option)
2. PYTHONUNBUFFERED: Prevents Python from buffering stdout and stderr (equivalent to python -u option)

Finally, we updated Pip, copied over the requirements.txt file, installed the dependencies, and copied over the Flask app itself.

Next, add a docker-compose.yml file to the project root:

```
version: '3.7'

services:
  web:
    build: ./services/web
    command: python manage.py run -h 0.0.0.0
    volumes:
      - ./services/web/:/usr/src/app/
```

```
    ports:
      - 5000:5000
    env_file:
      - ./.env.dev
```

> *Review the Compose file reference for info on how this file works.*

Then, create a .env.dev file in the project root to store environment variables for development:

```
FLASK_APP=project/__init__.py
FLASK_ENV=development
```

Build the image:

```
$ docker-compose build
```

Once the image is built, run the container:

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

Navigate to http://localhost:5000/ to again view the hello world sanity check.

> *Check for errors in the logs if this doesn't work via `docker-compose logs -f`.*

Postgres

To configure Postgres, we need to add a new service to the docker-compose.ymlfile, set up Flask-SQLAlchemy, and install Psycopg2.

First, add a new service called `db` to docker-compose.yml:

```
version: '3.7'


services:
  web:
    build: ./services/web
    command: python manage.py run -h 0.0.0.0
    volumes:
      - ./services/web/:/usr/src/app/
    ports:
      - 5000:5000
    env_file:
```

```
    - ./.env.dev
  depends_on:
    - db
 db:
  image: postgres:12.0-alpine
  volumes:
    - postgres_data:/var/lib/postgresql/data/
  environment:
    - POSTGRES_USER=hello_flask
    - POSTGRES_PASSWORD=hello_flask
    - POSTGRES_DB=hello_flask_dev


volumes:
 postgres_data:
```

To persist the data beyond the life of the container we configured a volume. This config will bind `postgres_data` to the "/var/lib/postgresql/data/" directory in the container.

We also added an environment key to define a name for the default database and set a username and password.

> *Review the "Environment Variables" section of the Postgres Docker Hub page for more info.*

Add a `DATABASE_URL` environment variable to .env.dev as well:

```
FLASK_APP=project/__init__.py
FLASK_ENV=development
DATABASE_URL=postgresql://hello_flask:hello_flask@db:5432/hello_flask_dev
```

Then, add a new file called config.py to the "project" directory, where we'll define environment-specific configuration variables:

```
import os



basedir = os.path.abspath(os.path.dirname(__file__))



class Config(object):
```

```
    SQLALCHEMY_DATABASE_URI = os.getenv("DATABASE_URL", "sqlite://")
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

Here, the database is configured based on the `DATABASE_URL` environment variable that we just defined. Take note of the default value.

Update **init**.py to pull in the config on init:

```
from flask import Flask, jsonify



app = Flask(__name__)
app.config.from_object("project.config.Config")



@app.route("/")
def hello_world():
    return jsonify(hello="world")
```

Update the Dockerfile to install the appropriate packages required for Psycopg2:

```
# pull official base image
FROM python:3.8.0-alpine

# set work directory
WORKDIR /usr/src/app

# set environment variables
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# install psycopg2 dependencies
RUN apk update \
    && apk add postgresql-dev gcc python3-dev musl-dev

# install dependencies
RUN pip install --upgrade pip
```

```
COPY ./requirements.txt /usr/src/app/requirements.txt
RUN pip install -r requirements.txt

# copy project
COPY . /usr/src/app/
```

Add Flask-SQLAlchemy and Psycopg2 to requirements.txt:

```
Flask==1.1.1
Flask-SQLAlchemy==2.4.1
psycopg2-binary==2.8.3
```

> *Review this GitHub Issue for more info on installing Psycopg2 in an Alpine-based Docker Image.*

Update **init**.py again to create a new SQLAlchemy instance and define a database model:

```
from flask import Flask, jsonify
from flask_sqlalchemy import SQLAlchemy


app = Flask(__name__)
app.config.from_object("project.config.Config")
db = SQLAlchemy(app)


class User(db.Model):
    __tablename__ = "users"

    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(128), unique=True, nullable=False)
    active = db.Column(db.Boolean(), default=True, nullable=False)

    def __init__(self, email):
        self.email = email


@app.route("/")
```

```
def hello_world():
    return jsonify(hello="world")
```

Finally, update manage.py:

```
from flask.cli import FlaskGroup

from project import app, db


cli = FlaskGroup(app)


@cli.command("create_db")
def create_db():
    db.drop_all()
    db.create_all()
    db.session.commit()


if __name__ == "__main__":
    cli()
```

This registers a new command, `create_db`, to the CLI so that we can run it from the command line, which we'll use shortly to apply the model to the database.

Build the new image and spin up the two containers:

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

Create the table:

```
$ docker-compose exec web python manage.py create_db
```

> *Get the following error?*

```
sqlalchemy.exc.OperationalError: (psycopg2.OperationalError)
FATAL:  database "hello_flask_dev" does not exist
```

> *Run `docker-compose down -v` to remove the volumes along with the containers. Then, re-build the images, run the containers, and apply the migrations.*

Ensure the `users` table was created:

```
$ docker-compose exec db psql --username=hello_flask --dbname=hello_flask_dev

psql (12.0)
Type "help" for help.

hello_flask_dev=# \l
                                 List of databases
     Name      |    Owner    | Encoding |  Collate   |   Ctype    |     Access privileges
---------------+-------------+----------+------------+------------+---------------------------
 hello_flask_dev | hello_flask | UTF8     | en_US.utf8 | en_US.utf8 |
 postgres        | hello_flask | UTF8     | en_US.utf8 | en_US.utf8 |
 template0       | hello_flask | UTF8     | en_US.utf8 | en_US.utf8 | =c/hello_flask           +
                 |             |          |            |            | hello_flask=CTc/hello_flask
 template1       | hello_flask | UTF8     | en_US.utf8 | en_US.utf8 | =c/hello_flask           +
                 |             |          |            |            | hello_flask=CTc/hello_flask
(4 rows)

hello_flask_dev=# \c hello_flask_dev
You are now connected to database "hello_flask_dev" as user "hello_flask".

hello_flask_dev=# \dt
         List of relations
 Schema | Name | Type  |    Owner
--------+------+-------+-------------
 public | user | table | hello_flask
(1 row)

hello_flask_dev=# \q
```

You can check that the volume was created as well by running:

```
$ docker volume inspect flask-on-docker_postgres_data
```

You should see something similar to:

```
[
    {
        "CreatedAt": "2019-10-22T14:43:18Z",
        "Driver": "local",
        "Labels": {
            "com.docker.compose.project": "flask-on-docker",
            "com.docker.compose.version": "1.24.1",
            "com.docker.compose.volume": "postgres_data"
        },
        "Mountpoint": "/var/lib/docker/volumes/flask-on-docker_postgres_data/_data",
        "Name": "flask-on-docker_postgres_data",
        "Options": null,
        "Scope": "local"
    }
]
```

Next, add an entrypoint.sh file to the "web" directory to verify that Postgres is up and healthy before creating the database table and running the Flask development server:

```
#!/bin/sh

if [ "$DATABASE" = "postgres" ]
then
    echo "Waiting for postgres..."

    while ! nc -z $SQL_HOST $SQL_PORT; do
      sleep 0.1
    done

    echo "PostgreSQL started"
fi

python manage.py create_db
```

```
exec "$@"
```

Take note of the environment variables.

Update the file permissions locally:

```
$ chmod +x services/web/entrypoint.sh
```

Then, update the Dockerfile to copy over the entrypoint.sh file and run it as the Docker entrypoint command:

```
# pull official base image
FROM python:3.8.0-alpine

# set work directory
WORKDIR /usr/src/app

# set environment variables
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# install psycopg2 dependencies
RUN apk update \
    && apk add postgresql-dev gcc python3-dev musl-dev

# install dependencies
RUN pip install --upgrade pip
COPY ./requirements.txt /usr/src/app/requirements.txt
RUN pip install -r requirements.txt

# copy project
COPY . /usr/src/app/

# run entrypoint.sh
ENTRYPOINT ["/usr/src/app/entrypoint.sh"]
```

Add the SQL_HOST, SQL_PORT, and DATABASE environment variables, for the entrypoint.sh script, to .env.dev:

```
FLASK_APP=project/__init__.py
FLASK_ENV=development
DATABASE_URL=postgresql://hello_flask:hello_flask@db:5432/hello_flask_dev
SQL_HOST=db
SQL_PORT=5432
DATABASE=postgres
```

Test it out again:

1. Re-build the images
2. Run the containers
3. Try http://localhost:5000/

Let's also had a CLI seed command for adding sample users to the userstable in manage.py:

```
from flask.cli import FlaskGroup

from project import app, db, User


cli = FlaskGroup(app)


@cli.command("create_db")
def create_db():
    db.drop_all()
    db.create_all()
    db.session.commit()


@cli.command("seed_db")
def seed_db():
    db.session.add(User(email="michael@mherman.org"))
```

```
    db.session.commit()


if __name__ == "__main__":
    cli()
```

Try it out:

```
$ docker-compose exec web python manage.py seed_db

$ docker-compose exec db psql --username=hello_flask --dbname=hello_flask_dev

psql (12.0)
Type "help" for help.

hello_flask_dev=# \c hello_flask_dev
You are now connected to database "hello_flask_dev" as user "hello_flask".

hello_flask_dev=# select * from users;
 id |        email        | active
----+---------------------+--------
  1 | michael@mherman.org | t
(1 row)

hello_flask_dev=# \q
```

> *Despite adding Postgres, we can still create an independent Docker image for Flask by not setting the `DATABASE_URL` environment variable. To test, build a new image and then run a new container:*

```
$ docker build -f ./services/web/Dockerfile -t hello_flask:latest ./services/web
$ docker run -p 5001:5000 \
    -e "FLASK_APP=project/__init__.py" -e "FLASK_ENV=development" \
    hello_flask python /usr/src/app/manage.py run -h 0.0.0.0
```

> *You should be able to view the hello world sanity check at http://localhost:5001.*

Gunicorn

Moving along, for production environments, let's add Gunicorn, a production-grade WSGI server, to the requirements file:

```
Flask==1.1.1
Flask-SQLAlchemy==2.4.1
gunicorn==19.9.0
psycopg2-binary==2.8.3
```

Since we still want to use Flask's built-in server in development, create a new compose file called docker-compose.prod.yml for production:

```
version: '3.7'

services:
  web:
    build: ./services/web
    command: gunicorn --bind 0.0.0.0:5000 manage:app
    ports:
      - 5000:5000
    env_file:
      - ./.env.prod
    depends_on:
      - db
  db:
    image: postgres:12.0-alpine
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    env_file:
      - ./.env.prod.db

volumes:
  postgres_data:
```

> *If you have multiple environments, you may want to look at using a docker-compose.override.yml configuration file. With this approach, you'd add your base config to a docker-compose.yml file and then use a docker-compose.override.yml file to override those config settings based on the environment.*

Take note of the default `command`. We're running Gunicorn rather than the Flask development server. We also removed the volume from the `web` service since we don't need it in production. Finally, we're using separate environment variable files to define environment variables for both services that will be passed to the container at runtime.

.env.prod:

```
FLASK_APP=project/__init__.py
FLASK_ENV=production
DATABASE_URL=postgresql://hello_flask:hello_flask@db:5432/hello_flask_prod
SQL_HOST=db
SQL_PORT=5432
DATABASE=postgres
```

.env.prod.db:

```
POSTGRES_USER=hello_flask
POSTGRES_PASSWORD=hello_flask
POSTGRES_DB=hello_flask_prod
```

Add the two files to the project root. You'll probably want to keep them out of version control, so add them to a .gitignore file.

Bring down the development containers (and the associated volumes with the `-v` flag):

```
$ docker-compose down -v
```

Then, build the production images and spin up the containers:

```
$ docker-compose -f docker-compose.prod.yml up -d --build
```

Verify that the `hello_flask_prod` database was created along with the `users` table. Test out http://localhost:5000/.

> *Again, if the container fails to start, check for errors in the logs via `docker-compose -f docker-compose.prod.yml logs -f`.*

Production Dockerfile

Did you notice that we're still running the `create_db` command, which drops all existing tables and then creates the tables from the models, every time the container is run? This is fine in development, but let's create a new entrypoint file for production.

entrypoint.prod.sh:

```
#!/bin/sh

if [ "$DATABASE" = "postgres" ]
then
    echo "Waiting for postgres..."

    while ! nc -z $SQL_HOST $SQL_PORT; do
      sleep 0.1
    done

    echo "PostgreSQL started"
fi

exec "$@"
```

> *Alternatively, instead of creating a new entrypoint file, you could alter the existing one like so:*

```
#!/bin/sh

if [ "$DATABASE" = "postgres" ]
then
    echo "Waiting for postgres..."

    while ! nc -z $SQL_HOST $SQL_PORT; do
      sleep 0.1
    done

    echo "PostgreSQL started"
fi
```

```
if [ "$FLASK_ENV" = "development" ]
then
    echo "Creating the database tables..."
    python manage.py create_db
    echo "Tables created"
fi

exec "$@"
```

Update the file permissions locally:

```
$ chmod +x services/web/entrypoint.prod.sh
```

To use this file, create a new Dockerfile called Dockerfile.prod for use with production builds:

```
###########
# BUILDER #
###########

# pull official base image
FROM python:3.8.0-alpine as builder

# set work directory
WORKDIR /usr/src/app

# set environment variables
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# install psycopg2 dependencies
RUN apk update \
    && apk add postgresql-dev gcc python3-dev musl-dev

# lint
RUN pip install --upgrade pip
```

```
RUN pip install flake8
COPY . /usr/src/app/
RUN flake8 --ignore=E501,F401 .

# install dependencies
COPY ./requirements.txt .
RUN pip wheel --no-cache-dir --no-deps --wheel-dir /usr/src/app/wheels -r requirements.txt



#########
# FINAL #
#########

# pull official base image
FROM python:3.8.0-alpine

# create directory for the app user
RUN mkdir -p /home/app

# create the app user
RUN addgroup -S app && adduser -S app -G app

# create the appropriate directories
ENV HOME=/home/app
ENV APP_HOME=/home/app/web
RUN mkdir $APP_HOME
WORKDIR $APP_HOME

# install dependencies
RUN apk update && apk add libpq
COPY --from=builder /usr/src/app/wheels /wheels
COPY --from=builder /usr/src/app/requirements.txt .
RUN pip install --upgrade pip
RUN pip install --no-cache /wheels/*
```

```
# copy entrypoint-prod.sh
COPY ./entrypoint.prod.sh $APP_HOME

# copy project
COPY . $APP_HOME

# chown all the files to the app user
RUN chown -R app:app $APP_HOME

# change to the app user
USER app

# run entrypoint.prod.sh
ENTRYPOINT ["/home/app/web/entrypoint.prod.sh"]
```

Here, we used a Docker multi-stage build to reduce the final image size. Essentially, `builder` is a temporary image that's used for building the Python wheels. The wheels are then copied over to the final production image and the `builder` image is discarded.

> *You could take the multi-stage build approach a step further and use a single Dockerfile instead of creating two Dockerfiles. Think of the pros and cons of using this approach over two different files.*

Did you notice that we created a non-root user? By default, Docker runs container processes as root inside of a container. This is a bad practice since attackers can gain root access to the Docker host if they manage to break out of the container. If you're root in the container, you'll be root on the host.

Update the `web` service within the docker-compose.prod.yml file to build with Dockerfile.prod:

```
web:
  build:
    context: ./services/web
    dockerfile: Dockerfile.prod
  command: gunicorn --bind 0.0.0.0:5000 manage:app
  ports:
    - 5000:5000
  env_file:
    - ./.env.prod
```

```
  depends_on:
    - db
```

Try it out:

```
$ docker-compose -f docker-compose.prod.yml down -v
$ docker-compose -f docker-compose.prod.yml up -d --build
$ docker-compose -f docker-compose.prod.yml exec web python manage.py create_db
```

Nginx

Next, let's add Nginx into the mix to act as a reverse proxy for Gunicorn to handle client requests as well as serve up static files.

Add the service to docker-compose.prod.yml:

```
nginx:
  build: ./services/nginx
  ports:
    - 1337:80
  depends_on:
    - web
```

Then, in the "services" directory, create the following files and folders:

```
└── nginx
    ├── Dockerfile
    └── nginx.conf
```

Dockerfile:

```
FROM nginx:1.17.4-alpine

RUN rm /etc/nginx/conf.d/default.conf
COPY nginx.conf /etc/nginx/conf.d
```

nginx.conf:

```
upstream hello_flask {
    server web:5000;
}
```

```
server {

    listen 80;

    location / {
        proxy_pass http://hello_flask;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $host;
        proxy_redirect off;
    }

}
```

Then, update the `web` service, in docker-compose.prod.yml, replacing `ports`with `expose`:

```
web:
  build:
    context: ./services/web
    dockerfile: Dockerfile.prod
  command: gunicorn --bind 0.0.0.0:5000 manage:app
  expose:
    - 5000
  env_file:
    - ./.env.prod
  depends_on:
    - db
```

Now, port 5000 is only exposed internally, to other Docker services. The port will no longer be published to the host machine.

> *For more on ports vs expose, review this Stack Overflow question.*

Test it out again:

```
$ docker-compose -f docker-compose.prod.yml down -v
$ docker-compose -f docker-compose.prod.yml up -d --build
$ docker-compose -f docker-compose.prod.yml exec web python manage.py create_db
```

Ensure the app is up and running at http://localhost:1337.

Your project structure should now look like:

```
├── .env.dev
├── .env.prod
├── .env.prod.db
├── .gitignore
├── docker-compose.prod.yml
├── docker-compose.yml
└── services
    ├── nginx
    │   ├── Dockerfile
    │   └── nginx.conf
    └── web
        ├── Dockerfile
        ├── Dockerfile.prod
        ├── entrypoint.prod.sh
        ├── entrypoint.sh
        ├── manage.py
        ├── project
        │   ├── __init__.py
        │   └── config.py
        └── requirements.txt
```

Bring the containers down once done:

```
$ docker-compose -f docker-compose.prod.yml down -v
```

Since Gunicorn is an application server, it will not serve up static files. So, how should both static and media files be handled in this particular configuration?

Static Files

Start by creating the following files and folders in the "services/web/project" folder:

```
└── static
    └── hello.txt
```

Add some text to hello.txt:

```
hi!
```

Add a new route handler to **init**.py:

```
@app.route("/static/<path:filename>")
def staticfiles(filename):
    return send_from_directory(app.config["STATIC_FOLDER"], filename)
```

Don't forget to import send_from_directory:

```
from flask import Flask, jsonify, send_from_directory
```

Finally, add the STATIC_FOLDER config to services/web/project/config.py

```
import os



basedir = os.path.abspath(os.path.dirname(__file__))



class Config(object):
    SQLALCHEMY_DATABASE_URI = os.getenv("DATABASE_URL", "sqlite://")
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    STATIC_FOLDER = f"{os.getenv('APP_FOLDER')}/project/static"
```

Development

Add the APP_FOLDER environment variable to .env.dev:

```
FLASK_APP=project/__init__.py
FLASK_ENV=development
DATABASE_URL=postgresql://hello_flask:hello_flask@db:5432/hello_flask_dev
SQL_HOST=db
SQL_PORT=5432
DATABASE=postgres
APP_FOLDER=/usr/src/app
```

To test, first re-build the images and spin up the new containers per usual. Once done, ensure http://localhost:5000/static/hello.txt serves up the file correctly.

Production

For production, add a volume to the `web` and `nginx` services in docker-compose.prod.yml so that each container will share a directory named "static":

```
version: '3.7'

services:
  web:
    build:
      context: ./services/web
      dockerfile: Dockerfile.prod
    command: gunicorn --bind 0.0.0.0:5000 manage:app
    volumes:
      - static_volume:/home/app/web/project/static
    expose:
      - 5000
    env_file:
      - ./.env.prod
    depends_on:
      - db
  db:
    image: postgres:12.0-alpine
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    env_file:
      - ./.env.prod.db
  nginx:
    build: ./services/nginx
    volumes:
      - static_volume:/home/app/web/project/static
    ports:
      - 1337:80
    depends_on:
      - web
```

```
volumes:
  postgres_data:
  static_volume:
```

Next, update the Nginx configuration to route static file requests to the "static" folder:

```
upstream hello_flask {
    server web:5000;
}

server {

    listen 80;

    location / {
        proxy_pass http://hello_flask;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $host;
        proxy_redirect off;
    }

    location /static/ {
        alias /home/app/web/project/static/;
    }

}
```

Add the `APP_FOLDER` environment variable to .env.prod:

```
FLASK_APP=project/__init__.py
FLASK_ENV=production
DATABASE_URL=postgresql://hello_flask:hello_flask@db:5432/hello_flask_prod
SQL_HOST=db
SQL_PORT=5432
DATABASE=postgres
APP_FOLDER=/home/app/web
```

> *Where does this directory path come from? Compare this path to the path added to .env.dev. Why do they do they differ?*

Spin down the development containers:

```
$ docker-compose down -v
```

Test:

```
$ docker-compose -f docker-compose.prod.yml up -d --build
```

Again, requests to `http://localhost:1337/static/*` will be served from the "static" directory.

Navigate to http://localhost:1337/static/hello.txt and ensure the static asset is loaded correctly.

You can also verify in the logs -- via `docker-compose -f docker-compose.prod.yml logs -f` -- that requests to the static files are served up successfully via Nginx:

```
nginx_1  | 172.19.0.1 - - [24/Oct/2019:13:29:37 +0000] "GET /static/hello.txt HTTP/1.1" 304 0 "-"
        "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:70.0) Gecko/20100101 Firefox/70.0" "-"
```

Bring the containers once done:

```
$ docker-compose -f docker-compose.prod.yml down -v
```

Media Files

To test out the handling of user-uploaded media files, add two new route handlers to **init**.py:

```
@app.route("/media/<path:filename>")
def mediafiles(filename):
    return send_from_directory(app.config["MEDIA_FOLDER"], filename)



@app.route("/upload", methods=["GET", "POST"])
def upload_file():
    if request.method == "POST":
        file = request.files["file"]
        filename = secure_filename(file.filename)
        file.save(os.path.join(app.config["MEDIA_FOLDER"], filename))
    return f"""
    <!doctype html>
```

```
    <title>upload new File</title>
    <form action="" method=post enctype=multipart/form-data>
      <p><input type=file name=file><input type=submit value=Upload>
    </form>
    """
```

Update the imports as well:

```
import os

from werkzeug import secure_filename
from flask import (
    Flask,
    jsonify,
    send_from_directory,
    request,
    redirect,
    url_for
)
from flask_sqlalchemy import SQLAlchemy
```

Add the MEDIA_FOLDER config to services/web/project/config.py:

```
import os



basedir = os.path.abspath(os.path.dirname(__file__))



class Config(object):
    SQLALCHEMY_DATABASE_URI = os.getenv("DATABASE_URL", "sqlite://")
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    STATIC_FOLDER = f"{os.getenv('APP_FOLDER')}/project/static"
    MEDIA_FOLDER = f"{os.getenv('APP_FOLDER')}/project/media"
```

Finally, create a new folder called "media" in the "project" folder.

Development

Test:

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

You should be able to upload an image at http://localhost:5000/upload, and then view the image at http://localhost:5000/uploads/IMAGE_FILE_NAME.

Production

For production, add another volume to the `web` and `nginx` services:

```
version: '3.7'


services:
  web:
    build:
      context: ./services/web
      dockerfile: Dockerfile.prod
    command: gunicorn --bind 0.0.0.0:5000 manage:app
    volumes:
      - static_volume:/home/app/web/project/static
      - media_volume:/home/app/web/project/media
    expose:
      - 5000
    env_file:
      - ./.env.prod
    depends_on:
      - db
  db:
    image: postgres:12.0-alpine
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    env_file:
      - ./.env.prod.db
  nginx:
    build: ./services/nginx
```

```
    volumes:
      - static_volume:/home/app/web/project/static
      - media_volume:/home/app/web/project/media
    ports:
      - 1337:80
    depends_on:
      - web

volumes:
  postgres_data:
  static_volume:
  media_volume:
```

Next, update the Nginx configuration to route media file requests to the "media" folder:

```
upstream hello_flask {
    server web:5000;
}

server {

    listen 80;

    location / {
        proxy_pass http://hello_flask;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $host;
        proxy_redirect off;
    }

    location /static/ {
        alias /home/app/web/project/static/;
    }

    location /media/ {
```

```
        alias /home/app/web/project/media/;
    }


}
```

Re-build:

```
$ docker-compose down -v


$ docker-compose -f docker-compose.prod.yml up -d --build
$ docker-compose -f docker-compose.prod.yml exec web python manage.py create_db
```

Test it out one final time:

1. Upload an image at http://localhost:1337/upload.
2. Then, view the image at http://localhost:1337/media/IMAGE_FILE_NAME.

Conclusion

In this tutorial, we walked through how to containerize a Flask application with Postgres for development. We also created a production-ready Docker Compose file that adds Gunicorn and Nginx into the mix to handle static and media files. You can now test out a production setup locally.

In terms of actual deployment to a production environment, you'll probably want to use a:

1. Fully managed database service -- like RDS or Cloud SQL -- rather than managing your own Postgres instance within a container.
2. Non-root user for the `db` and `nginx` services

You can find the code in the flask-on-docker repo.

Thanks for reading!