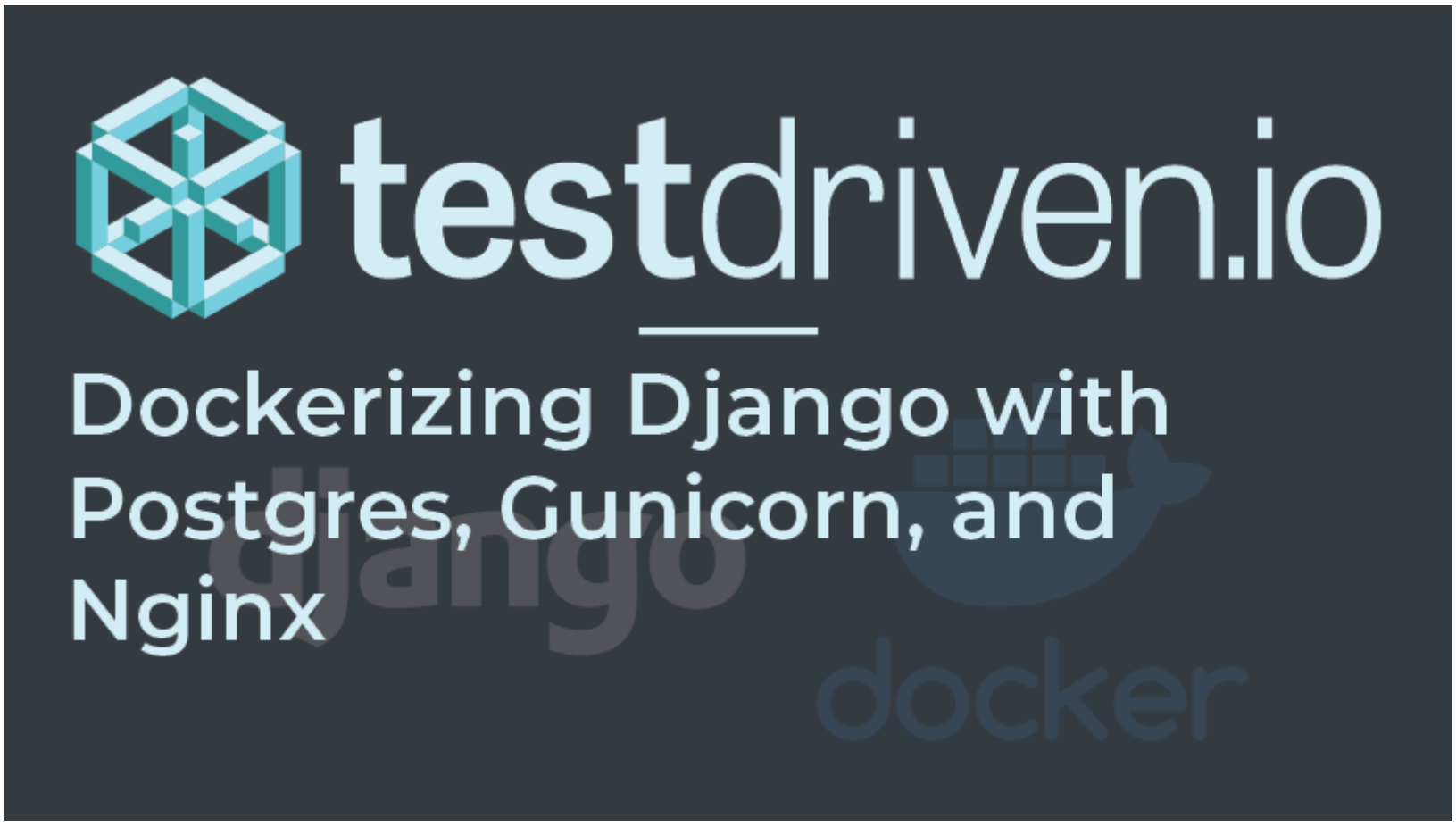

Dockerizing Django with Postgres, Gunicorn, and Nginx

Posted by Michael Herman | Last updated on October 21st, 2019 | django docker

This is a step-by-step tutorial that details how to configure Django 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 Django static and media files via Nginx.

Dependencies:

1. Django v2.2.6
2. Docker v19.03.2
3. Python v3.8.0

Project Setup

Create a new project directory along with a new Django project:

```
$ mkdir django-on-docker && cd django-on-docker
$ mkdir app && cd app
$ python3.8 -m venv env
$ source env/bin/activate
(env)$ pip install django==2.2.6
(env)$ django-admin.py startproject hello_django .
(env)$ python manage.py migrate
(env)$ python manage.py runserver
```

> Feel free to swap out virtualenv and Pip for Pipenv if that's your tool of choice.

Navigate to http://localhost:8000/ to view the Django welcome screen. Kill the server and exit from the virtual environment once done. We now have a simple Django project to work with.

Create a *requirements.txt* file in the "app" directory and add Django as a dependency:

```
Django==2.2.6
```

Feedback

Since we'll be moving to Postgres, go ahead and remove the *db.sqlite3* file from the "app" directory.

Your project directory should look like:

```
└── app
    ├── hello_django
    |   ├── __init__.py
    |   ├── settings.py
    |   ├── urls.py
    |   └── wsgi.py
    ├── manage.py
    └── requirements.txt
```

Docker

Install Docker, if you don't already have it, then add a *Dockerfile* to the "app" 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 Django project itself.

> Review Docker for Python Developers for more on structuring Dockerfiles as well as some best practices for configuring Docker for Python-based development.

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

```
version: '3.7'

services:
  web:
    build: ./app
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - ./app/:/usr/src/app/
    ports:
      - 8000:8000
    env_file:
      - ./.env.dev
```

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

Update the `SECRET_KEY`, `DEBUG`, and `ALLOWED_HOSTS` variables in *settings.py*:

Feedback

```
SECRET_KEY = os.environ.get("SECRET_KEY")

DEBUG = int(os.environ.get("DEBUG", default=0))

# 'DJANGO_ALLOWED_HOSTS' should be a single string of hosts with a space between each.
# For example: 'DJANGO_ALLOWED_HOSTS=localhost 127.0.0.1 [::1]'
ALLOWED_HOSTS = os.environ.get("DJANGO_ALLOWED_HOSTS").split(" ")
```

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

```
DEBUG=1
SECRET_KEY=foo
DJANGO_ALLOWED_HOSTS=localhost 127.0.0.1 [::1]
```

Build the image:

```
$ docker-compose build
```

Once the image is built, run the container:

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

Navigate to http://localhost:8000/ to again view the welcome screen.

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

Postgres

To configure Postgres, we'll need to add a new service to the *docker-compose.yml* file, update the Django settings, and install Psycopg2.

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

```
version: '3.7'

services:
  web:
    build: ./app
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - ./app/:/usr/src/app/
    ports:
      - 8000:8000
    env_file:
      - ./.env.dev
    depends_on:
      - db
  db:
    image: postgres:12.0-alpine
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    environment:
      - POSTGRES_USER=hello_django
      - POSTGRES_PASSWORD=hello_django
      - POSTGRES_DB=hello_django_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.

Feedback

We'll need some new environment variables for the `web` service as well, so update *.env.dev* like so:

```
DEBUG=1
SECRET_KEY=foo
DJANGO_ALLOWED_HOSTS=localhost 127.0.0.1 [::1]
SQL_ENGINE=django.db.backends.postgresql
SQL_DATABASE=hello_django_dev
SQL_USER=hello_django
SQL_PASSWORD=hello_django
SQL_HOST=db
SQL_PORT=5432
```

Update the `DATABASES` dict in *settings.py*:

```
DATABASES = {
    "default": {
        "ENGINE": os.environ.get("SQL_ENGINE", "django.db.backends.sqlite3"),
        "NAME": os.environ.get("SQL_DATABASE", os.path.join(BASE_DIR, "db.sqlite3")),
        "USER": os.environ.get("SQL_USER", "user"),
        "PASSWORD": os.environ.get("SQL_PASSWORD", "password"),
        "HOST": os.environ.get("SQL_HOST", "localhost"),
        "PORT": os.environ.get("SQL_PORT", "5432"),
    }
}
```

Here, the database is configured based on the environment variables that we just defined. Take note of the default values.

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 Psycopg2 to *requirements.txt*:

```
Django==2.2.6
psycopg2-binary==2.8.3
```

> Review [this GitHub Issue](#) for more info on installing Psycopg2 in an Alpine-based Docker Image.

Build the new image and spin up the two containers:

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

Run the migrations:

```
$ docker-compose exec web python manage.py migrate --noinput
```

Feedback

Get the following error?

```
django.db.utils.OperationalError: FATAL:  database "hello_django_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 default Django tables were created:

```
$ docker-compose exec db psql --username=hello_django --dbname=hello_django_dev

psql (12.0)
Type "help" for help.

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

hello_django_dev=# \c hello_django_dev
You are now connected to database "hello_django_dev" as user "hello_django".

hello_django_dev=# \dt
                    List of relations
 Schema |            Name             | Type  |    Owner
--------+-----------------------------+-------+--------------
 public | auth_group                  | table | hello_django
 public | auth_group_permissions      | table | hello_django
 public | auth_permission             | table | hello_django
 public | auth_user                   | table | hello_django
 public | auth_user_groups            | table | hello_django
 public | auth_user_user_permissions  | table | hello_django
 public | django_admin_log            | table | hello_django
 public | django_content_type         | table | hello_django
 public | django_migrations           | table | hello_django
 public | django_session              | table | hello_django
(10 rows)

hello_django_dev=# \q
```

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

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

You should see something similar to:

```
[
    {
        "CreatedAt": "2019-10-16T02:32:55Z",
        "Driver": "local",
        "Labels": {
            "com.docker.compose.project": "django-on-docker",
            "com.docker.compose.version": "1.24.1",
            "com.docker.compose.volume": "postgres_data"
        },
        "Mountpoint": "/var/lib/docker/volumes/django-on-docker_postgres_data/_data",
        "Name": "django-on-docker_postgres_data",
        "Options": null,
        "Scope": "local"
    }
]
```

Next, add an *entrypoint.sh* file to the "app" directory to verify that Postgres is healthy *before* applying the migrations and running the Django development server:

Feedback

```
#!/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 flush --no-input
python manage.py migrate

exec "$@"
```

Update the file permissions locally:

```
$ chmod +x app/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 entrypoint.sh
COPY ./entrypoint.sh /usr/src/app/entrypoint.sh

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

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

Add the `DATABASE` environment variable to *.env.dev*:

```
DEBUG=1
SECRET_KEY=foo
DJANGO_ALLOWED_HOSTS=localhost 127.0.0.1 [::1]
SQL_ENGINE=django.db.backends.postgresql
SQL_DATABASE=hello_django_dev
SQL_USER=hello_django
SQL_PASSWORD=hello_django
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:8000/

Feedback

Despite adding Postgres, we can still create an independent Docker image for Django as long as the `DATABASE` environment variable is not set to `postgres`. To test, build a new image and then run a new container:

```
$ docker build -f ./app/Dockerfile -t hello_django:latest ./app
$ docker run -p 8001:8000 \
    -e "SECRET_KEY=please_change_me" -e "DEBUG=1" -e "DJANGO_ALLOWED_HOSTS=*" \
    hello_django python /usr/src/app/manage.py runserver 0.0.0.0:8000
```

You should be able to view the welcome page at http://localhost:8001.

Gunicorn

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

```
Django==2.2.6
gunicorn==19.9.0
psycopg2-binary==2.8.3
```

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

```
version: '3.7'

services:
  web:
    build: ./app
    command: gunicorn hello_django.wsgi:application --bind 0.0.0.0:8000
    ports:
      - 8000:8000
    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 Django 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:

```
DEBUG=0
SECRET_KEY=change_me
DJANGO_ALLOWED_HOSTS=localhost 127.0.0.1 [::1]
SQL_ENGINE=django.db.backends.postgresql
SQL_DATABASE=hello_django_prod
SQL_USER=hello_django
SQL_PASSWORD=hello_django
SQL_HOST=db
SQL_PORT=5432
DATABASE=postgres
```

.env.prod.db:

Feedback

```
POSTGRES_USER=hello_django
POSTGRES_PASSWORD=hello_django
POSTGRES_DB=hello_django_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_django_prod` database was created along with the default Django tables. Test out the admin page at http://localhost:8000/admin. The static files are not being loaded anymore. This is expected since Debug mode is off. We'll fix this shortly.

> 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 database flush (which clears out the database) and migrate commands 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 "$@"
```

Update the file permissions locally:

```
$ chmod +x app/entrypoint.prod.sh
```

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

Feedback

```
###########
# 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.

Feedback

> 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: ./app
    dockerfile: Dockerfile.prod
  command: gunicorn hello_django.wsgi:application --bind 0.0.0.0:8000
  ports:
    - 8000:8000
  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 migrate --noinput
```

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

Then, in the local project root, 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:

Feedback

```
upstream hello_django {
    server web:8000;
}

server {

    listen 80;

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

}
```

> Review Using NGINX and NGINX Plus as an Application Gateway with uWSGI and Django for more info on configuring Nginx to work with Django.

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

```
web:
  build:
    context: ./app
    dockerfile: Dockerfile.prod
  command: gunicorn hello_django.wsgi:application --bind 0.0.0.0:8000
  expose:
    - 8000
  env_file:
    - ./.env.prod
  depends_on:
    - db
```

Now, port 8000 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 migrate --noinput
```

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
├── app
|   ├── Dockerfile
|   ├── Dockerfile.prod
|   ├── entrypoint.prod.sh
|   ├── entrypoint.sh
|   ├── hello_django
|   |   ├── __init__.py
|   |   ├── settings.py
|   |   ├── urls.py
|   |   └── wsgi.py
|   ├── manage.py
|   └── requirements.txt
├── docker-compose.prod.yml
├── docker-compose.yml
└── nginx
    ├── Dockerfile
    └── nginx.conf
```

Feedback

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

Update *settings.py*:

```
STATIC_URL = "/staticfiles/"
STATIC_ROOT = os.path.join(BASE_DIR, "staticfiles")
```

Development

Collect the static files in *entrypoint.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

python manage.py flush --no-input
python manage.py migrate
python manage.py collectstatic --no-input --clear

exec "$@"
```

Now, any request to `http://localhost:8000/staticfiles/*` will be served from the "staticfiles" directory.

To test, first re-build the images and spin up the new containers per usual. When the `collectstatic` command is run, static files will be placed in the "staticfiles" directory. Ensure static files are being served correctly at http://localhost:8000/admin.

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 "staticfiles":

```
version: '3.7'

services:
  web:
    build:
      context: ./app
      dockerfile: Dockerfile.prod
    command: gunicorn hello_django.wsgi:application --bind 0.0.0.0:8000
    volumes:
      - static_volume:/home/app/web/staticfiles
    expose:
      - 8000
    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: ./nginx
    volumes:
      - static_volume:/home/app/web/staticfiles
    ports:
      - 1337:80
    depends_on:
      - web

volumes:
  postgres_data:
  static_volume:
```

We need to also create the "/home/app/web/staticfiles" folder in *Dockerfile.prod*:

```
...

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

...
```

Why is this necessary?

Docker Compose normally mounts named volumes as root. And since we're using a non-root user, we'll get a permission denied error when the `collectstatic` command is run if the directory does not already exist

To get around this, you can either:

1. Create the folder in the Dockerfile (source)
2. Change the permissions of the directory after it's mounted (source)

We used the former.

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

Feedback

```
upstream hello_django {
    server web:8000;
}

server {

    listen 80;

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

    location /staticfiles/ {
        alias /home/app/web/staticfiles/;
    }

}
```

Spin down the development containers:

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

Test:

```
$ docker-compose -f docker-compose.prod.yml up -d --build
$ docker-compose -f docker-compose.prod.yml exec web python manage.py migrate --noinput
$ docker-compose -f docker-compose.prod.yml exec web python manage.py collectstatic --no-input --clear
```

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

Navigate to http://localhost:1337/admin and ensure the static assets load 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  | 172.31.0.1 - - [20/Oct/2019:21:29:10 +0000] "GET /admin/login/?next=/admin/ HTTP/1.1" 200 1834 "-"
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:69.0) Gecko/20100101 Firefox/69.0" "-"
nginx_1  | 172.31.0.1 - - [20/Oct/2019:21:29:10 +0000] "GET /staticfiles/admin/css/base.css HTTP/1.1" 200 16378
"http://localhost:1337/admin/login/?next=/admin/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:69.0)
Gecko/20100101 Firefox/69.0" "-"
nginx_1  | 172.31.0.1 - - [20/Oct/2019:21:29:10 +0000] "GET /staticfiles/admin/css/login.css HTTP/1.1" 200 1233
"http://localhost:1337/admin/login/?next=/admin/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:69.0)
Gecko/20100101 Firefox/69.0" "-"
nginx_1  | 172.31.0.1 - - [20/Oct/2019:21:29:10 +0000] "GET /staticfiles/admin/css/responsive.css HTTP/1.1" 200 17944
"http://localhost:1337/admin/login/?next=/admin/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:69.0)
Gecko/20100101 Firefox/69.0" "-"
nginx_1  | 172.31.0.1 - - [20/Oct/2019:21:29:10 +0000] "GET /staticfiles/admin/css/fonts.css HTTP/1.1" 200 423
"http://localhost:1337/staticfiles/admin/css/base.css" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:69.0)
Gecko/20100101 Firefox/69.0" "-"
nginx_1  | 172.31.0.1 - - [20/Oct/2019:21:29:10 +0000] "GET /staticfiles/admin/fonts/Roboto-Light-webfont.woff
HTTP/1.1" 200 85692 "http://localhost:1337/staticfiles/admin/css/fonts.css" "Mozilla/5.0 (Macintosh; Intel Mac OS X
10.14; rv:69.0) Gecko/20100101 Firefox/69.0" "-"
nginx_1  | 172.31.0.1 - - [20/Oct/2019:21:29:10 +0000] "GET /staticfiles/admin/fonts/Roboto-Regular-webfont.woff
HTTP/1.1" 200 85876 "http://localhost:1337/staticfiles/admin/css/fonts.css" "Mozilla/5.0 (Macintosh; Intel Mac OS X
10.14; rv:69.0) Gecko/20100101 Firefox/69.0" "-"
```

Bring the containers once done:

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

Media Files

To test out the handling of media files, start by creating a new Django app:

```
$ docker-compose up -d --build
$ docker-compose exec web python manage.py startapp upload
```

Add the new app to the `INSTALLED_APPS` list in *settings.py*:

Feedback

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",

    "upload",
]
```

app/upload/views.py:

```
from django.shortcuts import render
from django.core.files.storage import FileSystemStorage


def image_upload(request):
    if request.method == "POST" and request.FILES["image_file"]:
        image_file = request.FILES["image_file"]
        fs = FileSystemStorage()
        filename = fs.save(image_file.name, image_file)
        image_url = fs.url(filename)
        print(image_url)
        return render(request, "upload.html", {
            "image_url": image_url
        })
    return render(request, "upload.html")
```

Add a "templates", directory to the "app/upload" directory, and then add a new template called *upload.html*:

```
{% block content %}

  <form action="{% url "upload" %}" method="post" enctype="multipart/form-data">
    {% csrf_token %}
    <input type="file" name="image_file">
    <input type="submit" value="submit" />
  </form>

  {% if image_url %}
    <p>File uploaded at: <a href="{{ image_url }}">{{ image_url }}</a></p>
  {% endif %}

{% endblock %}
```

app/hello_django/urls.py:

```
from django.contrib import admin
from django.urls import path
from django.conf import settings
from django.conf.urls.static import static

from upload.views import image_upload

urlpatterns = [
    path("", image_upload, name="upload"),
    path("admin/", admin.site.urls),
]

if bool(settings.DEBUG):
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

app/hello_django/settings.py:

```
MEDIA_URL = "/mediafiles/"
MEDIA_ROOT = os.path.join(BASE_DIR, "mediafiles")
```

Development

Test:

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

Feedback

You should be able to upload an image at http://localhost:8000/, and then view the image at http://localhost:8000/mediafiles/IMAGE_FILE_NAME.

Production

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

```
version: '3.7'

services:
  web:
    build:
      context: ./app
      dockerfile: Dockerfile.prod
    command: gunicorn hello_django.wsgi:application --bind 0.0.0.0:8000
    volumes:
      - static_volume:/home/app/web/staticfiles
      - media_volume:/home/app/web/mediafiles
    expose:
      - 8000
    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: ./nginx
    volumes:
      - static_volume:/home/app/web/staticfiles
      - media_volume:/home/app/web/mediafiles
    ports:
      - 1337:80
    depends_on:
      - web

volumes:
  postgres_data:
  static_volume:
  media_volume:
```

Create the "/home/app/web/mediafiles" folder in *Dockerfile.prod*:

```
...

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

...
```

Update the Nginx config again:

Feedback

```
upstream hello_django {
    server web:8000;
}

server {

    listen 80;

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

    location /staticfiles/ {
        alias /home/app/web/staticfiles/;
    }

    location /mediafiles/ {
        alias /home/app/web/mediafiles/;
    }

}
```

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 migrate --noinput
$ docker-compose -f docker-compose.prod.yml exec web python manage.py collectstatic --no-input --clear
```

Test it out one final time:

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

Conclusion

In this tutorial, we walked through how to containerize a Django web 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

For other production tips, review this discussion.

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

> There's also a Pipenv version of the code available here.

Thanks for reading!

django **docker**

Twitter Facebook Reddit Hacker News More 543

Join our mailing list to be notified about

Test-Driven Development with Python,

Feedback

course updates and new tutorials.

Enter your email...

Submit

Flask, and Docker

Get the full course. Learn how to build, test, and deploy a microservice powered by Python, Flask, and Docker.

View the Course

© Copyright 2017 - 2020 TestDriven Labs.
Developed by Michael Herman.

Follow @testdrivenio

Feedback