How to deploy Django with Docker

Conceived on Oct 3, 2016 / Series: python, django, docker

I finally managed to deploy Django in a Docker container on production! I've been trying to switch to a full Docker development/production model since Docker came out, but only recently did the ecosystem mature enough to allow me to easily use Docker both for development (where it excels, in my opinion) and on production (where it's pretty okay and quite useful).

Finally, Django, with Docker, **on production!**

In this post, I will quickly give you all the relevant details and files you need to go from a newly checked-out repository to a full development environment in one command, and to deploy that service to production. As a bonus, I'll show you how to use Gitlab (which is awesome) to build your containers and store them in the Gitlab registry.

Let's begin!

Development

First of all, let's start with the `docker-compose.yml`. In case you don't know, `docker-compose` is a way to run multiple containers at once, easily connecting them to each other. This is a godsend when doing development, but not that useful in production (where you usually want to deploy services manually).

To make development easier, we'll write a `docker-compose.yml` to set up and run the essentials: Postgres, Django's dev server, and Caddy (just to proxy port 8000 to 80, you can remove it if you like port 8000).

We have to do some contortions with the Django devserver, because Docker doesn't care if Postgres is ready before starting the server, so Django sees that it can't contact the database and quits. So, we just wait until port 5432 is ready before starting the devserver.

To connect to Postgres, just set the database hostname to `db`, the user and database to `postgres`, and the password to `password`. That's pretty much all the settings you need for this. I've also helpfully set the `IN_DOCKER` environment variable so your settings file can know whether it's running in Docker or not.

Generally, with Docker, you have to rely heavily on environment variables for configuration, rather than, say, a `local_settings.py` file. That's not necessarily a bad thing, as environment variables can be pretty handy as well.

Here's the complete `docker-compose.yml`:

```
version: "2"

services:

  db:
    image: postgres
    environment:
      POSTGRES_PASSWORD: password
    volumes:
      - ../pgdata:/var/lib/postgresql/data

  web:
    # Docker hack to wait until Postgres is up, then run stuff:
    command: bash -c "while ! nc -w 1 -z db 5432; do sleep 0.1; done; ./manage.py migrate; wh
ile :; do ./manage.py runserver_plus 0.0.0.0:8000; sleep 1; done"
    image: djangy
    build: .
    volumes:
      - ../code
      - deps:/deps
      - db
    environment:
      IN_DOCKER: 1

  webserver:
    image: lumanjimaa/caddy
    depends_on:
      - web
    ports:
      - "80:8000"
    command: -port 8000 -host 0.0.0.0 "proxy / web:8000 { }"
```

With this file and the `Dockerfile` below, setting up a new developer on the team consists of:

```
$ git clone <myrepo>; cd myrepo
$ docker-compose up
```

That's it. They have a complete development environment that mirrors production on their local computer, with one command. That environment also handles hot reloads, as usual, and will persist the database data under a directory of the repo.

To start the entire stack up, run `docker-compose up` and open http://localhost/, you should see your app's front page.

If you need to run a `manage.py` command, you can do it like so:

```
$ docker-compose run web /code/manage.py whatever
```

Done and done! Complete isolation with no extra RAM or CPU usage!

Production

To complete the dev setup, we'll need the `Dockerfile`. Its job is to list all the commands needed to get a container from a newly-installed Linux instance all the way to running your application. docker-compose uses the Dockerfile to build your application's image, and then sets up the rest of the containers to talk to it.

This is the `Dockerfile`. I will include comments in the file itself so you can follow along:

```
# Start with a Python image.
FROM python:latest

# Some stuff that everyone has been copy-pasting
# since the dawn of time.
ENV PYTHONUNBUFFERED 1

# Install some necessary things.
RUN apt-get update
RUN apt-get install -y swig libssl-dev dpkg-dev netcat

# Copy all our files into the image.
RUN mkdir /code
WORKDIR /code
COPY . /code/

# Install our requirements.
RUN pip install -U pip
RUN pip install -Ur requirements.txt

# Collect our static media.
RUN python /code/manage.py collectstatic --noinput

# Specify the command to run when the image is run.
CMD ["/code/misc/tooling/prod_run.sh"]
```

It's pretty straightforward, except for that last line. Why do we need a script? Why not just `runserver`? What's in that file? The questions keep mounting.

It's just because I want to `migrate` on the server every time before a run. This is what the `prod_run.sh` script looks like:

```
#!/bin/bash

./manage.py migrate
uwsgi --ini uwsgi.ini
```

Pretty simple! I use uWSGI to run Django, all you need is the appropriate configuration. I like to stick it in an .ini file, which looks something like this:

```
[uwsgi]
module=project.wsgi:application
master=True
pidfile=/tmp/project-master.pid
vacuum=True
max-requests=5000
socket=127.0.0.1:12345
processes=4
harakiri=120
single-interpreter=True
enable-threads=True
```

Adjust to taste.

As a bonus, here's the systemd config file I use to start the container. The configuration will also automatically pull from the registry before starting, so all you have to do to run the latest image is to restart the service:

```
[Unit]
Description=Docker container
Requires=docker.service
After=docker.service

[Service]
Restart=always
ExecStartPre=/usr/bin/docker pull registry.gitlab.com/yourname/repo:master
ExecStart=/usr/bin/docker run --net=host --env-file=/somedir/project.env --name=project regis
try.gitlab.com/yourname/repo:master
ExecStop=/usr/bin/docker stop -t 2 project
ExecStopPost=/usr/bin/docker rm -f project

[Install]
WantedBy=default.target
```

Using Gitlab to build your images

As I said earlier, Gitlab is amazing. It can run pretty much anything in its CI stage, including building your Docker images. It also has an integrated Docker registry, which means that, every time you push your code to the repo, Gitlab can automatically build a container so you can go to a newly-provisioned, fresh server that has Docker installed and do:

```
docker pull registry.gitlab.com/yourname/repo:master
docker run --net=host --env-file=/your/env.file --name=project registry.gitlab.com/yourname/r
epo:master
```

`--net=host` will use the host's networking and save you a lot of trouble forwarding ports. It's less secure, because everything inside the container will be running on the host's network space, but, since the dev server only listens to localhost anyway (and would run on the host's net space without Docker), I'm fine with that.

For our case, the four CI stages that will run on Gitlab are:

- Run a static check using `flake8` (which you should always do).
- Run your tests (which you should always do as well).
- Build your Docker image and copy it to the registry.
- Deploy everything to production (in my case, this happens by triggering a Captain Webhook URL. Captain Webhook will just restart the systemd service that runs the container, which will automatically pull the latest image before starting, as detailed above.

Here's the `.gitlab-ci.yml` that will build your images:

```
image: python:latest

stages:
  - staticcheck
  - test
  - build
  - deploy

variables:
  CONTAINER_TEST_IMAGE: registry.gitlab.com/yourname/repo:$CI_BUILD_REF_NAME
  CONTAINER_RELEASE_IMAGE: registry.gitlab.com/yourname/repo:latest

build:
  image: docker:git
  services:
    - docker:dind
  before_script:
    - docker login -u gitlab-ci-token -p $CI_BUILD_TOKEN registry.gitlab.com
  stage: build
  script:
    - docker build -t $CONTAINER_TEST_IMAGE .
    - docker push $CONTAINER_TEST_IMAGE
  only:
    - master

staticcheck:
  before_script:
    - pip install -U flake8
  script:
    - flake8 .
  stage: staticcheck

test:
  before_script:
    - pip install -Ur requirements.txt
    - python manage.py migrate
    - python manage.py collectstatic --noinput
  script:
    - python manage.py test
  stage: test

deploy to production:
  stage: deploy
  script: curl https://yourhost.com/deploy/${PRODUCTION_DEPLOY_KEY}/
  environment: production
  only:
    - master
```

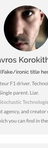
The two first steps (staticcheck and test) will run on every commit, but images will only be built and deployments will only be triggered for commits to master. If you use Git- or Github-flow or something similar, that's usually what you need. If you use some other branching model, you can configure releases/deployments appropriately.

Epilogue

Pretty much all you need to run Docker *both* locally, for development, and on production, is in those two files. If you want to use Gitlab's fantastic integration with everything, you have that third file, for no extra charge.

If you know of something I can install that will handle starting/restarting/updating my containers on the server, please let me know! I hear there are various solutions, like Kubernetes, but ideally I'd prefer something more lightweight. My ideal scenario is one where I can have a service or some software I can deploy containers to, and which will abstract all the service running and container updating away.

If you're aware of something that will do the job, or if you have any questions or feedback, leave a comment here or tweet at me. Thanks!



Stavros Korokithakis

(Flake/ironic title here)

Greek. Amateur F1 driver. Technology enthusiast. Single parent. Liar. Founder of Stochastic Technologies, a software development agency, and creator of various products which you can find in the résumé.