Docker orchestration example

To run our sample app on a orchestrated docker deployment, first, we'll clone the application. Head over to

https://github.com/carlesm/minidjangoapp

and clone it:

```
git clone git@github.com:carlesm/minidjangoapp.git
```

Once cloned, enter the minidjangoapp directory. Check there that the requirements.txt is OK. That is, it has all we need to run our application.

Dockerfile

Once requirements.txt is OK, we'll create the Dockerfile for our django application container.

Create a file called: Dockerfile (take care with capitals) with:

```
FROM python:2.7.14
RUN mkdir /app
WORKDIR /app
ADD requirements.txt /app/
RUN pip install -r requirements.txt
ADD . /app/
```

This file states:

1. Inherit from python container, with python 2.7.14.
2. Create a directory inside our container, called /app .
3. Start working inside that directory.
4. Copy requirements.txt file from our working directory on the host to /app on the container.
5. Install everything mandated by requirements.txt.
6. Copy the rest of the project to /app

Then we can test that it builds correctly:

```
docker build .
```

(The dot at the end means: on the current directory, and will search for a Dockerfile file.)

If it builds OK we can go to the next step, otherwise, we solve the building errors.

Orchestrate with docker-compose

To create an orchestration, that is, a set of different containers that will compose our application, we first create a file named docker-compose.yml on current (application) dir:

```
version: '3'
services:
  db:
    image: postgres
  web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/app
    ports:
      - "8000:8000"
    depends_on:
      - db
```

Take into account that this file is in YAML format, so spaces and tabs mean something. If possible use an editor que 'understands' YAML

Once you have this compose file, you can build it with:

```
docker-compose build
```

And, when built, we can start the orchestration with:

```
docker-compose up
```

If everything works, we'll have, on screen, the logs of both containers that comprise the orchestration, namely, a postgres db, and a django container (built with the Dockerfile we wrote above).

If, due to delays in the initial startup of the database container (postgres in its first run initializes the database, and that process can take a bit to run), django container fails due to not being able to connect to the database, just stop docker-compose (CRTL+C), and restart it, the next time, the database will boot up faster.

When running, we can connect to the django web server, on:

http://127.0.0.1:8000/

If we receive an error referring to the ALLOWED_HOSTS variable, we can edit the settings.py file, locate the ALLOWED_HOSTS variable, adding the values that Django mentions on the error page, and then rebuild the containers and boot them again:

```
docker-compose build
```

```
docker-compose up
```

If we then try to connect again, we'll receive an error page mentioning that some database tables do not exist, we should run then the migrate and the createsuperuser commands. But for it to work, we should run it inside the Django container with:

```
docker-compose run web python manage.py migrate
docker-compose run web python manage.py createsuperuser
```

EXTRAS

We can, evidently, expand our compose file, and fix some stuff. First of all there's a construct on the docker-compose.yml file that is only suitable for development:

```
    volumes:
      - .:/app
```

That line maps the directory we are working to the running app on the container. That's great for developing, as we can modify the code and django's internal server will reload it automatically. For production that implies that you have to run the container from the directory where the application source code is. As that is not what we'll want on production or testing or staging, we should, in those cases remove that line.

Another nifty thing we can do, is, run a complete IDE on the source code, for on-spot development. From the directory we have the source code we can run:

```
docker run -it -d -p 8001:80 -v .:/workspace/ kdelfour/cloud9-docker
```

And on port 8001 we'll have a nice IDE:

http://127.0.0.1:8001

We can, even, add it to the compose file:

```
c9:
  image: kdelfour/cloud9-docker
  volumes:
    - .:/workspace/
  ports:
    - "8001:80"
  depends_on:
    - web
```