WEB PROJECT

Deliverable 1:
Deployment of BookReviews

Ariza Pérez, José Ramon

`48257747K`

Buturuga, Mihaela Alexandra

`X9278777W`

Chen, Zihan

`Y1070565K`

Rodríguez Gonzàlez, Arnau

`39508430L`

Yang, Hao

`X4700996A`

Universitat de Lleida

Escola Politècnica Superior

Computer Engineering

2023, April 7th

Contents

1 Introduction **1**

2 Docker **2**

2.1 Dockerfile 2

2.2 Docker-compose 2

3 Environment variables **2**

4 Django configuration **3**

5 PostgreSQL configuration **3**

6 NGINX configuration **4**

6.1 Static files 4

7 Local deployment **5**

8 Render deployment **6**

9 Conclusions of a real deployment **7**

1 Introduction

The goal of this document is to explain the steps that were followed to deploy our BookReviews application, using Django as the web development framework, a PostgreSQL database and NGINX as a reverse proxy.

The project has been created inside a Docker container to interconnect all the services used to implement the app, with a Docker-Compose schema that configures every service.

The application can be deployed locally on a machine and also on the cloud, using Render as the Cloud Application Hosting instead of Fly.io, since it required a payment option to be used.

The GitHub directory can be accessed via the following link:

`https://github.com/GEI-WebProject/bookreviews.git`

Also, since the app is currently deployed on Render, you can access it with the following link:

`https://bookreviews.onrender.com/`

Considering that we are using the free tier that Render offers, the app is not constantly accessible if more than 15 minutes have passed since the last request was received. In that case, the app is suspended and is not deployed again until it gets a new petition. So, if you try to access it, and it goes very slowly, try again a few minutes later.

2 Docker

First, you need to install Docker in your computer. It is available in multiple operative systems, like Windows, Linux or macOS. Go to the website `https://www.docker.com/`, select your operative system and follow the instructions to get Docker installed.

Next, it is recommended to create a virtual environment to isolate the packages that need to be installed to deploy the application, in order to avoid incompatibilities between different projects. Also, as you need to install multiple packages, the project has a *requirements.txt* file that you can use to install all the dependencies at once with the following command:

```
$ pip install -r requirements.txt
```

This way, you ensure that the project will have the correct dependencies installed and it will work properly.

2.1 Dockerfile

The following step is to write the Dockerfiles required to deploy the application. A Dockerfile is a text document that contains all the instructions needed to build a Docker image.

This project has a Dockerfile for every service that runs the application: Django, PostgreSQL and NGINX. We have organized these services into separated folders.

2.2 Docker-compose

Docker Compose is a tool that is used to define and share multi-container applications. This tool allows you to create a YAML file to define the services involved. Then, with a single command can build the entire project, with the services interconnected or shut down.

3 Environment variables

In order to follow the 12-factor app methodology, the project uses environment variables, wich are defined in a separate *.env* file for each service. Each one of these files can be found in its corresponding service folder.

Although these files shouldn't be in the git repository since they contain compromised information like keys and passwords, we have included them so they can be evaluated by the teachers. Since these variables are for development and not for production, the files have been renamed to *.env.dev* to point that out.

4 Django configuration

The Python version used in the project is `PYTHON 3.10`. The Dockerfile of the Django app contains a Python base image and specifies some environment variables and commands that must be executed when the container is built.

The Django configuration was added to the *docker-compose.yml* file. This file contains the docker container name, the context where the project built, the commands that have to be run at startup, ports, volumes and dependencies with other services.

We want the Django service to migrate all the changes in the database, collect the static files that will be served by NGINX and to run the app with gunicorn. For that, the service has to execute the next command:

```
$ sh -c 'python3 manage.py migrate --no-input &&
        python manage.py collectstatic --no-input &&
        gunicorn --bind 0.0.0.0:8000 project_config.wsgi'
```

5 PostgreSQL configuration

The Dockerfile of PostgreSQL specifies the image of the database that will use the project to build the container. The versions used is the latest.

In the docker-compose file, it is added the service and its configuration. The folder corresponding to the database service contains a *.env.dev* file with all the environment variables from PostgreSQL.

Also, in the Django *.env.dev* file, it is added the DATABASE_URL variable that refers to the corresponding database used. For example, since the project must be deployed locally and remotely, and the databases are different, this environment variable can be set to specify which database is used via URL.

6 NGINX configuration

The Dockerfile of NGINX specifies the image of the reverse proxy service, respectively, that will use the project to build the container. The versions of both services are the latest.

A reverse proxy server is a type of proxy server that typically sits behind the firewall in a private network and directs client requests to the appropriate backend server, intercepting requests from clients.

It provides an additional level of abstraction and control to ensure the smooth flow of network traffic between clients and servers. It will also be configured to serve the static files of the Django-builtin interfaces and of our own.

In order to achieve this, since the app is served in the port 8000, the reverse proxy will listen a new port 8080 and redirect the clients through the 8000 port, that is the port that we configured without the proxy.

The configuration has to be done in the docker-compose file, exposing the 8000 port in the Django service and mapping the port 8080 of the computer to the port 8080 of the container. This service will depend on the Django service to be run.

Finally, in the NGINX folder is created a *conf.d* folder that contains a *default.conf* file with the NGINX configuration. This configuration is mapped in the docker-compose file from the system path to a path in the container.

6.1 Static files

Since we are using NGINX, we want the proxy to serve the static files that are used in our app. In order to do it, the project uses the WhiteNoise module to serve them in production, but NGINX has to make some configurations to serve them through itself.

For that, we have created a shared volume in Docker Compose that contains the static files in the project and are mapped from the app to the Django service and NGINX after they are collected when the whole project is built.

7 Local deployment

Finally, when the whole project is configured and all the views are created (contained in the project), you have to build the images of the services. As it has been mentioned before, with the Docker Compose schema, the services can be started all at once, considering the dependencies between them, according to the *docker-compose.yml* file.

So, to start all the containers, run the following command in your terminal:

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

The `--build` flag is used to build the containers before starting them.

If everything is configured correctly, once the containers are running, you will be able to access to the website in *localhost* with the next link:

```
https://localhost:8080
```

As you can see, the port is the one configured with NGINX to redirect the HTTP requests to the app.

8 Render deployment

Since we are deploying our application on Render, if you want to create your own Render account and deploy this app or another one, you will have to follow the next steps.

Firstly, you will have to create an account on `https://render.com/`. Then, you will need to create a remote PostgreSQL database on Render. You can select the free version that has memory limitations.

Once the database is created, you can deploy the app to the cloud. To do so, you have to create a new Web Service and configure it yourself. In the first step of the creation of the web service, it will be required to use a GitHub account with the repository that contains the project, or you can specify the public repository link.

Moreover, one interesting feature of Render is that after every commit pushed to the repository, Render will auto-deploy the app again with the upcoming changes.

After that, you will specify some fields:

- **Name**: The name of the application. In our case is bookreviews.
- **Build command**: This command runs in the root directory of your repository. It is typically a script that installs libraries, runs migrations, or compiles resources needed by your app. We will add `python manage.py migrate` to do the migrations of the database before every deployment.
- **Start command**: This command runs in the root directory of your app to start a webserver for your app. It contains the name specified when the Django project is started with the command `$ django-admin startproject <name>`

  In our case, this parameter is `gunicorn project_config.wsgi:application`

Finally, click on Advanced and specify all the environment variables that Render needs to deploy the application, like the database created on Render before in `DATABASE_URL`, the `PYTHON_VERSION`, `SECRET_KEY`, `ALLOWED_HOSTS`, `DEBUG`, etc. You can see in the Django environment file all the environment variables used in our project.

Finally, Render will start to deploy the app. It takes several minutes because the free tier is limited. If everything is configured correctly, you will be able to access the website with the link shown above in the Render website.

9 Conclusions of a real deployment

For the deployment of this application, we need a web server, a reverse proxy, and a database. Each functionality will be installed in a separate server. For SQL petitions, the application can connect to a Postgres server, and a reverse proxy like Nginx will redirect web requests to the appropriate backend server. Nginx will also serve static files.

The web server is the presentation tier and provides the user interface. This is usually a web page and is usually developed using HTML, CSS and Javascript. It is necessary to show the users a well-designed website that is accessible.

The application server corresponds to the middle tier, housing the business logic used to process user inputs. In our case, Django is the framework used to develop this layer because it provides a lot of built-in features that help to implement the app.

Concerning to the database server, it is the data or backend tier of a web application. In our case, PostgreSQL runs on database management software. It is necessary to have a database to manage all the entities that form the application: Books, Authors, Publishers, etc. Also, it can be used as a session storage management.

Although a reverse proxy is interesting for load balancing and caching content, you could not use it. If you don't want to use a proxy server, then Nginx can be removed, Even though we can remove the proxy server, the database service cannot be removed as it is needed for storing sessions and the data entities.

Also, this application has a backend running as REST API, OpenLibrary, to get the information of the books, authors and all the relations in the app and store them in the database.

Finally, it's important to note that the application deployment should be scalable, so if the website becomes popular and gets more visits than expected, we are able to provide our services continuously without interruption.