Politécnico de Leiria

Engenharia Informática - Computação Móvel

Dcokerization and Upload to Cloud of a Frontend-Backend-Databse System

João Santos (2232646)

Conducted within the framework of:

Cloud Computing

January of 2024

Contents

1 Introduction

This project aims to containerize a system composed of a frontend, backend and database, creating individual Dockerfiles and a Docker-compose file. The docker files to be placed on a cloud server, ensuring that the service is available on the cloud.
The base project to be used for the containers can be any implementation that follows the described structure.

2 Contextualization of the system to be dockerized

The systems utilized in this project is a system developed by a fellow graduate from IPL, Miguel Jesus, which was developed for the Portuguese federation of Lohan Tao Kenpo with the purpose of simplifying the managerial tasks regarding events, as well as the process of signing up by participants and checking of information. It is currently in negotiations for potentially being utilized by foreign Kenpo federations.

The system consists of three servers:

- A frontend server, developed with Vue3.js;
- A backend server, which is a python REST API mostly utilizing fastapi and sqlalchemy modules;
- A database server, developed with PosgreSQL.

The front end makes requests to the backend, and the backend to the database. This being the case, the servers can be isolated so that the database only receives requests from the backend, and that the backend only receives requests from the frontend, which must itself be left exposed to external traffic in order to provide the intended service.

Figure 1: Architecture

For the purposes of testing the database, the login creadentials ”miguelangeloleal@hotmail.com” and ”1234” should be used.

3 Dockerfiles and Docker-compose file

The containers should all be initialized inside the same network in order to be able to communicate with each order. If the Docker-compose file is utilized, there is no need for assigning them a network, as the compose will do so automatically.

3.1 Frontend

The Dockerfile for the frontend utilizes the image "node:18-alpine" as its base image.
It had "curl" added to the list of installations it performs solely for the purposes of troubleshooting some issues that were encountered regarding connectivity between containers. As such it can be safely removed from the Dockerfile without affecting its performance.
Since this container utilizes node, its dependencies are installed with "npm install".
The service of this container is available on port 3000, so this port is left exposed. Additionally, in order to reach the backend, this container required that the environment variable "VITE_API_URL" be set to "http://backend:8000", whith 8000 being the port that the backend service listens on, and with "backend" being replaced by the ip of the backend container. Since this server did not have any environment variables set up, it was necessary to create the .env file and define it there.
Finally, this container runs with the command "npm run dev".

3.2 Backend

The Dockerfile for the backend utilizes the image "python:3.10.12-slim" as its base image.
Since this container utilizes "poetry", it is necessary to install it before installing the dependencies, which is done with the command "pip install poetry==1.8.2", followed by installing said dependencies with "poetry install –no-root".
The port this server runs on is port 8000, so it is left exposed, and it utilizes the environment variable DATABASE_URL to identify the database server, as well as the specific database it is trying to reach and which user and password to use. In this manner, the variable follows this format: "postgres://postgres:123@db:5432/kapi", with the second instance of "postgres" corresponding to the user, the "123" to their password, db to the ip address of the database server, 5432 to the port the database should be listening on and "kapi" to the database to use.
Finally, this container runs with "poetry run uvicorn app:app –host 0.0.0.0 –port 8000".

3.3 Database

The Dockerfile for the database server utilizes the image "postgres:16" as its base image.
In accordance with the documentation of this image, the dump file from which the database can be extracted is placed inside the directory "/docker-entrypoint-initdb.d/", which automatically executes all of its contents on initialization. While it can be used for running scripts, when it comes to a dump file it extracts its contents into the main database. In this manner it is possible to migrate the database from a local installation of this solution into the container.
Also based on the documentation of this image, the environemnt variable "POSTGRES_USER", "POSTGRES_PASSWORD" and "POSTGRES_DB" are set to

"postgres", "123" and "kapi", having the effect of naming the main user as "postgres" and assigning it a password of "123" and naming the main database "kapi". The port "5432" is then exposed, as that is the port postgres listens on for its database service.
Lastly, a volume is assigned to the folder "/var/lib/postgresql/data" where the databse is stored, so that the data in this container becomes persistent.

It is worth noting that in local installations of this solution the backend server is ran inside of a virtually environment, to ensure it has the correct version of the several components it uses, and the frontend server requires the use of the command "nvm 18" to ensure the correct version of node is utilized. Since each container is its own isolated system there is no need for that in this solution.
Furthermore, it is paramount that the urls the servers use to identify each other actually use ip address instead of relying on name resolution, as testing their connectivity revealed that they are able to resolve each others' names, but can not resolve them when making api requests, strange as that may seem.

3.4 Docker-compose file

The docker compose file is based on the images produced from the previous Dockerfiles, save for the database.

Since running the compose file will result in the creation of a network with all these container connected to one another, it is not necessary to leave ports exposed on either the backend or the database. Thus, only the frontend retains its port exposure (on port 3000).

As a means to workaround the previously stated issue regarding name resolution during requests, the entrypoints for the frontend and backend are modified to also resolve the name of the server they intend to make requests to and use the resulting api to update the environment variables where they store the location of they target servers. This action is done in addition to their respective entrypoint commands, achieved with using "/bin/sh -c" to execute both commands in sequence.

Regarding the databse, a fresh image of postgres (also version 16) is utilized for this, with all the same configurations of the Dockerfile. This is done because it seemingly solves issues regarding permission errors generated by the mounting of the volume.

Both the individual containers and the compose file appear to yield the desired result, displaying proper connectivity between the containers in terms of allowing the service to function similarly to how it functions in my own computer.

4 Cloud Implementation

Since GoogleCloud does not support the use of postgres in its containers, it was only used for the frontend and backend containers, being substituted by

Aiven as the could option for the database.

4.1 Databse

The in prder to setup the database in Aiven it was necessary to modify the dump file, as Aiven does not allow its users acces to the user "postgres", as such each instance of this user inside the dump file was replaced with "postgres2". Following this, it was necessary to create the user "postgres2" and the database "kapi" inside Aiven's container. After creating this databse, it was necessary to grant priveleges to the new user with the command "GRANT ALL PRIVILEGES ON DATABASE kapi TO postgres2", and the command "GRANT CREATE ON SCHEMA public TO postgres2", which must be run inside the database "kapi". Only then was the extraction of the database from the dump file into the database "kapi" inside the Aiven container able to execute without errors.

4.2 Frontend and Backend

The setup of the fronend and the backend on the cloud was fairly simple, needing only to upload them to a Github repository and access it through GoogleCloud to gain access to the Dockerfiles and relevant files for each container, and to specify which port each container should leave open to listen on.
The only adjustments to be made were the environment variables which point towards the backend and the database. The variable pointing to the database in particular needed the username to be adjusted from "postgres" to "postgres2".

Unfortunately, the backend server does not seem to be functioning correctly.

The frontend can be access through "https://google-cloud-proj-hjqxpl7fuq-uc.a.run.app/" and the backend through "https://db-server-hjqxpl7fuq-ew.a.run.app/". Access to the database utilizes "postgres://avnadmin:AVNS_IgTcc-E9yvIE5d9ftwe@db-postgres-cloud-proj.a.aivencloud.com:11893/kapi?sslmode=require"

5 Difficulties faced

- My HDD where I hold the files used for this project has windows's filesystem, resulting in a windows exclusive error with postgres volumes regarding permissions, this lead to needing to create the volume in a different disk. This error was particularly difficult to diagnose as I am running linux in my computer, which lead me to exclusively focus on similar problems and solutions relating to linux machines;
- Creating the volume for the database container yielded ownership errors, which I was able to resolve once, but failed to replicate. In later testing it was discovered, by trial and error, that utilizing a new image of postgres instead of my own did not have the same issues;
- While DNS resolution works between individual containers, it fails when used for requests. After many failed attempts to resolve this issue, I decided to simply resolve the name and replace the value of the environment variable with the ip

address of the targeted container, thus ensuring dynamically allocated connectivity; - The system installed on my own computer fails to perform some requests, which was only detected after finishing the development of the containers, and was unable to fix, meaning the same issue is present in the containers. It does however work fine in the computer of Miguel Jesus, which tested the containers as well and found them to be working correctly. - While the Dockerfile for the backend installs poetry, running the compose file on the computer of Miguel Jesus yielded an error relating to not having poetry installed. For this reason it was necessary to include "poetry install" in the list of commands in the containers entrypoint within the Dockerfile. The same issue was detected when starting the backend container in Google Cloud, being resolved in the same manner.

6 Conclusion

This project had me unsure of weather or not I would be able to complete it, or need to opt for a different base project, as I was not, and am not, familiar with the base project utilized for the containers, however, through perseverance a sheer stubbornness I was able fix and workaround several issues I initially thought impossible on account of them being reported as CORS errors by the web console, when in reality they were something far more bizarre (the file that manages CORS was checked checked several times and is properly configured).
While the cloud implementation did not quite pan out the way I had expected, the Docker files and compose file do provide functional containers, which I consider a great success.
All things considered, while not all the goals for this project were met, I have become far more confident of my ability to containerize any given project on account of successfully containerizing the base project in spite of all the adversity faced here.