CSC 847
Project 1 - Report

Software Environment

This project is a 3-tier web application to manage students on a graphical UI.
It is composed of:
- A SQL database that contains a table of student data.
- A GoLang API (backend) that interacts with the database to manage students.
- A React Web App (frontend) to allow the end users to upload, display and search students.

The code is hosted on a public git repository.

Database

The database is a PostgreSQL version 14 hosted on Gcloud SQL database service.
For local development, I ran a local PostgreSQL database using Docker.
For testing, I used SQLite3.

Backend

The Backend is coded in Golang version 1.20 with Gin framework as HTTP server and Ent as ORM.

I used an MVC (Model Vue Controller) architecture to design the API and Postman to test the API.
In production, the backend is run in a container thanks to Docker.

Frontend

The Fronted is developed with React version 18.2 and Material UI as components library.

In production, the backend is run in a container thanks to Docker.

GCloud compute engine

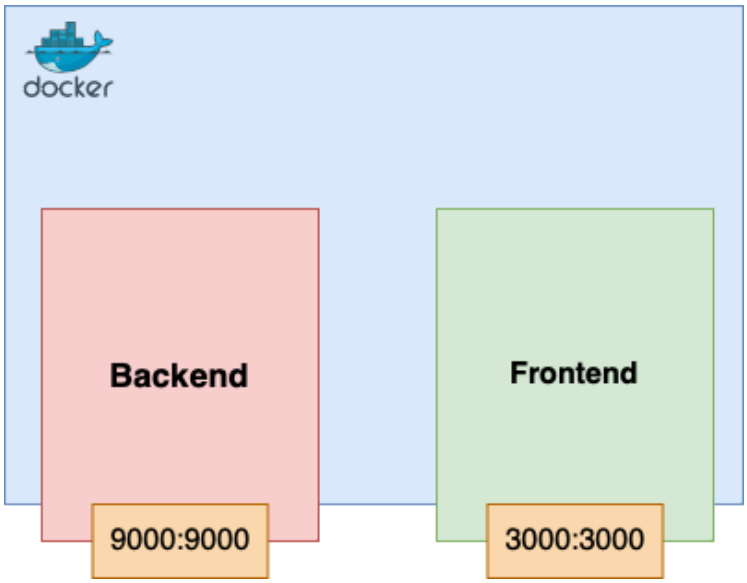
The instance is an E2-Medium with Ubuntu 22.04 LTS as the operating system.

It includes:
- Docker version 23.0.1 to run the backend and frontend.
- Nginx 1.18.0 as the reverse proxy to expose the backend and frontend.

Experience

The most interesting question I had to answer was: How can I have the most accurate local environment compared to the production in Cloud.
I used Docker to keep a constant environment and isolate my application into containers that will be deployed in the same way in production.

Local Architecture



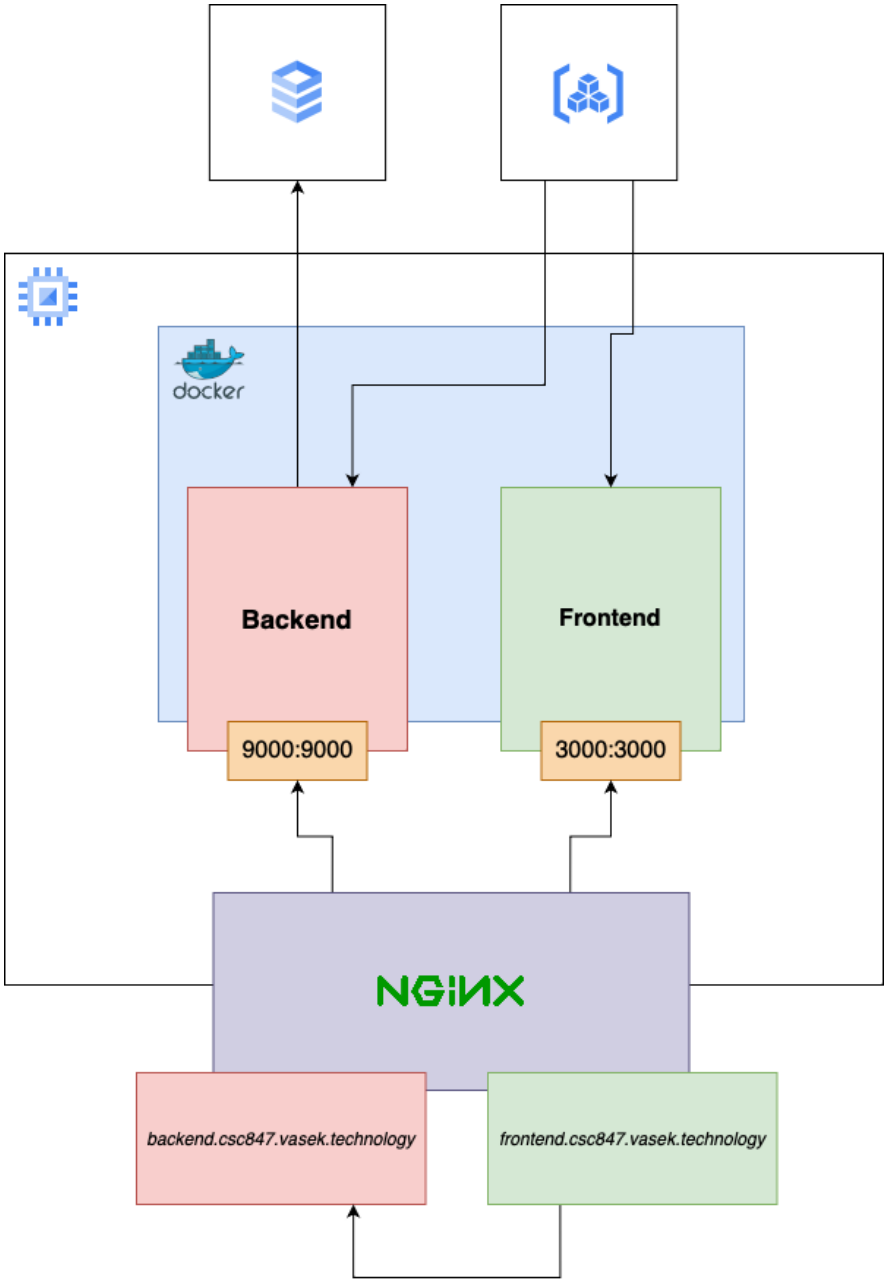
The local architecture is composed of a backend and a frontend ran in two distinct containers.
Images are built and run locally in the Docker engine.

Thanks Buildkit and its cache feature, it was really fast to publish and tests changes.
To simplify the management of those containers, I used Docker Compose.

Cloud Architecture

The Cloud (or production) architecture reuses the local architecture but adds some components to expose endpoints publicly.
Images are stored in iCloud Artefact Registry to simplify their access.
On deployment, the docker compose will pull those images and run them.

The backend will then use a Cloud MySQL instance to store students.
Finally, NGINX acts as a reverse proxy to expose the backend and frontend on secure HTTPS URLs.
The backend can be accessed at https://backend.csc847.vasek.technology and the frontend at https://frontend.csc847.vasek.technology.


docker
Backend
Frontend
9000:9000
3000:3000
NGINX
backend.csc847.vasek.technology
frontend.csc847.vasek.technology

Challenges

Change of environment

It was primordial to set up a correct working environment that replicate the production environment on the GCloud compute engine. Thanks, Docker, I ran a local PostgreSQL database and when the backend was finished, I also set up a local development stack so the frontend could interact with the backend the same way it does in production.

This required the usage of configuration through the environment and the development of multiple database connection providers that could be easily switched depending on the environment.
The setup of the testing environment was the most challenging. I had to create the local database using SQLite so it could run as a stateless process.

CI/CD

To ensure the correct integration of new features, I set up continuous integration and continuous development process.
When a new feature is pushed on main, an action is triggered to build a Docker image of the frontend and the backend so the GCloud compute engine can use the latest image without pulling and building applications on the machine.
This made me win a lot of time because the instance was not enough powerful to build the stack in less than 5 minutes while it takes only a couple of seconds to pull the new images.

This also required the correct usage of GitHub secret to not show my Google credentials to the public when images are pushed on a Google Artefact Registry.

Database connection

The connection to GCloud MySQL required a custom SQL connector because it was not natively handled by Ent.
This again required a custom environment and special management of GCloud authentication.