People's Democratic Republic of Algeria Ministry of Higher Education and Scientific Research

Graduate School of Science and Technology of Computer Science and Digital

Project Report:

This Project is carried out by:
BOUZENA Ali

Framed By:
Pr Kechadi Taher
Mme Khelouf Hanane

1. Introduction

This project centers around building a microservice-based application using Docker and Docker Compose, demonstrating how various technology stacks can be containerized and orchestrated. The goal is to develop a fully functional web application that integrates API and database services, all managed within Docker containers. The tech stack includes Flask for the web interface, Fast API for the API layer, and PostgreSQL as the database.

2. Technologies Used

- Docker and Docker Compose
- Flask (python) for the web application
- FastAPI (Python) for API service
- PostgreSQL for the database
- Adminer for database management visualization

3. System Flow Diagram

The overall system architecture comprises multiple services working in isolation but networked through Docker. The services include:

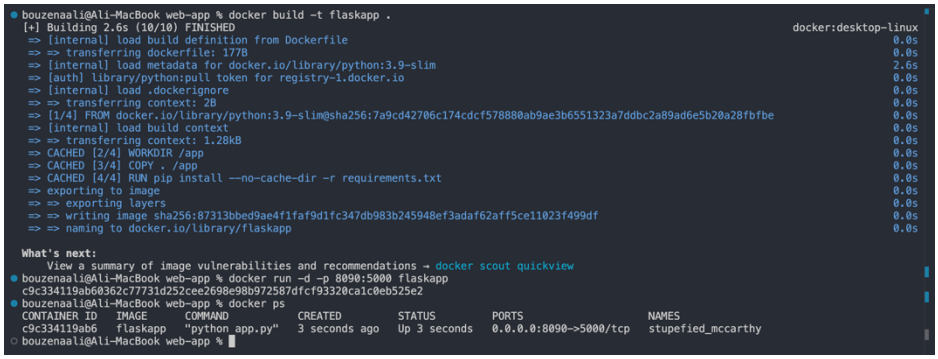
1. A web application running Flask.
2. An API service developed with FastAPI.
3. A PostgreSQL database.
4. Adminer for database management.

4. Steps Executed in the Project

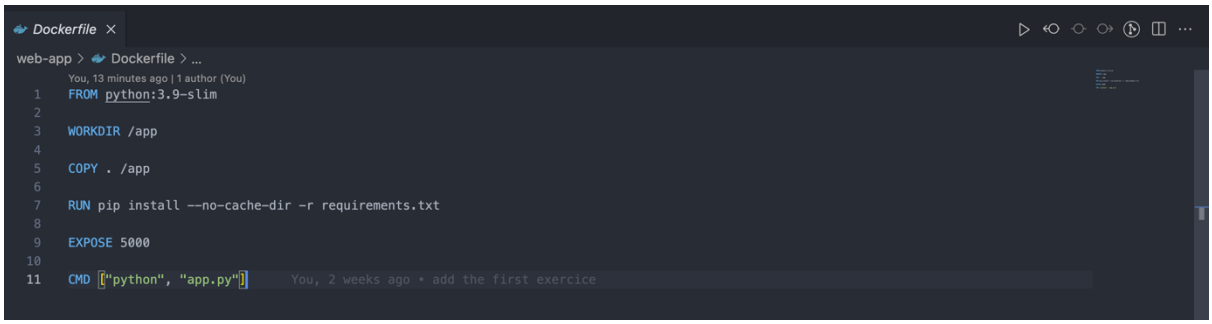
Exercise 1: Web-App Setup with Flask

- Created a Docker container for a Python-based Flask web application.

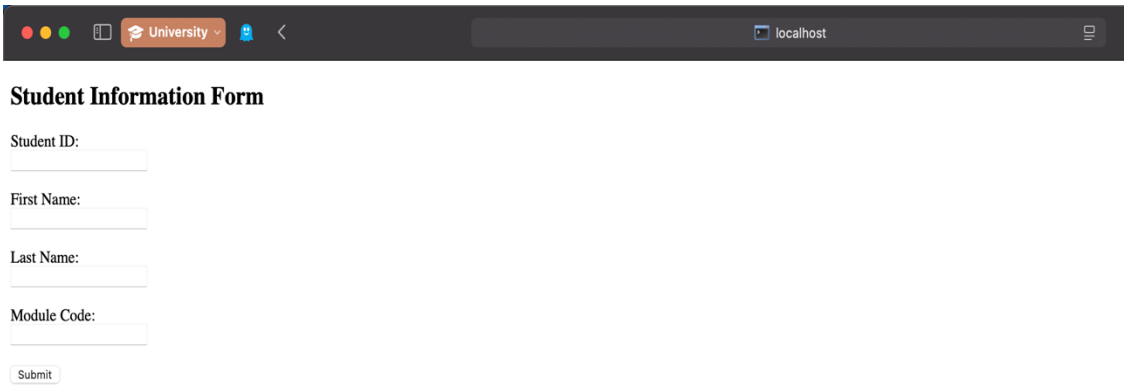
- Configured the app to run on port 8090 on the host and 5000 in the container.
- Ensured that the web application was functional with endpoints like `/add` and `/all`.



Building and running my web app container



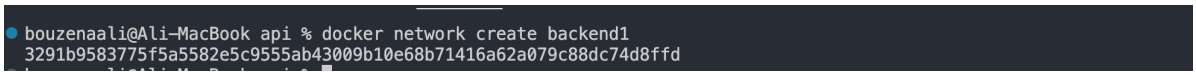
The dockerfile of web-app



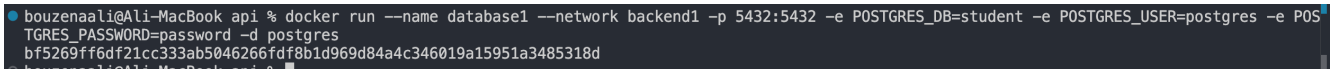
The result of the web-app

Exercise 2: API Setup with FastAPI

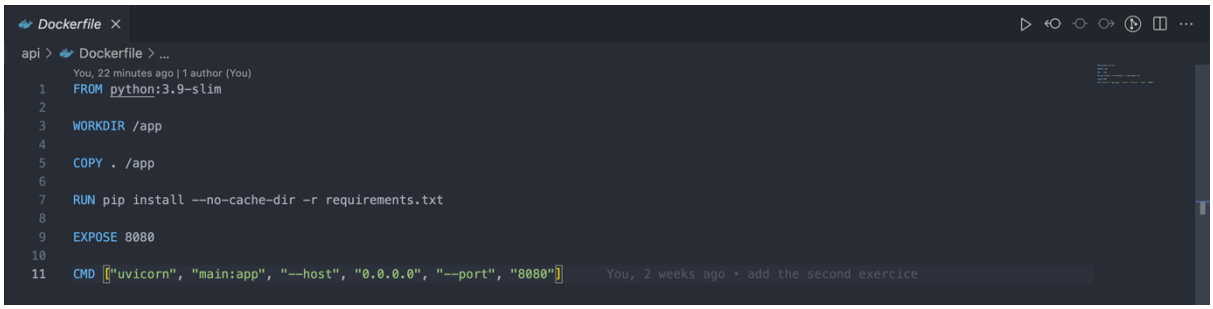
- Developed the API service with FastAPI, also containerized using Docker.
- Integrated the API with a temporary PostgreSQL database running in a separate container.
- Successfully exposed the API on port 8081, tested via the `/docs` FastAPI interface.
- Used **Docker network** to ensure seamless communication between the API service and the PostgreSQL database. This involved setting up a custom Docker network to allow both containers (API and database) to communicate without issues. This step was crucial for resolving connection challenges between the API and the database container.



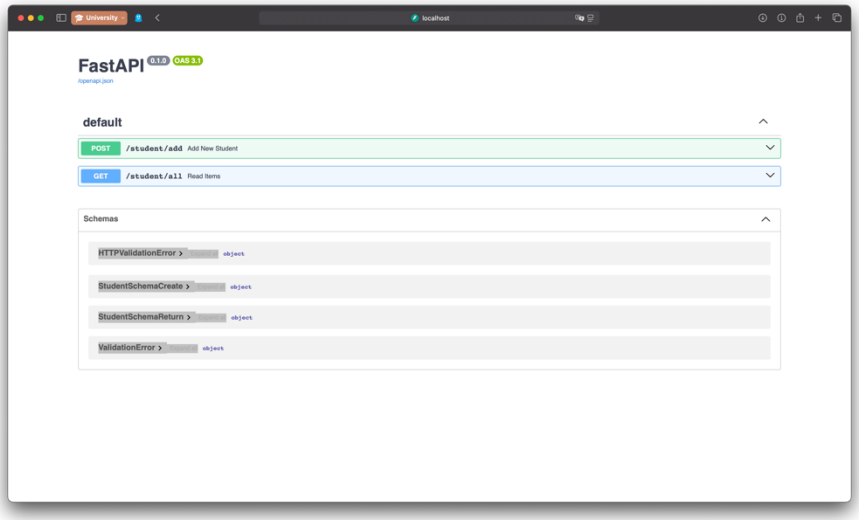
Create network named backend



Running temporary db in the backend network



Dockerfile of the api



The result of the api

Exercise 3: Database Service Setup

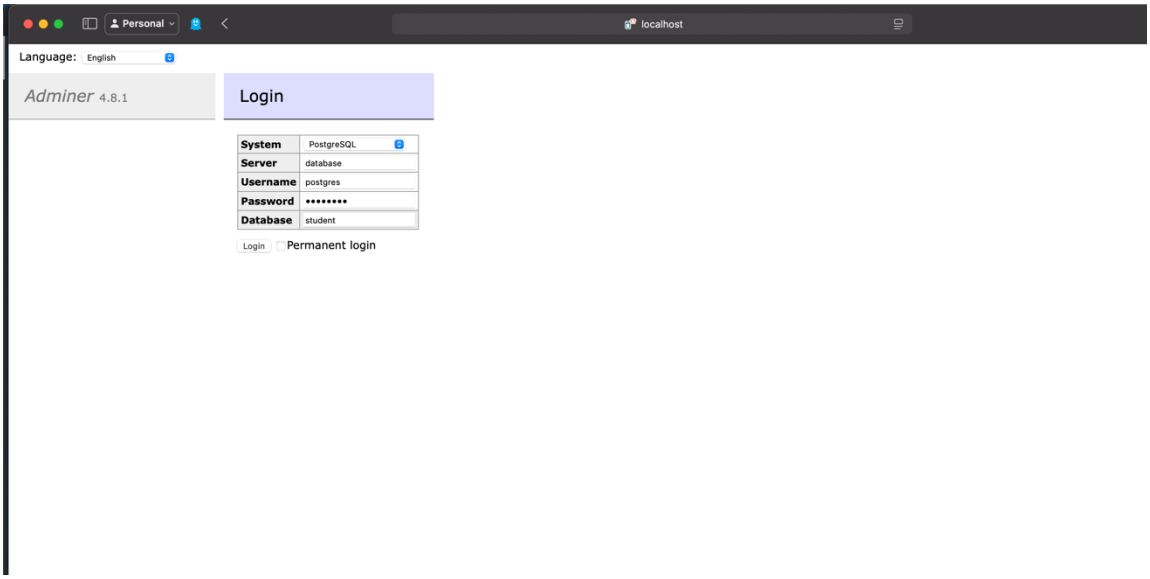
- Configured a PostgreSQL database in a container.
- Introduced **Adminer** to visualize and manage the database from a browser at port 8091.
- Created a custom Docker network to allow inter-container communication, enabling the API to connect to the database.

```
bouzenaali@Ali-MacBook PROJECT1_Ali_BOUZENA_212133002547 % docker run --name database1 -e POSTGRES_PASWORD=password -d postgres
d33332dbd19ad68c8863e1b8a1697303bddeaec84a058c4025642cd5163686de
```

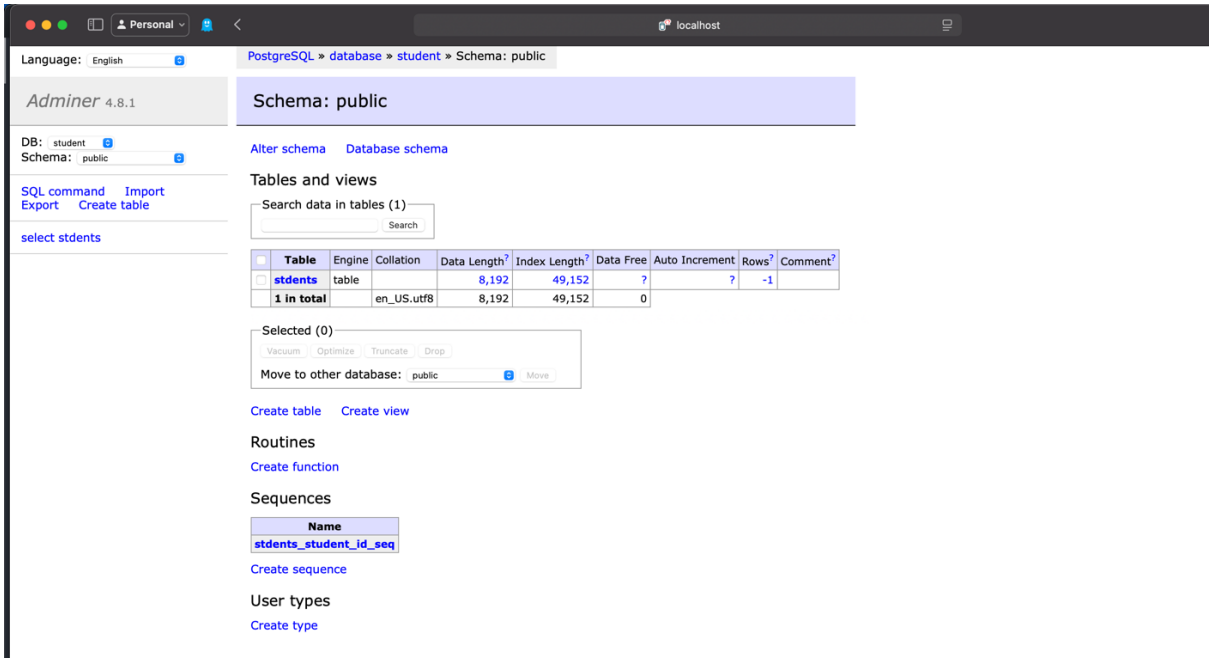
Configured a PostgreSQL database in a container.

```
bouzenaali@Ali-MacBook PROJECT1_Ali_BOUZENA_212133002547 % docker run -p 8092:8080 adminer
[Thu Oct 24 20:25:29 2024] PHP 7.4.33 Development Server (http://[::]:8080) started
```

Running adminer



Adminer before configure the network



Adminer works

Exercise 4: Docker Compose

- Consolidated all services (web-app, API, database, Adminer) into a single Docker Compose setup. ● Implemented network and volume management in the `docker-compose.yaml` file.
- Verified that the web application interacts with the API and the data persists in the database.

5. Conclusion

This project effectively demonstrates the use of Docker and Docker Compose for managing microservices. Each component was containerized and seamlessly integrated using best practices such as shared networks, environment variables, and persistent volumes to ensure smooth operation and data persistence.