COMP.CS.510-2022-2023 Web Development 2 – Architecting Project

Group information

Members: Juho Karhunen, 267562, juho.karhunen@tuni.fi
Juho Nuottimäki, 284608, juho.nuottimaki@tuni.fi
Joonas Pelttari, 274830, joonas.pelttari@tuni.fi
Group name: Unknown
GitLab Url: https://course-gitlab.tuni.fi/compcs510-spring2023/unknown

Introduction

In this document you will find out about our group's workflow during the project, architectural decisions made, what technologies were used, how to run the application and what our group learned during this project.

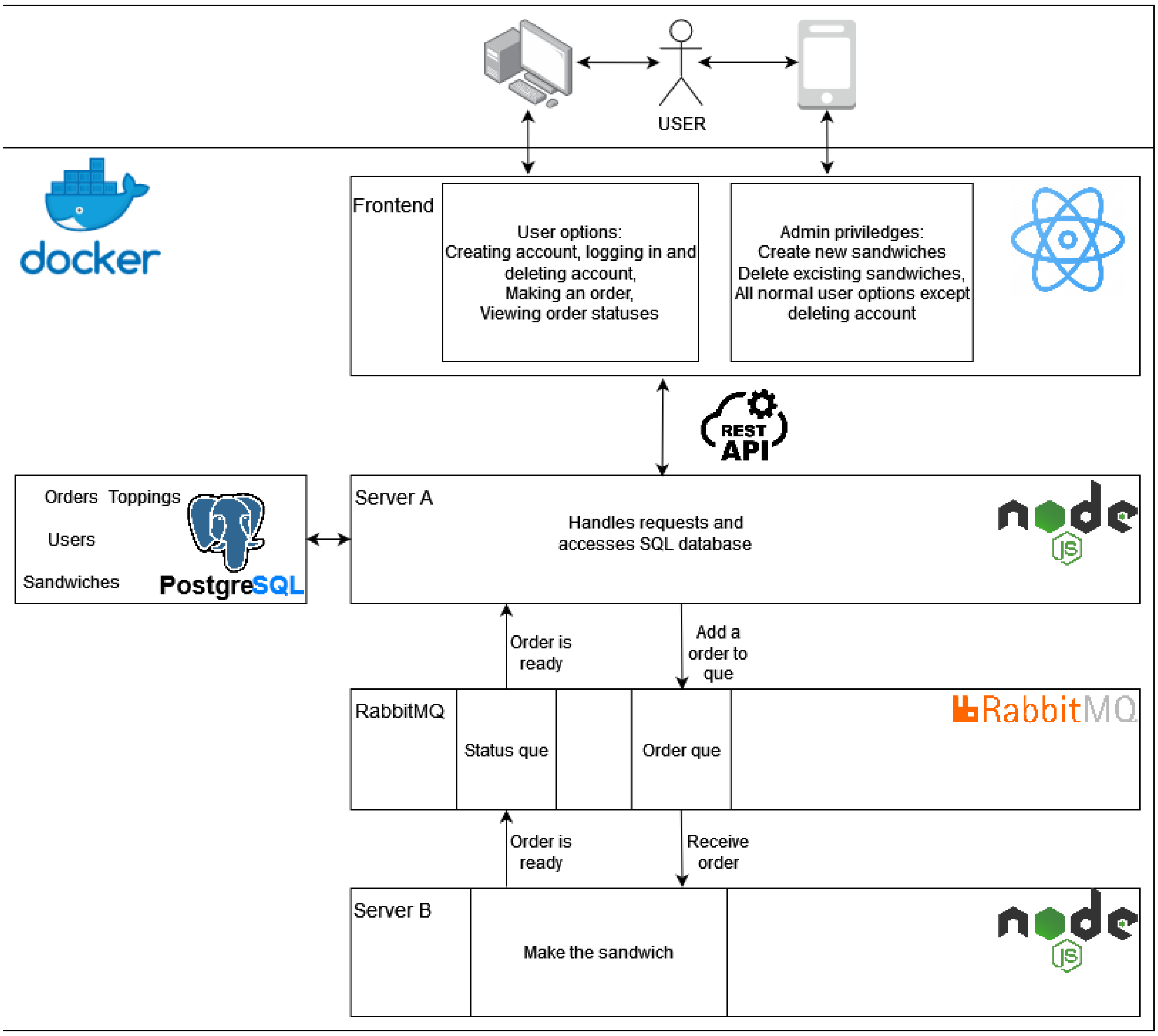
Working during the project

The first week after the project was released, we had our first meeting. There we talked a little bit about the project schedule and the overall responsibilities of each member. We decided that Karhunen would be working on the frontend, and Pelttari would work on server A and Nuottimäki on Server B. For next week's meeting we decided that we would all dive deeper into the documentation and try to test setting up the project on our own. When the second week's meeting was, we had a basic front end with server A and server B setup with no communication between these pieces. Then we decided to do some designing where we first decided to just do the very basic requirements and add features after that. The first front-end design was made by Karhunen and reviewed by the team members.

We decided that each member should devote 8 hours a week to the project. At the beginning we didn't use as much time but as the weeks progressed, we have had at least one coding session between all the team members and some coding on our own. Since the project was easily split up into three parts that each member could devote their time to, we found that working in parallel made the coding of each feature easier, since we could communicate and make quick decisions.

System architecture

Our project was designed as a microservice architecture, the different parts of the application are independent from each other, and they can easily be replaced or modified, if we just use same or at least similar communication methods between these services.



USER
Frontend
User options: Creating account, logging in and deleting account, Making an order, Viewing order statuses
Admin priviledges: Create new sandwiches Delete excisting sandwiches, All normal user options except deleting account
REST API
docker
Orders Toppings
Users
Sandwiches
PostgreSQL
Server A
Handles requests and accesses SQL database
Order is ready
Add a order to que
RabbitMQ
Status que
Order que
Order is ready
Receive order
Server B
Make the sandwich

Comparing our architecture to a monolithic design. For such a small application a monolithic architecture would have been easier to implement. A singular program that handles such a small and specified amount of requirements. However, if we were to add more functionalities or needed to change a full implementation of a component it might break the whole application and other parts of the program would need refactoring.

Another possible architecture could have been client-server. To some extent this is the architecture we have implemented here but the server has been divided into smaller modules each serving their own purpose. This could also have been implemented in a way that the server has only one application running. This application could handle everything from routing requests to handling the orders. Client could have then been a standalone desktop application that communicates with the server to do the order handling.

Used technologies

Docker Compose was used for containerization. All components in the application run in containers, this allows the application to be easily deployed to a multitude of different platforms and scaling the application this way is easier. For example, it would be possible to introduce a new server into the existing Docker container setup, which would be responsible for tracking the location of food couriers during the delivery process.

RabbitMQ was chosen to use for queuing messages between servers A and B. RabbitMQ provides a robust and scalable communication mechanism between the different components.

Nodejs was chosen as our server-side technology. Nodejs is an efficient technology that allows for building applications with great performance.

React was chosen as the front-end library mainly because two of our group members had prior experience of using it. React can be used to easily make reactive frontend applications with a lot of "moving" parts and changes in states without the frontend application getting too complicated by dividing different parts into component.

PostgreSQL was also chosen for the fact that we had prior experience of it in our group. PostgreSQL was easy to implement on our server and it provides a reliable and fast database solution.

We could have chosen to use VueJS as the front-end library since none of us had experience in that language and we could have learned development on that. However, we decided that we all have multiple other assignments from other courses, so we decided to use React to save some time. Otherwise, we think that the technologies we used were great for the project at hand.

How the produced system can be tested

The application can be tested in the virtual machine provided by the course. Please refer to usage of the application for more instructions about actual usage of the application.

Clone the project to the virtual machine:

```
$ git clone https://course-gitlab.tuni.fi/compcs510-spring2023/unknown.git
```

Navigate to the folder containing docker-compose.yaml and run podman with the following command. It will build the project and start the containers.

```
$ podman-compose -f ./docker-compose.yaml up -d –build
```

To shut down the containers run command:

```
$ podman-compose -f ./docker-compose.yaml down
```

If Docker Compose is used instead of podman, replace 'podman-compose' with 'docker compose'.

Open browser and navigate to http://tie-webarc-38.it.tuni.fi:3000. You *have to* be connected to the eduVPN. Now you should be able to see the frontend app.

Usage of the application

The user can use the application and make orders without logging in. Creating an account does not currently give the normal user any other usage options than deleting the account. This could be expanded later to binding the orders to the users.

Frontend was designed in such a manner that multiple users can use the application at the same time and they will have different orders. Even tho it's the same server that is handling the requests.

An admin account is created in the server that has more privileges. When logged in as the admin account you can create new sandwiches, and you can delete existing ones by navigating to the make order page and pressing the delete button.
Admin account credentials and API-key are found below.

username: admin55
password: secret55
API-key: sub30

Learning during the project

We chose to use the comment sections inside the GitLab issues to track our progress in the tasks. This was to not overflow the issue board with many issues. This might have been the wrong decision for a bigger project, since following the comment sections would have been harder. In our case since every member was working on a different part of the application so following their own issues was easy but if the members worked on same tasks this could have become an issue later.

Juho Karhunen

I learned a lot about frontend development with react and communicating with servers through fetch requests. The swagger API tool was a great learning experience since it makes development of servers and frontend side by side easier. And I will utilize it in the future. I learned a lot about teamwork and how to navigate developing servers and frontend side by side. I learned about frontend styling options and new ways to utilize the react use States. I also learned about containerization and deploying the containerized application to a virtual machine which was all new to me.

Juho Nuottimäki

I'm still fairly new to web development as I have only studied the first web course, so I was really able to deepen my knowledge on technologies surrounding these environments. I found containerization to be very interesting. I have been doing stuff with VMs before and have heard about containers being somewhat like them, but this was the first time I have actually done something with them. Containers seem to solve many problems that VMs have like being much easier and quicker to setup while still having the flexibility and isolation. Also combining the two in this project was also very rewarding. Splitting the work during the development process also taught me things about how modularity truly helps the project in many ways.

Joonas Pelttari

I learned a lot about containerizing applications which is a really useful skill as I didn’t have knowledge of containerization or Docker beforehand. RabbitMQ was also a new thing for me and after some problems about port exposing and amqp URLs I got into it and realized it is a pretty simple and good software to use. I had a lot of small problems when trying to make NodeJS, RabbitMQ, browser frontend or browser monitor software(rabbitmq monitor/postgre adminer) work together in a multi-container application but playing around with those things gave me really good lessons how they are supposed to work and be used. Also, some little details I popped into were the “–build” used with docker compose up. Without it for some reason Docker didn’t use the modified .js files but the earlier versions(some caching thing), which took me some time to realize why the bug fixes I wrote didn’t apply. Also I used ctrl+c to close Docker compose app and didn’t realize it doesn’t delete the container fully. Meaning that the database data is left and available on the next session too, causing weird things. Using docker compose down – as one should do – fixed this.

Further development

- Wait-for-it.sh should be used from local files and not from Internet.
- Passwords should be hashed and not stored as plain text in database!
- Server_a should have more error managing when handling requests. For example, a simple try catch-block in every request handling to prevent errors from request containing invalid data.
- Autotests
- Maybe a system for development/production builds.
- Database should have a volume for data persistency but in this project, it was not needed to have persistent data between container restarts so it was not implemented.