Identifying information

Group name: group-yj

GitLab repository link: https://course-gitlab.tuni.fi/compcs510-spring2024/group-yj

Group members:

Name	Student number	TUNI email
Juhana Kivelä	K438863	juhana.kivela@tuni.fi
Yue Zhu	152125749	yue.zhu@tuni.fi

Note! 3rd member Tuan Nguyen (tuan.q.nguyen@tuni.fi) was assigned to this group automatically, but we were unable to get contact to him.

Timetable

Our group used 6 weeks in total for this project starting on 18th of March (Week 1) and ending on 28 of April (Week 6). Our timetable is explained in detail in Table 1.

Week number	Tasks
1	Plan the project and create initial issues, create wireframe for UI
2	Server-A framework implementation, define Swagger API
3	Frontend dummy implementation, implement Server-A, containerize Server-A and MongoDB
4	Connect frontend to Server A, implement broker(RabbitMQ)
5	Implement and containerize Server B, refine frontend
6	Finalize two-way message broker, write the documentation

Table 1 - Week-by-week timetable for project work

Responsibility areas

Responsibility areas are explained in Table 2. In summary, Juhana Kivelä was in responsibility of backend implementation and Yue Zhu was in responsibility for frontend design and implementation.

Group member	Responsibility areas
Juhana Kivelä	Server-A, MongoDB, RabbitMQ, Server-B, containerization, documentation
Yue Zhu	Wireframes & frontend design, frontend implementation, connecting frontend to Server-A, documentation

Table 2 – Responsibility areas for each group member

Agile working & Scrum

Our group followed the Scrum practises for this group work. We created initial product backlog in the beginning of the course and refined it as needed. Our sprint duration was one week, and we had sprint backlog meeting every Monday. Our group did not have daily meetings, but issues were communicated and resolved as-they-come. Both group members used 10-15 hours for this project per week.

Our group used the GitLab issue boards as Kanban board with 4 tables:

- Open as Project backlog
- Sprint backlog
- Doing
- Closed

This ensured simple, but effective way to manage and keep on track on the current tasks. No other external tools were used for managing tasks.

System architecture

Architecture for the whole application

Our group implemented the service with the instructions received from group staff with the addition to implement MongoDB to store data. Our system consists of 5 services:

- Frontend, which is implemented with React.
- Server-A, which is responsible for communication with frontend, communicating with MongoDB, sending orders to RabbitMQ and receiving ready orders from it. Server-A is implemented with Node.js and uses OpenAPI as an interface to client. openapi.yaml file is located on “backend\server-a\api\openapi.yaml”
- MongoDB, which is only coupled with Server-A. No other service is allowed to connect to it to maintain service autonomy. Responsible for storing sandwiches and customer orders.
- RabbitMQ, which acts as a message broker between Server-A and Server-B. It has two channels “received-orders” and “handled-orders”.
- Server-B, which acts as a “restaurant side” service. It mimics completing or failing the customer orders.

The service communication is explained in Image 1. Client is only able to connect to Server-A (running on localhost:3001), and Server-A then directs traffic forward as needed.

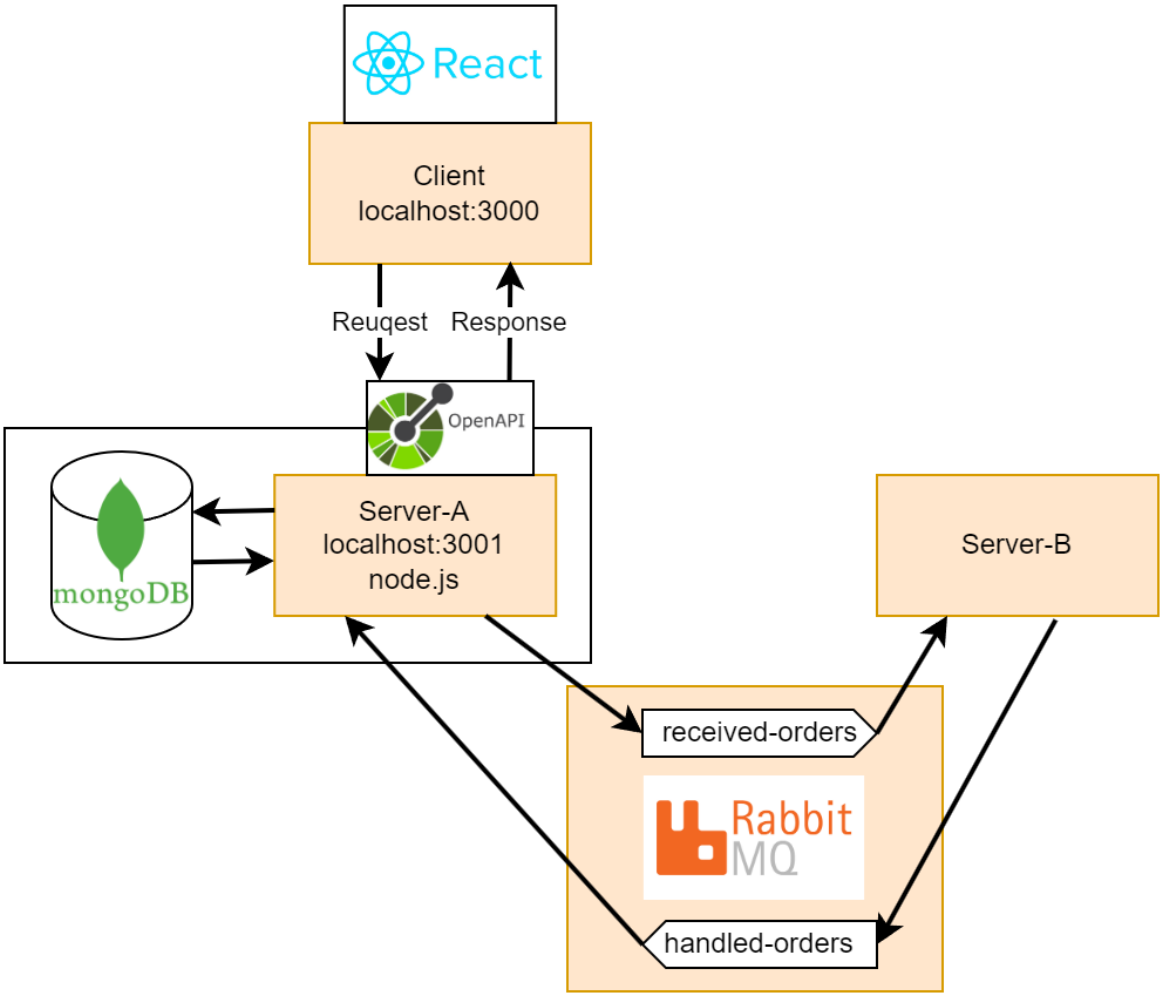

Image 1 - Illustrative image of the whole application

The backend is separated to 3 microservices: Server-A and MongoDB create one microservice, RabbitMQ creates another, and Server-B creates the last one. It is important that the backend follows service autonomy as it ensures that singular services can be scaled or replaced if needed.

Singular services are also divided into logical classes by their functionalities. This makes the maintaining of the application easier.

RabbitMQ is not defined in its own project folder, but instead Server-A and Server-B will create the needed channels if they don't exist yet. All of the RabbitMQ's configuration happens in the docker-compose.yml.

Frontend architecture :

Frontend architecture is visualized in detail in Image 2.

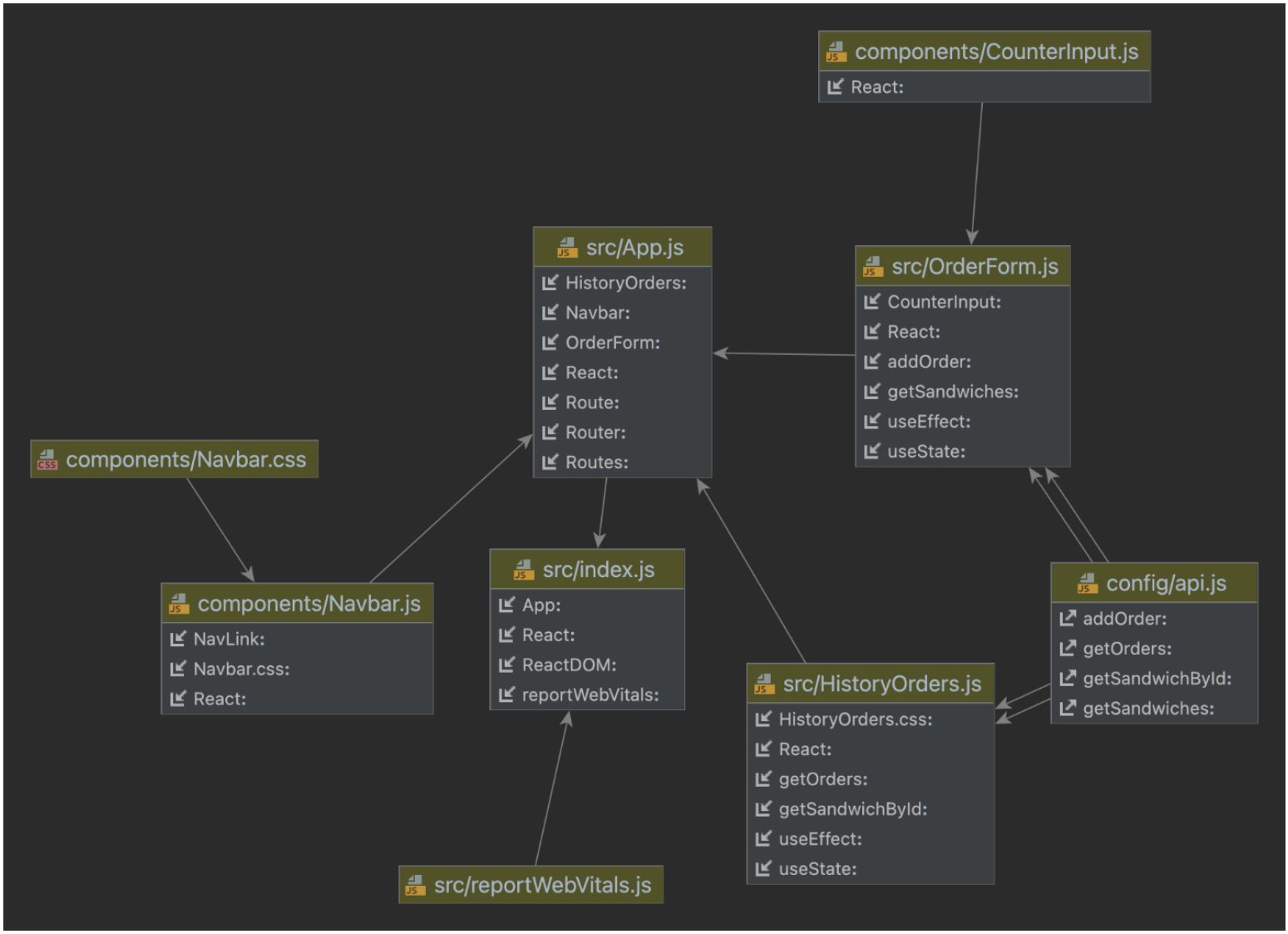

Image 2- UML diagram of the web frontend

Components:

- 'CounterInput.js', 'Navbar.js' : These are individual component files that contain functional components using 'useState' hooks for managing state within the component.
- 'Navbar.css', 'HistoryOrders.css': CSS files for styling the respective components.

Pages:

- 'OrderForm.js', 'HistoryOrders.js' : These represent different pages or views within the application. They handle the logic and state related to order forms and viewing the history of orders respectively.

API Configuration:

- ‘api.js’ : This module contains functions to interact with the backend API. These functions are used to perform HTTP requests and process responses related to orders and sandwiches.

Evaluation of the architecture

Frontend:

The positive thing about the frontend is that the number of sandwich types is not fixed. I can increase or decrease the number of sandwich types in the database, and the frontend will automatically update the menu. And users can add several types of sandwiches with different quantities in one order.

Another good point is the structure of the frontend. I put files with different functions under different directories, and files with the same functions under the same directory so that the structure is clear and well-organized.

One thing needs to be improved is that I use inline styles in 'orderForm.js', but a better practice will be to use an external CSS file to manage the styles.

- Yue Zhu

Backend:

The positive thing in the backend is the service autonomy, which divides the application into services with clear functionalities. The services are also divided to classes, which makes the development and maintainability easier. Also the OpenAPI makes Client <-> Server-B communication easier to manage, which is a plus.

If I would change one thing now, I would create some interface classes for the Server-A <-> MongoDB, Server-a <-> RabbitMQ and RabbitMQ <-> Server-B. Since right now any class is able to make direct calls for example to MongoDB, which adds complexity to the service.

Another thing that could be considered is to create configuration class for RabbitMQ, where the channels would be configured. This way Server-A and Server-B wouldn’t need to create them as it’s not their responsibility area.

- Juhana Kivelä

Used technologies

Node.js

Node was the base-layer for Server-A. The server stub was created automatically by SwaggerCodegen and it was then modified to add the needed functionalities.

React

React is used in the frontend. It's used as a JavaScript library for building the user interface. It enables the creation of reusable UI components that manage their state, leading to efficient updates and rendering of web pages.

RabbitMQ

As was recommended by the course staff, RabbitMQ is used as message broker between Server-A and Server-B.

MongoDB

Our group wanted to persist the data even if the application would crash & it'd need to reboot. MongoDB was used as it was already familiar to group members and it fits our needs well. Schemas are defined in *backend\server-a\mongo\schemas.js*.

How the application can be tested

1. Clone the repository:

   git clone git@course-gitlab.tuni.fi:compcs510-spring2024/group-yj.git

2. Launch the backend in the root directory:

   cd group-yj

   docker compose up -d

Starting the containers might take couple of minutes when doing for the first time. The terminal will be usable once all the services have started.

3. In the root directory, start the frontend:

   npm start --prefix ./frontend/

The frontend will install all dependencies automatically. A new window will open up at http://localhost:3000/ where the application can be used.

Learning during the project

I learned from various aspects during my experience with the project.

From technical aspect, I learned to create functional React components and how to manage their state. I have an enhanced understanding of asynchronous JavaScript through fetching data from APIs.

From the aspect of project management, I improved skills in structuring a project. I learned the importance of project planning and how to organize code effectively.

From the aspect of user interface and user experience, I gained insight into UI/UX design by creating an interface which is user-friendly, accessible and responsive.

From teamwork aspect, I learned the value of clear communication and collaboration when working on shared codebases. I also have a better understanding in how to use version control systems like Git effectively for collaborative projects.

- Yue Zhu

I already had previous experience in both frontend and backend so this time I wanted to focus on the "bigger picture" of the implementation and project management.

I learned to use Docker Compose to manage services and containerize them. I learned about connecting services together, what kind of challenges there can be and what needs to be considered. The biggest key-takeaways here for me is to think responsibility areas for services before implementing then and the second thing is to define clear interfaces for services.

I also learned new tools like OpenAPI and RabbitMQ. I've wanted to learn to implement message broker for longer time already, so this was a good practise. OpenAPI was also practical, and I can see its value especially if we would have more developers in the team.

During this project I worked a bit as a senior developer or product owner, so I assisted in any issues we encountered and did the Scrum part of group work. This was great experience and boosts my actual hands-on knowledge.

- Juhana Kivelä