Project documentation - Group 1

Project plan

Group info

The group has 3 members:

- Hung Anh Pham (Harry), H281246, hung.pham@tuni.fi
- Duy Anh Vu (Andy), H294381, duy.a.vu@tuni.fi
- Duc Hong, H292119, duc.hong@tuni.fi

Working during the project

Group members promised to spend at least 5 hours per week working on this project. Below are each member's responsibilities. Of course, besides these responsibilities, members will help out each other when necessary. We also use the Gitlab issue board to keep track of the tasks.

- Harry will be responsible for server B, configuring RabbitMQ & backend Docker Compose, taking care of designing the UI if needed.
- Andy will be responsible for implementing server A (sandwich API, user login, database, etc.)
- Duc will be responsible for developing the frontend of the application.

Used technologies

- The application is built using NodeJS
- React is chosen as the frontend library. MaterialUI will also be used to take advantage of predesigned components.
- The database service of choice is MongoDB, communicating with the NodeJS backend through Mongoose.
- RabbitMQ is the messaging service used for communication between server A and B. Amqplib is the library used to enable communication between NodeJS and RabbitMQ.
    - Alternative message brokers are also available to be considered instead of RabbitMQ, for example, NATS. NATS is more optimised for performance and ease-of-use, but not as reliable in terms of guaranteed delivery and persistence.
- Docker takes care of the application's services containerisation.
- Socket.io for maintained, bidirectional connection between server A and the frontend.

More details on used technologies are discussed in the following sections.

System architecture

General architecture

Below is the architecture overview diagram with the most important components. The application has a frontend and a backend. Frontend is a React application running in one Docker container, while backend consists of three Docker containers: server A, server B, and RabbitMQ; with server A connected to a MongoDB database.

Server A's main purpose is to handle communication between the frontend and the database, while also communicating with server B through the message broker RabbitMQ. Server B takes care of "preparing" the sandwich, which for the sake of simplicity in this project, is basically a delay of 10 seconds.

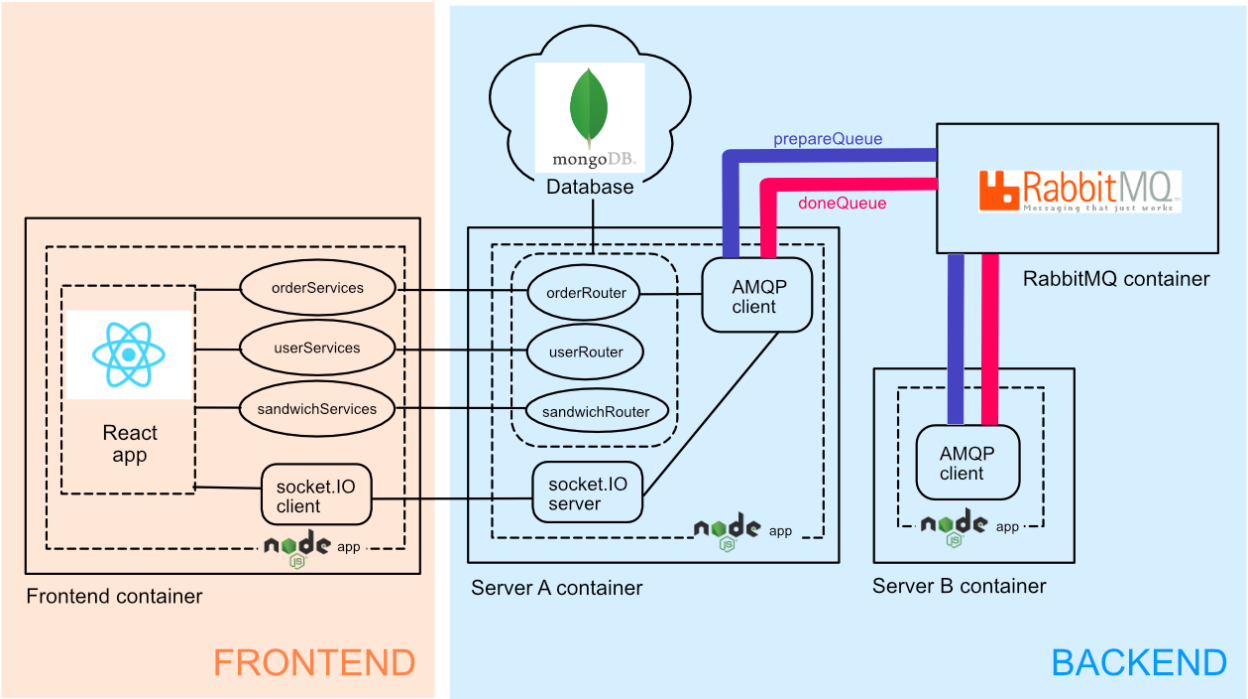

Figure 1: *Architecture of the application*

The frontend's three services are connected to server A's three routers to enable communication between the user interface and the database regarding sandwich orders, sandwich types, and user management. The connection between two socketIO clients enables the frontend to adjust sandwich order status when it is updated on the backend.

In order for server A and server B to communicate through RabbitMQ, a client for AMQP (Advanced Message Queuing Protocol) - the core protocol of RabbitMQ - needs to be present on both servers. There are two RabbitMQ message channels: prepareQueue and doneQueue. Server A will send a new sandwich order to server B through the prepareQueue, on which

server B listens, receives, and processes. After processing is done, server B sends back the finished sandwich through the doneQueue, which would be received by server A. Server A will then update the order status.

As it can be seen in the architecture diagram, each “service” of the system runs in its own Docker container. Each container (except server B, which does not need to communicate through its port) is exposed to the local network by a unique port number:

- Frontend’s exposed port is 3000 and is mapped to port 3000 on the host machine.
- Server A’s exposed port is 3001 and is mapped to port 3001 on the host machine.
- RabbitMQ’s exposed port is 5672, while its management plugin’s port is 15672.

Frontend

The frontend uses ReactJS components to render the application. We also use a React router to route between pages. This means the frontend is a Single Page Application, which only refreshes partially when and where needed, not the whole page. There are 4 pages:

- **Homepage** (path: “/”) displays all the sandwiches available for ordering.

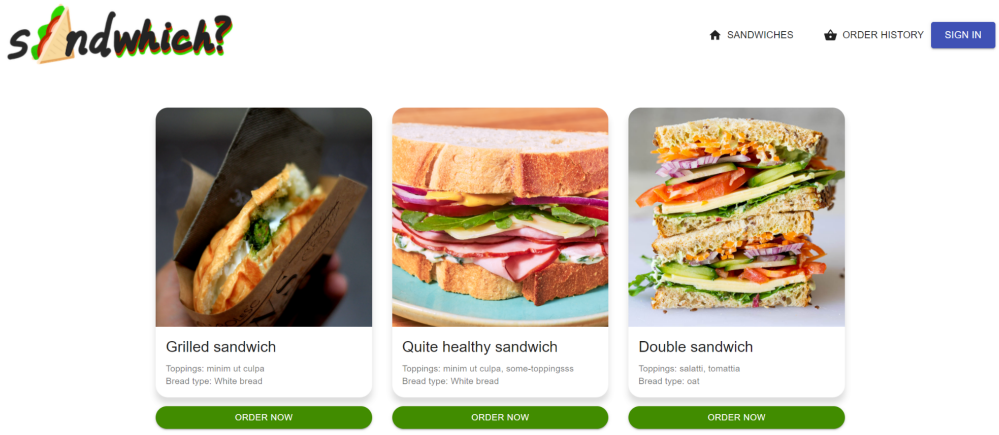

Figure 2: *The homepage*

- **Orders** (path: “/orders”) displays all the orders and their status.

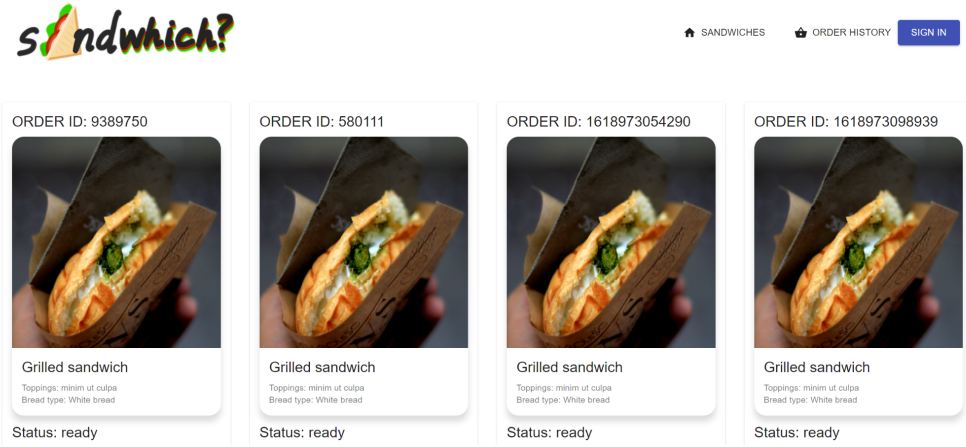

Figure 3: *The orders page*

- Sandwich Control (path: “/sandwiches”) displays two tabs, one for adding sandwich and one for modifying a sandwich.

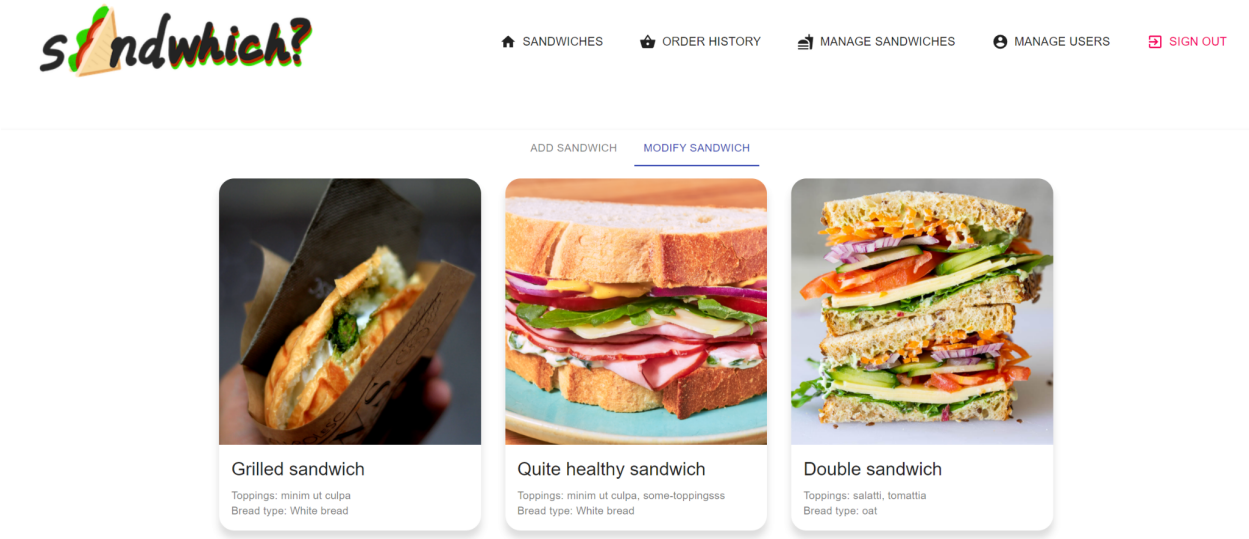

Figure 4: *The sandwich control page (only visible to signed in users)*

- User Control (path: “/users”) displays three tabs. One for getting/deleting user by username, one for adding user, and one for modifying user.

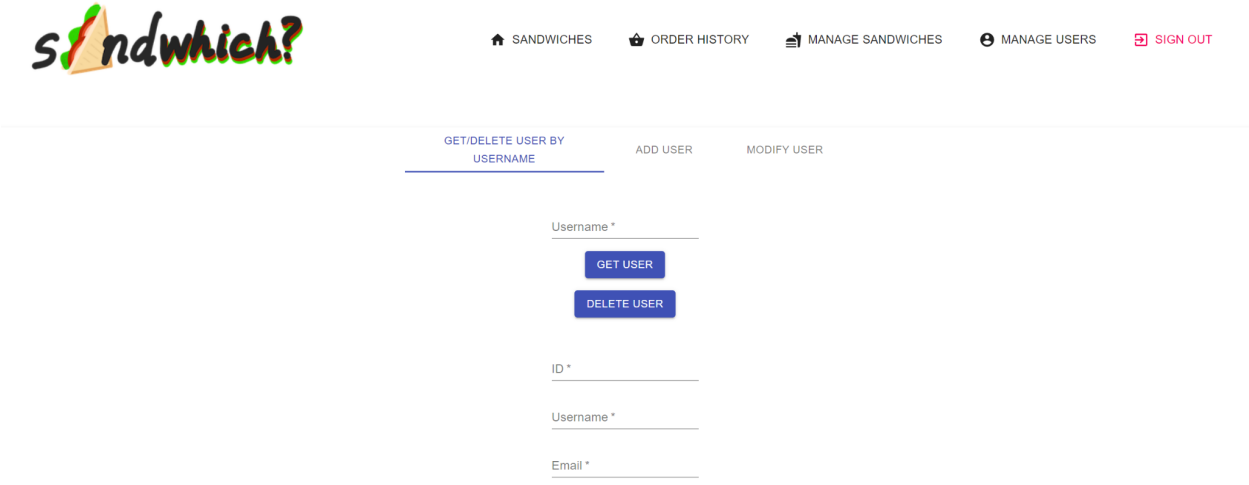

Figure 5: *The user control page (only visible to signed in users)*

Besides, the app contains different forms for users (get/delete/add/modify user) and (modify/delete/add sandwich). The login form is also given for the user to log in to the application.

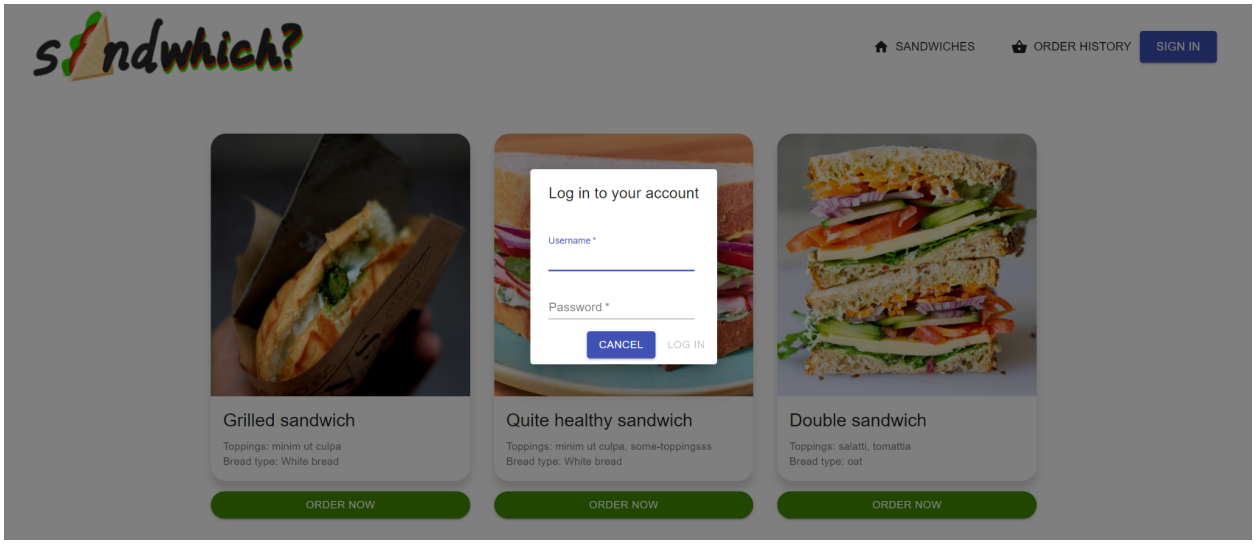

Figure 6: *The login form.*

The react router provides a way to “dynamic routing” in the application, which means the routing takes place as the application is rendering. As we want the headers and other common parts of the page shouldn’t be re-rendered when the user navigates between pages, routing between the pages only changes the body of the page. We also use react history to routing between pages, for example, when the user hits the logout button or hitting the buttons to navigate between pages.

On top of that, we use React components as the main technology in the frontend. The React component is built on the modular structure and composition relationship, in which we can build grained components and reuse them and build larger components on top of them. Each component has its own properties and states which can be shared within components. The component would automatically re-render when its states or properties change. We utilize the ready-made components from Material-UI library which provides a lot of well-implemented and nicely-documented components with a good user interface.

The frontend is implemented with 3 main parts. The first part is the services that fetch data from the backend. The services are implemented based on the Swagger documentation. The second part is the grained ReactJS components that help display the pages. The final part is the pages which are built on top of the components. Everything comes together inside the common parent component called “App”.

The frontend also uses the websockets-like technology to connect with the backend side when the backend wants to push the notification of the order status. The library used was SocketIO, providing a bidirectional connection. In this application, socket.IO was used for the backend side pushing notification of the order status to the frontend side to update and display the status of the orders in real-time.

How the frontend works:

First, when the page is accessed, the data of sandwiches and orders are immediately fetched to the client. These are owned by the app and distributed to other components for displaying relevant information about sandwiches and orders. Whenever a new request is made, the changes will be updated both on the backend side and the client side. After logging in, the user has full control of all the operations, and there are the forms for the user to fill in to fetch the desired information. The socket.IO is set up when the app starts, and when the socket.IO on the backend side pushes the notification, the frontend can change the status of the order immediately and notify the change by a snackbar. In short, when routing pages and making requests, the page is not refreshed, but updates the states and properties of the components instead.

Server A

Server A is a Node server built using ExpressJS framework. This server is responsible for communication with frontend, database (mentioned in the next section), and message broker. When thinking about old well-known MVC architecture, server A contains the **M**odel and the **C**ontroller parts. Server A is built to follow REST(Representational State Transfer) architecture.
Server A's controllers resolve requests on user, sandwich, and order. The request actions include **C**reate, **R**ead, **U**pdate, and **D**elete. For instance, a user wants to order a sandwich, the request with Create method will be sent to server A from the frontend, and the order will be created and persist in the database.

A big picture of main components in server A has been described in **Figure 1** above. In details, the components of server A and technologies used are:

- User controller: responsible for handling requests on user
- Sandwich controller: responsible for handling requests on sandwich
- Order controller: responsible for handling requests on order
- Models: MongoDB schema used for communication with database (using mongoose)
- RabbitMQ utilities: contain a function to send message to the message broker and a function to listen to the message from the message broker
- Helper functions: helping functions. Currently, they are API key generator and API key verificator.
- Middlewares: We use 3 middlewares currently
    - CORS: handle Cross-origin Resource Sharing so that the frontend can send request to server A, which is running on different origin (different address)
    - JSON: this parses the request payload from string into JavaScript object
    - A middleware to attach socket.io object into request object for use inside controllers
- Socket.io: This creates and maintains a connection between server A and frontend so that the new status of order can be updated to the frontend. This is very similar to websockets.

- Bcrypt: this is used to hash the password and to compare the password and the hashed password later on.
- AMQP lib: used to handle communications with RabbitMQ

For some operations from sandwich and user category, the authorization is done based on the API key given to the user when they logged in. This API key is sent inside the “X-API-KEY” header of the request from the frontend. For more details on the operation handled by the controllers and the detailed request/response, please refer to the Swagger documentation YAML file located in the root folder of the project. Note that this Swagger has been modified from the course’s template to suit our needs.

Database

MongoDB is utilized in this project. However, we use MongoDB Atlas, a cloud service, so there is no need to install this on your computer or create a separate container for it.
The database will store everything related to order, sandwich, and user. API keys are attached to corresponding users in the database. The password is hashed before it is saved to the database.
Once the user logs out, their API key will be resetted to empty in the database.

RabbitMQ

RabbitMQ is the message broker in this project. It manages and orchestrates messaging between server A and server B through both channels prepareQueue and doneQueue. These channels are created by server A and B, after those servers have connected to RabbitMQ through its container’s exposed port. The container is built using the official RabbitMQ image on Docker Hub.

Server B

Server B is a Node application that listens to new messages (sandwich orders) from the prepareQueue, sent by server A. After receiving a new order, server B processes it by giving a 10-second delay before assigning an order status (ready/failed) and send it to the doneQueue. The order should be received by server A, which listens on the doneQueue. These receiving & sending operations take advantage of amqlib, the NodeJS client that enables communication with RabbitMQ.

Figure 7 below describes how data flows when a new order is added.

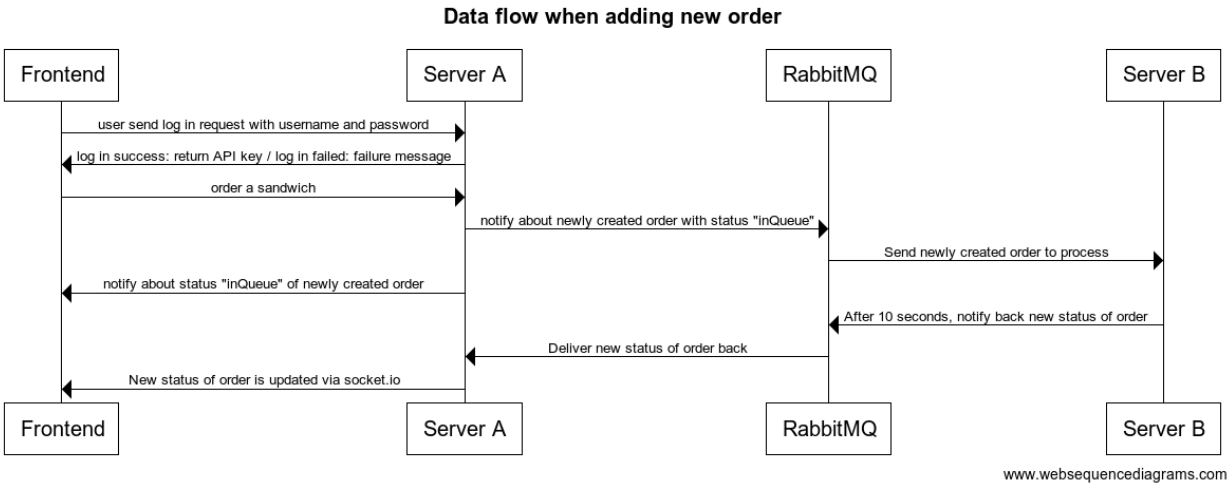

Figure 7: *A typical data flow when a user adds a new order*

Note that new order statuses are continuously updated to the database by server A.

How to test the system

1. First, pull the repository from here using Git.
2. Run Docker service on your machine if you have not done so.
3. Locate your terminal directory to the root folder of the project
4. Run `docker-compose up --build`. The flag `--build` forces the Docker images to rebuild even if they have already existed.
5. Wait until the frontend shows that it is ready from the terminal. This process might take a while in the first run. It is ready when you see something like this

```
Compiled successfully!

You can now view reactapp in the browser.

  Local:            http://localhost:3000/
  On Your Network:      your IP address    :3000/

Note that the development build is not optimized.
To create a production build, use yarn build.
```

6. From your machine, open your browser and locate yourself to http://localhost:3000 to use the application.
7. Without logging in, you can use only the order functionalities and look at the available sandwiches. To log in, please use one of the credentials mentioned in the next section.

User credentials

The app can be logged in with any of the following user credentials:

username	password
webdev	simple
webdev2	simple
admin	admin

After login with this account, we can do all the operations documented in Swagger documentation with a user-friendly GUI.

Learning during the project

This section will have separate subsections from each member of the group, containing their own learning experience from the assignment.

Harry

Personally, I learnt quite a lot during this project. I have some experience with web applications and Docker earlier, but this project certainly helps concretise and further advance my knowledge in these areas. RabbitMQ is completely new to me, so having the chance to learn the platform and implement it in a wider system was a nice experience as well. With this project, I now have a deeper insight into how different components in a web application are designed, developed and configured. This is my main goal of coming to this course, so it is certainly fulfilling.

Andy

For me, this is a really nice project. Previously I have some concrete experience on React and ExpressJS but not on message broker or Docker Compose. I have done some Docker previously, but not something like Compose. This project definitely helps me to further improve my skills. In addition, I also get an opportunity to try websockets. I have heard about it plenty of times but have never touched it.
Most importantly, I learn about dividing components based on a good architecture. This is also the purpose of the course so I am happy with it.

Duc

I learnt a lot from the project. The project improves my ability to work on the client side, which is not my strength at the moment. I have tried using ReactJS components before but now I can practice building a reasonable application that I have to think about designing the whole

application. This is the first time I also work with websocket, which enables me to try different ways of communication between backend and client side. The project helps me practice and deal with the real problems when developing an application.