# COMP.CS.510 – ‘Make me a sandwich’.

Authors: Esa Särkelä, Heidi Seppi and Mohammed Al-Gburi

## Architecture

The application uses the client-server pattern, where service consumers (clients) connect to the service provider (server) via the internet. Communication between frontend and backend is done via a HTTP requests using a RESTful API that is further defined and documented in the server-a swagger API documentation. The server uses the microservice pattern, in particular asynchronous messaging design pattern, where distributed backend components communicate asynchronously trough a message queue. A high-level view of the architecture is described in figure 1.

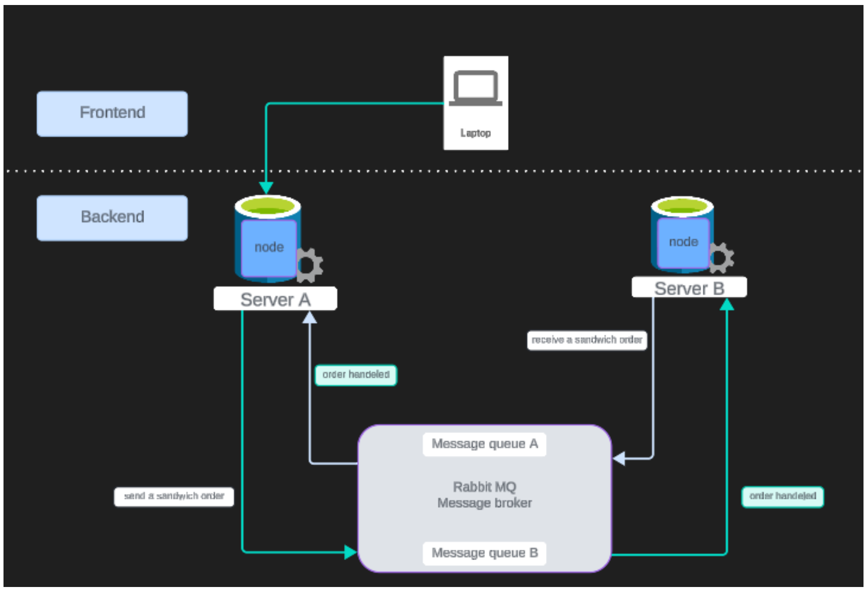


*Figure 1: High-level view of the overall application architecture. A distributed backend communicates with the frontend client via a RESTful API interface.*

### Frontend

The frontend or client side application is a single-page application (SPA) consisting of a few conditionally rendered views: a login view, a registration view, and the sandwich-ordering view. In the sandwich-ordering view, the application polls the backend server-a every second to receive updates on the sandwich statuses. Frontend is deployed using a nginx -web server. The frontend uses React as its primary front-end library, CSS for styling and Axios/fetch-api to perform HTTP requests to the backend. The application uses JSON web tokens for access management, which are stored in localStorage.

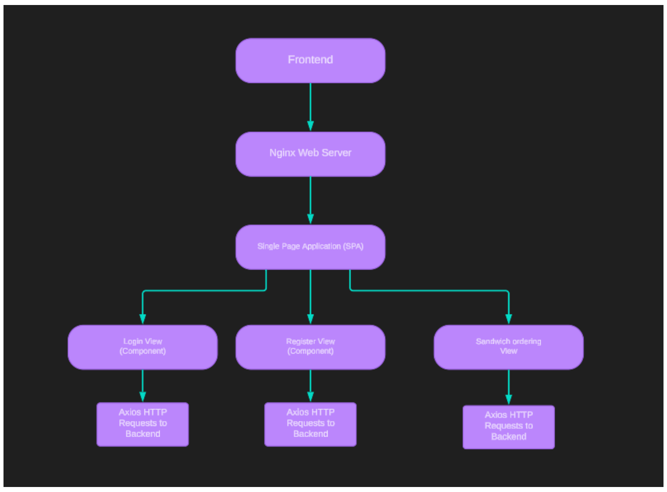

Frontend
Nginx Web Server
Single Page Application (SPA)
Login View (Component)
Register View (Component)
Sandwich ordering View
Axios HTTP Requests to Backend
Axios HTTP Requests to Backend
Axios HTTP Requests to Backend

*Figure 2: High-level view of the front-end architecture. showing the components in a single page application.*

## Backend

The backend consists of two independent servers, server-a and server-b, which communicate asynchronously via an instance of the RabbitMQ AMPQ (advanced message queuing protocol) message broker.

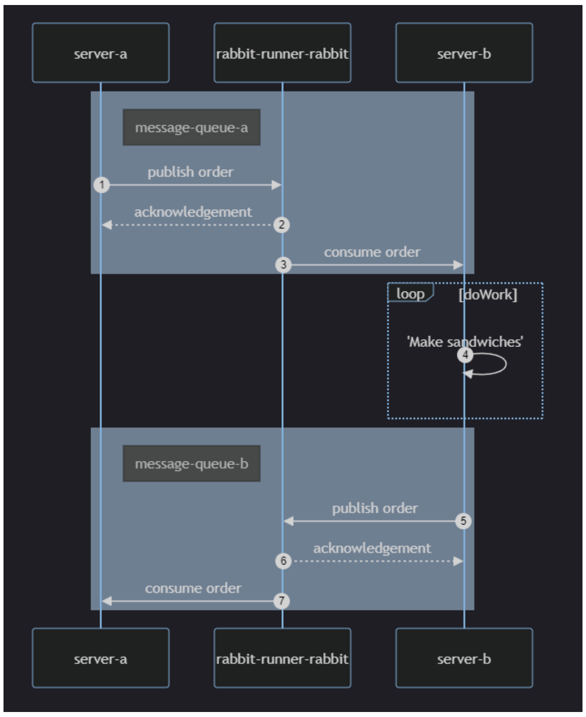


*Figure 3: Sequence diagram of backend. Distributed components server-a and -b exchange messages through message queues A and B.*

### Server-a

Server-a implements a RESTful API that has endpoints to manage users, sandwiches, and orders via CRUD operations. Server-a uses Node.js as the runtime environment, Express.js web framework for request handling, Mongoose object-relational mapping, JSON Web Tokens for authentication and authorization, middlewares to log requests, handle errors, authorize users and validate API keys. Adding and removing sandwiches require a pre-defined API-key in the X-API-KEY header of the request. The key is ‘sekret’. Server-a sends order tasks to message queue A and consumes messages from message queue B.

Server-a can be tested using Swagger (http://localhost:8080/docs). Pre-define API-key can be inserted using the “Authorize” button on the top of the Swagger documentation. Notice that endpoints which require JSON Web Tokens for authorization, can’t be tested with Swagger at the moment. This functionality could be added later on Swagger documentation to improve testing.

### Server-b

Server-b consumes orders from message queue A. Server-b is rather simple and only processes the orders by changing their state to ‘received’ on consuming them and ‘ready’ on completing them after 5 second an interval. Server-b publishes the altered orders on message queue B.

### RabbitMQ

The backend implements two RabbitMQ work queues that are used to send and receive orders between servers asynchronously. Server-a sends placed orders to message queue A from with server-b consumes them. Server-b sends processed orders to message queue B from which server-a consumes them.

### Database

A single MongoDB document database instance is used for storing app data: users, sandwiches, sandwich toppings and orders. A directory is mounted for the MongoDB container during deployment and any changes to the data will persist between restarts. The database is dropped and reseeded if there are no sandwiches in the database.

## Manual (how to try)

### Running the application

Docker and Docker Compose are used for project deployment. Clone the repository to your local machine, navigate to root directory, build, and mount the Docker images.

```
git clone https://course-gitlab.tuni.fi/compcs510-
spring2024/crazepiano.git \
    && cd crazepiano \
    && docker-compose up
```

### Accessing the application

Once the application is running, you can access it as follows:

Server-a: http://localhost:8080. The backend API is used to place sandwich orders. Documentation available at the http://localhost:8080/docs endpoint

Server-b: Does not expose any ports to the host machine.

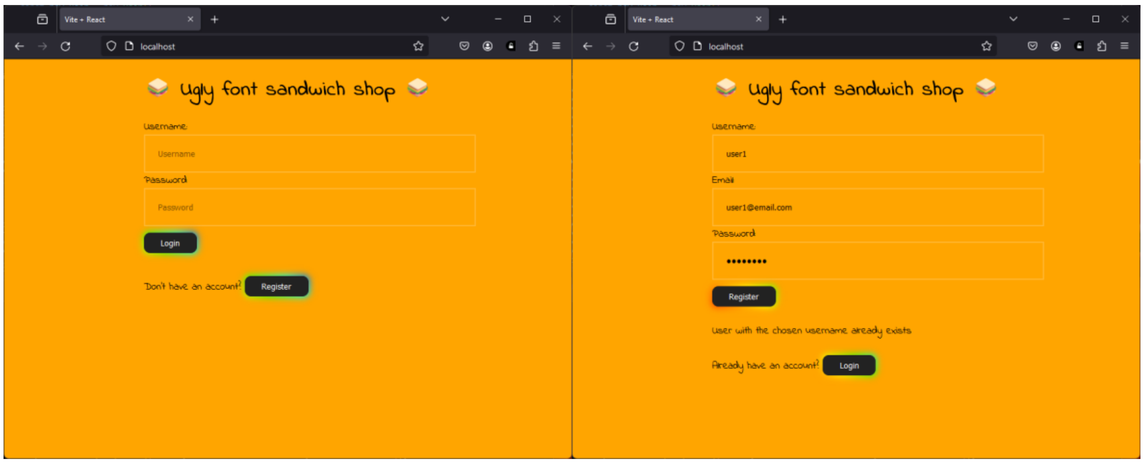
**Frontend: http://localhost:80. React application hosted on nginx. This is the intended interface that the end-users are to interact with.**

RabbitMQ: Does not expose any ports to the host machine.

MongoDB: Does not expose any ports to the host machine.

## Using the application

In the initial login view, the user can either login using existing credentials in the supplied form or register a new account by pressing the register button.



user1

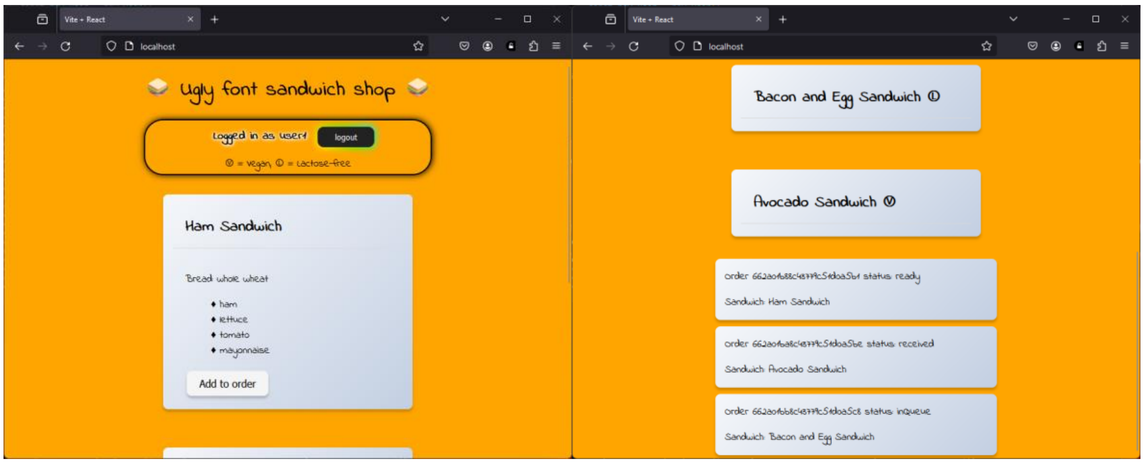
user1@email.com

••••••••

User with the chosen username already exists

*Figure 4: Left: login view. Right: register view. Any errors during registration or login will be displayed below the respective forms.*

On successful login, the user is presented with the sandwich ordering view, where they can order a selection of pre-defined sandwiches. By clicking on the sandwich card headers, the cards can be expanded, and orders can be placed by pressing the add to order -button. The orders are placed immediately upon clicking the button. The users' active and previous orders can be seen at the bottom half of the view and their status is updated automatically. The user can place as many orders as they wish, since our kitchen seems to work as fast as a computer CPU. The user can press the logout button to destroy their session token and return to the login-view.



Ugly font sandwich shop

Logged in as user1

Ⓥ = vegan, Ⓛ = lactose-free

Ham Sandwich

Bread: whole wheat

- ham
- lettuce
- tomato
- mayonnaise

Bacon and Egg Sandwich Ⓛ

Avocado Sandwich Ⓥ

order 662a01b88c48779c54d0a5b1 status: ready

Sandwich: Ham Sandwich

order 662a01ba8c48779c54d0a5be status: received

Sandwich: Avocado Sandwich

order 662a01bb8c48779c54d0a5c8 status: inqueue

Sandwich: Bacon and Egg Sandwich

*Figure 5: Sandwich ordering view. Left: a sandwich card expanded to display ingredients and the ordering button. Right: various orders, with different order states and sandwiches.*

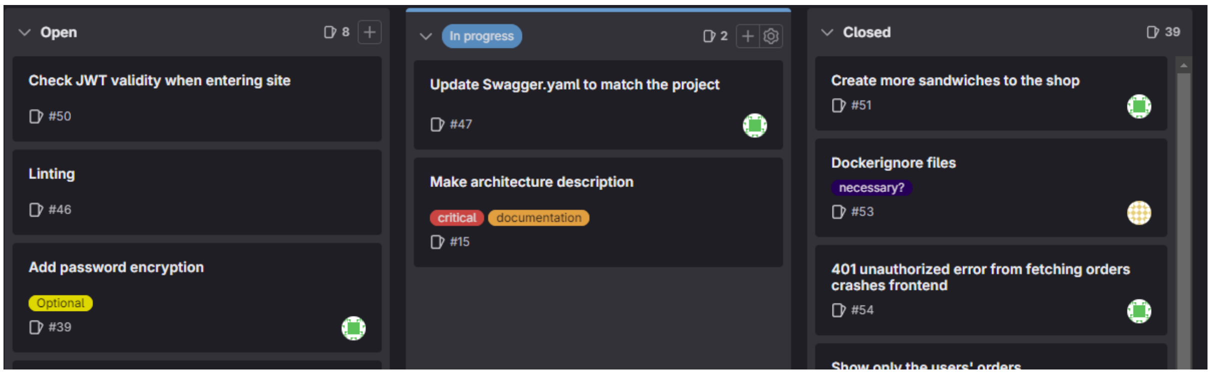
## User credentials

The application seeds the following credentials on boot:

| username | password | email |
|---|---|---|
| alice | daisy | alice@email.com |
| bob | builder | bob@othermail.com |
| user1 | password | user1@useremail.com |

## Project management

We used Git for version control, GitLab issue board for project management, did weekly Zoom meetings, and used Telegram for daily messaging. New features were implemented using Git feature branch workflow where development of each new feature took place in an independent branch and was merged to the main branch when ready.



Open 8

Check JWT validity when entering site
#50

Linting
#46

Add password encryption
Optional
#39

In progress 2

Update Swagger.yaml to match the project
#47

Make architecture description
critical documentation
#15

Closed 39

Create more sandwiches to the shop
#51

Dockerignore files
necessary?
#53

401 unauthorized error from fetching orders crashes frontend
#54

Show only the users' orders

*Figure 6: Snapshot of the GitLab issue board on 2024-05-24.*

## Issues

1. Some browser extensions can affect the use of the app. Desktop Google Chrome/Firefox was used exploratory testing and development.
2. Sometimes an expired token is stored in the browser that may cause the app to malfunction. Use localStorage.removeItem('token'); to make a new session.
3. Using localStorage for credentials instead of cookies
    a. can't set expiration time
    b. vulnerabilities to various attack vectors
4. If an order hangs on intermediate states such as received, they are never resolved properly. This can happen for example when restarting the application when an order is being processed.
5. In some machines rapid-runner-rabbit (RabbitMQ) might not start fast enough so the docker-compose up fails. In this case just running docker-compose up again should start the containers normally.