Tampereen yliopisto
Tampere University

Web Development 2 – Architecting

FINAL PROJECT DOCUMENTATION

Xuan-An Cao

Nhan Chau

Faculty of Information Technology and Communication Sciences

2024

TABLE OF CONTENTS

1. Project Plan

1.1. Course Project Group Information

- Group name: Vingroup
- Group members:
    - Xuan-An Cao, student ID: 151454314 (an.cao@tuni.fi)
    - Nhan Chau, student ID: 150557652 (nhan.chau@tuni.fi)
- Group Gitlab repository: https://course-gitlab.tuni.fi/compcs510-spring2024/vingroup.

1.2. Working Distribution During the Project

1.2.1. Project Timeline

The initial project timeline was set as follows:

Work Issues	**Time**
Technical research (AMQP, Docker…)	24.3 – 31.3
Backend implementation	1.4 – 12.4
Frontend implementation	13.4 – 26.4
Refactor, finalization and documentation	27.4 – 28.4

It should be noted that this is the initial timeline set by members before starting the project, and there might be changes on the actual progress based on both internal and external factors. Please check the Project Logs section for more up-to-date information.

1.2.2. Work Distribution

The following table describes the work distribution of the group members. Estimately, each group member spends around 2 hours/day on the project, which amounts to 10 hours/week, plus untracked weekend working hours.

Personnel	Issues	Issue Groups
Xuan-An Cao	Implementing RESTful API requests, data models, data access and AMQP communication between servers	Backend
Nhan Chau	Implementing WebSocket for order communications and dockerizing the backend	
Xuan-An Cao	• Implementing Home, Sandwich and Order components • Handling state management using RecoilJS; handling custom hooks and API services • Handling UI/UX styling using Ant Design libraries and Emotions styled components	Frontend
Nhan Chau	• Implementing Navbar, Login and Register components • Handling WebSocket for real-time communication between servers and frontend	

1.2.3. Project Logs

The working logs can be tracked with Gitlab commits on the

Time	Work
31.3	Finished technical researching and preparation for project initiation.
2.4	Set up Node project for server A and improved upon provided project Swagger API.
3.4	Finished implementing server A backend (routes, controllers, models).
6.4	Handled server A and server B communication using AMQP library.
7.4	Created temporary project documentation and Swagger API documentation, ready for mid-project check-in
9.4	Dockerized the project.
11.4	Fixed occurred bugs and created database reset mechanism. **Wrapping up backend implementation.**
13.4	Set up React + TypeScript + Vite project for frontend and implemented API services.
15.4	Handled state management with RecoilJS.
17.4	Handled styling configuration and implemented Home, Navbar and Sandwich routes.
19.4	Implemented Auth routes (Login, Register), and partially implemented Order route.
20.4	Finalized implementing Order route.
22.4	Set up communication between frontend, servers and RabbitMQ with polling technique.
23.4	Implemented Order List route.

26.4	Handled WebSocket for real-time communication between frontend and servers and RabbitMQ.
27.4	**Fixed occurred bugs and finalized the project code.**
28.4	Created final documentation for the project.

2. Project Documentation

2.1. Project Set Up and Initiation

To initiate the project on local machine, please ensure that Docker Desktop has been installed, and follows these steps:

1. **Clone the repository**:

```
git clone https://course-gitlab.tuni.fi/compcs510-spring2024/vingroup.git
```

2. **Adding essential credentials**:

Creating an .env file under /backend/server-a and provide these credentials in this format:

```
NAME=VALUE
```

Name	**Value**
MONGODB_URI	mongodb+srv://vingroup:p47m2N0Vw8FghzrH@sandwichcluster.f3l1aj0.mongodb.net/?retryWrites=true&w=majority&appName=SandwichCluster
PORT	8080
JWT_SECRET	FBISPECIALAGENTDALECOOPER

COOKIE_SECRET	FIREWALKWITHME

3. **Run docker compose**:

```
cd vingroup

sudo docker-compose up --build
```

   Note that the `build` flag is for the first run of the docker; if there is no change to the backend code after the first run, then the flag can be removed.

4. **Start frontend**

```
cd frontend

npm install

npm run dev
```

   The application can be accessed on http://localhost:5173/.

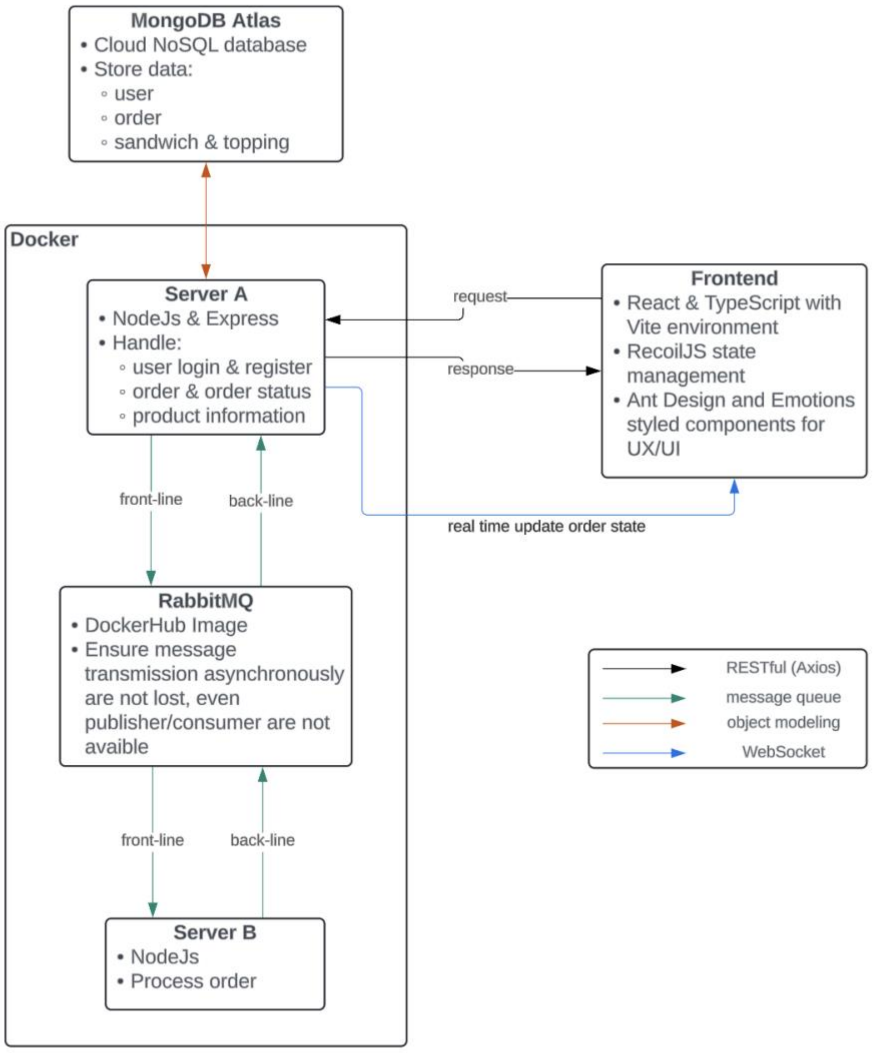
To reset the database to its initial state:

```
cd backend/server-a

npm run reset-db
```

The initial database can be retrieved at /backend/server-a/db/json.

2.2. System Infrastructure and Used Technologies



There are four main components of the system architecture: Node servers (server A and server B), RabbitMQ message broker, Mongo DB Atlas Database, and React-based frontend.

2.2.1. Servers

In the project architecture, Server A serves as the primary backend server, managing routing, controllers, and data models to handle incoming requests and orchestrate the application's core functionality. On the other hand, Server B operates as a standalone entity with a specific focus: receiving order information and publishing completed orders. This division of labor allows for a clear separation of concerns, with

Server A handling the broader aspects of application logic and Server B specializing in a distinct, crucial task within the system.

Both servers are implemented using **Node.js**, a powerful server-side JavaScript runtime environment. It allows developers to execute JavaScript code outside of a web browser, enabling server-side scripting and empowering the creation of highly scalable and efficient network applications. Node.js operates on a non-blocking, event-driven architecture, making it particularly well-suited for building real-time applications that require high concurrency and responsiveness. Its rich ecosystem of libraries and frameworks further enhances its versatility, facilitating the development of a wide range of applications, from web servers and APIs to microservices and IoT devices. Server A is also equipped with **Express**, a minimalist web framework for Node.js, designed to simplify the process of building web applications and APIs. It provides a robust set of features for routing, middleware management, and HTTP utility methods, allowing developers to create scalable and efficient server-side applications with ease. With its lightweight and flexible architecture, Express streamlines the development process, enabling developers to focus on writing clean and maintainable code while delivering high-performance web solutions.

Additionally, for user authentication purposes, server A also uses **bcryptjs** and **jsonwebtoken**. bcryptjs plays a vital role by securely hashing passwords, safeguarding user credentials against potential breaches, even in the event of a compromised database. This helps thwart potential security breaches, as even if the database is compromised, the hashed passwords remain cryptographically secure. On the other hand, jsonwebtoken simplifies the generation and validation of JSON Web Tokens (JWTs), enabling efficient stateless authentication. Together, bcryptjs and jsonwebtoken provide a robust foundation for implementing secure authentication workflows, mitigating risks associated with unauthorized access and data breaches in web applications.

2.2.2. RabbitMQ Message Broker

The two servers communicate through a **RabbitMQ message broker**, with the **AMQP library** ensuring the AMQP 0-9-1 protocol. RabbitMQ enables decoupling of application components by acting as an intermediary that receives, stores, and routes messages between producers and consumers. The AMQP library provides a set of programming interfaces and utilities for interacting with RabbitMQ, allowing developers to integrate message queuing functionality seamlessly into their applications while abstracting away the complexities of the underlying protocol. An alternative to the AMQP library is ZeroMQ (ZMQ), a lightweight messaging library that emphasizes high performance and low latency communication patterns. Unlike AMQP, which is a standardized protocol for message queuing, ZeroMQ provides a more flexible and lightweight framework for building custom messaging solutions. However, as AMQP is well-suited for complex distributed systems requiring features like message persistence and routing, it is chosen as the main communication tool of this project.

Whenever customer create and send an order request, server A transmits the data received through a frontline message queue to server B, where it is kept on hold for about 10 seconds to mimic the waiting time normally seen in a food store, and then fired back to the server A through a backline message queue, with the order status updated as finished.

2.2.3. MongoDB Atlas Database

The project opts for **MongoDB** as its data storage solution, utilizing MongoDB Atlas for cloud-based database management. MongoDB is a leading NoSQL database that offers a flexible and scalable solution for storing and managing unstructured or semi-structured data. Unlike traditional SQL databases, which rely on rigid schemas and tabular structures, MongoDB utilizes a document-oriented model, where data is stored in flexible JSON-like documents. This schema-less approach allows for dynamic and agile development, as schemas can evolve over time without requiring costly migrations. MongoDB also boasts powerful features such as secondary indexes, rich query capabilities, and automatic sharding for horizontal scaling, making it well-suited for modern applications with diverse and evolving data needs.

The project’s choice to use NoSQL databases like MongoDB over traditional SQL databases often stems from the unique requirements of modern applications. NoSQL databases offer greater flexibility and scalability, allowing developers to store and query diverse data types at scale without the constraints of fixed schemas. In NoSQL environment, developers commonly employ pure JavaScript to execute data operations. This entails expressing queries and manipulations directly within the application's codebase, seamlessly integrating with its overall logic. Conversely, in traditional SQL setups, developers predominantly rely on SQL queries, which adhere to a specific query language, to interact with the database.

The project also chooses **MongoDB Atlas database**, a cloud solution instead of a local one, since it offers numerous advantages for developers and organizations. By leveraging MongoDB Atlas, teams can offload the operational overhead of managing and maintaining database infrastructure, allowing them to focus on building and scaling their applications. Atlas provides automated backups, monitoring, and security features out of the box, ensuring data durability, availability, and compliance with industry standards. Moreover, MongoDB Atlas offers seamless integration with cloud providers like AWS, Azure, and Google Cloud Platform, enabling easy deployment across global regions for improved performance and reliability.

Finally, to communicate with the server A, **mongoose** library is used to streamline interactions with MongoDB databases in Node.js applications. It is a standard utilization in most MongoDB – Node projects, as the library provides a schema-based solution, allowing developers to define data models with clear structures and validation rules, akin to traditional relational databases, as well as simplifying common tasks such as data manipulation, validation, and schema management.

2.2.4. Frontend

The frontend is built on top of the **React** library, one of the most popular frontend developing frameworks/libraries. It revolutionizes the way developers create interactive web applications by introducing a component-based architecture and a declarative programming paradigm. React's component-based approach enables developers to break down user interfaces into modular, reusable components, each responsible for rendering a specific part of the UI. These components can be composed and nested within each other, facilitating a hierarchical structure that mirrors the UI's complexity. Additionally, React's declarative nature allows developers to describe the desired UI state, and React takes care of updating the DOM efficiently to reflect these changes. This declarative approach simplifies the process of building and maintaining complex user interfaces, as developers can focus on writing UI logic rather than dealing with low-level DOM manipulation. React's virtual DOM also optimizes performance by only updating the parts of the UI that have changed, leading to faster rendering and a smoother user experience.

HTTP requests from the frontend are supported by **Axios**. It provides a simple and elegant API that supports features such as request and response interception, automatic transformation of JSON data, and the ability to cancel requests. Axios allows developers to easily perform asynchronous operations to fetch data from external APIs, interact with servers, or send data to a backend. Its promise-based architecture simplifies error handling and enables the use of modern JavaScript features like async/await for writing clean and concise code. Axios is widely adopted in web development due to its versatility, ease of use, and robust feature set, making it an essential tool for managing network requests in frontend and backend applications alike.

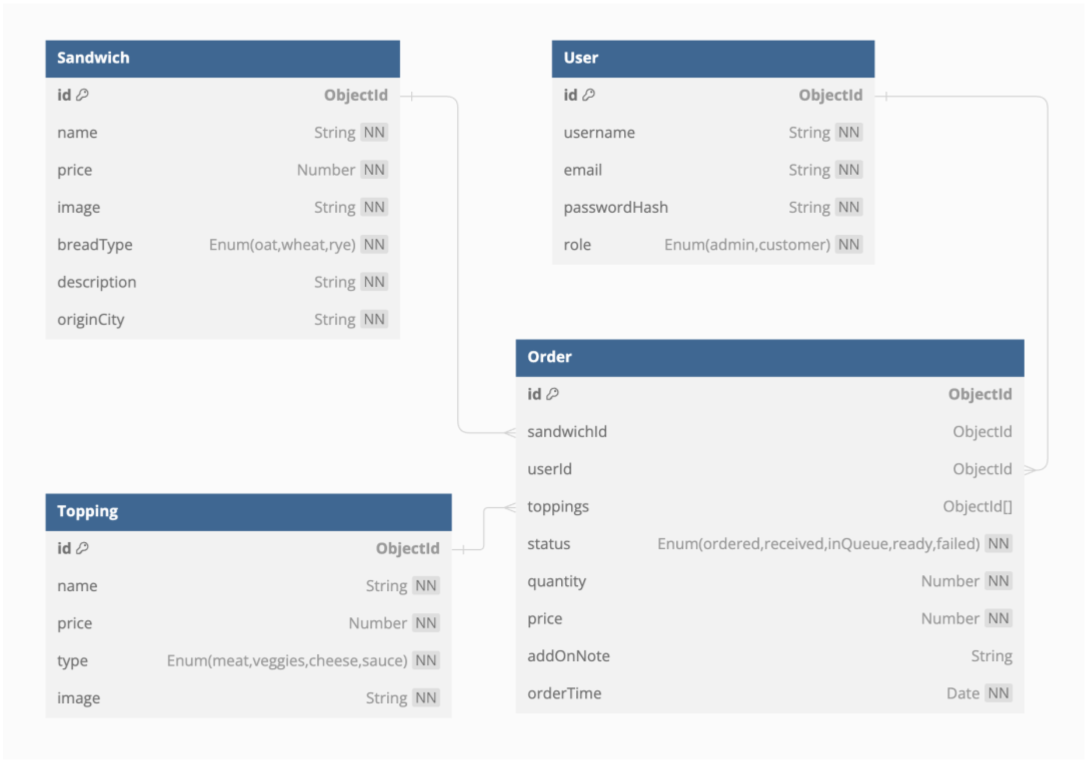
As mentioned before, the order data must be updated in real-time to smoothly streamline its change in status; however, Axios is not capable of detecting changes in the backend to query. One of the solutions is to create a polling request, which means repeatedly spending a GET request to the backend until it detects changes. Polling is a standard technique used by many query services, including GraphQL, however, this approach can lead to unnecessary network traffic, increased latency, and higher server processing overhead, especially when dealing with frequent or small updates. Instead, the application uses **Socket.io**, a JavaScript library that enables real-time, bidirectional communication between web clients and servers. It abstracts away the complexities of WebSockets and provides a simple API that works seamlessly across different platforms and browsers, and supports features like event-based messaging, room management, and automatic reconnection, making it a versatile and powerful tool for building real-time web applications.

In a React project, it is essential to use a state management to blabla. Redux is the most popular state management library in the market today. It relies on a single global store to manage application state, with actions dispatched to update the store and reducers to specify how state changes in response to those actions. Redux provides a strict architecture that promotes separation of concerns and enables powerful features like time-travel debugging and middleware integration. However, Redux can sometimes lead to boilerplate code and complex setup, especially in larger applications. On the other hand, **Recoil** is a newer library developed by Facebook that takes a more decentralized and flexible approach to state management. It introduces concepts like atoms and selectors, allowing developers to define and manipulate state in a more granular and composable manner. Recoil embraces React's component-based architecture and leverages hooks like useRecoilState and useRecoilValue for accessing and updating state within components directly. This approach can lead to cleaner and more concise code, with less ceremony compared to Redux. As the scope of this project is small and lean code is greatly appreciated, Recoil is the choice for state management.

2.3. Data Models and RESTful APIs

The data models and RESTful APIs have undergone significant improvements based on the specifications outlined in the provided Swagger API. These enhancements encompass thorough definitions that have been meticulously crafted to align precisely with the unique needs and objectives of the project.

2.4. Data Models



- Sandwich:
    - id: Unique identifier for each sandwich.
    - name: Name of the sandwich.
    - price: Price of the sandwich.
    - image: Image URL representing the sandwich.
    - breadType: Type of bread used for the sandwich (e.g., oat, rye, wheat).
    - description: Brief description of the sandwich.
    - originCity: Vietnamese city where the sandwich originated from.
- Topping:
    - id: Unique identifier for each topping.
    - name: Name of the topping.

    - price: Price of the topping.
    - type: Type of topping (e.g., meat, veggies, cheese, sauce).
    - image: Image URL representing the topping.
- User:
    - id: Unique identifier for each user.
    - username: Username chosen by the user. email: Email address of the user.
    - passwordHash: Encrypted hash of the user's password.
    - role: Role of the user (either "admin" or "customer").
- Order:
    - id: Unique identifier for each order.
    - sandwichId: Reference to the sandwich ordered.
    - toppings: Array of references to the toppings added to the sandwich in the order.
    - userId: Reference to the user who placed the order.
    - status: Current status of the order (e.g., ordered, received, inQueue, ready, failed).
    - quantity: Quantity of sandwiches in the order. price: Total price of the order.
    - addOnNote: Additional note or customization for the order.
    - orderTime: Date and time when the order was placed.

2.4.1. RESTful APIs

The updated Swagger API can be retrieved here: https://app.swaggerhub.com/apis/caoxantb/make-me-a-sandwich-vingroup-project/1.0.0#/.

- Sandwich:
  - GET `/sandwich`: Fetches a list of all available sandwiches offered by the sandwich shop.
  - POST `/sandwich`: Allows the addition of a new sandwich to the menu, including details like name, price, bread type, toppings, etc.
  - GET `/sandwich/{sandwichId}`: Retrieves detailed information about a specific sandwich by its unique identifier.
  - PUT `/sandwich/{sandwichId}`: Updates the details of a specific sandwich based on its unique identifier.
  - DELETE `/sandwich/{sandwichId}`: Removes a sandwich from the menu based on its unique identifier.
- Topping:
  - GET `/topping`: Retrieves a comprehensive list of all available toppings that can be added to sandwiches, providing details such as name, price, and type (e.g., meat, veggies, cheese, sauce).
- User:
  - GET `/user/current`: Retrieves information about the currently authenticated user, providing details such as username, email, role, etc.
  - POST `/user/register`: Allows the registration of a new user account, requiring essential details such as username, email, and password.

    - POST `/user/login`: Enables users to log in to their accounts using their credentials, initiating a session for authenticated access to protected resources.
    - POST `/user/logout`: Logs out the currently authenticated user, terminating the session and revoking access to protected resources.
    - GET `/user/{username}`: Fetches detailed information about a specific user account based on their unique username.
    - PUT `/user/{username}`: Allows the modification of user account details, such as username, email, and password, based on their unique username.
    - DELETE `/user/{username}`: Permanently deletes a user account and all associated data based on their unique username.
- Order:
    - GET `/order`: Retrieves all orders placed in the system.
    - POST `/order`: Places a new order for a sandwich, specifying details such as sandwich type, quantity, toppings, etc.
    - GET `/order/{orderId}`: Retrieves a specific order by its unique identifier.
    - GET `/order/own`: Retrieves orders that belong to the currently authenticated user, allowing users to view their order history.

3. Learning During the Project

Throughout the project, our team embarked on a journey to implement a comprehensive web application. On the backend, we delved into RESTful API requests, meticulously designing data models and mapping out endpoints to facilitate seamless communication between the server and the frontend. Integrating AMQP communication added another layer of complexity, requiring us to grasp the intricacies of message queuing systems and ensuring efficient data exchange between servers. We have not heard of AMQP until taking the course, and this is a great opportunity for us to understand this technique.

Dockerizing the backend was a significant step, teaching us the importance of containerization for easy deployment and scalability. It provided invaluable insights into managing dependencies and orchestrating the application environment effectively. Meanwhile, on the frontend, we focused on implementing WebSocket for real-time communication, mastering bidirectional channels to enable instant updates and interactions with the user. Using technologies like RecoilJS for state management and custom hooks for API services streamlined our development process, enhancing code organization and reusability. Styling the UI/UX with Ant Design and Emotions brought our application to life, as we learned to create visually appealing interfaces while prioritizing user experience.

However, beyond technical skills, effective collaboration and communication were the pillars of our success. Regular stand-ups and code reviews kept us aligned, ensuring everyone's contributions were valued and integrated seamlessly. As a team, we embraced challenges, celebrated victories, and emerged with newfound knowledge and camaraderie, ready to tackle future endeavors together.