Course Project Group Documentation

Project: Sandwich Order Web Application

Table of Contents

Group Information: 2
- Group Name: Web-architecture 2
- GitLab Repository URL: Web-architecture Repo 2
- Members Information: 2
- Working During the Project 2
  - Timetable 2
  - Responsibilities 3

Project Overview: 3
- Overview of Sandwich Ordering Process: 3
- Project Objectives: 4

Project Plan: 5
- System Architecture: 5
  - Workflow: 5
  - Tools Used: 5
- Sequence Diagram: 6

Project Development: 6
- Backend Development: 6
- Frontend Development: 7

Testing and Deployment: 9
- Testing Procedures: 9
- Deployment Steps: 9

Challenges and Learnings: 10
- Challenges: 10
- Learnings: 11

Additional Features: 11

Conclusion: 12

Appendices: 12

References: 16

Group Information:

Group Name: Web-architecture

GitLab Repository URL: Web-architecture Repo

Members Information:

Name	Student Number	Email
Anisul Mahmud	152152220	anisul.mahmud@tuni.fi
Md. Nasir Uddin Shuvo	152134471	nasir.shuvo@tuni.fi
Israt Jahan Runa	152134329	israt.runa@tuni.fi

Working During the Project

Timetable

Phase	Weeks
Research	Weeks 1-2
Design	Weeks 3-4
Implementation	Weeks 5-8

Responsibilities

Member	Responsibilities	Hours/Week
Anisul Mahmud	Front-end development, Backend development, Server management, Docker, Documentation	18
Md. Nasir Uddin Shuvo	Backend development, Server management, Database management, Docker, Documentation	11
Israt Jahan Runa	Documentation, Testing	8

Project Overview:

The project aims to develop a simple sandwich-ordering application with a focus on distributed components and communication. It comprises front-end, and back-end servers A and B, and a message broker for asynchronous communication. Docker containerization will be utilized for deployment.

Overview of Sandwich Ordering Process:

1. **User Interaction:** Users use the frontend interface to order sandwiches from predetermined menu selections. Users get alerts including their unique order IDs after successfully placing their orders.

2. **Communication:** The frontend communicates sandwich orders via HTTP messages to Server A.

3. **Server A Functionality:** Server A handles incoming frontend messages, manages data storage for orders, and forwards orders to Server B through the message broker.

4. **Database Integration:** Server A integrates with a chosen database to store sandwich orders and pre-made sandwich types, facilitating efficient data management.

5. **Backend Communication:** Server B listens for new orders from the message broker, confirms order readiness, and communicates back to Server A.

6. **User Notification:** Server A updates order statuses, ensuring users are promptly notified of any changes in their order status via the frontend interface.

Backend Components - Server A and Server B:

Server A: Handles frontend messages, accesses data storage, sends orders to Server B, and implements the provided Swagger API.

Server B: Listens for new orders, confirms readiness, and communicates back to Server A.

Message Broker - RabbitMQ: Utilizes RabbitMQ with two message queues: one for communication between Server A and Server B.

Frontend: Developed with React, users may place orders, see all orders, and access individual orders using unique IDs. It uses React Router for smooth navigation, Fetch API for backend connection, and CSS for style, including components such as Home, OrderPage, NavBar, NotFoundPage, and OrderDetails.

Project Objectives:

1. Develop a user-friendly sandwich ordering application with features for placing orders, viewing all orders, and accessing specific orders by ID.
2. Use a message broker (RabbitMQ) to manage asynchronous server communication, together with distributed components such as the frontend, backend servers A and B.
3. Ensure the application architecture is scalable and maintainable by using Docker containerization to provide effective communication amongst remote components during deployment.
4. Integrate MongoDB as the chosen database for Server A to store sandwich orders and pre-made sandwich types for efficient data management.
5. Implement missing functionality for HTTP methods in Server A and Server B to ensure seamless communication and order processing between the frontend and backend.
6. Develop user-friendly frontend UI using React, React Router for routing, Fetch API for backend communication, and CSS for styling, with components including Home, OrderPage, NavBar, NotFoundPage, and OrderDetails.
7. Thoroughly test the application, including integration testing for backend components, end-to-end testing for the entire application flow, and integration testing for React components with external services.
8. Document the project architecture, implementation details, and deployment steps to facilitate knowledge transfer and future maintenance by team members or external stakeholders.

Project Plan:

System Architecture:

The system architecture for the project involves two Node.js servers (Server A and Server B) communicating via RabbitMQ message broker and storing data in MongoDB. The architecture is designed to handle order processing, where Server A receives orders from the frontend/client through APIs, saves them to MongoDB, and sends a message via RabbitMQ to Server B. Server B receives the order message, processes it, updates the order status, and sends the updated status back to Server A. Server A then updates the order status in the MongoDB database and sends a success response to the frontend/client.

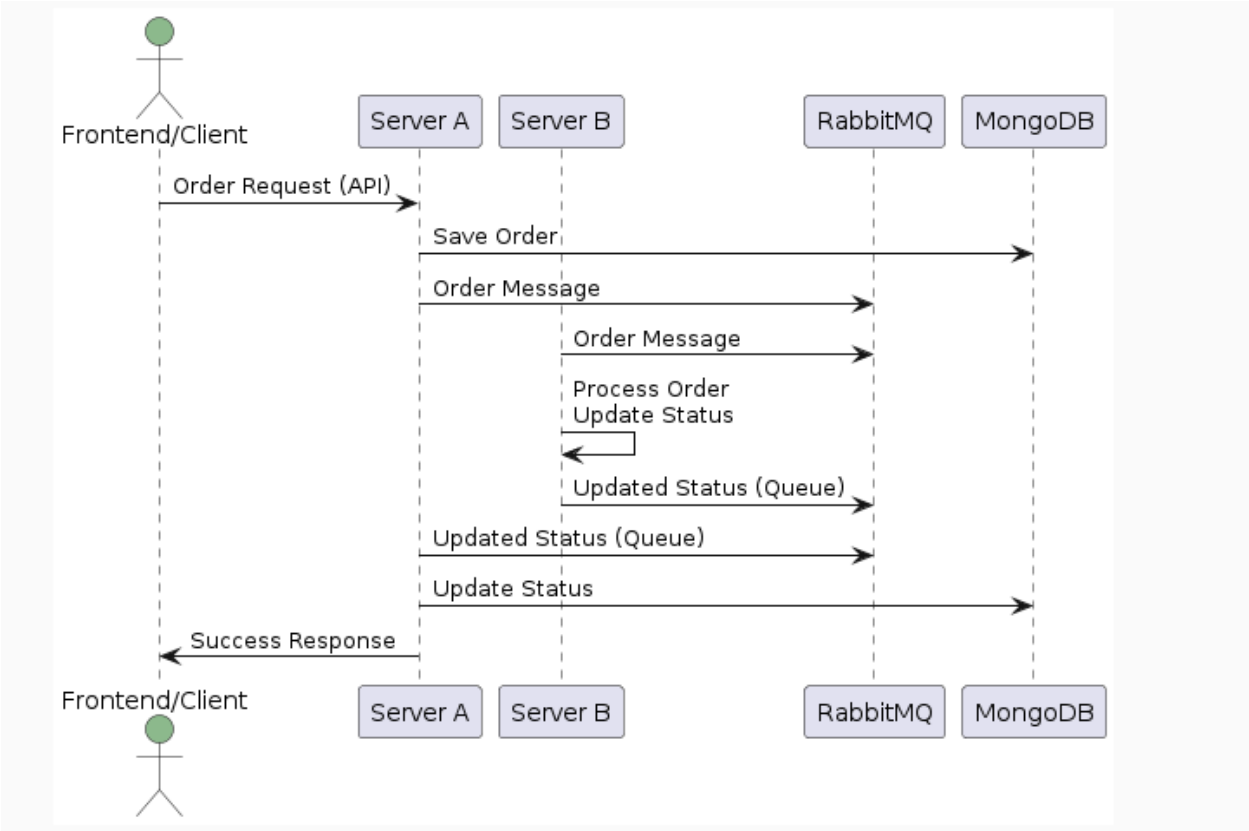
Workflow:

- ➔ Frontend/Client sends an order request to Server A via API.
- ➔ Server A receives the order request and saves it to MongoDB.
- ➔ Server A sends an order message to RabbitMQ for processing by Server B.
- ➔ Server B receives the order message from RabbitMQ.
- ➔ Server B processes the order and updates the order status.
- ➔ Server B sends the updated order status back to Server A via RabbitMQ.
- ➔ Server A receives the updated order status from Server B.
- ➔ Server A updates the order status in MongoDB.
- ➔ Server A sends a successful response to the frontend/client.

Tools Used:

1. Node.js: For building Server A and Server B.
2. RabbitMQ: For message queuing and communication between servers.
3. MongoDB: For storing order data and order status.
4. Docker & Docker Compose

Sequence Diagram:



This architecture ensures a scalable and efficient system for handling order processing in a distributed environment.

Project Development:

Backend Development:

The backend system facilitates order processing between two Node.js servers (Server A and Server B) via RabbitMQ message broker, with data storage managed by MongoDB. Here's a breakdown of each component's role:

Server A:

1. Receives order requests from the frontend.
2. Saves orders to MongoDB.
3. Sends order messages to RabbitMQ.
4. Listens for updated order statuses from Server B.

5. Updates order statuses in MongoDB.
6. Responds to the frontend with success messages

Server B:

1. Listens for order messages from RabbitMQ.
2. Processes orders.
3. Updates order statuses.
4. Sends updated order statuses back to Server A via RabbitMQ.

RabbitMQ:

1. Acts as the message broker facilitating communication between Server A and Server B.
2. Stores and routes order messages between servers.

MongoDB:

1. Stores order data, including order details and statuses.

Technology Stack:

1. **Node.js:** Used to develop Server A and Server B for efficient and scalable backend functionality.

2. **RabbitMQ:** Utilized as the message broker to facilitate communication between servers, ensuring reliable message delivery.

3. **MongoDB:** Chosen as the database solution for storing order data, offering flexibility and scalability for handling large volumes of data.

4. **Docker and Docker Compose:** Employed for containerized deployment of backend components, ensuring consistency across environments and facilitating easy scaling and management.

This architecture ensures a robust and efficient backend system capable of handling order processing tasks seamlessly while leveraging modern technologies to optimize performance and scalability.

Frontend Development:

This project is a sandwich ordering system developed using React. It allows users to place sandwich orders, view all orders, and view specific orders by their ID.

Features

Place Sandwich Order: Users can place an order for a sandwich. The order details are sent to the backend server for processing.

View All Orders: Users can view a list of all sandwich orders. Each order includes the order ID, the type of sandwich ordered, the sandwich ID, the status of the order, and the price.

View Specific Order: Users can view the details of a specific order by its ID. This feature allows users to track the status of their orders.

Technology Used:

React: This project is built using React, a popular JavaScript library for building user interfaces. React allows developers to create reusable UI components, manage component states, and efficiently update and render components in response to data changes.

React Router: This library is used for handling routing in the application. It allows us to render different components based on the current URL path.

Fetch API: The Fetch API is used for making HTTP requests to the backend server. It's a modern, promise-based API for making asynchronous HTTP requests in JavaScript.

CSS: CSS is used for styling the application.

Components:

a. **Home Component:** This component serves as the landing page where users can place new sandwich orders. It likely contains a form or interface for users to select their sandwich preferences and submit their orders.

b. **OrderPage Component:** This component displays a list of all sandwich orders. Each order is presented with details such as its ID, sandwich-type, status, and price. Additionally, there should be a 'View' button enabling users to inspect the details of specific orders.

c. **NavBar Component:** The navigation bar of your application, this component contains links to essential pages such as the Home page and the OrderPage. It ensures seamless navigation between different sections of your application.

d. **NotFoundPage Component:** This component is displayed when users attempt to access a route that doesn't exist within your application. It typically provides a message informing users of the non-existent page and might include a link to return to a valid page.

e. **OrderDetails Component:** Responsible for presenting the detailed information of a specific order, this is most commonly displayed when users click the 'View' button on an order in the OrderPage. It retrieves order information from the backend based on the order ID and shows it to the user.

Routing:

1. **Home Route ("/"):** The home route is mapped to the Home component. This is where users can place a new sandwich order.

2. **Order Route ("/order"):** This route is mapped to the OrderPage component. Here, users can view a list of all sandwich orders.

3. **Order Details Route ("/order/:orderId"):** This route is mapped to the OrderDetails component. It displays the details of a specific order. The:orderId in the path is a URL parameter, and its value is the ID of the order to display.

4. **Not Found Route ("*")**: This route is mapped to the NotFoundPage component. It's displayed when a user navigates to a route that doesn't match any of the defined routes in the application.

Testing and Deployment:

Testing Procedures:

After developing our backend and frontend, we tested them. Our testing techniques are outlined below.

Backend Testing:

1. **Integration Testing:** Integration tests were performed to check that different sections of the backend application functioned properly. Examine API endpoints, database interactions, and message broker integration.

2. **End-to-end Testing:** End-to-end tests were performed to check the application's whole flow, from request to response. Use Postman to replicate real-world settings.

Frontend Testing:

1. Integration Testing: Tested the integration of React components with external services, such as API calls. Mock API responses and simulate user interactions to verify component behavior.

Deployment Steps:

To deploy it, one has to follow these steps after cloning the project.

Deployment steps are:

1. **Installation:** Install Docker Desktop.

2. **Build the Docker images:** Navigate to the root directory of your project in the terminal. Run the command **docker-compose up --build**. This will initiate the building process for Docker images of your backend servers and MongoDB database.

3. **Start the Docker Containers:** Docker Compose will automatically launch the Docker containers when the image creation is completed successfully. Your backend servers and MongoDB database are now operational within Docker containers. You can see this both in the terminal and on the Docker desktop. **Additionally**, make sure to check the status of all containers on a regular basis. Containers may stop working due to firewall settings or random internet access concerns. In such circumstances, just restart the stopped containers from **Docker Desktop** to restore regular operations.

4. **Start the Frontend:** After the backend, switch to the frontend directory in your terminal. Execute the command **npm install** to install the required node modules for the front end. Then, launch the frontend server by running **npm run dev**.

5. **Access the application:** Open a web browser and navigate to localhost:3000 (or the port specified for your frontend server, you can see that on the terminal). You should now have access to your sandwich ordering application's user interface.

6. **Monitor the Application:** Keep an eye on your terminal and Docker Desktop for any error messages or issues during the deployment process.

By following these deployment steps, one may successfully deploy our sandwich ordering application with Docker for the backend and npm for the front end, granting users access to the program's features.

Challenges and Learnings:

Challenges:

1. **Integrating Distributed Components:** We had difficulty establishing a good connection between the frontend, backend servers A and B, and the message broker due to the asynchronous nature of communication and the necessity for trustworthy message forwarding. Overcoming these issues necessitated joint efforts and a thorough grasp of distributed system ideas.

2. **Scalability and Performance:** Designing the application architecture to provide scalability and optimal performance, particularly under heavy loads and concurrent user interactions, presented considerable hurdles to our team. To satisfy performance requirements, we needed to carefully allocate resources, apply load-balancing algorithms, and optimize code.

3. **Database Integration:** We encountered challenges while connecting MongoDB with Server A for appropriate data management. This required a thorough grasp of MongoDB's capabilities and best practices for handling sandwich orders and pre-made sandwich variants. We were able to solve these problems by collaborating and experimenting to create a strong database integration solution.

4. **Frontend-Backend Communication:** Implementing and testing communication between frontend and backend servers proved difficult in terms of stability and responsiveness. We noticed issues processing HTTP requests/responses and updating order statuses in real-time. We tackled these problems by working together and using our total knowledge, resulting in the effective implementation of a dependable communication system between frontend and backend components.

Learnings:

1. **Distributed System Design:** Throughout our project, we gained a better knowledge of distributed system fundamentals including message passing, event-driven design, and fault tolerance. Using these insights, we successfully built and deployed distributed components in our program, guaranteeing strong communication and fault tolerance.

2. **Scalability Strategies:** Reflecting on our project experience, we were able to effectively implement a variety of scaling options, such as microservices, Docker containerization, and load balancing. These tactics allowed our program to manage greater user loads easily while maintaining excellent performance, showcasing the efficacy of our teamwork.

3. **Frontend-Backend Integration:** Looking back on our project development, we learned the integration of frontend and backend components by creating RESTful APIs, implementing asynchronous communication using the Fetch API, and managing asynchronous answers and errors in React. Our collaborative approach made seamless integration possible, resulting in a coherent user experience.

Additional Features:

1. **Automated Order Placement:**
- ❖ Implement a button for the user to place an order.
- ❖ When the button is clicked, send a request to the backend to place the order.
- ❖ The backend handles the rest, including processing the order and updating its status.

2. **Order Status Stages:**
- ❖ Define three stages for the order status: "received," "in progress," and "completed.".
- ❖ Ensure that the backend communicates with the front end to update the order status correctly as it progresses through these stages.

3. **Automatic Refresh for Order Status:**

- ❖ Implement automatic refreshing of the order status without requiring user intervention.
- ❖ Whenever the status of an order changes on the backend, the frontend automatically updates to reflect the new status.

Conclusion:

Our experience building the sandwich-ordering application has been both tough and gratifying. We effectively met our project objectives and overcame many challenges via collective efforts and continual learning. From architectural design to distributed component implementation, our team overcame the intricacies of combining frontend and backend systems to ensure smooth communication and dependable performance. Scalability and database integration challenges were overcome via patience and a thorough understanding of essential technologies, resulting in efficient solutions and optimized performance. After reflecting on our experiences, we obtained significant insights on distributed system architecture, scalability techniques, and frontend-backend integration. These insights, together with our hands-on experience, have given us the abilities and expertise to take on comparable initiatives in the future.

Overall, our project emphasizes the value of teamwork, adaptation, and continual learning in overcoming obstacles and achieving effective results. As we close this project, we are proud of our successes and eager to apply our acquired knowledge to future endeavors.

Appendices:

A. Database schema

The database schema for the MongoDB orders collection is defined with Mongoose. It contains the following fields:

sandwichId: Represents the unique identifier of the sandwich.
id: Represents the unique identifier of the order.
status: Represents the current status of the order, which can be one of the following values: 'ordered', 'in progress', 'ready', or 'failed'.

The schema assures that the sandwichId and id fields are needed and of type Number, whilst the status field is of type String with an enum constraint that limits its potential values. The status field's default value is 'ordered'. This schema definition defines the structure of the order collection and allows for efficient data handling within the application.

B. API Documentation

Our application provides a RESTful API that is hosted at `localhost:8080` with the base path of `/v1`. The API follows the HTTP protocol and uses JSON for data interchange.

Order Endpoints

GET /order
Provides a list of all orders. Returns an empty array if no orders are found.

- Method: `GET`
- URL: `/order`
- Response: `200` (successful operation), returns an array of orders

POST /order
Add an order for a sandwich.

- Method: `POST`
- URL: `/order`
- Body Parameters: `order` (required)
- Response:
  - `200` (successful operation), returns the added order
  - `400` (Order not created)

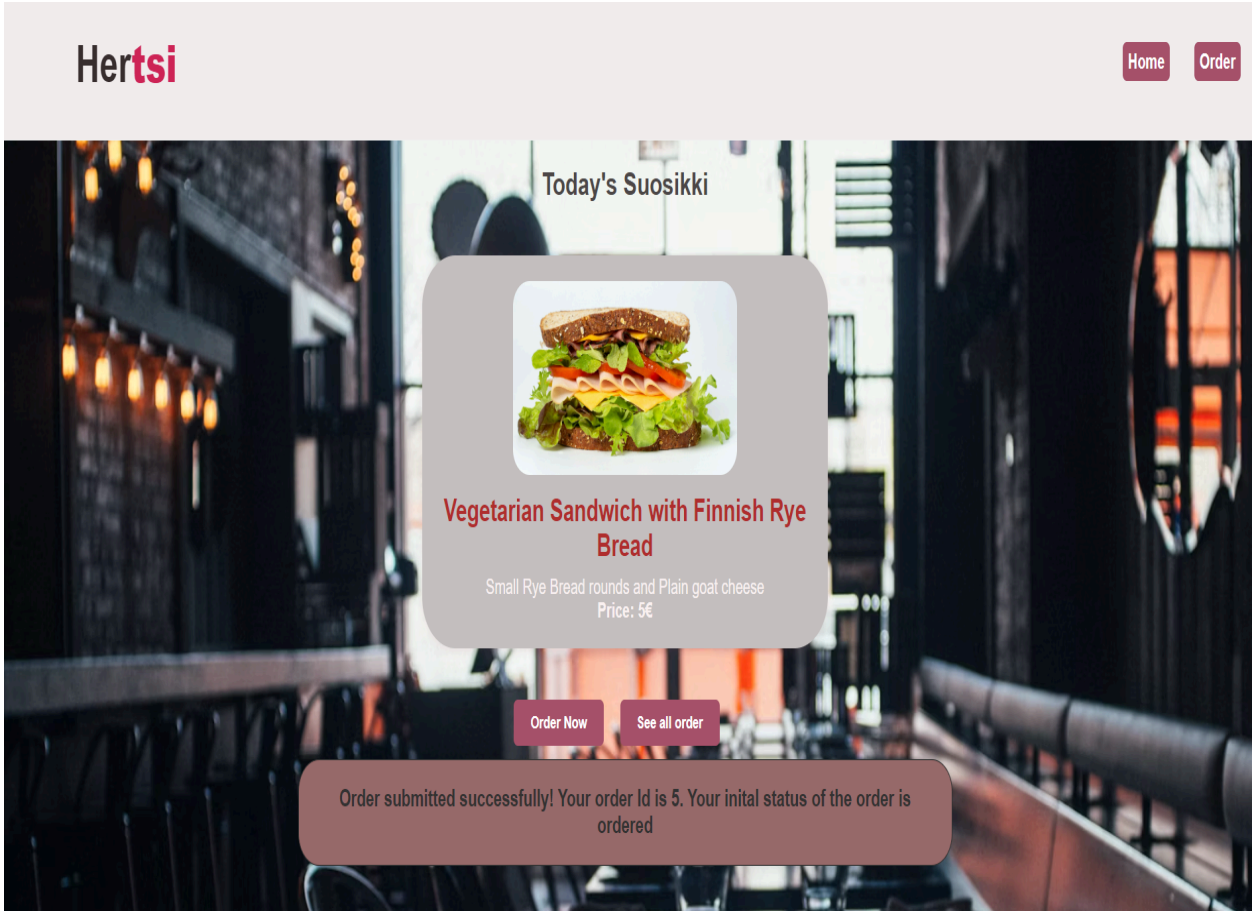
GET /order/{orderId}
Find an order by its ID. IDs must be positive integers.

- Method: `GET`
- URL: `/order/{orderId}`
- Path Parameters: `orderId` (required)
- Response:
  - `200` (successful operation), returns the order
  - `400` (Invalid ID supplied)
  - `404` (Order not found)

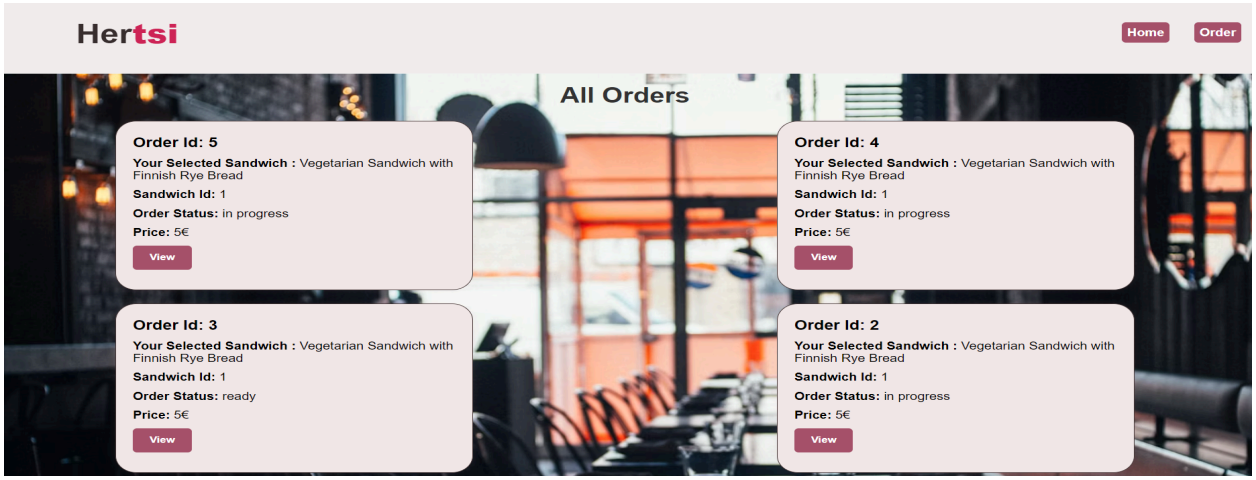
C. Application previews

This section includes images of the application's core features and interfaces. These screenshots provide a visual representation of the application's functionality and layout.

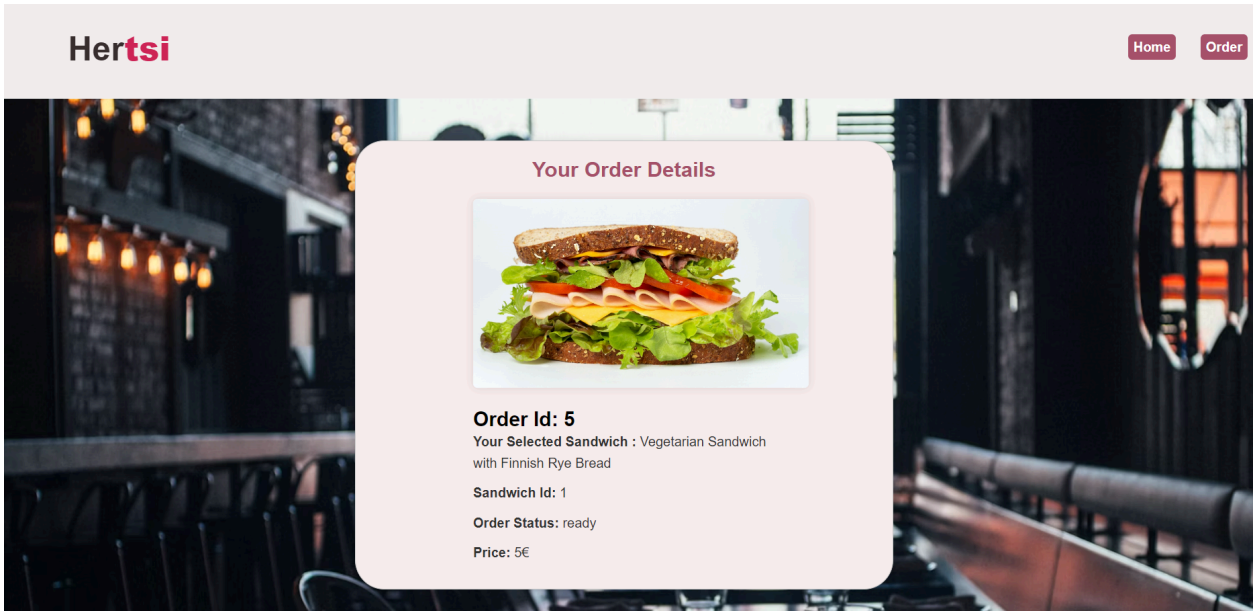
1. **Home Page:**



2. Order List Page:



3. Order Details Page:



References:

1. https://docs.docker.com/
2. https://www.rabbitmq.com/docs
3. https://www.svix.com/resources/guides/rabbitmq-docker-setup-guide/
4. https://www.mongodb.com/docs/
5. https://www.mongodb.com/resources/products/compatibilities/docker
6. https://swagger.io/docs/
7. https://legacy.reactjs.org/docs/getting-started.html