Documentation

Name	Email	Student Number
Asbah Amjad Usmani	asbahamjad.usmani@tuni.fi	H292962
Chi-Hao Lay	chi-hao.lay@tuni.fi	K424562

The work was splitted so that Asbah did *Server A* and Chi-Hao *Server B* and *frontend*.

Project Overview

This is a simple Sandwich Order Web Application. User can order sandwich with the help of frontend. Order is placed and processed in the backend. This project is divided into three main parts for modularity and performance balancing i.e. frontend, server A and server B in the backend. The sandwich order application follows loosely coupled architecture. For asynchronous communication, mediator approach is used. Hence, the communication part is taken care of by RabbitMQ.

All the parts of the software can be started with

```
docker-compose up --build
```

The ports exposed are as follows:

Application	Port
Server A	8080
Server B	8000
Frontend	3000

The ports are quite popular for development environments. In production, they should run behind a reverse proxy using HTTPS port 443.

Queues used are

- *orderGenerationQueue*, new orders are placed to this queue.
- *orderCompletionQueue*, completed "ready" orders are put to this one.

Testing

Automated tests are missing because since we are out of time. For manual testing:

1. `docker-compose up --build`
2. Wait for containers to start

3. Open `http://localhost:3000` in browser
4. Click a button "ADMINS"
5. Click "Add"
6. Fill every field.
7. Click "Sandwiches" in menu
8. Click "Order"
9. Click "Confirm"
10. Wait for 10 seconds

Front-end

The final build doesn't auto-refresh so in development mode it can be run as

```
cd frontend
yarn install
yarn start
```

Application Structure

The frontend is React *single-page application* it was created with *Create React App*. It only contains three views

- sandwich listing
- payment confirm
- order listing.

Most of the third-party dependencies have been avoided and experimented with different way to implement same things. The notable summary of technologies are:

- React
- Vanilla JavaScript
- CSS Modules for scoping the css
- Axios for API communication
- ~~Jest for testing~~ it was supposed to be tested, but got to hurry

The structure of the application is

```
// contains all components to be used
components/
    Component/
        Component.js
        Component.module.css
        Component.test.js

// event emitters
events/
    ContenSwitch.js
    NotificationEvent.js
```

```
// data to use in testing and development without API
mocks/

// all communication to backend is here
services/
    Service.js
    Service.test.js

// all views "pages"
views/
    OrderListView.js
    OrderListView.module.css

// View name mappings
routes.js

// Settings of the application and other constants
settings.js
```

Event Based Architeture

This is an experiment for switching views without Redux and React Contexts, and it uses events to request view changes. The app shell will load the listener for it.

After the listener is up the content is swapped by calling

```
fireContentSwitchEvent(viewName);
```

The similar architecture is used with showing notifications.

```
fireNotificationEvent(text, level);
```

CSS Architecture

The styles follow a style of defining css constants as

```
:root {
   --constant-name: value;
}
```

which forms the global base values to be used anywhere as `var(--constant-name)`. The idea of doing it like this is to allow overriding them and keeping them in one place while scoping the styles.

Services

The communication to backend is split to Services that only return the data or a rejects the Promise.

The structure is inspired by Angular, and was seen in Full Stack Open -course.

Polling for orders

As we are not using websockets the server cannot tell us which order is ready, so we use polling to refresh the listing.

1. Order listing received.
2. Set timeout for polling.
3. Poll the server.
4. Go to step 1.

The `setTimeout` is used because `setInterval` can cause buggy calls if view is changed and it is not cleared properly. Low interval is used since, server B has low response time.

Routing

The *routing* is done by putting component and router name mapping to `routers.js`. The `App.js` will then swap the content by route name if an `content-swich` event is dispatched.

State Persistence

Because of SPA application, it doesn't have dedicated urls to visit so whenever view changes, it saves the route name to *session storage*. If page is reloaded, the application will read the view name and open the view.

Backend

Server A

Structure of Server A

In Server A, different folders are created to refactor the code. Implementation of MongoDb connection and schema definitions are located in *models* directory, *rabbit-utils* is for sending and receiving order in the queues of RabbitMq, *routes* is used for the routing of API requests, and required services are defined in the *services* folder. Server A runs on node.js and uses libraries such as express, mongoose, cors and amqplib and these dependecies are defined in the *package.json* file.

Implementation

- Implementation of Order API
- Implementation of Sandwich API

Here are the functionalities that Server A provides:

1. Add a sandwich using the endpoint http://localhost:8080/sandwich/ with request POST. The response will be sandwich object id in JSON format.
2. Get a sandwich using the endpoint http://localhost:8080/sandwich/sandwichid with request GET. The response will be the sandwich object in JSON format.
3. Get a sandwich using the endpoint http://localhost:8080/sandwich with request GET. The response will be an array of Sandwich objects in JSON format.
4. Modify a sandwich with a POST request for the endpoint http://localhost:8080/sandwich/sandwichid. It will return the sandwich object id in JSON format.
5. Delete a sandwich using the request DELETE with the endpoint http://localhost:8080/sandwich/sandwichid. It will return the deleted Sandwichid.
6. Server A also handles the orders. We can add orders using the same endpoint URL with POST method http://localhost:8080/order/. It will return the newly created object type.
7. This endpoint with the method request GET http://localhost:8080/order/:orderId, will return the order details specific to an order id in JSON format.
8. The same endpoint URL http://localhost:8080/order with GET method, will return the list of order objects in an array format.
9. It also uses the MongoDB database to store new sandwiches, as well as retrieve details of the sandwiches from the database to show it at UI.
10. It uses two Rabbitmq channels: order generation and order completion to communicate with Server B. Order generation takes care of the order in the process from Server A to Server B. Order completion takes care of the order that is completed and sent from Server B to Server A.

More information about Server A is present in README.md file present in the directory of server A.

RabbitMQ

RabbitMQ is a message brokering service and it is used to provide asynchronous communication between serve-a and server-b. These tasks are handled in the *rabbit-utils* directory.In this project, rabbitmq image `rabbitmq:3-management-alpine` is used and all the required configurations are present in the root directory of `docker-compose.yml` file.

MongoDb

In this project, MongoDb image is used in docker-compose.yml file. MongoDb runs in docker on port: 27017. Server A saves the sandwich as well as the orders in mongodb for persistence purposes.

Server B

Server B runs on *node.js* and uses libraries *express* and *amqplib* and these dependecies are defined in the *package.json* file. Server B temporarily keeps the detail of orders in memory until 10 seconds have elapsed. It tracks the order generation as well as completion. Whenever Server A generates some orders, Server B consumes that for processing. When the consumption, as well as processing, are done, it sends a signal using rabbitmq message channels about the completion and order is shown completed.

Structure

Server B is dockerized just like other parts of application. In server B, *rabbit-utils* is for sending and receiving order in the queues of RabbitMq and preparation of order is performed in the *app* directory.

The server B doesn't follow any patterns particularly, it has only extended the given files by adding:

- separating action to `app/prepareOrder.js` allows extensibility if `doWork` -function in `rabbit-utils` is changed.
- application (ports, hosts etc…) configuration is put to `settings.js`. The `enums.js` is a file to keep constants like in frontend, ideally with a file watcher that would synchronize them.

Learning

Chi-Hao Lay

I was originally supposed to only do the frontend, although I was already very familiar with it, so I didn't learn much. I tried out different ways to stuff that I haven't before such as using custom events to communicate between components.

I have learned a little more about queues when trying to do Server B, they were a kind of magic to me before. It was surprisingly simple since, the structure was mostly already in the repo.

I learned about multi-stage builds in Docker and that you don't have to deploy microservices in separate servers but keep them in docker-compose and deploy to same server.

Conclusion

This project helps us to understand web application architecture, how to dockerize your application, and message brokering services like Rabbitmq. This assignment helps us to co-operate with each other to come out of the existing loop-holes and continue the project without any hurdles.