Tampere University **Unit of Computing Sciences**
COMP.SE.510 Web Development 2 - Architecting (Hervanta campus)

Group project – Web development 2

Group 1

Design and architecture

Thomas de Lachaux, 050708284, thomas.daignanfornierdelachaux@tuni.fi
Xavier Finkelstein, 050710515, xavier.finkelstein@tuni.fi
Aleksi Hirvonen, H253180, aleksi.hirvonen@tuni.fi

https://course-gitlab.tuni.fi/webarch-2021/group-1

Contents

1 Project plan

In this chapter, project plan for the project is explained. Project plan includes timetable for stages of the project which include research, design, and implementation for the architecture. This chapter also includes initial division of responsibility between group members for implementation.

In table 1, initial timetable for this project is presented. Table includes week division between different stages of the project.

Table 1. Initial timetable for project

Week No.	Task
11	Design & Initial technologies
12	Research
13-16	Implementation

Project repository was initialized in week 12 but project will be started with initial designing in week 11. Initial design includes frontend sketching, architectural design and initial technologies to use in the project using Figma. Week 12 will be used to start implementation and to learn new used technologies as new technologies such as RabbitMQ is required to implement message queues. Implementation will be done between week 13–16.

As there is three separate systems to implement (frontend, server-a, server-b), responsibilities are shared in system level. Thomas will handle frontend, Xavier will handle server-a and Aleksi server-b and RabbitMQ. Each of us spent about 30 hours on the project counting architecture, development, review and report.

For the Git organisation, we decided to work separatly on different branches, on branch per feature. The branch format is type-scope/tsubject. For example, fix-front/admin-panel. After the feature was finished, the developer submits a merge-request and at least one other developer must approuve before merging.

2 System architecture

In this chapter, architecture of the system is described. Architecture is described with component-diagram which consists of whole system and sequence diagram which visualizes new order from ordered-state to ready-state. Current architecture is also compared to other possible architectures that could have been used instead.

2.1 Architecture

Component diagram for whole system is presented in figure 1. System consists of frontend, server-a and server-b which all run as separate instances in docker containers. As external services, MongoDB is used to persist data and RabbitMQ for internal messaging between servers.

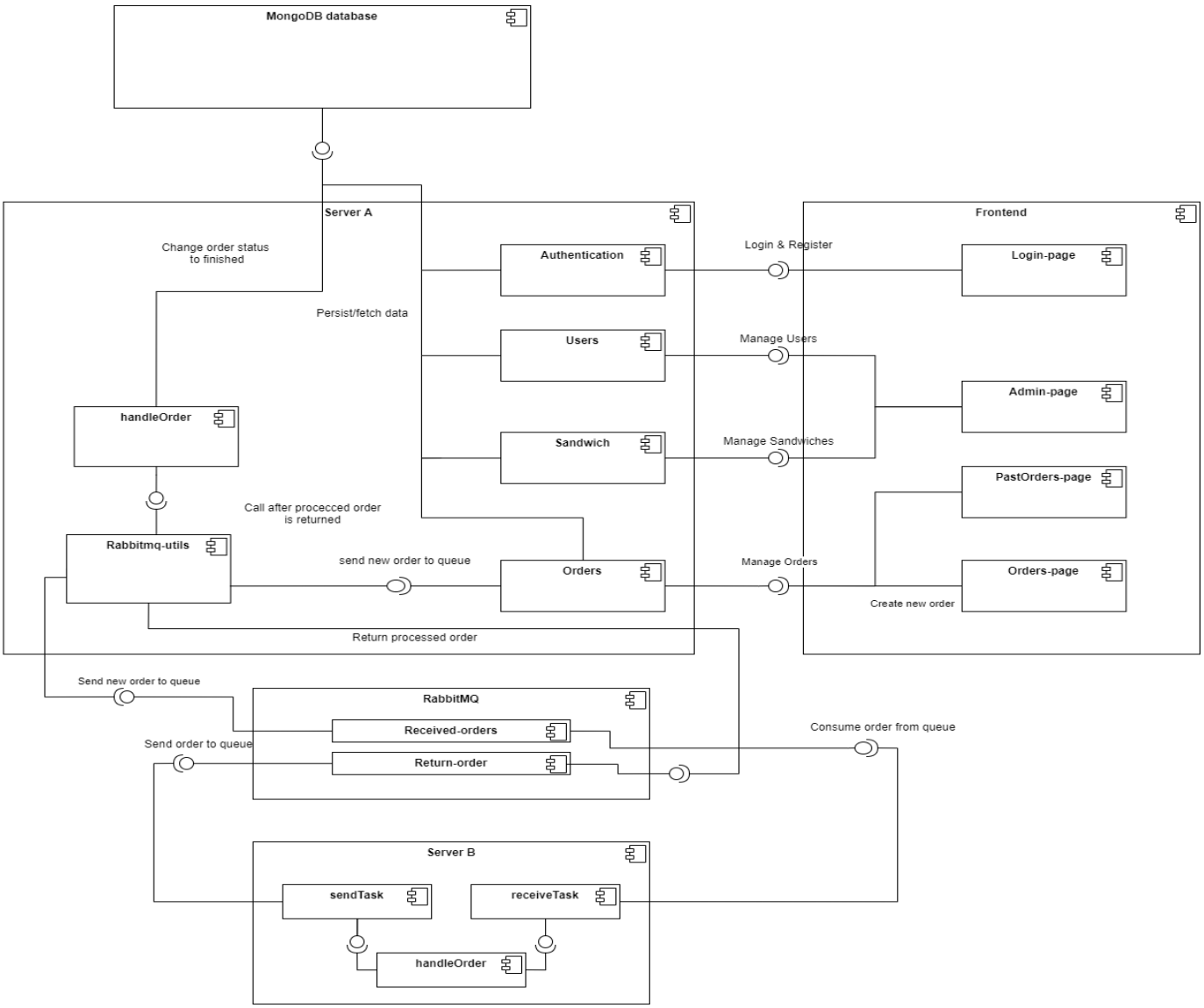

Figure 1: Component diagram for the system

Frontend is implemented with React and consists of 4 pages. Server A is implemented with NodeJs and Express and it consists of 4 endpoint-groups. All communication between frontend and server-a is handled with REST-protocol with JWT-token as credentials which makes authentication stateless. After user logs in, JWT-token is returned to client which is saved to *localStorage.* JWT-token is then used in *Authorization*-header to fetch data that require authentication. Login-page is used to log in to the application and register new users. After logging in, user is redirected to *Orders*-page.

Orders-page is used to create new orders. View includes all sandwiches, which can be added to cart and order-button which posts new order to *Orders*-endpoint. New order is sent to server A which saves order to database, returns response to post-request, and sends order to RabbitMQ *received-orders* queue which is then consumed by server B. Server B processes order with handleOrder-function and send order with *ready*-status to *return-order* queue which is then consumed by server A. Server A updates order with returned status-value.

Admin-page is used to manage sandwiches and users. Only users with admin-role can view this page and only admins can access server A endpoints which modify all users and sandwiches. Admins can delete users or upgrade/downgrade role of the user. Sandwiches can be created, modified, and deleted in admin-view.

PastOrders-page is used to see all orders of logged in user. Order includes ordered sandwiches, timestamp, and state of the order. Each user gets their own orders to PastOrders-page.

Figure 2 visualises sequence of actions when user makes new order from frontend. As user makes new order from frontend, it’s status is changed to orderer in frontend and it is sent with REST to Server A. Server A persist order to database with status *received* and tries to send data to RabbitMQs message queue named *received-orders*. Figure includes alternative flows depending on success of sending data to queue where first flow is succesful operation and second is failed operation.

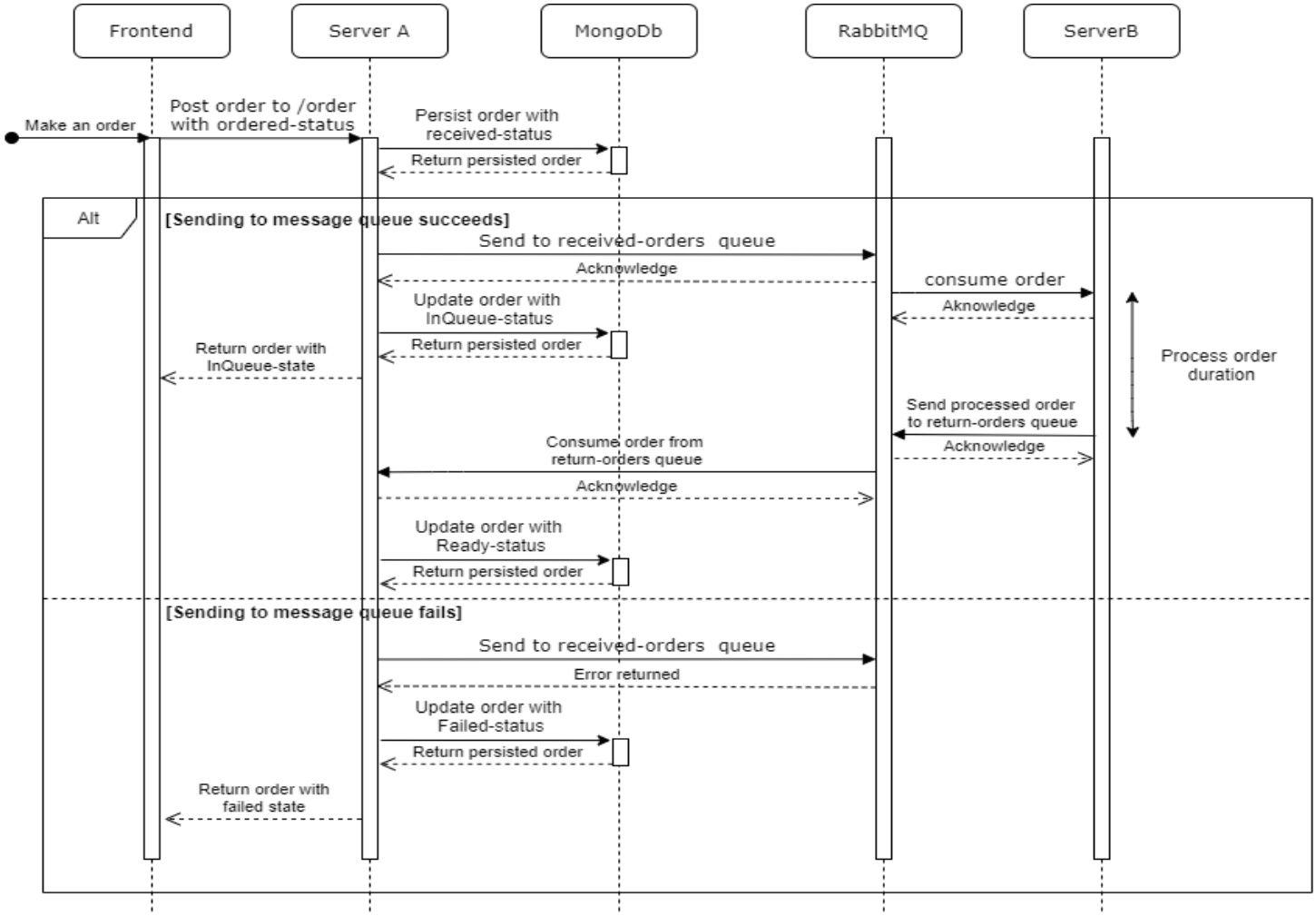

Figure 2. Sequence diagram for new order with alternative flows

In succesful operation, order is sent to queue *received-orders*. After acknowledgement of successful operation is sent by RabbitMQ, Server A updates orders state to InQueue and returns created order as response to initial post-request. In meanwhile, server B consumes order from *received-orders queue,* processes order, sets it's state to *ready* and sends it to *return-orders* queue where it is consumed by Server A. Server A updates persisted order with returned status-value.

In failed operation, order is sent to queue *received-orders* but transfer is erronous and error is returned. Order-status is then updated to database with *failed*-status and order as sent as response to initial post-request.

2.2 Evaluation and justification

The current architecture is composed of a front server, two back servers that each have a different function, a service broker and a database. The separation helps to approach the micro-service paradigm. For our scale, this separation is a bit overkill, and the two

servers could have been merged and the service broker removed. However, this architecture helps to have a maintainable and especially a scalable application.

However, a missing part and critical part when the application is scaled in the architecture is the CI/CD (Continuous Integration/Continuous Delivery) pipeline that could have been implemented with GitLab CI. Indeed, the CI would have been used to lint the code before any merge request. If the CI fails, the merge request would be unmergeable. Another utility of the CI/CD is when the MR is merged, it deploys automatically to the server. The pipeline could SSH to the machine, pull the repository, and build the new images. If we wanted to have zero downtimes between the deployment, we could use a container orchestrator such as Docker Swarm or Kubernetes.

3 Used technologies.

In this chapter, used technologies for the project will be explained. Project consists of frontend, server-a, mongodb, server-b and RabbitMQ which will all run in separate Docker containers.

3.1 Front-end

The frontend is implemented in Node.js with React.js. We decided to choose this framework over others as it is the most popular and most loved framework according to StateOfJS. The project has been coded using React Hooks, the modern and recommended way of React.js, instead of using classes. The bundler used to develop and build the app is Vite. It can be unexpected because the common way is to use Create React App which uses Webpack in the backend. However, during development, Vite does not transpile ES Modules into CommonJS as it is supported by modern browsers, so the refresh time takes less than 0.1 second. Nevertheless, Vite transpiles ES Modules during build into CommonJS to support older browsers using *RollupJS*.

Another choice to make is the CSS stack. We decided to use CSS only without any library to have the full design choice. However, we decided to not use standard CSS but to use CSS-in-JS using the styled-components library. Using CSS-in-JS allow us to make scoped components and not to have a global stylesheet and therefore avoid side-effects. Moreover, it allows us to use a Sass syntax. Finally, as the styled-components library create JSX components, the code is easier to read in the rendering part. An alternative to vanilla CSS would have been to use Tailwind.css, an utility-class based framework.

Then, a choice had to be made to create global states. Global states are used in the code to manage the user. We had the choice between stock React with the *useContext* hook or with Redux. We finally chose Redux, as it is more scalable, and modular.

A technology that may be useful is to use Server-Side-Rendering using Next.js, a framework over React.js that implements SSR, but also routing, and other features. In SSR, the code can be calculated server-side, so it can improve the performances in rendering on the client-side, as well as improve SEO, even if Google now handles crawling in client-side rendering websites. We haven't implemented it, but it could be useful if the app needs to scale.

The last technology that could have been useful to implement in the front-end, along with server A is Socket.io. Having a permanent socket connection between the backend and the front-end would have help to have a dynamic refresh of the content, with no action needed for the user.

3.2 Server A

Server A, the project API, has been implemented using the *Express.js* library, as it is the most popular and easy-to-use library to create JavaScript APIs. It is indeed much simpler to implement compared to the basic http module, as it provides core features for an API such as routes architecture, middleware capabilities, simple plugins adding or easy HTTP requests and responses manipulation. Thanks to this very powerful yet lightweight library, we were able implement the API very quickly while always respecting REST practices.

The API manipulates information about orders, sandwiches, and users. All this data is stored in a MongoDB that runs in a separate container, which is a document-based database system. We decided to use this database system as it is lightweight, efficient, and easily scalable, and these are features we are looking for when implementing a microservices-based architecture. It also fits our JavaScript API as it natively supports JSON objects, which is very useful and simpler for us as it allows nested objects and easy compatibility with JS. For communications from the API to MongoDB, we decided to use the *mongoose* library, as it allows us to easily create models, connections, and requests for the database while the underlying functioning is handled by the library itself. It made the process of building communications with *mongodb* very much easier.

3.3 Server B

Server-b is implemented with Node.js with amqplib as dependency. Amqplib is used to connect with RabbitMQ-instance. As server-b is used only as stub to handle orders, it only receives messages from rabbitmq and sends messages back to message queue after certain processing time.

RabbitMQ is easy to use as messaging service. There are multiple alternative message queues in the market including Apache Kafka, which could be used. However, for this application RabbitMQ is the good choice as it supports standardized protocols such as AMQP which could be switched easily to another queue implementation if needed. It was also easy to use and easy to integrate to the system. If large amounts of data should be distributed through queue, Kafka would be better option as it is designed for that.

Node.js is opensource back-end Javascript environment which is used to run JavaScript outside web browsers. As mature environment, it has multiple frameworks which can be used to build REST APIs, amqplib connections or any standardized methods of communicating with other services.

3.4 Technologies used in all projects.

There are some technologies that are common between all projects. The most important is ESLint, a highly configurable and modular JavaScript linter. We plugged it on all projects with Airbnb ESLint configuration (one of the most well-known) and Prettier to enforce a formatting.

An editorconfig file is placed on the root to enforce the editor to use adapted spacing and line ending parameters.

When we used Git, we forced us to use the commitlint convention, but it would have been better to enforce it using Husky and commitlint. However, this is possible to implement if the projects are separated in 3 different repositories.

The *wait-for-it.sh* script has not been implemented. The server A and the server B try repetitively to connect to the RabbitMQ and MongoDB services using Javascript promises.

4 How to test

To launch the project, only a few commands are needed:

```
$ git clone git@course-gitlab.tuni.fi:webarch-2021/group-1.git
$ cd group-1
$ docker-compose up -d
```

Be sure that you have the correct Docker (⩾ 1.13.1) and Docker Compose (⩾ 1.18.0) versions to run this project, and that ports 8080 and 3000 are accessible on your machine.

The only field in *docker-compose.yml* that must be modified is:
services.frontend.build.args.VITE_API_URL - change the value to the according domain name or IP before building, always on port 3000.

The project is then launchable as it is, but it is recommended to edit the secret values like the JWT secret, MongoDB and RabbitMQ credentials in *docker-compose.yml* to enforce security.

Then, you can connect on the *SERVER_ADDRESS:8080*. The first account you will create is an administrator. The next ones will be normal users.

5 Learning during the project

In this chapter, learning during this project is visited. Learning is presented with table 2, where all new learning is documented by group members.

Table 2. Learning diary

W*ho?*	What was learnt?	When?	GitLab-issue/merge
Aleksi	RabbitMQ basics which included Get started tutorials 1-5	30.3-1.4	Issue #4
Aleksi	RabbitMQ in use where given scripts were re-written and refactored	4.4	MR #9
Aleksi	Write UML-diagrams and learn more about component- and sequence-diagrams. For example, alternative execution was learnt.	22.4-24.4	
Thomas	Create a partially responsive website with CSS-in-JS	20.3-28.3	MR #1
Thomas	Create a dynamic administration panel	28.3-2.4	MR #13
Thomas	Review code from other participants		Almost all MR
Xavier	Create an authentication system with JWT	30.3 – 31.3	MR #5, #7
Xavier	Write a docker-compose file from scratch	2.4	MR #16
Xavier	REST API good practices with Express.js	28.3 – 31.3	MR #3, #5, #7

Table 2 is simple learning diary where each group member has filled new entry to table whenever new things were learnt. Table includes brief description of what was learnt, when it happened and possible GitLab issue or merge request which is connected to this learning.