Documentation of COMP.CS.510 Web Development 2 – Architecting Group Project

Project sandwich

https://course-gitlab.tuni.fi/COMP-CS-510-web-architectures_2021-2022/project-sandwich

Project Plan

Group

Members:

- Leevi Saarinen, 274232, leevi.saarinen@tuni.fi
- Joni Riikola, 283309, joni.riikola@tuni.fi
- Verneri Välilehto, 267007, verneri.valilehto@tuni.fi

Working methods

Gitlab was used to manage issues in the team and as well as the code repository of the project. We assigned the work roughly by the three main components: frontend for Leevi, server-a for Joni and server-b for Verneri. Some integration implementations as well as dockerization were made together with the whole group.

The team was committed to attend to a weekly meeting and to work about 3 hours per week on the project. We used agile methods when doing the project. In practise that meant that every week during the meeting, we planned what were to be done during that week. The way we planned and achieved progression can be seen in the next chapter.

Weekly progress:

Active work started in 21.3. As the project needs to be ready at 28.4. there was about 6 weeks to complete it.

Week 1: First group meeting, planning, work distribution, studying the project material.

Week 2: Task related study, creating bases for the application components: React app using create-react-app, server-a with the swagger tools and server-b.

Week 3: Frontend: Making a simple view and form for making sandwich orders. Adding services to make api calls.

Backend: Server A: Making some functionalities to create order, getting order by Id and getting every order.

Backend: Server B: Making simple communication to work between server-b and RabbitMQ. Handling resending some data to RabbitMQ.

Week 4: Some integration between the component communications.

Frontend: Improving order form and view, adding state management to react app, adding sandwich form.

Backend: Server A: Fixing different bugs and made data go through server –a from frontend to RabbitMQ and from RabbitMQ to frontend.

Backend: Server B: Fixing some bugs in communication between server-b and RabbitMQ. Implementing handling the state of order.

Week 5: Dockerizing the app, end-to-end testing and fixing faults.

Week 6: Fixing critical faults in the application in all areas. Finishing documentation.

What the group learned

Group learned more about teamworking in a software development. Working as an agile team that has divided tasks everyone to do. Sharing work equally to everyone and making sure everyone is making their part and is not falling down. Communicating about the project and possible problems. In web development our group learned to use Node.js and React and learned more about how the data flows through the app and how to make app to work. General programming skills was learned and they will help in the future. Docker was new to us all, so it is something we all learned. We had to think about documenting our project and we all learned about how to document what we have done and how it could be improved and maybe some issues with the project.

Architecture of the system

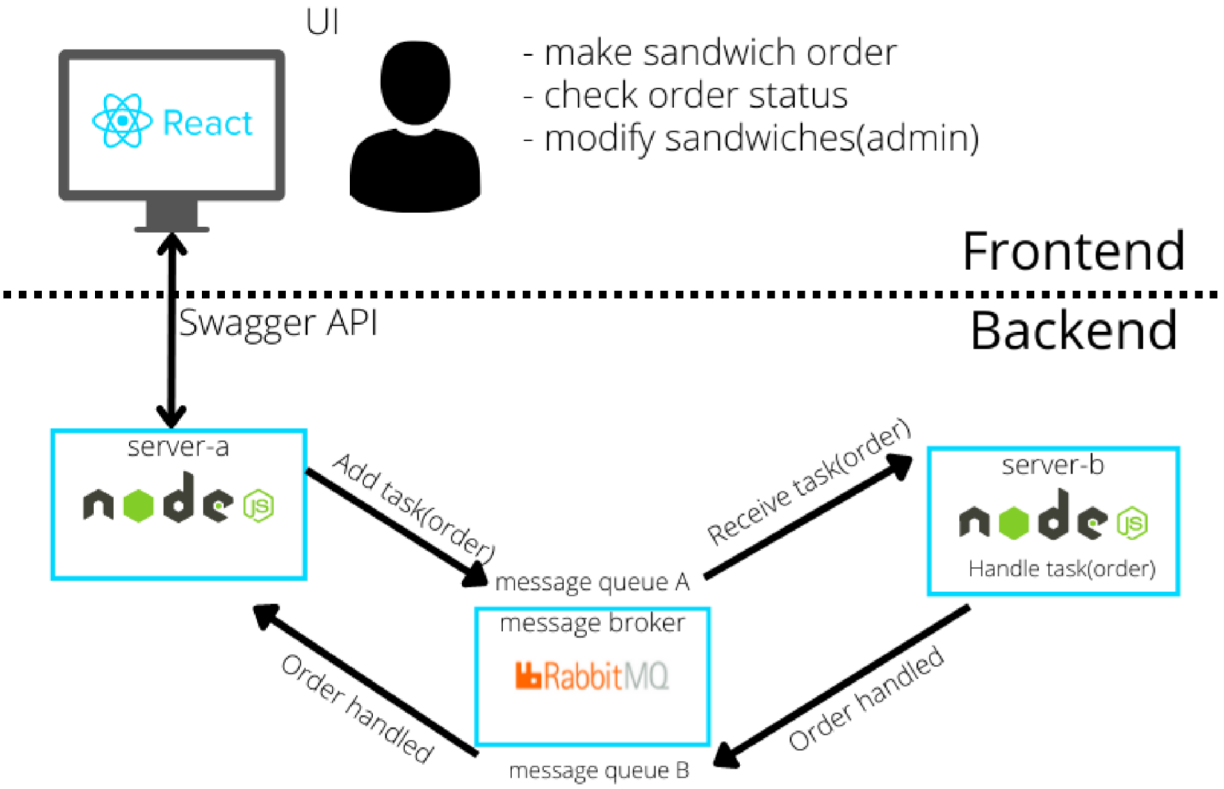

Image 1: High level architecture of the system

Technologies used

Here is listed all the technologies used during this project. These technologies are discussed more detailed in the parts where they have been used in next chapters that describe the architectures and implementations of different individual components.

- JavaScript and some of its libraries
    - Node.js
    - React.js
    - React redux
    - Redux Thunk
    - Axios
- Swagger
- Docker
- RabbitMQ
- AMPQ
- Gitlab
- Git

Docker was used to make the components run in containers to help running the app.

Backend

Backend consists of two servers named simply named server-a and server-b. There is also a message broker between the servers to handle the requests. The message broker is RabbitMQ which is a very popular open-source message broker.

Server A

Server A is made with Node.js. Server A is acquiring data from frontend and delivers it to RabbitMQ. It also receives order data from RabbitMQ.

Server A gets object from frontend and handles it by making another object that contains orderId that is created in Server A. This will be added to the database. Server A uses amqplib for communication with RabbitMQ based message broker. Server A pushes messages to RabbitMQ queue ‘received-orders’.

Server A receives orders that it sent earlier from RabbitMQ queue ‘sent-orders’. After obtaining it Server A will update status to database.

Server B

Server B is made with Node.js. Server B is simple http-server with very limited functionality. Only functionality is to receive and handle messages from queue.

Server B uses amqplib for communication with RabbitMQ based message broker. Server B subscibes to message brokers queue 'received-orders'. Server B receives and handles messages from queue as soon as possible, if server is not already handling another message. After defined delay server will handle message and send acknowledgement about it to RabbitMQ.

Defined RabbitMQ host must be running before Server B is started, so delayed start of Server B is handled in docker-compose.yml with "wait-for-it.sh" script.

Database

A database would be a good addition to the app in the future. However, currently there is not a database implemented for the application due to business of the development team. Database would handle keeping sandwich data, toppings data and possible login data of the users. Currently our app uses array as “database”. However, this array is not saved anywhere and resets when the app resets. In image 1 the database would be connected to server-a if it existed.

Frontend

Frontend is made with React.js framework. The app is a single page web app (SPA) meaning that all necessary resources and JavaScript are downloaded when the user enters to the app using browser. After loading all the necessary JavaScript and other resources, the app works without the need to load any other resources during form submissions, requests to server etc. This makes the user experience of the app better and reduces loading times.

The frontend uses react-redux library to handle the state of the application. Even though the app is currently very small, and the state management would not necessarily require a state management library such as redux, it is used to allow the further expansion of the application and increased number of states to handle.

Frontend uses Axios library which is a “Promise based HTTP client for the browser and node.js”. Axios is used to send the request into server A. Redux-thunk library is used to make the app able to use asynchronous functions.

Login functionality is not yet implemented for the application, but it should be done in future to handle the permissions of different user types (admin, customer). Currently anyone can order sandwiches and create new sandwiches to the system.

Patterns applied

Client-server model is used in this app. App is distributed, and each part of the app is making required requests for correct part of system. This is critical part of making distributed app possible.

Publish-subscribe pattern is used in communication between server a and server b. When server a is sending order to server b, it doesn't send it directly to server b. Instead it sends it to message broker. Server b has subscribed to message brokers correct queue. Server b will receive message from queue instantly, if it it's not already busy. Same is happening with communication from server b to server a.

Messaging pattern is applied in multiple instances in this app. REST api is used to communicate between frontend and backend. RabbitMQ is used to communicate between server a and server b. Using messaging patterns are critical for distributed apps.

How to use/test the system

Instructions

The application currently only works in virtual machine. It can be accessed by using ssh in tie-webarc-37.it.tuni.fi, which is the virtual machine provided to our group. The app should already be running and it is accessed from **tie-webarc-37.it.tuni.fi:3001**.

If it is not running, do following:

1. Run **ssh tie-webarc-37.it.tuni.fi** (requires some permissions probably)
2. In the virtual machine **git clone** the repository from gitlab (https://course-gitlab.tuni.fi/COMP-CS-510-web-architectures_2021-2022/project-sandwich)
3. Run **cd project-sandwich**
4. Run **docker-compose build**
5. Run **docker-compose up**
6. Go to **tie-webarc-37.it.tuni.fi:3001/** in browser

If the docker image does not start in linux terminal, run **chmod +c ./utils/wait-for-it.sh** in both server-a and server-b directories and run **docker-compose up** again.

Now the application should be running and can be used. There is a navbar in the top of the app which is quite easy to use.

As the application currently has not got any database, there aren't any sandwiches ready for the application to order. Therefore, first it is required to make some sandwiches by clicking *manage sandwiches* button from navbar or by going to **http://tie-webarc-37.it.tuni.fi:3001/admin/sandwich_form.**

Sandwich form is quite intuitive way to create sandwiches to the app. However, the toppings select does not work as might except. Toppings select works by selecting some toppings. Every time a new option is chosen, it is registered as a topping. For example: if first ham is selected and then houmous, both are saved as toppings to the sandwich upon submit.

Menu page http://tie-webarc-37.it.tuni.fi:3001/menu shows what the sandwiches contain.

When there are some sandwiches created in the app, an order can be made. Order is made in http://tie-webarc-37.it.tuni.fi:3001/order by choosing a sandwich from select and pressing submit. There is a list in this page that shows some data such as the statuses of orders.

To see the API go to http://tie-webarc-37.it.tuni.fi:3000/docs/ while the app is running.

Known issues

There are some known issues in this app. Unfortunately, some of them were found too late to fix for final submission.

Order page might crash, if page if refreshed manually. Crashing doesn't happen on other pages of the app. To recover from crash, user should go to homepage and navigate to order page from there.

Order page might also crash, if two or more customers are making order at the same time. This does apply on when two or more customers are from different ip adresses. Recovering from crash works same way as is previous scenario.

App works only on specific virtual machine. If app is ran on other virtual machine or on localhost, address of the host must be edited on frontend/src/services/order.js on lines 7, 13 and 21. Address of the host must be edited also on frontend/src/services/sandwiches.js on lines 7, 13 and 21. Better solution would be to use environmental variables that the docker would understand and use these when creating an instance of the app to define the correct ports and URLs.

It is possible to add sandwich with no toppings. It is also possible to add sandwich with same name as some previous sandwich. It is not possible to add sandwich without name. Selecting the toppings is

When order status is changed in backend, frontend does not automatically sense this change and therefore, order list information is not updated in real time. The updating is currently handled by fetching the data from the API every 10 seconds. Some more advanced solution such as WebSocket would make it possible for frontend to listen changes happening in server A.

Data is not saved anywhere and therefore every time the app is restarted it vanishes. This would be fixed by setting up a database. With orders this is not a big issue but with sandwiches it is because the menu needs to be recreated every time when starting the app. Another solution would be to add a list of test sandwiches

There is no authentication in the app yet so any user can do any actions in the app (add orders, add sandwiches) without restrictions.

Deleting sandwiches does not work. Button that would do that in sandwich list is disabled because deleting sandwiches crashed the order list component.