Tampereen yliopisto

# MAKE ME A SANDWICH -DESIGN DOCUMENT

Lihamylly.com

Niilo Rannikko 268085
Jukka-Pekka Keinänen H290650
Lauri Nuutinen H283219

# TABLE OF CONTENTS

# 1. INTRODUCTION

This is the design document for the Make-me-a-sandwich -group project on Tampere University's course COMP.CS.510 Web Development 2 – Architecting. The task for the project was to create a web application for ordering and managing sandwiches through a web page. This document explains the design and architecture choices of the project, gives instructions for setting up and testing the software and explains the working methods, tasks and personal reflection on learning during the project from group members.

# 2. DESIGN AND ARCHITECHTURE

## 2.1 Overview of the system

The idea of the task was to create a web application for ordering different types of sandwiches and viewing their status. The structure of the system follows the instructions of the task and consists of front-end two back-end servers and a message broker between the two servers. The basic idea with the structure is that when a user makes an order for new sandwich in the front-end, it sends a request to server A which then publishes the order to server b. Server B then continuously updates the status of the order to server A and then after a short delay informs about finished sandwich order to server A. All communication between server A and B goes through a dedicated message broker. Server A passes the sandwich status information to front-end for the user to see.  High level view of the system can be seen in the picture bellow.

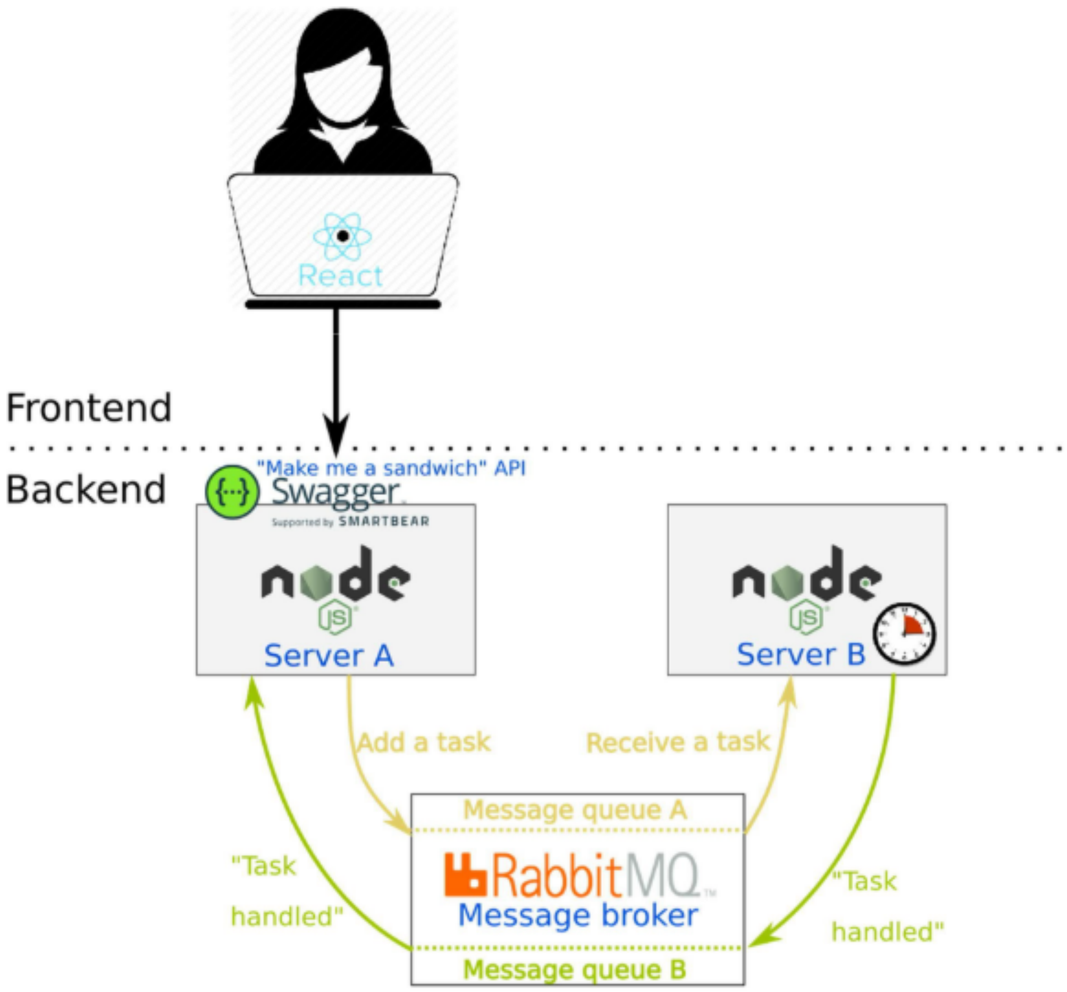

*Figure 1: High level view of the system architecture from the project instructions.*

The task was to implement the Server A API based on a readymade swagger API definition given by the course and then make the other components to work with the server A. Finally, the task was to Dockerize the whole software for easy deployment in any environment.

The figure below presents the application process as flowchart from sending the order from client to saving the completed order to database.

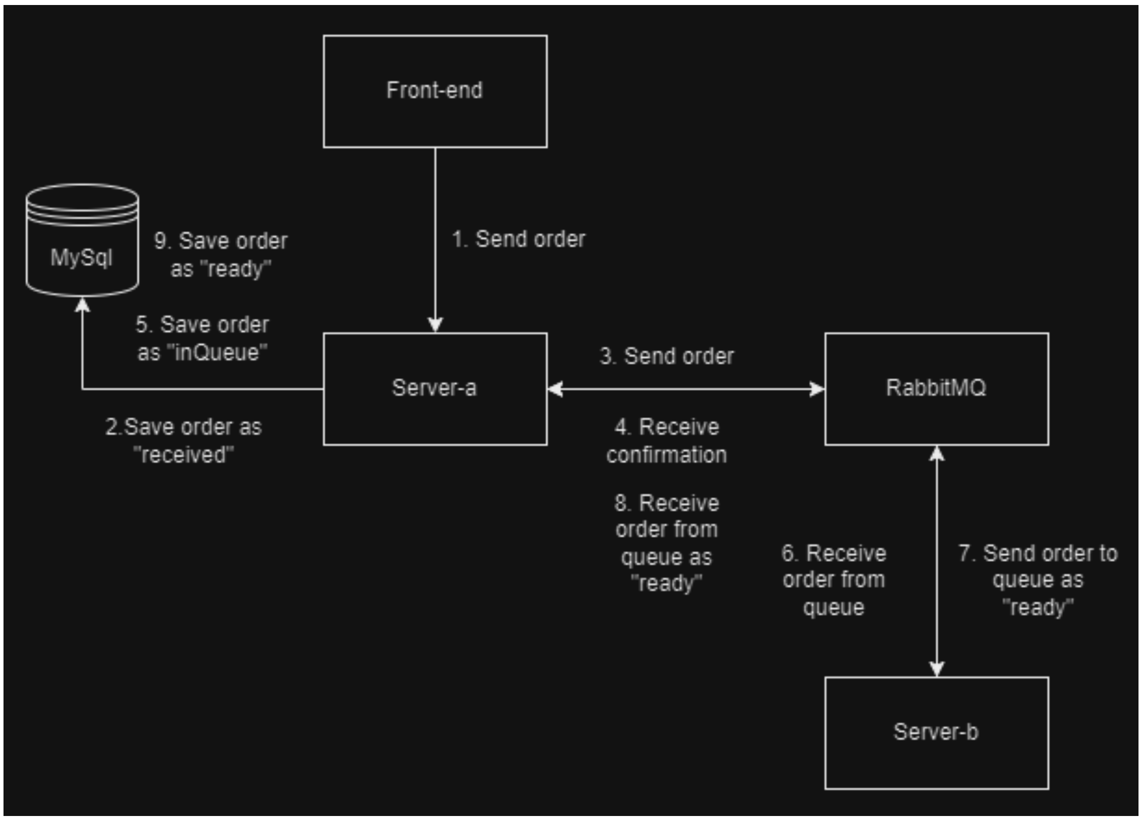
Front-end
1. Send order
MySql
9. Save order as "ready"
5. Save order as "inQueue"
Server-a
3. Send order
RabbitMQ
2.Save order as "received"
4. Receive confirmation
8. Receive order from queue as "ready"
6. Receive order from queue
7. Send order to queue as "ready"
Server-b

***Figure 2: Flowchart of the application.***

## 2.2 Used technologies

The technologies used in this project are:

- React + Vite in front-end
- Node.js for server a & b
- MySQL for the database and Sequalize library for easy use
- RabbitMQ for the message broker
- Docker for containerization

Most of the technologies are replaceable by other similar technologies like Vue instead of React, any other database technology instead of MySQL, Ruby or Python instead of Node.js and so on. The technologies were chosen based on familiarity from other work and worked without issues. All the selected technologies are widely used and popular so getting them to work in this application was straightforward. Working with something else like Vue and some NoSQL database like MongoDB could have been interesting as well but given that already many of the project subjects were new for all team members so we wanted to use familiar stuff where we could.

## 2.3 Front-end

The front-end component of the application is created using React and Vite. Front-end is a simple single page application which consists of a couple of components. The refreshing of the data is handled using polling technique where the front-end sends new requests to the server-a evert 3 seconds. The alternative option would have been using web sockets, but we decided to use polling as it seemed to be easier, and we didn't need web sockets to be necessary. Front-end is also containerized although it was not required but we decided to do it to make the deployment more straightforward. The front-end allows users to make new orders using sandwiches fetched from server-a. In addition to that, users can inspect all created orders and their statuses or search for specific order by id.

## 2.4 Server A

Server-a is responsible for receiving orders from the front-end, handling database operations and sending and receiving messages from message broker. The server-a was generated using swagger tools which generated the needed routes for orders, sandwiched and users. We decided to only implement the required routes and one additional route to fetch all sandwiches.

Server-a uses MySQL database to persist data when the system is running. The system does not persist the data between restarts since this was required. The database contains information about sandwiches, orders and toppings for each sandwich. The database is filled with sandwich information when the application is started. Sequelizer library is used to handle the database operations from server-a as it makes it simpler.

## 2.5 Server B

Server-b receives sandwich orders from server-a and sends them back to server-a with an updated status after seven seconds via the two rabbit message ques. The server-b follows the implementing server-b section to some degree and does not contain many interesting features.

## 2.6 RabbitMQ Message broker

RabbitMQ message broker is used for implementing the ques that handle communications between the servers. Queue-a is for incoming orders going from server-a to server-b and queue-b is for completed orders going from server-b to server-a. Both queues are rather similar, the largest difference being the queue-a having a prefetch limitation making it possible for the queue to only process one sandwich at a time. Otherwise both of the queues are setup to require acknowledgements and to be durable.

# 3. SETTING UP AND USING THE SYSTEM

Here are instructions on how to set up and test the system. The pre-requisites to run the program are Docker and Docker desktop installed and the code from the repository saved locally in the machine. For browser Google Chrome is recommended. The set up can most likely be done without Docker desktop but this guide is written with Docker Desktop in mind. Even though Docker should eliminate problems with using different platforms for the program, it is recommended to use windows as other operating systems had some issues with docker-compose during the project work.

1. Start Docker desktop to activate docker daemon
2. In terminal, navigate to folder where the code from the repository is stored
3. In terminal, make sure you are in the folder where docker-compose.yml-file is located and then run: docker compose up –d
    a. If you get error related to "bash\r", then run command "git config –global core.autocrlf false" and re-clone the repo on start from step 2.
4. You should see all 5 containers start up in terminal output and in docker desktop
    a. This might take a while and server a might restart a couple of times waiting the RabbitMQ message broker and MySQL database to fire up
5. In your browser navigate to http://localhost:3000 to get to the front page of the web application.
6. From here you can start testing the website:
    a. View all orders from "All orders on the upper left corner
    b. Create new orders from "Create order" by selecting a sandwich from the drop menu
    c. Searching orders be id from top right corner

# 4. LEARNING DURING THE PROJECT

The work during this project was done mainly individually with weekly to bi-weekly meetings where progress was checked, and the work divided. In the meetings we also discussed the design of the application, decided on technologies to use, and tried to solve any problems that might have occurred during the project.

Next are learnings and workings of each team member during the project.

## 4.1 Niilo

My tasks in this project were to implement parts of server a based on the swagger reference. Working with the server helped me better understand the usefulness of Swagger and how to set up APIs based on swagger definitions. The functions I implemented did a lot of communication with other parts of the system, so I got to familiarize myself with sending and receiving messages from message broker, communicating with the front-end and understanding the paths of commands and requests in a modular software system. Besides the functions I got to test the API with Postman and set up the dockerfile for the server. During meetings with our team, I closely studied other member's work and especially figuring out how to start the whole system with docker by setting up a docker-compose file was interesting. I feel the project helped me better understand communication between different components of the software and developed a good appreciation towards docker.

## 4.2 Jukka-Pekka

I implemented the front-end and parts of the server-a of the application. I learned more about docker and creating docker-compose and Dockerfiles. I also strengthened my knowledge of React and relational databases and using the relational database from the application point of view. I also learned to use swagger code generation. I also learnt more about message brokers and how to leverage them in the web application.

## 4.3 Lauri

I implemented the rabbit message queues and server B. Diving into the world of message queues was somewhat interesting. Previously I hadn't given much thought to how these things are set up but reading through the tutorials gave me an ok understanding on how things should work. After managing to make the message ques work properly making the server B was pretty effortless as I had already created a similar program for testing the ques. Even though the docker compose file for the project wasn’t my job, I feel like I learned the most from watching Jukka-Pekka work his magic with docker.