COMP.CS.510-2025 Advanced Web Development: Back End

Group Project Report

Tampere Universities

Arto Korpiharju

Jonatan Schmidlechner

23/04/2025

Contents

1. Project plan

1.1 Basic information

Group name: reverse-mensa

GitLab repo URL: https://course-gitlab.tuni.fi/compcs510-spring2025/reverse-mensa

Group members:

- Jonatan Schmidlechner, 150288280, jonatan.schmidlechner@tuni.fi
- Arto Korpiharju, 150356617, arto.korpiharju@tuni.fi

1.2 Timetable and work distribution

Our timetable and work distribution can be found in tables 1 and 2 respectively. Each member's promised working hours can also be found in table 2.

Table 1: Timetable

Project kickoff meeting: Design System architecture. Plan workflow: create gitlab issues, distribute work	26.3.2025
Research required things i.e. web sockets, message brokers and Docker, front end technologies **Implement** simple frontend and server B and containerize them.	26.3.2025 - 2.4.2025
Implement server A and submit mid-project check.	2.4.2025 - 9.4.2025
Create project documentation including UML diagrams for system architecture and analyzing alternative architectures	9.4.2025 - 23.4.2025

Table 2: Work distribution

Group member	Responsibilities	Promised work hours each week
Jonatan Schmidlechner	Servers B and A	8h
Arto Korpiharju	Frontend	6h

2. System architecture

2.1 Overview

The developed system consists of 5 separate containerized entities. Firstly, there is the emote generator, which generates raw emote data and sends them via a message producer to a Kafka message broker, which is the system's second entity. The Kafka message broker serves as a communication bridge between emote generator – Server B, and Server B – Server A. It contains two topics: Raw data topic for the data from the emote generator and aggregated data topic for aggregated data from Server B.

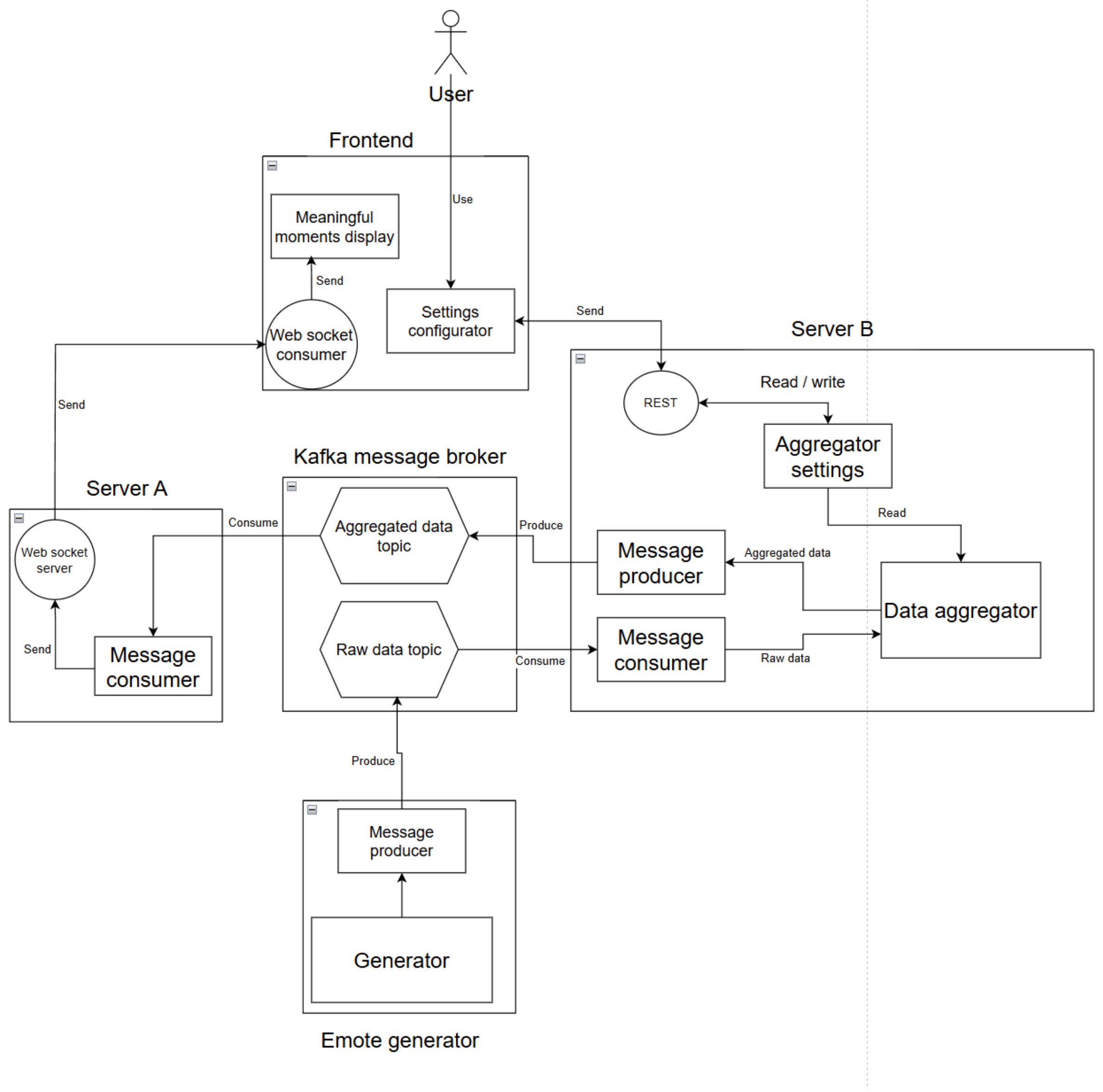

Figure 1: System architecture as an UML diagram

The third separate system entity is Server B. Its purpose is to determine "meaningful moments" from the raw emote data. It consumes the raw emote data via a message consumer and sends it to a data aggregator, which determines the meaningful moments. The aggregated data is sent to Kafka's aggregated data topic via a message producer. In addition, Server B contains aggregator settings, which the data aggregator reads its settings from. Finally, the server contains a REST API that allows reading and updating the aggregator settings. The system's frontend uses this REST API.

The fourth system entity is Server A, whose purpose is to send the meaningful moments to frontend. It consumes the aggregated data from Kafka via a message consumer. Additionally, the server contains a web socket server, which forms a web socket connection with the frontend. The message consumer sends the aggregated data via the web socket connection to the frontend.

Finally, there is the frontend, in which the user interacts with the system. The frontend includes a web socket consumer, which consumes aggregated data from Server A. The data is further sent to meaningful moments display, in which the user visually observes meaningful moments. In addition, the user can configure the settings of the data aggregator in real time via a settings configurator to impact the meaningful moments. The settings editor communicates with the Server B and its aggregator settings via the REST API.

2.2 Evaluation of the architecture

The evaluation of the designed system architecture depends on the system's requirements. A good thing about the architecture is that it splits the system into separate components, each of which has its own purpose. This helps maintain the system. Although one could simultaneously also say the opposite, splitting the system into separate components instead of using a monolith drastically increases the complexity of the system, which increases the initial development time and makes error handling and tracking harder. This makes maintaining it harder.

Whether the architecture is good or bad, it all boils down to what the system needs to achieve. The system is about reacting to live reactions. If the system has a modest number of users, the architecture could be considered bad. With a smaller number of users, the system does not need to be massively scaled, in which case the system doesn't benefit from the more complex system architecture that enables scaling. In this case a monolith architecture including only a frontend and a single backend would have been good enough and significantly simpler to implement and maintain. Whereas, if the system has millions of users, it is necessary to be able to scale some parts of the system to support massive number of users.

Even if the system needs to support millions of users, there are some problems, which could be solved by improving the architecture. The main problem we notice is that Server B is too large and thus hard to scale. Firstly, the aggregator settings should be moved out of the container, because taking multiple instances of server B would also take multiple instances of the settings, which probably is not desired. Assumingly, there should be only one instance of aggregator settings for the whole system. Secondly, with the current architecture you can't individually scale the smaller components in Server B. Maybe only the message consumer needs to be scaled, but not the data aggregator and the REST API. More realistically, the individual components in Server B need to be all scaled, but differently. Maybe 10 instances of message consumers are required, but only 3 instances of data aggregator and 2 instances of REST API are required. To enable this, we should split Server B into even smaller components.

3. Used technologies

The backend was coded using JavaScript and the runtime environment used was Node.js. For inter-service communication, Apache Kafka was utilized as a message broker. Kafka acted as an intermediary transmitting data between the emote generator, Server B, and Server A. Server B and frontend communications were done by REST API. This made it easy for the frontend to request and receive information over HTTP. Server A used WebSocket instead which is very useful for real time data updates. The frontend was built using React which allowed for a dynamic and responsive user interface.

Many of the technologies used have similar alternatives available. For frontend the other suggestion was to use Vue or stick with basic HTML. Both of these would have got the job done similarly, though HTML would have limited the frontend further.

Instead of REST API backend could have used technologies like GraphQL or gRPC. Both of these could have worked about as well as REST without many problems. As for WebSocket technologies like SSE or MQTT would have sufficed.

For communications, Apache for example offers Pulsar which can be employed similarly to Kafka though in this case the features included would have likely been of little use. In addition, Amazon Kinesis, RabbitMQ and Google Pub/Sub have similar roles. Amazon and Google products come with intergrations to their respective services while RabbitMQ would have been well fitting in this small project.

4. Testing instructions

1. To run our system you need docker engine and docker compose. The recommended way is to use docker desktop, which contains the two required components. Install docker desktop from here.
2. Clone the project to your desired destination by pasting the following command to terminal in your desired location:

   ```
   git clone git@course-gitlab.tuni.fi:compcs510-spring2025/reverse-mensa.git
   ```

3. Go into the cloned project by pasting the following command to terminal:

   ```
   cd reverse-mensa
   ```

4. First ensure docker is running by starting docker desktop (if using docker desktop). Paste the following command to terminal:

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

5. Wait for the project to finish building and open your browser (preferably Google chrome or Firefox) and go to http://localhost:8081/.

5. Group work progress

26.3	Jonatan Schmidlechner	We had our kickoff meeting, where we planned the system architecture. Here I saw a simple and understandable example of a service-oriented architecture and what it looks like.
31.3	Jonatan Schmidlechner	Implemented Server B and related stuff. Here I learned how to use Kafka (topic creation, producing, consuming). Additionally, I got more experience in implementing a simple server from scratch (routing, controllers, express, file structure, REST, config). I also learned about docker and docker compose, especially health-check to ensure Kafka is ready before other containers start.
4.4	Arto Korpiharju	Implemented basic frontend for Server B. Connected frontend to server by REST API and made few changes on the backend side for proper connection. I learned about REST connections and access control.
7.4	Jonatan Schmidlechner	Implemented Server A. I learned how to use web sockets. Additionally, I learned how to do volume mounting to sync container and host config files.
14.4	Jonatan Schmidlechner	I wrote the project report about system architecture. Especially the part about evaluating the architecture made me better think about and understand when we need a service-oriented architecture and what it should be like.
20.4	Arto Korpiharju	Implemented basic frontend for Server A significant moments as well as implemented raw data feed from server B to frontend. I learned about WebSocket connections and got to work with REST again.
22.4	Arto Korpiharju	Created a better User Interface. Created a proper UI using react. I learned about react and UI design.

6. Conclusion

Overall, the project was successfully completed in time. Both of us learned a lot about the technologies used as well as little about the alternatives that could have been used. We also had meetings and made sure both understood each other's code, the technologies as well as the architecture.