Project Documentation - Reacting in Real-Time: Emotes and Kafka in Action!

COMP.CS.510 Advanced Web Development - Back End
Lotta Lappalainen: lotta.lappalainen@tuni.fi 150416667
Julius Kari: julius.kari@tuni.fi 150297734
Kasper Kivistö: kasper.kivisto@tuni.fi 50302871

Group name: Bakkarit
Repo URL: https://course-gitlab.tuni.fi/compcs510-spring2025/bakkarit

1. Introduction

The goal of this project is to build a distributed system that enables live stream viewers to react in real-time to meaningful moments during a broadcast. At the same time, it provides broadcasters with tools to monitor, analyze, and customize how these reactions are presented and interpreted.

The system supports real-time emote reactions, which are sent by viewers and processed immediately. When multiple reactions occur in a short timeframe, the system detects and aggregates these bursts to highlight moments of high engagement. Viewers receive live updates through WebSocket connections, ensuring a dynamic and responsive experience. On the broadcaster side, a RESTful API with authentication offers control over stream interaction settings, allowing for deeper customization.

2. Project plan

Group members main responsibilities:

- **Julius**: Server B functionality.
- **Kasper**: Server A functionality, frontend's basic structure and help with Docker.
- **Lotta**: Frontend and docker containerizing.

Promised hours to group project per week:

Working hours are tracked in the GitLab issues. Each group member marks an estimate and used hours for each issue they work on. Branches are used to keep track of new features that are worked on, and most issues are linked to one branch. Merged branches should not be deleted for tracking purposes.

- **Julius**: 1-5 or as much as is needed.
- **Kasper**: 1-5 or as much as is needed.
- **Lotta**: 1-5 or as much as is needed.

Week 1: Planning & Research

Goal: Finalize system requirements, assign tasks, choose technologies.

Tasks:

- Decide on the technologies used.
- Study Kafka.
- Initialize Servers A and B.

Week 2: Backend setup and start frontend

Goal: Set up core backend components, get Kafka working and WebSocket messages to frontend.

Tasks:

- Set up Server A: handles WebSocket connections, receives emotes with Kafka.
- Set up Server B: aggregates and processes emote bursts.
- Frontend: Start a basic frontend.
- Docker: Start work with docker and create the files needed.

Week 3: Basic frontend integration

Goal: Connect Server A and Frontend with WebSocket. Make a basic integration and show emote data from Emote generator in browser.

Tasks:

- Show emotes in browser.
- Get raw emote data to Server B.
- Continue working on Docker.

Week 4: More interactive frontend and working Docker compose

Goal: Start adding more features to frontend. Get Docker compose to work and make sure the whole project and run from one command.

Tasks:

- Interactive and better structured frontend.
- Finish Docker compose work. Make sure the whole project can be run.

Week 5: Final touches, finish documenting and submit

Goal: Finalize the project. Make sure everything works, and no random crashes happen. Get the project documentation done. Submit the project for evaluation.

Tasks:

- Finish project document.
- Finish the code and be sure everything works.
- Submit the project.

3. System Architecture

- **High-Level Diagram:**

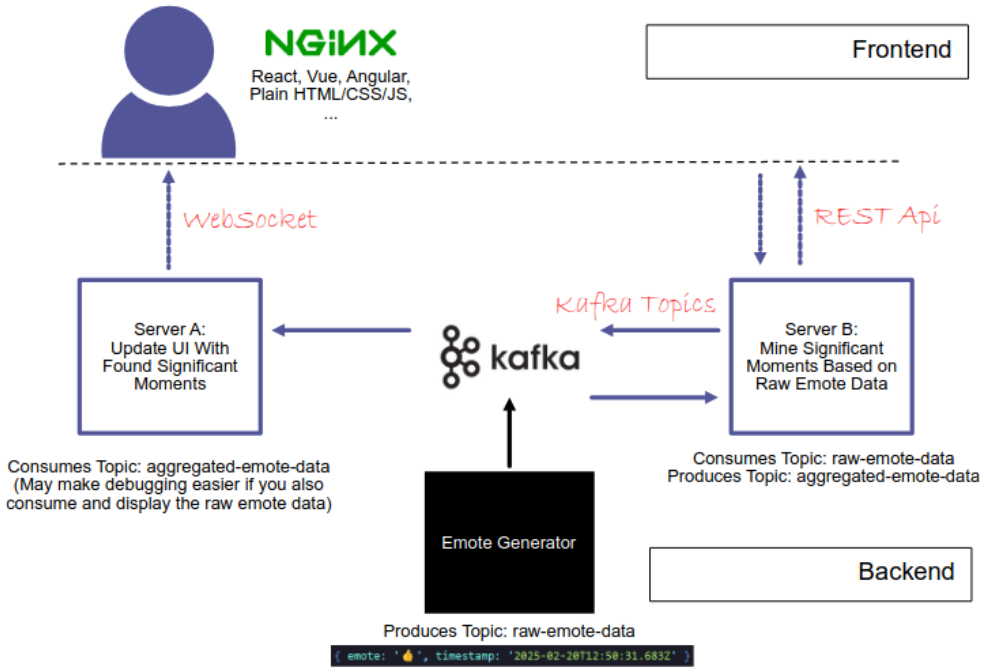

Image 1: High level diagram of the system. [1]

- **Components Overview:**
  - Emote Generator
    - Writes new emote data every second
    - 80% chance for a single emote
    - 20% chance for a burst of multiple (same) emotes
    - Contains an emote and a timestamp as ISO string
  - Kafka:
    - Transmits messages on two topics:
      - The raw emote data from the emote generator
      - The aggregated meaningful moments
  - Server B (Aggregation + REST API)
    - Consumes the raw emote data and aggregates meaningful moments from it
    - Writes those to the meaningful moments' topic
    - Offers a REST API for checking and changing the meaningful moments threshold, the number of messages to consume before analysis as well as checking and changing the collection of allowed emotes
    - Has a login and authentication using JWT for accessing the API

        - Stores the raw emote data, settings and login information in an SQLite3 database using Sequelize
    - Server A (WebSocket Publisher)
        - Consumes the meaningful moments topic and writes it to the frontend using WebSocket connection
    - Frontend Web App
        - Shows the aggregated data from the WebSocket connection
        - Can make asynchronous calls to server B to check and change the settings

4. Component Details

4.1 Emote generator

- **Purpose:** Simulate viewer reactions for testing/initial input.
- **Behavior:**
    - Emits data every second.
    - 80% chance of a single emote.
    - 20% chance of a burst of identical emotes.
- **Output Structure:**

```
{
 "emote": "👏",
 "timestamp": "2025-04-06T10:32:11.123Z"
}
```

4.2 Kafka Topics

- **raw-emotes-topic:**
    - Source of truth for all incoming emote data.
    - Consumed by Server B for aggregation.
- **meaningful-moments-topic:**
    - Holds aggregated bursts of reactions.
    - Consumed by Server A to feed the frontend.

4.3 Server B: Aggregation + Control API

- **Responsibilities:**

    - Subscribes to raw-emotes-topic.
    - Detects and aggregates "meaningful moments" (e.g., high-frequency emote bursts).
    - Publishes aggregated results to meaningful-moments-topic.
- **REST API Endpoints:**
    - POST /login → login and receive a JWT for authentication
    - GET /settings/threshold → current meaningful moment detection threshold.
    - PUT /settings/threshold → update threshold.
    - GET /settings/allowed-emotes → list of allowed emotes.
    - PUT /settings/allowed-emotes → update allowed emotes.
    - GET /settings/interval → get current number of emotes to consume before analysis
    - PUT /settings/interval → update interval
- **Aggregation Logic:**
    - A meaningful moment is when emotes in a short time window (e.g., 2 seconds) exceed a set threshold (e.g., 10).
    - Only count emotes from allowed list.

4.4 Server A: WebSocket Bridge

- **Responsibilities:**
    - Consumes messages from meaningful-moments-topic.
    - Forwards them in real-time to connected clients via WebSocket.
- **Kafka consumer**
    - Topic to consume: “aggregated-emote-data".
    - When debugging use: “raw-emote-data”.
- WebSocket
    - Frontend can connect to server A for real time messages.
    - Address for real time data inside the container: “ws://localhost/ws”.
    - Address for real time data outside the container for debugging: “ws://localhost:8080”.

4.5 Frontend: Viewer Dashboard

- **Components:**
    - LiveChat
        - Live Emote Burst Feed.

    - Settings
        - Threshold Slider / Input.
        - Allowed Emotes Editor.
    - Notification
        - Displays notifications to the user e.g. of successful login
    - Header
        - Renders the navbar conditionally based on if user is logged in
    - Login
        - Form to log in
    - Contact
        - The contact info of our group
- **Services**
    - Login
        - Functions for login, logout and getting the token
    - Settings
        - Functions for getting current settings
        - Functions for changing settings

5. Used technologies

- **Tech Stack Overview:**
    - Backend: Kafka, Node.js, Express, WebSocket server, Sequelize, SQLite3, Cors, Body-parser, JWT, Bcrypt.
    - Frontend: React / Vue / JavaScript.
    - Communication: Kafka (messaging), REST (control), WebSockets (real-time updates).

6. How to run the system

6.1 Setting variables

Create a .env file to the root of the project with these variables, the values are examples and can be changed:

```
TOKEN_SECRET=754eff4e5rw32j7uio89235ewfr9i6oyuef3rw29807i6yfrt
ADMIN_USERNAME=admin
ADMIN_PASSWORD=salasana123
```

To change settings, you need to log in with the ADMIN_USERNAME and ADMIN_PASSWORD variables.

6.2 Docker

To run the system, you need to have **docker** and **docker compose** installed (docker-compose works too). To run the system:

1. Download the repository locally to your machine.
2. Navigate to the root of the just copied repository.
3. Open terminal in the root where the “docker-compose.yml” is located.
4. Copy-paste both of these commands first a then b:
    a. “docker compose build” OR if using older version “docker-compose build”
    b. “docker compose up –d" OR “docker-compose up –d"
5. The docker starts, downloads needed images and builds the code.
6. After Docker is ready, navigate to “localhost” with your browser.

For frontend development running the whole Docker process is not recommended. Instead run the frontend locally and the backend in a container. To do this:

1. Navigate to the backend folder of the repository.
2. Open terminal on run command:
    a. “docker compose –f docker-compose.dev.yml up –d”
3. Then navigate to the frontend folder and open a new terminal and run a command:
    a. “npm run dev”

There are many endpoints and URLs in this project. If for any reason one wants to change these or start developing by oneself, here are some tips:

- Recommended environment layout (Docker/Kubernetes if used).
- Environment variables and settings for:
    - Kafka brokers
        - URL inside containers: kafka:9092
        - URL outside containers: http://localhost:9094
        - TOPIC for raw data: raw-emote-data
        - TOPIC for aggregated data: aggregated-emote-data
    - WebSocket server
        - Connect to: ws://localhost/ws
        - Redirects to: http://server_a:8080
    - REST endpoints to control settings
        - GET /settings/threshold
        - PUT /settings/threshold
        - GET /settings/allowed-emotes
        - PUT /settings/allowed-emotes
        - GET /settings/interval

- PUT /settings/interval

6.3 How to test the REST API

To test the API with docker-compose running you can call the endpoints with the following commands

- POST /login

You call this endpoint with the credentials that you put in the .env file in the project root directory.

Example request:

```
curl -X POST http://localhost:3001/login -H 'Content-Type: application/json' -d '{"username": "admin", "password": "salasana123"}'
```

Example response:

```
{"token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImFkbWluIiwiaWF0IjoxNzQ1ODQ4MDU2LCJleHAiOjE3NDU4NTE2NTZ9.cSHZJrwv7GIIwDROpumueVKYRgw9Z8f8x6N_55fAS_0"}
```

- GET /settings/threshold

Example request (replace <Token> with the one you received from the login endpoint):

```
curl -X GET http://localhost:3001/settings/threshold -H 'Authorization: Bearer <Token>'
```

Example response:

```
{"threshold":0.5}
```

- PUT /settings/threshold

Example request (replace <Token> with the one you received from the login endpoint):

```
curl -X PUT http://localhost:3001/settings/threshold -H "Content-Type: application/json" -H 'Authorization: Bearer <Token>' -d '{"threshold": 0.8}'
```

Example response

```
{“threshold”:0.8}
```

- GET /settings/allowed-emotes

Example request (replace <Token> with the one you received from the login endpoint):

```
curl -X GET http://localhost:3001/settings/allowed-emotes -H 'Authorization: Bearer <Token>'
```

Example response

```
[{"emote":"💖"},{"emote":"👍"},{"emote":"😢"},{"emote":"👺"}]
```

- PUT /settings/allowed-emotes

Example request (replace <Token> with the one you received from the login endpoint):

```
curl -X PUT http://localhost:3001/settings/allowed-emotes -H "Content-Type: application/json"  -H 'Authorization: Bearer <Token>' -d '[{"emote": "😢", "isAllowed": false}]'
```

Example response

```
[{"emote":"💖","isAllowed":true},{"emote":"👍","isAllowed":true},{"emote":"😢","isAllowed":false},{"emote":"👺","isAllowed":true}]
```

- GET /settings/interval

Example request (replace <Token> with the one you received from the login endpoint):

```
curl -X GET http://localhost:3001/settings/interval -H 'Authorization: Bearer <Token>'
```

Example response

```
{"interval": 100}
```

- PUT /settings/interval

Example request (replace <Token> with the one you received from the login endpoint):

```
curl -X PUT http://localhost:3001/settings/interval -H "Content-Type: application/json" -H 'Authorization: Bearer <Token>' -d '{"interval": 150}'
```

Example response

```
{“interval”: 150}
```

If there is some error with any of the requests, the API will return an error, like the following example for a server error, along with an error code (500 in this example).

```
{"error":"Internal server error"}
```

7. Learning diary

Related branches or commits are commented to the issues to give more context.

- Week 1:
  - Kasper Kivistö: Initializing server A and studying Kafka
    - GitLab Issue #3: Week1: Initialize server A and study Kafka
  - Kasper Kivistö: Implementing basic Kafka integration
    - GitLab Issue #4: Week1: Server A - Basic Kafka integration
- Week 2:
  - Kasper Kivistö: How WebSockets work and do basic integration
    - GitLab Issue #11: Week2: Server A - Basic WebSocket integration
  - Kasper Kivistö: Kafka and WS work locally, make it work in Docker.
    - GitLab Issue #14: Week 2: Server A - Get the code ready for docker
  - Kasper Kivistö: Start the documentation. Make a frame for it.
    - GitLab Issue #13: Week 2: Documentation start
  - Lotta Lappalainen: Create initial docker files for all servers
    - GitLab Issue #8: Week 2: Create initial Dockerfile for frontend
    - GitLab Issue #5: Week 2: Create initial Dockerfiles for both servers
  - Lotta Lappalainen: Configure nginx (initial)
    - GitLab Issue #7: Week 2: Configure nginx
  - Lotta Lappalainen: Create initial docker-compose file
    - GitLab Issue #6: Week 2: Create initial docker-compose file
  - Lotta Lappalainen: Start frontend
    - GitLab Issue #15: Week 2: Create some initial frontend
  - Lotta Lappalainen: Start documentation
    - GitLab Issue #17: Week 2: Documentation start
  - Julius Kari: Initializing server B:
    - GitLab Issue #10: Week 2: Server B - initialize the server
  - Julius Kari: How to set up Sequelize with SQLite3
    - GitLab Issue #12: Week 2: Server B - create a database
  - Julius Kari: How to set up REST API and validate an emote
    - GitLab Issue #16: Week 2: Server B - create a REST API
- Week 3:
  - Kasper Kivistö: Clean server A of unused code.

        - GitLab Issue #21: Week 3: Clean Server A code and remove not needed functions
    - Julius Kari: How to consume the raw emote data through kafka.
        - GitLab Issue #9: Week 3: Server B - create raw emote data consumer
- Week 4:
    - Kasper Kivistö: Make frontend more interactive. Create basic frame for the frontend.
        - GitLab Issue #20: Week 4: Make frontend interactive
    - Lotta Lappalainen: Clean up docker compose file
        - GitLab Issue #18: Week 4: Make sure docker compose works
- Week 5:
    - Kasper Kivistö: Finish Docker compose.
        - GitLab Issue #24: Week 5: Fix Docker compose
    - Lotta Lappalainen: Finish frontend
        - GitLab Issue #29: Week 5: Finish frontend
    - Julius Kari: Create aggregation of emotes.
        - GitLab Issue #23: Week 5: Server B - create raw emote data aggregation
    - Julius Kari: Send meaningful moments through kafka.
        - GitLab Issue #27: Week 5: Server B - send aggregated meaningful moments via Kafka
    - Julius Kari: Add authentication with JWTs.
        - GitLab Issue #28: Week 5: Server B - create a login
    - All: Update the documentation.
    - All: Final meeting and project submit.

8. For the future

The project is in a working state and should have all the necessary parts in it. But as the project was quite short and this course mainly focused on the backend side of things, we can see that improvements could be made. Some of the things that could have been added are:

- Unit and integration tests for:
  - Emote generation.
  - Aggregation logic.
  - REST API.
- Frontend testing:
  - WebSocket connection simulation.
  - UI interaction with backend.
- More added functionality.
  - Register functionality and user management
  - Monitoring and statistics for commentators and IT personnel.
  - Even more visually pleasing frontend.

9. Appendix

- Glossary:
  - *Emote*: A viewer's reaction symbol.
  - *Burst*: A rapid series of identical emotes.
  - *Meaningful Moment*: A burst that passes a reaction threshold.
- Libraries used:
  - Server A:
    - Kafkajs, https://www.npmjs.com/package/kafkajs
    - Ws, https://www.npmjs.com/package/ws
    - Cors, https://www.npmjs.com/package/cors
  - Server B:
    - Kafkajs, https://www.npmjs.com/package/kafkajs
    - Dotenv, https://www.npmjs.com/package/dotenv
    - Express, https://www.npmjs.com/package/express
    - Sequelize, https://www.npmjs.com/package/sequelize
    - SQLite3, https://www.npmjs.com/package/sqlite3
    - Body-parser, https://www.npmjs.com/package/body-parser
    - Cors, https://www.npmjs.com/package/cors
    - Jsonwebtoken, https://www.npmjs.com/package/jsonwebtoken
    - Bcrypt, https://www.npmjs.com/package/bcrypt
  - Frontend:
    - Axios, https://www.npmjs.com/package/axios
    - React, https://www.npmjs.com/package/react
    - React-dom, https://www.npmjs.com/package/react-dom
    - React-router-dom, https://www.npmjs.com/package/react-router-dom