

COMP.CS.510 Advanced Web Development – Back End
Group project assignment document
Spring 2025

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



Table of contents

1	Introduction	1
2	Documentation.....	1
2.1	Design and architecture	1
2.2	Project management.....	1
3	Architecture overview	2
4	Backend.....	3
4.1	Dockerize your backend and frontend	3
4.2	Server B	4
4.3	Server A.....	4
4.4	Message broker	4
4.5	Keep containers independent of host	5
5	Frontend	5
5.1	Required functionality	6
5.2	Implementation	6
6	Alternative technologies and architecture extensions.....	7
6.1	Too difficult?.....	7
7	Tools	7
8	Stages of work and submissions	8
8.1	Common to all submissions	8
8.2	Mid-project check-in.....	8
8.3	Final submission	8
9	Grading	9
10	Did you find errors in this document?	10

1 Introduction

In this group work project, you implement a system with a frontend and a distributed backend. The system enables the viewers of a live stream to participate by reacting to meaningful moments and the broadcasters to analyze the viewers reactions.

It is most essential to understand the architecture of your system. Hopefully, the project gives you an understanding on how the architecture is a consequence of not only design but also the components selected for implementation.

This is an advanced-level course. There will be less holding hands. We give you more liberties in, for example, choosing the technologies you prefer, but at the same time, we expect, that you can take ownership of your choices, can self-study and fix possible issues, and can defend your choices in the documentation.

Motivation and backstory

This system is designed to allow viewers of streamed events to participate and react to events. Perhaps during a sporting event or a song contest the viewers might want to react to something using simple emotes. Perhaps a referee disqualifies a goal, and you strongly disagree and react with an angry face emote and if a lot of other viewers react similarly, the screen is covered with angry faces for a moment. Maybe this can increase viewer immersion, and it can provide the broadcaster with automatic annotations of meaningful moments.

2 Documentation

Documentation is very important in this project.

2.1 Design and architecture

Along with your code, you must return documentation. This must describe *at least*:

- Architecture
 - the patterns you have applied
 - what components there are and what is their purpose
 - which technologies you use
 - if you have selected technologies other than suggested, make it clear
- How to try the system
- What are the user credentials required to access the system, if such are required

2.2 Project management

Git enables you to document what you plan to do, who has finally done what and when. Therefore, use Gitlab issues to assign tasks to group members. Once done, update the status of each issue.

3 Architecture overview

Figure 1 shows a high-level view of the system, including a web frontend and backend. The frontend provides a user interface, whereas the backend serves user requests asynchronously. The backend contains servers that communicate messages via Kafka.

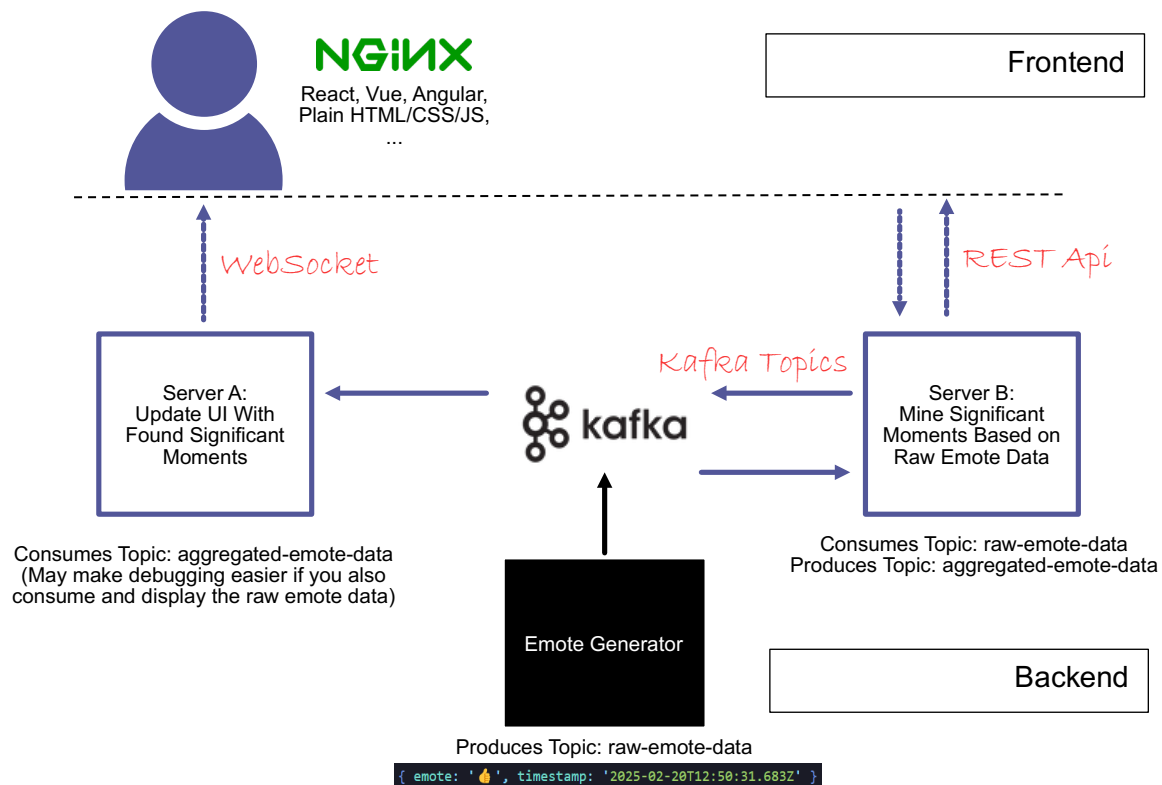


Figure 1. A high-level view of the system architecture

The backend contains several components.

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:

- 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 as well as checking and changing the collection of allowed emotes

Server A:

- Consumes the meaningful moments topic and writes it to the frontend using WebSocket connection

Frontend is a React/Vue/JS app that:

- Shows the aggregated data from the WebSocket connection
- Can make asynchronous calls to server B to check and change the settings (needs some UI)

4 Backend

4.1 Dockerize your backend and frontend

You are expected to implement the backend as Docker containers. Furthermore, you should use Docker Compose to facilitate configuration. Emote generator, Server B, Server A, Frontend, and Kafka should all reside in their own containers and there should be a shared virtual network between them. Only the frontend port should be exposed to the host machine, the rest of the components should not be accessible directly from the host machine.

When containerizing the frontend, one common solution is to start from an Nginx base image and then use the **COPY** command to both move the frontend files to that container and to copy any necessary configuration details to Nginx (e.g., `COPY nginx.conf /etc/nginx/conf.d/default.conf`). Notice, that the frontend container should not contain, for example, a Node.js backend – frontend is just the frontend.

4.2 Server B

Server B subscribes to the raw-emote-data -topic, parses the incoming messages and stores them in an array. Once Server B has collected a suitable number of messages, it will analyze them. How you choose to implement this analysis is not super important, but one way would be to first slice the timestamp to minute-level granularity, then keep count of individual emotes in that one-minute interval as well as the total amount of emotes in that interval (`emoteCounts[timestamp][emote]++` and `emoteCounts[timestamp].total++`) and then check that if the count of a particular emote divided by the number of all emotes is over some threshold, then we have a significant moment.

Once Server B has collected and analyzed that suitable number of messages, it will then send all found significant moments to aggregated-emote-data -topic.

Server B needs to offer a REST API for checking and changing of the settings. Implement the following actions:

- `/settings/interval` – read and update the number of messages to consume before analysis
- `/settings/threshold` – read and update the threshold after which a moment is considered “significant” (i.e., $\text{counts[emote]} / \text{totalEmotes} > \text{threshold}$? “Significant” : “Not significant”).
- `/settings/allowed-emotes` – read and update the list of allowed emotes. This does not need to be able to introduce more emotes, but could be used to, for example, temporarily remove some emotes from analysis.

Remember to make the API in a RESTful manner (e.g., methods, status codes).

4.3 Server A

Server A should at least consume the aggregated-emote-data -topic and write that to frontend using WebSocket communication. It could make the UI more interesting and useful, if you also show the raw emote data, but this is up to you.

4.3.1 Implementation of Server A and Server B

The recommended technology is NodeJs. Implement the server as a Docker image. It is recommended that this image is integrated into a Docker Compose file. For advice on dockerization of a Nodejs app, see <https://github.com/nodejs/docker-node/blob/master/docs/BestPractices.md>.

It is recommended to use the `kafkajs` package with NodeJs <https://kafka.js.org/> - `KafkaJS` · `KafkaJS`, a modern Apache Kafka client for Node.js

4.4 Message broker

You are expected to use Kafka for message brokering. If you choose the recommended path, there's a getting started tutorial on `kafkajs` page: <https://kafka.js.org/docs/getting-started> - `Getting Started` · `KafkaJS`

4.4.1 Required functionality

You need two message topics to deliver messages about emote reactions:

- *Raw-emote-data*: delivers random emote reactions from emote generator to Server B (can also deliver to server A)
- *Aggregated-emote-data*: deliver aggregated data (found “significant” moments) from Server B to Server A

The servers create these topics when they start to send messages.

4.4.2 Implementation

You don't need to create a Docker file for Kafka, because such images already exist at Docker Hub. You can include any of these images as such in a Docker Compose file. Bitnami/kafka is recommended, but the important thing is that Kraft should be used instead of Zookeeper.

See <https://hub.docker.com/r/bitnami/kafka>

4.5 Keep containers independent of host

When you create a Docker image, make sure all software resides within the image/container! This includes all source code, packages, and scripts.

That is, never store any software in a volume located within the Docker host! This has at least the following disadvantages:

- Reduced performance
- Dependency on host (file permissions, etc.)

A Docker container is supposed to be an independent entity. This principle breaks if you supply software from the host. Not following this principle likely causes problems in deployment and grading, which will reduce your points!

Also, create a network in your docker compose file and make sure all containers operate inside this network. You don't need to know, for example, what IP addresses each container gets, as docker handles that for you – you can simply refer to the services by their names. As for ports, those should be defined in the docker-compose.yml -file and injected into the containers as environment variables.

5 Frontend

The frontend is less strictly defined than the backend. You can choose some frontend framework, such as React, Vue, Angular, etc., or you can stick with more “vanilla” HTML/CSS/JS. Notice, that the frontend is just a frontend and should not contain any backend components, such as a NodeJs. It is recommended to just have Nginx serving the frontend files.

5.1 Required functionality

Frontend app should enable the users to:

- See significant moments (even if just a writing the JSON data to page)
- Interact with the settings REST API on Server B.

During usage, the entire webpage MUST NOT refresh. That means that all messaging happens asynchronously in the background. Server A should communicate via WebSockets and the communication with the REST API on server B could use normal Fetch API: [Fetch API - Web APIs | MDN](#)

This is the minimum implementation, but it's quite boring. You are encouraged to make it more interesting. This could include, for example:

- Also showing the raw emote data
- Playing an animation of a burst of emote floating up the screen whenever a significant moment is detected
- Generally making a domain-appropriate and interesting UI

5.2 Implementation

React


The aptly named "Create React App" is a good starting point:

- Web page: <https://create-react-app.dev/>
- Getting started: <https://create-react-app.dev/docs/getting-started/>
- Github repository: <https://github.com/facebook/create-react-app>

Deploy your Create React App with Docker and Nginx: <https://medium.com/yld-blog/deploy-yourcreate-react-app-with-docker-and-nginx-653e94ffb537>

Vue

Vue also offers "create-vue" as a good starting point:

- Quick start guide: [Quick Start | Vue.js](#)
- Github repository: [GitHub - vuejs/create-vue](#):  [The recommended way to start a Vite-powered Vue project](#)
- Multi-stage containerization: [How to Serve a Vue App with nginx in Docker | TypeOfNaN](#)

HTML/CSS/JS

This is the simplest option, as you don't need a build stage. And to complete the minimum

requirements, there's hardly any real benefit to choosing a frontend framework.

[Deploying a Static HTML Site with Docker and Nginx | by Zulfikar Muhamad | Medium](#)

6 Alternative technologies and architecture extensions

This document describes the **recommended** technologies to use and the **minimum required complexity** of the architecture. With some caveats, you can use your own choice of technologies and extend the architecture to include other components. If your group decides that you want to use other technologies than specified or extend the architecture, send email to the course personnel. You must specify and justify your choices. The course personnel reserves the right to decline any technology and architecture if this seems unsuitable.

It is important that you add the specification and justification of your choices also in the documentation.

The course personnel offers absolutely **no support** for alternative technologies or architectures.

6.1 Too difficult?

If the architecture seems too challenging, but you still want to get something working done, here's some ways you might be able to lower the difficulty. If you choose to implement an easier assignment, make sure to mention it in the documentation. Choosing an easier assignment will reflect negatively on your grading but is still better than not getting anything done.

- Kafka is a bit heavy for this simple architecture and is more geared towards massive real-time data streams. A simpler message broker, such as RabbitMQ (AMQP) could be easier to set up.
- Having the front end containerized separately might cause some friction. Having Server A also serve the frontend could simplify the architecture a bit.
- Remove emote generator and emote analysis and have Server B generate random "significant moments" directly. *The rest of the application needs to be really good in order to pass if you choose to simplify the architecture this much!*

7 Tools

Each group will get a GitLab repository. You must place all documentation and code into this repo.

For development, you need:

- a text editor or IDE suitable for editing code, like Atom or VS Code
- git: <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>
- NodeJs: <https://nodejs.org/en/download/>
- Docker: <https://docs.docker.com/install/>
- Docker Compose: <https://docs.docker.com/compose/install/>

These are pretty standard tools-of-the-trade nowadays and essential in many projects. Note, that if you install Docker Desktop, you will also have docker compose.

8 Stages of work and submissions

8.1 Common to all submissions

The course personnel will look at:

- Source code
 - Ensure this is easy to review
- Documentation (up to date)
 - Place single PDF (!) in root (!) of your repo
 - (See also the section about documentation in this document)
- Gitlab issue board (up to date)

Ensure what you submit is the **latest commit of master branch!** Therefore, after mid-project check-in, create another branch to enable working on the project.

8.2 Mid-project check-in

The purpose of the mid-project check-in is to see that the projects are progressing at an acceptable speed. Hopefully at this stage, you are well on your way with the backend with much of it implemented and documented.

Open an issue in your GitLab repo with the title “Mid-project check-in ready for review”. Assign the issue to the user account of ~~each~~ course assistant. **Only one assistant can be selected, so choose randomly.** In the description field of the issue, you can write if you have problems with something in the project. Then, the course personnel will try offer help.

Keep on working on the project after you have opened the issue. The course personnel will only check that the project is progressing. If any grave mistakes are found, this will be commented on the issue in GitLab.

8.3 Final submission

Once your repo is ready, open an issue in the GitLab repo with the title “Course project ready for review.” Assign the issue to the GitLab user account of ~~each~~ course assistant (it might not support assigning to multiple, in which case just pick one).

Once the issue is open, you must stop working on the project, because the course project is now done! HURRAY FOR YOUR MAGNIFICENT WORK! :-)

The course personnel grades the project and closes the issue.

9 Grading

The grading principles (see Table 2) are intentionally broad, as the scope of the project is so wide that an all-covering grading scale would be counterproductive. Thus, the course personnel reserves the right to decide case-by-case on grading.

Documentation. As learning about web architectures is the focus and not only coding, you will receive points from a documentation that shows you have thought about the architecture and studied the tools you used.

Deadlines. A failure to meet the deadlines will result in a points reduction. If you notice that you can't meet a deadline, inform the course personnel ASAP. You could receive an extension if there's a valid reason, but this applies only if done before the deadline.

Good job. If your frontend or backend is simply good and cool, you *might* receive extra points. These extra points cannot however compensate for more than some minor flaws.

Table 2. Grading principles

Points	Project quality overview	Description
12	Excellent work all-around	All the technical and project work documentation is as required and is of high quality. Both backend and frontend have been implemented almost flawlessly, possibly with optional functionality, additional technologies and/or with exceptional quality to make up for any shortcomings.
9	The work that has been done with some documented faults	All the technical and project work documentation is as required. Backend has been implemented, and some effort has been given to the frontend, but otherwise the project contains considerable flaw(s).
6	About half of the requirements have been filled. What remains missing has been documented.	Technical and project work documentation is done for the parts that have been completed. Backend has been implemented at least partially, but frontend implementation is not.
3	Good effort has been given to the project.	Technical and project work documentation is as well as it can be for an incomplete project. There are signs of trying to implement the backend.
0	No work has been done, nothing works, nothing has been written.	The course cannot be passed without an accepted group project.

10 Did you find errors in this document?

If you feel that something in the document is unclear, something necessary is left out, or something is erroneous, please email the course personnel to report your findings.

Good luck!