# IUM-TWEB REPORT

## Table of Contents

# 1- INTRODUCTION

We decided to do every one of our projects, and so every server, in a distinct project with a distinct git every time. The principle is to separate the servers to limit conflicts and unnecessary dependencies in project code. This also makes it easier to call the projects when grouping them through the Main Server.

Before starting, we took the time to define the specifications that our project needed to meet, ensuring that we were aligned with the objectives and development. This essential step lays the foundation for the project and saves a considerable amount of time during the development phase.

We started with the Spring Boot server connected to PostgreSQL, responsible for managing static data, which allowed us to work directly with this data. In parallel, we developed the Main server, integrating chat management using Socket.IO, while testing API requests to the Spring Boot server. Next, we implemented the MongoDB server to store chat history, considered dynamic data. Finally, we completed the development with the front-end.

We will then go into more detail about what was done and the challenges encountered on our three different servers, as well as how the notebooks were created.

# 2- DIVISION OF WORK

Regarding the division of work, Mathis is responsible for setting up the various servers and continuously improving them, as he is more comfortable with development. Meanwhile, Paul assists with these servers based on the foundation established by Mathis. This allows Paul to understand how the servers work and how requests are made between them, thereby improving his own understanding of web development.
If we consider the workload distribution, it corresponds to a 65-35% split, with 65% for Mathis and 35% for Paul.

# 3- SPRINGBOOT SERVER

## a. Introduction

We decided to start with this server because it provides us with a solid and functional foundation for the later implementation of our Main Server. By doing so, we can easily verify that the requests made in our JavaScript code are correct by simply displaying them at the localhost:3000 address in HTML, as we will see later.

For this server, we decided to build on the JPA for database access lab (lecture 24). We then created an administrator user named *postgres* with the password *postgres*.

## b. Building our SpringBoot server

We began by importing all the data from CSV files into our PostgreSQL database. Once the tables were created, we used JPA to connect our project to PostgreSQL. Next, we developed the necessary Spring Boot features for the REST APIs, including the Controller, Service, and Repository layers, to properly prepare for future calls to this server.

We started by using the code for the Character class provided in the starter code as a foundation for creating the various classes required for our project. We adapted this code to meet our specific needs, then initially tested the requests directly from this server to ensure they were functioning correctly and accessible from another server.

In our initial tests, we could enter a movie name into the search bar, and, in a raw and simple manner, it would return all movies containing those letters in the title. We then added additional elements to the dropdown menu below, such as searching by actor name. Here's what it looks like starting from the Spring Boot server:

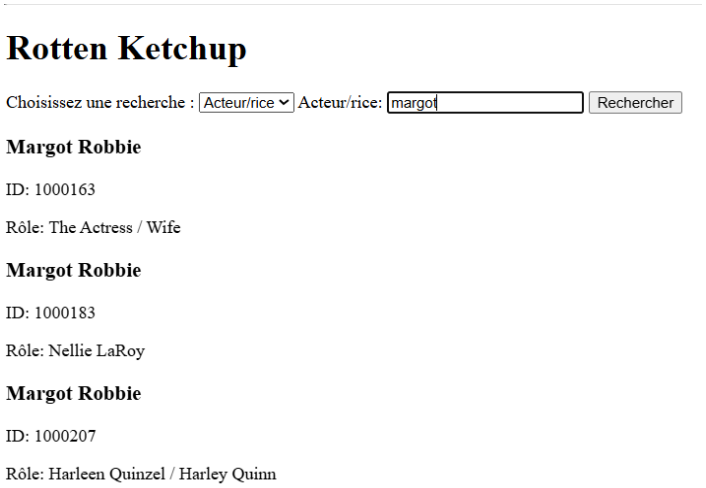


*Figure 1: First tries on the SpringBoot server*

After this preliminary work, we continued creating various queries in the repositories of our classes and adding more to facilitate the construction of our Main Server.

### c. Challenge and Issues – Conclusion

We encountered difficulties right from the start with importing the provided .csv files into PgAdmin 4. Since the columns were separated by commas, this caused issues during data import. Additionally, the data types of certain columns were incorrect, further complicating the process. Not knowing the correct parameters to handle these specifics in PgAdmin 4, we lost a significant amount of time before we even began coding.

On the other hand, building this server was conceptually straightforward but also time-consuming. This was primarily due to the need to understand the overall architecture of the project and the interaction between the different files. Once these connections were clarified, the use of JPA and SpringBoot became repetitive and straightforward.

## 4- MAIN SERVER

### a. Introduction

The Main Server is where we make requests to other servers through APIs. For this, we started with the Socket.io lab (lecture 18) to create an initial version of our chat, which was then incrementally implemented and improved.

## b. Building our Main Server

As mentioned earlier, we started with a functional chat using Socket.io, but the main challenge was adapting it to our specific needs. Since the MongoDB server had not yet been implemented, we initially designed the architecture of the Main Server. We structured the project to clearly separate the views, using a general layout that included components like the menu and footer, while dynamically displaying page content.

Additionally, the internal routes within the server, as well as those pointing to Spring Boot or MongoDB, were organized into separate files for better readability and maintainability.

Once the MongoDB server was ready, I modified the chat system on the main server. The chat is still managed by Socket.IO for real-time communication but is now also handled by the MongoDB server, which manages chat history based on different rooms.

Regarding the connection with the other two servers (Express and Spring Boot), this is easily managed within the route files. We can directly call the functions and requests already implemented in the two other servers:

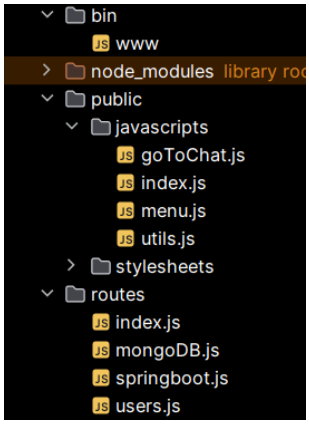


*Figure 2: Our Main Server's routes*

In the beginning, we started by simply calling the *FindByKeyword* function to create a basic search bar on our website.

Since this worked, we moved on to setting up proper architecture for the Main Server by creating layouts, pages, and components. To achieve this, we ensured that calls to the SpringBoot Server were grouped into separate route files, as were other routes necessary for the smooth functioning of our project.

Next, we continued calling the various requests already created on the Spring Boot server and gradually integrated them into the Main Server. As a first step, we displayed movies based on genre, followed by filtering by date, type, and language selected by the user.

Finally, we kept improving the page by refining its layouts progressively.

## c. Challenge and Issues – Conclusion

Regarding this server, we faced difficulties with calls to the Spring Boot server. Since the connection to this server was the first one established, the initial functions calling the Spring Boot server's controller functions did not yield satisfactory results on our front-end page. The data retrieved did not match the expected data. Consequently, we continually modified and improved our page to better meet the specifications outlined in our Word document.

# 5- MongoDB Server

## a. Introduction

For this server, we also started with a course lab (Lecture 20). This server is where we store chat messages. We determined that chat messages were the most suitable data to store in MongoDB due to their dynamic nature, unlike the static movie information. However, we also wanted to add movie ratings to MongoDB, enabling users to rate movies. Unfortunately, we could not implement this feature due to time constraints.

## b. Building our MongoDB Server

For MongoDB, we implemented two main functions: one for storing a message sent by a user in the database and another for retrieving all messages linked to a specific movie. To accomplish this, we structured the Express project into several layers: a controller to manage operations, a model to define the data structure in MongoDB, and a database module to handle the connection to the database.

Once the chat was functional, we created a notebook to filter relevant columns from the reviews.csv file from Rotten Tomatoes and associate them with the corresponding movie IDs. This allowed us to link messages in MongoDB to a specific room, with each room being associated with a movie. We then added this file to MongoDB to include all messages/reviews for each movie.

## c. Challenge and Issues

The main challenge was understanding how MongoDB calls worked, particularly the utility of Mongoose schemas. However, once we grasped this, building the server became relatively straightforward.

That said, we were disappointed that we didn't have enough time to add more features to MongoDB, such as movie ratings or a link to view proposed movies that redirect users to a streaming site for that movie.

# 6- NOTEBOOK SECTIONS (Mathis Pipart ONLY)

## a. Introduction

For the IUM (Interactive User Manual) section, I created two complementary notebooks to effectively analyze the data. The first notebook accesses the PostgreSQL database, filters the necessary data, and saves the table information into dataframes, which are then stored in a .pkl file (not included in the Git repository due to its size). The second notebook is dedicated to visualizing the data contained in the .pkl file and serves as the main workspace for in-depth data analysis.

## b. Notebook Implementation

The first notebook, while not extensive, is essential for the creation of the second. It remains simple, focusing solely on connecting to PostgreSQL and saving the dataframes in a `.pkl` file. In contrast, the second notebook is the primary focus of the project. It is here that I cleaned the data (although, in hindsight, it would have been more logical to do this in the first notebook). I also structured my analysis logically, building a coherent progression to add meaning to my work.

For visualization, I used a wide range of tools, including Plotly, Seaborn, and GeoPandas, to diversify the representations while ensuring they remained relevant and engaging. My goal was to combine varied and visually appealing visualizations with a clear and logical analytical approach.

### c. Challenge and Issues

The main challenge of this notebook, which was also its primary goal, was handling a large volume of data. While this might seem obvious, it led to several complications, particularly with dataframe merge operations, which were time-consuming. I often found myself waiting long periods for code to compile, only to discover it didn't work as intended.

Additionally, I struggled to define a clear guiding thread to structure the data analysis. Another significant challenge was balancing my time between the TWEB and IUM sections. I tended to devote less time to the IUM section, as I found working on the notebook less enjoyable, even though the subject itself remained very interesting.

## 7- CONCLUSION

In conclusion, we successfully completed the project on time while meeting the requirements of the specifications. All the main data was incorporated into the two different databases, considering their static and dynamic nature, and sorting this information appropriately. Our three servers are well-connected and communicate effectively. The requests from the main server to the other servers work as intended.

This project has been highly beneficial for our future personal projects. It taught us rigor and allowed us to consolidate our knowledge of JavaScript. We also learned new methods for building a website by connecting it to multiple distinct databases with different purposes.

## 8- EXTRA INFORMATION

All the information needed to run our project is found in the README.md file in the Main Directory on our GitHub.

## 9- BIBLIOGRAPHY

The laboratories used as starting points for this project are those covered during the Web Services course and made available for independent study, as indicated by Professor Fabio Ciravegna, in the Web Service course (*INF0002*).