UNIVERSITÀ DEGLI STUDI DI TORINO

Human-Computer Interaction and Web Technologies

Samuele Tommasi, Stefano Miceli

2024/2025

Contents

1 Introduction

This report documents the design, implementation, and integration of the web service architecture for the MovieDB platform. The system includes three servers—**main-server**, **postgre-server**, and **mongo-server**—structured to provide an efficient user experience. Each server follows a microservices architecture to ensure modularity, scalability, and maintainability.

The **main-server**, built with Express.js, serves as the central access point, providing both a dynamic frontend rendered with Handlebars.js and a REST API to bridge client requests with the underlying data services. The **postgre-server**, developed in Spring Boot with PostgreSQL, handles static data such as movie metadata and Oscar awards, while the **mongo-server**, developed in Express with MongoDB, manages dynamic content such as user reviews and feedback.

As required, the platform design leverages a clear separation of tasks, with each server focused on specific responsibilities, improving performance and reducing complexity. This architecture not only facilitates the efficient handling of diverse datasets but also enables easy scaling and future feature expansion.

2 Main Server

2.1 Solution

The **main-server** is the central access point of the MovieDB platform, offering two distinct services: a dynamic web application and a REST API. These services can be used independently, with the web app itself that consumes the API, maintaining a separation of concerns. This design ensures that the API can be isolated and deployed as a separate service without requiring modifications to the server.

The web app allows users to explore data about movies, Oscars, and reviews. Each of the three sections—**Movies**, **Oscars**, and **Reviews**—is designed as a standalone module due to the lack of linkage between datasets. Users can perform searches and apply a wide range of filters through an intuitive interface built with direct JavaScript DOM manipulation. Additionally, a real-time chat feature is available, allowing users to discuss the current section by activating a chat icon.

2.2 Issues

Designing a user-friendly and functional interface was one of the main challenges. Filter functionality required careful implementation to ensure that it was intuitive and visually appealing while remaining efficient. Working directly with the website's structure using only plain JavaScript for DOM manipulation, without the support of modern frameworks (e.g. React), made this task particularly time-consuming.

2.3 Requirements

The main-server completely meets the project requirements. It provides a robust REST API and a dynamic front-end that fully integrates with the back-end.

2.4 Limitations

The absence of a modern front-end framework posed a significant limitation. Implementing advanced features such as the filtering system required substantial effort, as everything had to be extensively tested and debugged. Additionally, the lack of linkage between the datasets restricted the interconnectivity of the platform's sections.

3 Postgre Server

3.1 Solution

The **postgre-server** is designed as a modular and scalable microservice using domain-driven development principles. By employing JPA Specifications and Pageable within utility classes, we created a dynamic and maintainable system for constructing complex queries.

This server is intentionally decoupled from any specific client implementation, offering generic enterprise APIs suitable for various client applications. This approach guarantees reusability, even if some of its capabilities are not fully utilized by the MovieDB web app. The database was left in a non-normalized state based on guidance from the lecturer, as normalization was not the primary focus of this project.

3.2 Issues

One of the main challenges was the steep learning curve of JPA. While the framework is powerful, it required significant effort to understand its features, such as Specifications for building complex queries dynamically. We learned to avoid writing explicit SQL queries in the repository, only to discover certain SQL functions like `DISTINCT` and `GROUP BY` were not natively supported within JPA Specifications. These limitations led to additional workarounds and adjustments in our implementation.

3.3 Requirements

The **postgre-server** fully complies with the project requirements by providing a flexible API for handling static data. This includes advanced filtering, sorting, and pagination capabilities, meeting all the expected specifications.

3.4 Limitations

The decision to keep the database non-normalized introduced some challenges in implementing certain functionalities, as data relationships were not structured optimally. However, since normalization was not a priority for this project, these issues were addressed pragmatically, balancing functionality and efficiency without striving for perfection.

4 Mongo Server

4.1 Solution

The **mongo-server** is the smallest component of the MovieDB platform, designed to manage and serve the reviews dataset. Despite its simplicity, it also uses domain-driven development principles to set up a strong foundation for future expansion.

To maintain consistency with the **postgre-server**, we implemented a counterpart of the filtering classes and utilities, ensuring a standardized approach across both servers. This design highlights the strict methodology applied throughout the project, even in smaller components.

Additionally, we considered implementing a user feedback feature that would allow users to provide a score (+1 or -1) to build a relationship table

between movies, Oscars, and reviews. While this feature was not implemented due to time constraints and the project's scope, the server design accommodates such expansions in the future.

4.2 Issues

The simplicity of the **mongo-server** meant we did not encounter any major technical issues. The server was straightforward to implement and benefited from the lessons learned during the development of the **postgre-server**.

4.3 Requirements

The server meets all the project requirements, providing flexible APIs for managing and querying the reviews dataset. Its design aligns with the overall architecture of the platform and uses the best practices for modularity and scalability.

4.4 Limitations

Given its simplicity, the **mongo-server** did not present any significant limitations. However, its functionality is currently limited to managing the reviews dataset.

5 Conclusions

The development of the MovieDB platform introduces us into designing and implementing a microservice-based architecture for a web application. By dividing the system into three distinct servers we achieved modularity, scalability, and maintainability of the system, while ensuring each component served a clear and specific purpose.

The **main-server** demonstrated the importance of balancing user experience with technical considerations, as we focused on creating an intuitive and functional web interface. The **postgre-server** was the first instance of working with JPA Specifications to build dynamic and reusable APIs. Meanwhile, the **mongo-server**, though simpler, allowed us to work with non relational databases and microservice architecture.

Looking ahead, we recognize opportunities to further enhance the platform, such as implementing the user feedback feature to bridge the gap between datasets. This would add significant value to the system by enabling a more interconnected and insightful exploration of movies, Oscars, and reviews.

6 Division of work

The workload for this project was distributed evenly among all group members to ensure a balanced and collaborative effort. Each team member contributed to every major aspect of the project, including the design, implementation and documentation.

This approach allowed everyone to gain experience in multiple areas of the project while ensuring that no single member was overburdened.

7 Extra Informations

Full instructions for running the webservice are provided in the *ReadME* file of the GitHub repository.