# *IUM-TWEB 12 CFU – Movies Library Project Report*

## *Task 1 – Front-end Implementation (mainServer)*

**Solutions Adopted:**

- **Dropdown search system:** We chose an input with a dropdown menu, allowing the user to easily select whether to search by title, genre, or actor. This improved usability and avoided ambiguous input.
- **Responsive grid poster view:** The movie posters are displayed in a responsive grid with pagination (12 per page) to optimize user experience, especially on mobile devices, by reducing scrolling and improving loading performance.
- **Complete movie detail page:** A single screen aggregates all relevant movie data. Lazy loading of reviews and carousels prevent long and heavy pages.
- **Real-time chat without storage:** A lightweight chat system for temporary discussions was implemented without persistence, keeping the server simple and fast.

**Issues and Limitations:**

- **Heterogeneous data handling:** Backend responses often included redundant or missing information. We had to filter and restructure data for proper display, especially in dynamic lists.
- **Handlebars and dynamic rendering:** We encountered difficulties using `{{#each}}` with complex objects and in synchronizing JS event handlers with dynamically loaded DOM elements.
- **Pagination and navigation:** Implementing navigation without directional arrows caused usability issues. Managing navigation buttons required careful logic to prevent errors.

## *Task 2 – MongoDB Server Implementation*

**Solutions Adopted:**

- **Modular architecture:** Each resource (languages, posters, reviews, etc.) has its own controller and router, making maintenance and scalability easier.
- **Use of Mongoose schemas:** Standard for data validation and efficient querying, ensuring data consistency and code readability.
- **Clear endpoints and asynchronous calls:** REST endpoints implemented with `async/await` and `Promise.all` to perform multiple parallel queries, optimizing performance.
- **Pagination via query parameters:** Used for reviews to prevent overload on the client side.
- **Error handling and consistent responses:** Always returning `200 OK` with empty arrays instead of `204 No Content` simplified frontend parsing.

**Issues and Limitations:**

- **Missing or incomplete data:** Some films lacked data (e.g., posters, reviews), requiring robust handling to prevent crashes or poor UX.
- **Pagination with `skip()` and `limit()`:** Offset issues and navigation logic bugs were fixed by refining page calculations and disabling navigation buttons appropriately.
- **Coordinating multiple queries:** Ensuring synchronization between different async queries without blocking the response was challenging.

## *Task 3 – Spring Boot Server Implementation*

**Solutions Adopted:**

- **Layered architecture (Repository-Service-Controller):**
  - **Repository:** Defines CRUD and custom queries using Spring Data JPA.
  - **Service:** Encapsulates business logic, improving testability and reusability.
  - **Controller:** Handles HTTP requests and responses, keeping code clean and separated.
- **Use of JPA/Hibernate:** Enables entity mapping to PostgreSQL tables without writing raw SQL, speeding up development and reducing bugs.
- **Built-in pagination with `Pageable`:** Essential for handling large datasets and improving scalability and responsiveness.
- **Custom queries based on naming conventions:** Used to fetch related data simply and clearly.

**Issues and Limitations:**

- **Query optimization:** Initially difficult to design efficient queries without using complex SQL.
- **Pagination and filter coordination:** Combining search filters with pagination required attention to avoid empty or misleading results.
- **Minimal error handling:** Needed to return consistent responses even in case of missing data.

## *Task 4 – Data Cleaning and DB Configuration and Data exploration*

**Solutions Adopted:**

- **Data distribution:** We chose to split the data into two databases, based on data type:
  - **MongoDB** for frequently changing, loosely related data.
  - **PostgreSQL** for static relational data, closely tied to movies.
- **Data cleaning:** Raw CSVs were explored with Pandas, converted into strongly typed DataFrames, and refined using filtering and normalization policies. (e.g. Reviews were mapped to a standard decimal scale with a custom method; Oscar nominations and reviews were matched via movie name and release date.)
- **ER modeling:** To optimize the queries, and data storages, we introduced the movie id fk also into extra datasets (reviews, oscar) then we removed all the redundant attributes related to movies.
- **Data Stories:** The stories we developed addressed the following themes:

  1. Analysis of gender equality;
  2. Analysis of the distribution of various themes relative to the movie duration;
  3. Analysis of how the average movie duration has changed over the years;
  4. Comparison between the number of movies, oscar, and ratings.

**Issues and Limitations:**

- **Data normalization:** Modeling functions that converted raw data into clean usable entries, while maintaining coverage was difficult. Some data had to be removed due to missing references, reducing overall completeness.
- **Data Stories:** Limited use of charts (10 plots).

*Summary Table – Strengths and Limitations by Task*

| Project Area | Strengths | Limitations / Challenges |
|---|---|---|
| **Frontend (Task 1)** | - Clean, user-friendly interface with search by title, genre, or actor | - Lacks advanced filters (e.g., sort by rating, date) |
| | - Responsive grid view with pagination | - Pagination lacks directional arrows; base buttons only |
| | - Complete detail view with lazy-loaded reviews | - Error messages are minimal and uninformative |
| | - Lightweight real-time chat (no persistence) | |
| **MongoDB Backend** | - Modular architecture (one controller/router per resource) | - No advanced filters or caching |
| **(Task 2)** | - Mongoose schemas for data validation and structure | - No authentication or security layers |
| | - Pagination for large datasets (reviews) | - Initial issues with skip/limit and offset handling |
| | - Consistent responses (empty arrays instead of 204) | - Difficulties in handling incomplete data and synchronizing multiple async queries |
| **Spring Boot Backend** | - Clean layered architecture (Repo-Service-Controller) | - Minimal error handling |
| **(Task 3)** | - Simplified queries with Spring Data JPA | - No caching or security mechanisms |
| | - Custom queries for efficient data access | |
| | - Well-integrated relational DB (PostgreSQL) | |
| **General** | - Modular, scalable architecture across services | - No user authentication or overall security |
| | - Efficient async handling (Promise.all) on MongoDB side | - No caching or complex filtering for optimization |
| | - Basic error management and data coherence | |

## *Project Conclusions and Requirements*

The entire project focused on:

- **Modularity and Maintainability:** Each backend and frontend component was designed independently to allow future updates and easier debugging.
- **User Experience:** The frontend prioritizes clarity and usability, with flexible search options and paginated navigation.
- **Performance and Scalability:** Backend supports pagination, efficient queries, and uses modern technologies (MongoDB, Spring Boot, JPA) for handling large datasets.
- **Missing Data Handling:** Pragmatic solutions were adopted to avoid breaking the app due to incomplete or missing data.

We met all the requirements, aiming to apply the knowledge we acquired in the best possible way to tackle the main challenges of the project, such as synchronizing data, between the frontend and the two backends, (MongoDB and PostgreSQL), handling asynchronous operations, and keeping the system lightweight and user-friendly without adding unnecessary complexity. Features like authentication, advanced filters, and caching have been postponed for future development. Throughout the entire project, ChatGPT proved to be an invaluable tool for debugging, writing custom CSS, refactoring code, and assisting with documentation. While every single step was fully understood by the team, the support provided by ChatGPT significantly accelerated the design and development process. Its contribution helped us maintain a clean codebase, solve issues more efficiently, and focus on the broader goals of the project.

## *Work Division*

| Name | Tasks and Contributions |
|---|---|
| Sandri Gabriele | Equal distribution of work across all tasks |
| Sandri Mattia | Equal distribution of work across all tasks |

Both team members contributed equally to the project and decisions throughout the project.
As twin brothers, we worked side by side during the entire development process, sharing design decisions, debugging sessions, and implementation efforts.
The code was committed to GitHub to ensure balanced participation and traceability.

## *Extra Informations*

No extra informations, to run open readme.md of every project.

*Schema RestApi:*